

Gradle User Manual

Version 8.13

Version 8.13

Table of Contents

OVERVIEW	6
Gradle User Manual	6
RELEASES	8
Installing Gradle	8
Compatibility Matrix	13
The Feature Lifecycle	16
UPGRADING	19
Upgrading your build from Gradle 8.x to the latest	19
RUNNING GRADLE BUILDS	83
CORE CONCEPTS	84
Gradle Basics	84
Gradle Wrapper Basics	86
Command-Line Interface Basics	89
Settings File Basics	91
Build File Basics	92
Dependency Management Basics	95
Task Basics	97
Plugin Basics	100
Gradle Incremental Builds and Build Caching	103
Build Scans	106
AUTHORING GRADLE BUILDS	109
CORE CONCEPTS	110
Gradle Directories	110
Multi-Project Build Basics	112
Build Lifecycle	120
Writing Settings Files	125
Writing Build Scripts	132
Using Tasks	152
Writing Tasks	165
Using Plugins	170
Writing Plugins	195
GRADLE TYPES	204
Understanding Properties and Providers	204
Understanding Collections	213
Understanding Services and Service Injection	219
STRUCTURING BUILDS	240
Structuring Projects with Gradle	240
Declaring Dependencies between Subprojects	246

Sharing Build Logic between Subprojects	250
Composite Builds	260
Configuration On Demand	269
DEVELOPING TASKS	272
Understanding Tasks	272
Controlling Task Execution	290
Organizing Tasks	306
Configuring Tasks Lazily	314
Developing Parallel Tasks	341
Advanced Tasks	356
Using Shared Build Services	372
DEVELOPING PLUGINS	381
Understanding Plugins	381
Understanding Implementation Options for Plugins	392
Implementing Pre-compiled Script Plugins	393
Implementing Binary Plugins	400
Testing Gradle plugins	430
Publishing Plugins to the Gradle Plugin Portal	443
OTHER TOPICS	457
Working With Files	457
Initialization Scripts	510
Dataflow Actions	518
Testing Build Logic with TestKit	521
Using Ant from Gradle	533
OPTIMIZING BUILD PERFORMANCE	548
Configuring the Build Environment	548
Gradle-managed Directories	558
Logging	565
Improve the Performance of Gradle Builds	575
Configuration cache	596
Continuous Builds	638
Inspecting Gradle Builds	639
Isolated Projects	644
File System Watching	648
THE BUILD CACHE	653
Build Cache	653
Use cases for the build cache	666
Build cache performance	669
Important concepts	673
Caching Java projects	678
Caching Android projects	683

Debugging and diagnosing cache misses	686
Solving common problems	694
DEPENDENCY MANAGEMENT	704
CORE CONCEPTS	705
1. Declaring dependencies	705
2. Dependency Configurations	711
3. Declaring repositories	714
4. Centralizing dependencies	719
5. Dependency Constraints and Conflict Resolution	723
6. Dependency Resolution	724
7. Variant Aware Dependency Resolution	732
DECLARING DEPENDENCIES	737
Declaring Dependencies Basics	737
Viewing Dependencies	747
Declaring Versions and Ranges	753
Declaring Dependency Constraints	765
Declaring Dependency Configurations	769
DECLARING REPOSITORIES	781
Declaring Repositories Basics	781
Centralizing Repository Declarations	786
Repository Types	788
Metadata Formats	796
Supported Protocols	800
Filtering Repository Content	812
CENTRALIZING DEPENDENCIES	817
Platforms	817
Version Catalogs	820
Using Catalogs with Platforms	835
MANAGING DEPENDENCIES	839
Locking Versions	839
Using Resolution Rules	849
Modifying Dependency Metadata	874
Dependency Caching	897
UNDERSTANDING DEPENDENCY RESOLUTION	904
Understanding the Dependency Resolution Model	904
Understanding Variant Selection	909
Capabilities	925
Variants and Attributes	929
CONTROLLING DEPENDENCY RESOLUTION	946
Dependency Resolution	946
Dependency Graph Resolution	947

Artifact Resolution	952
Artifact Views	958
Artifact Transforms	965
PUBLISHING LIBRARIES	990
Publishing a project as module	990
Understanding Gradle Module Metadata	994
Signing artifacts	999
Customizing publishing	1000
The Maven Publish Plugin	1011
The Ivy Publish Plugin	1028
OTHER TOPICS	1039
Verifying dependencies	1039
Aligning dependency versions	1063
Modeling library features	1070
PLATFORMS	1082
JVM BUILDS	1083
Building Java & JVM projects	1083
Testing in Java & JVM projects	1108
Managing Dependencies of JVM Projects	1141
JAVA TOOLCHAINS	1146
Toolchains for JVM projects	1146
Toolchain Resolver Plugins	1162
JVM PLUGINS	1165
The Java Library Plugin	1165
The Application Plugin	1177
The Java Platform Plugin	1184
The Groovy Plugin	1190
The Scala Plugin	1199
INTEGRATION	1212
Gradle & Third-party Tools	1212
Tooling API	1213
Test Reporting API	1215
REFERENCE	1219
Gradle Wrapper Reference	1219
Gradle Daemon	1229
Command-Line Interface Reference	1240
GRADLE DSL/API	1260
A Groovy Build Script Primer	1260
Gradle Kotlin DSL Primer	1265
CORE PLUGINS	1294
Gradle Plugin Reference	1294

HOW TO GUIDES	1297
How to share outputs between projects.....	1297
LICENSE INFORMATION	1304
License Information	1304

OVERVIEW

Gradle User Manual

Gradle Build Tool



Gradle Build Tool is a fast, dependable, and adaptable open-source [build automation](#) tool with an elegant and extensible declarative build language.

In this User Manual, Gradle Build Tool is abbreviated **Gradle**.

Why Gradle?

Gradle is a widely used and mature tool with an active community and a strong developer ecosystem.

- Gradle is the most popular build system for the JVM and is the default system for Android and Kotlin Multi-Platform projects. It has a rich community plugin ecosystem.
- Gradle can automate a wide range of software build scenarios using either its built-in functionality, third-party plugins, or custom build logic.
- Gradle provides a high-level, declarative, and expressive build language that makes it easy to read and write build logic.
- Gradle is fast, scalable, and can build projects of any size and complexity.
- Gradle produces dependable results while benefiting from optimizations such as incremental builds, build caching, and parallel execution.

Gradle, Inc. provides a free service called [Build Scan®](#) that provides extensive information and insights about your builds. You can view scans to identify problems or share them for debugging help.

Supported Languages and Frameworks

Gradle supports Android, Java, Kotlin Multiplatform, Groovy, Scala, Javascript, and C/C++.



Compatible IDEs

All major IDEs support Gradle, including Android Studio, IntelliJ IDEA, Visual Studio Code, Eclipse,

and NetBeans.



You can also invoke Gradle via its [command-line interface](#) (CLI) in your terminal or through your continuous integration (CI) server.

Releases

Information on Gradle releases and how to install Gradle is found on the [Installation page](#).

User Manual

The Gradle User Manual is the official documentation for the Gradle Build Tool:

- [Running Gradle Builds](#) — Learn how to use Gradle with your project.
- [Authoring Gradle Builds](#) — Learn how to develop tasks and plugins to customize your build.
- [Working with Dependencies](#) — Learn how to add dependencies to your build.
- [Authoring JVM Builds](#) — Learn how to use Gradle with your Java project.
- [Optimizing Builds](#) — Learn how to use caches and other tools to optimize your build.

Education

- [Training Courses](#) — Head over to the [courses page](#) to sign up for free Gradle training.

Support

- [Forum](#) — The fastest way to get help is through the [Gradle Forum](#).
- [Slack](#) — Community members and core contributors answer questions directly on our [Slack Channel](#).

Licenses

Gradle Build Tool source code is open and licensed under the [Apache License 2.0](#). Gradle user manual and DSL reference manual are licensed under [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#).

Copyright

Copyright © 2024 Gradle, Inc. All rights reserved. Gradle is a trademark of Gradle, Inc.

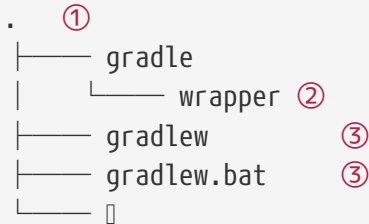
For inquiries related to commercial use or licensing, contact Gradle Inc. directly.

RELEASES

Installing Gradle

Gradle Installation

If all you want to do is run an existing Gradle project, then you don't need to install Gradle if the build uses the [Gradle Wrapper](#). This is identifiable by the presence of the `gradlew` or `gradlew.bat` files in the root of the project:



- ① Project root directory.
- ② [Gradle Wrapper](#).
- ③ Scripts for executing Gradle builds.

If the `gradlew` or `gradlew.bat` files are already present in your project, **you do not need to install Gradle**. But you need to make sure your system [satisfies Gradle's prerequisites](#).

You can follow the steps in the [Upgrading Gradle section](#) if you want to update the Gradle version for your project. Please use the [Gradle Wrapper](#) to upgrade Gradle.

Android Studio comes with a working installation of Gradle, so you **don't need to install Gradle separately when only working within that IDE**.

If you do not meet the criteria above and decide to install Gradle on your machine, first check if Gradle is already installed by running `gradle -v` in your terminal. If the command does not return anything, then Gradle is not installed, and you can follow the instructions below.

You can install Gradle Build Tool on Linux, macOS, or Windows. The installation can be done manually or using a package manager like [SDKMAN!](#) or [Homebrew](#).

You can find all Gradle releases and their checksums on the [releases page](#).

Prerequisites

Gradle runs on all major operating systems. It requires [Java Development Kit](#) (JDK) version 8 or higher to run. You can check the [compatibility matrix](#) for more information.

To check, run `java -version`:

```
❯ java -version
```

```
openjdk version "11.0.18" 2023-01-17
OpenJDK Runtime Environment Homebrew (build 11.0.18+0)
OpenJDK 64-Bit Server VM Homebrew (build 11.0.18+0, mixed mode)
```

Gradle uses the JDK it finds in your path, the JDK used by your IDE, or the JDK specified by your project. In this example, the \$PATH points to JDK17:

```
❏ echo $PATH
/opt/homebrew/opt/openjdk@17/bin
```

You can also set the `JAVA_HOME` environment variable to point to a specific JDK installation directory. This is especially useful when multiple JDKs are installed:

```
❏ echo %JAVA_HOME%
C:\Program Files\Java\jdk1.7.0_80
```

```
❏ echo $JAVA_HOME
/Library/Java/JavaVirtualMachines/jdk-16.jdk/Contents/Home
```

Gradle supports [Kotlin](#) and [Groovy](#) as the main build languages. Gradle ships with its own Kotlin and Groovy libraries, therefore they do not need to be installed. Existing installations are ignored by Gradle.

[See the full compatibility notes for Java, Groovy, Kotlin, and Android.](#)

Linux installation

▼ *Installing with a package manager*

SDKMAN! is a tool for managing parallel versions of multiple Software Development Kits on most Unix-like systems (macOS, Linux, Cygwin, Solaris and FreeBSD). Gradle is deployed and maintained by SDKMAN!:

```
❏ sdk install gradle
```

Other package managers are available, but the version of Gradle distributed by them is not controlled by Gradle, Inc. Linux package managers may distribute a modified version of Gradle that is incompatible or incomplete when compared to the official version.

▼ *Installing manually*

Step 1 - [Download](#) the latest Gradle distribution

The distribution ZIP file comes in two flavors:

- Binary-only (bin)

- Complete (all) with docs and sources

We recommend downloading the bin file; it is a smaller file that is quick to download (and the latest documentation is available online).

Step 2 - Unpack the distribution

Unzip the distribution zip file in the directory of your choosing, e.g.:

```
❯ mkdir /opt/gradle
❯ unzip -d /opt/gradle gradle-8.13-bin.zip
❯ ls /opt/gradle/gradle-8.13
LICENSE NOTICE bin README init.d lib media
```

Step 3 - Configure your system environment

To install Gradle, the path to the unpacked files needs to be in your Path. Configure your **PATH** environment variable to include the **bin** directory of the unzipped distribution, e.g.:

```
❯ export PATH=$PATH:/opt/gradle/gradle-8.13/bin
```

Alternatively, you could also add the environment variable **GRADLE_HOME** and point this to the unzipped distribution. Instead of adding a specific version of Gradle to your **PATH**, you can add **\$GRADLE_HOME/bin** to your **PATH**. When upgrading to a different version of Gradle, simply change the **GRADLE_HOME** environment variable.

```
export GRADLE_HOME=/opt/gradle/gradle-8.13
export PATH=${GRADLE_HOME}/bin:${PATH}
```

macOS installation

▼ *Installing with a package manager*

SDKMAN! is a tool for managing parallel versions of multiple Software Development Kits on most Unix-like systems (macOS, Linux, Cygwin, Solaris and FreeBSD). Gradle is deployed and maintained by SDKMAN!:

```
❯ sdk install gradle
```

Using **Homebrew**:

```
❯ brew install gradle
```

Using **MacPorts**:

```
❏ sudo port install gradle
```

Other package managers are available, but the version of Gradle distributed by them is not controlled by Gradle, Inc.

▼ *Installing manually*

Step 1 - **Download** the latest Gradle distribution

The distribution ZIP file comes in two flavors:

- Binary-only (bin)
- Complete (all) with docs and sources

We recommend downloading the bin file; it is a smaller file that is quick to download (and the latest documentation is available online).

Step 2 - **Unpack the distribution**

Unzip the distribution zip file in the directory of your choosing, e.g.:

```
❏ mkdir /usr/local/gradle
❏ unzip gradle-8.13-bin.zip -d /usr/local/gradle
❏ ls /usr/local/gradle/gradle-8.13
LICENSE NOTICE  README  bin  init.d  lib
```

Step 3 - **Configure your system environment**

To install Gradle, the path to the unpacked files needs to be in your Path. Configure your **PATH** environment variable to include the **bin** directory of the unzipped distribution, e.g.:

```
❏ export PATH=$PATH:/usr/local/gradle/gradle-8.13/bin
```

Alternatively, you could also add the environment variable **GRADLE_HOME** and point this to the unzipped distribution. Instead of adding a specific version of Gradle to your **PATH**, you can add **\$GRADLE_HOME/bin** to your **PATH**. When upgrading to a different version of Gradle, simply change the **GRADLE_HOME** environment variable.

It's a good idea to edit **.bash_profile** in your home directory to add **GRADLE_HOME** variable:

```
export GRADLE_HOME=/usr/local/gradle/gradle-8.13
export PATH=$GRADLE_HOME/bin:$PATH
```

Windows installation

▼ *Installing manually*

Step 1 - **Download** the latest Gradle distribution

The distribution ZIP file comes in two flavors:

- Binary-only (bin)
- Complete (all) with docs and sources

We recommend downloading the bin file.

Step 2 - **Unpack the distribution**

Create a new directory `C:\Gradle` with **File Explorer**.

Open a second **File Explorer** window and go to the directory where the Gradle distribution was downloaded. Double-click the ZIP archive to expose the content. Drag the content folder `gradle-8.13` to your newly created `C:\Gradle` folder.

Alternatively, you can unpack the Gradle distribution ZIP into `C:\Gradle` using the archiver tool of your choice.

Step 3 - **Configure your system environment**

To install Gradle, the path to the unpacked files needs to be in your Path.

In **File Explorer** right-click on the **This PC** (or **Computer**) icon, then click **Properties** → **Advanced System Settings** → **Environmental Variables**.

Under **System Variables** select **Path**, then click **Edit**. Add an entry for `C:\Gradle\gradle-8.13\bin`. Click **OK** to save.

Alternatively, you can add the environment variable `GRADLE_HOME` and point this to the unzipped distribution. Instead of adding a specific version of Gradle to your **Path**, you can add `%GRADLE_HOME%\bin` to your **Path**. When upgrading to a different version of Gradle, just change the `GRADLE_HOME` environment variable.

Verify the installation

Open a console (or a Windows command prompt) and run `gradle -v` to run gradle and display the version, e.g.:

```
❯ gradle -v
```

```
-----  
Gradle 8.13  
-----
```

```
Build time:    2024-06-17 18:10:00 UTC  
Revision:     6028379bb5a8512d0b2c1be6403543b79825ef08
```

```
Kotlin:      1.9.23
Groovy:      3.0.21
Ant:         Apache Ant(TM) version 1.10.13 compiled on January 4 2023
Launcher JVM: 11.0.23 (Eclipse Adoptium 11.0.23+9)
Daemon JVM:  /Library/Java/JavaVirtualMachines/temurin-11.jdk/Contents/Home (no JDK
specified, using current Java home)
OS:         Mac OS X 14.5 aarch64
```

You can verify the integrity of the Gradle distribution by downloading the SHA-256 file (available from the [releases page](#)) and following these [verification instructions](#).

Compatibility Matrix

The sections below describe Gradle's compatibility with several integrations. Versions not listed here may or may not work.

Java Runtime

Gradle runs on the Java Virtual Machine (JVM), which is often provided by either a JDK or JRE. A JVM version between 8 and 23 is required to execute Gradle. JVM 24 and later versions are not yet supported.

Executing the Gradle daemon with JVM 16 or earlier has been deprecated and will become an error in Gradle 9.0. The Gradle wrapper, Gradle client, Tooling API client, and TestKit client will remain compatible with JVM 8.

JDK 6 and 7 can be used for [compilation](#). Testing with JVM 6 and 7 is deprecated and will not be supported in Gradle 9.0.

Any fully supported version of Java can be used for compilation or testing. However, the latest Java version may only be supported for compilation or testing, not for running Gradle. Support is achieved using [toolchains](#) and applies to all tasks supporting toolchains.

See the table below for the Java version supported by a specific Gradle release:

Table 1. Java Compatibility

Java version	Support for toolchains	Support for running Gradle
8	N/A	2.0
9	N/A	4.3
10	N/A	4.7
11	N/A	5.0
12	N/A	5.4
13	N/A	6.0
14	N/A	6.3
15	6.7	6.7

Java version	Support for toolchains	Support for running Gradle
16	7.0	7.0
17	7.3	7.3
18	7.5	7.5
19	7.6	7.6
20	8.1	8.3
21	8.4	8.5
22	8.7	8.8
23	8.10	8.10
24	N/A	N/A

NOTE

We only list versions in the table above once we have tested that they work without any warnings. However, thanks to the toolchain support, Gradle will often work with the latest Java version before then. We encourage users to try it out and let us know.

Kotlin

Gradle is tested with Kotlin 1.6.10 through 2.1.20-Beta2. Beta and RC versions may or may not work.

Table 2. Embedded Kotlin version

Embedded Kotlin version	Minimum Gradle version	Kotlin Language version
1.3.10	5.0	1.3
1.3.11	5.1	1.3
1.3.20	5.2	1.3
1.3.21	5.3	1.3
1.3.31	5.5	1.3
1.3.41	5.6	1.3
1.3.50	6.0	1.3
1.3.61	6.1	1.3
1.3.70	6.3	1.3
1.3.71	6.4	1.3
1.3.72	6.5	1.3
1.4.20	6.8	1.3
1.4.31	7.0	1.4
1.5.21	7.2	1.4
1.5.31	7.3	1.4

Embedded Kotlin version	Minimum Gradle version	Kotlin Language version
1.6.21	7.5	1.4
1.7.10	7.6	1.4
1.8.10	8.0	1.8
1.8.20	8.2	1.8
1.9.0	8.3	1.8
1.9.10	8.4	1.8
1.9.20	8.5	1.8
1.9.22	8.7	1.8
1.9.23	8.9	1.8
1.9.24	8.10	1.8
2.0.20	8.11	1.8
2.0.21	8.12	1.8

Groovy

Gradle is tested with Groovy 1.5.8 through 4.0.0.

Gradle plugins written in Groovy must use Groovy 3.x for compatibility with Gradle and Groovy DSL build scripts.

Android

Gradle is tested with Android Gradle Plugin 7.3 through 8.8. Alpha and beta versions may or may not work.

Target Platforms

Gradle supports a defined set of platform targets, which are combinations of:

- Operating system and version
- Architecture
- File system watching compatibility

The following table lists the officially supported platforms for Gradle:

Table 3. Supported Platforms

OS	Architecture
Ubuntu 22	amd64
Windows 10	amd64
macOS 12	amd64

OS	Architecture
	aarch64
Ubuntu 16	amd64
	aarch64
Alpine 3.20	amd64
CentOS 9	amd64

NOTE

Currently, all Gradle tests run with the default file-systems of the platform, i.e. `ext4` for Ubuntu, Amazon Linux and CentOS, `NTFS` for Windows, and `APFS` for macOS.

Platforms not listed above may work with Gradle but are not actively tested.

The Feature Lifecycle

Gradle is under constant development. New versions are delivered on a regular and frequent basis (approximately every six weeks) as described in [the section on end-of-life support](#).

Continuous improvement combined with frequent delivery allows new features to be available to users early. Early users provide invaluable feedback, which is incorporated into the development process.

Getting new functionality into the hands of users regularly is a core value of the Gradle platform.

At the same time, API and feature stability are taken very seriously and considered a core value of the Gradle platform. Design choices and automated testing are engineered into the development process and formalized by [the section on backward compatibility](#).

The Gradle *feature lifecycle* has been designed to meet these goals. It also communicates to users of Gradle what the state of a feature is. The term *feature* typically means an API or DSL method or property in this context, but it is not restricted to this definition. Command line arguments and modes of execution (e.g. the Build Daemon) are two examples of other features.

Feature States

Features can be in one of four states:

1. [Internal](#)
2. [Incubating](#)
3. [Public](#)
4. [Deprecated](#)

1. Internal

Internal features are not designed for public use and are only intended to be used by Gradle itself. They can change in any way at any point in time without any notice. Therefore, we recommend avoiding the use of such features. *Internal* features are not documented. If it appears in this User

Manual, the DSL Reference, or the API Reference, then the feature is not *internal*.

Internal features may evolve into public features.

2. Incubating

Features are introduced in the *incubating* state to allow real-world feedback to be incorporated into the feature before making it public. It also gives users willing to test potential future changes early access.

A feature in an *incubating* state may change in future Gradle versions until it is no longer *incubating*. Changes to *incubating* features for a Gradle release will be highlighted in the release notes for that release. The *incubation* period for new features varies depending on the feature's scope, complexity, and nature.

Features in *incubation* are indicated. In the source code, all methods/properties/classes that are *incubating* are annotated with `incubating`. This results in a special mark for them in the DSL and API references.

If an *incubating* feature is discussed in this User Manual, it will be explicitly said to be in the *incubating* state.

Feature Preview API

The feature preview API allows certain *incubating* features to be activated by adding `enableFeaturePreview('FEATURE')` in your *settings* file. Individual preview features will be announced in release notes.

When *incubating* features are either promoted to *public* or removed, the feature preview flags for them become obsolete, have no effect, and should be removed from the settings file.

3. Public

The default state for a non-internal feature is *public*. Anything documented in the User Manual, DSL Reference, or API reference that is not explicitly said to be *incubating* or *deprecated* is considered *public*. Features are said to be **promoted** from an *incubating* state to *public*. The release notes for each release indicate which previously *incubating* features are being promoted by the release.

A *public* feature will **never** be removed or intentionally changed without undergoing *deprecation*. All public features are subject to the backward compatibility policy.

4. Deprecated

Some features may be replaced or become irrelevant due to the natural evolution of Gradle. Such features will eventually be removed from Gradle after being *deprecated*. A *deprecated* feature may become stale until it is finally removed according to the backward compatibility policy.

Deprecated features are indicated to be so. In the source code, all methods/properties/classes that are *deprecated* are annotated with “@java.lang.Deprecated” which is reflected in the DSL and API References. In most cases, there is a replacement for the deprecated element, which will be

described in the documentation. Using a *deprecated* feature will result in a runtime warning in Gradle's output.

The use of *deprecated* features should be avoided. The release notes for each release indicate any features being *deprecated* by the release.

Backward compatibility policy

Gradle provides backward compatibility across major versions (e.g., **1.x**, **2.x**, etc.). Once a public feature is introduced in a Gradle release, it will remain indefinitely unless deprecated. Once deprecated, it may be removed in the next major release. Deprecated features may be supported across major releases, but this is not guaranteed.

Release end-of-life Policy

Every day, a new nightly build of Gradle is created.

This contains all of the changes made through Gradle's extensive continuous integration tests during that day. Nightly builds may contain new changes that may or may not be stable.

The Gradle team creates a pre-release distribution called a release candidate (RC) for each minor or major release. When no problems are found after a short time (usually a week), the release candidate is promoted to a general availability (GA) release. If a regression is found in the release candidate, a new RC distribution is created, and the process repeats. Release candidates are supported for as long as the release window is open, but they are not intended to be used for production. Bug reports are greatly appreciated during the RC phase.

The Gradle team may create additional patch releases to replace the final release due to critical bug fixes or regressions. For instance, Gradle 5.2.1 replaces the Gradle 5.2 release.

Once a release candidate has been made, all feature development moves on to the next release for the latest major version. As such, each minor Gradle release causes the previous minor releases in the same major version to become end-of-life (EOL). EOL releases do not receive bug fixes or feature backports.

For major versions, Gradle will backport critical fixes and security fixes to the last minor in the previous major version. For example, when Gradle 7 was the latest major version, several releases were made in the 6.x line, including Gradle 6.9 (and subsequent releases).

As such, each major Gradle release causes:

- The previous major version becomes maintenance only. It will only receive critical bug fixes and security fixes.
- The major version before the previous one to become end-of-life (EOL), and that release line will not receive any new fixes.

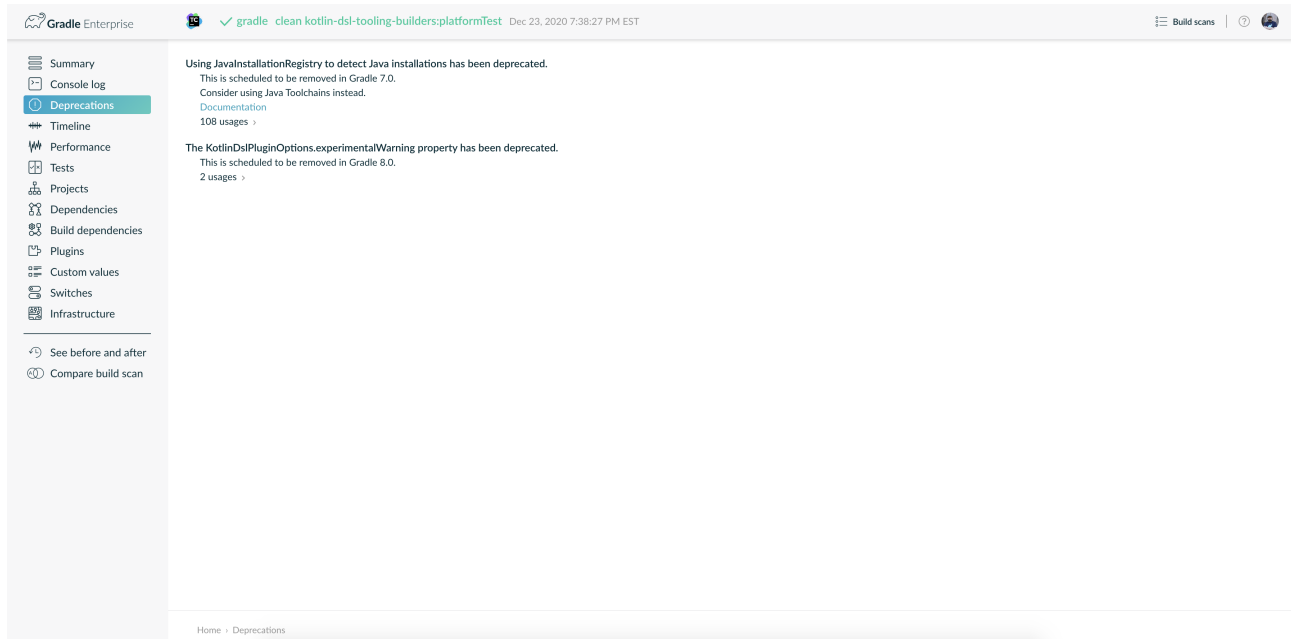
UPGRADING

Upgrading your build from Gradle 8.x to the latest

This chapter provides the information you need to migrate your Gradle 8.x builds to the latest Gradle release. For migrating from Gradle 4.x, 5.x, 6.x, or 7.x, see the [older migration guide](#) first.

We recommend the following steps for all users:

1. Try running `gradle help --scan` and view the [deprecations view](#) of the generated build scan.



This lets you see any deprecation warnings that apply to your build.

Alternatively, you can run `gradle help --warning-mode=all` to see the deprecations in the console, though it may not report as much detailed information.

2. Update your plugins.

Some plugins will break with this new version of Gradle because they use internal APIs that have been removed or changed. The previous step will help you identify potential problems by issuing deprecation warnings when a plugin tries to use a deprecated part of the API.

3. Run `gradle wrapper --gradle-version 8.13` to update the project to 8.13.
4. Try to run the project and debug any errors using the [Troubleshooting Guide](#).

Upgrading from 8.12 and earlier

Potential breaking changes

Changes to JvmTestSuite

The `testType` property was removed from `JvmTestSuite` and both the `TestSuiteTargetName` and

`TestSuiteType` attributes have been removed. Test reports and JaCoCo reports can now be aggregated between projects by specifying the name of the test suite in the target project to aggregate.

See below for additional details.

Changes to Test Report Aggregation and Jacoco Aggregation

Several changes have been made to the incubating [Test Report Aggregation](#) and [JaCoCo Report Aggregation](#) plugins.

The plugins now create a single test results variant for each test suite, containing all test results for the entire suite, instead of one variant for each test target. This change allows the aggregation plugins to aggregate test suites with multiple targets, where previously this would result in an ambiguous variant selection error.

In the future, as we continue to develop these plugins, we plan to once again create one results variant per test suite target, allowing test results from certain targets to be explicitly aggregated.

The `testType` property on [JacocoCoverageReport](#) and [AggregateTestReport](#) has been removed and replaced with a new `testSuiteName` property:

Previously:

```
reporting {
    reports {
        val testCodeCoverageReport by creating(JacocoCoverageReport::class) {
            testType = TestSuiteType.UNIT_TEST
        }
    }
}
```

Now:

```
reporting {
    reports {
        val testCodeCoverageReport by creating(JacocoCoverageReport::class) {
            testSuiteName = "test"
        }
    }
}
```

Changed behavior when calling `BuildLauncher.addJvmArguments`

Issue ([#31426](#)) was fixed, that caused `BuildLauncher.addJvmArguments` to override flags coming from the `org.gradle.jvmargs` system property. Please ensure that you are not relying on this behavior when upgrading to Gradle 8.13. If system properties needs to be overridden, `BuildLauncher.setJvmArguments` should be used instead.

```
val buildLauncher: BuildLauncher = connector.connect().newBuild()
buildLauncher.setJvmArguments("-Xmx2048m", "-Dmy.custom.property=value")
```

Upgrade to ASM 9.7.1

ASM was upgraded from 9.6 to [9.7.1](#) to ensure earlier compatibility for Java 24.

Source level deprecation of `Project.task` methods

Eager task creation methods on [the `Project` interface](#) have been marked `@Deprecated` and will generate compiler and IDE warnings when used in build scripts or plugin code. There is not yet a Gradle deprecation warning emitted for their use.

However, if the build is configured to fail on warnings during Kotlin script or plugin code compilation, this change may cause the build to fail.

A standard Gradle deprecation warning will be printed upon use when these methods are fully deprecated in a future version.

Deprecations

Recursively querying `AttributeContainer` in lazy provider

In Gradle 9.0, querying the contents of an `AttributeContainer` from within an attribute value provider of the same container will become an error.

The following example showcases the forbidden behavior:

```
AttributeContainer container = getAttributeContainer();
Attribute<String> firstAttribute = Attribute.of("first", String.class);
Attribute<String> secondAttribute = Attribute.of("second", String.class);
container.attributeProvider(firstAttribute, project.getProviders().provider() -> {
    // Querying the contents of the container within an attribute value provider
    // will become an error.
    container.getAttribute(secondAttribute);
    return "first";
}));
```

Deprecated `org.gradle.api.artifacts.transform.VariantTransformConfigurationException`

There is no good public use case for this exception, and it is not intended to be thrown by users. It will be replaced by `org.gradle.api.internal.artifacts.transform.VariantTransformConfigurationException` for internal use only in Gradle 9.0.

Deprecated properties in the incubating `UpdateDaemonJvm`

The following properties of `UpdateDaemonJvm` are now deprecated:

- `jvmVersion`
- `jvmVendor`

They are replaced by `languageVersion` and `vendor` respectively. This allows the configuration of a Java toolchain spec and the `UpdateDaemonJvm` task to be interchangeable.

Note that due to the change of type for the vendor property, executing `updateDaemonJvm` with the `jvmVendor` property will result in the task failing. See [the documentation](#) for the new configuration option.

Declaring boolean properties with `is`-prefix and `Boolean` types

Gradle property names are derived by following the Java Bean specification with one exception. Gradle recognizes methods with a `Boolean` return type and a `is`-prefix as a boolean property. This is behavior inherited from Groovy originally. Groovy 4 more closely follows the Java Bean specification and [no longer supports this exception](#).

Gradle will emit a deprecation warning when it detects that a boolean property is derived from a method with a `Boolean` return type and `is`-prefix. In Gradle 9.0, these methods will no longer be treated as defining a Gradle property. This may cause tasks to be considered up-to-date, even when a `Boolean` property appears to be an input.

There are two options to fix this:

1. Introduce a new method that starts with `get` instead of `is` which has the same behavior. The old method does not need to be removed (in order to preserve binary compatibility), but may need adjustments as indicated below.
 - It is recommended to deprecate the `is`- method, and then remove it in a future major version.
2. Change the type of the property (both get and set) to `boolean`. **This is a breaking change.**

For task input properties using the first option, you should also annotate the old `is`- method with `@Deprecated` and `@ReplacedBy` to ensure it is not used by Gradle. For example, this code:

```
class MyValue {
    private Boolean property = Boolean.TRUE;

    @Input
    Boolean isProperty() { return property; }
}
```

Should be replaced with the following:

```
class MyValue {
    private Boolean property = Boolean.TRUE;

    @Deprecated
    @ReplacedBy("getProperty")
```

```
Boolean isProperty() { return property; }

@Input
Boolean getProperty() { return property; }
}
```

Upgrading from 8.11 and earlier

Potential breaking changes

Upgrade to Kotlin 2.0.21

The embedded Kotlin has been updated from 2.0.20 to [Kotlin 2.0.21](#).

Upgrade to Ant 1.10.15

Ant has been updated to [Ant 1.10.15](#).

Upgrade to Zinc 1.10.4

Zinc has been updated to [1.10.4](#).

Swift SDK discovery

To determine the location of the Mac OS X SDK for Swift, Gradle now passes the `--sdk macosx` arguments to `xcrun`. This is necessary because the SDK could be discovered inconsistently without this argument across different environments.

Source level deprecation of `TaskContainer.create` methods

Eager task creation methods on [the `TaskContainer` interface](#) have been marked `@Deprecated` and will generate compiler and IDE warnings when used in build scripts or plugin code. There is not yet a Gradle deprecation warning emitted for their use.

However, if the build is configured to fail on warnings during Kotlin script or plugin code compilation, this behavior may cause the build to fail.

A standard Gradle deprecation warning will be printed upon use when these methods are fully deprecated in a future version.

Deprecations

Deprecated Ambiguous Transformation Chains

Previously, when at least two equal-length chains of [artifact transforms](#) were available that would produce compatible variants that would each satisfy a resolution request, Gradle would arbitrarily, and silently, pick one.

Now, Gradle emits a deprecation warning that explains this situation:

```
There are multiple distinct artifact transformation chains of the same length that
```

would satisfy this request. This behavior has been deprecated. This will fail with an error in Gradle 9.0.

Found multiple transformation chains that produce a variant of 'root project :' with requested attributes:

- color 'red'
- texture 'smooth'

Found the following transformation chains:

- From configuration ':squareBlueSmoothElements':
 - With source attributes:
 - artifactType 'txt'
 - color 'blue'
 - shape 'square'
 - texture 'smooth'
 - Candidate transformation chains:
 - Transformation chain: 'ColorTransform':
 - 'BrokenColorTransform':
 - Converts from attributes:
 - color 'blue'
 - texture 'smooth'
 - To attributes:
 - color 'red'
 - Transformation chain: 'ColorTransform2':
 - 'BrokenColorTransform2':
 - Converts from attributes:
 - color 'blue'
 - texture 'smooth'
 - To attributes:
 - color 'red'

Remove one or more registered transforms, or add additional attributes to them to ensure only a single valid transformation chain exists.

In such a scenario, Gradle has no way to know which of the two (or more) possible transformation chains should be used. Picking an arbitrary chain can lead to inefficient performance or unexpected behavior changes when seemingly unrelated parts of the build are modified. This is potentially a very complex situation and the message now fully explains the situation by printing all the registered transforms in order, along with their source (input) variants for each candidate chain.

When encountering this type of failure, build authors should either:

1. Add additional, distinguishing attributes when registering transforms present in the chain, to ensure that only a single chain will be selectable to satisfy the request
2. Request additional attributes to disambiguate which chain is selected (if they result in non-identical final attributes)
3. Remove unnecessary registered transforms from the build

This will become an error in Gradle 9.0.

init must run alone

The **init** task must run by itself. This task should not be combined with other tasks in a single Gradle invocation.

Running **init** in the same invocation as other tasks will become an error in Gradle 9.0.

For instance, this will **not** be allowed:

```
> gradlew init tasks
```

Calling **Task.getProject()** from a task action

Calling **Task.getProject()** from a task action at execution time is now deprecated and will be made an error in Gradle 10.0. This method can still be used during configuration time.

The deprecation is only issued if the configuration cache is **not** enabled. When the configuration cache is enabled, calls to **Task.getProject()** are reported as configuration cache problems instead.

This deprecation was originally introduced in [Gradle 7.4](#) but was only issued when the **STABLE_CONFIGURATION_CACHE** feature flag was enabled. That feature flag no longer controls this deprecation. This is another step towards moving users away from idioms that are incompatible with the configuration cache, which will become the only mode supported by Gradle in a future release.

Please refer to the [configuration cache documentation](#) for alternatives to invoking **Task.getProject()** at execution time that are compatible with the configuration cache.

Groovy "space assignment" syntax

Currently, there are multiple ways to set a property with Groovy DSL syntax:

```
---
propertyName = value
setProperty(propertyName, value)
setProperty(propertyName, value)
propertyName(propertyName, value)
propertyName value
---
```

The latter one, "space-assignment", is a Gradle-specific feature that is not part of the Groovy language. In regular Groovy, this is just a method call: **propertyName(value)**, and Gradle generates **propertyName** method in the runtime if this method hasn't been present already. This feature may be a source of confusion (especially for new users) and adds an extra layer of complexity for users and the Gradle codebase without providing any significant value. Sometimes, classes declare methods with the same name, and these may even have semantics that are different from a plain assignment.

These generated methods are now deprecated and will be removed in Gradle 10.0, and both

`propertyName value` and `propertyName(value)` will stop working unless the explicit method `propertyName` is defined. Use explicit assignment `propertyName = value` instead.

For explicit methods, consider using the `propertyName(value)` syntax instead of `propertyName value` for clarity. For example, `jvmArgs "some", "arg"` can be replaced with `jvmArgs("some", "arg")` or with `jvmArgs = ["some", "arg"]` for `Test` tasks.

If you have a big project, to replace occurrences of space-assignment syntax you can use, for example, the following `sed` command:

```
---
find . -name 'build.gradle' -type f -exec sed -i.bak -E 's/([A-Za-z]|^)(replaceme)[
\t]*([^= \t{]|\1\2 = \3/g' {} +
---
```

You should replace `replaceme` with one or more property names you want to replace, separated by `|`, e.g. `(url|group)`.

DependencyInsightReportTask.getDependencySpec

The method was deprecated because it was not intended for public use in build scripts.

ReportingExtension.baseDir

`ReportingExtension.getBaseDir()`, `ReportingExtension.setBaseDir(File)`, and `ReportingExtension.setBaseDir(Object)` were deprecated. They should be replaced with `ReportingExtension.getBaseDirectory()` property.

Upgrading from 8.10 and earlier

Potential breaking changes

Upgrade to Kotlin 2.0.20

The embedded Kotlin has been updated from 1.9.24 to [Kotlin 2.0.20](#). Also see the [Kotlin 2.0.10](#) and [Kotlin 2.0.0](#) release notes.

The default `kotlin-test` version in JVM test suites has been upgraded to 2.0.20 as well.

Kotlin DSL scripts are still compiled with Kotlin language version set to 1.8 for backward compatibility.

Gradle daemon JVM configuration via toolchain

The type of the property `UpdateDaemonJvm.jvmVersion` is now `Property<JavaLanguageVersion>`.

If you configured the task in a build script, you will need to replace:

```
jvmVersion = JavaVersion.VERSION_17
```

With:

```
jvmVersion = JavaLanguageVersion.of(17)
```

Using the CLI options to configure which JVM version to use for the Gradle Daemon has no impact.

Name matching changes

The name-matching logic has been updated to treat numbers as word boundaries for camelCase names. Previously, a request like `unique` would match both `uniqueA` and `unique1`. Such a request will now fail due to ambiguity. To avoid issues, use the exact name instead of a shortened version.

This change impacts:

- Task selection
- Project selection
- Configuration selection in dependency report tasks

Deprecations

Deprecated JavaHome property of ForkOptions

The `JavaHome` property of the `ForkOptions` type has been deprecated and will be removed in Gradle 9.0.

Use `JVM Toolchains`, or the `executable` property instead.

Deprecated mutating buildscript configurations

Starting in Gradle 9.0, mutating configurations in a script's `buildscript` block will result in an error. This applies to project, settings, init, and standalone scripts.

The buildscript configurations block is only intended to control buildscript classpath resolution.

Consider the following script that creates a new buildscript configuration in a Settings script and resolves it:

```
buildscript {
    configurations {
        create("myConfig")
    }
    dependencies {
        "myConfig"("org:foo:1.0")
    }
}

val files = buildscript.configurations["myConfig"].files
```

This pattern is sometimes used to resolve dependencies in Settings, where there is no other way to obtain a Configuration. Resolving dependencies in this context is not recommended. Using a detached configuration is a possible but discouraged alternative.

The above example can be modified to use a detached configuration:

```
val myConfig = buildscript.configurations.detachedConfiguration(
    buildscript.dependencies.create("org:foo:1.0")
)

val files = myConfig.files
```

Selecting Maven variants by configuration name

Starting in Gradle 9.0, selecting variants by name from non-Ivy external components will be forbidden.

Selecting variants by name from local components will still be permitted; however, this pattern is discouraged. Variant aware dependency resolution should be preferred over selecting variants by name for local components.

The following dependencies will fail to resolve when targeting a non-Ivy external component:

```
dependencies {
    implementation(group: "com.example", name: "example", version: "1.0",
configuration: "conf")
    implementation("com.example:example:1.0") {
        targetConfiguration = "conf"
    }
}
```

Deprecated manually adding to configuration container

Starting in Gradle 9.0, manually adding configuration instances to a configuration container will result in an error. Configurations should only be added to the container through the eager or lazy factory methods. Detached configurations and copied configurations should not be added to the container.

Calling the following methods on [ConfigurationContainer](#) will be forbidden: - add(Configuration) - addAll(Collection) - addLater(Provider) - addAllLater(Provider)

Deprecated `ProjectDependency#getDependencyProject()`

The `ProjectDependency#getDependencyProject()` method has been deprecated and will be removed in Gradle 9.0.

Accessing the mutable project instance of other projects should be avoided.

To discover details about all projects that were included in a resolution, inspect the full [ResolutionResult](#). Project dependencies are exposed in the [DependencyResult](#). See the user guide section on [programmatic dependency resolution](#) for more details on this API. This is the only reliable way to find all projects that are used in a resolution. Inspecting only the declared `ProjectDependency`s` may miss transitive or substituted project dependencies.

To get the identity of the target project, use the new Isolated Projects safe project path method: `ProjectDependency#getPath()`.

To access or configure the target project, consider this direct replacement:

```
val projectDependency: ProjectDependency = getSomeProjectDependency()

// Old way:
val someProject = projectDependency.dependencyProject

// New way:
val someProject = project.project(projectDependency.path)
```

This approach will not fetch project instances from different builds.

Deprecated `ResolvedConfiguration.GetFiles()` and `LenientConfiguration.GetFiles()`

The `ResolvedConfiguration.GetFiles()` and `LenientConfiguration.GetFiles()` methods have been deprecated and will be removed in Gradle 9.0.

These deprecated methods do not track task dependencies, unlike their replacements.

```
val deprecated: Set<File> = conf.resolvedConfiguration.files
val replacement: FileCollection = conf.incoming.files

val lenientDeprecated: Set<File> =
    conf.resolvedConfiguration.lenientConfiguration.files
val lenientReplacement: FileCollection = conf.incoming.artifactView {
    isLenient = true
}.files
```

Deprecated `AbstractOptions`

The `AbstractOptions` class has been deprecated and will be removed in Gradle 9.0. All classes extending `AbstractOptions` will no longer extend it.

As a result, the `AbstractOptions#define(Map)` method will no longer be present. This method exposes a non-type-safe API and unnecessarily relies on reflection. It can be replaced by directly setting the properties specified in the map.

Additionally, `CompileOptions#fork(Map)`, `CompileOptions#debug(Map)`, and `GroovyCompileOptions#fork(Map)`, which depend on `define`, are also deprecated for removal in Gradle 9.0.

Consider the following example of the deprecated behavior and its replacement:

```
tasks.withType(JavaCompile) {
    // Deprecated behavior
```

```

options.define(encoding: 'UTF-8')
options.fork(memoryMaximumSize: '1G')
options.debug(debugLevel: 'lines')

// Can be replaced by
options.encoding = 'UTF-8'

options.fork = true
options.forkOptions.memoryMaximumSize = '1G'

options.debug = true
options.debugOptions.debugLevel = 'lines'
}

```

Deprecated `Dependency#contentEquals(Dependency)`

The `Dependency#contentEquals(Dependency)` method has been deprecated and will be removed in Gradle 9.0.

The method was originally intended to compare dependencies based on their actual target component, regardless of whether they were of different dependency type. The existing method does not behave as specified by its Javadoc, and we do not plan to introduce a replacement that does.

Potential migrations include using `Object.equals(Object)` directly, or comparing the fields of dependencies manually.

Deprecated `Project#exec` and `Project#javaexec`

The `Project#exec(Closure)`, `Project#exec(Action)`, `Project#javaexec(Closure)`, `Project#javaexec(Action)` methods have been deprecated and will be removed in Gradle 9.0.

These methods are scheduled for removal as part of the ongoing effort to make writing configuration-cache-compatible code easier. There is no way to use these methods without breaking configuration cache requirements so it is recommended to migrate to a compatible alternative. The appropriate replacement for your use case depends on the context in which the method was previously called.

At execution time, for example in `@TaskAction` or `doFirst/doLast` callbacks, the use of `Project` instance is not allowed when the configuration cache is enabled. To run external processes, tasks should use an `injected ExecOperation` service, which has the same API and can act as a drop-in replacement. The standard Java/Groovy/Kotlin process APIs, like `java.lang.ProcessBuilder` can be used as well.

At configuration time, only special Provider-based APIs must be used to run external processes when the configuration cache is enabled. You can use `ProviderFactory.exec` and `ProviderFactory.javaexec` to obtain the output of the process. A custom `ValueSource` implementation can be used for more sophisticated scenarios. The [configuration cache guide](#) has a more elaborate example of using these APIs.

Detached Configurations should not use `extendsFrom`

Detached configurations should not extend other configurations using `extendsFrom`.

This behavior has been deprecated and will become an error in Gradle 9.0.

To create extension relationships between configurations, you should change to using non-detached configurations created via the other factory methods present in the project's `ConfigurationContainer`.

Deprecated customized Gradle logging

The `Gradle#useLogger(Object)` method has been deprecated and will be removed in Gradle 9.0.

This method was originally intended to customize logs printed by Gradle. However, it only allows intercepting a subset of the logs and cannot work with the `configuration cache`. We do not plan to introduce a replacement for this feature.

Unnecessary options on compile options and doc tasks have been deprecated

Gradle's API allowed some properties that represented nested groups of properties to be replaced wholesale with a setter method. This was awkward and unusual to do and would sometimes require the use of internal APIs. The setters for these properties will be removed in Gradle 9.0 to simplify the API and ensure consistent behavior. Instead of using the setter method, these properties should be configured by calling the getter and configuring the object directly or using the convenient configuration method. For example, in `CompileOptions`, instead of calling the `setForkOptions` setter, you can call `getForkOptions()` or `forkOptions(Action)`.

The affected properties are:

- `CompileOptions.getDebugOptions`
- `CompileOptions.getForkOptions`
- `GroovyCompileOptions.getForkOptions`
- `ScalaDoc.getScalaDocOptions`
- `BaseScalaCompileOptions.getForkOptions`
- `BaseScalaCompileOptions.getIncrementalOptions`

Deprecated `Javadoc.isVerbose()` and `Javadoc.setVerbose(boolean)`

These methods on `Javadoc` have been deprecated and will be removed in Gradle 9.0.

- `isVerbose()` is replaced by `getOptions().isVerbose()`
- Calling `setVerbose(boolean)` with `true` is replaced by `getOptions().verbose()`
- Calling `setVerbose(false)` did nothing.

Upgrading from 8.9 and earlier

Potential breaking changes

JavaCompile tasks may fail when using a JRE even if compilation is not necessary

The `JavaCompile` tasks may sometimes fail when using a JRE instead of a JDK. This is due to changes in the toolchain resolution code, which enforces the presence of a compiler when one is requested. The `java-base` plugin uses the `JavaCompile` tasks it creates to determine the default source and target compatibility when `sourceCompatibility/targetCompatibility` or `release` are not set. With the new enforcement, the absence of a compiler causes this to fail when only a JRE is provided, even if no compilation is needed (e.g., in projects with no sources).

This can be fixed by setting the `sourceCompatibility/targetCompatibility` explicitly in the `java` extension, or by setting `sourceCompatibility/targetCompatibility` or `release` in the relevant task(s).

Upgrade to Kotlin 1.9.24

The embedded Kotlin has been updated from 1.9.23 to [Kotlin 1.9.24](#).

Upgrade to Ant 1.10.14

Ant has been updated to [Ant 1.10.14](#).

Upgrade to JaCoCo 0.8.12

JaCoCo has been updated to [0.8.12](#).

Upgrade to Groovy 3.0.22

Groovy has been updated to [Groovy 3.0.22](#).

Deprecations

Running Gradle on older JVMs

Starting in Gradle 9.0, Gradle will require JVM 17 or later to run. Most Gradle APIs will be compiled to target JVM 17 bytecode.

Gradle will still support compiling Java code to target JVM version 6 or later. The [target JVM version](#) of the compiled code can be configured separately from the JVM version used to run Gradle.

All Gradle clients (wrapper, launcher, Tooling API and TestKit) will remain compatible with JVM 8 and will be compiled to target JVM 8 bytecode. Only the Gradle daemon will require JVM 17 or later. These clients can be configured to run Gradle builds with a different JVM version than the one used to run the client:

- Using [Daemon JVM criteria](#) (an incubating feature)
- Setting the `org.gradle.java.home` [Gradle property](#)
- Using the [ConfigurableLauncher#setJavaHome](#) method on the Tooling API

Alternatively, the `JAVA_HOME` environment variable can be set to a JVM 17 or newer, which will run both the client and daemon with the same version of the JVM.

Running Gradle builds with `--no-daemon` or using [ProjectBuilder](#) in tests will require JVM version 17 or later. The worker API will remain compatible with JVM 8, and running JVM tests will require JVM 8.

We decided to upgrade the minimum version of the Java runtime for a number of reasons:

- Dependencies are beginning to drop support for older versions and may not release security patches.
- Significant language improvements between Java 8 and Java 17 cannot be used without upgrading.
- Some of the most popular plugins already require JVM 17 or later.
- Download metrics for Gradle distributions show that JVM 17 is widely used.

Deprecated consuming non-consumable configurations from Ivy

In prior versions of Gradle, it was possible to consume non-consumable configurations of a project using published Ivy metadata. An Ivy dependency may sometimes be substituted for a project dependency, either explicitly through the [DependencySubstitutions](#) API or through included builds. When this happens, configurations in the substituted project could be selected that were marked as non-consumable.

Consuming non-consumable configurations in this manner is deprecated and will result in an error in Gradle 9.0.

Deprecated extending configurations in the same project

In prior versions of Gradle, it was possible to extend a configuration in a different project.

The hierarchy of a Project's configurations should not be influenced by configurations in other projects. Cross-project hierarchies can lead to unexpected behavior when configurations are extended in a way that is not intended by the configuration's owner.

Projects should also never access the mutable state of another project. Since Configurations are mutable, extending configurations across project boundaries restricts the parallelism that Gradle can apply.

Extending configurations in different projects is deprecated and will result in an error in Gradle 9.0.

Upgrading from 8.8 and earlier

Potential breaking changes

Change to toolchain provisioning

In previous versions of Gradle, toolchain provisioning could leave a partially provisioned toolchain in place **with a marker file indicating that the toolchain was fully provisioned**. This could lead to strange behavior with the toolchain. In Gradle 8.9, the toolchain is fully provisioned before the marker file is written. However, to not detect potentially broken toolchains, a different marker file

([.ready](#)) is used. This means all your existing toolchains will be re-provisioned the first time you use them with Gradle 8.9. Gradle 8.9 also writes the old marker file ([provisioned.ok](#)) to indicate that the toolchain was fully provisioned. This means that if you return to an older version of Gradle, an 8.9-provisioned toolchain will **not** be re-provisioned.

Upgrade to Kotlin 1.9.23

The embedded Kotlin has been updated from 1.9.22 to [Kotlin 1.9.23](#).

Change the encoding of daemon log files

In previous versions of Gradle, the daemon log file, located at `$GRADLE_USER_HOME/daemon/8.13/`, was encoded with the default JVM encoding. This file is now always encoded with UTF-8 to prevent clients who may use different default encodings from reading data incorrectly. This change may affect third-party tools trying to read this file.

Compiling against Gradle implementation classpath

In previous versions of Gradle, Java projects that had no declared dependencies could implicitly compile against Gradle's runtime classes. This means that some projects were able to compile without any declared dependencies even though they referenced Gradle runtime classes. This situation is unlikely to arise in projects since IDE integration and test execution would be compromised. However, if you need to utilize the Gradle API, declare a `gradleApi` dependency or apply the `java-gradle-plugin` plugin.

Configuration cache implementation packages now under `org.gradle.internal`

References to Gradle types not part of the public API should be avoided, as their direct use is unsupported. Gradle internal implementation classes may suffer breaking changes (or be renamed or removed) from one version to another without warning.

Users need to distinguish between the API and internal parts of the Gradle codebase. This is typically achieved by including `internal` in the implementation package names. However, before this release, the configuration cache subsystem did not follow this pattern.

To address this issue, all code initially under the `org.gradle.configurationcache*` packages has been moved to new internal packages (`org.gradle.internal.*`).

File-system watching on macOS 11 (Big Sur) and earlier is disabled

Since Gradle 8.8, file-system watching has only been supported on macOS 12 (Monterey) and later. We added a check to automatically disable file-system watching on macOS 11 (Big Sur) and earlier versions.

Possible change to JDK8-based compiler output when annotation processors are used

The Java compilation infrastructure has been updated to use the [Problems API](#). This change will supply the Tooling API clients with structured, rich information about compilation issues.

The feature should not have any visible impact on the usual build output, with JDK8 being an exception. When annotation processors are used in the compiler, the output message differs

slightly from the previous ones.

The change mainly manifests itself in typename printed. For example, Java standard types like `java.lang.String` will be reported as `java.lang.String` instead of `String`.

Upgrading from 8.7 and earlier

Deprecations

Deprecate mutating configuration after observation

To ensure the accuracy of dependency resolution, Gradle checks that Configurations are not mutated after they have been used as part of a dependency graph.

- Resolvable configurations should not have their resolution strategy, dependencies, hierarchy, etc., modified after they have been resolved.
- Consumable configurations should not have their dependencies, hierarchy, attributes, etc. modified after they have been published or consumed as a variant.
- Dependency scope configurations should not have their dependencies, constraints, etc., modified after a configuration that extends from them is observed.

In prior versions of Gradle, many of these circumstances were detected and handled by failing the build. However, some cases went undetected or did not trigger build failures. In Gradle 9.0, all changes to a configuration, once observed, will become an error. After a configuration of any type has been observed, it should be considered immutable. This validation covers the following properties of a configuration:

- Resolution Strategy
- Dependencies
- Constraints
- Exclude Rules
- Artifacts
- Role (consumable, resolvable, dependency scope)
- Hierarchy (`extendsFrom`)
- Others (Transitive, Visible)

Starting in Gradle 8.8, a deprecation warning will be emitted in cases that were not already an error. Usually, this deprecation is caused by mutating a configuration in a `beforeResolve` hook. This hook is only executed after a configuration is fully resolved but not when it is partially resolved for computing task dependencies.

Consider the following code that showcases the deprecated behavior:

build.gradle.kts

```
plugins {
    id("java-library")
}

configurations.runtimeClasspath {
    // `beforeResolve` is not called before the configuration is partially
    // resolved for
    // build dependencies, but only before a full graph resolution.
    // Configurations should not be mutated in this hook
    incoming.beforeResolve {
        // Add a dependency on `com:foo` if not already present
        if (allDependencies.none { it.group == "com" && it.name == "foo" }) {

            configurations.implementation.get().dependencies.add(project.dependencies.create("com:foo:1.0"))
        }
    }
}

tasks.register("resolve") {
    val conf: FileCollection = configurations["runtimeClasspath"]

    // Wire build dependencies
    dependsOn(conf)

    // Resolve dependencies
    doLast {
        assert(conf.files.map { it.name } == listOf("foo-1.0.jar"))
    }
}
```

For the following use cases, consider these alternatives when replacing a `beforeResolve` hook:

- **Adding dependencies:** Use a `DependencyFactory` and `addLater` or `addAllLater` on `DependencySet`.
- **Changing dependency versions:** Use `preferred version constraints`.
- **Adding excludes:** Use `Component Metadata Rules` to adjust dependency-level excludes, or `withDependencies` to add excludes to a configuration.
- **Roles:** Configuration roles should be set upon creation and not changed afterward.
- **Hierarchy:** Configuration hierarchy (`extendsFrom`) should be set upon creation. Mutating the hierarchy prior to resolution is highly discouraged but permitted within a `withDependencies` hook.

- **Resolution Strategy:** Mutating a configuration's `ResolutionStrategy` is still permitted in a `beforeResolve` hook; however, this is not recommended.

Filtered Configuration `file` and `fileCollection` methods are deprecated

In an ongoing effort to simplify the Gradle API, the following methods that support filtering based on declared dependencies have been deprecated:

On `Configuration`:

- `files(Dependency...)`
- `files(Spec)`
- `files(Closure)`
- `fileCollection(Dependency...)`
- `fileCollection(Spec)`
- `fileCollection(Closure)`

On `ResolvedConfiguration`:

- `getFiles(Spec)`
- `getFirstLevelModuleDependencies(Spec)`

On `LenientConfiguration`:

- `getFirstLevelModuleDependencies(Spec)`
- `getFiles(Spec)`
- `getArtifacts(Spec)`

To mitigate this deprecation, consider the example below that leverages the `ArtifactView` API along with the `componentFilter` method to select a subset of a Configuration's artifacts:

build.gradle.kts

```
val conf by configurations.createing

dependencies {
    conf("com.thing:foo:1.0")
    conf("org.example:bar:1.0")
}

tasks.register("filterDependencies") {
    val files: FileCollection = conf.incoming.artifactView {
        componentFilter {
            when(it) {
                is ModuleComponentIdentifier ->
                    it.group == "com.thing" && it.module == "foo"
            }
        }
    }.files
}
```

```

        else -> false
    }
}
}.files

doLast {
    assert(files.map { it.name } == listOf("foo-1.0.jar"))
}
}

```

build.gradle

```

configurations {
    conf
}

dependencies {
    conf "com.thing:foo:1.0"
    conf "org.example:bar:1.0"
}

tasks.register("filterDependencies") {
    FileCollection files = configurations.conf.incoming.artifactView {
        componentFilter {
            it instanceof ModuleComponentIdentifier
            && it.group == "com.thing"
            && it.module == "foo"
        }
    }.files

    doLast {
        assert files*.name == ["foo-1.0.jar"]
    }
}

```

Contrary to the deprecated **Dependency** filtering methods, **componentFilter** does not consider the transitive dependencies of the component being filtered. This allows for more granular control over which artifacts are selected.

Deprecated Namer of Task and Configuration

Task and **Configuration** have a **Namer** inner class (also called **Namer**) that can be used as a common way to retrieve the name of a task or configuration. Now that these types implement **Named**, these classes are no longer necessary and have been deprecated. They will be removed in Gradle 9.0. Use **Named.Namer.INSTANCE** instead.

The super interface, `Namer`, is **not** being deprecated.

Unix mode-based file permissions deprecated

A new API for defining file permissions has been added in Gradle 8.3, see:

- [FilePermissions](#).
- [ConfigurableFilePermissions](#).

The new API has now been promoted to stable, and the old methods have been deprecated:

- `CopyProcessingSpec.getFileMode`
- `CopyProcessingSpec.setFileMode`
- `CopyProcessingSpec.getDirMode`
- `CopyProcessingSpec.setDirMode`
- `FileTreeElement.getMode`
- `FileCopyDetails.setMode`

Deprecated setting retention period directly on local build cache

In previous versions, cleanup of the local build cache entries ran every 24 hours, and this interval could not be configured. The retention period was configured using `buildCache.local.removeUnusedEntriesAfterDays`.

In Gradle 8.0, a [new mechanism](#) was added to configure the cleanup and retention periods for various resources in Gradle User Home. In Gradle 8.8, this mechanism was extended to permit the retention configuration of local build cache entries, providing improved control and consistency.

- Specifying `Cleanup.DISABLED` or `Cleanup.ALWAYS` will now prevent or force the cleanup of the local build cache
- Build cache entry retention is now configured via an `init-script`, [in the same manner as other caches](#).

If you want build cache entries to be retained for 30 days, **remove** any calls to the deprecated method:

```
buildCache {
    local {
        // Remove this line
        removeUnusedEntriesAfterDays = 30
    }
}
```

Add a file like this in `~/.gradle/init.d`:

```
beforeSettings {
```

```
    caches {  
        buildCache.setRemoveUnusedEntriesAfterDays(30)  
    }  
}
```

Calling `buildCache.local.removeUnusedEntriesAfterDays` is deprecated, and this method will be removed in Gradle 9.0. If set to a non-default value, this deprecated setting will take precedence over `Settings.caches.buildCache.setRemoveUnusedEntriesAfterDays()`.

Deprecated Kotlin DSL `gradle-enterprise` plugin block extension

In `settings.gradle.kts` (Kotlin DSL), you can use `gradle-enterprise` in the plugins block to apply the Gradle Enterprise plugin with the same version as `gradle --scan`.

```
plugins {  
    'gradle-enterprise'  
}
```

There is no equivalent to this in `settings.gradle` (Groovy DSL).

Gradle Enterprise has been renamed Develocity, and the `com.gradle.enterprise` plugin has been renamed `com.gradle.develocity`. Therefore, the `gradle-enterprise` plugin block extension has been deprecated and will be removed in Gradle 9.0.

The Develocity plugin must be applied with an explicit plugin ID and version. There is no `develocity` shorthand available in the plugins block:

```
plugins {  
    id("com.gradle.develocity") version "3.17.3"  
}
```

If you want to continue using the Gradle Enterprise plugin, you can specify the deprecated plugin ID:

```
plugins {  
    id("com.gradle.enterprise") version "3.17.3"  
}
```

We encourage you to use the [latest released Develocity plugin version](#), even when using an older Gradle version.

Potential breaking changes

Changes in the Problems API

We have implemented several refactorings of the Problems API, including a significant change in how problem definitions and contextual information are handled. The complete design

specification can be found [here](#).

In implementing this spec, we have introduced the following breaking changes to the `ProblemSpec` interface:

- The `label(String)` and `description(String)` methods have been replaced with the `id(String, String)` method and its overloaded variants.

Changes to collection properties

The following incubating API introduced in 8.7 have been removed:

- `MapProperty.insert*(...)`
- `HasMultipleValues.append*(...)`

Replacements that better handle conventions are under consideration for a future 8.x release.

Upgrade to Groovy 3.0.21

Groovy has been updated to [Groovy 3.0.21](#).

Since the previous version was 3.0.17, the [3.0.18](#) and [3.0.19](#), and [3.0.20](#) changes are also included.

Some changes in static type checking have resulted in source-code incompatibilities. Starting with 3.0.18, if you cast a closure to an `Action` without generics, the closure parameter will be `Object` instead of any explicit type specified. This can be fixed by adding the appropriate type to the cast, and the redundant parameter declaration can be removed:

```
// Before
tasks.create("foo", { Task it -> it.description = "Foo task" } as Action)

// Fixed
tasks.create("foo", { it.description = "Foo task" } as Action<Task>)
```

Upgrade to ASM 9.7

ASM was upgraded from 9.6 to [9.7](#) to ensure earlier compatibility for Java 23.

Upgrading from 8.6 and earlier

Potential breaking changes

Upgrade to Kotlin 1.9.22

The embedded Kotlin has been updated from 1.9.10 to [Kotlin 1.9.22](#).

Upgrade to Apache SSHD 2.10.0

Apache SSHD has been updated from 2.0.0 to [2.10.0](#).

Replacement and upgrade of JSch

JSch has been replaced by `com.github.mwiede:jsch` and updated from 0.1.55 to [0.2.16](#)

Upgrade to Eclipse JGit 5.13.3

Eclipse JGit has been updated from 5.7.0 to [5.13.3](#).

This includes reworking the way that Gradle configures JGit for SSH operations by moving from JSch to Apache SSHD.

Upgrade to Apache Commons Compress 1.25.0

Apache Commons Compress has been updated from 1.21 to [1.25.0](#). This change may affect the checksums of the produced jars, zips, and other archive types because the metadata of the produced artifacts may differ.

Upgrade to ASM 9.6

ASM was upgraded from 9.5 to [9.6](#) for better support of multi-release jars.

Upgrade of the version catalog parser

The version catalog parser has been upgraded and is now compliant with [version 1.0.0 of the TOML spec](#).

This should not impact catalogs that use the [recommended syntax](#) or were generated by Gradle for publication.

Deprecations

Deprecated registration of plugin conventions

Using plugin conventions has been emitting warnings since Gradle 8.2. Now, registering plugin conventions will also trigger deprecation warnings. For more information, see the [section about plugin convention deprecation](#).

Referencing tasks and domain objects by `"name"()` in Kotlin DSL

In Kotlin DSL, it is possible to reference a task or other domain object by its name using the `"name"()` notation.

There are several ways to look up an element in a container by name:

```
tasks {
    "wrapper"() // 1 - returns TaskProvider<Task>
    "wrapper"(Wrapper::class) // 2 - returns TaskProvider<Wrapper>
    "wrapper"(Wrapper::class) { // 3 - configures a task named wrapper of type Wrapper
    }
    "wrapper" { // 4 - configures a task named wrapper of type Task
    }
```

```
}
```

The first notation is deprecated and will be removed in Gradle 9.0. Instead of using `"name"()` to reference a task or domain object, use `named("name")` or one of the other supported notations.

The above example would be written as:

```
tasks {  
    named("wrapper") // returns TaskProvider<Task>  
}
```

The Gradle API and Groovy build scripts are not impacted by this.

Deprecated invalid URL decoding behavior

Before Gradle 8.3, Gradle would decode a `CharSequence` given to `Project.uri(Object)` using an algorithm that accepted invalid URLs and improperly decoded others. Gradle now uses the `URI` class to parse and decode URLs, but with a fallback to the legacy behavior in the event of an error.

Starting in Gradle 9.0, the fallback will be removed, and an error will be thrown instead.

To fix a deprecation warning, invalid URLs that require the legacy behavior should be re-encoded to be valid URLs, such as in the following examples:

Table 4. Legacy URL Conversions

Original Input	New Input	Reasoning
<code>file:relative/path</code>	<code>relative/path</code>	The <code>file</code> scheme does not support relative paths.
<code>file:relative/path%21</code>	<code>relative/path!</code>	Without a scheme, the path is taken as-is, without decoding.
<code>https://example.com/my folder/</code>	<code>https://example.com/my%20folder/</code>	Spaces are not valid in URLs.
<code>https://example.com/my%badly%encoded%path</code>	<code>https://example.com/my%25%25badly%25encoded%25path</code>	<code>%</code> must be encoded as <code>%25</code> in URLs, and no <code>%</code> -escapes should be invalid.
<code>file::somepath</code>	<code>somepath</code>	URIs should be hierarchical.

Deprecated `SelfResolvingDependency`

The `SelfResolvingDependency` interface has been deprecated for removal in Gradle 9.0. This type dates back to the first versions of Gradle, where some dependencies could be resolved independently. Now, all dependencies should be resolved as part of a dependency graph using a `Configuration`.

Currently, `ProjectDependency` and `FileCollectionDependency` implement this interface. In Gradle 9.0, these types will no longer implement `SelfResolvingDependency`. Instead, they will both directly

implement **Dependency**.

As such, the following methods of **ProjectDependency** and **FileCollectionDependency** will no longer be available:

- **resolve**
- **resolve(boolean)**
- **getBuildDependencies**

Consider the following scripts that showcase the deprecated interface and its replacement:

build.gradle.kts

```
plugins {
    id("java-library")
}

dependencies {
    implementation(files("bar.txt"))
    implementation(project(":foo"))
}

tasks.register("resolveDeprecated") {
    // Wire build dependencies (calls getBuildDependencies)
    dependsOn(configurations["implementation"].dependencies.toSet())

    // Resolve dependencies
    doLast {

        configurations["implementation"].dependencies.withType<FileCollectionDependency>() {
            assert(resolve().map { it.name } == listOf("bar.txt"))
            assert(resolve(true).map { it.name } == listOf("bar.txt"))
        }

        configurations["implementation"].dependencies.withType<ProjectDependency>() {
            // These methods do not even work properly.
            assert(resolve().map { it.name } == listOf<String>())
            assert(resolve(true).map { it.name } == listOf<String>())
        }
    }
}

tasks.register("resolveReplacement") {
    val conf = configurations["runtimeClasspath"]

    // Wire build dependencies
    dependsOn(conf)
```

```
// Resolve dependencies
val files = conf.files
doLast {
    assert(files.map { it.name } == listOf("bar.txt", "foo.jar"))
}
}
```

Deprecated members of the `org.gradle.util` package now report their deprecation

These members will be removed in Gradle 9.0.

- `Collection.stringize(Collection)`

Upgrading from 8.5 and earlier

Potential breaking changes

Upgrade to JaCoCo 0.8.11

JaCoCo has been updated to [0.8.11](#).

`DependencyAdder` renamed to `DependencyCollector`

The incubating `DependencyAdder` interface has been renamed to `DependencyCollector`. A `getDependencies` method has been added to the interface that returns all declared dependencies.

Deprecations

Deprecated calling `registerFeature` using the `main` source set

Calling `registerFeature` on the `java` extension using the `main` source set is deprecated and will change behavior in Gradle 9.0.

Currently, features created while calling `usingSourceSet` with the `main` source set are initialized differently than features created while calling `usingSourceSet` with any other source set. Previously, when using the `main` source set, new `implementation`, `compileOnly`, `runtimeOnly`, `api`, and `compileOnlyApi` configurations were created, and the compile and runtime classpaths of the `main` source set were configured to extend these configurations.

Starting in Gradle 9.0, the `main` source set will be treated like any other source set. With the `java-library` plugin applied (or any other plugin that applies the `java` plugin), calling `usingSourceSet` with the `main` source set will throw an exception. This is because the `java` plugin already configures a `main` feature. Only if the `java` plugin is not applied will the `main` source set be permitted when calling `usingSourceSet`.

Code that currently registers features with the main source set, such as:

build.gradle.kts

```
plugins {  
    id("java-library")  
}  
  
java {  
    registerFeature("feature") {  
        usingSourceSet(sourceSets["main"])  
    }  
}
```

build.gradle

```
plugins {  
    id("java-library")  
}  
  
java {  
    registerFeature("feature") {  
        usingSourceSet(sourceSets.main)  
    }  
}
```

Should instead, create a separate source set for the feature and register the feature with that source set:

build.gradle.kts

```
plugins {  
    id("java-library")  
}  
  
sourceSets {  
    create("feature")  
}  
  
java {  
    registerFeature("feature") {  
        usingSourceSet(sourceSets["feature"])  
    }  
}
```

build.gradle

```
plugins {  
    id("java-library")  
}  
  
sourceSets {  
    feature  
}  
  
java {  
    registerFeature("feature") {  
        usingSourceSet(sourceSets.feature)  
    }  
}
```

Deprecated publishing artifact dependencies with explicit name to Maven repositories

Publishing dependencies with an explicit artifact with a name different from the dependency's `artifactId` to Maven repositories has been deprecated. This behavior is still permitted when publishing to Ivy repositories. It will result in an error in Gradle 9.0.

When publishing to Maven repositories, Gradle will interpret the dependency below as if it were declared with coordinates `org:notfoo:1.0`:

build.gradle.kts

```
dependencies {  
    implementation("org:foo:1.0") {  
        artifact {  
            name = "notfoo"  
        }  
    }  
}
```

build.gradle

```
dependencies {  
    implementation("org:foo:1.0") {  
        artifact {  
            name = "notfoo"  
        }  
    }  
}
```

```
}
```

Instead, this dependency should be declared as:

build.gradle.kts

```
dependencies {  
    implementation("org:notfoo:1.0")  
}
```

build.gradle

```
dependencies {  
    implementation("org:notfoo:1.0")  
}
```

Deprecated `ArtifactIdentifier`

The `ArtifactIdentifier` class has been deprecated for removal in Gradle 9.0.

Deprecate mutating `DependencyCollector` dependencies after observation

Starting in Gradle 9.0, mutating dependencies sourced from a `DependencyCollector`, after those dependencies have been observed will result in an error. The `DependencyCollector` interface is used to declare dependencies within the test suites DSL.

Consider the following example where a test suite's dependency is mutated after it is observed:

build.gradle.kts

```
plugins {  
    id("java-library")  
}  
  
testing.suites {  
    named<JvmTestSuite>("test") {  
        dependencies {  
            // Dependency is declared on a `DependencyCollector`  
            implementation("com:foo")  
        }  
    }  
}
```

```

configurations.testImplementation {
    // Calling `all` here realizes/observes all lazy sources, including the
    `DependencyCollector`
    // from the test suite block. Operations like resolving a configuration
    similarly realize lazy sources.
    dependencies.all {
        if (this is ExternalDependency && group == "com" && name == "foo" &&
version == null) {
            // Dependency is mutated after observation
            version {
                require("2.0")
            }
        }
    }
}

```

In the above example, the build logic uses iteration and mutation to try to set a default version for a particular dependency if the version is not already set. Build logic like the above example creates challenges in resolving declared dependencies, as reporting tools will display this dependency as if the user declared the version as "2.0", even though they never did. Instead, the build logic can avoid iteration and mutation by declaring a **preferred** version constraint on the dependency's coordinates. This allows the dependency management engine to use the version declared on the constraint if no other version is declared.

Consider the following example that replaces the above iteration with an indiscriminate **preferred** version constraint:

build.gradle.kts

```

dependencies {
    constraints {
        testImplementation("com:foo") {
            version {
                prefer("2.0")
            }
        }
    }
}

```

Upgrading from 8.4 and earlier

Potential breaking changes

Upgrade to Kotlin 1.9.20

The embedded Kotlin has been updated to [Kotlin 1.9.20](#).

Changes to Groovy task conventions

The `groovy-base` plugin is now responsible for configuring source and target compatibility version conventions on all `GroovyCompile` tasks.

If you are using this task **without applying `groovy-base`**, you will have to manually set compatibility versions on these tasks. In general, the `groovy-base` plugin should be applied whenever working with Groovy language tasks.

`Provider.filter`

The type of argument passed to `Provider.filter` is changed from `Predicate` to `Spec` for a more consistent API. This change should not affect anyone using `Provider.filter` with a lambda expression. However, this might affect plugin authors if they don't use SAM conversions to create a lambda.

Deprecations

Deprecated members of the `org.gradle.util` package now report their deprecation

These members will be removed in Gradle 9.0:

- `VersionNumber.parse(String)`
- `VersionNumber.compareTo(VersionNumber)`

Deprecated depending on resolved configuration

When resolving a `Configuration`, selecting that same configuration as a variant is sometimes possible. Configurations should be used for one purpose (resolution, consumption or dependency declarations), so this can only occur when a configuration is marked as both consumable and resolvable.

This can lead to circular dependency graphs, as the resolved configuration is used for two purposes.

To avoid this problem, plugins should mark all resolvable configurations as `canBeConsumed=false` or use the `resolvable(String)` configuration factory method when creating configurations meant for resolution.

In Gradle 9.0, consuming configurations in this manner will no longer be allowed and result in an error.

Including projects without an existing directory

Gradle will warn if a project is added to the build where the associated `projectDir` does not exist or is not writable. Starting with version 9.0, Gradle will not run builds if a project directory is missing or read-only. If you intend to dynamically synthesize projects, make sure to create directories for

them as well:

settings.gradle.kts

```
include("project-without-directory")
project(":project-without-directory").projectDir.mkdirs()
```

settings.gradle

```
include 'project-without-directory'
project(":project-without-directory").projectDir.mkdirs()
```

Upgrading from 8.3 and earlier

Potential breaking changes

Upgrade to Kotlin 1.9.10

The embedded Kotlin has been updated to [Kotlin 1.9.10](#).

XML parsing now requires recent parsers

Gradle 8.4 now configures XML parsers with security features enabled. If your build logic depends on old XML parsers that don't support secure parsing, your build may fail. If you encounter a failure, check and update or remove any dependency on legacy XML parsers.

If you are an Android user, please upgrade your AGP version to 8.3.0 or higher to fix the issue caused by AGP itself. See the [Update XML parser used in AGP for Gradle 8.4 compatibility](#) for more details.

If you are unable to upgrade XML parsers coming from your build logic dependencies, you can force the use of the XML parsers built into the JVM. In OpenJDK, for example, this can be done by adding the following to **gradle.properties**:

```
systemProp.java.xml.parsers.SAXParserFactory=com.sun.org.apache.xerces.internal.jaxp.SAXParserFactoryImpl
systemProp.java.xml.transform.TransformerFactory=com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl
systemProp.java.xml.parsers.DocumentBuilderFactory=com.sun.org.apache.xerces.internal.jaxp.DocumentBuilderFactoryImpl
```

See the [CVE-2023-42445](#) advisory for more details and ways to enable secure XML processing on previous Gradle versions.

EAR plugin with customized JEE 1.3 descriptor

Gradle 8.4 forbids external XML entities when parsing XML documents. If you use the EAR plugin and configure the `application.xml` descriptor via the EAR plugin's DSL and customize the descriptor using `withXml {}` and use `asElement()` in the customization block, then the build will now fail for security reasons.

build.gradle.kts

```
plugins {
    id("ear")
}
ear {
    deploymentDescriptor {
        version = "1.3"
        withXml {
            asElement()
        }
    }
}
```

build.gradle

```
plugins {
    id("ear")
}
ear {
    deploymentDescriptor {
        version = "1.3"
        withXml {
            asElement()
        }
    }
}
```

If you happen to use `asNode()` instead of `asElement()`, then nothing changes, given `asNode()` simply ignores external DTDs.

You can work around this by running your build with the `javax.xml.accessExternalDTD` system property set to `http`.

On the command line, add this to your Gradle invocation:

```
-Djavax.xml.accessExternalDTD=http
```

To make this workaround persistent, add the following line to your `gradle.properties`:

```
systemProp.java.xml.accessExternalDTD=http
```

Note that this will enable HTTP access to external DTDs for the whole build JVM. See the [JAXP documentation](#) for more details.

Deprecations

Deprecated `GenerateMavenPom` methods

The following methods on `GenerateMavenPom` are deprecated and will be removed in Gradle 9.0. They were never intended to be public API.

- `getVersionRangeMapper`
- `withCompileScopeAttributes`
- `withRuntimeScopeAttributes`

Upgrading from 8.2 and earlier

Potential breaking changes

Deprecated `Project.buildDir` can cause script compilation failure

With the deprecation of `Project.buildDir`, buildscripts that are compiled with warnings as errors could fail if the deprecated field is used.

See [the deprecation entry](#) for details.

`TestLauncher` API no longer ignores build failures

The `TestLauncher` interface is part of the Tooling API, specialized for running tests. It is a logical extension of the `BuildLauncher` that can only launch tasks. A discrepancy has been reported in their behavior: if the same failing test is executed, `BuildLauncher` will report a build failure, but `TestLauncher` won't. Originally, this was a design decision in order to continue the execution and run the tests in all test tasks and not stop at the first failure. At the same time, this behavior can be confusing for users as they can experience a failing test in a successful build. To make the two APIs more uniform, we made `TestLauncher` also fail the build, which is a potential breaking change. Tooling API clients should explicitly pass `--continue` to the build to continue the test execution even if a test task fails.

Fixed variant selection behavior with `ArtifactView` and `ArtifactCollection`

The dependency resolution APIs for selecting different artifacts or files (`Configuration.getIncoming().artifactView { }` and `Configuration.getIncoming().getArtifacts()`) captured immutable copies of the underlying `Configuration`'s attributes to use for variant selection. If the `Configuration`'s attributes were changed after these methods were called, the artifacts selected by these methods could be unexpected.

Consider the case where the set of attributes on a **Configuration** is changed after an **ArtifactView** is created:

build.gradle.kts

```
tasks {
    myTask {
        inputFiles.from(configurations.classpath.incoming.artifactView {
            attributes {
                // Add attributes to select a different type of artifact
            }
        }.files)
    }
}

configurations {
    classpath {
        attributes {
            // Add more attributes to the configuration
        }
    }
}
```

The **inputFiles** property of **myTask** uses an artifact view to select a different type of artifact from the configuration **classpath**. Since the artifact view was created before the attributes were added to the configuration, Gradle could not select the correct artifact.

Some builds may have worked around this by also putting the additional attributes into the artifact view. This is no longer necessary.

Upgrade to Kotlin 1.9.0

The embedded Kotlin has been updated from 1.8.20 to [Kotlin 1.9.0](#). The Kotlin language and API levels for the Kotlin DSL are still set to 1.8 for backward compatibility. See the release notes for [Kotlin 1.8.22](#) and [Kotlin 1.8.21](#).

Kotlin 1.9 dropped support for Kotlin language and API level 1.3. If you build Gradle plugins written in Kotlin with this version of Gradle and need to support Gradle <7.0 you need to stick to using the Kotlin Gradle Plugin <1.9.0 and configure the Kotlin language and API levels to 1.3. See the [Compatibility Matrix](#) for details about other versions.

Eager evaluation of **Configuration** attributes

Gradle 8.3 updates the **org.gradle.libraryelements** and **org.gradle.jvm.version** attributes of JVM Configurations to be present at the time of creation, as opposed to previously, where they were only present after the Configuration had been resolved or consumed. In particular, the value for **org.gradle.jvm.version** relies on the project's configured toolchain, meaning that querying the

value for this attribute will finalize the value of the project's Java toolchain.

Plugins or build logic that eagerly queries the attributes of JVM configurations may now cause the project's Java toolchain to be finalized earlier than before. Attempting to modify the toolchain after it has been finalized will result in error messages similar to the following:

```
The value for property 'implementation' is final and cannot be changed any further.  
The value for property 'languageVersion' is final and cannot be changed any further.  
The value for property 'vendor' is final and cannot be changed any further.
```

This situation may arise when plugins or build logic eagerly query an existing JVM Configuration's attributes to create a new Configuration with the same attributes. Previously, this logic would have omitted the two above-noted attributes entirely, while now, the same logic will copy the attributes and finalize the project's Java toolchain. To avoid early toolchain finalization, attribute-copying logic should be updated to query the source Configuration's attributes lazily:

build.gradle.kts

```
fun <T> copyAttribute(attribute: Attribute<T>, from: AttributeContainer, to:  
AttributeContainer) =  
    to.attributeProvider<T>(attribute, provider {  
        from.getAttribute(attribute)!! })  
  
val source = configurations["runtimeClasspath"].attributes  
configurations {  
    create("customRuntimeClasspath") {  
        source.keySet().forEach { key ->  
            copyAttribute(key, source, attributes)  
        }  
    }  
}
```

build.gradle

```
def source = configurations.runtimeClasspath.attributes  
configurations {  
    customRuntimeClasspath {  
        source.keySet().each { key ->  
            attributes.attributeProvider(key, provider { source.getAttribute  
(key) })  
        }  
    }  
}
```

Deprecations

Deprecated `Project.buildDir` is to be replaced by `Project.layout.buildDirectory`

The `Project.buildDir` property is deprecated. It uses eager APIs and has ordering issues if the value is read in build logic and then later modified. It could result in outputs ending up in different locations.

It is replaced by a `DirectoryProperty` found at `Project.layout.buildDirectory`. See the `ProjectLayout` interface for details.

Note that, at this stage, Gradle will not print deprecation warnings if you still use `Project.buildDir`. We know this is a big change, and we want to give the authors of major plugins time to stop using it.

Switching from a `File` to a `DirectoryProperty` requires adaptations in build logic. The main impact is that you cannot use the property inside a `String` to expand it. Instead, you should leverage the `dir` and `file` methods to compute your desired location.

Here is an example of creating a file where the following:

build.gradle.kts

```
// Returns a java.io.File
file("$buildDir/myOutput.txt")
```

build.gradle

```
// Returns a java.io.File
file("$buildDir/myOutput.txt")
```

Should be replaced by:

build.gradle.kts

```
// Compatible with a number of Gradle lazy APIs that accept also java.io.File
val output: Provider<RegularFile> =
    layout.buildDirectory.file("myOutput.txt")

// If you really need the java.io.File for a non lazy API
output.get().asFile

// Or a path for a lazy String based API
output.map { it.asFile.path }
```

build.gradle

```
// Compatible with a number of Gradle lazy APIs that accept also java.io.File
Provider<RegularFile> output = layout.buildDirectory.file("myOutput.txt")

// If you really need the java.io.File for a non lazy API
output.get().asFile

// Or a path for a lazy String based API
output.map { it.asFile.path }
```

Here is another example for creating a directory where the following:

build.gradle.kts

```
// Returns a java.io.File
file("$buildDir/outputLocation")
```

build.gradle

```
// Returns a java.io.File
file("$buildDir/outputLocation")
```

Should be replaced by:

build.gradle.kts

```
// Compatible with a number of Gradle APIs that accept a java.io.File
val output: Provider<Directory> = layout.buildDirectory.dir("outputLocation")

// If you really need the java.io.File for a non lazy API
output.get().asFile

// Or a path for a lazy String based API
output.map { it.asFile.path }
```

build.gradle

```
// Compatible with a number of Gradle APIs that accept a java.io.File
Provider<Directory> output = layout.buildDirectory.dir("outputLocation")

// If you really need the java.io.File for a non lazy API
output.get().asFile

// Or a path for a lazy String based API
output.map { it.asFile.path }
```

Deprecated `ClientModule` dependencies

`ClientModule` dependencies are deprecated and will be removed in Gradle 9.0.

Client module dependencies were originally intended to allow builds to override incorrect or missing component metadata of external dependencies by defining the metadata locally. This functionality has since been replaced by [Component Metadata Rules](#).

Consider the following client module dependency example:

build.gradle.kts

```
dependencies {
    implementation(module("org:foo:1.0") {
        dependency("org:bar:1.0")
        module("org:baz:1.0") {
            dependency("com:example:1.0")
        }
    })
}
```

build.gradle

```
dependencies {
    implementation module("org:foo:1.0") {
        dependency "org:bar:1.0"
        module("org:baz:1.0") {
            dependency "com:example:1.0"
        }
    }
}
```

This can be replaced with the following component metadata rule:

build-logic/src/main/kotlin/my-plugin.gradle.kts

```
@CacheableRule
abstract class AddDependenciesRule @Inject constructor(val dependencies:
List<String>) : ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        listOf("compile", "runtime").forEach { base ->
            context.details.withVariant(base) {
                withDependencies {
                    dependencies.forEach {
                        add(it)
                    }
                }
            }
        }
    }
}
```

build.gradle.kts

```
dependencies {
    components {
        withModule<AddDependenciesRule>("org:foo") {
            params(listOf(
                "org:bar:1.0",
                "org:baz:1.0"
            ))
        }
        withModule<AddDependenciesRule>("org:baz") {
            params(listOf("com:example:1.0"))
        }
    }

    implementation("org:foo:1.0")
}
```

build-logic/src/main/groovy/my-plugin.gradle

```
@CacheableRule
abstract class AddDependenciesRule implements ComponentMetadataRule {

    List<String> dependencies

    @Inject
    AddDependenciesRule(List<String> dependencies) {
```

```

        this.dependencies = dependencies
    }

    @Override
    void execute(ComponentMetadataContext context) {
        ["compile", "runtime"].each { base ->
            context.details.withVariant(base) {
                withDependencies {
                    dependencies.each {
                        add(it)
                    }
                }
            }
        }
    }
}

```

build.gradle

```

dependencies {
    components {
        withModule("org:foo", AddDependenciesRule) {
            params([
                "org:bar:1.0",
                "org:baz:1.0"
            ])
        }
        withModule("org:baz", AddDependenciesRule) {
            params(["com:example:1.0"])
        }
    }

    implementation "org:foo:1.0"
}

```

Earliest supported Develocity plugin version is 3.13.1

Starting in Gradle 9.0, the earliest supported Develocity plugin version is 3.13.1. The plugin versions from 3.0 up to 3.13 will be ignored when applied.

Upgrade to version 3.13.1 or later of the Develocity plugin. You can find the [latest available version on the Gradle Plugin Portal](#). More information on the compatibility can be found [here](#).

Upgrading from 8.1 and earlier

Potential breaking changes

Upgrade to Kotlin 1.8.20

The embedded Kotlin has been updated to [Kotlin 1.8.20](#). For more information, see [What's new in Kotlin 1.8.20](#).

Note that there is a known issue with Kotlin compilation avoidance that can cause `OutOfMemory` exceptions in `compileKotlin` tasks if the compilation classpath contains very large JAR files. This applies to builds applying the Kotlin plugin v1.8.20 or the `kotlin-dsl` plugin.

You can work around it by disabling Kotlin compilation avoidance in your `gradle.properties` file:

```
kotlin.incremental.useClasspathSnapshot=false
```

See [KT-57757](#) for more information.

Upgrade to Groovy 3.0.17

Groovy has been updated to [Groovy 3.0.17](#).

Since the previous version was 3.0.15, the [3.0.16](#) changes are also included.

Upgrade to Ant 1.10.13

Ant has been updated to [Ant 1.10.13](#).

Since the previous version was 1.10.11, the [1.10.12](#) changes are also included.

Upgrade to CodeNarc 3.2.0

The default version of CodeNarc has been updated to [CodeNarc 3.2.0](#).

Upgrade to PMD 6.55.0

PMD has been updated to [PMD 6.55.0](#).

Since the previous version was 6.48.0, all changes since then are included.

Upgrade to JaCoCo 0.8.9

JaCoCo has been updated to [0.8.9](#).

Plugin compatibility changes

A plugin compiled with Gradle ≥ 8.2 that makes use of the Kotlin DSL functions `Project.the<T>()`, `Project.the(KClass)` or `Project.configure<T> {}` cannot run on Gradle ≤ 6.1 .

Deferred or avoided configuration of some tasks

When performing dependency resolution, Gradle creates an internal representation of the available [Configurations](#). This requires inspecting all configurations and artifacts. Processing artifacts created by tasks causes those tasks to be realized and configured.

This internal representation is now created more lazily, which can change the order in which tasks are configured. Some tasks may never be configured.

This change may cause code paths that relied on a particular order to no longer function, such as conditionally adding attributes to a configuration based on the presence of certain attributes.

This impacted the [bnd plugin](#) and [JUnit5 build](#).

We recommend not modifying domain objects (configurations, source sets, tasks, etc) from configuration blocks for other domain objects that may not be configured.

For example, avoid doing something like this:

```
configurations {
    val myConfig = create("myConfig")
}

tasks.register("myTask") {
    // This is not safe, as the execution of this block may not occur, or may
    not occur in the order expected
    configurations["myConfig"].attributes {
        attribute(Usage.USAGE_ATTRIBUTE, objects.named(Usage::class.java,
Usage.JAVA_RUNTIME))
    }
}
```

Deprecations

[CompileOptions](#) method deprecations

The following methods on [CompileOptions](#) are deprecated:

- [getAnnotationProcessorGeneratedSourcesDirectory\(\)](#)
- [setAnnotationProcessorGeneratedSourcesDirectory\(File\)](#)
- [setAnnotationProcessorGeneratedSourcesDirectory\(Provider<File>\)](#)

Current usages of these methods should migrate to [DirectoryProperty](#) [getGeneratedSourceOutputDirectory\(\)](#)

Using configurations incorrectly

Gradle will now warn at runtime when methods of [Configuration](#) are called inconsistently with the configuration's intended usage.

This change is part of a larger ongoing effort to make the intended behavior of configurations more consistent and predictable and to unlock further speed and memory improvements.

Currently, the following methods should only be called with these listed allowed usages:

- [resolve\(\)](#) - RESOLVABLE configurations only

- `files(Closure)`, `files(Spec)`, `files(Dependency...)`, `fileCollection(Spec)`, `fileCollection(Closure)`, `fileCollection(Dependency...)` - RESOLVABLE configurations only
- `getResolvedConfigurations()` - RESOLVABLE configurations only
- `defaultDependencies(Action)` - DECLARABLE configurations only
- `shouldResolveConsistentlyWith(Configuration)` - RESOLVABLE configurations only
- `disableConsistentResolution()` - RESOLVABLE configurations only
- `getDependencyConstraints()` - DECLARABLE configurations only
- `copy()`, `copy(Spec)`, `copy(Closure)`, `copyRecursive()`, `copyRecursive(Spec)`, `copyRecursive(Closure)` - RESOLVABLE configurations only

Intended usage is noted in the `Configuration` interface's Javadoc. This list is likely to grow in future releases.

Starting in Gradle 9.0, using a configuration inconsistently with its intended usage will be prohibited.

Also note that although it is not currently restricted, the `getDependencies()` method is only intended for use with DECLARABLE configurations. The `getAllDependencies()` method, which retrieves all declared dependencies on a configuration and any superconfigurations, will not be restricted to any particular usage.

Deprecated access to plugin conventions

The concept of conventions is outdated and superseded by [extensions](#) to provide custom DSLs.

To reflect this in the Gradle API, the following elements are deprecated:

- `org.gradle.api.Project.getConvention()`
- `org.gradle.api.plugins.Convention`
- `org.gradle.api.internal.HasConvention`

Gradle Core plugins still register their conventions in addition to their extensions for backwards compatibility.

It is deprecated to access any of these conventions and their properties. Doing so will now emit a deprecation warning. This will become an error in Gradle 9.0. You should prefer accessing the extensions and their properties instead.

For specific examples, see the next sections.

Prominent community plugins already migrated to using extensions to provide custom DSLs. Some of them still register conventions for backward compatibility. Registering conventions does not emit a deprecation warning yet to provide a migration window. Future Gradle versions will do.

Also note that Plugins compiled with Gradle \leq 8.1 that make use of the Kotlin DSL functions `Project.the<T>()`, `Project.the(KClass)` or `Project.configure<T> {}` will emit a deprecation warning when run on Gradle \geq 8.2. To fix this these plugins should be recompiled with Gradle \geq 8.2 or

changed to access extensions directly using `extensions.getByType<T>()` instead.

Deprecated **base** plugin conventions

The convention properties contributed by the **base** plugin have been deprecated and scheduled for removal in Gradle 9.0. For more context, see the [section about plugin convention deprecation](#).

The conventions are replaced by the **base { }** configuration block backed by [BasePluginExtension](#). The old convention object defines the `distsDirName`, `libsDirName`, and `archivesBaseName` properties with simple getter and setter methods. Those methods are available in the extension only to maintain backward compatibility. Build scripts should solely use the properties of type **Property**:

build.gradle.kts

```
plugins {
    base
}

base {
    archivesName.set("gradle")
    distsDirectory.set(layout.buildDirectory.dir("custom-dist"))
    libsDirectory.set(layout.buildDirectory.dir("custom-libs"))
}
```

build.gradle

```
plugins {
    id 'base'
}

base {
    archivesName = "gradle"
    distsDirectory = layout.buildDirectory.dir('custom-dist')
    libsDirectory = layout.buildDirectory.dir('custom-libs')
}
```

Deprecated **application** plugin conventions

The convention properties the **application** plugin contributed have been deprecated and scheduled for removal in Gradle 9.0. For more context, see the [section about plugin convention deprecation](#).

The following code will now emit deprecation warnings:

build.gradle.kts

```
plugins {  
    application  
}  
  
applicationDefaultJvmArgs = listOf("-Dgreeting.language=en") // Accessing a  
convention
```

build.gradle

```
plugins {  
    id 'application'  
}  
  
applicationDefaultJvmArgs = ['-Dgreeting.language=en'] // Accessing a  
convention
```

This should be changed to use the `application { }` configuration block, backed by [JavaApplication](#), instead:

build.gradle.kts

```
plugins {  
    application  
}  
  
application {  
    applicationDefaultJvmArgs = listOf("-Dgreeting.language=en")  
}
```

build.gradle

```
plugins {  
    id 'application'  
}  
  
application {  
    applicationDefaultJvmArgs = ['-Dgreeting.language=en']  
}
```

Deprecated `java` plugin conventions

The convention properties the `java` plugin contributed have been deprecated and scheduled for removal in Gradle 9.0. For more context, see the [section about plugin convention deprecation](#).

The following code will now emit deprecation warnings:

build.gradle.kts

```
plugins {  
    id("java")  
}  
  
configure<JavaPluginConvention> { // Accessing a convention  
    sourceCompatibility = JavaVersion.VERSION_18  
}
```

build.gradle

```
plugins {  
    id 'java'  
}  
  
sourceCompatibility = 18 // Accessing a convention
```

This should be changed to use the `java { }` configuration block, backed by [JavaPluginExtension](#), instead:

build.gradle.kts

```
plugins {  
    id("java")  
}  
  
java {  
    sourceCompatibility = JavaVersion.VERSION_18  
}
```

build.gradle

```
plugins {  
    id 'java'
```

```
}

java {
    sourceCompatibility = JavaVersion.VERSION_18
}
```

Deprecated **war** plugin conventions

The convention properties contributed by the **war** plugin have been deprecated and scheduled for removal in Gradle 9.0. For more context, see the [section about plugin convention deprecation](#).

The following code will now emit deprecation warnings:

build.gradle.kts

```
plugins {
    id("war")
}

configure<WarPluginConvention> { // Accessing a convention
    webAppDirName = "src/main/webapp"
}
```

build.gradle

```
plugins {
    id 'war'
}

webAppDirName = 'src/main/webapp' // Accessing a convention
```

Clients should configure the **war** task directly. Also, [tasks.withType\(War.class\).configureEach\(...\)](#) can be used to configure each task of type **War**.

build.gradle.kts

```
plugins {
    id("war")
}

tasks.war {
```

```
webAppDirectory.set(file("src/main/webapp"))
}
```

build.gradle

```
plugins {
    id 'war'
}

war {
    webAppDirectory = file('src/main/webapp')
}
```

Deprecated **ear** plugin conventions

The convention properties contributed by the **ear** plugin have been deprecated and scheduled for removal in Gradle 9.0. For more context, see the [section about plugin convention deprecation](#).

The following code will now emit deprecation warnings:

build.gradle.kts

```
plugins {
    id("ear")
}

configure<EarPluginConvention> { // Accessing a convention
    appDirName = "src/main/app"
}
```

build.gradle

```
plugins {
    id 'ear'
}

appDirName = 'src/main/app' // Accessing a convention
```

Clients should configure the **ear** task directly. Also, `tasks.withType(Ear.class).configureEach(...)` can be used to configure each task of type **Ear**.

build.gradle.kts

```
plugins {  
    id("ear")  
}  
  
tasks.ear {  
    appDirectory.set(file("src/main/app"))  
}
```

build.gradle

```
plugins {  
    id 'ear'  
}  
  
ear {  
    appDirectory = file('src/main/app') // use application metadata found in  
    this folder  
}
```

Deprecated **project-report** plugin conventions

The convention properties contributed by the **project-reports** plugin have been deprecated and scheduled for removal in Gradle 9.0. For more context, see the [section about plugin convention deprecation](#).

The following code will now emit deprecation warnings:

build.gradle.kts

```
plugins {  
    `project-report`  
}  
  
configure<ProjectReportsPluginConvention> {  
    projectReportDirName = "custom" // Accessing a convention  
}
```

build.gradle

```
plugins {  
    id 'project-report'  
}  
  
projectReportDirName = "custom" // Accessing a convention
```

Configure your report task instead:

build.gradle.kts

```
plugins {  
    `project-report`  
}  
  
tasks.withType<HtmlDependencyReportTask>() {  
  
    projectReportDirectory.set(project.layout.buildDirectory.dir("reports/custom"  
    ))  
}
```

build.gradle

```
plugins {  
    id 'project-report'  
}  
  
tasks.withType(HtmlDependencyReportTask) {  
    projectReportDirectory = project.layout.buildDirectory.dir(  
        "reports/custom"  
    )  
}
```

Configuration method deprecations

The following method on **Configuration** is deprecated for removal:

- **getAll()**

Obtain the set of all configurations from the project's **configurations** container instead.

Relying on automatic test framework implementation dependencies

In some cases, Gradle will load JVM test framework dependencies from the Gradle distribution to execute tests. This existing behavior can lead to test framework dependency version conflicts on the test classpath. To avoid these conflicts, this behavior is deprecated and will be removed in Gradle 9.0. Tests using TestNG are unaffected.

To prepare for this change in behavior, either declare the required dependencies explicitly or migrate to [Test Suites](#), where these dependencies are managed automatically.

Test Suites

Builds that use test suites will not be affected by this change. Test suites manage the test framework dependencies automatically and do not require dependencies to be explicitly declared. See [the user manual](#) for further information on migrating to test suites.

Manually declaring dependencies

In the absence of test suites, dependencies must be manually declared on the test runtime classpath:

- If using JUnit 5, an explicit `runtimeOnly` dependency on `junit-platform-launcher` is required in addition to the existing `implementation` dependency on the test engine.
- If using JUnit 4, only the existing `implementation` dependency on `junit 4` is required.
- If using JUnit 3, a test `runtimeOnly` dependency on `junit 4` is required in addition to a `compileOnly` dependency on `junit 3`.

build.gradle.kts

```
dependencies {  
    // If using JUnit Jupiter  
    testImplementation("org.junit.jupiter:junit-jupiter:5.9.2")  
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")  
  
    // If using JUnit Vintage  
    testCompileOnly("junit:junit:4.13.2")  
    testRuntimeOnly("org.junit.vintage:junit-vintage-engine:5.9.2")  
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")  
  
    // If using JUnit 4  
    testImplementation("junit:junit:4.13.2")  
  
    // If using JUnit 3  
    testCompileOnly("junit:junit:3.8.2")  
    testRuntimeOnly("junit:junit:4.13.2")  
}
```

build.gradle

```
dependencies {  
    // If using JUnit Jupiter  
    testImplementation 'org.junit.jupiter:junit-jupiter:5.9.2'  
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'  
  
    // If using JUnit Vintage  
    testCompileOnly 'junit:junit:4.13.2'  
    testRuntimeOnly 'org.junit.vintage:junit-vintage-engine:5.9.2'  
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'  
  
    // If using JUnit 4  
    testImplementation 'junit:junit:4.13.2'  
  
    // If using JUnit 3  
    testCompileOnly 'junit:junit:3.8.2'  
    testRuntimeOnly 'junit:junit:4.13.2'  
}
```

BuildIdentifier and ProjectComponentSelector method deprecations

The following methods on **BuildIdentifier** are deprecated:

- **getName()**
- **isCurrentBuild()**

You could use these methods to distinguish between different project components with the same name but from different builds. However, for certain composite build setups, these methods do not provide enough information to guarantee uniqueness.

Current usages of these methods should migrate to **BuildIdentifier.getBuildPath()**.

Similarly, the method **ProjectComponentSelector.getBuildName()** is deprecated. Use **ProjectComponentSelector.getBuildPath()** instead.

Upgrading from 8.0 and earlier

CACHEDIR.TAG files are created in global cache directories

Gradle now emits a **CACHEDIR.TAG** file in some global cache directories, as specified in [Cache marking](#).

This may cause these directories to no longer be searched or backed up by some tools. To disable it, use the following code in an [init script](#) in the Gradle User Home:

init.gradle.kts

```
beforeSettings {
    caches {
        // Disable cache marking for all caches
        markingStrategy.set(MarkingStrategy.NONE)
    }
}
```

init.gradle

```
beforeSettings { settings ->
    settings.caches {
        // Disable cache marking for all caches
        markingStrategy = MarkingStrategy.NONE
    }
}
```

Configuration cache options renamed

In this release, the configuration cache feature was promoted from incubating to stable. As such, all properties originally mentioned in the feature documentation (which had an **unsafe** part in their names, e.g., **org.gradle.unsafe.configuration-cache**) were renamed, in some cases, by removing the **unsafe** part of the name.

Incubating property	Finalized property
org.gradle.unsafe.configuration-cache	org.gradle.configuration-cache
org.gradle.unsafe.configuration-cache-problems	org.gradle.configuration-cache.problems*
org.gradle.unsafe.configuration-cache.max-problems	org.gradle.configuration-cache.max-problems

Note that the original **org.gradle.unsafe.configuration-cache...** properties continue to be honored in this release, and no warnings will be produced if they are used, but they will be deprecated and removed in a future release.

Potential breaking changes

Kotlin DSL scripts emit compilation warnings

Compilation warnings from Kotlin DSL scripts are printed to the console output. For example, the use of deprecated APIs in Kotlin DSL will emit warnings each time the script is compiled.

This is a potentially breaking change if you are consuming the console output of Gradle builds.

Configuring Kotlin compiler options with the `kotlin-dsl` plugin applied

If you are configuring custom Kotlin compiler options on a project with the `kotlin-dsl` plugin applied you might encounter a breaking change.

In previous Gradle versions, the `kotlin-dsl` plugin was adding required compiler arguments on `afterEvaluate {}`. Now that the Kotlin Gradle Plugin provides `lazy configuration` properties, our `kotlin-dsl` plugin switched to adding required compiler arguments to the lazy properties directly. As a consequence, if you were setting `freeCompilerArgs` the `kotlin-dsl` plugin is now failing the build because its required compiler arguments are overridden by your configuration.

build.gradle.kts

```
plugins {
    `kotlin-dsl`
}

tasks.withType(KotlinCompile::class).configureEach {
    kotlinOptions { // Deprecated non-lazy configuration options
        freeCompilerArgs = listOf("-Xcontext-receivers")
    }
}
```

With the configuration above you would get the following build failure:

```
* What went wrong
Execution failed for task ':compileKotlin'.
> Kotlin compiler arguments of task ':compileKotlin' do not work for the 'kotlin-dsl'
plugin. The 'freeCompilerArgs' property has been reassigned. It must instead be
appended to. Please use 'freeCompilerArgs.addAll("\your\","args\")' to fix this.
```

You must change this to adding your custom compiler arguments to the lazy configuration properties of the Kotlin Gradle Plugin for them to be appended to the ones required by the `kotlin-dsl` plugin:

build.gradle.kts

```
plugins {
    `kotlin-dsl`
}

tasks.withType(KotlinCompile::class).configureEach {
    compilerOptions { // New lazy configuration options
        freeCompilerArgs.addAll("-Xcontext-receivers")
    }
}
```

```
}
```

If you were already adding to `freeCompilerArgs` instead of setting its value, you should not experience a build failure.

New API introduced may clash with existing Gradle DSL code

When a new property or method is added to an existing type in the Gradle DSL, it may clash with names already used in user code.

When a name clash occurs, one solution is to rename the element in user code.

This is a non-exhaustive list of API additions in 8.1 that may cause name collisions with existing user code.

- `JavaExec.getJvmArguments()`
- `JavaExecSpec.getJvmArguments()`

Using unsupported API to start external processes at configuration time is no longer allowed with the configuration cache enabled

Since Gradle 7.5, using `Project.exec`, `Project.javaexec`, and standard Java and Groovy APIs to run external processes at configuration time has been considered an error only if the [feature preview STABLE_CONFIGURATION_CACHE](#) was enabled. With the configuration cache promotion to a stable feature in Gradle 8.1, this error is detected regardless of the feature preview status. The [configuration cache chapter](#) has more details to help with the migration to the new provider-based APIs to execute external processes at configuration time.

Builds that do not use the configuration cache, or only start external processes at execution time are not affected by this change.

Deprecations

Mutating core plugin configuration usage

The allowed usage of a configuration should be immutable after creation. Mutating the allowed usage on a configuration created by a Gradle core plugin is deprecated. This includes calling any of the following `Configuration` methods:

- `setCanBeConsumed(boolean)`
- `setCanBeResolved(boolean)`

These methods now emit deprecation warnings on these configurations, except for certain special cases which make allowances for the existing behavior of popular plugins. This rule does not yet apply to detached configurations or configurations created in buildscripts and third-party plugins. Calling `setCanBeConsumed(false)` on `apiElements` or `runtimeElements` is not yet deprecated in order to avoid warnings that would be otherwise emitted when using select popular third-party plugins.

This change is part of a larger ongoing effort to make the intended behavior of configurations more consistent and predictable, and to unlock further speed and memory improvements in this area of Gradle.

The ability to change the allowed usage of a configuration after creation will be removed in Gradle 9.0.

Reserved configuration names

Configuration names "detachedConfiguration" and "detachedConfigurationX" (where X is any integer) are reserved for internal use when creating detached configurations.

The ability to create non-detached configurations with these names will be removed in Gradle 9.0.

Calling select methods on the `JavaPluginExtension` without the `java` component present

Starting in Gradle 8.1, calling any of the following methods on `JavaPluginExtension` without the presence of the default `java` component is deprecated:

- `withJavadocJar()`
- `withSourcesJar()`
- `consistentResolution(Action)`

This `java` component is added by the `JavaPlugin`, which is applied by any of the Gradle JVM plugins including:

- `java-library`
- `application`
- `groovy`
- `scala`

Starting in Gradle 9.0, calling any of the above listed methods without the presence of the default `java` component will become an error.

`WarPlugin#configureConfiguration(ConfigurationContainer)`

Starting in Gradle 8.1, calling `WarPlugin#configureConfiguration(ConfigurationContainer)` is deprecated. This method was intended for internal use and was never intended to be used as part of the public interface.

Starting in Gradle 9.0, this method will be removed without replacement.

Relying on conventions for custom Test tasks

By default, when applying the `java` plugin, the `testClassesDirs` and `classpath` of all `Test` tasks have the same convention. Unless otherwise changed, the default behavior is to execute the tests from the default `test TestSuite` by configuring the task with the `classpath` and `testClassesDirs` from the `test` suite. This behavior will be removed in Gradle 9.0.

While this existing default behavior is correct for the use case of executing the default unit test

suite under a different environment, it does not support the use case of executing an entirely separate set of tests.

If you wish to continue including these tests, use the following code to avoid the deprecation warning in 8.1 and prepare for the behavior change in 9.0. Alternatively, consider migrating to test suites.

build.gradle.kts

```
val test by testing.suites.existing(JvmTestSuite::class)
tasks.named<Test>("myTestTask") {
    testClassesDirs = files(test.map { it.sources.output.classesDirs })
    classpath = files(test.map { it.sources.runtimeClasspath })
}
```

build.gradle

```
tasks.myTestTask {
    testClassesDirs = testing.suites.test.sources.output.classesDirs
    classpath = testing.suites.test.sources.runtimeClasspath
}
```

Modifying Gradle Module Metadata after a publication has been populated

Altering the [GMM](#) (e.g., changing a component configuration variants) **after** a Maven or Ivy publication has been populated from their components is now deprecated. This feature will be removed in Gradle 9.0.

Eager population of the publication can happen if the following methods are called:

- Maven
 - [MavenPublication.getArtifacts\(\)](#)
- Ivy
 - [IvyPublication.getArtifacts\(\)](#)
 - [IvyPublication.getConfigurations\(\)](#)
 - [IvyPublication.configurations\(Action\)](#)

Previously, the following code did not generate warnings, but it created inconsistencies between published artifacts:

build.gradle.kts

```
publishing {
    publications {
        create<MavenPublication>("maven") {
            from(components["java"])
        }
        create<IvyPublication>("ivy") {
            from(components["java"])
        }
    }
}

// These calls eagerly populate the Maven and Ivy publications

(publishing.publications["maven"] as MavenPublication).artifacts
(publishing.publications["ivy"] as IvyPublication).artifacts

val javaComponent = components["java"] as AdhocComponentWithVariants
javaComponent.withVariantsFromConfiguration(configurations["apiElements"]) {
    skip() }
javaComponent.withVariantsFromConfiguration(configurations["runtimeElements"]
) { skip() }
```

build.gradle

```
publishing {
    publications {
        maven(MavenPublication) {
            from components.java
        }
        ivy(IvyPublication) {
            from components.java
        }
    }
}

// These calls eagerly populate the Maven and Ivy publications

publishing.publications.maven.artifacts
publishing.publications.ivy.artifacts

components.java.withVariantsFromConfiguration(configurations.apiElements) {
    skip() }
components.java.withVariantsFromConfiguration(configurations.runtimeElements)
{ skip() }
```

In this example, the Maven and Ivy publications will contain the main JAR artifacts for the project, whereas the GMM [module file](#) will omit them.

Running tests on JVM versions 6 and 7

Running JVM tests on JVM versions older than 8 is deprecated. Testing on these versions will become an error in Gradle 9.0

Applying Kotlin DSL precompiled scripts published with Gradle < 6.0

Applying Kotlin DSL precompiled scripts published with Gradle < 6.0 is deprecated. Please use a version of the plugin published with Gradle >= 6.0.

Applying the `kotlin-dsl` together with Kotlin Gradle Plugin < 1.8.0

Applying the `kotlin-dsl` together with Kotlin Gradle Plugin < 1.8.0 is deprecated. Please let Gradle control the version of `kotlin-dsl` by removing any explicit `kotlin-dsl` version constraints from your build logic. This will let the `kotlin-dsl` plugin decide which version of the Kotlin Gradle Plugin to use. If you explicitly declare which version of the Kotlin Gradle Plugin to use for your build logic, update it to >= 1.8.0.

Accessing `libraries` or `bundles` from dependency version catalogs in the `plugins {}` block of a Kotlin script

Accessing `libraries` or `bundles` from dependency version catalogs in the `plugins {}` block of a Kotlin script is deprecated. Please only use `versions` or `plugins` from dependency version catalogs in the `plugins {}` block.

Using `ValidatePlugins` task without a Java Toolchain

Using a task of type `ValidatePlugins` without applying the `Java Toolchains` plugin is deprecated, and will become an error in Gradle 9.0.

To avoid this warning, please apply the plugin to your project:

build.gradle.kts

```
plugins {  
    id("jvm-toolchains")  
}
```

build.gradle

```
plugins {  
    id 'jvm-toolchains'  
}
```

The Java Toolchains plugin is applied automatically by the [Java library plugin](#) or other JVM plugins. So you can apply any of them to your project and it will fix the warning.

Deprecated members of the `org.gradle.util` package now report their deprecation

These members will be removed in Gradle 9.0.

- `WrapUtil.toDomainObjectSet(...)`
- `GUtil.toCamelCase(...)`
- `GUtil.toLowerCase(...)`
- `ConfigureUtil`

Deprecated JVM vendor IBM Semeru

The enum constant `JvmVendorSpec.IBM_SEMERU` is now deprecated and will be removed in Gradle 9.0.

Please replace it by its equivalent `JvmVendorSpec.IBM` to avoid warnings and potential errors in the next major version release.

Setting custom build layout on `StartParameter` and `GradleBuild`

Following the [related previous deprecation](#) of the behaviour in Gradle 7.1, it is now also deprecated to use related `StartParameter` and `GradleBuild` properties. These properties will be removed in Gradle 9.0.

Setting custom build file using `buildFile` property in `GradleBuild` task has been deprecated.

Please use the `dir` property instead to specify the root of the nested build. Alternatively, consider using one of the recommended alternatives for `GradleBuild` task.

Setting custom build layout using `StartParameter` methods `setBuildFile(File)` and `setSettingsFile(File)` as well as the counterpart getters `getBuildFile()` and `getSettingsFile()` have been deprecated.

Please use standard locations for settings and build files:

- settings file in the root of the build
- build file in the root of each subproject

Deprecated `org.gradle.cache.cleanup` property

The `org.gradle.cache.cleanup` property in `gradle.properties` under Gradle User Home has been deprecated. Please use the [cache cleanup DSL](#) instead to disable or modify the cleanup configuration.

Since the `org.gradle.cache.cleanup` property may still be needed for older versions of Gradle, this property may still be present and no deprecation warnings will be printed as long as it is also configured via the DSL. The DSL value will always take preference over the `org.gradle.cache.cleanup` property. If the desired configuration is to disable cleanup for older versions of Gradle (using `org.gradle.cache.cleanup`), but to enable cleanup with the default values

for Gradle versions at or above Gradle 8, then cleanup should be configured to use `Cleanup.DEFAULT`:

cache-settings.gradle

```
if (GradleVersion.current() >= GradleVersion.version('8.0')) {  
    apply from: "gradle8/cache-settings.gradle"  
}
```

cache-settings.gradle.kts

```
if (GradleVersion.current() >= GradleVersion.version("8.0")) {  
    apply(from = "gradle8/cache-settings.gradle")  
}
```

gradle8/cache-settings.gradle

```
beforeSettings { settings ->  
    settings.caches {  
        cleanup = Cleanup.DEFAULT  
    }  
}
```

gradle8/cache-settings.gradle.kts

```
beforeSettings {  
    caches {  
        cleanup.set(Cleanup.DEFAULT)  
    }  
}
```

Deprecated using relative paths to specify Java executables

Using relative file paths to point to Java executables is now deprecated and will become an error in Gradle 9. This is done to reduce confusion about what such relative paths should resolve against.

Calling `Task.getConvention()`, `Task.getExtensions()` from a task action

Calling `Task.getConvention()`, `Task.getExtensions()` from a task action at execution time is now

deprecated and will be made an error in Gradle 9.0.

See the [configuration cache chapter](#) for details on how to migrate these usages to APIs that are supported by the configuration cache.

Deprecated running test task successfully when no test executed

Running the `Test` task successfully when no test was executed is now deprecated and will become an error in Gradle 9. Note that it is not an error when no test sources are present, in this case the `test` task is simply skipped. It is only an error when test sources are present, but no test was selected for execution. This is changed to avoid accidental successful test runs due to erroneous configuration.

Changes in the IDE integration

Workaround for false positive errors shown in Kotlin DSL `plugins {}` block using version catalog is not needed anymore

Version catalog accessors for plugin aliases in the `plugins {}` block aren't shown as errors in IntelliJ IDEA and Android Studio Kotlin script editor anymore.

If you were using the `@Suppress("DSL_SCOPE_VIOLATION")` annotation as a workaround, you can now remove it.

If you were using the [Gradle Libs Error Suppressor](#) IntelliJ IDEA plugin, you can now uninstall it.

After upgrading Gradle to 8.1 you will need to clear the IDE caches and restart.

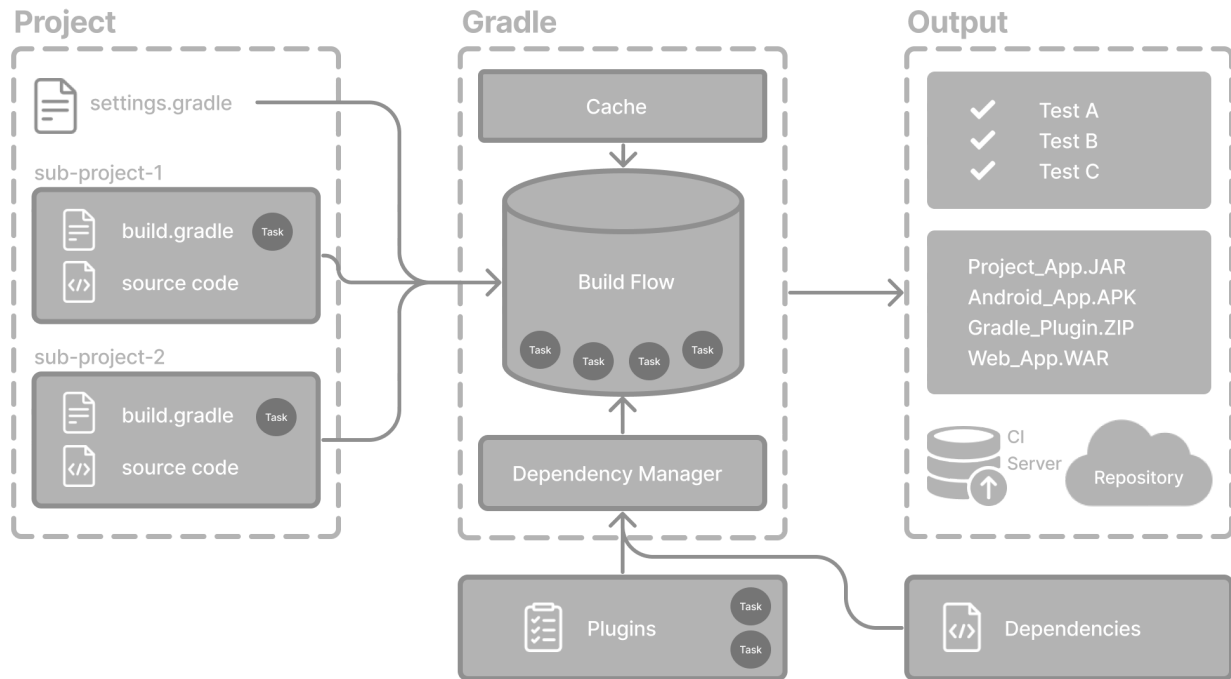
Also see [the deprecated usages of version catalogs in the Kotlin DSL `plugins {}` block](#) above.

RUNNING GRADLE BUILDS

CORE CONCEPTS

Gradle Basics

Gradle **automates building, testing, and deployment of software** from information in **build scripts**.



Gradle core concepts

Projects

A Gradle **project** is a piece of software that can be built, such as an application or a library.

Single project builds include a single project called the **root project**.

Multi-project builds include **one root project** and **any number of subprojects**.

Build Scripts

Build scripts detail to Gradle what steps to take to build the project.

Each project can include one or more build scripts.

Dependency Management

Dependency management is an automated technique for declaring and resolving external resources required by a project.

Each project typically includes a number of external dependencies that Gradle will resolve during the build.

Tasks

Tasks are a basic unit of work such as compiling code or running your test.

Each project contains one or more tasks defined inside a build script or a plugin.

Plugins

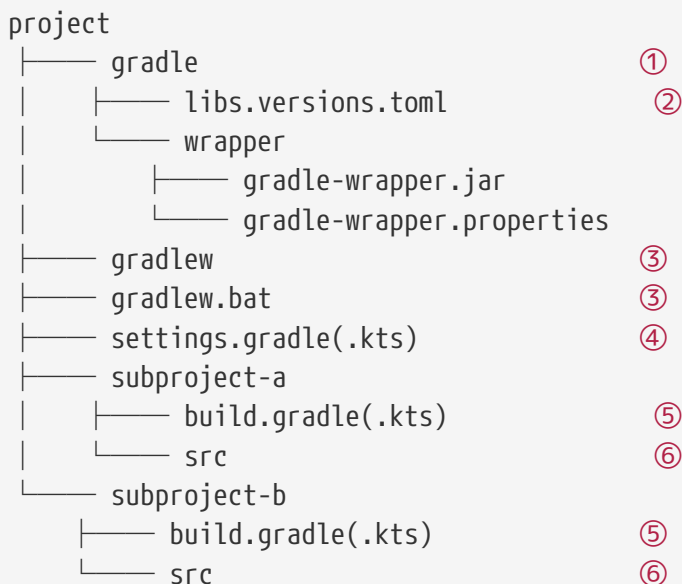
Plugins are used to **extend Gradle's capability** and optionally contribute **tasks** to a project.

Gradle project structure

Many developers will interact with Gradle for the first time through an existing project.

The presence of the `gradlew` and `gradlew.bat` files in the root directory of a project is a clear indicator that Gradle is used.

A Gradle project will look similar to the following:



- ① Gradle directory to store wrapper files and more
- ② Gradle version catalog for dependency management
- ③ Gradle wrapper scripts
- ④ Gradle settings file to define a root project name and subprojects
- ⑤ Gradle build scripts of the two subprojects - `subproject-a` and `subproject-b`
- ⑥ Source code and/or additional files for the projects

Invoking Gradle

IDE

Gradle is **built-in to many IDEs** including Android Studio, IntelliJ IDEA, Visual Studio Code, Eclipse, and NetBeans.

Gradle can be automatically invoked when you build, clean, or run your app in the IDE.

It is recommended that you consult the manual for the IDE of your choice to learn more about how Gradle can be used and configured.

Command line

Gradle can be invoked in the command line once [installed](#). For example:

```
$ gradle build
```

NOTE

Most projects do not use the installed version of Gradle.

Gradle Wrapper

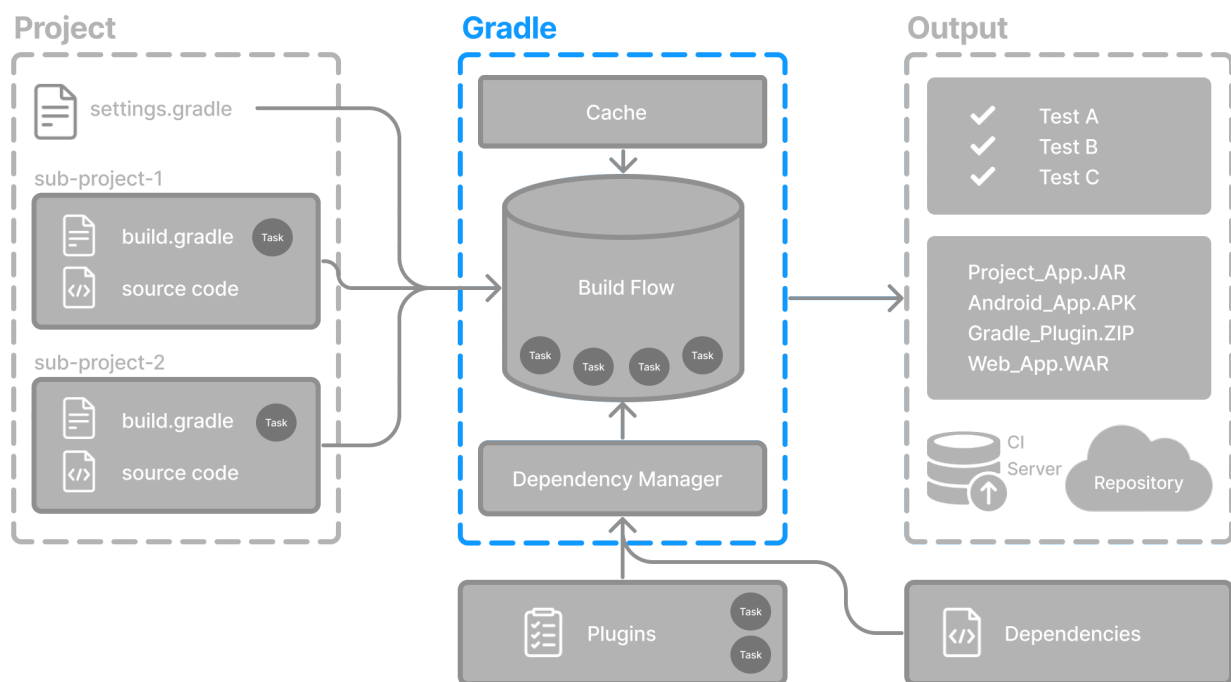
The Wrapper is a script that invokes a declared version of Gradle and is **the recommended way to execute a Gradle build**. It is found in the project root directory as a `gradlew` or `gradlew.bat` file:

```
$ gradlew build    // Linux or OSX
$ gradlew.bat build // Windows
```

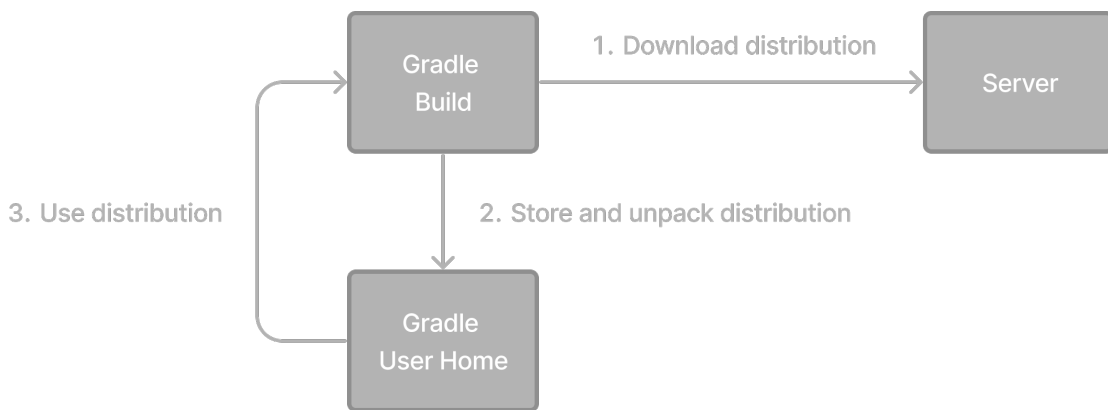
Next Step: [Learn about the Gradle Wrapper](#) >>

Gradle Wrapper Basics

The **recommended way to execute any Gradle build** is with the Gradle Wrapper.



The *Wrapper* script invokes a declared version of Gradle, downloading it beforehand if necessary.



The Wrapper is available as a `gradlew` or `gradlew.bat` file.

The Wrapper provides the following benefits:

- Standardizes a project on a given Gradle version.
- Provisions the same Gradle version for different users.
- Provisions the Gradle version for different execution environments (IDEs, CI servers...).

Using the Gradle Wrapper

It is always recommended to execute a build with the Wrapper to ensure a reliable, controlled, and standardized execution of the build.

Depending on the operating system, you run `gradlew` or `gradlew.bat` instead of the `gradle` command.

Typical Gradle invocation:

```
$ gradle build
```

To run the Wrapper on a Linux or OSX machine:

```
$ ./gradlew build
```

To run the Wrapper on Windows PowerShell:

```
$ .\gradlew.bat build
```

The command is run in the same directory that the Wrapper is located in. If you want to run the command in a different directory, you must provide the relative path to the Wrapper:

```
$ ../gradlew build
```

The following console output demonstrates the use of the Wrapper on a Windows machine, in the command prompt (cmd), for a Java-based project:

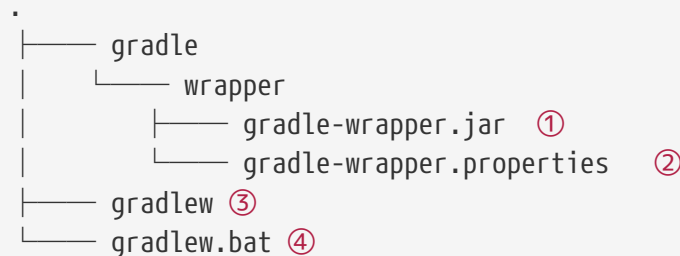
```
$ gradlew.bat build

Downloading https://services.gradle.org/distributions/gradle-5.0-all.zip
.....
Unzipping C:\Documents and Settings\Claudia\.gradle\wrapper\dists\gradle-5.0-
all\ac27o8rbd0ic8ih41or9l32mv\gradle-5.0-all.zip to C:\Documents and
Settings\Claudia\.gradle\wrapper\dists\gradle-5.0-al\ac27o8rbd0ic8ih41or9l32mv
Set executable permissions for: C:\Documents and
Settings\Claudia\.gradle\wrapper\dists\gradle-5.0-
all\ac27o8rbd0ic8ih41or9l32mv\gradle-5.0\bin\gradle

BUILD SUCCESSFUL in 12s
1 actionable task: 1 executed
```

Understanding the Wrapper files

The following files are part of the Gradle Wrapper:



- ① **gradle-wrapper.jar**: This is a small JAR file that contains the Gradle Wrapper code. It is responsible for downloading and installing the correct version of Gradle for a project if it's not already installed.
- ② **gradle-wrapper.properties**: This file contains configuration properties for the Gradle Wrapper, such as the distribution URL (where to download Gradle from) and the distribution type (ZIP or TARBALL).
- ③ **gradlew**: This is a shell script (Unix-based systems) that acts as a wrapper around **gradle-wrapper.jar**. It is used to execute Gradle tasks on Unix-based systems without needing to manually install Gradle.
- ④ **gradlew.bat**: This is a batch script (Windows) that serves the same purpose as **gradlew** but is used on Windows systems.

IMPORTANT | You should never alter these files.

If you want to view or update the Gradle version of your project, use the command line. Do not edit the wrapper files manually:

```
$ ./gradlew --version
$ ./gradlew wrapper --gradle-version 7.2
```

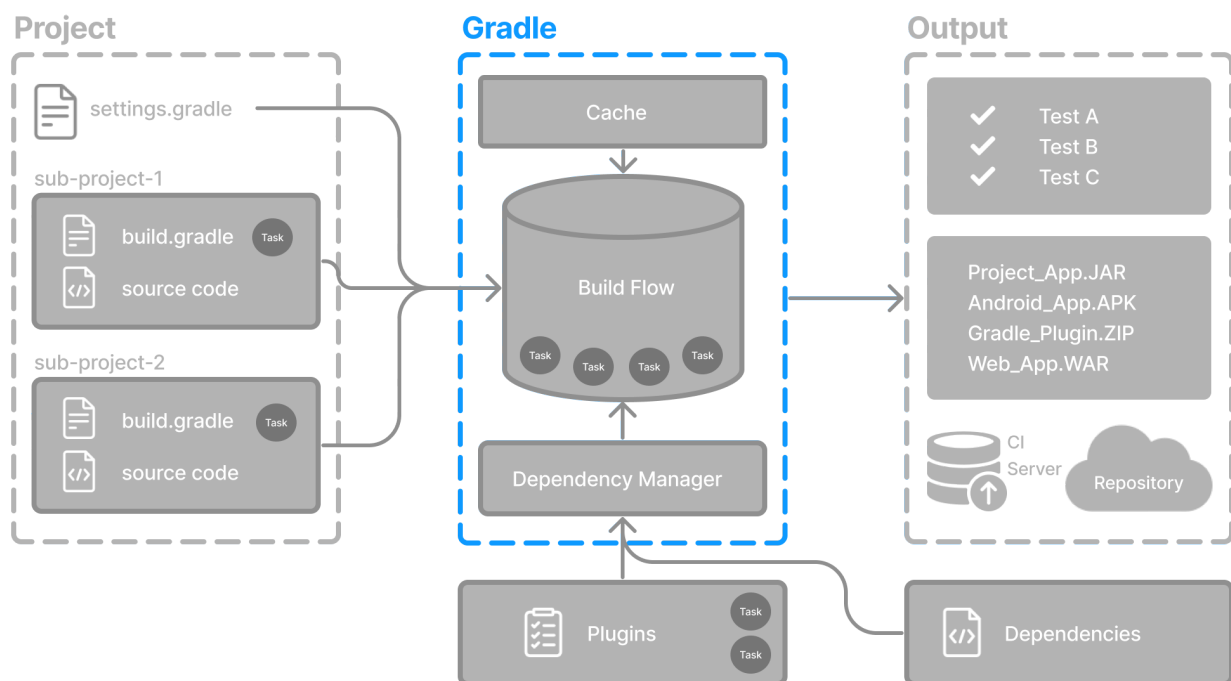
```
$ gradlew.bat --version
$ gradlew.bat wrapper --gradle-version 7.2
```

Consult the [Gradle Wrapper reference](#) to learn more.

Next Step: [Learn about the Gradle CLI](#) >>

Command-Line Interface Basics

The command-line interface is the primary **method of interacting with Gradle** outside the IDE.



Use of the [Gradle Wrapper](#) is highly encouraged.

Substitute `./gradlew` (in macOS / Linux) or `gradlew.bat` (in Windows) for `gradle` in the following examples.

Executing Gradle on the command line conforms to the following structure:

```
gradle [taskName...] [--option-name...]
```

Options are allowed *before* and *after* task names.

```
gradle [--option-name...] [taskName...]
```

If multiple tasks are specified, you should separate them with a space.

```
gradle [taskName1 taskName2...] [--option-name...]
```

Options that accept values can be specified with or without `=` between the option and argument. The use of `=` is recommended.

```
gradle [...] --console=plain
```

Options that enable behavior have long-form options with inverses specified with `--no-`. The following are opposites.

```
gradle [...] --build-cache  
gradle [...] --no-build-cache
```

Many long-form options have short-option equivalents. The following are equivalent:

```
gradle --help  
gradle -h
```

Command-line usage

The following sections describe the use of the Gradle command-line interface. Some plugins also add their own command line options.

Executing tasks

To execute a task called `taskName` on the root project, type:

```
$ gradle :taskName
```

This will run the single `taskName` and all of its [dependencies](#).

Specify options for tasks

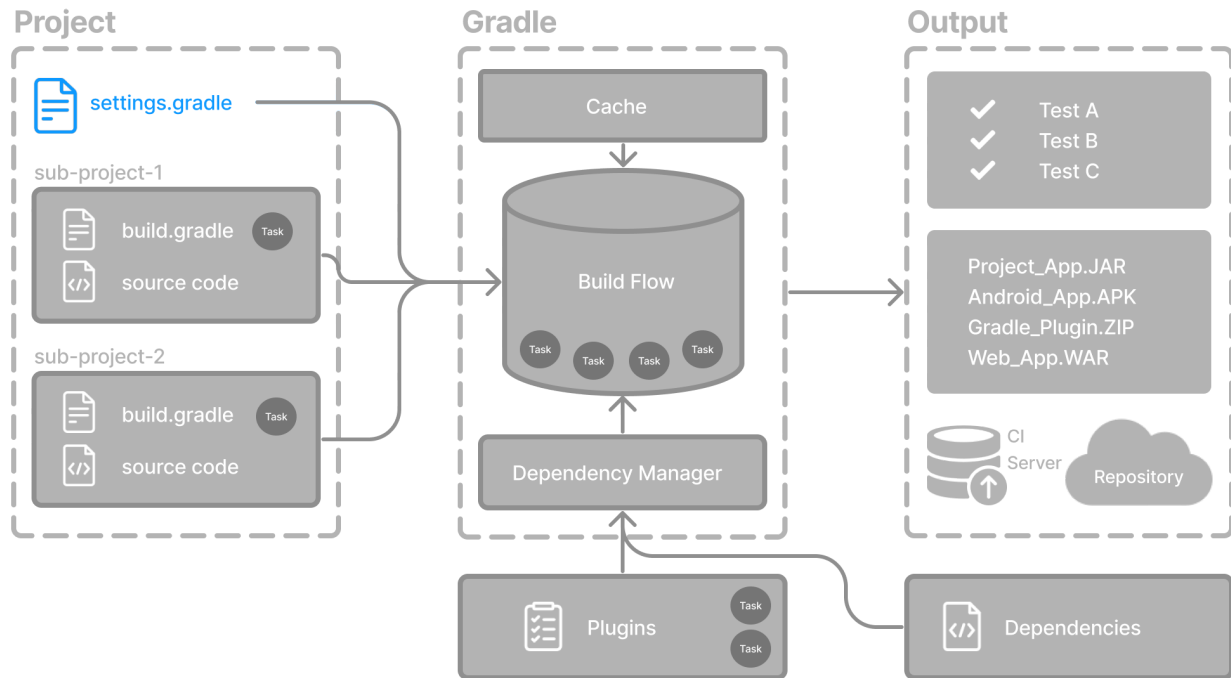
To pass an option to a task, prefix the option name with `--` after the task name:

```
$ gradle taskName --exampleOption=exampleValue
```

Consult the [Gradle Command Line Interface reference](#) to learn more.

Settings File Basics

The settings file is the **entry point** of every Gradle project.



The primary purpose of the *settings* file is to add subprojects to your build.

Gradle supports single and multi-project builds.

- For single-project builds, the settings file is optional.
- For multi-project builds, the settings file is mandatory and declares all subprojects.

Settings script

The settings file is a script. It is either a `settings.gradle` file written in Groovy or a `settings.gradle.kts` file in Kotlin.

The [Groovy DSL](#) and the [Kotlin DSL](#) are the only accepted languages for Gradle scripts.

The settings file is typically found in the root directory of the project.

Let's take a look at an example and break it down:

settings.gradle.kts

```
rootProject.name = "root-project" ①
```

```
include("sub-project-a") ②
```

```
include("sub-project-b")
include("sub-project-c")
```

- ① Define the project name.
- ② Add subprojects.

settings.gradle

```
rootProject.name = 'root-project' ①

include('sub-project-a')           ②
include('sub-project-b')
include('sub-project-c')
```

- ① Define the project name.
- ② Add subprojects.

1. Define the project name

The settings file defines your project name:

```
rootProject.name = "root-project"
```

There is only one root project per build.

2. Add subprojects

The settings file defines the structure of the project by including subprojects, if there are any:

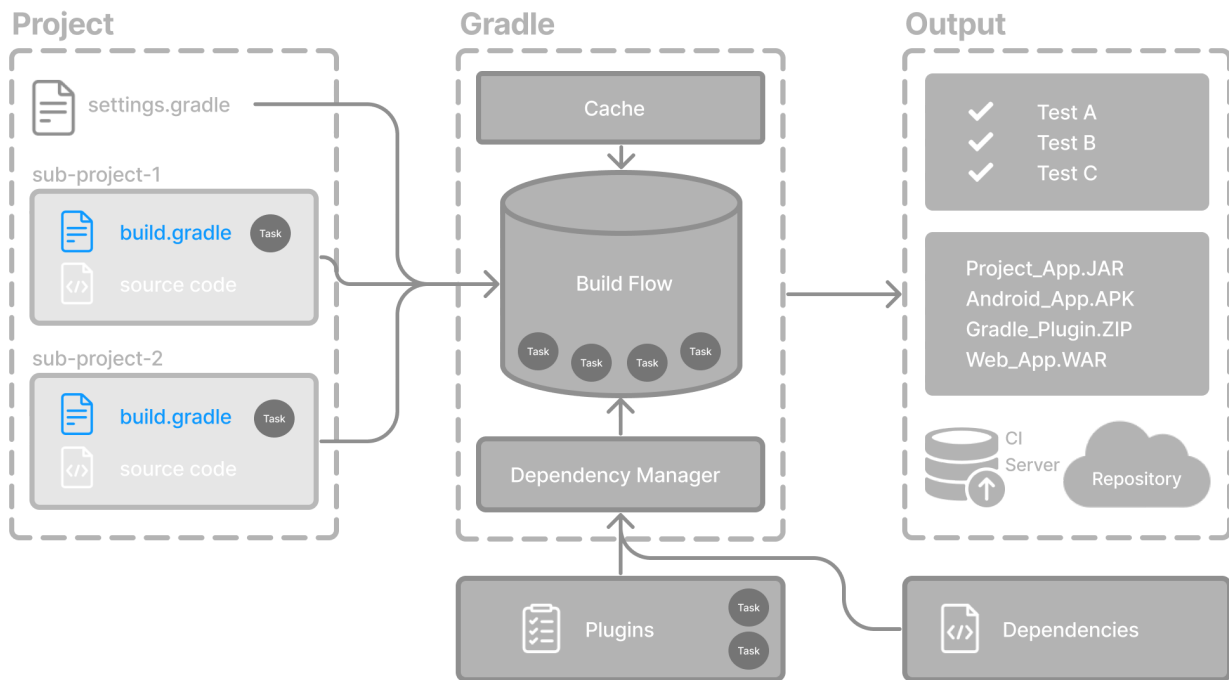
```
include("app")
include("business-logic")
include("data-model")
```

Consult the [Writing Settings File](#) page to learn more.

Next Step: [Learn about the Build scripts >>](#)

Build File Basics

Generally, a build script details **build configuration, tasks, and plugins**.



Every Gradle build comprises at least one *build script*.

In the build file, two types of dependencies can be added:

1. The libraries and/or plugins on which Gradle and the build script depend.
2. The libraries on which the project sources (i.e., source code) depend.

Build scripts

The build script is either a `build.gradle` file written in Groovy or a `build.gradle.kts` file in Kotlin.

The [Groovy DSL](#) and the [Kotlin DSL](#) are the only accepted languages for Gradle scripts.

Let's take a look at an example and break it down:

build.gradle.kts

```
plugins {  
    id("application")  
}  
  
application {  
    mainClass = "com.example.Main"  
}
```

① Add plugins.

② Use convention properties.

build.gradle

```
plugins {  
    id 'application' ①  
}  
  
application {  
    mainClass = 'com.example.Main' ②  
}
```

① Add plugins.

② Use convention properties.

1. Add plugins

Plugins extend Gradle's functionality and can contribute tasks to a project.

Adding a plugin to a build is called *applying* a plugin and makes additional functionality available.

```
plugins {  
    id("application")  
}
```

The **application** plugin facilitates creating an executable JVM application.

Applying the **Application plugin** also implicitly applies the **Java plugin**. The **java** plugin adds Java compilation along with testing and bundling capabilities to a project.

2. Use convention properties

A plugin adds tasks to a project. It also adds properties and methods to a project.

The **application** plugin defines tasks that package and distribute an application, such as the **run** task.

The Application plugin provides a way to declare the main class of a Java application, which is required to execute the code.

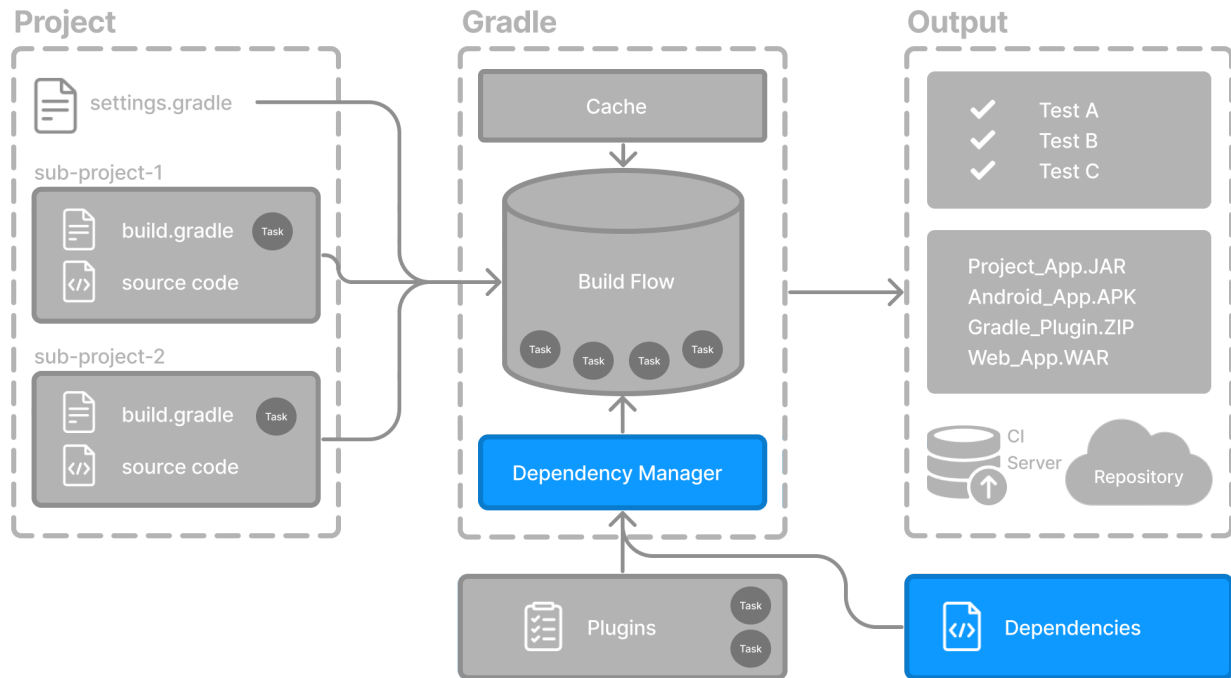
```
application {  
    mainClass = "com.example.Main"  
}
```

In this example, the main class (i.e., the point where the program's execution begins) is **com.example.Main**.

Consult the [Writing Build Scripts](#) page to learn more.

Dependency Management Basics

Gradle has built-in support for **dependency management**.



Dependency management is an automated technique for declaring and resolving external resources required by a project.

Gradle build scripts define the process to build projects that may require external dependencies. Dependencies refer to JARs, plugins, libraries, or source code that support building your project.

Version Catalog

Version catalogs provide a way to centralize your dependency declarations in a `libs.versions.toml` file.

The catalog makes sharing dependencies and version configurations between subprojects simple. It also allows teams to enforce versions of libraries and plugins in large projects.

The version catalog typically contains four sections:

1. `[versions]` to declare the version numbers that plugins and libraries will reference.
2. `[libraries]` to define the libraries used in the build files.
3. `[bundles]` to define a set of dependencies.
4. `[plugins]` to define plugins.

`[versions]`

```

androidGradlePlugin = "7.4.1"
mockito = "2.16.0"

[libraries]
googleMaterial = { group = "com.google.android.material", name = "material", version =
"1.1.0-alpha05" }
mockitoCore = { module = "org.mockito:mockito-core", version.ref = "mockito" }

[plugins]
androidApplication = { id = "com.android.application", version.ref =
"androidGradlePlugin" }

```

The file is located in the **gradle** directory so that it can be used by Gradle and IDEs automatically. The version catalog should be checked into source control: **gradle/libs.versions.toml**.

Declaring Your Dependencies

To add a dependency to your project, specify a dependency in the dependencies block of your **build.gradle(.kts)** file.

The following **build.gradle.kts** file adds a plugin and two dependencies to the project using the version catalog above:

```

plugins {
    alias(libs.plugins.androidApplication) ❶
}

dependencies {
    // Dependency on a remote binary to compile and run the code
    implementation(libs.googleMaterial) ❷

    // Dependency on a remote binary to compile and run the test code
    testImplementation(libs.mockitoCore) ❸
}

```

- ❶ Applies the Android Gradle plugin to this project, which adds several features that are specific to building Android apps.
- ❷ Adds the Material dependency to the project. Material Design provides components for creating a user interface in an Android App. This library will be used to compile and run the Kotlin source code in this project.
- ❸ Adds the Mockito dependency to the project. Mockito is a mocking framework for testing Java code. This library will be used to compile and run the *test* source code in this project.

Dependencies in Gradle are grouped by **configurations**.

- The **material** library is added to the **implementation** configuration, which is used for compiling and running *production* code.
- The **mockito-core** library is added to the **testImplementation** configuration, which is used for

compiling and running *test* code.

NOTE | There are many more configurations available.

Viewing Project Dependencies

You can view your dependency tree in the terminal using the `./gradlew :app:dependencies` command:

```
$ ./gradlew :app:dependencies

> Task :app:dependencies

-----
Project ':app'
-----

implementation - Implementation only dependencies for source set 'main'. (n)
\--- com.google.android.material:material:1.1.0-alpha05 (n)

testImplementation - Implementation only dependencies for source set 'test'. (n)
\--- org.mockito:mockito-core:2.16.0 (n)

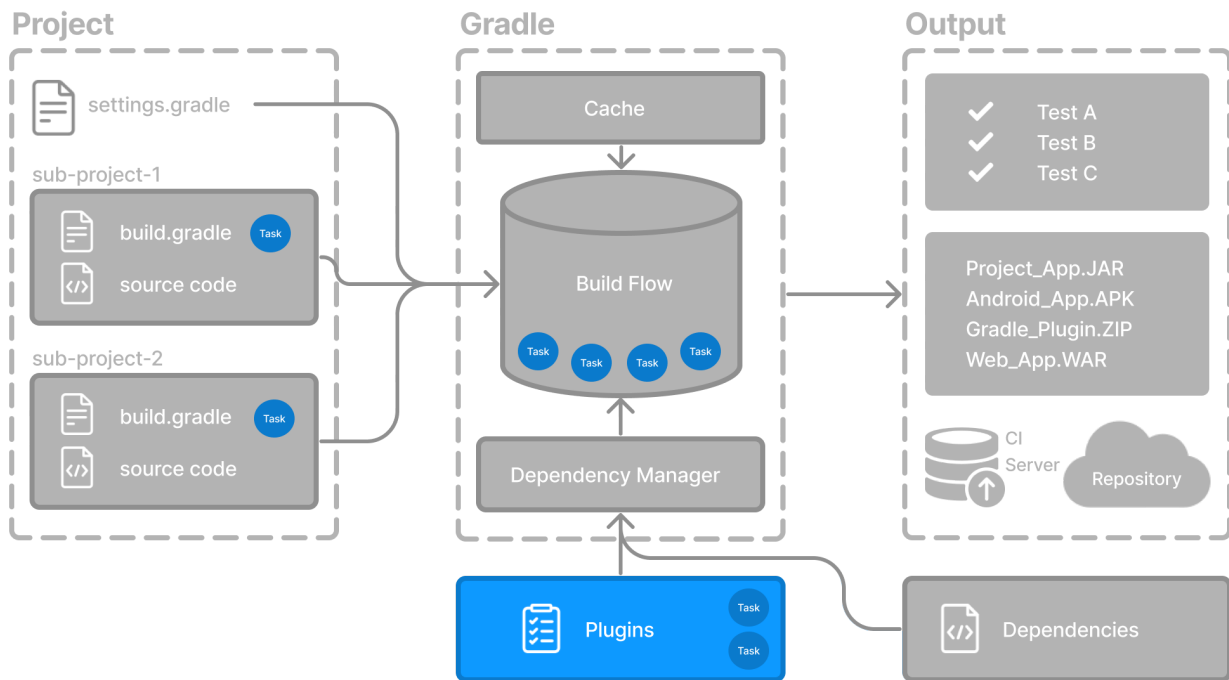
...
```

Consult the [Dependency Management chapter](#) to learn more.

Next Step: [Learn about Tasks](#) >>

Task Basics

A task represents some **independent unit of work** that a build performs, such as compiling classes, creating a JAR, generating Javadoc, or publishing archives to a repository.



You run a Gradle **build** task using the **gradle** command or by invoking the Gradle Wrapper (`./gradlew` or `gradlew.bat`) in your project directory:

```
$ ./gradlew build
```

Available tasks

All available tasks in your project come from Gradle plugins and build scripts.

You can list all the available tasks in the project by running the following command in the terminal:

```
$ ./gradlew tasks
```

Application tasks

`run` - Runs this project as a JVM application

Build tasks

`assemble` - Assembles the outputs of this project.

`build` - Assembles and tests this project.

...

Documentation tasks

`javadoc` - Generates Javadoc API documentation for the main source code.

...

Other tasks

compileJava - Compiles main Java source.

...

Running tasks

The `run` task is executed with `./gradlew run`:

```
$ ./gradlew run

> Task :app:compileJava
> Task :app:processResources NO-SOURCE
> Task :app:classes

> Task :app:run
Hello World!

BUILD SUCCESSFUL in 904ms
2 actionable tasks: 2 executed
```

In this example Java project, the output of the `run` task is a `Hello World` statement printed on the console.

Task dependency

Many times, a task requires another task to run first.

For example, for Gradle to execute the `build` task, the Java code must first be compiled. Thus, the `build` task *depends* on the `compileJava` task.

This means that the `compileJava` task will run *before* the `build` task:

```
$ ./gradlew build

> Task :app:compileJava
> Task :app:processResources NO-SOURCE
> Task :app:classes
> Task :app:jar
> Task :app:startScripts
> Task :app:distTar
> Task :app:distZip
> Task :app:assemble
> Task :app:compileTestJava
> Task :app:processTestResources NO-SOURCE
```

```
> Task :app:testClasses
> Task :app:test
> Task :app:check
> Task :app:build
```

```
BUILD SUCCESSFUL in 764ms
7 actionable tasks: 7 executed
```

Build scripts can optionally define task dependencies. Gradle then automatically determines the task execution order.

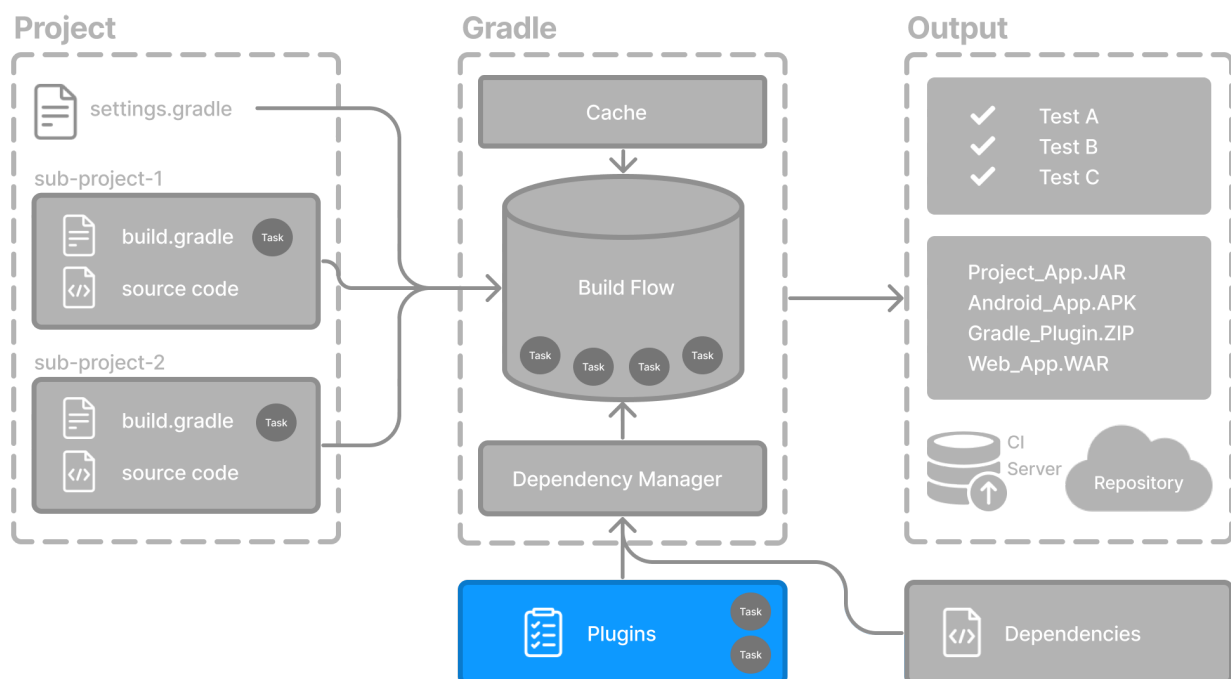
Consult the [Task development chapter](#) to learn more.

Next Step: [Learn about Plugins >>](#)

Plugin Basics

Gradle is built on a plugin system. Gradle itself is primarily composed of infrastructure, such as a sophisticated dependency resolution engine. The rest of its functionality comes from plugins.

A plugin is a piece of software that **provides additional functionality to the Gradle build system**.



Plugins can be applied to a Gradle build script to **add new tasks, configurations, or other build-related capabilities**:

The Java Library Plugin - `java-library`

Used to define and build Java libraries. It compiles Java source code with the `compileJava` task, generates Javadoc with the `javadoc` task, and packages the compiled classes into a JAR file with the `jar` task.

The Google Services Gradle Plugin - `com.google.gms:google-services`

Enables Google APIs and Firebase services in your Android application with a configuration block called `googleServices{}` and a task called `generateReleaseAssets`.

The Gradle Bintray Plugin - `com.jfrog.bintray`

Allows you to publish artifacts to Bintray by configuring the plugin using the `bintray{}` block.

Plugin distribution

Plugins are distributed in three ways:

1. **Core plugins** - Gradle develops and maintains a set of [Core Plugins](#).
2. **Community plugins** - Gradle's community shares plugins via the [Gradle Plugin Portal](#).
3. **Local plugins** - Gradle enables users to create custom plugins using [APIs](#).

Applying plugins

Applying a plugin to a project allows the plugin to extend the project's capabilities.

You apply plugins in the build script using a **plugin id** (a globally unique identifier / name) and a version:

```
plugins {  
    id «plugin id» version «plugin version»  
}
```

1. Core plugins

Gradle Core plugins are a set of plugins that are included in the Gradle distribution itself. These plugins provide essential functionality for building and managing projects.

Some examples of core plugins include:

- **java**: Provides support for building Java projects.
- **groovy**: Adds support for compiling and testing Groovy source files.
- **ear**: Adds support for building EAR files for enterprise applications.

Core plugins are unique in that they provide short names, such as `java` for the core [JavaPlugin](#), when applied in build scripts. They also do not require versions. To apply the `java` plugin to a project:

build.gradle.kts

```
plugins {  
    id("java")  
}
```

There are many [Gradle Core Plugins](#) users can take advantage of.

2. Community plugins

Community plugins are plugins developed by the Gradle community, rather than being part of the core Gradle distribution. These plugins provide additional functionality that may be specific to certain use cases or technologies.

The [Spring Boot Gradle plugin](#) packages executable JAR or WAR archives, and runs [Spring Boot](#) Java applications.

To apply the `org.springframework.boot` plugin to a project:

build.gradle.kts

```
plugins {  
    id("org.springframework.boot") version "3.1.5"  
}
```

Community plugins can be published at the [Gradle Plugin Portal](#), where other Gradle users can easily discover and use them.

3. Local plugins

Custom or local plugins are developed and used within a specific project or organization. These plugins are not shared publicly and are tailored to the specific needs of the project or organization.

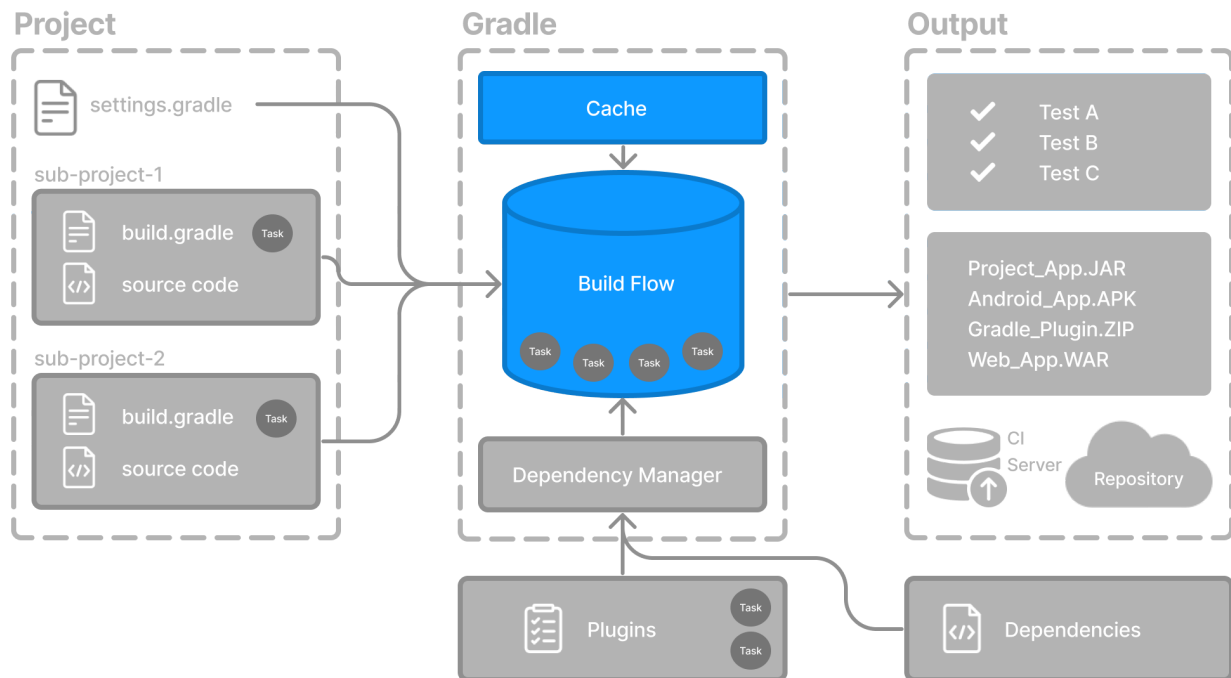
Local plugins can encapsulate common build logic, provide integrations with internal systems or tools, or abstract complex functionality into reusable components.

Gradle provides users with the ability to develop custom plugins using APIs. To create your own plugin, you'll typically follow these steps:

1. **Define the plugin class:** create a new class that implements the `Plugin<Project>` interface.

```
// Define a 'HelloPlugin' plugin  
class HelloPlugin : Plugin<Project> {  
    override fun apply(project: Project) {  
        // Define the 'hello' task  
        val helloTask = project.tasks.register("hello") {  
            doLast {  
                println("Hello, Gradle!")  
            }  
        }  
    }  
}
```

2. **Build and optionally publish your plugin:** generate a JAR file containing your plugin code and optionally publish this JAR to a repository (local or remote) to be used in other projects.



Incremental builds

An **incremental build** is a build that avoids running tasks whose inputs have not changed since the previous build. Re-executing such tasks is unnecessary if they would only re-produce the same output.

For incremental builds to work, tasks must define their inputs and outputs. Gradle will determine whether the input or outputs have changed at build time. If they have changed, Gradle will execute the task. Otherwise, it will skip execution.

Incremental builds are always enabled, and the best way to see them in action is to turn on *verbose mode*. With verbose mode, each task state is labeled during a build:

```
$ ./gradlew compileJava --console=verbose

> Task :buildSrc:generateExternalPluginSpecBuilders UP-TO-DATE
> Task :buildSrc:extractPrecompiledScriptPluginPlugins UP-TO-DATE
> Task :buildSrc:compilePluginsBlocks UP-TO-DATE
> Task :buildSrc:generatePrecompiledScriptPluginAccessors UP-TO-DATE
> Task :buildSrc:generateScriptPluginAdapters UP-TO-DATE
> Task :buildSrc:compileKotlin UP-TO-DATE
> Task :buildSrc:compileJava NO-SOURCE
> Task :buildSrc:compileGroovy NO-SOURCE
> Task :buildSrc:pluginDescriptors UP-TO-DATE
> Task :buildSrc:processResources UP-TO-DATE
> Task :buildSrc:classes UP-TO-DATE
> Task :buildSrc:jar UP-TO-DATE
> Task :list:compileJava UP-TO-DATE
> Task :utilities:compileJava UP-TO-DATE
> Task :app:compileJava UP-TO-DATE
```

```
BUILD SUCCESSFUL in 374ms
12 actionable tasks: 12 up-to-date
```

When you run a task that has been previously executed and hasn't changed, then **UP-TO-DATE** is printed next to the task.

TIP

To permanently enable verbose mode, add `org.gradle.console=verbose` to your `gradle.properties` file.

Build caching

Incremental Builds are a great optimization that helps avoid work already done. If a developer continuously changes a single file, there is likely no need to rebuild all the other files in the project.

However, what happens when the same developer switches to a new branch created last week? The files are rebuilt, even though the developer is building something that has been built before.

This is where a **build cache** is helpful.

The build cache stores previous build results and restores them when needed. It prevents the redundant work and cost of executing time-consuming and expensive processes.

When the build cache has been used to repopulate the local directory, the tasks are marked as **FROM-CACHE**:

```
$ ./gradlew compileJava --build-cache

> Task :buildSrc:generateExternalPluginSpecBuilders UP-TO-DATE
> Task :buildSrc:extractPrecompiledScriptPluginPlugins UP-TO-DATE
> Task :buildSrc:compilePluginsBlocks UP-TO-DATE
> Task :buildSrc:generatePrecompiledScriptPluginAccessors UP-TO-DATE
> Task :buildSrc:generateScriptPluginAdapters UP-TO-DATE
> Task :buildSrc:compileKotlin UP-TO-DATE
> Task :buildSrc:compileJava NO-SOURCE
> Task :buildSrc:compileGroovy NO-SOURCE
> Task :buildSrc:pluginDescriptors UP-TO-DATE
> Task :buildSrc:processResources UP-TO-DATE
> Task :buildSrc:classes UP-TO-DATE
> Task :buildSrc:jar UP-TO-DATE
> Task :list:compileJava FROM-CACHE
> Task :utilities:compileJava FROM-CACHE
> Task :app:compileJava FROM-CACHE

BUILD SUCCESSFUL in 364ms
12 actionable tasks: 3 from cache, 9 up-to-date
```

Once the local directory has been repopulated, the next execution will mark tasks as **UP-TO-DATE** and not **FROM-CACHE**.

Consult the [Build cache chapter](#) to learn more.

Build Scans

A build scan is a **representation of metadata captured** as you run your build.



Gradle [captures your build metadata](#) and sends it to the [Build Scan Service](#). The service then transforms the metadata into information you can analyze and share with others.

Build Scan® for 'lab' yesterday

scans.gradle.com/s/ikej32xgkuhy

Gradle Enterprise

lab :app:compileJava Sep 19 2023 15:14:45 PDT

Summary

Started yesterday at 15:14:45 PDT, finished yesterday at 15:14:48 PDT

Gradle 8.1.1, Gradle Enterprise plugin 3.12.6

Composite build (1 included build)

Explore console log

1 task failure

The :app:compileJava task failed. View task in console log

Could not resolve all files for configuration ':app:compileClasspath'.

> Could not download support-compat-28.0.0.aar (com.android.support:support-compat:28.0.0)

> Could not get resource 'https://packages.atlassian.com/maven-external/com/android/support/support-compat/28.0.0/support-compat-28.0.0.aar'.

> Could not HEAD 'https://d34y9yt11qeow3.cloudfront.net/filestore/d2/d252b640ed832cf8addc35ef0a9f9186dc7738a5?response-content-type=application%2'.

Explore failure

0 build deprecations

This build did not contain any deprecations.

14 tasks executed in 2 projects, 1 failure in 3s, with 10 avoided tasks saving 3.594s

:app:compileJava	FAILED	2.867s
:buildSrc:compileKotlin	UP-TO-DATE	0.007s
:buildSrc:generateExternalPluginSpecBuilders	UP-TO-DATE	0.003s
:buildSrc:compilePluginsBlocks	UP-TO-DATE	0.001s
:buildSrc:generatePrecompiledScriptPluginAccessors	UP-TO-DATE	0.001s
:buildSrc:generateScriptPluginAdapters	UP-TO-DATE	0.001s

Explore timeline

Request a trial

The information that scans collect can be an invaluable resource when troubleshooting, collaborating on, or optimizing the performance of your builds.

For example, with a build scan, it's no longer necessary to copy and paste error messages or include all the details about your environment each time you want to ask a question on Stack Overflow, Slack, or the Gradle Forum. Instead, copy the link to your latest build scan.

Build Scan® for 'lab' yesterday

scans.gradle.com/s/ikej32xgkuhy/failure#1

Gradle Enterprise

lab :app:compileJava Sep 19 2023 15:14:45 PDT

Summary

Console log

Failure

Deprecations

Timeline

Performance

Tests

Projects

Dependencies

Build dependencies

Plugins

Custom values

Switches

Infrastructure

Delete Build Scan

Request a trial

Failure 1 of 1 The :app:compileJava task failed. View task in console log

Could not resolve all files for configuration ':app:compileClasspath'.

> Could not download support-compat-28.0.0.aar (com.android.support:support-compat:28.0.0)

> Could not get resource 'https://packages.atlassian.com/maven-external/com/android/support/support-compat/28.0.0/support-compat-28.0.0.aar'.

> Could not HEAD 'https://d34y9yt11qeow3.cloudfront.net/filestore/d2/d252b640ed832cf8addc35ef0a9f9186dc7738a5?response-content-type=application%2'.

Exception

org.gradle.api.tasks.TaskExecutionException: Execution failed for task ':app:compileJava'.

at org.gradle.api.internal.tasks.execution.CatchExceptionTaskExecutor.execute(CatchExceptionTaskExecutor.java:38)

...

Caused by: org.gradle.api.internal.artifacts.ivyservice.DefaultLenientConfiguration\$ArtifactResolveException: Could not resolve all files for conf

at org.gradle.api.internal.artifacts.configurations.DefaultConfiguration.mapFailure(DefaultConfiguration.java:1716)

...

Caused by: org.gradle.internal.resolve.ArtifactResolveException: Could not download support-compat-28.0.0.aar (com.android.support:support-compat:

at org.gradle.api.internal.artifacts.ivyservice.ivyresolve.ErrorHandlingModuleComponentRepository\$ErrorHandlingModuleComponentRepositoryAccess

...

Caused by: org.gradle.api.resources.ResourceException: Could not get resource 'https://packages.atlassian.com/maven-external/com/android/support/s

at org.gradle.internal.resource.ResourceExceptions.failure(ResourceExceptions.java:74)

...

Caused by: org.gradle.internal.resource.transport.http.HttpErrorStatusCodeException: Could not HEAD 'https://d34y9yt11qeow3.cloudfront.net/filesto

at org.gradle.internal.resource.transport.http.HttpClientHelper.processResponse(HttpClientHelper.java:234)

...

Enable Build Scans

To enable build scans on a gradle command, add `--scan` to the command line option:

```
./gradlew build --scan
```

You may be prompted to agree to the terms to use Build Scans.

Visit the [Build Scans page](#) to learn more.

Captured Information

To see what data is captured and sent in a Build Scan, refer to the [Captured Information](#) section in the Gradle Develocity Plugin documentation.

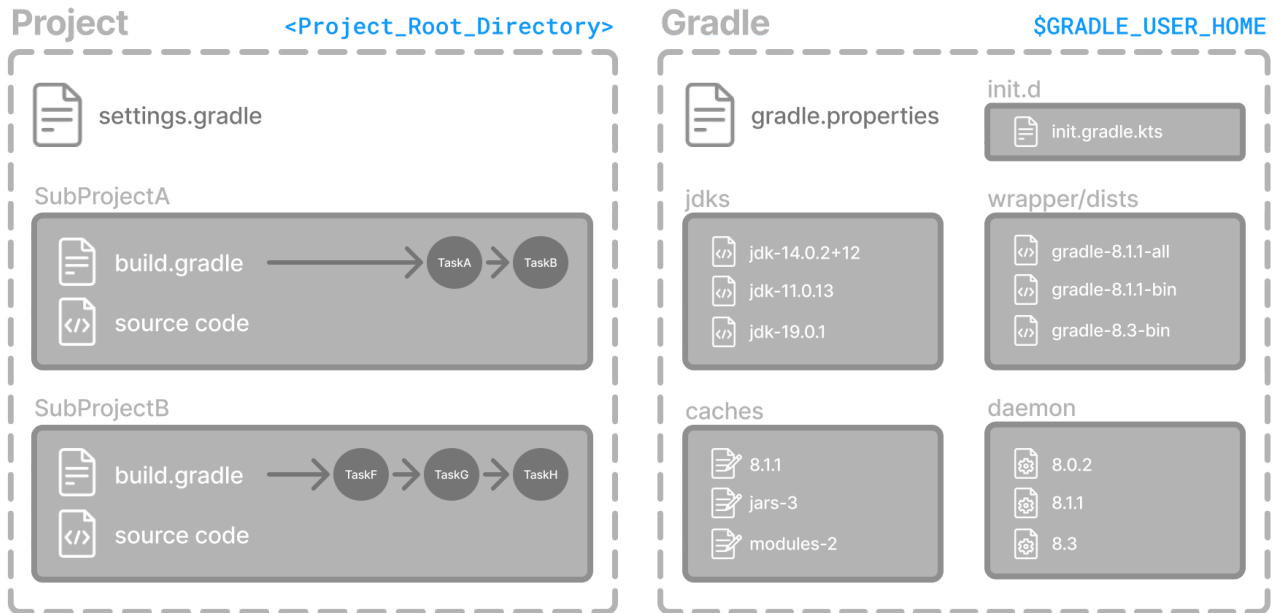
Next Step: [Start the Tutorial >>](#)

AUTHORING GRADLE BUILDS

CORE CONCEPTS

Gradle Directories

Gradle uses two main directories to perform and manage its work: the [Gradle User Home directory](#) and the [Project Root directory](#).



Gradle User Home directory

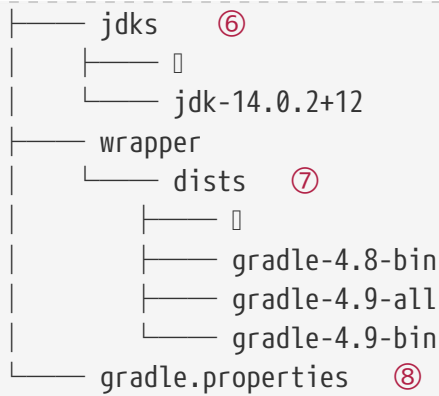
By default, the Gradle User Home (`~/.gradle` or `C:\Users\<USERNAME>\.gradle`) stores global configuration properties, initialization scripts, caches, and log files.

It can be set with the environment variable `GRADLE_USER_HOME`.

TIP Not to be confused with the `GRADLE_HOME`, the optional installation directory for Gradle.

It is roughly structured as follows:

```
├── caches ①
│   ├── 4.8 ②
│   ├── 4.9 ②
│   └── 
│       ├── jars-3 ③
│       └── modules-2 ③
├── daemon ④
│   ├── 
│   ├── 4.8
│   └── 4.9
├── init.d ⑤
└── my-setup.gradle
```



- ① Global cache directory (for everything that is not project-specific).
- ② Version-specific caches (e.g., to support incremental builds).
- ③ Shared caches (e.g., for artifacts of dependencies).
- ④ Registry and logs of the [Gradle Daemon](#).
- ⑤ Global [initialization scripts](#).
- ⑥ JDKs downloaded by the [toolchain support](#).
- ⑦ Distributions downloaded by the [Gradle Wrapper](#).
- ⑧ Global [Gradle configuration properties](#).

Consult the [Gradle Directories reference](#) to learn more.

Project Root directory

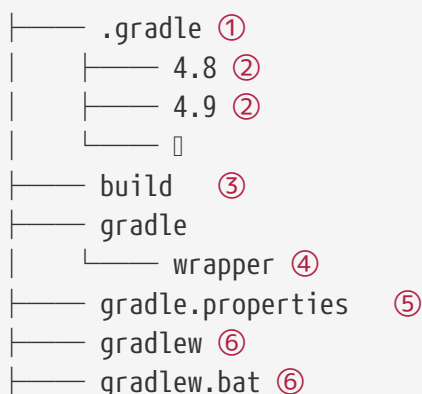
The project root directory contains all source files from your project.

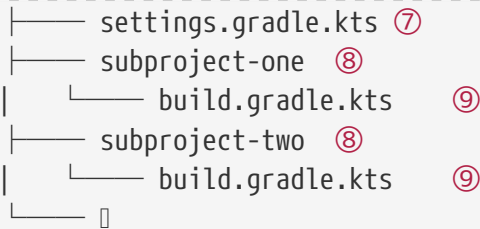
It also contains files and directories Gradle generates, such as `.gradle` and `build`, as well as the Gradle configuration directory: `gradle`.

TIP `gradle` and `.gradle` directories are different.

While `gradle` is usually checked into source control, `build` and `.gradle` directories contain the output of your builds, caches, and other transient files Gradle uses to support features like incremental builds.

The anatomy of a typical project root directory looks as follows:





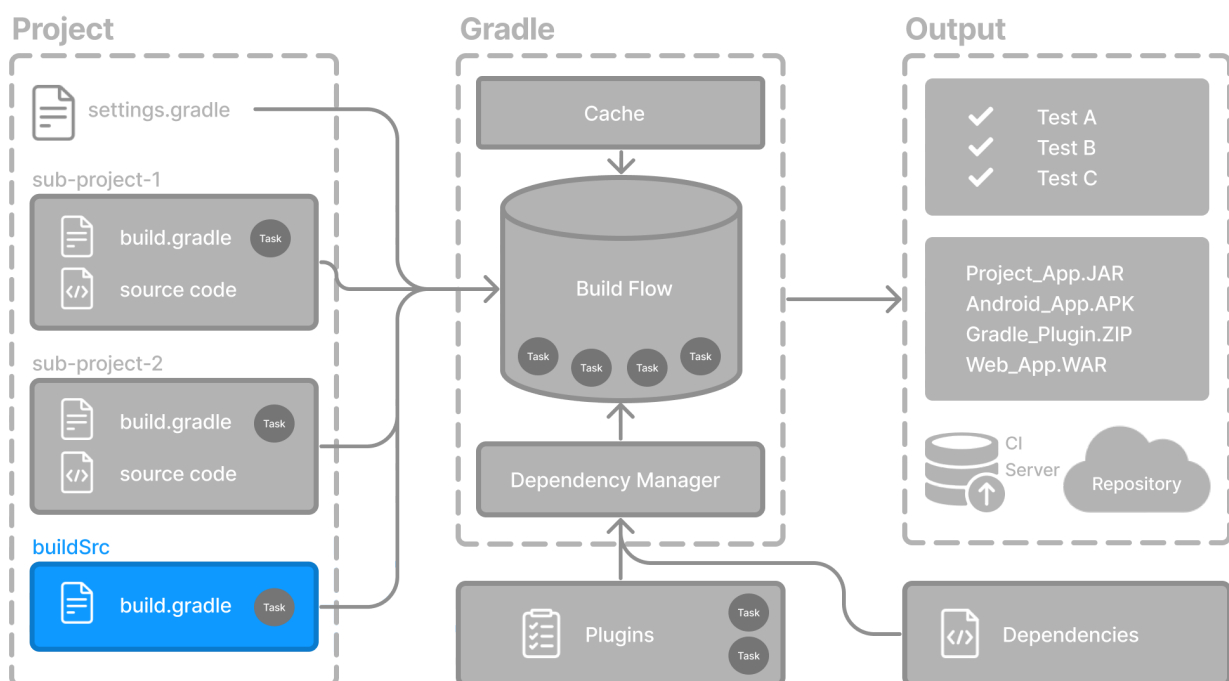
- ① Project-specific cache directory generated by Gradle.
- ② Version-specific caches (e.g., to support incremental builds).
- ③ The build directory of this project into which Gradle generates all build artifacts.
- ④ Contains the JAR file and configuration of the [Gradle Wrapper](#).
- ⑤ Project-specific [Gradle configuration properties](#).
- ⑥ Scripts for executing builds using the [Gradle Wrapper](#).
- ⑦ The project's [settings file](#) where the list of subprojects is defined.
- ⑧ Usually, a project is organized into one or multiple subprojects.
- ⑨ Each subproject has its own Gradle build script.

Consult the [Gradle Directories reference](#) to learn more.

Next Step: [Learn how to structure Multi-Project Builds >>](#)

Multi-Project Build Basics

Gradle supports multi-project builds.



While some small projects and monolithic applications may contain a single build file and source

tree, it is often more common for a project to have been split into smaller, interdependent modules. The word "interdependent" is vital, as you typically want to link the many modules together through a single build.

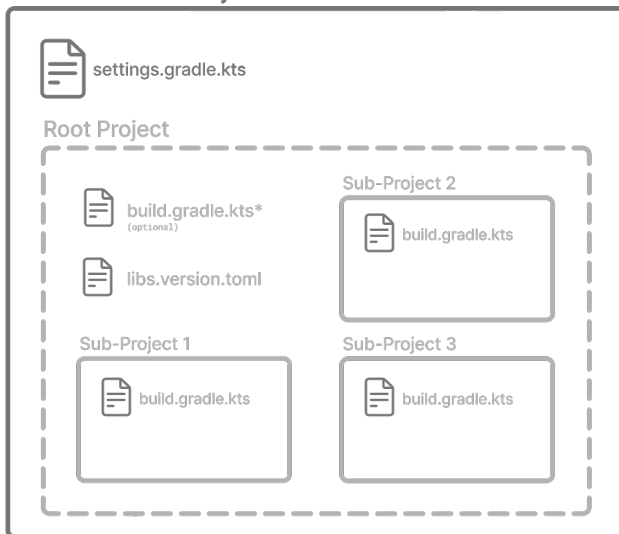
Gradle supports this scenario through *multi-project* builds. This is sometimes referred to as a multi-module project. Gradle refers to modules as subprojects.

A multi-project build consists of one root project and one or more subprojects.

Multi-Project structure

The following represents the structure of a multi-project build that contains three subprojects:

Generic Multi-Project Build:



The directory structure should look as follows:

```
├── .gradle
│   └── []
├── gradle
│   ├── libs.versions.toml
│   └── wrapper
├── gradlew
├── gradlew.bat
├── settings.gradle.kts ①
├── sub-project-1
│   └── build.gradle.kts ②
├── sub-project-2
│   └── build.gradle.kts ②
└── sub-project-3
    └── build.gradle.kts ②
```

① The `settings.gradle.kts` file should include all subprojects.

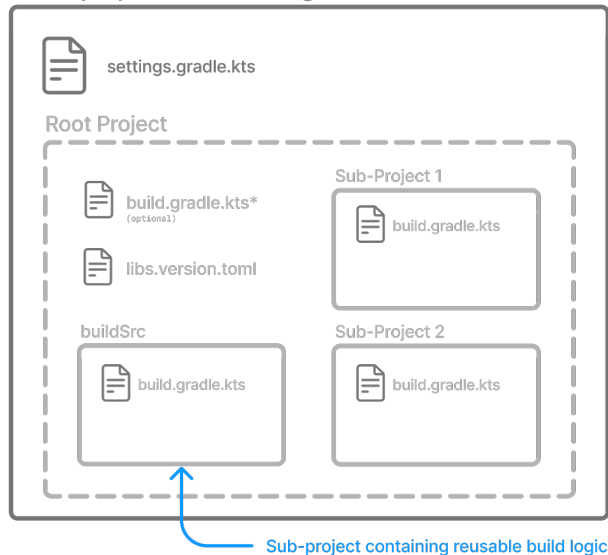
② Each subproject should have its own `build.gradle.kts` file.

Multi-Project standards

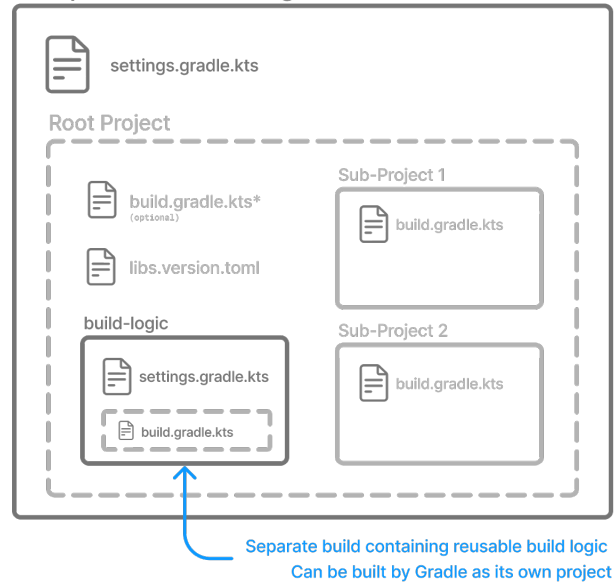
The Gradle community has two standards for multi-project build structures:

1. **Multi-Project Builds using buildSrc** - where **buildSrc** is a subproject-like directory at the Gradle project root containing all the build logic.
2. **Composite Builds** - a build that includes other builds where **build-logic** is a build directory at the Gradle project root containing reusable build logic.

Multi-project Build - using buildSrc:



Composite Build - using includeBuild:



1. Multi-Project Builds using buildSrc

Multi-project builds allow you to organize projects with many modules, wire dependencies between those modules, and easily share common build logic amongst them.

For example, a build that has many modules called **mobile-app**, **web-app**, **api**, **lib**, and **documentation** could be structured as follows:

```
.
├── gradle
├── gradlew
├── settings.gradle.kts
├── buildSrc
│   ├── build.gradle.kts
│   └── src/main/kotlin/shared-build-conventions.gradle.kts
├── mobile-app
│   └── build.gradle.kts
├── web-app
│   └── build.gradle.kts
├── api
│   └── build.gradle.kts
├── lib
│   └── build.gradle.kts
```

```
└── documentation
    └── build.gradle.kts
```

The modules will have dependencies between them such as **web-app** and **mobile-app** depending on **lib**. This means that in order for Gradle to build **web-app** or **mobile-app**, it must build **lib** first.

In this example, the root settings file will look as follows:

settings.gradle.kts

```
include("mobile-app", "web-app", "api", "lib", "documentation")
```

settings.gradle

```
include("mobile-app", "web-app", "api", "lib", "documentation")
```

NOTE The order in which the subprojects (modules) are included does not matter.

The **buildSrc** directory is automatically recognized by Gradle. It is a good place to define and maintain shared configuration or imperative build logic, such as custom tasks or plugins.

buildSrc is automatically included in your build as a special subproject if a **build.gradle(.kts)** file is found under **buildSrc**.

If the **java** plugin is applied to the **buildSrc** project, the compiled code from **buildSrc/src/main/java** is put in the classpath of the root build script, making it available to any subproject (**web-app**, **mobile-app**, **lib**, etc...) in the build.

Consult how to declare [dependencies between subprojects](#) to learn more.

2. Composite Builds

Composite Builds, also referred to as *included builds*, are best for sharing logic between builds (*not subprojects*) or isolating access to shared build logic (i.e., convention plugins).

Let's take the previous example. The logic in **buildSrc** has been turned into a project that contains plugins and can be published and worked on independently of the root project build.

The plugin is moved to its own build called **build-logic** with a build script and settings file:

```
├── .
│   ├── gradle
│   ├── gradlew
│   └── settings.gradle.kts
```

```

├── build-logic
│   ├── settings.gradle.kts
│   └── conventions
│       ├── build.gradle.kts
│       └── src/main/kotlin/shared-build-conventions.gradle.kts
├── mobile-app
│   └── build.gradle.kts
├── web-app
│   └── build.gradle.kts
├── api
│   └── build.gradle.kts
├── lib
│   └── build.gradle.kts
└── documentation
    └── build.gradle.kts

```

NOTE

The fact that **build-logic** is located in a subdirectory of the root project is irrelevant. The folder could be located outside the root project if desired.

The root settings file includes the entire **build-logic** **build**:

settings.gradle.kts

```

pluginManagement {
    includeBuild("build-logic")
}
include("mobile-app", "web-app", "api", "lib", "documentation")

```

Consult how to [create composite builds](#) with `includeBuild` to learn more.

Multi-Project path

A project path has the following pattern: it starts with an optional colon, which denotes the root project.

The root project, `:`, is the only project in a path not specified by its name.

The rest of a project path is a colon-separated sequence of project names, where the next project is a subproject of the previous project:

```

:sub-project-1

```

You can see the project paths when running **gradle projects**:

```

-----
Root project 'project'
-----

```

```
Root project 'project'
+--- Project ':sub-project-1'
\--- Project ':sub-project-2'
```

Project paths usually reflect the filesystem layout, but there are exceptions. Most notably for [composite builds](#).

Identifying project structure

You can use the `gradle projects` command to identify the project structure.

As an example, let's use a multi-project build with the following structure:

```
$ gradle -q projects

Projects:

-----
Root project 'multiproject'
-----

Root project 'multiproject'
+--- Project ':api'
+--- Project ':services'
|   +--- Project ':services:shared'
|   \--- Project ':services:webservice'
\--- Project ':shared'

To see a list of the tasks of a project, run gradle <project-path>:tasks
For example, try running gradle :api:tasks
```

Multi-project builds are collections of tasks you can run. The difference is that you may want to control *which* project's tasks get executed.

The following sections will cover your two options for executing tasks in a multi-project build.

Executing tasks by name

The command `gradle test` will execute the `test` task in any subprojects relative to the current working directory that has that task.

If you run the command from the root project directory, you will run `test` in `api`, `shared`, `services:shared` and `services:webservice`.

If you run the command from the `services` project directory, you will only execute the task in `services:shared` and `services:webservice`.

The basic rule behind Gradle's behavior is to **execute all tasks down the hierarchy with *this* name**. And **complain if there is *no* such task found** in any of the subprojects traversed.

NOTE

Some task selectors, like **help** or **dependencies**, will only run the task on the project they are invoked on and not on all the subprojects to reduce the amount of information printed on the screen.

Executing tasks by fully qualified name

You can use a task's fully qualified name to execute a specific task in a particular subproject. For example: **gradle :services:webservice:build** will run the **build** task of the *webservice* subproject.

The fully qualified name of a task is its **project path** plus the task name.

This approach works for any task, so if you want to know what tasks are in a particular subproject, use the **tasks** task, e.g. **gradle :services:webservice:tasks**.

Multi-Project building and testing

The **build** task is typically used to compile, test, and check a single project.

In multi-project builds, you may often want to do all of these tasks across various projects. The **buildNeeded** and **buildDependents** tasks can help with this.

In [this example](#), the **:services:person-service** project depends on both the **:api** and **:shared** projects. The **:api** project also depends on the **:shared** project.

Assuming you are working on a single project, the **:api** project, you have been making changes but have not built the entire project since performing a **clean**. You want to build any necessary supporting JARs but only perform code quality and unit tests on the parts of the project you have changed.

The **build** task does this:

```
$ gradle :api:build
> Task :shared:compileJava
> Task :shared:processResources
> Task :shared:classes
> Task :shared:jar
> Task :api:compileJava
> Task :api:processResources
> Task :api:classes
> Task :api:jar
> Task :api:assemble
> Task :api:compileTestJava
> Task :api:processTestResources
> Task :api:testClasses
> Task :api:test
> Task :api:check
> Task :api:build
```

BUILD SUCCESSFUL in 0s

If you have just gotten the latest version of the source from your version control system, which included changes in other projects that `:api` depends on, you might want to build all the projects you depend on AND test them too.

The `buildNeeded` task builds AND tests all the projects from the project dependencies of the `testRuntime` configuration:

```
$ gradle :api:buildNeeded
> Task :shared:compileJava
> Task :shared:processResources
> Task :shared:classes
> Task :shared:jar
> Task :api:compileJava
> Task :api:processResources
> Task :api:classes
> Task :api:jar
> Task :api:assemble
> Task :api:compileTestJava
> Task :api:processTestResources
> Task :api:testClasses
> Task :api:test
> Task :api:check
> Task :api:build
> Task :shared:assemble
> Task :shared:compileTestJava
> Task :shared:processTestResources
> Task :shared:testClasses
> Task :shared:test
> Task :shared:check
> Task :shared:build
> Task :shared:buildNeeded
> Task :api:buildNeeded
```

BUILD SUCCESSFUL in 0s

You may want to refactor some part of the `:api` project used in other projects. If you make these changes, testing only the `:api` project is insufficient. You must test all projects that depend on the `:api` project.

The `buildDependents` task tests ALL the projects that have a project dependency (in the `testRuntime` configuration) on the specified project:

```
$ gradle :api:buildDependents
> Task :shared:compileJava
> Task :shared:processResources
> Task :shared:classes
> Task :shared:jar
> Task :api:compileJava
> Task :api:processResources
```

```
> Task :api:classes
> Task :api:jar
> Task :api:assemble
> Task :api:compileTestJava
> Task :api:processTestResources
> Task :api:testClasses
> Task :api:test
> Task :api:check
> Task :api:build
> Task :services:person-service:compileJava
> Task :services:person-service:processResources
> Task :services:person-service:classes
> Task :services:person-service:jar
> Task :services:person-service:assemble
> Task :services:person-service:compileTestJava
> Task :services:person-service:processTestResources
> Task :services:person-service:testClasses
> Task :services:person-service:test
> Task :services:person-service:check
> Task :services:person-service:build
> Task :services:person-service:buildDependents
> Task :api:buildDependents
```

BUILD SUCCESSFUL in 0s

Finally, you can build and test everything in all projects. Any task you run in the root project folder will cause that same-named task to be run on all the children.

You can run `gradle build` to build and test ALL projects.

Consult the [Structuring Builds chapter](#) to learn more.

Next Step: [Learn about the Gradle Build Lifecycle >>](#)

Build Lifecycle

As a build author, you define tasks and specify dependencies between them. Gradle guarantees that tasks will execute in the order dictated by these dependencies.

Your build scripts and plugins configure this task dependency graph.

For example, if your project includes tasks such as `build`, `assemble`, and `createDocs`, you can configure the build script so that they are executed in the order: `build` → `assemble` → `createDocs`.

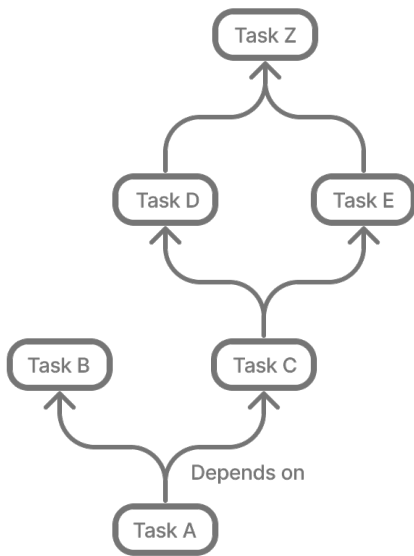
Task Graphs

Gradle builds the task graph **before** executing any task.

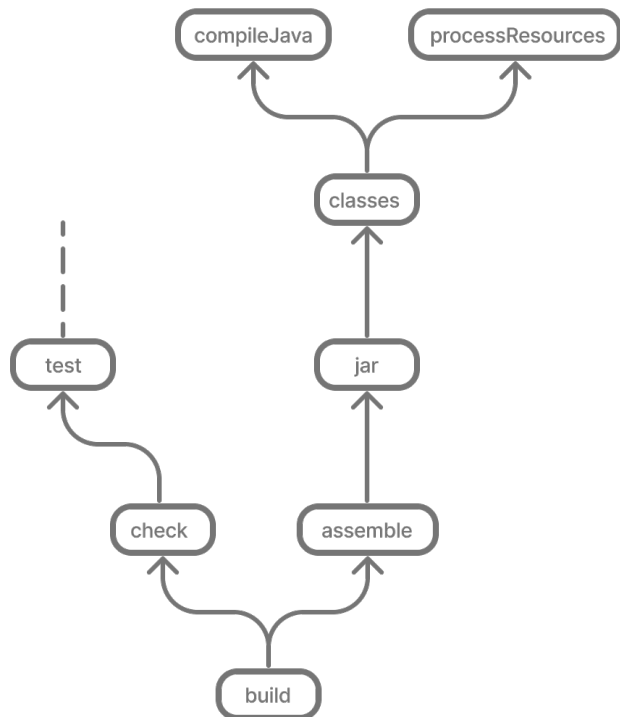
Across all projects in the build, tasks form a [Directed Acyclic Graph](#) (DAG).

This diagram shows two example task graphs, one abstract and the other concrete, with dependencies between tasks represented as arrows:

Generic task graph



Partial task graph for a standard Java build



Both plugins and build scripts contribute to the task graph via the [task dependency mechanism](#) and [annotated inputs/outputs](#).

Build Phases

A Gradle build has three distinct phases.



Gradle runs these phases in order:

Phase 1. Initialization

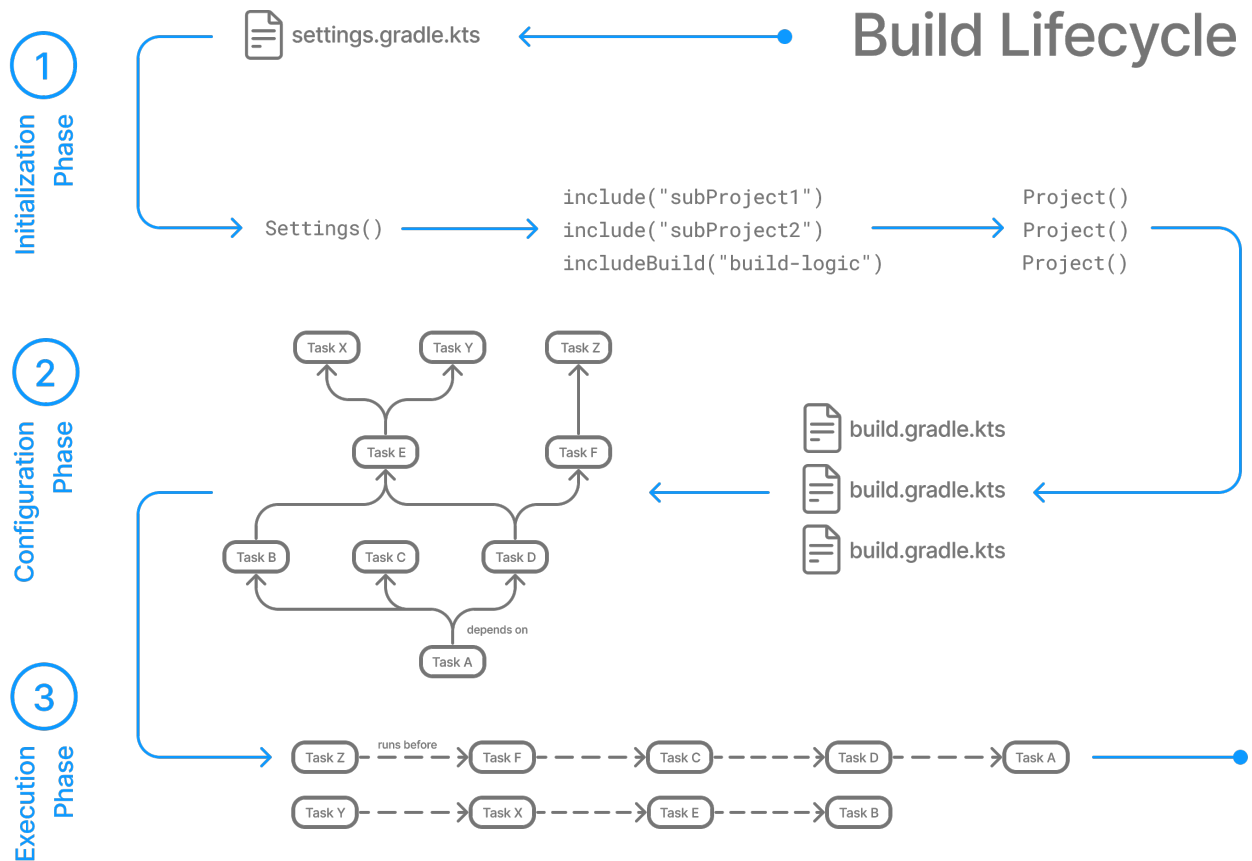
- Detects the `settings.gradle(.kts)` file.
- Creates a `Settings` instance.
- Evaluates the settings file to determine which projects (and included builds) make up the build.
- Creates a `Project` instance for every project.

Phase 2. Configuration

- Evaluates the build scripts, `build.gradle(.kts)`, of every project participating in the build.
- Creates a task graph for requested tasks.

Phase 3. Execution

- Schedules and executes the selected tasks.
- Dependencies between tasks determine execution order.
- Execution of tasks can occur in parallel.



Example

The following example shows which parts of settings and build files correspond to various build phases:

settings.gradle.kts

```
rootProject.name = "basic"
println("This is executed during the initialization phase.")
```

build.gradle.kts

```
println("This is executed during the configuration phase.")
```

```

tasks.register("configured") {
    println("This is also executed during the configuration phase, because
:configured is used in the build.")
}

tasks.register("test") {
    doLast {
        println("This is executed during the execution phase.")
    }
}

tasks.register("testBoth") {
    doFirst {
        println("This is executed first during the execution phase.")
    }
    doLast {
        println("This is executed last during the execution phase.")
    }
    println("This is executed during the configuration phase as well, because
:testBoth is used in the build.")
}

```

settings.gradle

```

rootProject.name = 'basic'
println 'This is executed during the initialization phase.'

```

build.gradle

```

println 'This is executed during the configuration phase.'

tasks.register('configured') {
    println 'This is also executed during the configuration phase, because
:configured is used in the build.'
}

tasks.register('test') {
    doLast {
        println 'This is executed during the execution phase.'
    }
}

tasks.register('testBoth') {
    doFirst {
        println 'This is executed first during the execution phase.'
    }
    doLast {

```

```
        println 'This is executed last during the execution phase.'
    }
    println 'This is executed during the configuration phase as well, because
:testBoth is used in the build.'
}
```

The following command executes the `test` and `testBoth` tasks specified above. Because Gradle only configures requested tasks and their dependencies, the `configured` task never configures:

```
> gradle test testBoth
This is executed during the initialization phase.

> Configure project :
This is executed during the configuration phase.
This is executed during the configuration phase as well, because :testBoth is used in
the build.

> Task :test
This is executed during the execution phase.

> Task :testBoth
This is executed first during the execution phase.
This is executed last during the execution phase.

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

```
> gradle test testBoth
This is executed during the initialization phase.

> Configure project :
This is executed during the configuration phase.
This is executed during the configuration phase as well, because :testBoth is used in
the build.

> Task :test
This is executed during the execution phase.

> Task :testBoth
This is executed first during the execution phase.
This is executed last during the execution phase.

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

Phase 1. Initialization

In the **initialization phase**, Gradle detects the set of projects (root and subprojects) and included builds participating in the build.

Gradle first evaluates the settings file, `settings.gradle(.kts)`, and instantiates a `Settings` object. Then, Gradle instantiates `Project` instances for each project.

Phase 2. Configuration

In the **configuration phase**, Gradle adds tasks and other properties to the projects found by the initialization phase.

Phase 3. Execution

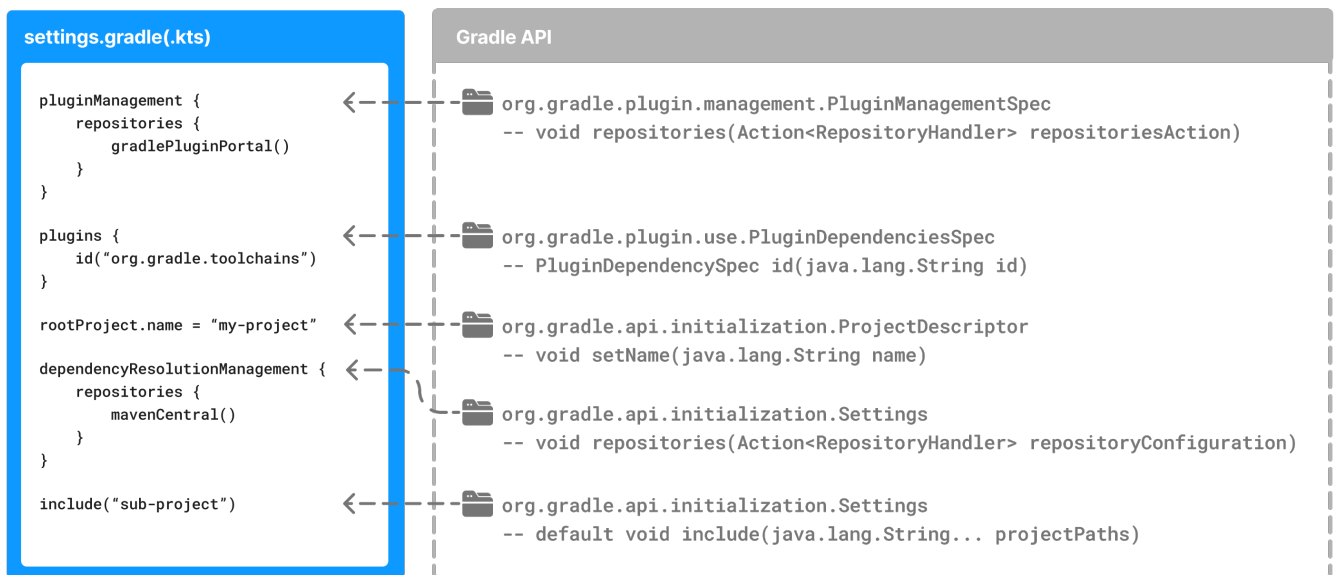
In the **execution phase**, Gradle runs tasks.

Gradle uses the task execution graphs generated by the configuration phase to determine which tasks to execute.

Next Step: [Learn how to write Settings files >>](#)

Writing Settings Files

The settings file is the entry point of every Gradle build.



Early in the Gradle Build lifecycle, the **initialization phase** finds the settings file in your **project root directory**.

When the settings file `settings.gradle(.kts)` is found, Gradle instantiates a `Settings` object.

One of the purposes of the `Settings` object is to allow you to declare all the projects to be included in the build.

Settings Scripts

The settings script is either a `settings.gradle` file in Groovy or a `settings.gradle.kts` file in Kotlin.

Before Gradle assembles the projects for a build, it creates a `Settings` instance and executes the settings file against it.



`settings.gradle(.kts)` \longrightarrow `Settings()`

As the settings script executes, it configures this `Settings`. Therefore, the *settings file* defines the `Settings` object.

IMPORTANT

There is a one-to-one correspondence between a `Settings` instance and a `settings.gradle(.kts)` file.

The `Settings` Object

The `Settings` object is part of the [Gradle API](#).

- In the Groovy DSL, the `Settings` object documentation is found [here](#).
- In the Kotlin DSL, the `Settings` object documentation is found [here](#).

Many top-level properties and blocks in a settings script are part of the Settings API.

For example, we can set the root project name in the settings script using the `Settings.rootProject` property:

```
settings.rootProject.name = "application"
```

Which is usually shortened to:

settings.gradle.kts

```
rootProject.name = "application"
```

settings.gradle

```
rootProject.name = 'application'
```

Standard `Settings` properties

The `Settings` object exposes a standard set of properties in your settings script.

The following table lists a few commonly used properties:

Name	Description
<code>buildCache</code>	The build cache configuration.
<code>plugins</code>	The container of plugins that have been applied to the settings.
<code>rootDir</code>	The root directory of the build. The root directory is the project directory of the root project.
<code>rootProject</code>	The root project of the build.
<code>settings</code>	Returns this settings object.

The following table lists a few commonly used methods:

Name	Description
<code>include()</code>	Adds the given projects to the build.
<code>includeBuild()</code>	Includes a build at the specified path to the composite build.

Settings Script structure

A Settings script is a series of method calls to the Gradle API that often use `{ ... }`, a special shortcut in both the Groovy and Kotlin languages. A `{ }` block is called a *lambda* in Kotlin or a *closure* in Groovy.

Simply put, the `plugins{ }` block is a method invocation in which a Kotlin *lambda* object or Groovy *closure* object is passed as the argument. It is the short form for:

```
plugins(function() {  
    id("plugin")  
})
```

Blocks are mapped to Gradle API methods.

The code inside the function is executed against a `this` object called a *receiver* in Kotlin lambda and a *delegate* in Groovy closure. Gradle determines the correct `this` object and invokes the correct corresponding method. The `this` of the method invocation `id("plugin")` object is of type `PluginDependenciesSpec`.

The settings file is composed of Gradle API calls built on top of the DSLs. Gradle executes the script line by line, top to bottom.

Let's take a look at an example and break it down:

settings.gradle.kts

```
pluginManagement { ①
    repositories {
        gradlePluginPortal()
    }
}

plugins { ②
    id("org.gradle.toolchains.foojay-resolver-convention") version "0.9.0"
}

rootProject.name = "simple-project" ③

dependencyResolutionManagement { ④
    repositories {
        mavenCentral()
    }
}

include("sub-project-a") ⑤
include("sub-project-b")
include("sub-project-c")
```

settings.gradle

```
pluginManagement { ①
    repositories {
        gradlePluginPortal()
    }
}

plugins { ②
    id("org.gradle.toolchains.foojay-resolver-convention") version "0.9.0"
}

rootProject.name = 'simple-project' ③

dependencyResolutionManagement { ④
    repositories {
        mavenCentral()
    }
}

include("sub-project-a") ⑤
include("sub-project-b")
```

```
include("sub-project-c")
```

- ① Define the location of plugins
- ② Apply settings plugins.
- ③ Define the root project name.
- ④ Define dependency resolution strategies.
- ⑤ Add subprojects to the build.

1. Define the location of plugins

The settings file can manage plugin versions and repositories for your build using the `pluginManagement` block. It provides a way to define which plugins should be used in your project and from which repositories they should be resolved.

settings.gradle.kts

```
pluginManagement { ①
    repositories {
        gradlePluginPortal()
    }
}
```

settings.gradle

```
pluginManagement { ①
    repositories {
        gradlePluginPortal()
    }
}
```

2. Apply settings plugins

The settings file can optionally [apply plugins](#) that are required for configuring the settings of the project. These are commonly the [Develocity plugin](#) and the [Toolchain Resolver plugin](#) in the example below.

Plugins applied in the settings file only affect the `Settings` object.

settings.gradle.kts

```
plugins { ②  
    id("org.gradle.toolchains.foojay-resolver-convention") version "0.9.0"  
}
```

settings.gradle

```
plugins { ②  
    id("org.gradle.toolchains.foojay-resolver-convention") version "0.9.0"  
}
```

3. Define the root project name

The settings file defines your project name using the `rootProject.name` property:

settings.gradle.kts

```
rootProject.name = "simple-project" ③
```

settings.gradle

```
rootProject.name = 'simple-project' ③
```

There is only one root project per build.

4. Define dependency resolution strategies

The settings file can optionally [define rules and configurations](#) for dependency resolution across your project(s). It provides a centralized way to manage and customize dependency resolution.

settings.gradle.kts

```
dependencyResolutionManagement { ④  
    repositories {  
        mavenCentral()  
    }  
}
```

```
}
```

settings.gradle

```
dependencyResolutionManagement { ④
    repositories {
        mavenCentral()
    }
}
```

You can also include version catalogs in this section.

5. Add subprojects to the build

The settings file defines the structure of the project by adding all the subprojects using the **include** statement:

settings.gradle.kts

```
include("sub-project-a") ⑤
include("sub-project-b")
include("sub-project-c")
```

settings.gradle

```
include("sub-project-a") ⑤
include("sub-project-b")
include("sub-project-c")
```

You can also include entire builds using **includeBuild**.

Settings File Scripting

There are many more properties and methods on the **Settings** object that you can use to configure your build.

It's important to remember that while many Gradle scripts are typically written in short Groovy or Kotlin syntax, every item in the settings script is essentially invoking a method on the **Settings** object in the Gradle API:

```
include("app")
```

Is actually:

```
settings.include("app")
```

Additionally, the full power of the Groovy and Kotlin languages is available to you.

For example, instead of using **include** many times to add subprojects, you can iterate over the list of directories in the project root folder and include them automatically:

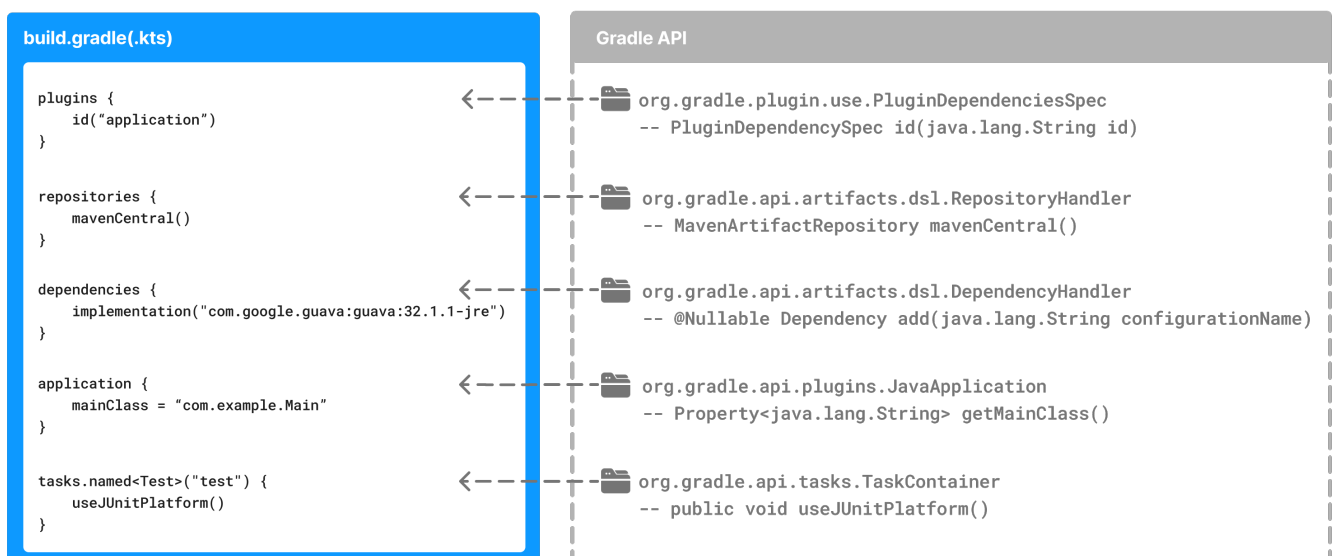
```
rootDir.listFiles().filter { it.isDirectory && (new File(it,
"build.gradle.kts").exists()) }.forEach {
    include(it.name)
}
```

TIP | This type of logic should be developed in a plugin.

Next Step: [Learn how to write Build scripts >>](#)

Writing Build Scripts

The initialization phase in the Gradle Build lifecycle finds the root project and subprojects included in your [project root directory](#) using the settings file.



Then, for each project included in the settings file, Gradle creates a **Project** instance.

Gradle then looks for a corresponding build script file, which is used in the configuration phase.

Build Scripts

Every Gradle build comprises one or more **projects**; a *root* project and *subprojects*.

A project typically corresponds to a software component that needs to be built, like a library or an application. It might represent a library JAR, a web application, or a distribution ZIP assembled from the JARs produced by other projects.

On the other hand, it might represent a thing to be done, such as deploying your application to staging or production environments.

Gradle scripts are written in either Groovy DSL or Kotlin DSL (domain-specific language).

A **build script** configures a **project** and is associated with an object of type **Project**.



As the build script executes, it configures **Project**.

The build script is either a `*.gradle` file in Groovy or a `*.gradle.kts` file in Kotlin.

IMPORTANT

Build scripts configure **Project** objects and their children.

The **Project** object

The **Project** object is part of the [Gradle API](#):

- In the Groovy DSL, the **Project** object documentation is found [here](#).
- In the Kotlin DSL, the **Project** object documentation is found [here](#).

Many top-level properties and blocks in a build script are part of the Project API.

For example, the following build script uses the `Project.name` property to print the name of the project:

build.gradle.kts

```
println(name)
println(project.name)
```

build.gradle

```
println name
```

```
println project.name
```

```
$ gradle -q check
project-api
project-api
```

Both `println` statements print out the same property.

The first uses the top-level reference to the `name` property of the `Project` object. The second statement uses the `project` property available to any build script, which returns the associated `Project` object.

Standard project properties

The `Project` object exposes a standard set of properties in your build script.

The following table lists a few commonly used properties:

Name	Type	Description
<code>name</code>	<code>String</code>	The name of the project directory.
<code>path</code>	<code>String</code>	The fully qualified name of the project.
<code>description</code>	<code>String</code>	A description for the project.
<code>dependencies</code>	<code>DependencyHandler</code>	Returns the dependency handler of the project.
<code>repositories</code>	<code>RepositoryHandler</code>	Returns the repository handler of the project.
<code>layout</code>	<code>ProjectLayout</code>	Provides access to several important locations for a project.
<code>group</code>	<code>Object</code>	The group of this project.
<code>version</code>	<code>Object</code>	The version of this project.

The following table lists a few commonly used methods:

Name	Description
<code>uri()</code>	Resolves a file path to a URI, relative to the project directory of this project.
<code>task()</code>	Creates a Task with the given name and adds it to this project.

Build Script structure

The Build script is composed of `{ ... }`, a special object in both Groovy and Kotlin. This object is called a *lambda* in Kotlin or a *closure* in Groovy.

Simply put, the `plugins{ }` block is a method invocation in which a Kotlin *lambda* object or Groovy *closure* object is passed as the argument. It is the short form for:

```
plugins(function() {  
    id("plugin")  
})
```

Blocks are mapped to Gradle API methods.

The code inside the function is executed against a **this** object called a *receiver* in Kotlin lambda and a *delegate* in Groovy closure. Gradle determines the correct **this** object and invokes the correct corresponding method. The **this** of the method invocation `id("plugin")` object is of type `PluginDependenciesSpec`.

The build script is essentially composed of Gradle API calls built on top of the DSLs. Gradle executes the script line by line, top to bottom.

Let's take a look at an example and break it down:

build.gradle.kts

```
plugins { ①  
    id("application")  
}  
  
repositories { ②  
    mavenCentral()  
}  
  
dependencies { ③  
    testImplementation("org.junit.jupiter:junit-jupiter-engine:5.9.3")  
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")  
    implementation("com.google.guava:guava:32.1.1-jre")  
}  
  
application { ④  
    mainClass = "com.example.Main"  
}  
  
tasks.named<Test>("test") { ⑤  
    useJUnitPlatform()  
}  
  
tasks.named<Javadoc>("javadoc").configure {  
    exclude("app/Internal*.java")  
    exclude("app/internal/*")  
}  
  
tasks.register<Zip>("zip-reports") {  
    from("Reports/")  
    include("*")  
}
```

```
archiveFileName.set("Reports.zip")
destinationDirectory.set(file("/dir"))
}
```

build.gradle

```
plugins { ①
    id 'application'
}

repositories { ②
    mavenCentral()
}

dependencies { ③
    testImplementation 'org.junit.jupiter:junit-jupiter-engine:5.9.3'
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
    implementation 'com.google.guava:guava:32.1.1-jre'
}

application { ④
    mainClass = 'com.example.Main'
}

tasks.named('test', Test) { ⑤
    useJUnitPlatform()
}

tasks.named('javadoc', Javadoc).configure {
    exclude 'app/Internal*.java'
    exclude 'app/internal/*'
}

tasks.register('zip-reports', Zip) {
    from 'Reports/'
    include '*'
    archiveFileName = 'Reports.zip'
    destinationDirectory = file('/dir')
}
```

- ① Apply plugins to the build.
- ② Define the locations where dependencies can be found.
- ③ Add dependencies.
- ④ Set properties.

⑤ Register and configure tasks.

1. Apply plugins to the build

Plugins are used to extend Gradle. They are also used to modularize and reuse project configurations.

Plugins can be applied using the `PluginDependenciesSpec` `plugins` script block.

The `plugins` block is preferred:

build.gradle.kts

```
plugins { ①  
    id("application")  
}
```

build.gradle

```
plugins { ①  
    id 'application'  
}
```

In the example, the `application` plugin, which is included with Gradle, has been applied, describing our project as a Java application.

2. Define the locations where dependencies can be found

A project generally has a number of dependencies it needs to do its work. Dependencies include plugins, libraries, or components that Gradle must download for the build to succeed.

The build script lets Gradle know where to look for the binaries of the dependencies. More than one location can be provided:

build.gradle.kts

```
repositories { ②  
    mavenCentral()  
}
```

build.gradle

```
repositories { ②  
    mavenCentral()  
}
```

In the example, the **guava** library and the JetBrains Kotlin plugin (**org.jetbrains.kotlin.jvm**) will be downloaded from the [Maven Central Repository](#).

3. Add dependencies

A project generally has a number of dependencies it needs to do its work. These dependencies are often libraries of precompiled classes that are imported in the project's source code.

Dependencies are managed via [configurations](#) and are retrieved from repositories.

Use the **DependencyHandler** returned by **Project.getDependencies()** method to manage the dependencies. Use the **RepositoryHandler** returned by **Project.getRepositories()** method to manage the repositories.

build.gradle.kts

```
dependencies { ③  
    testImplementation("org.junit.jupiter:junit-jupiter-engine:5.9.3")  
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")  
    implementation("com.google.guava:guava:32.1.1-jre")  
}
```

build.gradle

```
dependencies { ③  
    testImplementation 'org.junit.jupiter:junit-jupiter-engine:5.9.3'  
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'  
    implementation 'com.google.guava:guava:32.1.1-jre'  
}
```

In the example, the application code uses Google's **guava** libraries. Guava provides utility methods for collections, caching, primitives support, concurrency, common annotations, string processing, I/O, and validations.

4. Set properties

A plugin can add properties and methods to a project using extensions.

The **Project** object has an associated **ExtensionContainer** object that contains all the settings and properties for the plugins that have been applied to the project.

In the example, the **application** plugin added an **application** property, which is used to detail the main class of our Java application:

build.gradle.kts

```
application { ④  
    mainClass = "com.example.Main"  
}
```

build.gradle

```
application { ④  
    mainClass = 'com.example.Main'  
}
```

5. Register and configure tasks

Tasks perform some basic piece of work, such as compiling classes, or running unit tests, or zipping up a WAR file.

While tasks are typically defined in plugins, you may need to register or configure tasks in build scripts.

Registering a task adds the task to your project.

You can register tasks in a project using the **TaskContainer.register(java.lang.String)** method:

build.gradle.kts

```
tasks.register<Zip>("zip-reports") {  
    from("Reports/")  
    include("*")  
    archiveFileName.set("Reports.zip")  
    destinationDirectory.set(file("/dir"))  
}
```

build.gradle

```
tasks.register('zip-reports', Zip) {  
    from 'Reports/'  
    include '*'  
    archiveFileName = 'Reports.zip'  
    destinationDirectory = file('/dir')  
}
```

You may have seen usage of the `TaskContainer.create(java.lang.String)` method **which should be avoided**.

```
tasks.create<Zip>("zip-reports") { }
```

TIP | `register()`, which enables [task configuration avoidance](#), is preferred over `create()`.

You can locate a task to configure it using the `TaskCollection.named(java.lang.String)` method:

build.gradle.kts

```
tasks.named<Test>("test") { ⑤  
    useJUnitPlatform()  
}
```

build.gradle

```
tasks.named('test', Test) { ⑤  
    useJUnitPlatform()  
}
```

The example below configures the `Javadoc` task to automatically generate HTML documentation from Java code:

build.gradle.kts

```
tasks.named<Javadoc>("javadoc").configure {  
    exclude("app/Internal*.java")  
    exclude("app/internal/*")  
}
```

```
}
```

build.gradle

```
tasks.named('javadoc', Javadoc).configure {  
    exclude 'app/Internal*.java'  
    exclude 'app/internal/*'  
}
```

Build Scripting

A build script is made up of zero or more statements and script blocks:

```
println(project.layout.projectDirectory);
```

Statements can include method calls, property assignments, and local variable definitions:

```
version = '1.0.0.GA'
```

A script block is a method call which takes a closure/lambda as a parameter:

```
configurations {  
}
```

The closure/lambda configures some delegate object as it executes:

```
repositories {  
    google()  
}
```

A build script is also a Groovy or a Kotlin script:

build.gradle.kts

```
tasks.register("upper") {  
    doLast {  
        val someString = "mY_nAmE"  
        println("Original: $someString")  
        println("Upper case: ${someString.toUpperCase()}")  
    }  
}
```

```
}
```

build.gradle

```
tasks.register('upper') {  
    doLast {  
        String someString = 'mY_nAmE'  
        println "Original: $someString"  
        println "Upper case: ${someString.toUpperCase()}"  
    }  
}
```

```
$ gradle -q upper  
Original: mY_nAmE  
Upper case: MY_NAME
```

It can contain elements allowed in a Groovy or Kotlin script, such as method definitions and class definitions:

build.gradle.kts

```
tasks.register("count") {  
    doLast {  
        repeat(4) { print("$it ") }  
    }  
}
```

build.gradle

```
tasks.register('count') {  
    doLast {  
        4.times { print "$it " }  
    }  
}
```

```
$ gradle -q count  
0 1 2 3
```

Flexible task registration

Using the capabilities of the Groovy or Kotlin language, you can register multiple tasks in a loop:

build.gradle.kts

```
repeat(4) { counter ->
    tasks.register("task$counter") {
        doLast {
            println("I'm task number $counter")
        }
    }
}
```

build.gradle

```
4.times { counter ->
    tasks.register("task$counter") {
        doLast {
            println "I'm task number $counter"
        }
    }
}
```

```
$ gradle -q task1
I'm task number 1
```

Gradle Types

In Gradle, **types**, **properties**, and **providers** are foundational for managing and configuring build logic:

- **Types:** Gradle defines types (like `Task`, `Configuration`, `File`, etc.) to represent build components. You can extend these types to create custom tasks or domain objects.
- **Properties:** Gradle properties (e.g., `Property<T>`, `ListProperty<T>`, `SetProperty<T>`) are used for build configuration. They allow lazy evaluation, meaning their values are calculated only when needed, enhancing flexibility and performance.
- **Providers:** A `Provider<T>` represents a value that is computed or retrieved lazily. Providers are often used with properties to defer value computation until necessary. This is especially useful for integrating dynamic, runtime values into your build.

You can learn more about this in [Understanding Gradle Types](#).

Declare Variables

Build scripts can declare two variables: **local variables** and **extra properties**.

Local Variables

Declare local variables with the **val** keyword. Local variables are only visible in the scope where they have been declared. They are a feature of the underlying Kotlin language.

Declare local variables with the **def** keyword. Local variables are only visible in the scope where they have been declared. They are a feature of the underlying Groovy language.

build.gradle.kts

```
val dest = "dest"

tasks.register<Copy>("copy") {
    from("source")
    into(dest)
}
```

build.gradle

```
def dest = 'dest'

tasks.register('copy', Copy) {
    from 'source'
    into dest
}
```

Extra Properties

[Gradle's enhanced objects](#), including projects, tasks, and source sets, can hold user-defined properties.

Add, read, and set extra properties via the owning object's **extra** property. Alternatively, you can access extra properties via Kotlin delegated properties using **by extra**.

Add, read, and set extra properties via the owning object's **ext** property. Alternatively, you can use an **ext** block to add multiple properties simultaneously.

build.gradle.kts

```
plugins {
```

```

        id("java-library")
    }

    val springVersion by extra("3.1.0.RELEASE")
    val emailNotification by extra { "build@master.org" }

    sourceSets.all { extra["purpose"] = null }

    sourceSets {
        main {
            extra["purpose"] = "production"
        }
        test {
            extra["purpose"] = "test"
        }
        create("plugin") {
            extra["purpose"] = "production"
        }
    }

    tasks.register("printProperties") {
        val springVersion = springVersion
        val emailNotification = emailNotification
        val productionSourceSets = provider {
            sourceSets.matching { it.extra["purpose"] == "production" }.map {
it.name }
        }
        doLast {
            println(springVersion)
            println(emailNotification)
            productionSourceSets.get().forEach { println(it) }
        }
    }
}

```

build.gradle

```

plugins {
    id 'java-library'
}

ext {
    springVersion = "3.1.0.RELEASE"
    emailNotification = "build@master.org"
}

sourceSets.all { ext.purpose = null }

sourceSets {
    main {

```

```

        purpose = "production"
    }
    test {
        purpose = "test"
    }
    plugin {
        purpose = "production"
    }
}

tasks.register('printProperties') {
    def springVersion = springVersion
    def emailNotification = emailNotification
    def productionSourceSets = provider {
        sourceSets.matching { it.purpose == "production" }.collect { it.name
    }
    }
    doLast {
        println springVersion
        println emailNotification
        productionSourceSets.get().each { println it }
    }
}

```

```

$ gradle -q printProperties
3.1.0.RELEASE
build@master.org
main
plugin

```

This example adds two extra properties to the `project` object via `by extra`. Additionally, this example adds a property named `purpose` to each source set by setting `extra["purpose"]` to `null`. Once added, you can read and set these properties via `extra`.

This example adds two extra properties to the `project` object via an `ext` block. Additionally, this example adds a property named `purpose` to each source set by setting `ext.purpose` to `null`. Once added, you can read and set all these properties just like predefined ones.

Gradle requires special syntax for adding a property so that it can fail fast. For example, this allows Gradle to recognize when a script attempts to set a property that does not exist. You can access extra properties anywhere where you can access their owning object. This gives extra properties a wider scope than local variables. Subprojects can access extra properties on their parent projects.

For more information about extra properties, see [ExtraPropertiesExtension](#) in the API documentation.

Configure Arbitrary Objects

The example `greet()` task shows an example of arbitrary object configuration:

build.gradle.kts

```
class UserInfo(  
    var name: String? = null,  
    var email: String? = null  
)  
  
tasks.register("greet") {  
    val user = UserInfo().apply {  
        name = "Isaac Newton"  
        email = "isaac@newton.me"  
    }  
    doLast {  
        println(user.name)  
        println(user.email)  
    }  
}
```

build.gradle

```
class UserInfo {  
    String name  
    String email  
}  
  
tasks.register('greet') {  
    def user = configure(new UserInfo()) {  
        name = "Isaac Newton"  
        email = "isaac@newton.me"  
    }  
    doLast {  
        println user.name  
        println user.email  
    }  
}
```

```
$ gradle -q greet  
Isaac Newton  
isaac@newton.me
```

Closure Delegates

Each closure has a **delegate** object. Groovy uses this delegate to look up variable and method references to nonlocal variables and closure parameters. Gradle uses this for *configuration closures*, where the **delegate** object refers to the object being configured.

build.gradle

```
dependencies {
    assert delegate == project.dependencies
    testImplementation('junit:junit:4.13')
    delegate.testImplementation('junit:junit:4.13')
}
```

Default imports

To make build scripts more concise, Gradle automatically adds a set of import statements to scripts.

As a result, instead of writing `throw new org.gradle.api.tasks.StopExecutionException()`, you can write `throw new StopExecutionException()` instead.

Gradle implicitly adds the following imports to each script:

```
import org.gradle.*
import org.gradle.api.*
import org.gradle.api.artifacts.*
import org.gradle.api.artifacts.capability.*
import org.gradle.api.artifacts.component.*
import org.gradle.api.artifacts.dsl.*
import org.gradle.api.artifacts.ivy.*
import org.gradle.api.artifacts.maven.*
import org.gradle.api.artifacts.query.*
import org.gradle.api.artifacts.repositories.*
import org.gradle.api.artifacts.result.*
import org.gradle.api.artifacts.transform.*
import org.gradle.api.artifacts.type.*
import org.gradle.api.artifacts.verifications.*
import org.gradle.api.attributes.*
import org.gradle.api.attributes.java.*
import org.gradle.api.attributes.plugin.*
import org.gradle.api.cache.*
import org.gradle.api.capabilities.*
import org.gradle.api.component.*
import org.gradle.api.configuration.*
import org.gradle.api.credentials.*
import org.gradle.api.distribution.*
import org.gradle.api.distribution.plugins.*
```

```
import org.gradle.api.execution.*
import org.gradle.api.file.*
import org.gradle.api.flow.*
import org.gradle.api.initialization.*
import org.gradle.api.initialization.definition.*
import org.gradle.api.initialization.dsl.*
import org.gradle.api.initialization.resolve.*
import org.gradle.api.invocation.*
import org.gradle.api.java.archives.*
import org.gradle.api.jvm.*
import org.gradle.api.launcher.cli.*
import org.gradle.api.logging.*
import org.gradle.api.logging.configuration.*
import org.gradle.api.model.*
import org.gradle.api.plugins.*
import org.gradle.api.plugins.antlr.*
import org.gradle.api.plugins.catalog.*
import org.gradle.api.plugins.jvm.*
import org.gradle.api.plugins.quality.*
import org.gradle.api.plugins.scala.*
import org.gradle.api.problems.*
import org.gradle.api.project.*
import org.gradle.api.provider.*
import org.gradle.api.publish.*
import org.gradle.api.publish.ivy.*
import org.gradle.api.publish.ivy.plugins.*
import org.gradle.api.publish.ivy.tasks.*
import org.gradle.api.publish.maven.*
import org.gradle.api.publish.maven.plugins.*
import org.gradle.api.publish.maven.tasks.*
import org.gradle.api.publish.plugins.*
import org.gradle.api.publish.tasks.*
import org.gradle.api.reflect.*
import org.gradle.api.reporting.*
import org.gradle.api.reporting.components.*
import org.gradle.api.reporting.dependencies.*
import org.gradle.api.reporting.dependents.*
import org.gradle.api.reporting.model.*
import org.gradle.api.reporting.plugins.*
import org.gradle.api.resources.*
import org.gradle.api.services.*
import org.gradle.api.specs.*
import org.gradle.api.tasks.*
import org.gradle.api.tasks.ant.*
import org.gradle.api.tasks.application.*
import org.gradle.api.tasks.bundling.*
import org.gradle.api.tasks.compile.*
import org.gradle.api.tasks.diagnostics.*
import org.gradle.api.tasks.diagnostics.artifact.transforms.*
import org.gradle.api.tasks.diagnostics.configurations.*
import org.gradle.api.tasks.incremental.*
```

```
import org.gradle.api.tasks.javadoc.*
import org.gradle.api.tasks.options.*
import org.gradle.api.tasks.scala.*
import org.gradle.api.tasks.testing.*
import org.gradle.api.tasks.testing.junit.*
import org.gradle.api.tasks.testing.junitplatform.*
import org.gradle.api.tasks.testing.testng.*
import org.gradle.api.tasks.util.*
import org.gradle.api.tasks.wrapper.*
import org.gradle.api.toolchain.management.*
import org.gradle.authentication.*
import org.gradle.authentication.aws.*
import org.gradle.authentication.http.*
import org.gradle.build.event.*
import org.gradle.buildconfiguration.tasks.*
import org.gradle.buildinit.*
import org.gradle.buildinit.plugins.*
import org.gradle.buildinit.specs.*
import org.gradle.buildinit.tasks.*
import org.gradle.caching.*
import org.gradle.caching.configuration.*
import org.gradle.caching.http.*
import org.gradle.caching.local.*
import org.gradle.concurrent.*
import org.gradle.external.javadoc.*
import org.gradle.ide.visualstudio.*
import org.gradle.ide.visualstudio.plugins.*
import org.gradle.ide.visualstudio.tasks.*
import org.gradle.ide.xcode.*
import org.gradle.ide.xcode.plugins.*
import org.gradle.ide.xcode.tasks.*
import org.gradle.ivy.*
import org.gradle.jvm.*
import org.gradle.jvm.application.scripts.*
import org.gradle.jvm.application.tasks.*
import org.gradle.jvm.tasks.*
import org.gradle.jvm.toolchain.*
import org.gradle.language.*
import org.gradle.language.assembler.*
import org.gradle.language.assembler.plugins.*
import org.gradle.language.assembler.tasks.*
import org.gradle.language.base.*
import org.gradle.language.base.artifact.*
import org.gradle.language.base.compile.*
import org.gradle.language.base.plugins.*
import org.gradle.language.base.sources.*
import org.gradle.language.c.*
import org.gradle.language.c.plugins.*
import org.gradle.language.c.tasks.*
import org.gradle.language.cpp.*
import org.gradle.language.cpp.plugins.*
```

```
import org.gradle.language.cpp.tasks.*
import org.gradle.language.java.artifact.*
import org.gradle.language.jvm.tasks.*
import org.gradle.language.nativeplatform.*
import org.gradle.language.nativeplatform.tasks.*
import org.gradle.language.objectivec.*
import org.gradle.language.objectivec.plugins.*
import org.gradle.language.objectivec.tasks.*
import org.gradle.language.objectivec.cpp.*
import org.gradle.language.objectivec.cpp.plugins.*
import org.gradle.language.objectivec.cpp.tasks.*
import org.gradle.language.plugins.*
import org.gradle.language.rc.*
import org.gradle.language.rc.plugins.*
import org.gradle.language.rc.tasks.*
import org.gradle.language.scala.tasks.*
import org.gradle.language.swift.*
import org.gradle.language.swift.plugins.*
import org.gradle.language.swift.tasks.*
import org.gradle.maven.*
import org.gradle.model.*
import org.gradle.nativeplatform.*
import org.gradle.nativeplatform.platform.*
import org.gradle.nativeplatform.plugins.*
import org.gradle.nativeplatform.tasks.*
import org.gradle.nativeplatform.test.*
import org.gradle.nativeplatform.test.cpp.*
import org.gradle.nativeplatform.test.cpp.plugins.*
import org.gradle.nativeplatform.test.cunit.*
import org.gradle.nativeplatform.test.cunit.plugins.*
import org.gradle.nativeplatform.test.cunit.tasks.*
import org.gradle.nativeplatform.test.googletest.*
import org.gradle.nativeplatform.test.googletest.plugins.*
import org.gradle.nativeplatform.test.plugins.*
import org.gradle.nativeplatform.test.tasks.*
import org.gradle.nativeplatform.test.xctest.*
import org.gradle.nativeplatform.test.xctest.plugins.*
import org.gradle.nativeplatform.test.xctest.tasks.*
import org.gradle.nativeplatform.toolchain.*
import org.gradle.nativeplatform.toolchain.plugins.*
import org.gradle.normalization.*
import org.gradle.platform.*
import org.gradle.platform.base.*
import org.gradle.platform.base.binary.*
import org.gradle.platform.base.component.*
import org.gradle.platform.base.plugins.*
import org.gradle.plugin.devel.*
import org.gradle.plugin.devel.plugins.*
import org.gradle.plugin.devel.tasks.*
import org.gradle.plugin.management.*
import org.gradle.plugin.use.*
```

```
import org.gradle.plugins.ear.*
import org.gradle.plugins.ear.descriptor.*
import org.gradle.plugins.ide.*
import org.gradle.plugins.ide.api.*
import org.gradle.plugins.ide.eclipse.*
import org.gradle.plugins.ide.idea.*
import org.gradle.plugins.signing.*
import org.gradle.plugins.signing.signatory.*
import org.gradle.plugins.signing.signatory.pgp.*
import org.gradle.plugins.signing.type.*
import org.gradle.plugins.signing.type.pgp.*
import org.gradle.process.*
import org.gradle.swiftpm.*
import org.gradle.swiftpm.plugins.*
import org.gradle.swiftpm.tasks.*
import org.gradle.testing.base.*
import org.gradle.testing.base.plugins.*
import org.gradle.testing.jacoco.plugins.*
import org.gradle.testing.jacoco.tasks.*
import org.gradle.testing.jacoco.tasks.rules.*
import org.gradle.testkit.runner.*
import org.gradle.util.*
import org.gradle.vcs.*
import org.gradle.vcs.git.*
import org.gradle.work.*
import org.gradle.workers.*
```

Next Step: [Learn how to use Tasks >>](#)

Using Tasks

The work that Gradle can do on a project is defined by one or more *tasks*.

TaskC

```
tasks.register('generateReport', Task) {
    doLast {
        def file = new File("$buildDir/report.txt")
        file.text = "${results}"
        println "Generated file: ${file.path}"
    }
    dependsOn tasks.assemble
}
```

TaskD

```
tasks.register('docFilesJar', Jar) {
    group = 'documentation'
    description = 'Generate documentation.'
    archiveVersion = null
    archiveFileName = 'doc-files.jar'
    from 'src/main/template'
}

tasks.named('jar', Jar) {
    from docFilesJar
}
```

1. TASK NAME

2. TASK REGISTRATION / ACTION

3. TASK GROUP / DESCRIPTION

4. TASK DEPENDENCY

5. TASK CONFIGURATION

6. TASK TYPE

A task represents some independent unit of work that a build performs. This might be compiling some classes, creating a JAR, generating Javadoc, or publishing some archives to a repository.

When a user runs `./gradlew build` in the command line, Gradle will execute the `build` task along with any other tasks it depends on.

List available tasks

Gradle provides several default tasks for a project, which are listed by running `./gradlew tasks`:

```
> Task :tasks
```

```
-----
Tasks runnable from root project 'myTutorial'
-----
```

```
Build Setup tasks
```

```
-----
init - Initializes a new Gradle build.
wrapper - Generates Gradle wrapper files.
```

```
Help tasks
```

```
-----
buildEnvironment - Displays all buildscript dependencies declared in root project
'myTutorial'.
```

```
...
```

Tasks either come from **build scripts** or **plugins**.

Once we apply a plugin to our project, such as the `application` plugin, additional tasks become available:

build.gradle.kts

```
plugins {  
    id("application")  
}
```

build.gradle

```
plugins {  
    id 'application'  
}
```

```
$ ./gradlew tasks
```

```
> Task :tasks
```

```
-----  
Tasks runnable from project ':app'  
-----
```

```
Application tasks  
-----
```

```
run - Runs this project as a JVM application
```

```
Build tasks  
-----
```

```
assemble - Assembles the outputs of this project.
```

```
build - Assembles and tests this project.
```

```
Documentation tasks  
-----
```

```
javadoc - Generates Javadoc API documentation for the main source code.
```

```
Other tasks  
-----
```

```
compileJava - Compiles main Java source.
```

```
...
```

Many of these tasks, such as `assemble`, `build`, and `run`, should be familiar to a developer.

Task classification

There are two classes of tasks that can be executed:

1. **Actionable tasks** have some action(s) attached to do work in your build: `compileJava`.
2. **Lifecycle tasks** are tasks with no actions attached: `assemble`, `build`.

Typically, a **lifecycle** tasks depends on many **actionable** tasks, and is used to execute many tasks at once.

Task registration and action

Let's take a look at a simple "Hello World" task in a build script:

build.gradle.kts

```
tasks.register("hello") {
    doLast {
        println("Hello world!")
    }
}
```

build.gradle

```
tasks.register('hello') {
    doLast {
        println 'Hello world!'
    }
}
```

In the example, the build script **registers** a single task called `hello` using the [TaskContainer](#) API, and adds an **action** to it.

If the tasks in the project are listed, the `hello` task is available to Gradle:

```
$ ./gradlew app:tasks --all
```

```
> Task :app:tasks
```

```
-----  
Tasks runnable from project ':app'
```

```
-----  
Other tasks
```

```
-----
compileJava - Compiles main Java source.
compileTestJava - Compiles test Java source.
hello
processResources - Processes main resources.
processTestResources - Processes test resources.
startScripts - Creates OS-specific scripts to run the project as a JVM application.
```

You can execute the task in the build script with `./gradlew hello`:

```
$ ./gradlew hello
Hello world!
```

When Gradle executes the `hello` task, it executes the **action** provided. In this case, the action is simply a block containing some code: `println("Hello world!")`.

Task group and description

The `hello` task from the previous section can be detailed with a **description** and assigned to a **group** with the following update:

build.gradle.kts

```
tasks.register("hello") {
    group = "Custom"
    description = "A lovely greeting task."
    doLast {
        println("Hello world!")
    }
}
```

build.gradle

```
tasks.register('hello') {
    group = 'Custom'
    description = 'A lovely greeting task.'
    doLast {
        println 'Hello world!'
    }
}
```

Once the task is assigned to a group, it will be listed by `./gradlew tasks`:

```
$ ./gradlew tasks

> Task :tasks

Custom tasks
-----
hello - A lovely greeting task.
```

To view information about a task, use the `help --task <task-name>` command:

```
$.gradlew help --task hello

> Task :help
Detailed task information for hello

Path
:app:hello

Type
Task (org.gradle.api.Task)

Options
--rerun      Causes the task to be re-run even if up-to-date.

Description
A lovely greeting task.

Group
Custom
```

As we can see, the `hello` task belongs to the `custom` group.

Task dependencies

You can declare tasks that depend on other tasks:

build.gradle.kts

```
tasks.register("hello") {
    doLast {
        println("Hello world!")
    }
}
tasks.register("intro") {
    dependsOn("hello")
    doLast {
        println("I'm Gradle")
    }
}
```

```
}  
}
```

build.gradle

```
tasks.register('hello') {  
    doLast {  
        println 'Hello world!'  
    }  
}  
tasks.register('intro') {  
    dependsOn tasks.hello  
    doLast {  
        println "I'm Gradle"  
    }  
}
```

```
$ gradle -q intro  
Hello world!  
I'm Gradle
```

The dependency of **taskX** to **taskY** may be declared before **taskY** is defined:

build.gradle.kts

```
tasks.register("taskX") {  
    dependsOn("taskY")  
    doLast {  
        println("taskX")  
    }  
}  
tasks.register("taskY") {  
    doLast {  
        println("taskY")  
    }  
}
```

build.gradle

```
tasks.register('taskX') {  
    dependsOn 'taskY'  
    doLast {
```

```

        println 'taskX'
    }
}
tasks.register('taskY') {
    doLast {
        println 'taskY'
    }
}

```

```

$ gradle -q taskX
taskY
taskX

```

The **hello** task from the previous example is updated to include a dependency:

build.gradle.kts

```

tasks.register("hello") {
    group = "Custom"
    description = "A lovely greeting task."
    doLast {
        println("Hello world!")
    }
    dependsOn(tasks.assemble)
}

```

build.gradle

```

tasks.register('hello') {
    group = "Custom"
    description = "A lovely greeting task."
    doLast {
        println("Hello world!")
    }
    dependsOn(tasks.assemble)
}

```

The **hello** task now depends on the **assemble** task, which means that Gradle must execute the **assemble** task **before** it can execute the **hello** task:

```

$ ./gradlew :app:hello

```

```
> Task :app:compileJava UP-TO-DATE
> Task :app:processResources NO-SOURCE
> Task :app:classes UP-TO-DATE
> Task :app:jar UP-TO-DATE
> Task :app:startScripts UP-TO-DATE
> Task :app:distTar UP-TO-DATE
> Task :app:distZip UP-TO-DATE
> Task :app:assemble UP-TO-DATE

> Task :app:hello
Hello world!
```

Task configuration

Once registered, tasks can be accessed via the [TaskProvider](#) API for further configuration.

For instance, you can use this to add dependencies to a task at runtime dynamically:

build.gradle.kts

```
repeat(4) { counter ->
    tasks.register("task$counter") {
        doLast {
            println("I'm task number $counter")
        }
    }
}
tasks.named("task0") { dependsOn("task2", "task3") }
```

build.gradle

```
4.times { counter ->
    tasks.register("task$counter") {
        doLast {
            println "I'm task number $counter"
        }
    }
}
tasks.named('task0') { dependsOn('task2', 'task3') }
```

```
$ gradle -q task0
I'm task number 2
I'm task number 3
```

I'm task number 0

Or you can add behavior to an existing task:

build.gradle.kts

```
tasks.register("hello") {
    doLast {
        println("Hello Earth")
    }
}
tasks.named("hello") {
    doFirst {
        println("Hello Venus")
    }
}
tasks.named("hello") {
    doLast {
        println("Hello Mars")
    }
}
tasks.named("hello") {
    doLast {
        println("Hello Jupiter")
    }
}
```

build.gradle

```
tasks.register('hello') {
    doLast {
        println 'Hello Earth'
    }
}
tasks.named('hello') {
    doFirst {
        println 'Hello Venus'
    }
}
tasks.named('hello') {
    doLast {
        println 'Hello Mars'
    }
}
tasks.named('hello') {
    doLast {
```

```
        println 'Hello Jupiter'
    }
}
```

```
$ gradle -q hello
Hello Venus
Hello Earth
Hello Mars
Hello Jupiter
```

TIP

The calls `doFirst` and `doLast` can be executed multiple times. They add an action to the beginning or the end of the task's actions list. When the task executes, the actions in the action list are executed in order.

Here is an example of the `named` method being used to configure a task added by a plugin:

build.gradle.kts

```
tasks.dokkaHtml.configure {
    outputDirectory.set(buildDir)
}
```

build.gradle

```
tasks.named("dokkaHtml") {
    outputDirectory.set(buildDir)
}
```

Task types

Gradle tasks are a subclass of `Task`.

In the build script, the `HelloTask` class is created by extending `DefaultTask`:

build.gradle.kts

```
// Extend the DefaultTask class to create a HelloTask class
abstract class HelloTask : DefaultTask() {
    @TaskAction
```

```

    fun hello() {
        println("hello from HelloTask")
    }
}

// Register the hello Task with type HelloTask
tasks.register<HelloTask>("hello") {
    group = "Custom tasks"
    description = "A lovely greeting task."
}

```

build.gradle

```

// Extend the DefaultTask class to create a HelloTask class
class HelloTask extends DefaultTask {
    @TaskAction
    void hello() {
        println("hello from HelloTask")
    }
}

// Register the hello Task with type HelloTask
tasks.register("hello", HelloTask) {
    group = "Custom tasks"
    description = "A lovely greeting task."
}

```

The **hello** task is registered with the **type HelloTask**.

Executing our new **hello** task:

```

$ ./gradlew hello

> Task :app:hello
hello from HelloTask

```

Now the **hello** task is of type **HelloTask** instead of type **Task**.

The Gradle **help** task reveals the change:

```

$ ./gradlew help --task hello

> Task :help
Detailed task information for hello

```

Path
:app:hello

Type
HelloTask (Build_gradle\$HelloTask)

Options
--rerun Causes the task to be re-run even if up-to-date.

Description
A lovely greeting task.

Group
Custom tasks

Built-in task types

Gradle provides many built-in task types with common and popular functionality, such as copying or deleting files.

This example task copies `*.war` files from the `source` directory to the `target` directory using the `Copy` built-in task:

build.gradle.kts

```
tasks.register<Copy>("copyTask") {  
    from("source")  
    into("target")  
    include("*.war")  
}
```

build.gradle

```
tasks.register('copyTask', Copy) {  
    from("source")  
    into("target")  
    include("*.war")  
}
```

There are many task types developers can take advantage of, including `GroovyDoc`, `Zip`, `Jar`, `JacocoReport`, `Sign`, or `Delete`, which are available in the link:[DSL](#).

Next Step: [Learn how to write Tasks >>](#)

Writing Tasks

Gradle tasks are created by extending `DefaultTask`.

However, the generic `DefaultTask` provides no action for Gradle. If users want to extend the capabilities of Gradle and their build script, they must either use a **built-in task** or create a **custom task**:

1. **Built-in task** - Gradle provides built-in utility tasks such as `Copy`, `Jar`, `Zip`, `Delete`, etc...
2. **Custom task** - Gradle allows users to subclass `DefaultTask` to create their own task types.

Create a task

The simplest and quickest way to create a **custom** task is in a build script:

To create a task, inherit from the `DefaultTask` class and implement a `@TaskAction` handler:

build.gradle.kts

```
abstract class CreateFileTask : DefaultTask() {
    @TaskAction
    fun action() {
        val file = File("myfile.txt")
        file.createNewFile()
        file.writeText("HELLO FROM MY TASK")
    }
}
```

build.gradle

```
class CreateFileTask extends DefaultTask {
    @TaskAction
    void action() {
        def file = new File("myfile.txt")
        file.createNewFile()
        file.text = "HELLO FROM MY TASK"
    }
}
```

The `CreateFileTask` implements a simple set of **actions**. First, a file called "myfile.txt" is created in the main project. Then, some text is written to the file.

Register a task

A task is **registered** in the build script using the `TaskContainer.register()` method, which allows it to be then used in the build logic.

build.gradle.kts

```
abstract class CreateFileTask : DefaultTask() {
    @TaskAction
    fun action() {
        val file = File("myfile.txt")
        file.createNewFile()
        file.writeText("HELLO FROM MY TASK")
    }
}

tasks.register<CreateFileTask>("createFileTask")
```

build.gradle

```
class CreateFileTask extends DefaultTask {
    @TaskAction
    void action() {
        def file = new File("myfile.txt")
        file.createNewFile()
        file.text = "HELLO FROM MY TASK"
    }
}

tasks.register("createFileTask", CreateFileTask)
```

Task group and description

Setting the **group** and **description** properties on your tasks can help users understand how to use your task:

build.gradle.kts

```
abstract class CreateFileTask : DefaultTask() {
    @TaskAction
    fun action() {
        val file = File("myfile.txt")
        file.createNewFile()
    }
}
```

```

        file.writeText("HELLO FROM MY TASK")
    }
}

tasks.register<CreateFileTask>("createFileTask") {
    group = "custom"
    description = "Create myfile.txt in the current directory"
}

```

build.gradle

```

class CreateFileTask extends DefaultTask {
    @TaskAction
    void action() {
        def file = new File("myfile.txt")
        file.createNewFile()
        file.text = "HELLO FROM MY TASK"
    }
}

tasks.register("createFileTask", CreateFileTask) {
    group = "custom"
    description = "Create myfile.txt in the current directory"
}

```

Once a task is added to a group, it is visible when listing tasks.

Task input and outputs

For the task to do useful work, it typically needs some **inputs**. A task typically produces **outputs**.

build.gradle.kts

```

abstract class CreateAFileTask : DefaultTask() {
    @get:Input
    abstract val fileText: Property<String>

    @Input
    val fileName = "myfile.txt"

    @OutputFile
    val myFile: File = File(fileName)

    @TaskAction
    fun action() {

```

```

        myFile.createNewFile()
        myFile.writeText(fileText.get())
    }
}

```

build.gradle

```

abstract class CreateAFileTask extends DefaultTask {
    @Input
    abstract Property<String> getFileText()

    @Input
    final String fileName = "myfile.txt"

    @OutputFile
    final File myFile = new File(fileName)

    @TaskAction
    void action() {
        myFile.createNewFile()
        myFile.text = fileText.get()
    }
}

```

Configure a task

A task is optionally **configured** in a build script using the `TaskCollection.named()` method.

The `CreateAFileTask` class is updated so that the text in the file is configurable:

build.gradle.kts

```

abstract class CreateAFileTask : DefaultTask() {
    @get:Input
    abstract val fileText: Property<String>

    @Input
    val fileName = "myfile.txt"

    @OutputFile
    val myFile: File = File(fileName)

    @TaskAction
    fun action() {
        myFile.createNewFile()
    }
}

```

```

        myFile.writeText(fileText.get())
    }
}

tasks.register<CreateAFileTask>("createAFileTask") {
    group = "custom"
    description = "Create myfile.txt in the current directory"
    fileText.convention("HELLO FROM THE CREATE FILE TASK METHOD") // Set
convention
}

tasks.named<CreateAFileTask>("createAFileTask") {
    fileText.set("HELLO FROM THE NAMED METHOD") // Override with custom
message
}

```

build.gradle

```

abstract class CreateAFileTask extends DefaultTask {
    @Input
    abstract Property<String> getFileText()

    @Input
    final String fileName = "myfile.txt"

    @OutputFile
    final File myFile = new File(fileName)

    @TaskAction
    void action() {
        myFile.createNewFile()
        myFile.text = fileText.get()
    }
}

tasks.register("createAFileTask", CreateAFileTask) {
    group = "custom"
    description = "Create myfile.txt in the current directory"
    fileText.convention("HELLO FROM THE CREATE FILE TASK METHOD") // Set
convention
}

tasks.named("createAFileTask", CreateAFileTask) {
    fileText.set("HELLO FROM THE NAMED METHOD") // Override with custom
message
}

```

In the `named()` method, we find the `createAFileTask` task and set the text that will be written to the file.

When the task is executed:

```
$ ./gradlew createAFileTask

> Configure project :app

> Task :app:createAFileTask

BUILD SUCCESSFUL in 5s
2 actionable tasks: 1 executed, 1 up-to-date
```

A text file called `myfile.txt` is created in the project root folder:

myfile.txt

```
HELLO FROM THE NAMED METHOD
```

Consult the [Developing Gradle Tasks chapter](#) to learn more.

Next Step: [Learn how to use Plugins >>](#)

Using Plugins

Much of Gradle's functionality is delivered via plugins, including core plugins distributed with Gradle, third-party plugins, and script plugins defined within builds.

Plugins introduce new tasks (e.g., `JavaCompile`), domain objects (e.g., `SourceSet`), conventions (e.g., locating Java source at `src/main/java`), and extend core or other plugin objects.

Plugins in Gradle are essential for automating common build tasks, integrating with external tools or services, and tailoring the build process to meet specific project needs. They also serve as the primary mechanism for organizing build logic.

Benefits of plugins

Writing many tasks and duplicating configuration blocks in build scripts can get messy. Plugins offer several advantages over adding logic directly to the build script:

- **Promotes Reusability:** Reduces the need to duplicate similar logic across projects.
- **Enhances Modularity:** Allows for a more modular and organized build script.
- **Encapsulates Logic:** Keeps imperative logic separate, enabling more declarative build scripts.

Plugin distribution

You can leverage plugins from Gradle and the Gradle community or create your own.

Plugins are available in three ways:

1. **Core plugins** - Gradle develops and maintains a set of [Core Plugins](#).
2. **Community plugins** - Gradle plugins shared in a remote repository such as Maven or the [Gradle Plugin Portal](#).
3. **Custom plugins** - Gradle enables users to create plugins using [APIs](#).

Types of plugins

Plugins can be implemented as **binary plugins**, **precompiled script plugins**, or **script plugins**:

1. Script Plugins

Script plugins are Groovy DSL or Kotlin DSL scripts that are applied directly to a Gradle build script using the `apply from:` syntax. They are applied inline within a build script to add functionality or customize the build process. **They are not recommended** but it's important to understand how to work:

build.gradle.kts

```
// Define a plugin
class HelloWorldPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        project.tasks.register("helloWorld") {
            group = "Example"
            description = "Prints 'Hello, World!' to the console"
            doLast {
                println("Hello, World!")
            }
        }
    }
}

// Apply the plugin
apply<HelloWorldPlugin>()
```

2. Precompiled Script Plugins

Precompiled script plugins are Groovy DSL or Kotlin DSL scripts compiled and distributed as Java class files packaged in some library. They are meant to be consumed as a binary Gradle plugin, so they are applied to a project using the `plugins {}` block. The plugin ID by which the precompiled script can be referenced is derived from its name and optional package declaration.

plugin/src/main/kotlin/my-plugin.gradle.kts

```
// This script is automatically exposed to downstream consumers as the `my-plugin`  
plugin  
tasks {  
    register("myCopyTask", Copy::class) {  
        group = "sample"  
        from("build.gradle.kts")  
        into("build/copy")  
    }  
}
```

consumer/build.gradle.kts

```
plugins {  
    id("my-plugin") version "1.0"  
}
```

3. BuildSrc and Convention Plugins

These are a hybrid of precompiled plugins and binary plugins that provide a way to reuse complex logic across projects and allow for better organization of build logic.

buildSrc/src/main/kotlin/shared-build-conventions.gradle.kts

```
plugins {  
    java  
}  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    testImplementation("org.junit.jupiter:junit-jupiter:5.8.1")  
    implementation("com.google.guava:guava:30.1.1-jre")  
}  
  
tasks.named<Test>("test") {  
    useJUnitPlatform()  
}  
  
tasks.register<Copy>("backupTestXml") {  
    from("build/test-results/test")  
    into("/tmp/results/")  
    exclude("binary/**")  
}
```

app/build.gradle.kts

```
plugins {  
    application  
    id("shared-build-conventions")  
}
```

4. Binary Plugins

Binary plugins are compiled plugins typically written in Java or Kotlin DSL that are packaged as JAR files. They are applied to a project using the `plugins {}` block. They offer better performance and maintainability compared to script plugins or precompiled script plugins.

plugin/src/main/kotlin/plugin/MyPlugin.kt

```
class MyPlugin : Plugin<Project> {  
    override fun apply(project: Project) {  
        project.run {  
            tasks {  
                register("myCopyTask", Copy::class) {  
                    group = "sample"  
                    from("build.gradle.kts")  
                    into("build/copy")  
                }  
            }  
        }  
    }  
}
```

consumer/build.gradle.kts

```
plugins {  
    id("my-plugin") version "1.0"  
}
```

The difference between a **binary plugin** and a **script plugin** lies in how they are shared and executed:

- A **binary plugin** is compiled into bytecode, and the bytecode is shared.
- A **script plugin** is shared as source code, and it is compiled at the time of use.

Binary plugins can be written in any language that produces JVM bytecode, such as Java, Kotlin, or Groovy. In contrast, script plugins can only be written using Kotlin DSL or Groovy DSL.

However, there is also a middle ground: **precompiled script plugins**. These are written in Kotlin DSL or Groovy DSL, like script plugins, but are compiled into bytecode and shared like binary plugins.

A plugin often starts as a script plugin (because they are easy to write). Then, as the code becomes

more valuable, it's migrated to a binary plugin that can be easily tested and shared between multiple projects or organizations.

Using plugins

To use the build logic encapsulated in a plugin, Gradle needs to perform two steps. First, it needs to **resolve** the plugin, and then it needs to **apply** the plugin to the target, usually a **Project**.

1. **Resolving** a plugin means finding the correct version of the JAR that contains a given plugin and adding it to the script classpath. Once a plugin is resolved, its API can be used in a build script. Script plugins are self-resolving in that they are resolved from the specific file path or URL provided when applying them. Core binary plugins provided as part of the Gradle distribution are automatically resolved.
2. **Applying** a plugin means executing the plugin's `Plugin.apply(T)` on a project.

The [plugins DSL](#) is recommended to resolve and apply plugins in one step.

Applying plugins

Gradle provides the **core plugins** (e.g., `JavaPlugin`, `GroovyPlugin`, `MavenPublishPlugin`, etc.) as part of its distribution, which means they are automatically resolved.

Core plugins are applied in a build script using the plugin name:

```
plugins {  
    id «plugin name»  
}
```

For example:

build.gradle.kts

```
plugins {  
    id("java")  
}
```

build.gradle

```
plugins {  
    id 'java'  
}
```

Non-core plugins must be resolved before they can be applied. Non-core plugins are identified by a

unique ID and a version in the build file:

```
plugins {  
    id «plugin id» version «plugin version»  
}
```

For example:

build.gradle.kts

```
plugins {  
    id("com.gradleup.shadow") version "8.3.4"  
}
```

build.gradle

```
plugins {  
    id 'com.gradleup.shadow' version '8.3.4'  
}
```

And the location of the plugin must be specified in the settings file:

settings.gradle.kts

```
pluginManagement { ①  
    repositories {  
        gradlePluginPortal()  
    }  
}
```

settings.gradle

```
pluginManagement { ①  
    repositories {  
        gradlePluginPortal()  
    }  
}
```

There are additional considerations for resolving and applying plugins:

#	To	Use	For example:
1	Apply a plugin to a project.	The plugins block in the build file.	<pre>plugins { id("org.barfuin.gradle.taskinfo") version "2.1.0" }</pre>
2	Apply a plugin to multiple projects.	The subprojects or allprojects blocks in the root build file. Not Recommended	<pre>plugins { id("org.barfuin.gradle.taskinfo") version "2.1.0" } allprojects { apply(plugin = "org.barfuin.gradle.taskinfo") repositories { mavenCentral() } }</pre>
3	Apply a plugin to multiple projects.	The plugins block in the root build file.	<pre>plugins { id("com.gradleup.shadow") version "8.3.4" apply false id("io.ratpack.ratpack-java") version "1.8.2" apply false }</pre>
4	Apply a plugin to multiple projects.	A convention plugin in the buildSrc directory. Recommended.	<pre>plugins { id("my-convention.gradle.taskinfo") }</pre>

#	To	Use	For example:
5	Apply a plugin needed <i>for the build script itself</i> .	The <code>buildscript</code> block in the build file itself. Legacy.	<pre> buildscript { repositories { mavenCentral() } dependencies { classpath("org.barfuin.gradle.taskinfo:gradle-taskinfo:2.1.0") } } apply(plugin = "org.barfuin.gradle.taskinfo") </pre>
6	Apply a script plugins. Applying a plugin when type-safe accessors are not available.	The legacy <code>apply()</code> method in the build file. Not Recommended. Legacy.	<pre> apply<MyCustomBarfuinTaskInfoPlugin>() </pre>

1. Applying plugins using the `plugins{}` block

The plugin DSL provides a concise and convenient way to declare plugin dependencies.

The `plugins` block configures an instance of `PluginDependenciesSpec`:

```

plugins {
    application                // by name
    java                       // by name
    id("java")                 // by id - recommended
    id("org.jetbrains.kotlin.jvm") version "1.9.0" // by id - recommended
}

```

Core Gradle plugins are unique in that they provide short names, such as `java` for the core `JavaPlugin`.

To apply a core plugin, the short **name** can be used:

build.gradle.kts

```

plugins {
    java
}

```

build.gradle

```
plugins {  
    id 'java'  
}
```

All other binary plugins must use the fully qualified form of the plugin id (e.g., `com.github.foo.bar`).

To apply a community plugin from [Gradle plugin portal](#), the fully qualified **plugin id**, a globally unique identifier, must be used:

build.gradle.kts

```
plugins {  
    id("org.springframework.boot") version "3.3.1"  
}
```

build.gradle

```
plugins {  
    id 'org.springframework.boot' version '3.3.1'  
}
```

See `PluginDependenciesSpec` for more information on using the Plugin DSL.

Limitations of the plugins DSL

The plugins DSL provides a convenient syntax for users and the ability for Gradle to determine which plugins are used quickly. This allows Gradle to:

- Optimize the loading and reuse of plugin classes.
- Provide editors with detailed information about the potential properties and values in the build script.

However, the DSL requires that plugins be defined statically.

There are some key differences between the `plugins {}` block mechanism and the "traditional" `apply()` method mechanism. There are also some constraints and possible limitations.

The `plugins{}` block can only be used in a project's build script `build.gradle(.kts)` and the `settings.gradle(.kts)` file. It must appear before any other block. It cannot be used in script plugins or init scripts.

Constrained Syntax

The `plugins {}` block does not support arbitrary code.

It is constrained to be idempotent (produce the same result every time) and side effect-free (safe for Gradle to execute at any time).

The form is:

```
plugins {  
    id(«plugin id»)                ①  
    id(«plugin id») version «plugin version»  ②  
}
```

① for core Gradle plugins or plugins already available to the build script

② for binary Gradle plugins that need to be resolved

Where `«plugin id»` and `«plugin version»` are a string.

Where `«plugin id»` and `«plugin version»` must be constant, literal strings.

The `plugins{}` block must also be a top-level statement in the build script. It cannot be nested inside another construct (e.g., an if-statement or for-loop).

2. Applying plugins to all subprojects{} or allprojects{}

Suppose you have a [multi-project build](#), you probably want to apply plugins to some or all of the subprojects in your build but not to the `root` project.

While the default behavior of the `plugins{}` block is to immediately **resolve** and **apply** the plugins, you can use the `apply false` syntax to tell Gradle not to apply the plugin to the current project. Then, use the `plugins{}` block without the version in subprojects' build scripts:

settings.gradle.kts

```
include("hello-a")  
include("hello-b")  
include("goodbye-c")
```

build.gradle.kts

```
plugins {  
    // These plugins are not automatically applied.  
    // They can be applied in subprojects as needed (in their respective  
    build files).  
    id("com.example.hello") version "1.0.0" apply false  
    id("com.example.goodbye") version "1.0.0" apply false  
}
```

```
allprojects {
    // Apply the common 'java' plugin to all projects (including the root)
    plugins.apply("java")
}

subprojects {
    // Apply the 'java-library' plugin to all subprojects (excluding the
    root)
    plugins.apply("java-library")
}
```

hello-a/build.gradle.kts

```
plugins {
    id("com.example.hello")
}
```

hello-b/build.gradle.kts

```
plugins {
    id("com.example.hello")
}
```

goodbye-c/build.gradle.kts

```
plugins {
    id("com.example.goodbye")
}
```

settings.gradle

```
include 'hello-a'
include 'hello-b'
include 'goodbye-c'
```

build.gradle

```
plugins {
    // These plugins are not automatically applied.
    // They can be applied in subprojects as needed (in their respective
    build files).
    id 'com.example.hello' version '1.0.0' apply false
    id 'com.example.goodbye' version '1.0.0' apply false
}
```

```
allprojects {
    // Apply the common 'java' plugin to all projects (including the root)
    apply(plugin: 'java')
}

subprojects {
    // Apply the 'java-library' plugin to all subprojects (excluding the
    root)
    apply(plugin: 'java-library')
}
```

hello-a/build.gradle

```
plugins {
    id 'com.example.hello'
}
```

hello-b/build.gradle

```
plugins {
    id 'com.example.hello'
}
```

goodbye-c/build.gradle

```
plugins {
    id 'com.example.goodbye'
}
```

You can also encapsulate the versions of external plugins by composing the build logic using your own [convention plugins](#).

3. Applying plugins declared in the root project

You can apply plugins from the root or parent project in a multi-project build to share common logic and behavior with other projects. The root/parent project is the project at the top of the directory hierarchy.

You should use the `plugins {}` block because it ensures that the plugin is applied and configured before the project's evaluation phase. This way, you can safely use type-safe accessors for any model elements introduced by the plugin:

settings.gradle.kts

```
rootProject.name = "multi-project-build"
```

```
include("domain", "infra", "http")
```

build.gradle.kts

```
plugins {  
    id("com.gradleup.shadow") version "8.3.4" apply false  
    id("io.ratpack.ratpack-java") version "1.8.2" apply false  
}
```

domain/build.gradle.kts

```
plugins {  
    `java-library`  
}  
  
dependencies {  
    api("javax.measure:unit-api:1.0")  
    implementation("tec.units:unit-ri:1.0.3")  
}
```

infra/build.gradle.kts

```
plugins {  
    `java-library`  
    id("com.gradleup.shadow")  
}
```

http/build.gradle.kts

```
plugins {  
    java  
    id("io.ratpack.ratpack-java")  
}  
  
dependencies {  
    implementation(project(":domain"))  
    implementation(project(":infra"))  
    implementation(ratpack.dependency("dropwizard-metrics"))  
}
```

In the root/parent **build.gradle(.kts)**, plugins are declared but not applied (via **apply false**). This approach, while optional, makes the plugins available to be explicitly applied in specific subprojects. Without **apply false**, plugins declared in the root project cannot be explicitly applied only to certain subprojects.

NOTE | **apply false** is optional. If **apply false** is not used when declaring plugins in the root

build file, those plugins would automatically be applied to the root project.

In the `infra` subproject, the `com.gradleup.shadow` plugin, which was made available in the root project, is explicitly applied. The `http` subproject applies `io.ratpack.ratpack-java`. The `domain` subproject does not apply a plugin from the root.

4. Applying convention plugins from the `buildSrc` directory

`buildSrc` is an optional directory at the Gradle project root that contains build logic (i.e., plugins) used in building the main project. You can apply plugins that reside in a project's `buildSrc` directory as long as they have a defined ID.

The following example shows how to tie the plugin implementation class `my.MyPlugin`, defined in `buildSrc`, to the id "my-plugin":

buildSrc/build.gradle.kts

```
plugins {
    `java-gradle-plugin`
}

gradlePlugin {
    plugins {
        create("myPlugins") {
            id = "my-plugin"
            implementationClass = "my.MyPlugin"
        }
    }
}
```

buildSrc/build.gradle

```
plugins {
    id 'java-gradle-plugin'
}

gradlePlugin {
    plugins {
        myPlugins {
            id = 'my-plugin'
            implementationClass = 'my.MyPlugin'
        }
    }
}
```

The plugin can then be applied by ID:

build.gradle.kts

```
plugins {  
    id("my-plugin")  
}
```

build.gradle

```
plugins {  
    id 'my-plugin'  
}
```

5. Applying plugins using the `buildscript{} block`

To define libraries or plugins used in the build script itself, you can use the `buildscript` block. The `buildscript` block is also used for specifying where to find those dependencies.

This approach is less common with newer versions of Gradle, as the `plugins {}` block simplifies plugin usage. However, `buildscript {}` may be necessary when dealing with custom or non-standard plugin repositories as well as libraries dependencies:

build.gradle.kts

```
import org.yaml.snakeyaml.Yaml  
import java.io.File  
  
buildscript {  
    repositories {  
        maven {  
            url = uri("https://plugins.gradle.org/m2/")  
        }  
        mavenCentral() // Where to find the plugin  
    }  
    dependencies {  
        classpath("org.yaml:snakeyaml:1.19") // The library's classpath  
        dependency  
        classpath("com.gradleup.shadow:shadow-gradle-plugin:8.3.4") // Plugin  
        dependency for legacy plugin application  
    }  
}
```

```
// Applies legacy Shadow plugin
apply(plugin = "com.gradleup.shadow")

// Uses the library in the build script
val yamlContent = """
    name: Project
    """.trimIndent()
val yaml = Yaml()
val data: Map<String, Any> = yaml.load(yamlContent)
```

build.gradle

```
import org.yaml.snakeyaml.Yaml

buildscript {
    repositories { // Where to find the plugin or library
        maven {
            url = uri("https://plugins.gradle.org/m2/")
        }
        mavenCentral()
    }
    dependencies {
        classpath 'org.yaml:snakeyaml:1.19' // The library's classpath
    }
}

// Applies legacy Shadow plugin
apply plugin: 'com.gradleup.shadow'

// Uses the library in the build script
def yamlContent = """
    name: Project Name
    """
def yaml = new Yaml()
def data = yaml.load(yamlContent)
```

6. Applying script plugins using the legacy `apply()` method

A script plugin is an ad-hoc plugin, typically written and applied in the same build script. It is applied using the [legacy application method](#):

build.gradle.kts

```
class MyPlugin : Plugin<Project> {  
    override fun apply(project: Project) {  
        println("Plugin ${this.javaClass.simpleName} applied on  
${project.name}")  
    }  
}  
  
apply<MyPlugin>()
```

build.gradle

```
class MyPlugin implements Plugin<Project> {  
    @Override  
    void apply(Project project) {  
        println("Plugin ${this.getClass().getSimpleName()} applied on  
${project.name}")  
    }  
}  
  
apply plugin: MyPlugin
```

Plugin Management

The `pluginManagement{}` block is used to configure repositories for plugin resolution and to define version constraints for plugins that are applied in the build scripts.

The `pluginManagement{}` block can be used in a `settings.gradle(.kts)` file, where it must be the first block in the file:

settings.gradle.kts

```
pluginManagement {  
    plugins {  
    }  
    resolutionStrategy {  
    }  
    repositories {  
    }  
}  
rootProject.name = "plugin-management"
```

settings.gradle

```
pluginManagement {  
    plugins {  
    }  
    resolutionStrategy {  
    }  
    repositories {  
    }  
}  
rootProject.name = 'plugin-management'
```

The block can also be used in [Initialization Script](#):

init.gradle.kts

```
settingsEvaluated {  
    pluginManagement {  
        plugins {  
        }  
        resolutionStrategy {  
        }  
        repositories {  
        }  
    }  
}
```

init.gradle

```
settingsEvaluated { settings ->  
    settings.pluginManagement {  
        plugins {  
        }  
        resolutionStrategy {  
        }  
        repositories {  
        }  
    }  
}
```

Custom Plugin Repositories

By default, the `plugins{}` DSL resolves plugins from the public [Gradle Plugin Portal](#).

Many build authors would also like to resolve plugins from private Maven or Ivy repositories because they contain proprietary implementation details or to have more control over what plugins are available to their builds.

To specify custom plugin repositories, use the `repositories{}` block inside `pluginManagement{}` :

settings.gradle.kts

```
pluginManagement {  
    repositories {  
        maven(url = file("../maven-repo"))  
        gradlePluginPortal()  
        ivy(url = file("../ivy-repo"))  
    }  
}
```

settings.gradle

```
pluginManagement {  
    repositories {  
        maven {  
            url = file('../maven-repo')  
        }  
        gradlePluginPortal()  
        ivy {  
            url = file('../ivy-repo')  
        }  
    }  
}
```

This tells Gradle to first look in the Maven repository at `../maven-repo` when resolving plugins and then to check the Gradle Plugin Portal if the plugins are not found in the Maven repository. If you don't want the Gradle Plugin Portal to be searched, omit the `gradlePluginPortal()` line. Finally, the Ivy repository at `../ivy-repo` will be checked.

Plugin Version Management

A `plugins{}` block inside `pluginManagement{}` allows all plugin versions for the build to be defined in a single location. Plugins can then be applied by id to any build script via the `plugins{}` block.

One benefit of setting plugin versions this way is that the `pluginManagement.plugins{}` does not have

the same **constrained syntax** as the build script `plugins{}` block. This allows plugin versions to be taken from `gradle.properties`, or loaded via another mechanism.

Managing plugin versions via `pluginManagement`:

settings.gradle.kts

```
pluginManagement {
    val helloPluginVersion: String by settings
    plugins {
        id("com.example.hello") version "${helloPluginVersion}"
    }
}
```

build.gradle.kts

```
plugins {
    id("com.example.hello")
}
```

gradle.properties

```
helloPluginVersion=1.0.0
```

settings.gradle

```
pluginManagement {
    plugins {
        id 'com.example.hello' version "${helloPluginVersion}"
    }
}
```

build.gradle

```
plugins {
    id 'com.example.hello'
}
```

gradle.properties

```
helloPluginVersion=1.0.0
```

The plugin version is loaded from `gradle.properties` and configured in the settings script, allowing

the plugin to be added to any project without specifying the version.

Plugin Resolution Rules

Plugin resolution rules allow you to modify plugin requests made in `plugins{} blocks`, e.g., changing the requested version or explicitly specifying the implementation artifact coordinates.

To add resolution rules, use the `resolutionStrategy{} inside the pluginManagement{} block`:

settings.gradle.kts

```
pluginManagement {
    resolutionStrategy {
        eachPlugin {
            if (requested.id.namespace == "com.example") {
                useModule("com.example:sample-plugins:1.0.0")
            }
        }
    }
    repositories {
        maven {
            url = uri("./maven-repo")
        }
        gradlePluginPortal()
        ivy {
            url = uri("./ivy-repo")
        }
    }
}
```

settings.gradle

```
pluginManagement {
    resolutionStrategy {
        eachPlugin {
            if (requested.id.namespace == 'com.example') {
                useModule('com.example:sample-plugins:1.0.0')
            }
        }
    }
    repositories {
        maven {
            url = file('./maven-repo')
        }
        gradlePluginPortal()
        ivy {
            url = file('./ivy-repo')
        }
    }
}
```

```
}  
}  
}
```

This tells Gradle to use the specified plugin implementation artifact instead of its built-in default mapping from plugin ID to Maven/Ivy coordinates.

Custom Maven and Ivy plugin repositories must contain [plugin marker artifacts](#) and the artifacts that implement the plugin. Read [Gradle Plugin Development Plugin](#) for more information on publishing plugins to custom repositories.

See [PluginManagementSpec](#) for complete documentation for using the `pluginManagement{}` block.

Plugin Marker Artifacts

Since the `plugins{}` DSL block only allows for declaring plugins by their globally unique plugin `id` and `version` properties, Gradle needs a way to look up the coordinates of the plugin implementation artifact.

To do so, Gradle will look for a Plugin Marker Artifact with the coordinates `plugin.id:plugin.id.gradle.plugin:plugin.version`. This marker needs to have a dependency on the actual plugin implementation. Publishing these markers is automated by the [java-gradle-plugin](#).

For example, the following complete sample from the [sample-plugins](#) project shows how to publish a `com.example.hello` plugin and a `com.example.goodbye` plugin to both an Ivy and Maven repository using the combination of the [java-gradle-plugin](#), the [maven-publish](#) plugin, and the [ivy-publish](#) plugin.

build.gradle.kts

```
plugins {  
    `java-gradle-plugin`  
    `maven-publish`  
    `ivy-publish`  
}  
  
group = "com.example"  
version = "1.0.0"  
  
gradlePlugin {  
    plugins {  
        create("hello") {  
            id = "com.example.hello"  
            implementationClass = "com.example.hello.HelloPlugin"  
        }  
        create("goodbye") {  
            id = "com.example.goodbye"        }  
    }  
}
```

```

        implementationClass = "com.example.goodbye.GoodbyePlugin"
    }
}

publishing {
    repositories {
        maven {
            url = uri(layout.buildDirectory.dir("maven-repo"))
        }
        ivy {
            url = uri(layout.buildDirectory.dir("ivy-repo"))
        }
    }
}

```

build.gradle

```

plugins {
    id 'java-gradle-plugin'
    id 'maven-publish'
    id 'ivy-publish'
}

group = 'com.example'
version = '1.0.0'

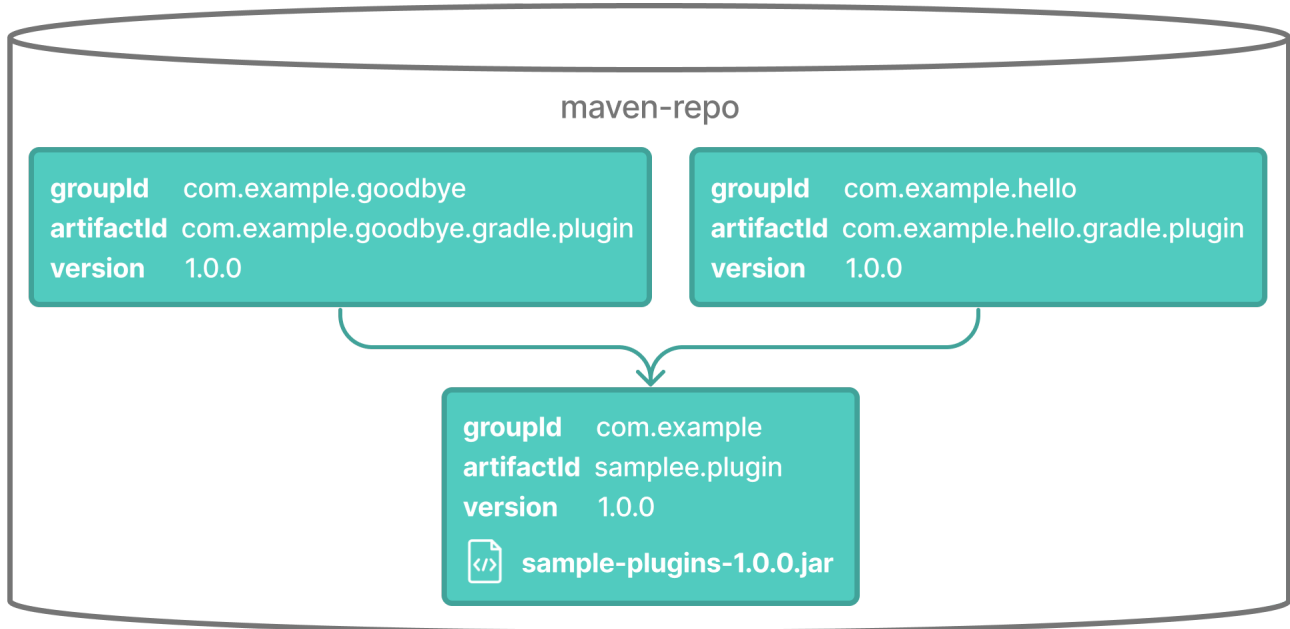
gradlePlugin {
    plugins {
        hello {
            id = 'com.example.hello'
            implementationClass = 'com.example.hello.HelloPlugin'
        }
        goodbye {
            id = 'com.example.goodbye'
            implementationClass = 'com.example.goodbye.GoodbyePlugin'
        }
    }
}

publishing {
    repositories {
        maven {
            url = layout.buildDirectory.dir('maven-repo')
        }
        ivy {
            url = layout.buildDirectory.dir('ivy-repo')
        }
    }
}

```

```
}
```

Running **gradle publish** in the sample directory creates the following Maven repository layout (the Ivy layout is similar):



Legacy Plugin Application

With the introduction of the [plugins DSL](#), users should have little reason to use the legacy method of applying plugins. It is documented here in case a build author cannot use the plugin DSL due to restrictions in how it currently works.

build.gradle.kts

```
apply(plugin = "java")
```

build.gradle

```
apply plugin: 'java'
```

Plugins can be applied using a *plugin id*. In the above case, we are using the short name "java" to apply the [JavaPlugin](#).

Rather than using a plugin id, plugins can also be applied by simply specifying the class of the plugin:

build.gradle.kts

```
apply<JavaPlugin>()
```

build.gradle

```
apply plugin: JavaPlugin
```

The `JavaPlugin` symbol in the above sample refers to the `JavaPlugin`. This class does not strictly need to be imported as the `org.gradle.api.plugins` package is automatically imported in all build scripts (see [Default imports](#)).

Furthermore, one needs to append the `::class` suffix to identify a class literal in Kotlin instead of `.class` in Java.

Furthermore, it is unnecessary to append `.class` to identify a class literal in Groovy as it is in Java.

You may also see the `apply` method used to include an entire build file:

build.gradle.kts

```
apply(from = "other.gradle.kts")
```

build.gradle

```
apply from: 'other.gradle'
```

Using a Version Catalog

When a project uses a version catalog, plugins can be referenced via aliases when applied.

Let's take a look at a simple Version Catalog:

libs.versions.toml

```
[versions]  
groovy = "3.0.5"
```

```

checkstyle = "8.37"

[libraries]
groovy-core = { module = "org.codehaus.groovy:groovy", version.ref = "groovy"
}
groovy-json = { module = "org.codehaus.groovy:groovy-json", version.ref =
"groovy" }
groovy-nio = { module = "org.codehaus.groovy:groovy-nio", version.ref =
"groovy" }
commons-lang3 = { group = "org.apache.commons", name = "commons-lang3",
version = { strictly = "[3.8, 4.0[" , prefer="3.9" } }

[bundles]
groovy = ["groovy-core", "groovy-json", "groovy-nio"]

[plugins]
versions = { id = "com.github.ben-manes.versions", version = "0.45.0" }

```

Then a plugin can be applied to any build script using the **alias** method:

build.gradle.kts

```

plugins {
    `java-library`
    alias(libs.plugins.versions)
}

```

build.gradle

```

plugins {
    id 'java-library'
    alias(libs.plugins.versions)
}

```

TIP | Gradle generates type safe accessors for catalog items.

Next Step: [Learn how to write Plugins >>](#)

Writing Plugins

If Gradle or the Gradle community does not offer the specific capabilities your project needs, creating your own custom plugin could be a solution.

Additionally, if you find yourself duplicating build logic across subprojects and need a better way to organize it, convention plugins can help.

Script plugin

A plugin is any class that implements the `Plugin` interface. For example, this is a "hello world" plugin:

build.gradle.kts

```
abstract class SamplePlugin : Plugin<Project> { ①
    override fun apply(project: Project) { ②
        project.tasks.register("ScriptPlugin") {
            doLast {
                println("Hello world from the build file!")
            }
        }
    }
}

apply<SamplePlugin>() ③
```

build.gradle

```
class SamplePlugin implements Plugin<Project> { ①
    void apply(Project project) { ②
        project.tasks.register("ScriptPlugin") {
            doLast {
                println("Hello world from the build file!")
            }
        }
    }
}

apply plugin: SamplePlugin ③
```

- ① Extend the `org.gradle.api.Plugin` interface.
- ② Override the `apply` method.
- ③ `apply` the plugin to the project.

1. Extend the `org.gradle.api.Plugin` interface

Create a class that extends the `Plugin` interface:

build.gradle.kts

```
abstract class SamplePlugin : Plugin<Project> {  
}
```

build.gradle

```
class SamplePlugin implements Plugin<Project> {  
}
```

2. Override the **apply** method

Add tasks and other logic in the **apply()** method:

build.gradle.kts

```
override fun apply() {  
}
```

build.gradle

```
void apply(Project project) {  
}
```

3. **apply** the plugin to your project

When **SamplePlugin** is applied in your project, Gradle calls the **fun apply() {}** method defined. This adds the **ScriptPlugin** task to your project:

build.gradle.kts

```
apply<SamplePlugin>()
```

```
build.gradle
```

```
apply plugin: SamplePlugin
```

Note that this is a simple **hello-world** example and does **not** reflect best practices.

IMPORTANT | Script plugins are **not** recommended.

The best practice for developing plugins is to create *convention plugins* or *binary plugins*.

Pre-compiled script plugin

Pre-compiled script plugins offer an easy way to rapidly prototype and experiment. They let you package build logic as `*.gradle(.kts)` script files using the Groovy or Kotlin DSL. These scripts reside in specific directories, such as `src/main/groovy` or `src/main/kotlin`.

To apply one, simply use its **ID** derived from the script filename (without `.gradle`). You can think of the file itself as the plugin, so you do not need to subclass the `Plugin` interface in a precompiled script.

Let's take a look at an example with the following structure:

```
.
├── buildSrc
│   ├── build.gradle.kts
│   └── src
│       ├── main
│       │   └── kotlin
│       │       └── my-create-file-plugin.gradle.kts
```

Our `my-create-file-plugin.gradle.kts` file contains the following code:

```
buildSrc/src/main/kotlin/my-create-file-plugin.gradle.kts
```

```
abstract class CreateFileTask : DefaultTask() {
    @get:Input
    abstract val fileText: Property<String>

    @Input
    val fileName = "myfile.txt"

    @OutputFile
    val myFile: File = File(fileName)

    @TaskAction
```

```

        fun action() {
            myFile.createNewFile()
            myFile.writeText(fileText.get())
        }
    }

tasks.register<CreateFileTask>("createMyFileTaskInConventionPlugin") {
    group = "from my convention plugin"
    description = "Create myfile.txt in the current directory"
    fileText.set("HELLO FROM MY CONVENTION PLUGIN")
}

```

buildSrc/src/main/groovy/my-create-file-plugin.gradle

```

abstract class CreateFileTask extends DefaultTask {
    @Input
    abstract Property<String> getFileText()

    @Input
    String fileName = "myfile.txt"

    @OutputFile
    File getMyFile() {
        return new File(fileName)
    }

    @TaskAction
    void action() {
        myFile.createNewFile()
        myFile.writeText(fileText.get())
    }
}

tasks.register("createMyFileTaskInConventionPlugin", CreateFileTask) {
    group = "from my convention plugin"
    description = "Create myfile.txt in the current directory"
    fileText.set("HELLO FROM MY CONVENTION PLUGIN")
}

```

The pre-compiled script can now be applied in the `build.gradle(.kts)` file of any subproject:

build.gradle.kts

```

plugins {
    id("my-create-file-plugin") // Apply the pre-compiled convention plugin
}

```

```
`kotlin-dsl`  
}
```

build.gradle

```
plugins {  
    id 'my-create-file-plugin' // Apply the pre-compiled convention plugin  
    id 'groovy' // Apply the Groovy DSL plugin  
}
```

The `createFileTask` task from the plugin is now available in your subproject.

Binary Plugins

A binary plugin is a plugin that is implemented in a compiled language and is packaged as a JAR file. It is resolved as a dependency rather than compiled from source.

For most use cases, convention plugins must be updated infrequently. Having each developer execute the plugin build as part of their development process is wasteful, and we can instead distribute them as binary dependencies.

There are two ways to update the convention plugin in the example above into a binary plugin.

1. Use [composite builds](#):

settings.gradle.kts

```
includeBuild("my-plugin")
```

2. [Publish the plugin](#) to a repository:

build.gradle.kts

```
plugins {  
    id("com.gradle.plugin.my-plugin") version "1.0.0"  
}
```

Let's go with the second solution. This plugin has been re-written in Kotlin and is called `MyCreateFileBinaryPlugin.kt`. It is still stored in `buildSrc`:

buildSrc/src/main/kotlin/MyCreateFileBinaryPlugin.kt

```
import org.gradle.api.DefaultTask  
import org.gradle.api.Plugin
```

```

import org.gradle.api.Project
import org.gradle.api.provider.Property
import org.gradle.api.tasks.Input
import org.gradle.api.tasks.OutputFile
import org.gradle.api.tasks.TaskAction
import java.io.File

abstract class CreateFileTask : DefaultTask() {
    @get:Input
    abstract val fileText: Property<String>

    @Input
    val filePath =
project.layout.settingsDirectory.file("myfile.txt").asFile.path

    @OutputFile
    val myFile: File = File(filePath)

    @TaskAction
    fun action() {
        myFile.createNewFile()
        myFile.writeText(fileText.get())
    }
}

class MyCreateFileBinaryPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        project.tasks.register("createFileTaskFromBinaryPlugin",
CreateFileTask::class.java) {
            group = "from my binary plugin"
            description = "Create myfile.txt in the current directory"
            fileText.set("HELLO FROM MY BINARY PLUGIN")
        }
    }
}

```

The plugin can be published and given an `id` using a `gradlePlugin{}` block so that it can be referenced in the root:

buildSrc/build.gradle.kts

```

group = "com.example"
version = "1.0.0"

gradlePlugin {
    plugins {
        create("my-binary-plugin") {

```

```

        id = "com.example.my-binary-plugin"
        implementationClass = "MyCreateFileBinaryPlugin"
    }
}

publishing {
    repositories {
        mavenLocal()
    }
}

```

buildSrc/build.gradle

```

group = 'com.example'
version = '1.0.0'

gradlePlugin {
    plugins {
        create("my-binary-plugin") {
            id = "com.example.my-binary-plugin"
            implementationClass = "MyCreateFileBinaryPlugin"
        }
    }
}

publishing {
    repositories {
        mavenLocal()
    }
}

```

Then, the plugin can be applied in the build file:

build.gradle.kts

```

plugins {
    id("my-create-file-plugin") // Apply the pre-compiled convention plugin
    id("com.example.my-binary-plugin") // Apply the binary plugin
    `kotlin-dsl`
}

```

build.gradle

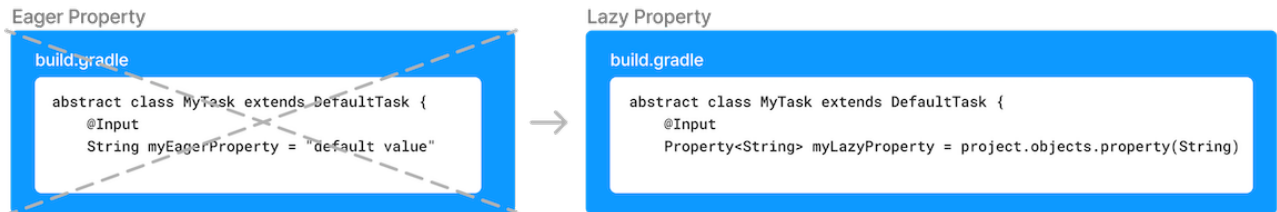
```
plugins {  
    id 'my-create-file-plugin' // Apply the pre-compiled convention plugin  
    id 'com.example.my-binary-plugin' // Apply the binary plugin  
    id 'groovy' // Apply the Groovy DSL plugin  
}
```

Consult the [Developing Plugins chapter](#) to learn more.

GRADLE TYPES

Understanding Properties and Providers

Gradle provides properties that are important for [lazy configuration](#). When implementing a custom task or plugin, it's imperative that you use these lazy properties.



Gradle represents lazy properties with two interfaces:

1. [Property](#) - Represents a value that can be queried and changed.
2. [Provider](#) - Represents a value that can only be queried and cannot be changed.

Properties and providers manage *values and configurations* in a build script.

In this example, `configuration` is a `Property<String>` that is set to the `configurationProvider Provider<String>`. The `configurationProvider` lazily provides the value `"Hello, Gradle!"`:

build.gradle.kts

```
abstract class MyIntroTask : DefaultTask() {  
    @get:Input  
    abstract val configuration: Property<String>  
  
    @TaskAction  
    fun printConfiguration() {  
        println("Configuration value: ${configuration.get()}")  
    }  
}  
  
val configurationProvider: Provider<String> = project.provider { "Hello,  
Gradle!" }  
  
tasks.register("myIntroTask", MyIntroTask::class) {  
    configuration.set(configurationProvider)  
}
```

build.gradle

```
abstract class MyIntroTask extends DefaultTask {
```

```

@Input
abstract Property<String> getConfiguration()

@TaskAction
void printConfiguration() {
    println "Configuration value: ${configuration.get()}"
}
}

Provider<String> configurationProvider = project.provider { "Hello, Gradle!"
}

tasks.register("myIntroTask", MyIntroTask) {
    it.setConfiguration(configurationProvider)
}

```

Understanding Properties

Properties in Gradle are variables that hold values. They can be defined and accessed within the build script to store information like file paths, version numbers, or custom values.

Properties can be set and retrieved using the **project** object:

build.gradle.kts

```

// Setting a property
val simpleMessageProperty: Property<String> =
project.objects.property(String::class)
simpleMessageProperty.set("Hello, World from a Property!")
// Accessing a property
println(simpleMessageProperty.get())

```

build.gradle

```

// Setting a property
def simpleMessageProperty = project.objects.property(String)
simpleMessageProperty.set("Hello, World from a Property!")
// Accessing a property
println(simpleMessageProperty.get())

```

Properties:

- Properties with these types are configurable.

- **Property** extends the **Provider** interface.
- The method **Property.set(T)** specifies a value for the property, overwriting whatever value may have been present.
- The method **Property.set(Provider)** specifies a **Provider** for the value for the property, overwriting whatever value may have been present. This allows you to wire together **Provider** and **Property** instances before the values are configured.
- A **Property** can be created by the factory method **ObjectFactory.property(Class)**.

Understanding Providers

Providers are objects that represent a value that may not be immediately available. Providers are useful for lazy evaluation and can be used to model values that may change over time or depend on other tasks or inputs:

build.gradle.kts

```
// Setting a provider
val simpleMessageProvider: Provider<String> = project.providers.provider {
    "Hello, World from a Provider!" }
// Accessing a provider
println(simpleMessageProvider.get())
```

build.gradle

```
// Setting a provider
def simpleMessageProvider = project.providers.provider { "Hello, World from a
Provider!" }
// Accessing a provider
println(simpleMessageProvider.get())
```

Providers:

- Properties with these types are read-only.
- The method **Provider.get()** returns the current value of the property.
- A **Provider** can be created from another **Provider** using **Provider.map(Transformer)**.
- Many other types extend **Provider** and can be used wherever a **Provider** is required.

Using Gradle Managed Properties

Gradle's managed properties allow you to declare properties as abstract getters (Java, Groovy) or abstract properties (Kotlin).

Gradle then automatically provides the implementation for these properties, managing their state.

A property may be *mutable*, meaning that it has both a `get()` method and `set()` method:

build.gradle.kts

```
abstract class MyPropertyTask : DefaultTask() {
    @get:Input
    abstract val messageProperty: Property<String> // message property

    @TaskAction
    fun printMessage() {
        println(messageProperty.get())
    }
}

tasks.register<MyPropertyTask>("myPropertyTask") {
    messageProperty.set("Hello, Gradle!")
}
```

build.gradle

```
abstract class MyPropertyTask extends DefaultTask {
    @Input
    abstract Property<String> messageProperty = project.objects.property
    (String)

    @TaskAction
    void printMessage() {
        println(messageProperty.get())
    }
}

tasks.register('myPropertyTask', MyPropertyTask) {
    messageProperty.set("Hello, Gradle!")
}
```

Or *read-only*, meaning that it has only a `get()` method. The *read-only* properties are *providers*:

build.gradle.kts

```
abstract class MyProviderTask : DefaultTask() {
    final val messageProvider: Provider<String> = project.providers.provider
    { "Hello, Gradle!" } // message provider
}
```

```

    @TaskAction
    fun printMessage() {
        println(messageProvider.get())
    }
}

tasks.register<MyProviderTask>("MyProviderTask") {
}

```

build.gradle

```

abstract class MyProviderTask extends DefaultTask {
    final Provider<String> messageProvider = project.providers.provider {
        "Hello, Gradle!" }

    @TaskAction
    void printMessage() {
        println(messageProvider.get())
    }
}

tasks.register('MyProviderTask', MyProviderTask)

```

Mutable Managed Properties

A mutable managed property is declared using an abstract getter method of type `Property<T>`, where `T` can be any serializable type or a [fully managed Gradle type](#). The property must not have any setter methods.

Here is an example of a task type with an `uri` property of type `URI`:

Download.java

```

public abstract class Download extends DefaultTask {
    @Input
    public abstract Property<URI> getUri(); // abstract getter of type Property<T>

    @TaskAction
    void run() {
        System.out.println("Downloading " + getUri().get()); // Use the `uri` property
    }
}

```

Note that for a property to be considered a mutable managed property, the property's getter

methods must be **abstract** and have **public** or **protected** visibility.

The property type must be one of the following:

Property Type	Note
<code>Property<T></code>	Where T is typically Double , Integer , Long , String , or Bool
<code>RegularFileProperty</code>	Configurable regular file location, whose value is mutable
<code>DirectoryProperty</code>	Configurable directory location, whose value is mutable
<code>ListProperty<T></code>	List of elements of type T
<code>SetProperty<T></code>	Set of elements of type T
<code>MapProperty<K, V></code>	Map of K type keys with V type values
<code>ConfigurableFileCollection</code>	A mutable FileCollection which represents a collection of file system locations
<code>ConfigurableFileTree</code>	A mutable FileTree which represents a hierarchy of files

Read-only Managed Properties (Providers)

You can declare a read-only managed property, also known as a provider, using a getter method of type `Provider<T>`. The method implementation needs to derive the value. It can, for example, derive the value from other properties.

Here is an example of a task type with a **uri** provider that is derived from a **location** property:

Download.java

```
public abstract class Download extends DefaultTask {
    @Input
    public abstract Property<String> getLocation();

    @Internal
    public Provider<URI> getUri() {
        return getLocation().map(l -> URI.create("https://" + l));
    }

    @TaskAction
    void run() {
        System.out.println("Downloading " + getUri().get()); // Use the `uri`
        provider (read-only property)
    }
}
```

Read-only Managed Nested Properties (Nested Providers)

You can declare a read-only managed nested property by adding an abstract getter method for the property to a type annotated with `@Nested`. The property should not have any setter methods. Gradle provides the implementation for the getter method and creates a value for the property.

This pattern is useful when a custom type has a nested complex type which has the same lifecycle. If the lifecycle is different, consider using `Property<NestedType>` instead.

Here is an example of a task type with a `resource` property. The `Resource` type is also a custom Gradle type and defines some managed properties:

Download.java

```
public abstract class Download extends DefaultTask {
    @Nested
    public abstract Resource getResource(); // Use an abstract getter method annotated
    with @Nested

    @TaskAction
    void run() {
        // Use the `resource` property
        System.out.println("Downloading https://" + getResource().getHostName().get()
+ "/" + getResource().getPath().get());
    }
}

public interface Resource {
    @Input
    Property<String> getHostName();
    @Input
    Property<String> getPath();
}
```

Read-only Managed "name" Property (Provider)

If the type contains an abstract property called "name" of type `String`, Gradle provides an implementation for the getter method, and extends each constructor with a "name" parameter, which comes before all other constructor parameters.

If the type is an interface, Gradle will provide a constructor with a single "name" parameter and `@Inject` semantics.

You can have your type implement or extend the `Named` interface, which defines such a read-only "name" property:

```
import org.gradle.api.Named

interface MyType : Named {
    // Other properties and methods...
```

```

}

class MyTypeImpl(override val name: String) : MyType {
    // Implement other properties and methods...
}

// Usage
val instance = MyTypeImpl("myName")
println(instance.name) // Prints: myName

```

Using Gradle Managed Types

A managed type as an abstract class or interface with no fields and whose properties are all managed. These types have their state entirely managed by Gradle.

For example, this managed type is defined as an interface:

Resource.java

```

public interface Resource {
    @Input
    Property<String> getHostName();
    @Input
    Property<String> getPath();
}

```

A *named managed type* is a managed type that additionally has an abstract property "name" of type `String`. Named managed types are especially useful as the element type of `NamedDomainObjectContainer`:

build.gradle.kts

```

interface MyNamedType {
    val name: String
}

class MyNamedTypeImpl(override val name: String) : MyNamedType

class MyPluginExtension(project: Project) {
    val myNamedContainer: NamedDomainObjectContainer<MyNamedType> =
        project.container(MyNamedType::class.java) { name ->
            project.objects.newInstance(MyNamedTypeImpl::class.java, name)
        }
}

```

build.gradle

```
interface MyNamedType {
    String getName()
}

class MyNamedTypeImpl implements MyNamedType {
    String name

    MyNamedTypeImpl(String name) {
        this.name = name
    }
}

class MyPluginExtension {
    NamedDomainObjectContainer<MyNamedType> myNamedContainer

    MyPluginExtension(Project project) {
        myNamedContainer = project.container(MyNamedType) { name ->
            new MyNamedTypeImpl(name)
        }
    }
}
```

Using Java Bean Properties

Sometimes you may see properties implemented in the Java bean property style. That is, they do not use a `Property<T>` or `Provider<T>` types but are instead implemented with concrete setter and getter methods (or corresponding conveniences in Groovy or Kotlin).

This style of property definition is legacy in Gradle and is discouraged:

```
public class MyTask extends DefaultTask {
    private String someProperty;

    public String getSomeProperty() {
        return someProperty;
    }

    public void setSomeProperty(String someProperty) {
        this.someProperty = someProperty;
    }

    @TaskAction
    public void myAction() {
        System.out.println("SomeProperty: " + someProperty);
    }
}
```

```
}
```

Understanding Collections

Gradle provides types for maintaining collections of objects, intended to work well to extends Gradle's DSLs and provide useful features such as lazy configuration.

Available collections

These collection types are used for managing collections of objects, particularly in the context of build scripts and plugins:

1. **DomainObjectSet<T>**: Represents a set of objects of type T. This set does not allow duplicate elements, and you can add, remove, and query objects in the set.
2. **NamedDomainObjectSet<T>**: A specialization of **DomainObjectSet** where each object has a unique name associated with it. This is often used for collections where each element needs to be uniquely identified by a name.
3. **NamedDomainObjectList<T>**: Similar to **NamedDomainObjectSet**, but represents a list of objects where order matters. Each element has a unique name associated with it, and you can access elements by index as well as by name.
4. **NamedDomainObjectContainer<T>**: A container for managing objects of type T, where each object has a unique name. This container provides methods for adding, removing, and querying objects by name.
5. **ExtensiblePolymorphicDomainObjectContainer<T>**: An extension of **NamedDomainObjectContainer** that allows you to define instantiation strategies for different types of objects. This is useful when you have a container that can hold multiple types of objects, and you want to control how each type of object is instantiated.

These types are commonly used in Gradle plugins and build scripts to manage collections of objects, such as tasks, configurations, or custom domain objects.

1. DomainObjectSet

A **DomainObjectSet** simply holds a set of configurable objects.

Compared to **NamedDomainObjectContainer**, a **DomainObjectSet** doesn't manage the objects in the collection. They need to be created and added manually.

You can create an instance using the **ObjectFactory.domainObjectSet()** method:

build.gradle.kts

```
abstract class MyPluginExtensionDomainObjectSet {  
    // Define a domain object set to hold strings  
    val myStrings: DomainObjectSet<String> =
```

```
project.objects.domainObjectSet(String::class)
```

```
// Add some strings to the domain object set
fun addString(value: String) {
    myStrings.add(value)
}
}
```

build.gradle

```
abstract class MyPluginExtensionDomainObjectSet {
    // Define a domain object set to hold strings
    DomainObjectSet<String> myStrings = project.objects.domainObjectSet
    (String)

    // Add some strings to the domain object set
    void addString(String value) {
        myStrings.add(value)
    }
}
```

2. NamedDomainObjectSet

A **NamedDomainObjectSet** holds a set of configurable objects, where each element has a name associated with it.

This is similar to **NamedDomainObjectContainer**, however a **NamedDomainObjectSet** doesn't manage the objects in the collection. They need to be created and added manually.

You can create an instance using the [ObjectFactory.namedDomainObjectSet\(\)](#) method.

build.gradle.kts

```
abstract class Person(val name: String)

abstract class MyPluginExtensionNamedDomainObjectSet {
    // Define a named domain object set to hold Person objects
    private val people: NamedDomainObjectSet<Person> =
    project.objects.namedDomainObjectSet(Person::class)

    // Add a person to the set
    fun addPerson(name: String) {
        people.plus(name)
    }
}
```

```
}
```

build.gradle

```
abstract class Person {
    String name
}

abstract class MyPluginExtensionNamedDomainObjectSet {
    // Define a named domain object set to hold Person objects
    NamedDomainObjectSet<Person> people = project.objects
    .namedDomainObjectSet(Person)

    // Add a person to the set
    void addPerson(String name) {
        people.create(name)
    }
}
```

3. NamedDomainObjectList

A **NamedDomainObjectList** holds a list of configurable objects, where each element has a name associated with it.

This is similar to **NamedDomainObjectContainer**, however a **NamedDomainObjectList** doesn't manage the objects in the collection. They need to be created and added manually.

You can create an instance using the [ObjectFactory.namedDomainObjectList\(\)](#) method.

build.gradle.kts

```
abstract class Person(val name: String)

abstract class MyPluginExtensionNamedDomainObjectList {
    // Define a named domain object list to hold Person objects
    private val people: NamedDomainObjectList<Person> =
    project.objects.namedDomainObjectList(Person::class)

    // Add a person to the container
    fun addPerson(name: String) {
        people.plus(name)
    }
}
```

build.gradle

```
abstract class Person {
    String name
}

abstract class MyPluginExtensionNamedDomainObjectList {
    // Define a named domain object container to hold Person objects
    NamedDomainObjectList<Person> people = project.container(Person)

    // Add a person to the container
    void addPerson(String name) {
        people.create(name: name)
    }
}
```

4. NamedDomainObjectContainer

A **NamedDomainObjectContainer** manages a set of objects, where each element has a name associated with it.

The container takes care of creating and configuring the elements, and provides a DSL that build scripts can use to define and configure elements. It is intended to hold objects which are themselves configurable, for example a set of custom Gradle objects.

Gradle uses **NamedDomainObjectContainer** type extensively throughout the API. For example, the **project.tasks** object used to manage the tasks of a project is a **NamedDomainObjectContainer<Task>**.

You can create a container instance using the **ObjectFactory** service, which provides the **ObjectFactory.domainObjectContainer()** method. This is also available using the **Project.container()** method, however in a custom Gradle type it's generally better to use the injected **ObjectFactory** service instead of passing around a **Project** instance.

You can also create a container instance using a **read-only managed property**.

build.gradle.kts

```
abstract class Person(val name: String)

abstract class MyPluginExtensionNamedDomainObjectContainer {
    // Define a named domain object container to hold Person objects
    private val people: NamedDomainObjectContainer<Person> =
        project.container(Person::class)

    // Add a person to the container
    fun addPerson(name: String) {
```

```

        people.create(name)
    }
}

```

build.gradle

```

abstract class Person {
    String name
}

abstract class MyPluginExtensionNamedDomainObjectContainer {
    // Define a named domain object container to hold Person objects
    NamedDomainObjectContainer<Person> people = project.container(Person)

    // Add a person to the container
    void addPerson(String name) {
        people.create(name: name)
    }
}

```

In order to use a type with any of the `domainObjectContainer()` methods, it must either

- be a [named managed type](#); or
- expose a property named “name” as the unique, and constant, name for the object. The `domainObjectContainer(Class)` variant of the method creates new instances by calling the constructor of the class that takes a string argument, which is the desired name of the object.

Objects created this way are treated as custom Gradle types, and so can make use of the features discussed in this chapter, for example service injection or managed properties.

See the above link for `domainObjectContainer()` method variants that allow custom instantiation strategies:

```

public interface DownloadExtension {
    NamedDomainObjectContainer<Resource> getResources();
}

public interface Resource {
    // Type must have a read-only 'name' property
    String getName();

    Property<URI> getUri();

    Property<String> getUserName();
}

```

For each container property, Gradle automatically adds a block to the Groovy and Kotlin DSL that you can use to configure the contents of the container:

build.gradle.kts

```
plugins {
    id("org.gradle.sample.download")
}

download {
    // Can use a block to configure the container contents
    resources {
        register("gradle") {
            uri = uri("https://gradle.org")
        }
    }
}
```

build.gradle

```
plugins {
    id("org.gradle.sample.download")
}

download {
    // Can use a block to configure the container contents
    resources {
        register('gradle') {
            uri = uri('https://gradle.org')
        }
    }
}
```

5. **ExtensiblePolymorphicDomainObjectContainer**

An **ExtensiblePolymorphicDomainObjectContainer** is a **NamedDomainObjectContainer** that allows you to define instantiation strategies for different types of objects.

You can create an instance using the **ObjectFactory.polymorphicDomainObjectContainer()** method:

build.gradle.kts

```
abstract class Animal(val name: String)

class Dog(name: String, val breed: String) : Animal(name)
```

```

abstract class
MyPluginExtensionExtensiblePolymorphicDomainObjectContainer(objectFactory:
ObjectFactory) {
    // Define a container for animals
    private val animals: ExtensiblePolymorphicDomainObjectContainer<Animal> =
objectFactory.polymorphicDomainObjectContainer(Animal::class)

    // Add a dog to the container
    fun addDog(name: String, breed: String) {
        var dog : Dog = Dog(name, breed)
        animals.add(dog)
    }
}

```

build.gradle

```

abstract class Animal {
    String name
}

abstract class Dog extends Animal {
    String breed
}

abstract class MyPluginExtensionExtensiblePolymorphicDomainObjectContainer {
    // Define a container for animals
    ExtensiblePolymorphicDomainObjectContainer<Animal> animals

    MyPluginExtensionExtensiblePolymorphicDomainObjectContainer(ObjectFactory
objectFactory) {
        // Create the container
        animals = objectFactory.polymorphicDomainObjectContainer(Animal)
    }

    // Add a dog to the container
    void addDog(String name, String breed) {
        animals.create(Dog, name: name, breed: breed)
    }
}

```

Understanding Services and Service Injection

Gradle provides a number of useful services that can be used by custom Gradle types. For example, the [WorkerExecutor](#) service can be used by a task to run work in parallel, as seen in the [worker API](#) section. The services are made available through *service injection*.

Available services

The following services are available for injection:

1. **ObjectFactory** - Allows model objects to be created.
2. **ProjectLayout** - Provides access to key project locations.
3. **BuildLayout** - Provides access to important locations for a Gradle build.
4. **ProviderFactory** - Creates **Provider** instances.
5. **WorkerExecutor** - Allows a task to run work in parallel.
6. **FileSystemOperations** - Allows a task to run operations on the filesystem such as deleting files, copying files or syncing directories.
7. **ArchiveOperations** - Allows a task to run operations on archive files such as ZIP or TAR files.
8. **ExecOperations** - Allows a task to run external processes with dedicated support for running external **java** programs.
9. **ToolingModelBuilderRegistry** - Allows a plugin to registers a Gradle tooling API model.
10. **TestEventReporterFactory** - Allows a plugin to access Gradle's test events and its corresponding API.

Out of the above, **ProjectLayout** and **WorkerExecutor** services are only available for injection in project plugins. **BuildLayout** is only available in settings plugins and settings files. **ProjectLayout** is unavailable in Worker API actions.

While other services can technically be injected, this practice is not supported and may lead to breaking changes in the future. Only inject the services explicitly listed above to ensure stability and compatibility.

WARNING | You should avoid injecting types not listed above.

1. ObjectFactory

ObjectFactory is a service for creating custom Gradle types, allowing you to define nested objects and DSLs in your build logic. It provides methods for creating instances of different types, such as properties (**Property<T>**), collections (**ListProperty<T>**, **SetProperty<T>**, **MapProperty<K, V>**), file-related objects (**RegularFileProperty**, **DirectoryProperty**, **ConfigurableFileCollection**, **ConfigurableFileTree**), and more.

You can obtain an instance of **ObjectFactory** using the **project.objects** property. Here's a simple example demonstrating how to use **ObjectFactory** to create a property and set its value:

build.gradle.kts

```
tasks.register("myObjectFactoryTask") {
    doLast {
        val objectFactory = project.objects
```

```

        val myProperty = objectFactory.property(String::class)
        myProperty.set("Hello, Gradle!")
        println(myProperty.get())
    }
}

```

build.gradle

```

tasks.register("myObjectFactoryTask") {
    doLast {
        def objectFactory = project.objects
        def myProperty = objectFactory.property(String)
        myProperty.set("Hello, Gradle!")
        println myProperty.get()
    }
}

```

TIP It is preferable to let Gradle create objects automatically by using [managed properties](#).

Using `ObjectFactory` to create these objects ensures that they are properly managed by Gradle, especially in terms of configuration avoidance and lazy evaluation. This means that the values of these objects are only calculated when needed, which can improve build performance.

In the following example, a project extension called `DownloadExtension` receives an `ObjectFactory` instance through its constructor. The constructor uses this to create a nested `Resource` object (also a custom Gradle type) and makes this object available through the `resource` property:

DownloadExtension.java

```

public class DownloadExtension {
    // A nested instance
    private final Resource resource;

    @Inject
    public DownloadExtension(ObjectFactory objectFactory) {
        // Use an injected ObjectFactory to create a Resource object
        resource = objectFactory.newInstance(Resource.class);
    }

    public Resource getResource() {
        return resource;
    }
}

public interface Resource {
    Property<URI> getUri();
}

```

```
}
```

Here is another example using `javax.inject.Inject`:

build.gradle.kts

```
abstract class MyObjectFactoryTask
@Inject constructor(private var objectFactory: ObjectFactory) : DefaultTask()
{

    @TaskAction
    fun doTaskAction() {
        val outputDirectory = objectFactory.directoryProperty()
        outputDirectory.convention(project.layout.projectDirectory)
        println(outputDirectory.get())
    }
}

tasks.register("myInjectedObjectFactoryTask", MyObjectFactoryTask::class) {}
```

build.gradle

```
abstract class MyObjectFactoryTask extends DefaultTask {
    private ObjectFactory objectFactory

    @Inject //@javax.inject.Inject
    MyObjectFactoryTask(ObjectFactory objectFactory) {
        this.objectFactory = objectFactory
    }

    @TaskAction
    void doTaskAction() {
        var outputDirectory = objectFactory.directoryProperty()
        outputDirectory.convention(project.layout.projectDirectory)
        println(outputDirectory.get())
    }
}

tasks.register("myInjectedObjectFactoryTask", MyObjectFactoryTask) {}
```

The `MyObjectFactoryTask` task uses an `ObjectFactory` instance, which is injected into the task's constructor using the `@Inject` annotation.

2. ProjectLayout

ProjectLayout is a service that provides access to the layout of a Gradle project's directories and files. It's part of the `org.gradle.api.file` package and allows you to query the project's layout to get information about source sets, build directories, and other file-related aspects of the project.

You can obtain a **ProjectLayout** instance from a **Project** object using the `project.layout` property. Here's a simple example:

build.gradle.kts

```
tasks.register("showLayout") {
    doLast {
        val layout = project.layout
        println("Project Directory: ${layout.projectDirectory}")
        println("Build Directory: ${layout.buildDirectory.get()}")
    }
}
```

build.gradle

```
tasks.register('showLayout') {
    doLast {
        def layout = project.layout
        println "Project Directory: ${layout.projectDirectory}"
        println "Build Directory: ${layout.buildDirectory.get()}"
    }
}
```

Here is an example using `javax.inject.Inject`:

build.gradle.kts

```
abstract class MyProjectLayoutTask
@Inject constructor(private var projectLayout: ProjectLayout) : DefaultTask()
{

    @TaskAction
    fun doTaskAction() {
        val outputDirectory = projectLayout.projectDirectory
        println(outputDirectory)
    }
}
```

```
tasks.register("myInjectedProjectLayoutTask", MyProjectLayoutTask::class) {}
```

build.gradle

```
abstract class MyProjectLayoutTask extends DefaultTask {
    private ProjectLayout projectLayout

    @Inject // @javax.inject.Inject
    MyProjectLayoutTask(ProjectLayout projectLayout) {
        this.projectLayout = projectLayout
    }

    @TaskAction
    void doTaskAction() {
        var outputDirectory = projectLayout.projectDirectory
        println(outputDirectory)
    }
}

tasks.register("myInjectedProjectLayoutTask", MyProjectLayoutTask) {}
```

The `MyProjectLayoutTask` task uses a `ProjectLayout` instance, which is injected into the task's constructor using the `@Inject` annotation.

3. BuildLayout

`BuildLayout` is a service that provides access to the root and settings directory in a Settings plugin or a Settings script, it is analogous to `ProjectLayout`. It's part of the `org.gradle.api.file` package to access standard build-wide file system locations as lazily computed value.

NOTE

These APIs are currently incubating but eventually should replace existing accessors in Settings, which return eagerly computed locations:

`Settings.rootDir` → `Settings.layout.rootDirectory`

`Settings.settingsDir` → `Settings.layout.settingsDirectory`

You can obtain a `BuildLayout` instance from a `Settings` object using the `settings.layout` property. Here's a simple example:

settings.gradle.kts

```
println("Root Directory: ${settings.layout.rootDirectory}")
println("Settings Directory: ${settings.layout.settingsDirectory}")
```

settings.gradle

```
println "Root Directory: ${settings.layout.rootDirectory}"
println "Settings Directory: ${settings.layout.settingsDirectory}"
```

Here is an example using `javax.inject.Inject`:

settings.gradle.kts

```
abstract class MyBuildLayoutPlugin @Inject constructor(private val
buildLayout: BuildLayout) : Plugin<Settings> {
    override fun apply(settings: Settings) {
        println(buildLayout.rootDirectory)
    }
}

apply<MyBuildLayoutPlugin>()
```

settings.gradle

```
abstract class MyBuildLayoutPlugin implements Plugin<Settings> {
    private BuildLayout buildLayout

    @Inject // @javax.inject.Inject
    MyBuildLayoutPlugin(BuildLayout buildLayout) {
        this.buildLayout = buildLayout
    }

    @Override void apply(Settings settings) {
        // the meat and potatoes of the plugin
        println buildLayout.rootDirectory
    }
}

apply plugin: MyBuildLayoutPlugin
```

This code defines a `MyBuildLayoutPlugin` plugin that implements the `Plugin` interface for the `Settings` type. The plugin expects a `BuildLayout` instance to be injected into its constructor using the `@Inject` annotation.

4. ProviderFactory

ProviderFactory is a service that provides methods for creating different types of providers. Providers are used to model values that may be computed lazily in your build scripts.

The **ProviderFactory** interface provides methods for creating various types of providers, including:

- **provider(Callable<T> value)** to create a provider with a value that is lazily computed based on a **Callable**.
- **provider(Provider<T> value)** to create a provider that simply wraps an existing provider.
- **property(Class<T> type)** to create a property provider for a specific type.
- **gradleProperty(Class<T> type)** to create a property provider that reads its value from a Gradle project property.

Here's a simple example demonstrating the use of **ProviderFactory** using **project.providers**:

build.gradle.kts

```
tasks.register("printMessage") {
    doLast {
        val providerFactory = project.providers
        val messageProvider = providerFactory.provider { "Hello, Gradle!" }
        println(messageProvider.get())
    }
}
```

build.gradle

```
tasks.register('printMessage') {
    doLast {
        def providerFactory = project.providers
        def messageProvider = providerFactory.provider { "Hello, Gradle!" }
        println messageProvider.get()
    }
}
```

The task named **printMessage** uses the **ProviderFactory** to create a **provider** that supplies the message string.

Here is an example using **javax.inject.Inject**:

build.gradle.kts

```
abstract class MyProviderFactoryTask
@Inject constructor(private var providerFactory: ProviderFactory) :
DefaultTask() {

    @TaskAction
    fun doTaskAction() {
        val outputDirectory = providerFactory.provider { "build/my-file.txt"
    }
        println(outputDirectory.get())
    }
}

tasks.register("myInjectedProviderFactoryTask", MyProviderFactoryTask::class)
{}
```

build.gradle

```
abstract class MyProviderFactoryTask extends DefaultTask {
    private ProviderFactory providerFactory

    @Inject //@javax.inject.Inject
    MyProviderFactoryTask(ProviderFactory providerFactory) {
        this.providerFactory = providerFactory
    }

    @TaskAction
    void doTaskAction() {
        var outputDirectory = providerFactory.provider { "build/my-file.txt"
    }
        println(outputDirectory.get())
    }
}

tasks.register("myInjectedProviderFactoryTask", MyProviderFactoryTask) {}
```

The `ProviderFactory` service is injected into the `MyProviderFactoryTask` task's constructor using the `@Inject` annotation.

5. WorkerExecutor

`WorkerExecutor` is a service that allows you to perform parallel execution of tasks using worker processes. This is particularly useful for tasks that perform CPU-intensive or long-running

operations, as it allows them to be executed in parallel, improving build performance.

Using `WorkerExecutor`, you can submit units of work (called actions) to be executed in separate worker processes. This helps isolate the work from the main Gradle process, providing better reliability and performance.

Here's a basic example of how you might use `WorkerExecutor` in a build script:

build.gradle.kts

```
abstract class MyWorkAction : WorkAction<WorkParameters.None> {
    private val greeting: String = "Hello from a Worker!"

    override fun execute() {
        println(greeting)
    }
}

abstract class MyWorkerTask
@Inject constructor(private var workerExecutor: WorkerExecutor) :
DefaultTask() {
    @get:Input
    abstract val booleanFlag: Property<Boolean>
    @TaskAction
    fun doThings() {
        workerExecutor.noIsolation().submit(MyWorkAction::class.java) {}
    }
}

tasks.register("myWorkTask", MyWorkerTask::class) {}
```

build.gradle

```
abstract class MyWorkAction implements WorkAction<WorkParameters.None> {
    private final String greeting;

    @Inject
    public MyWorkAction() {
        this.greeting = "Hello from a Worker!";
    }

    @Override
    public void execute() {
        System.out.println(greeting);
    }
}
```

```

abstract class MyWorkerTask extends DefaultTask {
    @Input
    abstract Property<Boolean> getBooleanFlag()

    @Inject
    abstract WorkerExecutor getWorkerExecutor()

    @TaskAction
    void doThings() {
        workerExecutor.noIsolation().submit(MyWorkAction) {}
    }
}

tasks.register("myWorkTask", MyWorkerTask) {}

```

See [the worker API](#) for more details.

6. FileSystemOperations

FileSystemOperations is a service that provides methods for performing file system operations such as copying, deleting, and syncing. It is part of the `org.gradle.api.file` package and is typically used in custom tasks or plugins to interact with the file system.

Here is an example using `javax.inject.Inject`:

build.gradle.kts

```

abstract class MyFileSystemOperationsTask
@Inject constructor(private var fileSystemOperations: FileSystemOperations) :
DefaultTask() {

    @TaskAction
    fun doTaskAction() {
        fileSystemOperations.sync {
            from("src")
            into("dest")
        }
    }
}

tasks.register("myInjectedFileSystemOperationsTask",
MyFileSystemOperationsTask::class)

```

build.gradle

```
abstract class MyFileSystemOperationsTask extends DefaultTask {
    private FileSystemOperations fileSystemOperations

    @Inject // @javax.inject.Inject
    MyFileSystemOperationsTask(FileSystemOperations fileSystemOperations) {
        this.fileSystemOperations = fileSystemOperations
    }

    @TaskAction
    void doTaskAction() {
        fileSystemOperations.sync {
            from 'src'
            into 'dest'
        }
    }
}

tasks.register("myInjectedFileSystemOperationsTask",
    MyFileSystemOperationsTask)
```

The `FileSystemOperations` service is injected into the `MyFileSystemOperationsTask` task's constructor using the `@Inject` annotation.

With some ceremony, it is possible to use `FileSystemOperations` in an ad-hoc task defined in a build script:

build.gradle.kts

```
interface InjectedFsOps {
    @get:Inject val fs: FileSystemOperations
}

tasks.register("myAdHocFileSystemOperationsTask") {
    val injected = project.objects.newInstance<InjectedFsOps>()
    doLast {
        injected.fs.copy {
            from("src")
            into("dest")
        }
    }
}
```

build.gradle

```
interface InjectedFsOps {
    @Inject //@javax.inject.Inject
    FileSystemOperations getFs()
}

tasks.register('myAdHocFileSystemOperationsTask') {
    def injected = project.objects.newInstance(InjectedFsOps)
    doLast {
        injected.fs.copy {
            from 'source'
            into 'destination'
        }
    }
}
```

First, you need to declare an interface with a property of type `FileSystemOperations`, here named `InjectedFsOps`, to serve as an injection point. Then call the method `ObjectFactory.newInstance` to generate an implementation of the interface that holds an injected service.

TIP This is a good time to consider extracting the ad-hoc task into a proper class.

7. ArchiveOperations

`ArchiveOperations` is a service that provides methods for accessing the contents of archives, such as ZIP and TAR files. It is part of the `org.gradle.api.file` package and is typically used in custom tasks or plugins to unpack archive files.

Here is an example using `javax.inject.Inject`:

build.gradle.kts

```
abstract class MyArchiveOperationsTask
@Inject constructor(
    private val archiveOperations: ArchiveOperations,
    private val layout: ProjectLayout,
    private val fs: FileSystemOperations
) : DefaultTask() {
    @TaskAction
    fun doTaskAction() {
        fs.sync {

            from(archiveOperations.zipTree(layout.projectDirectory.file("sources.jar")))
            into(layout.buildDirectory.dir("unpacked-sources"))
        }
    }
}
```

```

    }
}

tasks.register("myInjectedArchiveOperationsTask",
MyArchiveOperationsTask::class)

```

build.gradle

```

abstract class MyArchiveOperationsTask extends DefaultTask {
    private ArchiveOperations archiveOperations
    private ProjectLayout layout
    private FileSystemOperations fs

    @Inject
    MyArchiveOperationsTask(ArchiveOperations archiveOperations,
ProjectLayout layout, FileSystemOperations fs) {
        this.archiveOperations = archiveOperations
        this.layout = layout
        this.fs = fs
    }

    @TaskAction
    void doTaskAction() {
        fs.sync {
            from/archiveOperations.zipTree(layout.projectDirectory.file(
"sources.jar"))
            into(layout.buildDirectory.dir("unpacked-sources"))
        }
    }
}

tasks.register("myInjectedArchiveOperationsTask", MyArchiveOperationsTask)

```

The `ArchiveOperations` service is injected into the `MyArchiveOperationsTask` task's constructor using the `@Inject` annotation.

With some ceremony, it is possible to use `ArchiveOperations` in an ad-hoc task defined in a build script:

build.gradle.kts

```

interface InjectedArcOps {
    @get:Inject val arcOps: ArchiveOperations
}

```

```
tasks.register("myAdHocArchiveOperationsTask") {
    val injected = project.objects.newInstance<InjectedArcOps>()
    val archiveFile = "${project.projectDir}/sources.jar"
    doLast {
        injected.arcOps.zipTree(archiveFile)
    }
}
```

build.gradle

```
interface InjectedArcOps {
    @Inject //@javax.inject.Inject
    ArchiveOperations getArcOps()
}

tasks.register('myAdHocArchiveOperationsTask') {
    def injected = project.objects.newInstance(InjectedArcOps)
    def archiveFile = "${project.projectDir}/sources.jar"

    doLast {
        injected.arcOps.zipTree(archiveFile)
    }
}
```

First, you need to declare an interface with a property of type `ArchiveOperations`, here named `InjectedArcOps`, to serve as an injection point. Then call the method `ObjectFactory.newInstance` to generate an implementation of the interface that holds an injected service.

TIP This is a good time to consider extracting the ad-hoc task into a proper class.

8. ExecOperations

`ExecOperations` is a service that provides methods for executing external processes (commands) from within a build script. It is part of the `org.gradle.process` package and is typically used in custom tasks or plugins to run command-line tools or scripts as part of the build process.

Here is an example using `javax.inject.Inject`:

build.gradle.kts

```
abstract class MyExecOperationsTask
@Inject constructor(private var execOperations: ExecOperations) :
DefaultTask() {
```

```

@TaskAction
fun doTaskAction() {
    execOperations.exec {
        commandLine("ls", "-la")
    }
}

tasks.register("myInjectedExecOperationsTask", MyExecOperationsTask::class)

```

build.gradle

```

abstract class MyExecOperationsTask extends DefaultTask {
    private ExecOperations execOperations

    @Inject //@javax.inject.Inject
    MyExecOperationsTask(ExecOperations execOperations) {
        this.execOperations = execOperations
    }

    @TaskAction
    void doTaskAction() {
        execOperations.exec {
            commandLine 'ls', '-la'
        }
    }
}

tasks.register("myInjectedExecOperationsTask", MyExecOperationsTask)

```

The `ExecOperations` is injected into the `MyExecOperationsTask` task's constructor using the `@Inject` annotation.

With some ceremony, it is possible to use `ExecOperations` in an ad-hoc task defined in a build script:

build.gradle.kts

```

interface InjectedExecOps {
    @get:Inject val execOps: ExecOperations
}

tasks.register("myAdHocExecOperationsTask") {
    val injected = project.objects.newInstance<InjectedExecOps>()

    doLast {

```

```

        injected.execOps.exec {
            commandLine("ls", "-la")
        }
    }
}

```

build.gradle

```

interface InjectedExecOps {
    @Inject //@javax.inject.Inject
    ExecOperations getExecOps()
}

tasks.register('myAdHocExecOperationsTask') {
    def injected = project.objects.newInstance(InjectedExecOps)

    doLast {
        injected.execOps.exec {
            commandLine 'ls', '-la'
        }
    }
}

```

First, you need to declare an interface with a property of type `ExecOperations`, here named `InjectedExecOps`, to serve as an injection point. Then call the method `ObjectFactory.newInstance` to generate an implementation of the interface that holds an injected service.

TIP This is a good time to consider extracting the ad-hoc task into a proper class.

9. ToolingModelBuilderRegistry

`ToolingModelBuilderRegistry` is a service that allows you to register custom tooling model builders. Tooling models are used to provide rich IDE integration for Gradle projects, allowing IDEs to understand and work with the project's structure, dependencies, and other aspects.

The `ToolingModelBuilderRegistry` interface is part of the `org.gradle.tooling.provider.model` package and is typically used in custom Gradle plugins that provide enhanced IDE support.

Here's a simplified example:

build.gradle.kts

```

// Implements the ToolingModelBuilder interface.
// This interface is used in Gradle to define custom tooling models that can

```

```
// be accessed by IDEs or other tools through the Gradle tooling API.
class OrtModelBuilder : ToolingModelBuilder {
    private val repositories: MutableMap<String, String> = mutableMapOf()

    private val platformCategories: Set<String> = setOf("platform",
"enforced-platform")

    private val visitedDependencies: MutableSet<ModuleComponentIdentifier> =
mutableSetOf()
    private val visitedProjects: MutableSet<ModuleVersionIdentifier> =
mutableSetOf()

    private val logger = Logging.getLogger(OrtModelBuilder::class.java)
    private val errors: MutableList<String> = mutableListOf()
    private val warnings: MutableList<String> = mutableListOf()

    override fun canBuild(modelName: String): Boolean {
        return false
    }

    override fun buildAll(modelName: String, project: Project): Any? {
        return null
    }
}

// Plugin is responsible for registering a custom tooling model builder
// (OrtModelBuilder) with the ToolingModelBuilderRegistry, which allows
// IDEs and other tools to access the custom tooling model.
class OrtModelPlugin(private val registry: ToolingModelBuilderRegistry) :
Plugin<Project> {
    override fun apply(project: Project) {
        registry.register(OrtModelBuilder())
    }
}
}
```

build.gradle

```
// Implements the ToolingModelBuilder interface.
// This interface is used in Gradle to define custom tooling models that can
// be accessed by IDEs or other tools through the Gradle tooling API.
class OrtModelBuilder implements ToolingModelBuilder {
    private Map<String, String> repositories = [:]

    private Set<String> platformCategories = ["platform", "enforced-platform"
"]

    private Set<ModuleComponentIdentifier> visitedDependencies = []
    private Set<ModuleVersionIdentifier> visitedProjects = []
}
```

```

private static final logger = Logging.getLogger(OrtModelBuilder.class)
private List<String> errors = []
private List<String> warnings = []

@Override
boolean canBuild(String modelName) {
    return false
}

@Override
Object buildAll(String modelName, Project project) {
    return null
}
}

// Plugin is responsible for registering a custom tooling model builder
// (OrtModelBuilder) with the ToolingModelBuilderRegistry, which allows
// IDEs and other tools to access the custom tooling model.
class OrtModelPlugin implements Plugin<Project> {
    ToolingModelBuilderRegistry registry

    OrtModelPlugin(ToolingModelBuilderRegistry registry) {
        this.registry = registry
    }

    void apply(Project project) {
        registry.register(new OrtModelBuilder())
    }
}

```

You can learn more at [Tooling API](#).

10. TestEventReporterFactory

WARNING | This API is incubating.

TestEventReporterFactory is a service that provides access to the test event reporting API.

You can learn more at [Test Reporting API](#).

Constructor injection

There are 2 ways that an object can receive the services that it needs. The first option is to add the service as a parameter of the class constructor. The constructor must be annotated with the `javax.inject.Inject` annotation. Gradle uses the declared type of each constructor parameter to determine the services that the object requires. The order of the constructor parameters and their names are not significant and can be whatever you like.

Here is an example that shows a task type that receives an `ObjectFactory` via its constructor:

Download.java

```
public class Download extends DefaultTask {
    private final DirectoryProperty outputDirectory;

    // Inject an ObjectFactory into the constructor
    @Inject
    public Download(ObjectFactory objectFactory) {
        // Use the factory
        outputDirectory = objectFactory.directoryProperty();
    }

    @OutputDirectory
    public DirectoryProperty getOutputDirectory() {
        return outputDirectory;
    }

    @TaskAction
    void run() {
        // ...
    }
}
```

Property injection

Alternatively, a service can be injected by adding a property getter method annotated with the `javax.inject.Inject` annotation to the class. This can be useful, for example, when you cannot change the constructor of the class due to backwards compatibility constraints. This pattern also allows Gradle to defer creation of the service until the getter method is called, rather than when the instance is created. This can help with performance. Gradle uses the declared return type of the getter method to determine the service to make available. The name of the property is not significant and can be whatever you like.

The property getter method must be `public` or `protected`. The method can be `abstract` or, in cases where this isn't possible, can have a dummy method body. The method body is discarded.

Here is an example that shows a task type that receives a two services via property getter methods:

Download.java

```
public abstract class Download extends DefaultTask {
    // Use an abstract getter method
    @Inject
    protected abstract ObjectFactory getObjectFactory();

    // Alternatively, use a getter method with a dummy implementation
```

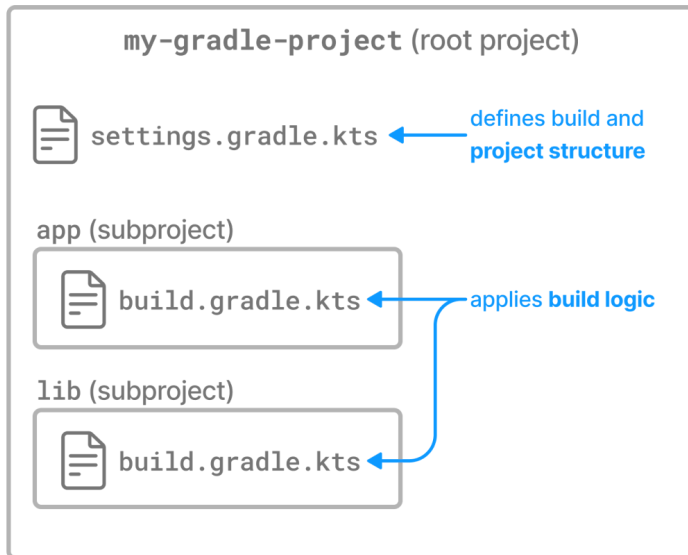
```
@Inject
protected WorkerExecutor getWorkerExecutor() {
    // Method body is ignored
    throw new UnsupportedOperationException();
}

@TaskAction
void run() {
    WorkerExecutor workerExecutor = getWorkerExecutor();
    ObjectFactory objectFactory = getObjectFactory();
    // Use the executor and factory ...
}
}
```

STRUCTURING BUILDS

Structuring Projects with Gradle

It is important to structure your Gradle project to optimize build performance. A multi-project build is the standard in Gradle.



A multi-project build consists of one root project and one or more subprojects. Gradle can build the root project and any number of the subprojects in a single execution.

Project locations

Multi-project builds contain a single root project in a directory that Gradle views as the root path: `..`

Subprojects are located physically under the root path: `./subproject`.

A subproject has a [path](#), which denotes the position of that subproject in the multi-project build. In most cases, the project path is consistent with its location in the file system.

The project structure is created in the `settings.gradle(.kts)` file. The settings file must be present in the root directory.

A simple multi-project build

Let's look at a *basic* multi-project build example that contains a root project and a single subproject.

The root project is called `basic-multiproject`, located somewhere on your machine. From Gradle's perspective, the root is the top-level directory `..`

The project contains a single subproject called `./app`:

```
•
```

```
├── app
│   └── ...
│       ├── build.gradle.kts
│       └── settings.gradle.kts
```

```
├── app
│   └── ...
│       ├── build.gradle
│       └── settings.gradle
```

This is the recommended project structure for starting any Gradle project. The [build init plugin](#) also generates skeleton projects that follow this structure - a root project with a single subproject:

The `settings.gradle(.kts)` file describes the project structure to Gradle:

settings.gradle.kts

```
rootProject.name = "basic-multiproject"
include("app")
```

settings.gradle

```
rootProject.name = 'basic-multiproject'
include 'app'
```

In this case, Gradle will look for a build file for the `app` subproject in the `./app` directory.

You can view the structure of a multi-project build by running the `projects` command:

```
$ ./gradlew -q projects
```

```
Projects:
```

```
-----
Root project 'basic-multiproject'
-----
```

```
Root project 'basic-multiproject'
\--- Project ':app'
```

To see a list of the tasks of a project, run `gradle <project-path>:tasks`
For example, try running `gradle :app:tasks`

In this example, the `app` subproject is a Java application that applies the [application plugin](#) and configures the main class. The application prints `Hello World` to the console:

app/build.gradle.kts

```
plugins {  
    id("application")  
}  
  
application {  
    mainClass = "com.example.Hello"  
}
```

app/build.gradle

```
plugins {  
    id 'application'  
}  
  
application {  
    mainClass = 'com.example.Hello'  
}
```

app/src/main/java/com/example/Hello.java

```
package com.example;  
  
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello, world!");  
    }  
}
```

You can run the application by executing the `run` task from the [application plugin](#) in the project root:

```
$ ./gradlew -q run  
Hello, world!
```

Adding a subproject

In the settings file, you can use the `include` method to add another subproject to the root project:

settings.gradle.kts

```
include("project1", "project2:child1", "project3:child1")
```

settings.gradle

```
include 'project1', 'project2:child1', 'project3:child1'
```

The `include` method takes [project paths](#) as arguments. The project path is assumed to be equal to the relative physical file system path. For example, a path `services:api` is mapped by default to a folder `./services/api` (relative to the project root `.`).

More examples of how to work with the project path can be found in the DSL documentation of [Settings.include\(java.lang.String\[\]\)](#).

Let's add another subproject called `lib` to the previously created project.

All we need to do is add another `include` statement in the root settings file:

settings.gradle.kts

```
rootProject.name = "basic-multiproject"
include("app")
include("lib")
```

settings.gradle

```
rootProject.name = 'basic-multiproject'
include 'app'
include 'lib'
```

Gradle will then look for the build file of the new `lib` subproject in the `./lib/` directory:

```
.
```

```
├── app
│   ├── ...
│   └── build.gradle.kts
├── lib
│   ├── ...
│   └── build.gradle.kts
└── settings.gradle.kts
```

```
.
├── app
│   ├── ...
│   └── build.gradle
├── lib
│   ├── ...
│   └── build.gradle
└── settings.gradle
```

Project Descriptors

To further describe the project architecture to Gradle, the settings file provides *project descriptors*.

You can modify these descriptors in the settings file at any time.

To access a descriptor, you can:

settings.gradle.kts

```
include("project-a")
println(rootProject.name)
println(project(":project-a").name)
```

settings.gradle

```
include('project-a')
println rootProject.name
println project(':project-a').name
```

Using this descriptor, you can change the name, project directory, and build file of a project:

settings.gradle.kts

```
rootProject.name = "main"
include("project-a")
project(":project-a").projectDir = file("custom/my-project-a")
project(":project-a").buildFileName = "project-a.gradle.kts"
```

settings.gradle

```
rootProject.name = 'main'
include('project-a')
project(':project-a').projectDir = file('custom/my-project-a')
project(':project-a').buildFileName = 'project-a.gradle'
```

Consult the [ProjectDescriptor](#) class in the API documentation for more information.

Modifying a subproject path

Let's take a hypothetical project with the following structure:

```
.
├── app
│   └── ...
│       └── build.gradle.kts
├── subs // Gradle may see this as a subproject
│   ├── web // Gradle may see this as a subproject
│   │   └── my-web-module // Intended subproject
│   └── ...
│       └── build.gradle.kts
└── settings.gradle.kts
```

```
.
├── app
│   └── ...
│       └── build.gradle
├── subs // Gradle may see this as a subproject
│   ├── web // Gradle may see this as a subproject
│   │   └── my-web-module // Intended subproject
│   └── ...
│       └── build.gradle
```

```
└── settings.gradle
```

If your `settings.gradle(.kts)` looks like this:

```
include(':subs:web:my-web-module')
```

Gradle sees a subproject with a logical project name of `:subs:web:my-web-module` and two, possibly unintentional, other subprojects logically named `:subs` and `:subs:web`. This can lead to phantom build directories, especially when using `allprojects{}` or `subproject{}`.

To avoid this, you can use:

```
include(':my-web-module')
project(':my-web-module').projectDir = "subs/web/my-web-module"
```

So that you only end up with a single subproject named `:my-web-module`.

So, while the physical project layout is the same, the logical results are different.

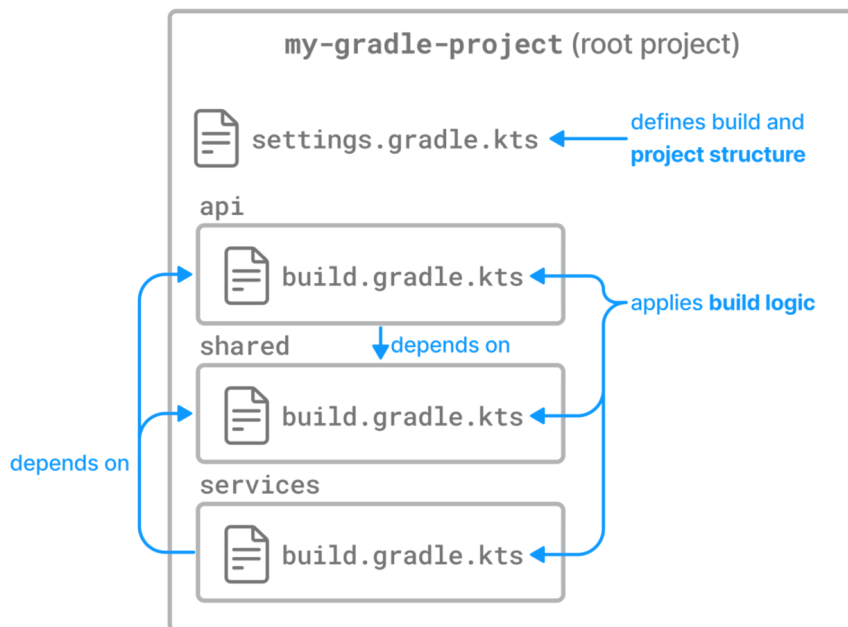
Naming recommendations

As your project grows, naming and consistency get increasingly more important. To keep your builds maintainable, we recommend the following:

1. **Keep default project names for subprojects:** It is possible to configure custom project names in the settings file. However, it's an unnecessary extra effort for the developers to track which projects belong to what folders.
2. **Use lower case hyphenation for all project names:** All letters are lowercase, and words are separated with a dash (-) character.
3. **Define the root project name in the settings file:** The `rootProject.name` effectively assigns a name to the build, used in reports like Build Scans. If the root project name is not set, the name will be the container directory name, which can be unstable (i.e., you can check out your project in any directory). The name will be generated randomly if the root project name is not set and checked out to a file system's root (e.g., / or `C:\`).

Declaring Dependencies between Subprojects

What if one subproject depends on another subproject? What if one project needs the artifact produced by another project?



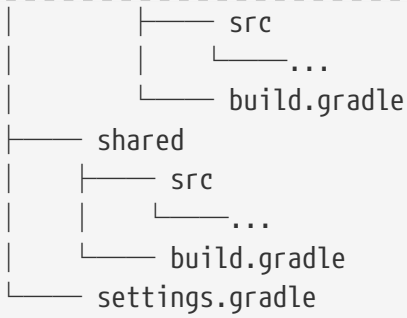
This is a common use case for multi-project builds. Gradle offers [project dependencies](#) for this.

Depending on another project

Let's explore a theoretical multi-project build with the following layout:

```
.
├── api
│   ├── src
│   │   └── ...
│   └── build.gradle.kts
├── services
│   ├── person-service
│   │   ├── src
│   │   │   └── ...
│   │   └── build.gradle.kts
│   └── build.gradle.kts
├── shared
│   ├── src
│   │   └── ...
│   └── build.gradle.kts
└── settings.gradle.kts
```

```
.
├── api
│   ├── src
│   │   └── ...
│   └── build.gradle
├── services
│   └── person-service
```



In this example, there are three subprojects called `shared`, `api`, and `person-service`:

1. The `person-service` subproject depends on the other two subprojects, `shared` and `api`.
2. The `api` subproject depends on the `shared` subproject.

We use the `:` separator to define a [project path](#) such as `services:person-service` or `:shared`. Consult the DSL documentation of [Settings.include\(java.lang.String\[\]\)](#) for more information about defining project paths.

settings.gradle.kts

```
rootProject.name = "dependencies-java"
include("api", "shared", "services:person-service")
```

shared/build.gradle.kts

```
plugins {
    id("java")
}

repositories {
    mavenCentral()
}

dependencies {
    testImplementation("junit:junit:4.13")
}
```

api/build.gradle.kts

```
plugins {
    id("java")
}

repositories {
    mavenCentral()
}
```

```
dependencies {
    testImplementation("junit:junit:4.13")
    implementation(project(":shared"))
}
```

services/person-service/build.gradle.kts

```
plugins {
    id("java")
}

repositories {
    mavenCentral()
}

dependencies {
    testImplementation("junit:junit:4.13")
    implementation(project(":shared"))
    implementation(project(":api"))
}
```

settings.gradle

```
rootProject.name = 'basic-dependencies'
include 'api', 'shared', 'services:person-service'
```

shared/build.gradle

```
plugins {
    id 'java'
}

repositories {
    mavenCentral()
}

dependencies {
    testImplementation "junit:junit:4.13"
}
```

api/build.gradle

```
plugins {
    id 'java'
}
```

```
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    testImplementation "junit:junit:4.13"  
    implementation project(':shared')  
}
```

services/person-service/build.gradle

```
plugins {  
    id 'java'  
}  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    testImplementation "junit:junit:4.13"  
    implementation project(':shared')  
    implementation project(':api')  
}
```

A project dependency affects execution order. It causes the other project to be built first and adds the output with the classes of the other project to the classpath. It also adds the dependencies of the other project to the classpath.

If you execute `./gradlew :api:compile`, first the `shared` project is built, and then the `api` project is built.

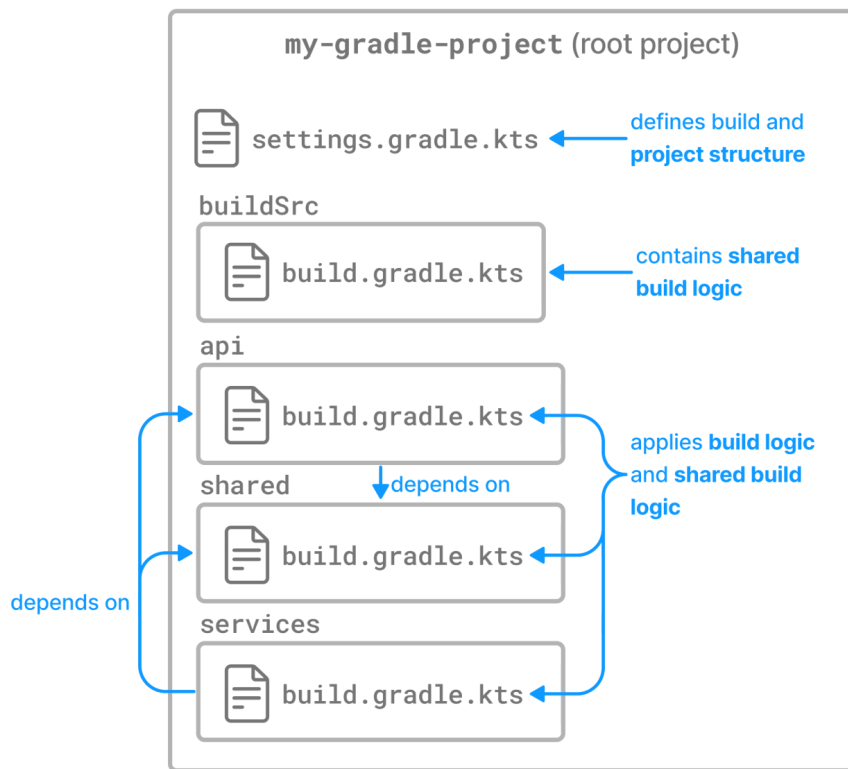
Depending on artifacts produced by another project

Sometimes, you might want to depend on the output of a specific task within another project rather than the entire project. However, explicitly declaring a task dependency from one project to another is discouraged as it introduces unnecessary coupling between tasks.

The recommended way to model dependencies, where a task in one project depends on the output of another, is to produce the output and mark it as an "outgoing" artifact. Gradle's [dependency management engine](#) allows you to share arbitrary artifacts between projects and build them on demand.

Sharing Build Logic between Subprojects

Subprojects in a multi-project build typically share some common dependencies.



Instead of copying and pasting the same Java version and libraries in each subproject build script, Gradle provides a special directory for storing shared build logic that can be automatically applied to subprojects.

Share logic in `buildSrc`

`buildSrc` is a Gradle-recognized and protected directory which comes with some benefits:

1. Reusable Build Logic:

`buildSrc` allows you to organize and centralize your custom build logic, tasks, and plugins in a structured manner. The code written in `buildSrc` can be reused across your project, making it easier to maintain and share common build functionality.

2. Isolation from the Main Build:

Code placed in `buildSrc` is isolated from the other build scripts of your project. This helps keep the main build scripts cleaner and more focused on project-specific configurations.

3. Automatic Compilation and Classpath:

The contents of the `buildSrc` directory are automatically compiled and included in the classpath of your main build. This means that classes and plugins defined in `buildSrc` can be directly used in your project's build scripts without any additional configuration.

4. Ease of Testing:

Since `buildSrc` is a separate build, it allows for easy testing of your custom build logic. You can write tests for your build code, ensuring that it behaves as expected.

5. Gradle Plugin Development:

If you are developing custom Gradle plugins for your project, `buildSrc` is a convenient place to house the plugin code. This makes the plugins easily accessible within your project.

The `buildSrc` directory is treated as an `included build`.

For multi-project builds, there can be only one `buildSrc` directory, which must be in the root project directory.

NOTE

The downside of using `buildSrc` is that any change to it will invalidate every task in your project and require a rerun.

`buildSrc` uses the same `source code conventions` applicable to Java, Groovy, and Kotlin projects. It also provides direct access to the Gradle API.

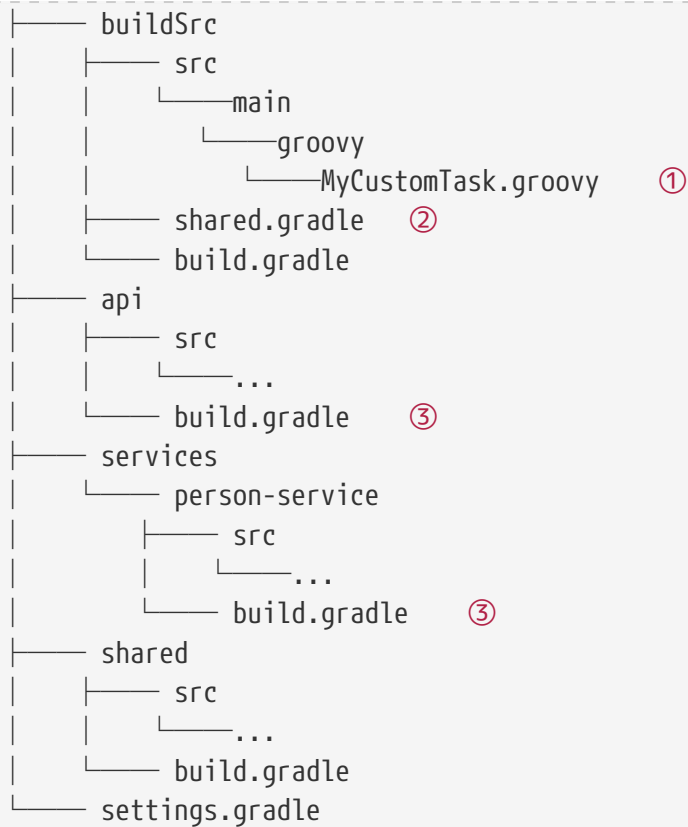
A typical project including `buildSrc` has the following layout:

```
.
├── buildSrc
│   ├── src
│   │   ├── main
│   │   │   └── kotlin
│   │   │       └── MyCustomTask.kt ①
│   └── shared.gradle.kts ②
│       └── build.gradle.kts
├── api
│   ├── src
│   │   └── ...
│   └── build.gradle.kts ③
├── services
│   ├── person-service
│   │   ├── src
│   │   │   └── ...
│   │   └── build.gradle.kts ③
├── shared
│   ├── src
│   │   └── ...
│   └── build.gradle.kts
└── settings.gradle.kts
```

① Create the `MyCustomTask` task.

② A shared build script.

③ Uses the `MyCustomTask` task and shared build script.



- ① Create the `MyCustomTask` task.
- ② A shared build script.
- ③ Uses the `MyCustomTask` task and shared build script.

In the `buildSrc`, the build script `shared.gradle(.kts)` is created. It contains dependencies and other build information that is common to multiple subprojects:

shared.gradle.kts

```
repositories {
    mavenCentral()
}

dependencies {
    implementation("org.slf4j:slf4j-api:1.7.32")
}
```

shared.gradle

```
repositories {
    mavenCentral()
}

dependencies {
```

```
        implementation 'org.slf4j:slf4j-api:1.7.32'
    }
```

In the `buildSrc`, the `MyCustomTask` is also created. It is a helper task that is used as part of the build logic for multiple subprojects:

MyCustomTask.kt

```
import org.gradle.api.DefaultTask
import org.gradle.api.tasks.TaskAction

open class MyCustomTask : DefaultTask() {
    @TaskAction
    fun calculateSum() {
        // Custom logic to calculate the sum of two numbers
        val num1 = 5
        val num2 = 7
        val sum = num1 + num2

        // Print the result
        println("Sum: $sum")
    }
}
```

MyCustomTask.groovy

```
import org.gradle.api.DefaultTask
import org.gradle.api.tasks.TaskAction

class MyCustomTask extends DefaultTask {
    @TaskAction
    void calculateSum() {
        // Custom logic to calculate the sum of two numbers
        int num1 = 5
        int num2 = 7
        int sum = num1 + num2

        // Print the result
        println "Sum: $sum"
    }
}
```

The `MyCustomTask` task is used in the build script of the `api` and `shared` projects. The task is automatically available because it's part of `buildSrc`.

The `shared.gradle(.kts)` file is also applied:

build.gradle.kts

```
// Apply any other configurations specific to your project

// Use the build script defined in buildSrc
apply(from = rootProject.file("buildSrc/shared.gradle.kts"))

// Use the custom task defined in buildSrc
tasks.register<MyCustomTask>("myCustomTask")
```

build.gradle

```
// Apply any other configurations specific to your project

// Use the build script defined in buildSrc
apply from: rootProject.file('buildSrc/shared.gradle')

// Use the custom task defined in buildSrc
tasks.register('myCustomTask', MyCustomTask)
```

Share logic using convention plugins

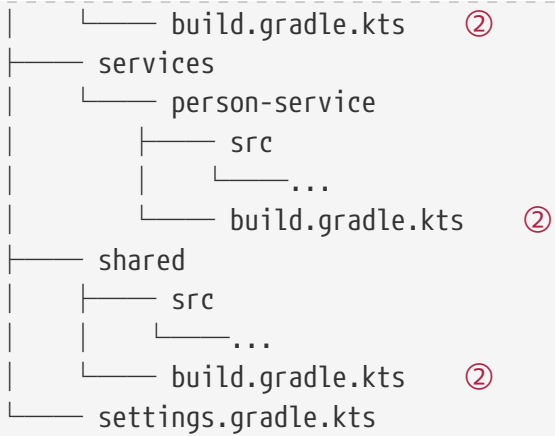
Gradle's recommended way of organizing build logic is to use its plugin system.

We can write a plugin that encapsulates the build logic common to several subprojects in a project. This kind of plugin is called a **convention plugin**.

While writing plugins is outside the scope of this section, the recommended way to build a Gradle project is to put common build logic in a convention plugin located in the `buildSrc`.

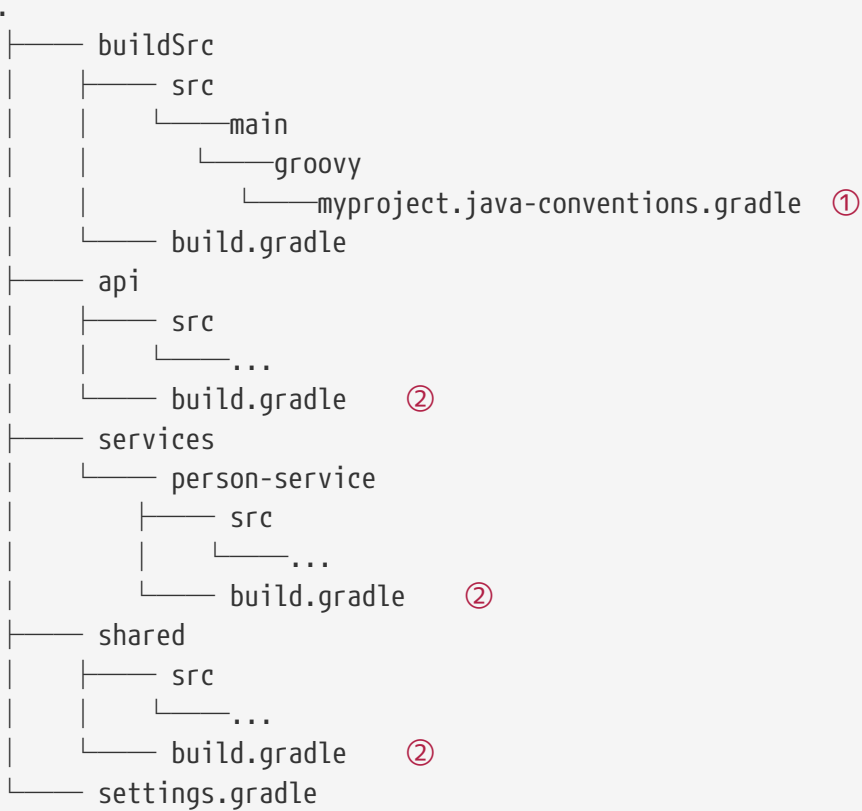
Let's take a look at an example project:

```
.
├── buildSrc
│   ├── src
│   │   ├── main
│   │   └── kotlin
│   └── myproject.java-conventions.gradle.kts ①
├── build.gradle.kts
├── api
│   └── src
│       └── ...
```



① Create the `myproject.java-conventions` convention plugin.

② Applies the `myproject.java-conventions` convention plugin.



① Create the `myproject.java-conventions` convention plugin.

② Applies the `myproject.java-conventions` convention plugin.

This build contains three subprojects:

settings.gradle.kts

```
rootProject.name = "dependencies-java"
include("api", "shared", "services:person-service")
```

settings.gradle

```
rootProject.name = 'dependencies-java'
include 'api', 'shared', 'services:person-service'
```

The source code for the convention plugin created in the `buildSrc` directory is as follows:

buildSrc/src/main/kotlin/myproject.java-conventions.gradle.kts

```
plugins {
    id("java")
}

group = "com.example"
version = "1.0"

repositories {
    mavenCentral()
}

dependencies {
    testImplementation("junit:junit:4.13")
}
```

buildSrc/src/main/groovy/myproject.java-conventions.gradle

```
plugins {
    id 'java'
}

group = 'com.example'
version = '1.0'

repositories {
    mavenCentral()
}
```

```
dependencies {  
    testImplementation "junit:junit:4.13"  
}
```

For the convention plugin to compile, basic configuration needs to be applied in the build file of the `buildSrc` directory:

buildSrc/build.gradle.kts

```
plugins {  
    `kotlin-dsl`  
}  
  
repositories {  
    mavenCentral()  
}
```

buildSrc/build.gradle

```
plugins {  
    id 'groovy-gradle-plugin'  
}
```

The convention plugin is applied to the `api`, `shared`, and `person-service` subprojects:

api/build.gradle.kts

```
plugins {  
    id("myproject.java-conventions")  
}  
  
dependencies {  
    implementation(project(":shared"))  
}
```

shared/build.gradle.kts

```
plugins {  
    id("myproject.java-conventions")  
}
```

```
}
```

services/person-service/build.gradle.kts

```
plugins {  
    id("myproject.java-conventions")  
}  
  
dependencies {  
    implementation(project(":shared"))  
    implementation(project(":api"))  
}
```

api/build.gradle

```
plugins {  
    id 'myproject.java-conventions'  
}  
  
dependencies {  
    implementation project(':shared')  
}
```

shared/build.gradle

```
plugins {  
    id 'myproject.java-conventions'  
}
```

services/person-service/build.gradle

```
plugins {  
    id 'myproject.java-conventions'  
}  
  
dependencies {  
    implementation project(':shared')  
    implementation project(':api')  
}
```

Do not use cross-project configuration

An improper way to share build logic between subprojects is *cross-project configuration* via the `subprojects {}` and `allprojects {}` DSL constructs.

TIP | Avoid using `subprojects {}` and `allprojects {}`.

With [cross-project configuration](#), build logic can be injected into a subproject which is not obvious when looking at its build script.

In the long run, cross-project configuration usually grows in complexity and becomes a burden. Cross-project configuration can also introduce configuration-time coupling between projects, which can prevent optimizations like configuration-on-demand from working properly.

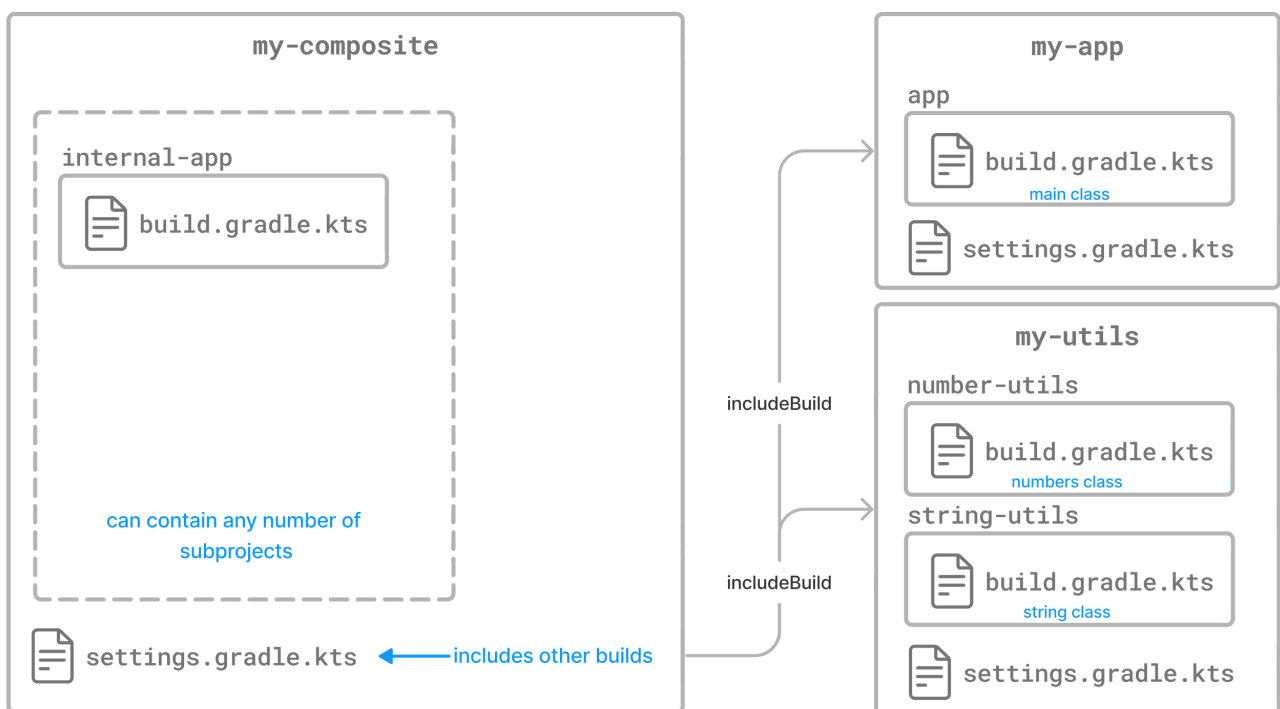
Convention plugins versus cross-project configuration

The two most common uses of cross-project configuration can be better modeled using convention plugins:

1. Applying plugins or other configurations to subprojects of a certain type.
Often, the cross-project configuration logic is `if subproject is of type X, then configure Y`. This is equivalent to applying `X-conventions` plugin directly to a subproject.
2. Extracting information from subprojects of a certain type.
This use case can be modeled using [outgoing configuration variants](#).

Composite Builds

A composite build is a build that includes other builds.



A composite build is similar to a Gradle multi-project build, except that instead of including `subprojects`, entire `builds` are included.

Composite builds allow you to:

- Combine builds that are usually developed independently, for instance, when trying out a bug

fix in a library that your application uses.

- Decompose a large multi-project build into smaller, more isolated chunks that can be worked on independently or together as needed.

A build that is included in a composite build is referred to as an **included build**. Included builds do not share any configuration with the composite build or the other included builds. Each included build is configured and executed in isolation.

Defining a composite build

The following example demonstrates how two Gradle builds, normally developed separately, can be combined into a composite build.

```
my-composite
├── gradle
├── gradlew
├── settings.gradle.kts
├── build.gradle.kts
├── my-app
│   ├── settings.gradle.kts
│   └── app
│       ├── build.gradle.kts
│       └── src/main/java/org/sample/my-app/Main.java
└── my-utils
    ├── settings.gradle.kts
    ├── number-utils
    │   ├── build.gradle.kts
    │   └── src/main/java/org/sample/numberutils/Numbers.java
    └── string-utils
        ├── build.gradle.kts
        └── src/main/java/org/sample/stringutils/Strings.java
```

The **my-utils** multi-project build produces two Java libraries, **number-utils** and **string-utils**. The **my-app** build produces an executable using functions from those libraries.

The **my-app** build does not depend directly on **my-utils**. Instead, it declares binary dependencies on the libraries produced by **my-utils**:

my-app/app/build.gradle.kts

```
plugins {
    id("application")
}

application {
    mainClass = "org.sample.myapp.Main"
}
```

```
dependencies {
    implementation("org.sample:number-utils:1.0")
    implementation("org.sample:string-utils:1.0")
}
```

my-app/app/build.gradle

```
plugins {
    id 'application'
}

application {
    mainClass = 'org.sample.myapp.Main'
}

dependencies {
    implementation 'org.sample:number-utils:1.0'
    implementation 'org.sample:string-utils:1.0'
}
```

Defining a composite build via `--include-build`

The `--include-build` command-line argument turns the executed build into a composite, substituting dependencies from the included build into the executed build.

For example, the output of `./gradlew run --include-build ../my-utils` run from `my-app`:

```
$ ./gradlew --include-build ../my-utils run
link:https://docs.gradle.org/8.13/samples/build-organization/composite-builds/basic/tests/basicCli.out[role=include]
```

Defining a composite build via the settings file

It's possible to make the above arrangement persistent by using `Settings.includeBuild(java.lang.Object)` to declare the included build in the `settings.gradle(.kts)` file.

The settings file can be used to add subprojects and included builds simultaneously.

Included builds are added by location:

settings.gradle.kts

```
includeBuild("my-utils")
```

In the example, the `settings.gradle(.kts)` file combines otherwise separate builds:

settings.gradle.kts

```
rootProject.name = "my-composite"

includeBuild("my-app")
includeBuild("my-utils")
```

settings.gradle

```
rootProject.name = 'my-composite'

includeBuild 'my-app'
includeBuild 'my-utils'
```

To execute the `run` task in the `my-app` build from `my-composite`, run `./gradlew my-app:app:run`.

You can optionally define a `run` task in `my-composite` that depends on `my-app:app:run` so that you can execute `./gradlew run`:

build.gradle.kts

```
tasks.register("run") {
    dependsOn(gradle.includedBuild("my-app").task(":app:run"))
}
```

build.gradle

```
tasks.register('run') {
    dependsOn gradle.includedBuild('my-app').task(':app:run')
}
```

Including builds that define Gradle plugins

A special case of included builds are builds that define Gradle plugins.

These builds should be included using the `includeBuild` statement inside the `pluginManagement {}` block of the settings file.

Using this mechanism, the included build may also contribute a settings plugin that can be applied in the settings file itself:

settings.gradle.kts

```
pluginManagement {  
    includeBuild("../url-verifier-plugin")  
}
```

settings.gradle

```
pluginManagement {  
    includeBuild '../url-verifier-plugin'  
}
```

Restrictions on included builds

Most builds can be included in a composite, including other composite builds. There are some restrictions.

In a regular build, Gradle ensures that each project has a unique *project path*. It makes projects identifiable and addressable without conflicts.

In a composite build, Gradle adds additional qualification to each project from an included build to avoid project path conflicts. The full path to identify a project in a composite build is called a *build-tree path*. It consists of a *build path* of an included build and a *project path* of the project.

By default, build paths and project paths are derived from directory names and structure on disk. Since included builds can be located anywhere on disk, their build path is determined by the name of the containing directory. This can sometimes lead to conflicts.

To summarize, the included builds must fulfill these requirements:

- Each included build must have a unique build path.
- Each included build path must not conflict with any project path of the main build.

These conditions guarantee that each project can be uniquely identified even in a composite build.

If conflicts arise, the way to resolve them is by changing the *build name* of an included build:

settings.gradle.kts

```
includeBuild("some-included-build") {  
    name = "other-name"
```

```
}
```

NOTE

When a composite build is included in another composite build, both builds have the same parent. In other words, the nested composite build structure is flattened.

Interacting with a composite build

Interacting with a composite build is generally similar to a regular multi-project build. Tasks can be executed, tests can be run, and builds can be imported into the IDE.

Executing tasks

Tasks from an included build can be executed from the command-line or IDE in the same way as tasks from a regular multi-project build. Executing a task will result in task dependencies being executed, as well as those tasks required to build dependency artifacts from other included builds.

You can call a task in an included build using a fully qualified path, for example, `:included-build-name:project-name:taskName`. Project and task names can be [abbreviated](#).

```
$ ./gradlew :included-build:subproject-a:compileJava
> Task :included-build:subproject-a:compileJava

$ ./gradlew :i-b:sA:cJ
> Task :included-build:subproject-a:compileJava
```

To [exclude a task from the command line](#), you need to provide the fully qualified path to the task.

NOTE

Included build tasks are automatically executed to generate required dependency artifacts, or the [including build can declare a dependency on a task from an included build](#).

Importing into the IDE

One of the most useful features of composite builds is IDE integration.

Importing a composite build permits sources from separate Gradle builds to be easily developed together. For every included build, each subproject is included as an IntelliJ IDEA Module or Eclipse Project. Source dependencies are configured, providing cross-build navigation and refactoring.

Declaring dependencies substituted by an included build

By default, Gradle will configure each included build to determine the dependencies it can provide. The algorithm for doing this is simple. Gradle will inspect the group and name for the projects in the included build and substitute project dependencies for any external dependency matching `${project.group}:${project.name}`.

NOTE

By default, substitutions are not registered for the *main* build.

To make the (sub)projects of the main build addressable by `${project.group}:${project.name}`, you can tell Gradle to treat the main build like an included build by self-including it: `includeBuild(".")`.

There are cases when the default substitutions determined by Gradle are insufficient or must be corrected for a particular composite. For these cases, explicitly declaring the substitutions for an included build is possible.

For example, a single-project build called `anonymous-library`, produces a Java utility library but does not declare a value for the group attribute:

build.gradle.kts

```
plugins {  
    java  
}
```

build.gradle

```
plugins {  
    id 'java'  
}
```

When this build is included in a composite, it will attempt to substitute for the dependency module `undefined:anonymous-library` (`undefined` being the default value for `project.group`, and `anonymous-library` being the root project name). Clearly, this isn't useful in a composite build.

To use the unpublished library in a composite build, you can explicitly declare the substitutions that it provides:

settings.gradle.kts

```
includeBuild("anonymous-library") {  
    dependencySubstitution {  
        substitute(module("org.sample:number-utils")).using(project(":"))  
    }  
}
```

settings.gradle

```
includeBuild('anonymous-library') {
```

```

dependencySubstitution {
    substitute module('org.sample:number-utils') using project(':')
}

```

With this configuration, the `my-app` composite build will substitute any dependency on `org.sample:number-utils` with a dependency on the root project of `anonymous-library`.

Deactivate included build substitutions for a configuration

If you need to [resolve](#) a published version of a module that is also available as part of an included build, you can deactivate the included build substitution rules on the [ResolutionStrategy](#) of the Configuration that is resolved. This is necessary because the rules are globally applied in the build, and Gradle does not consider published versions during resolution by default.

For example, we create a separate `publishedRuntimeClasspath` configuration that gets resolved to the published versions of modules that also exist in one of the local builds. This is done by deactivating global dependency substitution rules:

build.gradle.kts

```

configurations.create("publishedRuntimeClasspath") {
    resolutionStrategy.useGlobalDependencySubstitutionRules = false

    extendsFrom(configurations.runtimeClasspath.get())
    isCanBeConsumed = false
    attributes.attribute(Usage.USAGE_ATTRIBUTE,
        objects.named(Usage.JAVA_RUNTIME))
}

```

build.gradle

```

configurations.create('publishedRuntimeClasspath') {
    resolutionStrategy.useGlobalDependencySubstitutionRules = false

    extendsFrom(configurations.runtimeClasspath)
    canBeConsumed = false
    attributes.attribute(Usage.USAGE_ATTRIBUTE, objects.named(Usage, Usage
        .JAVA_RUNTIME))
}

```

A use-case would be to compare published and locally built JAR files.

Cases where included build substitutions must be declared

Many builds will function automatically as an included build, without declared substitutions. Here are some common cases where declared substitutions are required:

- When the `archivesBaseName` property is used to set the name of the published artifact.
- When a configuration other than `default` is published.
- When the `MavenPom.addFilter()` is used to publish artifacts that don't match the project name.
- When the `maven-publish` or `ivy-publish` plugins are used for publishing and the publication coordinates don't match `${project.group}:${project.name}`.

Cases where composite build substitutions won't work

Some builds won't function correctly when included in a composite, even when dependency substitutions are explicitly declared. This limitation is because a substituted project dependency will always point to the `default` configuration of the target project. Any time the artifacts and dependencies specified for the default configuration of a project don't match what is published to a repository, the composite build may exhibit different behavior.

Here are some cases where the published module metadata may be different from the project default configuration:

- When a configuration other than `default` is published.
- When the `maven-publish` or `ivy-publish` plugins are used.
- When the `POM` or `ivy.xml` file is tweaked as part of publication.

Builds using these features function incorrectly when included in a composite build.

Depending on tasks in an included build

While included builds are isolated from one another and cannot declare direct dependencies, a composite build can declare task dependencies on its included builds. The included builds are accessed using `Gradle.getIncludedBuilds()` or `Gradle.includedBuild(java.lang.String)`, and a task reference is obtained via the `IncludedBuild.task(java.lang.String)` method.

Using these APIs, it is possible to declare a dependency on a task in a particular included build:

build.gradle.kts

```
tasks.register("run") {
    dependsOn(gradle.includedBuild("my-app").task(":app:run"))
}
```

build.gradle

```
tasks.register('run') {  
    dependsOn gradle.includedBuild('my-app').task(':app:run')  
}
```

Or you can declare a dependency on tasks with a certain path in some or all of the included builds:

build.gradle.kts

```
tasks.register("publishDeps") {  
    dependsOn(gradle.includedBuilds.map {  
it.task(":publishMavenPublicationToMavenRepository") })  
}
```

build.gradle

```
tasks.register('publishDeps') {  
    dependsOn gradle.includedBuilds*.task(  
' :publishMavenPublicationToMavenRepository')  
}
```

Limitations of composite builds

Limitations of the current implementation include:

- No support for included builds with publications that don't mirror the project default configuration.
See [Cases where composite builds won't work](#).
- Multiple composite builds may conflict when run in parallel if more than one includes the same build.
Gradle does not share the project lock of a shared composite build between Gradle invocations to prevent concurrent execution.

Configuration On Demand

Configuration-on-demand attempts to configure only the relevant projects for the requested tasks, i.e., it only evaluates the build script file of projects participating in the build. This way, the configuration time of a large multi-project build can be reduced.

The configuration-on-demand feature is *incubating*, so only some builds are guaranteed to work correctly. The feature works well for [decoupled](#) multi-project builds.

In configuration-on-demand mode, projects are configured as follows:

- The root project is always configured.
- The project in the directory where the build is executed is also configured, but only when Gradle is executed without any tasks.
This way, the default tasks behave correctly when projects are configured on demand.
- The standard project dependencies are supported, and relevant projects are configured.
If project A has a compile dependency on project B, then building A causes the configuration of both projects.
- The task dependencies declared via the task path are supported and cause relevant projects to be configured.
Example: `someTask.dependsOn(":some-other-project:someOtherTask")`
- A task requested via task path from the command line (or tooling API) causes the relevant project to be configured.
For example, building `project-a:project-b:someTask` causes configuration of `project-b`.

Enable **configuration-on-demand**

You can enable configuration-on-demand using the `--configure-on-demand` flag or adding `org.gradle.configureondemand=true` to the `gradle.properties` file.

To configure on demand with every build run, see [Gradle properties](#).

To configure on demand for a given build, see [command-line performance-oriented options](#).

Decoupled projects

Gradle allows projects to access each other's configurations and tasks during the configuration and execution phases. While this flexibility empowers build authors, it limits Gradle's ability to perform optimizations such as [parallel project builds](#) and [configuration on demand](#).

Projects are considered decoupled when they interact solely through declared dependencies and task dependencies. Any direct modification or reading of another project's object creates coupling between the projects. Coupling during configuration can result in flawed build outcomes when using 'configuration on demand', while coupling during execution can affect parallel execution.

One common source of coupling is configuration injection, such as using `allprojects{}` or `subprojects{}` in build scripts.

To avoid coupling issues, it's recommended to:

- Refrain from referencing other subprojects' build scripts and prefer [cross-project configuration](#) from the root project.
- Avoid dynamically changing other projects' configurations during execution.

As Gradle evolves, it aims to provide features that leverage decoupled projects while offering solutions for common use cases like configuration injection without introducing coupling.

Parallel projects

Gradle's parallel execution feature optimizes CPU utilization to accelerate builds by concurrently executing tasks from different projects.

To enable parallel execution, use the `--parallel` command-line argument or configure your build environment. Gradle automatically determines the optimal number of parallel threads based on CPU cores.

During parallel execution, each worker handles a specific project exclusively. Task dependencies are respected, with workers prioritizing upstream tasks. However, tasks may not execute in alphabetical order, as in sequential mode. It's crucial to correctly declare task dependencies and inputs/outputs to avoid ordering issues.

DEVELOPING TASKS

Understanding Tasks

A task represents some **independent unit of work** that a build performs, such as compiling classes, creating a JAR, generating Javadoc, or publishing archives to a repository.

SubProject

build.gradle

```
tasks.register('docFilesJar', Jar) {  
    group = 'documentation'  
    description = 'Generate documentation.'  
    archiveVersion = null  
    archiveFileName = 'doc-files.jar'  
    from 'src/main/template'  
}  
  
tasks.named('jar', Jar) {  
    from docFilesJar  
}  
  
abstract class DocFilesCreationTask : DefaultTask() {  
}
```

1. TASK REGISTRATION

2. TASK CONFIGURATION

3. TASK IMPLEMENTATION

Before reading this chapter, it's recommended that you first read the [Learning The Basics](#) and complete the [Tutorial](#).

Listing tasks

All available tasks in your project come from Gradle plugins and build scripts.

You can list all the available tasks in a project by running the following command in the terminal:

```
$ ./gradlew tasks
```

Let's take a very basic Gradle project as an example. The project has the following structure:

```
gradle-project  
├── app  
│   ├── build.gradle.kts    // empty file - no build logic  
│   ├── ...                // some java code  
├── settings.gradle.kts    // includes app subproject  
├── gradle  
└── ...
```

```
├── gradlew
└── gradlew.bat
```

```
gradle-project
├── app
│   ├── build.gradle    // empty file - no build logic
│   └── ...             // some java code
├── settings.gradle     // includes app subproject
├── gradle
│   └── ...
├── gradlew
└── gradlew.bat
```

The settings file contains the following:

settings.gradle.kts

```
rootProject.name = "gradle-project"
include("app")
```

settings.gradle

```
rootProject.name = 'gradle-project'
include('app')
```

Currently, the **app** subproject's build file is empty.

To see the tasks available in the **app** subproject, run **./gradlew :app:tasks:**

```
$ ./gradlew :app:tasks
```

```
> Task :app:tasks
```

```
-----
Tasks runnable from project ':app'
-----
```

```
Help tasks
-----
```

buildEnvironment - Displays all buildscript dependencies declared in project ':app'.
dependencies - Displays all dependencies declared in project ':app'.
dependencyInsight - Displays the insight into a specific dependency in project ':app'.
help - Displays a help message.
javaToolchains - Displays the detected java toolchains.
kotlinDslAccessorsReport - Prints the Kotlin code for accessing the currently available project extensions and conventions.
outgoingVariants - Displays the outgoing variants of project ':app'.
projects - Displays the sub-projects of project ':app'.
properties - Displays the properties of project ':app'.
resolvableConfigurations - Displays the configurations that can be resolved in project ':app'.
tasks - Displays the tasks runnable from project ':app'.

We observe that only a small number of help tasks are available at the moment. This is because the core of Gradle only provides tasks that analyze your build. Other tasks, such as the those that build your project or compile your code, are added by plugins.

Let's explore this by adding the [Gradle core base plugin](#) to the `app` build script:

app/build.gradle.kts

```
plugins {  
    id("base")  
}
```

app/build.gradle

```
plugins {  
    id('base')  
}
```

The `base` plugin adds central lifecycle tasks. Now when we run `./gradlew app:tasks`, we can see the `assemble` and `build` tasks are available:

```
$ ./gradlew :app:tasks
```

```
> Task :app:tasks
```

```
-----  
Tasks runnable from project ':app'
```

```
-----  
Build tasks
```

```

-----
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
clean - Deletes the build directory.

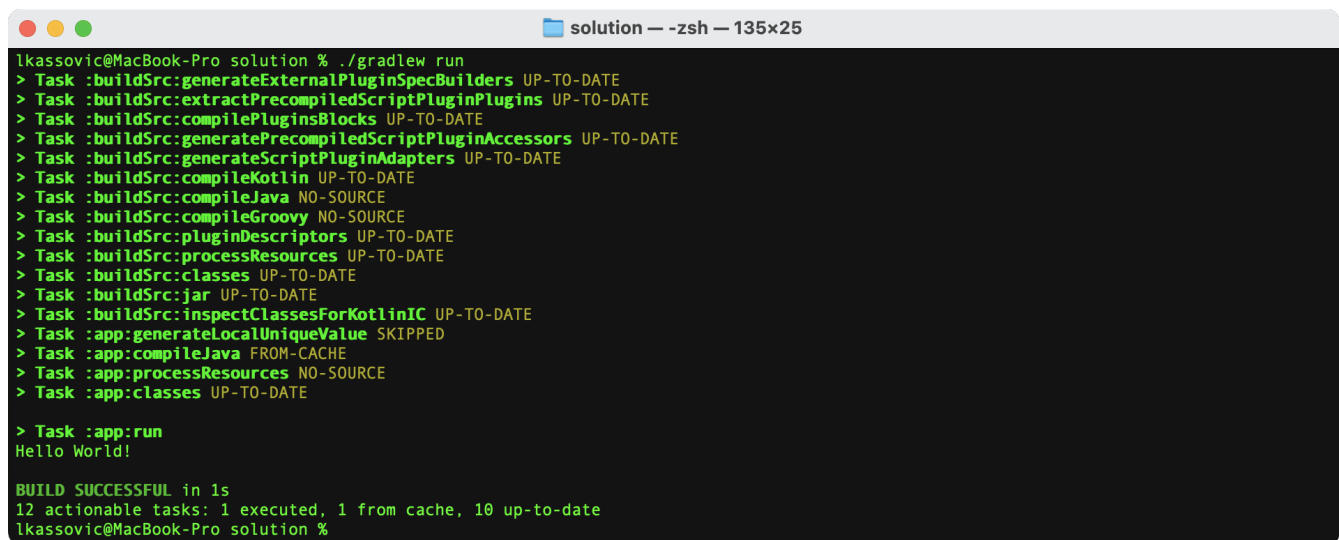
Help tasks
-----
buildEnvironment - Displays all buildscript dependencies declared in project ':app'.
dependencies - Displays all dependencies declared in project ':app'.
dependencyInsight - Displays the insight into a specific dependency in project ':app'.
help - Displays a help message.
javaToolchains - Displays the detected java toolchains.
outgoingVariants - Displays the outgoing variants of project ':app'.
projects - Displays the sub-projects of project ':app'.
properties - Displays the properties of project ':app'.
resolvableConfigurations - Displays the configurations that can be resolved in project ':app'.
tasks - Displays the tasks runnable from project ':app'.

Verification tasks
-----
check - Runs all checks.

```

Task outcomes

When Gradle executes a task, it labels the task with outcomes via the console.



```

solution — -zsh — 135x25
lkassovic@MacBook-Pro solution % ./gradlew run
> Task :buildSrc:generateExternalPluginSpecBuilders UP-TO-DATE
> Task :buildSrc:extractPrecompiledScriptPluginPlugins UP-TO-DATE
> Task :buildSrc:compilePluginsBlocks UP-TO-DATE
> Task :buildSrc:generatePrecompiledScriptPluginAccessors UP-TO-DATE
> Task :buildSrc:generateScriptPluginAdapters UP-TO-DATE
> Task :buildSrc:compileKotlin UP-TO-DATE
> Task :buildSrc:compileJava NO-SOURCE
> Task :buildSrc:compileGroovy NO-SOURCE
> Task :buildSrc:pluginDescriptors UP-TO-DATE
> Task :buildSrc:processResources UP-TO-DATE
> Task :buildSrc:classes UP-TO-DATE
> Task :buildSrc:jar UP-TO-DATE
> Task :buildSrc:inspectClassesForKotlinIC UP-TO-DATE
> Task :app:generateLocalUniqueValue SKIPPED
> Task :app:compileJava FROM-CACHE
> Task :app:processResources NO-SOURCE
> Task :app:classes UP-TO-DATE

> Task :app:run
Hello World!

BUILD SUCCESSFUL in 1s
12 actionable tasks: 1 executed, 1 from cache, 10 up-to-date
lkassovic@MacBook-Pro solution %

```

These labels are based on whether a task has actions to execute and if Gradle executed them. Actions include, but are not limited to, compiling code, zipping files, and publishing archives.

(no label) or EXECUTED

Task executed its actions.

- Task has actions and Gradle executed them.
- Task has no actions and some dependencies, and Gradle executed one or more of the

dependencies. See also [Lifecycle Tasks](#).

UP-TO-DATE

Task's outputs did not change.

- Task has outputs and inputs but they have not changed. See [Incremental Build](#).
- Task has actions, but the task tells Gradle it did not change its outputs.
- Task has no actions and some dependencies, but all the dependencies are **UP-TO-DATE**, **SKIPPED** or **FROM-CACHE**. See [Lifecycle Tasks](#).
- Task has no actions and no dependencies.

FROM-CACHE

Task's outputs could be found from a previous execution.

- Task has outputs restored from the build cache. See [Build Cache](#).

SKIPPED

Task did not execute its actions.

- Task has been explicitly excluded from the command-line. See [Excluding tasks from execution](#).
- Task has an **onlyIf** predicate return false. See [Using a predicate](#).

NO-SOURCE

Task did not need to execute its actions.

- Task has inputs and outputs, but **no sources** (i.e., inputs were not found).

Task group and description

Task groups and descriptions are used to organize and describe tasks.

Groups

Task groups are used to categorize tasks. When you run `./gradlew tasks`, tasks are listed under their respective groups, making it easier to understand their purpose and relationship to other tasks. Groups are set using the **group** property.

Descriptions

Descriptions provide a brief explanation of what a task does. When you run `./gradlew tasks`, the descriptions are shown next to each task, helping you understand its purpose and how to use it. Descriptions are set using the **description** property.

Let's consider a basic Java application as an example. The build contains a subproject called **app**.

Let's list the available tasks in **app** at the moment:

```
$ ./gradlew :app:tasks  
  
> Task :app:tasks
```

Tasks runnable from project ':app'

Application tasks

run - Runs this project as a JVM application.

Build tasks

assemble - Assembles the outputs of this project.

Here, the `:run` task is part of the `Application` group with the description `Runs this project as a JVM application`. In code, it would look something like this:

app/build.gradle.kts

```
tasks.register("run") {  
    group = "Application"  
    description = "Runs this project as a JVM application."  
}
```

app/build.gradle

```
tasks.register("run") {  
    group = "Application"  
    description = "Runs this project as a JVM application."  
}
```

Private and hidden tasks

Gradle doesn't support marking a task as *private*.

However, tasks will only show up when running `:tasks` if `task.group` is set or no other task depends on it.

For instance, the following task will not appear when running `./gradlew :app:tasks` because it does not have a group; it is called a *hidden* task:

app/build.gradle.kts

```
tasks.register("helloTask") {
```

```
    println("Hello")
}
```

app/build.gradle

```
tasks.register("helloTask") {
    println 'Hello'
}
```

Although **helloTask** is not listed, it can still be executed by Gradle:

```
$ ./gradlew :app:tasks

> Task :app:tasks

-----
Tasks runnable from project ':app'
-----

Application tasks
-----
run - Runs this project as a JVM application

Build tasks
-----
assemble - Assembles the outputs of this project.
```

Let's add a group to the same task:

app/build.gradle.kts

```
tasks.register("helloTask") {
    group = "Other"
    description = "Hello task"
    println("Hello")
}
```

app/build.gradle

```
tasks.register("helloTask") {
    group = "Other"
    description = "Hello task"
```

```
println 'Hello'
}
```

Now that the group is added, the task is visible:

```
$ ./gradlew :app:tasks

> Task :app:tasks

-----
Tasks runnable from project ':app'
-----

Application tasks
-----
run - Runs this project as a JVM application

Build tasks
-----
assemble - Assembles the outputs of this project.

Other tasks
-----
helloTask - Hello task
```

In contrast, `./gradlew tasks --all` will show all tasks; *hidden* and *visible* tasks are listed.

Grouping tasks

If you want to customize which tasks are shown to users when listed, you can group tasks and set the visibility of each group.

NOTE

Remember, even if you *hide* tasks, they are still available, and Gradle can still run them.

Let's start with an example built by Gradle `init` for a Java application with multiple subprojects. The project structure is as follows:

```
gradle-project
├── app
│   ├── build.gradle.kts
│   └── src                // some java code
│       └── ...
├── utilities
└── build.gradle.kts
```

```

├── src                // some java code
│   └── ...
├── list
│   ├── build.gradle.kts
│   └── src            // some java code
│       └── ...
├── buildSrc
│   ├── build.gradle.kts
│   ├── settings.gradle.kts
│   └── src            // common build logic
│       └── ...
├── settings.gradle.kts
├── gradle
├── gradlew
└── gradlew.bat

```

```

gradle-project
├── app
│   ├── build.gradle
│   └── src            // some java code
│       └── ...
├── utilities
│   ├── build.gradle
│   └── src            // some java code
│       └── ...
├── list
│   ├── build.gradle
│   └── src            // some java code
│       └── ...
├── buildSrc
│   ├── build.gradle
│   ├── settings.gradle
│   └── src            // common build logic
│       └── ...
├── settings.gradle
├── gradle
├── gradlew
└── gradlew.bat

```

Run **app:tasks** to see available tasks in the **app** subproject:

```

$ ./gradlew :app:tasks

> Task :app:tasks

```

----- Tasks runnable from project ':app' -----

Application tasks -----

run - Runs this project as a JVM application

Build tasks -----

assemble - Assembles the outputs of this project.

build - Assembles and tests this project.

buildDependents - Assembles and tests this project and all projects that depend on it.

buildNeeded - Assembles and tests this project and all projects it depends on.

classes - Assembles main classes.

clean - Deletes the build directory.

jar - Assembles a jar archive containing the classes of the 'main' feature.

testClasses - Assembles test classes.

Distribution tasks -----

assembleDist - Assembles the main distributions

distTar - Bundles the project as a distribution.

distZip - Bundles the project as a distribution.

installDist - Installs the project as a distribution as-is.

Documentation tasks -----

javadoc - Generates Javadoc API documentation for the 'main' feature.

Help tasks -----

buildEnvironment - Displays all buildscript dependencies declared in project ':app'.

dependencies - Displays all dependencies declared in project ':app'.

dependencyInsight - Displays the insight into a specific dependency in project ':app'.

help - Displays a help message.

javaToolchains - Displays the detected java toolchains.

kotlinDslAccessorsReport - Prints the Kotlin code for accessing the currently available project extensions and conventions.

outgoingVariants - Displays the outgoing variants of project ':app'.

projects - Displays the sub-projects of project ':app'.

properties - Displays the properties of project ':app'.

resolvableConfigurations - Displays the configurations that can be resolved in project ':app'.

tasks - Displays the tasks runnable from project ':app'.

Verification tasks -----

check - Runs all checks.

test - Runs the test suite.

If we look at the list of tasks available, even for a standard Java project, it's extensive. Many of these tasks are rarely required directly by developers using the build.

We can configure the `:tasks` task and limit the tasks shown to a certain group.

Let's create our own group so that all tasks are hidden by default by updating the `app` build script:

app/build.gradle.kts

```
val myBuildGroup = "my app build"           // Create a group name

tasks.register<TaskReportTask>("tasksAll") { // Register the tasksAll task
    group = myBuildGroup
    description = "Show additional tasks."
    setShowDetail(true)
}

tasks.named<TaskReportTask>("tasks") {       // Move all existing tasks to
    the group
    displayGroup = myBuildGroup
}
```

app/build.gradle

```
def myBuildGroup = "my app build"           // Create a group name

tasks.register("tasksAll", TaskReportTask) { // Register the tasksAll task
    group = myBuildGroup
    description = "Show additional tasks."
    setShowDetail(true)
}

tasks.named("tasks", TaskReportTask) {      // Move all existing tasks to
    the group
    displayGroup = myBuildGroup
}
```

Now, when we list tasks available in `app`, the list is shorter:

```
$ ./gradlew :app:tasks
```

```
> Task :app:tasks
```

```
-----
Tasks runnable from project ':app'
```

```
-----  
My app build tasks  
-----
```

```
tasksAll - Show additional tasks.
```

Task categories

Gradle distinguishes between two categories of tasks:

1. **Lifecycle tasks**
2. **Actionable tasks**

Lifecycle tasks define targets you can call, such as `:build` your project. Lifecycle tasks do not provide Gradle with actions. They must be *wired* to actionable tasks. The `base` Gradle plugin only adds lifecycle tasks.

Actionable tasks define actions for Gradle to take, such as `:compileJava`, which compiles the Java code of your project. Actions include creating JARs, zipping files, publishing archives, and much more. Plugins like the `java-library` plugin adds actionable tasks.

Let's update the build script of the previous example, which is currently an empty file so that our `app` subproject is a Java library:

app/build.gradle.kts

```
plugins {  
    id("java-library")  
}
```

app/build.gradle

```
plugins {  
    id('java-library')  
}
```

Once again, we list the available tasks to see what new tasks are available:

```
$ ./gradlew :app:tasks
```

```
> Task :app:tasks  
  
-----
```

Tasks runnable from project ':app'

Build tasks

assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that depend on it.
buildNeeded - Assembles and tests this project and all projects it depends on.
classes - Assembles main classes.
clean - Deletes the build directory.
jar - Assembles a jar archive containing the classes of the 'main' feature.
testClasses - Assembles test classes.

Documentation tasks

javadoc - Generates Javadoc API documentation for the 'main' feature.

Help tasks

buildEnvironment - Displays all buildscript dependencies declared in project ':app'.
dependencies - Displays all dependencies declared in project ':app'.
dependencyInsight - Displays the insight into a specific dependency in project ':app'.
help - Displays a help message.
javaToolchains - Displays the detected java toolchains.
outgoingVariants - Displays the outgoing variants of project ':app'.
projects - Displays the sub-projects of project ':app'.
properties - Displays the properties of project ':app'.
resolvableConfigurations - Displays the configurations that can be resolved in project ':app'.
tasks - Displays the tasks runnable from project ':app'.

Verification tasks

check - Runs all checks.
test - Runs the test suite.

We see that many new tasks are available such as `jar` and `testClasses`.

Additionally, the `java-library` plugin has wired actionable tasks to lifecycle tasks. If we call the `:build` task, we can see several tasks have been executed, including the `:app:compileJava` task.

```
$/gradlew :app:build
```

```
> Task :app:compileJava
> Task :app:processResources NO-SOURCE
> Task :app:classes
> Task :app:jar
> Task :app:assemble
> Task :app:compileTestJava
```

```
> Task :app:processTestResources NO-SOURCE
> Task :app:testClasses
> Task :app:test
> Task :app:check
> Task :app:build
```

The actionable `:compileJava` task is wired to the lifecycle `:build` task.

Incremental tasks

A key feature of Gradle tasks is their incremental nature.

Gradle can reuse results from prior builds. Therefore, if we've built our project before and made only minor changes, rerunning `:build` will not require Gradle to perform extensive work.

For example, if we modify only the test code in our project, leaving the production code unchanged, executing the build will solely recompile the test code. Gradle marks the tasks for the production code as `UP-TO-DATE`, indicating that it remains unchanged since the last successful build:

```
./gradlew :app:build

gradle@MacBook-Pro temp1 % ./gradlew :app:build
> Task :app:compileJava UP-TO-DATE
> Task :app:processResources NO-SOURCE
> Task :app:classes UP-TO-DATE
> Task :app:jar UP-TO-DATE
> Task :app:assemble UP-TO-DATE
> Task :app:compileTestJava
> Task :app:processTestResources NO-SOURCE
> Task :app:testClasses
> Task :app:test
> Task :app:check UP-TO-DATE
> Task :app:build UP-TO-DATE
```

Caching tasks

Gradle can reuse results from past builds using the build cache.

To enable this feature, activate the build cache by using the `--build-cache` [command line](#) parameter or by setting `org.gradle.caching=true` in your `gradle.properties` file.

This optimization has the potential to accelerate your builds significantly:

```
./gradlew :app:clean :app:build --build-cache

> Task :app:compileJava FROM-CACHE
> Task :app:processResources NO-SOURCE
> Task :app:classes UP-TO-DATE
```

```
> Task :app:jar
> Task :app:assemble
> Task :app:compileTestJava FROM-CACHE
> Task :app:processTestResources NO-SOURCE
> Task :app:testClasses UP-TO-DATE
> Task :app:test FROM-CACHE
> Task :app:check UP-TO-DATE
> Task :app:build
```

When Gradle can fetch outputs of a task from the cache, it labels the task with **FROM-CACHE**.

The build cache is handy if you switch between branches regularly. Gradle supports both local and remote build caches.

Developing tasks

When developing Gradle tasks, you have two choices:

1. Use an existing Gradle task type such as **Zip**, **Copy**, or **Delete**
2. Create your own Gradle task type such as **MyResolveTask** or **CustomTaskUsingToolchains**.

Task types are simply subclasses of the Gradle **Task** class.

With Gradle tasks, there are three states to consider:

1. **Registering** a task - using a task (implemented by you or provided by Gradle) in your build logic.
2. **Configuring** a task - defining inputs and outputs for a registered task.
3. **Implementing** a task - creating a custom task class (i.e., custom class type).

Registration is commonly done with the **register()** method.

Configuring a task is commonly done with the **named()** method.

Implementing a task is commonly done by extending Gradle's **DefaultTask** class:

```
tasks.register<Copy>("myCopy") ①

tasks.named<Copy>("myCopy") { ②
    from("resources")
    into("target")
    include("**/*.txt", "**/*.xml", "**/*.properties")
}

abstract class MyCopyTask : DefaultTask() { ③
    @TaskAction
    fun copyFiles() {
        val sourceDir = File("sourceDir")
        val destinationDir = File("destinationDir")
```

```

        sourceDir.listFiles()?.forEach { file ->
            if (file.isFile && file.extension == "txt") {
                file.copyTo(File(destinationDir, file.name))
            }
        }
    }
}

```

- ① Register the `myCopy` task of type `Copy` to let Gradle know we intend to use it in our build logic.
- ② Configure the registered `myCopy` task with the inputs and outputs it needs according to its [API](#).
- ③ Implement a custom task type called `MyCopyTask` which extends `DefaultTask` and defines the `copyFiles` task action.

```

tasks.register("myCopy", Copy) ①

tasks.named("myCopy", Copy) { ②
    from "resources"
    into "target"
    include "**/*.txt", "**/*.xml", "**/*.properties"
}

abstract class MyCopyTask extends DefaultTask { ③
    @TaskAction
    void copyFiles() {
        fileTree('sourceDir').matching {
            include '**/*.txt'
        }.forEach { file ->
            file.copyTo(file.path.replace('sourceDir', 'destinationDir'))
        }
    }
}

```

- ① Register the `myCopy` task of type `Copy` to let Gradle know we intend to use it in our build logic.
- ② Configure the registered `myCopy` task with the inputs and outputs it needs according to its [API](#).
- ③ Implement a custom task type called `MyCopyTask` which extends `DefaultTask` and defines the `copyFiles` task action.

1. Registering tasks

You define actions for Gradle to take by registering tasks in build scripts or plugins.

Tasks are defined using strings for task names:

build.gradle.kts

```
tasks.register("hello") {
    doLast {
        println("hello")
    }
}
```

build.gradle

```
tasks.register('hello') {
    doLast {
        println 'hello'
    }
}
```

In the example above, the task is added to the `TasksCollection` using the `register()` method in `TaskContainer`.

2. Configuring tasks

Gradle tasks must be configured to complete their action(s) successfully. If a task needs to ZIP a file, it must be configured with the file name and location. You can refer to the [API](#) for the Gradle `Zip` task to learn how to configure it appropriately.

Let's look at the `Copy` task provided by Gradle as an example. We first register a task called `myCopy` of type `Copy` in the build script:

build.gradle.kts

```
tasks.register<Copy>("myCopy")
```

build.gradle

```
tasks.register('myCopy', Copy)
```

This registers a copy task with no default behavior. Since the task is of type `Copy`, a Gradle supported

task type, it can be configured using its [API](#).

The following examples show several ways to achieve the same configuration:

1. Using the `named()` method:

Use `named()` to configure an existing task registered elsewhere:

build.gradle.kts

```
tasks.named<Copy>("myCopy") {  
    from("resources")  
    into("target")  
    include("**/*.txt", "**/*.xml", "**/*.properties")  
}
```

build.gradle

```
tasks.named('myCopy') {  
    from 'resources'  
    into 'target'  
    include '**/*.txt', '**/*.xml', '**/*.properties'  
}
```

2. Using a configuration block:

Use a block to configure the task immediately upon registering it:

build.gradle.kts

```
tasks.register<Copy>("copy") {  
    from("resources")  
    into("target")  
    include("**/*.txt", "**/*.xml", "**/*.properties")  
}
```

build.gradle

```
tasks.register('copy', Copy) {  
    from 'resources'  
    into 'target'  
    include '**/*.txt', '**/*.xml', '**/*.properties'
```

```
}
```

3. Name method as call:

A popular option that is only supported in Groovy is the shorthand notation:

```
copy {  
    from("resources")  
    into("target")  
    include("**/*.txt", "**/*.xml", "**/*.properties")  
}
```

NOTE This option breaks task configuration avoidance and is not recommended!

Regardless of the method chosen, the task is configured with the name of the files to be copied and the location of the files.

3. Implementing tasks

Gradle provides many task types including `Delete`, `Javadoc`, `Copy`, `Exec`, `Tar`, and `Pmd`. You can implement a custom task type if Gradle does not provide a task type that meets your build logic needs.

To create a custom task class, you extend `DefaultTask` and make the extending class abstract:

app/build.gradle.kts

```
abstract class MyCopyTask : DefaultTask() {  
  
}
```

app/build.gradle

```
abstract class MyCopyTask extends DefaultTask {  
  
}
```

Controlling Task Execution

Task dependencies allow tasks to be executed in a specific order based on their dependencies. This

ensures that tasks dependent on others are only executed after those dependencies have completed.

Project

build.gradle.kts

```
tasks.register('lifecycleTask', Task) {  
    group = 'CI'  
    description = 'Tasks for the CI only'  
    doLast {  
        println "CI build tasks are complete"  
    }  
    dependsOn tasks.build  
}
```

1. TASK ORDERING

Task dependencies can be categorized as either **implicit** or **explicit**:

Implicit dependencies

These dependencies are automatically inferred by Gradle based on the tasks' actions and configuration. For example, if `taskB` uses the output of `taskA` (e.g., a file generated by `taskA`), Gradle will automatically ensure that `taskA` is executed before `taskB` to fulfill this dependency.

Explicit dependencies

These dependencies are explicitly declared in the build script using the `dependsOn`, `mustRunAfter`, or `shouldRunAfter` methods. For example, if you want to ensure that `taskB` always runs after `taskA`, you can explicitly declare this dependency using `taskB.mustRunAfter(taskA)`.

Both implicit and explicit dependencies play a crucial role in defining the order of task execution and ensuring that tasks are executed in the correct sequence to produce the desired build output.

Task dependencies

Gradle inherently understands the dependencies among tasks. Consequently, it can determine the tasks that need execution when you target a specific task.

Let's take an example application with an `app` subproject and a `some-logic` subproject:

settings.gradle.kts

```
rootProject.name = "gradle-project"  
include("app")  
include("some-logic")
```

settings.gradle

```
rootProject.name = 'gradle-project'
```

```
include('app')
include('some-logic')
```

Let's imagine that the `app` subproject depends on the subproject called `some-logic`, which contains some Java code. We add this dependency in the `app` build script:

app/build.gradle.kts

```
plugins {
    id("application") // app is now a java application
}

application {
    mainClass.set("hello.HelloWorld") // main class name required by
    the application plugin
}

dependencies {
    implementation(project(":some-logic")) // dependency on some-logic
}
```

app/build.gradle

```
plugins {
    id('application') // app is now a java application
}

application {
    mainClass = 'hello.HelloWorld' // main class name required by
    the application plugin
}

dependencies {
    implementation(project(':some-logic')) // dependency on some-logic
}
```

If we run `:app:build` again, we see the Java code of `some-logic` is also compiled by Gradle automatically:

```
$/gradlew :app:build
```

```
> Task :app:processResources NO-SOURCE
```

```
> Task :app:processTestResources NO-SOURCE
> Task :some-logic:compileJava UP-TO-DATE
> Task :some-logic:processResources NO-SOURCE
> Task :some-logic:classes UP-TO-DATE
> Task :some-logic:jar UP-TO-DATE
> Task :app:compileJava
> Task :app:classes
> Task :app:jar UP-TO-DATE
> Task :app:startScripts
> Task :app:distTar
> Task :app:distZip
> Task :app:assemble
> Task :app:compileTestJava UP-TO-DATE
> Task :app:testClasses UP-TO-DATE
> Task :app:test
> Task :app:check
> Task :app:build
```

BUILD SUCCESSFUL in 430ms

9 actionable tasks: 5 executed, 4 up-to-date

Adding dependencies

There are several ways you can define the dependencies of a task.

Defining dependencies using task names and the `dependsOn()` method is simplest.

The following is an example which adds a dependency from `taskX` to `taskY`:

```
tasks.register("taskX") {
    dependsOn("taskY")
}
```

```
tasks.register("taskX") {
    dependsOn "taskY"
}
```

```
$ gradle -q taskX
taskY
taskX
```

For more information about task dependencies, see the [Task API](#).

Ordering tasks

In some cases, it is useful to control the *order* in which two tasks will execute, without introducing an explicit dependency between those tasks.

The primary difference between a task *ordering* and a task *dependency* is that an ordering rule does not influence which tasks will be executed, only the order in which they will be executed.

Task ordering can be useful in a number of scenarios:

- Enforce sequential ordering of tasks (e.g., **build** never runs before **clean**).
- Run build validations early in the build (e.g., validate I have the correct credentials before starting the work for a release build).
- Get feedback faster by running quick verification tasks before long verification tasks (e.g., unit tests should run before integration tests).
- A task that aggregates the results of all tasks of a particular type (e.g., test report task combines the outputs of all executed test tasks).

Two ordering rules are available: "*must run after*" and "*should run after*".

To specify a "must run after" or "should run after" ordering between 2 tasks, you use the [Task.mustRunAfter\(java.lang.Object...\)](#) and [Task.shouldRunAfter\(java.lang.Object...\)](#) methods. These methods accept a task instance, a task name, or any other input accepted by [Task.dependsOn\(java.lang.Object...\)](#).

When you use "must run after", you specify that **taskY** must always run after **taskX** when the build requires the execution of **taskX** and **taskY**. So if you only run **taskY** with **mustRunAfter**, you won't cause **taskX** to run. This is expressed as **taskY.mustRunAfter(taskX)**.

build.gradle.kts

```
val taskX by tasks.registering {
    doLast {
        println("taskX")
    }
}
val taskY by tasks.registering {
    doLast {
        println("taskY")
    }
}
taskY {
    mustRunAfter(taskX)
}
```

build.gradle

```
def taskX = tasks.register('taskX') {
    doLast {
        println 'taskX'
    }
}
def taskY = tasks.register('taskY') {
    doLast {
        println 'taskY'
    }
}
taskY.configure {
    mustRunAfter taskX
}
```

```
$ gradle -q taskY taskX
taskX
taskY
```

The "should run after" ordering rule is similar but less strict, as it will be ignored in two situations:

1. If using that rule introduces an ordering cycle.
2. When using parallel execution and all task dependencies have been satisfied apart from the "should run after" task, then this task will be run regardless of whether or not its "should run after" dependencies have been run.

You should use "should run after" where the ordering is helpful but not strictly required:

build.gradle.kts

```
val taskX by tasks.registering {
    doLast {
        println("taskX")
    }
}
val taskY by tasks.registering {
    doLast {
        println("taskY")
    }
}
taskY {
    shouldRunAfter(taskX)
}
```

build.gradle

```
def taskX = tasks.register('taskX') {
    doLast {
        println 'taskX'
    }
}
def taskY = tasks.register('taskY') {
    doLast {
        println 'taskY'
    }
}
taskY.configure {
    shouldRunAfter taskX
}
```

```
$ gradle -q taskY taskX
taskX
taskY
```

In the examples above, it is still possible to execute `taskY` without causing `taskX` to run:

```
$ gradle -q taskY
taskY
```

The "should run after" ordering rule will be ignored if it introduces an ordering cycle:

build.gradle.kts

```
val taskX by tasks.registering {
    doLast {
        println("taskX")
    }
}
val taskY by tasks.registering {
    doLast {
        println("taskY")
    }
}
val taskZ by tasks.registering {
    doLast {
        println("taskZ")
    }
}
```

```

    }
}
taskX { dependsOn(taskY) }
taskY { dependsOn(taskZ) }
taskZ { shouldRunAfter(taskX) }

```

build.gradle

```

def taskX = tasks.register('taskX') {
    doLast {
        println 'taskX'
    }
}
def taskY = tasks.register('taskY') {
    doLast {
        println 'taskY'
    }
}
def taskZ = tasks.register('taskZ') {
    doLast {
        println 'taskZ'
    }
}
taskX.configure { dependsOn(taskY) }
taskY.configure { dependsOn(taskZ) }
taskZ.configure { shouldRunAfter(taskX) }

```

```

$ gradle -q taskX
taskZ
taskY
taskX

```

Note that `taskY.mustRunAfter(taskX)` or `taskY.shouldRunAfter(taskX)` does not imply any execution dependency between the tasks:

- It is possible to execute `taskX` and `taskY` independently. The ordering rule only has an effect when both tasks are scheduled for execution.
- When run with `--continue`, it is possible for `taskY` to execute if `taskX` fails.

Finalizer tasks

Finalizer tasks are automatically added to the task graph when the finalized task is scheduled to run.

To specify a finalizer task, you use the `Task.finalizedBy(java.lang.Object...)` method. This method

accepts a task instance, a task name, or any other input accepted by `Task.dependsOn(java.lang.Object...)`:

build.gradle.kts

```
val taskX by tasks.registering {
    doLast {
        println("taskX")
    }
}
val taskY by tasks.registering {
    doLast {
        println("taskY")
    }
}

taskX { finalizedBy(taskY) }
```

build.gradle

```
def taskX = tasks.register('taskX') {
    doLast {
        println 'taskX'
    }
}
def taskY = tasks.register('taskY') {
    doLast {
        println 'taskY'
    }
}

taskX.configure { finalizedBy taskY }
```

```
$ gradle -q taskX
taskX
taskY
```

Finalizer tasks are executed even if the finalized task fails or if the finalized task is considered **UP-TO-DATE**:

build.gradle.kts

```
val taskX by tasks.registering {
    doLast {
        println("taskX")
        throw RuntimeException()
    }
}
val taskY by tasks.registering {
    doLast {
        println("taskY")
    }
}

taskX { finalizedBy(taskY) }
```

build.gradle

```
def taskX = tasks.register('taskX') {
    doLast {
        println 'taskX'
        throw new RuntimeException()
    }
}
def taskY = tasks.register('taskY') {
    doLast {
        println 'taskY'
    }
}

taskX.configure { finalizedBy taskY }
```

```
$ gradle -q taskX
taskX
taskY
```

FAILURE: Build failed with an exception.

* Where:

Build file '/home/user/gradle/samples/build.gradle' line: 4

* What went wrong:

Execution failed for task ':taskX'.

> java.lang.RuntimeException (no error message)

* Try:

- > Run with `--stacktrace` option to get the stack trace.
- > Run with `--info` or `--debug` option to get more log output.
- > Run with `--scan` to get full insights.
- > Get more help at <https://help.gradle.org>.

BUILD FAILED in 0s

Finalizer tasks are useful when the build creates a resource that must be cleaned up, regardless of whether the build fails or succeeds. An example of such a resource is a web container that is started before an integration test task and must be shut down, even if some tests fail.

Skipping tasks

Gradle offers multiple ways to skip the execution of a task.

1. Using a predicate

You can use `Task.onlyIf` to attach a predicate to a task. The task's actions will only be executed if the predicate is evaluated to be `true`.

The predicate is passed to the task as a parameter and returns `true` if the task will execute and `false` if the task will be skipped. The predicate is evaluated just before the task is executed.

Passing an optional reason string to `onlyIf()` is useful for explaining why the task is skipped:

build.gradle.kts

```
val hello by tasks.registering {
    doLast {
        println("hello world")
    }
}

hello {
    val skipProvider = providers.gradleProperty("skipHello")
    onlyIf("there is no property skipHello") {
        !skipProvider.isPresent()
    }
}
```

build.gradle

```
def hello = tasks.register('hello') {
    doLast {
        println 'hello world'
    }
}
```

```

    }
}

hello.configure {
    def skipProvider = providers.gradleProperty("skipHello")
    onlyIf("there is no property skipHello") {
        !skipProvider.present
    }
}

```

```

$ gradle hello -PskipHello
> Task :hello SKIPPED

BUILD SUCCESSFUL in 0s

```

To find why a task was skipped, run the build with the `--info` logging level.

```

$ gradle hello -PskipHello --info
...

> Task :hello SKIPPED
Skipping task ':hello' as task onlyIf 'there is no property skipHello' is false.
:hello (Thread[included builds,5,main]) completed. Took 0.018 secs.

BUILD SUCCESSFUL in 13s

```

2. Using `StopExecutionException`

If the logic for skipping a task can't be expressed with a predicate, you can use the `StopExecutionException`.

If this exception is thrown by an action, the task action as well as the execution of any following action is skipped. The build continues by executing the next task:

build.gradle.kts

```

val compile by tasks.registering {
    doLast {
        println("We are doing the compile.")
    }
}

compile {

```

```

doFirst {
    // Here you would put arbitrary conditions in real life.
    if (true) {
        throw StopExecutionException()
    }
}
tasks.register("myTask") {
    dependsOn(compile)
    doLast {
        println("I am not affected")
    }
}

```

build.gradle

```

def compile = tasks.register('compile') {
    doLast {
        println 'We are doing the compile.'
    }
}

compile.configure {
    doFirst {
        // Here you would put arbitrary conditions in real life.
        if (true) {
            throw new StopExecutionException()
        }
    }
}

tasks.register('myTask') {
    dependsOn('compile')
    doLast {
        println 'I am not affected'
    }
}

```

```

$ gradle -q myTask
I am not affected

```

This feature is helpful if you work with tasks provided by Gradle. It allows you to add *conditional* execution of the built-in actions of such a task.^[1]

3. Enabling and Disabling tasks

Every task has an **enabled** flag, which defaults to **true**. Setting it to **false** prevents executing the task's actions.

A disabled task will be labeled **SKIPPED**:

build.gradle.kts

```
val disableMe by tasks.registering {
    doLast {
        println("This should not be printed if the task is disabled.")
    }
}

disableMe {
    enabled = false
}
```

build.gradle

```
def disableMe = tasks.register('disableMe') {
    doLast {
        println 'This should not be printed if the task is disabled.'
    }
}

disableMe.configure {
    enabled = false
}
```

```
$ gradle disableMe
> Task :disableMe SKIPPED

BUILD SUCCESSFUL in 0s
```

4. Task timeouts

Every task has a **timeout** property, which can be used to limit its execution time. When a task reaches its timeout, its task execution thread is interrupted. The task will be marked as **FAILED**.

Finalizer tasks are executed. If **--continue** is used, other tasks continue running.

Tasks that don't respond to interrupts can't be timed out. All of Gradle's built-in tasks respond to

timeouts.

build.gradle.kts

```
tasks.register("hangingTask") {  
    doLast {  
        Thread.sleep(100000)  
    }  
    timeout = Duration.ofMillis(500)  
}
```

build.gradle

```
tasks.register("hangingTask") {  
    doLast {  
        Thread.sleep(100000)  
    }  
    timeout = Duration.ofMillis(500)  
}
```

Task rules

Sometimes you want to have a task whose behavior depends on a large or infinite number value range of parameters. A very nice and expressive way to provide such tasks are task rules:

build.gradle.kts

```
tasks.addRule("Pattern: ping<ID>") {  
    val taskName = this  
    if (startsWith("ping")) {  
        task(taskName) {  
            doLast {  
                println("Pinging: " + (taskName.replace("ping", "")))  
            }  
        }  
    }  
}
```

build.gradle

```
tasks.addRule("Pattern: ping<ID>") { String taskName ->
```

```

    if (taskName.startsWith("ping")) {
        task(taskName) {
            doLast {
                println "Pinging: " + (taskName - 'ping')
            }
        }
    }
}

```

```

$ gradle -q pingServer1
Pinging: Server1

```

The `String` parameter is used as a description for the rule, which is shown with `./gradlew tasks`.

Rules are not only used when calling tasks from the command line. You can also create `dependsOn` relations on rule based tasks:

build.gradle.kts

```

tasks.addRule("Pattern: ping<ID>") {
    val taskName = this
    if (startsWith("ping")) {
        task(taskName) {
            doLast {
                println("Pinging: " + (taskName.replace("ping", "")))
            }
        }
    }
}

tasks.register("groupPing") {
    dependsOn("pingServer1", "pingServer2")
}

```

build.gradle

```

tasks.addRule("Pattern: ping<ID>") { String taskName ->

    if (taskName.startsWith("ping")) {
        task(taskName) {
            doLast {
                println "Pinging: " + (taskName - 'ping')
            }
        }
    }
}

```

```
    }  
  }  
}  
  
tasks.register('groupPing') {  
    dependsOn 'pingServer1', 'pingServer2'  
}
```

```
$ gradle -q groupPing  
Pinging: Server1  
Pinging: Server2
```

If you run `./gradlew -q tasks`, you won't find a task named `pingServer1` or `pingServer2`, but this script is executing logic based on the request to run those tasks.

Exclude tasks from execution

You can exclude a task from execution using the `-x` or `--exclude-task` command-line option and provide the task's name to exclude.

```
$ ./gradlew build -x test
```

For instance, you can run the `check` task but exclude the `test` task from running. This approach can lead to unexpected outcomes, particularly if you exclude an actionable task that produces results needed by other tasks. Instead of relying on the `-x` parameter, defining a suitable lifecycle task for the desired action is recommended.

Using `-x` is a practice that should be avoided, although still commonly observed.

Organizing Tasks

There are two types of tasks, actionable and lifecycle tasks.

Actionable tasks in Gradle are tasks that perform actual work, such as compiling code. Lifecycle tasks are tasks that do not do work themselves. These tasks have no actions, instead, they bundle actionable tasks and serve as *targets* for the build.

Project

build.gradle.kts

```
tasks.register("buildApp") {  
    group = "build"  
    description = "Builds the Java application"  
    doLast {  
        println("Build completed.")  
    }  
}  
  
tasks.named("build") {  
    dependsOn("buildApp")  
}
```

← 1. ACTIONABLE TASK

← 2. LIFECYCLE TASK

A well-organized setup of lifecycle tasks enhances the accessibility of your build for new users and simplifies integration with CI.

Lifecycle tasks

Lifecycle tasks can be particularly beneficial for separating work between users or machines (CI vs local). For example, a developer on a local machine might not want to run an entire build on every single change.

Let's take a standard **app** as an example which applies the **base** plugin.

NOTE

The **Gradle base plugin** defines several lifecycle tasks, including **build**, **assemble**, and **check**.

We group the **build**, **check** task, and the **run** task by adding the following lines to the **app** build script:

app/build.gradle.kts

```
tasks.build {  
    group = myBuildGroup  
}  
  
tasks.check {  
    group = myBuildGroup  
    description = "Runs checks (including tests)."  
}  
  
tasks.named("run") {  
    group = myBuildGroup  
}
```

app/build.gradle

```
tasks.build {
    group = myBuildGroup
}

tasks.check {
    group = myBuildGroup
    description = "Runs checks (including tests)."
}

tasks.named('run') {
    group = myBuildGroup
}
```

If we now look at the `app:tasks` list, we can see the three tasks are available:

```
$ ./gradlew :app:tasks

> Task :app:tasks

-----
Tasks runnable from project ':app'
-----

My app build tasks
-----
build - Assembles and tests this project.
check - Runs checks (including tests).
run - Runs this project as a JVM application
tasksAll - Show additional tasks.
```

This is already useful if the standard lifecycle tasks are sufficient. Moving the groups around helps clarify the tasks you expect to used in your build.

In many cases, there are more specific requirements that you want to address. One common scenario is running quality checks without running tests. Currently, the `:check` task runs tests and the code quality checks. Instead, we want to run code quality checks all the time, but not the lengthy test.

To add a quality check lifecycle task, we introduce an additional lifecycle task called `qualityCheck` and a plugin called `spotbugs`.

To add a lifecycle task, use `tasks.register()`. The only thing you need to provide is a name. Put this task in our group and wire the actionable tasks that belong to this new lifecycle task using the `dependsOn()` method:

app/build.gradle.kts

```
plugins {  
    id("com.github.spotbugs") version "6.0.7"           // spotbugs plugin  
}  
  
tasks.register("qualityCheck") {                       // qualityCheck task  
    group = myBuildGroup                               // group  
    description = "Runs checks (excluding tests)."  
    dependsOn(tasks.classes, tasks.spotbugsMain)      // dependencies  
    dependsOn(tasks.testClasses, tasks.spotbugsTest) // dependencies  
}
```

app/build.gradle

```
plugins {  
    id 'com.github.spotbugs' version '6.0.7'           // spotbugs plugin  
}  
  
tasks.register('qualityCheck') {                      // qualityCheck task  
    group = myBuildGroup                               // group  
    description = 'Runs checks (excluding tests).'  
    dependsOn tasks.classes, tasks.spotbugsMain       // dependencies  
    dependsOn tasks.testClasses, tasks.spotbugsTest  // dependencies  
}
```

Note that you don't need to list all the tasks that Gradle will execute. Just specify the targets you want to collect here. Gradle will determine which other tasks it needs to call to reach these goals.

In the example, we add the `classes` task, a lifecycle task to compile all our production code, and the `spotbugsMain` task, which checks our production code.

We also add a description that will show up in the task list that helps distinguish the two check tasks better.

Now, if run `./gradlew :app:tasks`, we can see that our new `qualityCheck` lifecycle task is available:

```
$ ./gradlew :app:tasks
```

```
> Task :app:tasks
```

```
-----  
Tasks runnable from project ':app'  
-----
```

My app build tasks

build - Assembles and tests this project.
check - Runs checks (including tests).
qualityCheck - Runs checks (excluding tests).
run - Runs this project as a JVM application
tasksAll - Show additional tasks.

If we run it, we can see that it runs checkstyle but not the tests:

```
$ ./gradlew :app:qualityCheck

> Task :buildSrc:checkKotlinGradlePluginConfigurationErrors
> Task :buildSrc:generateExternalPluginSpecBuilders UP-TO-DATE
> Task :buildSrc:extractPrecompiledScriptPluginPlugins UP-TO-DATE
> Task :buildSrc:compilePluginsBlocks UP-TO-DATE
> Task :buildSrc:generatePrecompiledScriptPluginAccessors UP-TO-DATE
> Task :buildSrc:generateScriptPluginAdapters UP-TO-DATE
> Task :buildSrc:compileKotlin UP-TO-DATE
> Task :buildSrc:compileJava NO-SOURCE
> Task :buildSrc:compileGroovy NO-SOURCE
> Task :buildSrc:pluginDescriptors UP-TO-DATE
> Task :buildSrc:processResources UP-TO-DATE
> Task :buildSrc:classes UP-TO-DATE
> Task :buildSrc:jar UP-TO-DATE
> Task :app:processResources NO-SOURCE
> Task :app:processTestResources NO-SOURCE
> Task :list:compileJava UP-TO-DATE
> Task :utilities:compileJava UP-TO-DATE
> Task :app:compileJava
> Task :app:classes
> Task :app:compileTestJava
> Task :app:testClasses
> Task :app:spotbugsTest
> Task :app:spotbugsMain
> Task :app:qualityCheck

BUILD SUCCESSFUL in 1s
16 actionable tasks: 5 executed, 11 up-to-date
```

So far, we have looked at tasks in individual subprojects, which is useful for local development when you work on code in one subproject.

With this setup, developers only need to know that they can call Gradle with `:subproject-name:tasks` to see which tasks are available and useful for them.

Global lifecycle tasks

Another place to invoke lifecycle tasks is within the root build; this is especially useful for Continuous Integration (CI).

Gradle tasks play a crucial role in CI or CD systems, where activities like compiling all code, running tests, or building and packaging the complete application are typical. To facilitate this, you can include lifecycle tasks that span multiple subprojects.

NOTE

Gradle has been around for a long time, and you will frequently observe build files in the root directory serving various purposes. In older Gradle versions, many tasks were defined within the root Gradle build file, resulting in various issues. Therefore, exercise caution when determining the content of this file.

One of the few elements that should be placed in the root build file is global lifecycle tasks.

Let's continue using the Gradle `init` Java application multi-project as an example.

This time, we're incorporating a build script in the root project. We'll establish two groups for our global lifecycle tasks: one for tasks relevant to local development, such as running all checks, and another exclusively for our CI system.

Once again, we narrowed down the tasks listed to our specific groups:

build.gradle.kts

```
val globalBuildGroup = "My global build"
val ciBuildGroup = "My CI build"

tasks.named<TaskReportTask>("tasks") {
    displayGroups = listOf<String>(globalBuildGroup, ciBuildGroup)
}
```

build.gradle

```
def globalBuildGroup = "My global build"
def ciBuildGroup = "My CI build"

tasks.named(TaskReportTask, "tasks") {
    displayGroups = [globalBuildGroup, ciBuildGroup]
}
```

You could hide the CI tasks if you wanted to by updating `displayGroups`.

Currently, the root project exposes no tasks:

```
$ ./gradlew :tasks
```

```
> Task :tasks
```

```
-----  
Tasks runnable from root project 'gradle-project'  
-----
```

```
No tasks
```

NOTE In this file, we don't apply a plugin!

Let's add a **qualityCheckApp** task to execute all code quality checks in the **app** subproject. Similarly, for CI purposes, we implement a **checkAll** task that runs all tests:

build.gradle.kts

```
tasks.register("qualityCheckApp") {  
    group = globalBuildGroup  
    description = "Runs checks on app (globally)"  
    dependsOn(":app:qualityCheck" )  
}  
  
tasks.register("checkAll") {  
    group = ciBuildGroup  
    description = "Runs checks for all projects (CI)"  
    dependsOn(subprojects.map { ":${it.name}:check" })  
    dependsOn(gradle.includedBuilds.map { it.task(":checkAll") })  
}
```

build.gradle

```
tasks.register("qualityCheckApp") {  
    group = globalBuildGroup  
    description = "Runs checks on app (globally)"  
    dependsOn(":app:qualityCheck")  
}  
  
tasks.register("checkAll") {  
    group = ciBuildGroup  
    description = "Runs checks for all projects (CI)"  
    dependsOn subprojects.collect { ":${it.name}:check" }  
    dependsOn gradle.includedBuilds.collect { it.task(":checkAll") }  
}
```

So we can now ask Gradle to show us the tasks for the root project and, by default, it will only show us the `qualityCheckAll` task (and optionally the `checkAll` task depending on the value of `displayGroups`).

It should be clear what a user should run locally:

```
$ ./gradlew :tasks

> Task :tasks

-----
Tasks runnable from root project 'gradle-project'
-----

My CI build tasks
-----
checkAll - Runs checks for all projects (CI)

My global build tasks
-----
qualityCheckApp - Runs checks on app (globally)
```

If we run the `:checkAll` task, we see that it compiles all the code and runs the code quality checks (including `spotbug`):

```
$ ./gradlew :checkAll

> Task :buildSrc:checkKotlinGradlePluginConfigurationErrors
> Task :buildSrc:generateExternalPluginSpecBuilders UP-TO-DATE
> Task :buildSrc:extractPrecompiledScriptPluginPlugins UP-TO-DATE
> Task :buildSrc:compilePluginsBlocks UP-TO-DATE
> Task :buildSrc:generatePrecompiledScriptPluginAccessors UP-TO-DATE
> Task :buildSrc:generateScriptPluginAdapters UP-TO-DATE
> Task :buildSrc:compileKotlin UP-TO-DATE
> Task :buildSrc:compileJava NO-SOURCE
> Task :buildSrc:compileGroovy NO-SOURCE
> Task :buildSrc:pluginDescriptors UP-TO-DATE
> Task :buildSrc:processResources UP-TO-DATE
> Task :buildSrc:classes UP-TO-DATE
> Task :buildSrc:jar UP-TO-DATE
> Task :utilities:processResources NO-SOURCE
> Task :app:processResources NO-SOURCE
> Task :utilities:processTestResources NO-SOURCE
> Task :app:processTestResources NO-SOURCE
> Task :list:compileJava
> Task :list:processResources NO-SOURCE
> Task :list:classes
> Task :list:jar
> Task :utilities:compileJava
```

```
> Task :utilities:classes
> Task :utilities:jar
> Task :utilities:compileTestJava NO-SOURCE
> Task :utilities:testClasses UP-TO-DATE
> Task :utilities:test NO-SOURCE
> Task :utilities:check UP-TO-DATE
> Task :list:compileTestJava
> Task :list:processTestResources NO-SOURCE
> Task :list:testClasses
> Task :app:compileJava
> Task :app:classes
> Task :app:compileTestJava
> Task :app:testClasses
> Task :list:test
> Task :list:check
> Task :app:test
> Task :app:spotbugsTest
> Task :app:spotbugsMain
> Task :app:check
> Task :checkAll

BUILD SUCCESSFUL in 1s
21 actionable tasks: 12 executed, 9 up-to-date
```

Configuring Tasks Lazily

Knowing when and where a particular value is configured is difficult to track as a build grows in complexity. Gradle provides several ways to manage this using **lazy configuration**.

Eager Project

build.gradle

```
task('generateDocumentation') {
    // expensive task configuration code
}
```



Lazy Project

build.gradle

```
tasks.register('generateDocumentation') {
    // expensive task configuration code
}
```

Understanding Lazy properties

Gradle provides lazy properties, which delay calculating a property's value until it's actually required.

Lazy properties provide three main benefits:

1. **Deferred Value Resolution:** Allows wiring Gradle models without needing to know when a property's value will be known. For example, you may want to set the input source files of a task based on the source directories property of an extension, but the extension property value isn't known until the build script or some other plugin configures them.
2. **Automatic Task Dependency Management:** Connects output of one task to input of another,

automatically determining task dependencies. Property instances carry information about which task, if any, produces their value. Build authors do not need to worry about keeping task dependencies in sync with configuration changes.

3. **Improved Build Performance:** Avoids resource-intensive work during configuration, impacting build performance positively. For example, when a configuration value comes from parsing a file but is only used when functional tests are run, using a property instance to capture this means that the file is parsed only when the functional tests are run (and not when `clean` is run, for example).

Gradle represents lazy properties with two interfaces:

Provider

Represents a value that can only be queried and cannot be changed.

- Properties with these types are read-only.
- The method `Provider.get()` returns the current value of the property.
- A `Provider` can be created from another `Provider` using `Provider.map(Transformer)`.
- Many other types extend `Provider` and can be used wherever a `Provider` is required.

Property

Represents a value that can be queried and changed.

- Properties with these types are configurable.
- `Property` extends the `Provider` interface.
- The method `Property.set(T)` specifies a value for the property, overwriting whatever value may have been present.
- The method `Property.set(Provider)` specifies a `Provider` for the value for the property, overwriting whatever value may have been present. This allows you to wire together `Provider` and `Property` instances before the values are configured.
- A `Property` can be created by the factory method `ObjectFactory.property(Class)`.

Lazy properties are intended to be passed around and only queried when required. This typically happens during the `execution phase`.

The following demonstrates a task with a configurable `greeting` property and a read-only `message` property:

build.gradle.kts

```
abstract class Greeting : DefaultTask() { ①
    @get:Input
    abstract val greeting: Property<String> ②

    @Internal
    val message: Provider<String> = greeting.map { it + " from Gradle" } ③
}
```

```

@TaskAction
fun printMessage() {
    logger.quiet(message.get())
}

tasks.register<Greeting>("greeting") {
    greeting.set("Hi") ④
    greeting = "Hi" ⑤
}

```

build.gradle

```

abstract class Greeting extends DefaultTask { ①
    @Input
    abstract Property<String> getGreeting() ②

    @Internal
    final Provider<String> message = greeting.map { it + ' from Gradle' } ③

    @TaskAction
    void printMessage() {
        logger.quiet(message.get())
    }
}

tasks.register("greeting", Greeting) {
    greeting.set('Hi') ④
    greeting = 'Hi' ⑤
}

```

- ① A task that displays a greeting
- ② A configurable greeting
- ③ Read-only property calculated from the greeting
- ④ Configure the greeting
- ⑤ Alternative notation to calling Property.set()

```
$ gradle greeting
```

```
> Task :greeting
Hi from Gradle
```

```
BUILD SUCCESSFUL in 0s
```

1 actionable task: 1 executed

The `Greeting` task has a property of type `Property<String>` to represent the configurable greeting and a property of type `Provider<String>` to represent the calculated, read-only, message. The message `Provider` is created from the greeting `Property` using the `map()` method; its value is kept up-to-date as the value of the greeting property changes.

Creating a Property or Provider instance

Neither `Provider` nor its subtypes, such as `Property`, are intended to be implemented by a build script or plugin. Gradle provides factory methods to create instances of these types instead.

In the previous example, two factory methods were presented:

- `ObjectFactory.property(Class)` create a new `Property` instance. An instance of the `ObjectFactory` can be referenced from `Project.getObjects()` or by injecting `ObjectFactory` through a constructor or method.
- `Provider.map(Transformer)` creates a new `Provider` from an existing `Provider` or `Property` instance.

See the [Quick Reference](#) for all of the types and factories available.

A `Provider` can also be created by the factory method `ProviderFactory.provider(Callable)`.

NOTE

There are no specific methods to create a provider using a `groovy.lang.Closure`.

When writing a plugin or build script with Groovy, you can use the `map(Transformer)` method with a closure, and Groovy will convert the closure to a `Transformer`.

Similarly, when writing a plugin or build script with Kotlin, the Kotlin compiler will convert a Kotlin function into a `Transformer`.

Connecting properties together

An important feature of lazy properties is that they can be connected together so that changes to one property are automatically reflected in other properties.

Here is an example where the property of a task is connected to a property of a project extension:

build.gradle.kts

```
// A project extension
interface MessageExtension {
    // A configurable greeting
    abstract val greeting: Property<String>
}

// A task that displays a greeting
```

```

abstract class Greeting : DefaultTask() {
    // Configurable by the user
    @get:Input
    abstract val greeting: Property<String>

    // Read-only property calculated from the greeting
    @Internal
    val message: Provider<String> = greeting.map { it + " from Gradle" }

    @TaskAction
    fun printMessage() {
        logger.quiet(message.get())
    }
}

// Create the project extension
val messages = project.extensions.create<MessageExtension>("messages")

// Create the greeting task
tasks.register<Greeting>("greeting") {
    // Attach the greeting from the project extension
    // Note that the values of the project extension have not been configured yet
    greeting = messages.greeting
}

messages.apply {
    // Configure the greeting on the extension
    // Note that there is no need to reconfigure the task's `greeting` property. This is automatically updated as the extension property changes
    greeting = "Hi"
}

```

build.gradle

```

// A project extension
interface MessageExtension {
    // A configurable greeting
    Property<String> getGreeting()
}

// A task that displays a greeting
abstract class Greeting extends DefaultTask {
    // Configurable by the user
    @Input
    abstract Property<String> getGreeting()

    // Read-only property calculated from the greeting
    @Internal

```

```

final Provider<String> message = greeting.map { it + ' from Gradle' }

@TaskAction
void printMessage() {
    logger.quiet(message.get())
}

// Create the project extension
project.extensions.create('messages', MessageExtension)

// Create the greeting task
tasks.register("greeting", Greeting) {
    // Attach the greeting from the project extension
    // Note that the values of the project extension have not been configured
    yet
    greeting = messages.greeting
}

messages {
    // Configure the greeting on the extension
    // Note that there is no need to reconfigure the task's `greeting`
    property. This is automatically updated as the extension property changes
    greeting = 'Hi'
}

```

```
$ gradle greeting
```

```
> Task :greeting
Hi from Gradle
```

```
BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

This example calls the `Property.set(Provider)` method to attach a `Provider` to a `Property` to supply the value of the property. In this case, the `Provider` happens to be a `Property` as well, but you can connect any `Provider` implementation, for example one created using `Provider.map()`

Working with files

In [Working with Files](#), we introduced four collection types for `File`-like objects:

Read-only Type	Configurable Type
FileCollection	ConfigurableFileCollection
FileTree	ConfigurableFileTree

All of these types are also considered lazy types.

There are more strongly typed models used to represent elements of the file system: [Directory](#) and [RegularFile](#). These types shouldn't be confused with the standard Java [File](#) type as they are used to tell Gradle that you expect more specific values such as a directory or a non-directory, regular file.

Gradle provides two specialized [Property](#) subtypes for dealing with values of these types: [RegularFileProperty](#) and [DirectoryProperty](#). [ObjectFactory](#) has methods to create these: [ObjectFactory.fileProperty\(\)](#) and [ObjectFactory.directoryProperty\(\)](#).

A [DirectoryProperty](#) can also be used to create a lazily evaluated [Provider](#) for a [Directory](#) and [RegularFile](#) via [DirectoryProperty.dir\(String\)](#) and [DirectoryProperty.file\(String\)](#) respectively. These methods create providers whose values are calculated relative to the location for the [DirectoryProperty](#) they were created from. The values returned from these providers will reflect changes to the [DirectoryProperty](#).

build.gradle.kts

```
// A task that generates a source file and writes the result to an output
directory
abstract class GenerateSource : DefaultTask() {
    // The configuration file to use to generate the source file
    @get:InputFile
    abstract val configFile: RegularFileProperty

    // The directory to write source files to
    @get:OutputDirectory
    abstract val outputDir: DirectoryProperty

    @TaskAction
    fun compile() {
        val inFile = configFile.get().asFile
        logger.quiet("configuration file = $inFile")
        val dir = outputDir.get().asFile
        logger.quiet("output dir = $dir")
        val className = inFile.readText().trim()
        val srcFile = File(dir, "${className}.java")
        srcFile.writeText("public class ${className} { }")
    }
}

// Create the source generation task
tasks.register<GenerateSource>("generate") {
    // Configure the locations, relative to the project and build directories
    configFile = layout.projectDirectory.file("src/config.txt")
    outputDir = layout.buildDirectory.dir("generated-source")
}

// Change the build directory
```

```
// Don't need to reconfigure the task properties. These are automatically
updated as the build directory changes
layout.buildDirectory = layout.projectDirectory.dir("output")
```

build.gradle

```
// A task that generates a source file and writes the result to an output
directory
abstract class GenerateSource extends DefaultTask {
    // The configuration file to use to generate the source file
    @InputFile
    abstract RegularFileProperty getConfigFile()

    // The directory to write source files to
    @OutputDirectory
    abstract DirectoryProperty getOutputDir()

    @TaskAction
    def compile() {
        def inFile = configFile.get().asFile
        logger.quiet("configuration file = $inFile")
        def dir = outputDir.get().asFile
        logger.quiet("output dir = $dir")
        def className = inFile.text.trim()
        def srcFile = new File(dir, "${className}.java")
        srcFile.text = "public class ${className} { ... }"
    }
}

// Create the source generation task
tasks.register('generate', GenerateSource) {
    // Configure the locations, relative to the project and build directories
    configFile = layout.projectDirectory.file('src/config.txt')
    outputDir = layout.buildDirectory.dir('generated-source')
}

// Change the build directory
// Don't need to reconfigure the task properties. These are automatically
updated as the build directory changes
layout.buildDirectory = layout.projectDirectory.dir('output')
```

```
$ gradle generate
```

```
> Task :generate
configuration file = /home/user/gradle/samples/src/config.txt
output dir = /home/user/gradle/samples/output/generated-source
```

```
BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

```
$ gradle generate
```

```
> Task :generate
configuration file = /home/user/gradle/samples/kotlin/src/config.txt
output dir = /home/user/gradle/samples/kotlin/output/generated-source
```

```
BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

This example creates providers that represent locations in the project and build directories through `Project.getLayout()` with `ProjectLayout.getBuildDirectory()` and `ProjectLayout.getProjectDirectory()`.

To close the loop, note that a `DirectoryProperty`, or a simple `Directory`, can be turned into a `FileTree` that allows the files and directories contained in the directory to be queried with `DirectoryProperty.getAsFileTree()` or `Directory.getAsFileTree()`. From a `DirectoryProperty` or a `Directory`, you can create `FileCollection` instances containing a set of the files contained in the directory with `DirectoryProperty.files(Object...)` or `Directory.files(Object...)`.

Working with task inputs and outputs

Many builds have several tasks connected together, where one task consumes the outputs of another task as an input.

To make this work, we need to configure each task to know where to look for its inputs and where to place its outputs. Ensure that the producing and consuming tasks are configured with the same location and attach task dependencies between the tasks. This can be cumbersome and brittle if any of these values are configurable by a user or configured by multiple plugins, as task properties need to be configured in the correct order and locations, and task dependencies kept in sync as values change.

The `Property` API makes this easier by keeping track of the value of a property and the task that produces the value.

As an example, consider the following plugin with a producer and consumer task which are wired together:

build.gradle.kts

```
abstract class Producer : DefaultTask() {
    @get:OutputFile
    abstract val outputFile: RegularFileProperty

    @TaskAction
```

```

    fun produce() {
        val message = "Hello, World!"
        val output = outputFile.get().asFile
        output.writeText( message)
        logger.quiet("Wrote '${message}' to ${output}")
    }
}

abstract class Consumer : DefaultTask() {
    @get:InputFile
    abstract val inputFile: RegularFileProperty

    @TaskAction
    fun consume() {
        val input = inputFile.get().asFile
        val message = input.readText()
        logger.quiet("Read '${message}' from ${input}")
    }
}

val producer = tasks.register<Producer>("producer")
val consumer = tasks.register<Consumer>("consumer")

consumer {
    // Connect the producer task output to the consumer task input
    // Don't need to add a task dependency to the consumer task. This is
    // automatically added
    inputFile = producer.flatMap { it.outputFile }
}

producer {
    // Set values for the producer lazily
    // Don't need to update the consumer.inputFile property. This is
    // automatically updated as producer.outputFile changes
    outputFile = layout.buildDirectory.file("file.txt")
}

// Change the build directory.
// Don't need to update producer.outputFile and consumer.inputFile. These are
// automatically updated as the build directory changes
layout.buildDirectory = layout.projectDirectory.dir("output")

```

build.gradle

```

abstract class Producer extends DefaultTask {
    @OutputFile
    abstract RegularFileProperty getOutputFile()

    @TaskAction

```

```

void produce() {
    String message = 'Hello, World!'
    def output = outputFile.get().asFile
    output.text = message
    logger.quiet("Wrote '${message}' to ${output}")
}
}

abstract class Consumer extends DefaultTask {
    @InputFile
    abstract RegularFileProperty getInputFile()

    @TaskAction
    void consume() {
        def input = inputFile.get().asFile
        def message = input.text
        logger.quiet("Read '${message}' from ${input}")
    }
}

def producer = tasks.register("producer", Producer)
def consumer = tasks.register("consumer", Consumer)

consumer.configure {
    // Connect the producer task output to the consumer task input
    // Don't need to add a task dependency to the consumer task. This is
    // automatically added
    inputFile = producer.flatMap { it.outputFile }
}

producer.configure {
    // Set values for the producer lazily
    // Don't need to update the consumer.inputFile property. This is
    // automatically updated as producer.outputFile changes
    outputFile = layout.buildDirectory.file('file.txt')
}

// Change the build directory.
// Don't need to update producer.outputFile and consumer.inputFile. These are
// automatically updated as the build directory changes
layout.buildDirectory = layout.projectDirectory.dir('output')

```

```
$ gradle consumer
```

```
> Task :producer
```

```
Wrote 'Hello, World!' to /home/user/gradle/samples/output/file.txt
```

```
> Task :consumer
```

```
Read 'Hello, World!' from /home/user/gradle/samples/output/file.txt
```

```
BUILD SUCCESSFUL in 0s  
2 actionable tasks: 2 executed
```

```
$ gradle consumer
```

```
> Task :producer  
Wrote 'Hello, World!' to /home/user/gradle/samples/kotlin/output/file.txt
```

```
> Task :consumer  
Read 'Hello, World!' from /home/user/gradle/samples/kotlin/output/file.txt
```

```
BUILD SUCCESSFUL in 0s  
2 actionable tasks: 2 executed
```

In the example above, the task outputs and inputs are connected before any location is defined. The setters can be called at any time before the task is executed, and the change will automatically affect all related input and output properties.

Another important thing to note in this example is the absence of any explicit task dependency. Task outputs represented using **Providers** keep track of which task produces their value, and using them as task inputs will implicitly add the correct task dependencies.

Implicit task dependencies also work for input properties that are not files:

build.gradle.kts

```
abstract class Producer : DefaultTask() {  
    @get:OutputFile  
    abstract val outputFile: RegularFileProperty  
  
    @TaskAction  
    fun produce() {  
        val message = "Hello, World!"  
        val output = outputFile.get().asFile  
        output.writeText( message)  
        logger.quiet("Wrote '${message}' to ${output}")  
    }  
}  
  
abstract class Consumer : DefaultTask() {  
    @get:Input  
    abstract val message: Property<String>  
  
    @TaskAction  
    fun consume() {  
        logger.quiet(message.get())  
    }  
}
```

```

    }
}

val producer = tasks.register<Producer>("producer") {
    // Set values for the producer lazily
    // Don't need to update the consumer.inputFile property. This is
    // automatically updated as producer.outputFile changes
    outputFile = layout.buildDirectory.file("file.txt")
}

tasks.register<Consumer>("consumer") {
    // Connect the producer task output to the consumer task input
    // Don't need to add a task dependency to the consumer task. This is
    // automatically added
    message = producer.flatMap { it.outputFile }.map { it.asFile.readText() }
}

```

build.gradle

```

abstract class Producer extends DefaultTask {
    @OutputFile
    abstract RegularFileProperty getOutputFile()

    @TaskAction
    void produce() {
        String message = 'Hello, World!'
        def output = outputFile.get().asFile
        output.text = message
        logger.quiet("Wrote '${message}' to ${output}")
    }
}

abstract class Consumer extends DefaultTask {
    @Input
    abstract Property<String> getMessage()

    @TaskAction
    void consume() {
        logger.quiet(message.get())
    }
}

def producer = tasks.register('producer', Producer) {
    // Set values for the producer lazily
    // Don't need to update the consumer.inputFile property. This is
    // automatically updated as producer.outputFile changes
    outputFile = layout.buildDirectory.file('file.txt')
}

tasks.register('consumer', Consumer) {
    // Connect the producer task output to the consumer task input

```

```
// Don't need to add a task dependency to the consumer task. This is
automatically added
message = producer.flatMap { it.outputFile }.map { it.asFile.text }
}
```

```
$ gradle consumer

> Task :producer
Wrote 'Hello, World!' to /home/user/gradle/samples/build/file.txt

> Task :consumer
Hello, World!

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

```
$ gradle consumer

> Task :producer
Wrote 'Hello, World!' to /home/user/gradle/samples/kotlin/build/file.txt

> Task :consumer
Hello, World!

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

Working with collections

Gradle provides two lazy property types to help configure **Collection** properties.

These work exactly like any other **Provider** and, just like file providers, they have additional modeling around them:

- For **List** values the interface is called **ListProperty**.
You can create a new **ListProperty** using **ObjectFactory.listProperty(Class)** and specifying the element type.
- For **Set** values the interface is called **SetProperty**.
You can create a new **SetProperty** using **ObjectFactory.setProperty(Class)** and specifying the element type.

This type of property allows you to overwrite the entire collection value with **HasMultipleValues.set(Iterable)** and **HasMultipleValues.set(Provider)** or add new elements through the various **add** methods:

- [HasMultipleValues.add\(T\)](#): Add a single element to the collection
- [HasMultipleValues.add\(Provider\)](#): Add a lazily calculated element to the collection
- [HasMultipleValues.addAll\(Provider\)](#): Add a lazily calculated collection of elements to the list

Just like every [Provider](#), the collection is calculated when [Provider.get\(\)](#) is called. The following example shows the [ListProperty](#) in action:

build.gradle.kts

```
abstract class Producer : DefaultTask() {
    @get:OutputFile
    abstract val outputFile: RegularFileProperty

    @TaskAction
    fun produce() {
        val message = "Hello, World!"
        val output = outputFile.get().asFile
        output.writeText( message)
        logger.quiet("Wrote '${message}' to ${output}")
    }
}

abstract class Consumer : DefaultTask() {
    @get:InputFiles
    abstract val inputFiles: ListProperty<RegularFile>

    @TaskAction
    fun consume() {
        inputFiles.get().forEach { inputFile ->
            val input = inputFile.asFile
            val message = input.readText()
            logger.quiet("Read '${message}' from ${input}")
        }
    }
}

val producerOne = tasks.register<Producer>("producerOne")
val producerTwo = tasks.register<Producer>("producerTwo")
tasks.register<Consumer>("consumer") {
    // Connect the producer task outputs to the consumer task input
    // Don't need to add task dependencies to the consumer task. These are
    // automatically added
    inputFiles.add(producerOne.get().outputFile)
    inputFiles.add(producerTwo.get().outputFile)
}

// Set values for the producer tasks lazily
// Don't need to update the consumer.inputFiles property. This is
// automatically updated as producer.outputFile changes
```

```

producerOne { outputFile = layout.buildDirectory.file("one.txt") }
producerTwo { outputFile = layout.buildDirectory.file("two.txt") }

// Change the build directory.
// Don't need to update the task properties. These are automatically updated
as the build directory changes
layout.buildDirectory = layout.projectDirectory.dir("output")

```

build.gradle

```

abstract class Producer extends DefaultTask {
    @OutputFile
    abstract RegularFileProperty getOutputFile()

    @TaskAction
    void produce() {
        String message = 'Hello, World!'
        def output = outputFile.get().asFile
        output.text = message
        logger.quiet("Wrote '${message}' to ${output}")
    }
}

abstract class Consumer extends DefaultTask {
    @InputFiles
    abstract ListProperty<RegularFile> getInputFiles()

    @TaskAction
    void consume() {
        inputFiles.get().each { inputFile ->
            def input = inputFile.asFile
            def message = input.text
            logger.quiet("Read '${message}' from ${input}")
        }
    }
}

def producerOne = tasks.register('producerOne', Producer)
def producerTwo = tasks.register('producerTwo', Producer)
tasks.register('consumer', Consumer) {
    // Connect the producer task outputs to the consumer task input
    // Don't need to add task dependencies to the consumer task. These are
    automatically added
    inputFiles.add(producerOne.get().outputFile)
    inputFiles.add(producerTwo.get().outputFile)
}

// Set values for the producer tasks lazily
// Don't need to update the consumer.inputFiles property. This is

```

```
automatically updated as producer.outputFile changes
producerOne.configure { outputFile = layout.buildDirectory.file('one.txt') }
producerTwo.configure { outputFile = layout.buildDirectory.file('two.txt') }

// Change the build directory.
// Don't need to update the task properties. These are automatically updated
as the build directory changes
layout.buildDirectory = layout.projectDirectory.dir('output')
```

```
$ gradle consumer

> Task :producerOne
Wrote 'Hello, World!' to /home/user/gradle/samples/output/one.txt

> Task :producerTwo
Wrote 'Hello, World!' to /home/user/gradle/samples/output/two.txt

> Task :consumer
Read 'Hello, World!' from /home/user/gradle/samples/output/one.txt
Read 'Hello, World!' from /home/user/gradle/samples/output/two.txt

BUILD SUCCESSFUL in 0s
3 actionable tasks: 3 executed
```

```
$ gradle consumer

> Task :producerOne
Wrote 'Hello, World!' to /home/user/gradle/samples/kotlin/output/one.txt

> Task :producerTwo
Wrote 'Hello, World!' to /home/user/gradle/samples/kotlin/output/two.txt

> Task :consumer
Read 'Hello, World!' from /home/user/gradle/samples/kotlin/output/one.txt
Read 'Hello, World!' from /home/user/gradle/samples/kotlin/output/two.txt

BUILD SUCCESSFUL in 0s
3 actionable tasks: 3 executed
```

Working with maps

Gradle provides a lazy [MapProperty](#) type to allow [Map](#) values to be configured. You can create a [MapProperty](#) instance using [ObjectFactory.mapProperty\(Class, Class\)](#).

Similar to other property types, a [MapProperty](#) has a [set\(\)](#) method that you can use to specify the value for the property. Some additional methods allow entries with lazy values to be added to the

map.

build.gradle.kts

```
abstract class Generator: DefaultTask() {
    @get:Input
    abstract val properties: MapProperty<String, Int>

    @TaskAction
    fun generate() {
        properties.get().forEach { entry ->
            logger.quiet("${entry.key} = ${entry.value}")
        }
    }
}

// Some values to be configured later
var b = 0
var c = 0

tasks.register<Generator>("generate") {
    properties.put("a", 1)
    // Values have not been configured yet
    properties.put("b", providers.provider { b })
    properties.putAll(providers.provider { mapOf("c" to c, "d" to c + 1) })
}

// Configure the values. There is no need to reconfigure the task
b = 2
c = 3
```

build.gradle

```
abstract class Generator extends DefaultTask {
    @Input
    abstract MapProperty<String, Integer> getProperties()

    @TaskAction
    void generate() {
        properties.get().each { key, value ->
            logger.quiet("${key} = ${value}")
        }
    }
}

// Some values to be configured later
def b = 0
```

```
def c = 0

tasks.register('generate', Generator) {
    properties.put("a", 1)
    // Values have not been configured yet
    properties.put("b", providers.provider { b })
    properties.putAll(providers.provider { [c: c, d: c + 1] })
}

// Configure the values. There is no need to reconfigure the task
b = 2
c = 3
```

```
$ gradle generate
```

```
> Task :generate
```

```
a = 1
```

```
b = 2
```

```
c = 3
```

```
d = 4
```

```
BUILD SUCCESSFUL in 0s
```

```
1 actionable task: 1 executed
```

Applying a convention to a property

Often, you want to apply some *convention*, or default value to a property to be used if no value has been configured. You can use the `convention()` method for this. This method accepts either a value or a `Provider`, and this will be used as the value until some other value is configured.

build.gradle.kts

```
tasks.register("show") {
    val property = objects.property(String::class)

    // Set a convention
    property.convention("convention 1")

    println("value = " + property.get())

    // Can replace the convention
    property.convention("convention 2")
    println("value = " + property.get())

    property.set("explicit value")
```

```
// Once a value is set, the convention is ignored
property.convention("ignored convention")

doLast {
    println("value = " + property.get())
}
}
```

build.gradle

```
tasks.register("show") {
    def property = objects.property(String)

    // Set a convention
    property.convention("convention 1")

    println("value = " + property.get())

    // Can replace the convention
    property.convention("convention 2")
    println("value = " + property.get())

    property.set("explicit value")

    // Once a value is set, the convention is ignored
    property.convention("ignored convention")

    doLast {
        println("value = " + property.get())
    }
}
```

```
$ gradle show
value = convention 1
value = convention 2

> Task :show
value = explicit value

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Where to apply conventions from?

There are several appropriate locations for setting a convention on a property at configuration time

(i.e., before execution).

build.gradle.kts

```
// setting convention when registering a task from plugin
class GreetingPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        project.getTasks().register<GreetingTask>("hello") {
            greeter.convention("Greeter")
        }
    }
}

apply<GreetingPlugin>()

tasks.withType<GreetingTask>().configureEach {
    // setting convention from build script
    guest.convention("Guest")
}

abstract class GreetingTask : DefaultTask() {
    // setting convention from constructor
    @get:Input
    abstract val guest: Property<String>

    init {
        guest.convention("person2")
    }

    // setting convention from declaration
    @Input
    val greeter = project.objects.property<String>().convention("person1")

    @TaskAction
    fun greet() {
        println("hello, ${guest.get()}, from ${greeter.get()}")
    }
}
```

build.gradle

```
// setting convention when registering a task from plugin
class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        project.getTasks().register("hello", GreetingTask) {
            greeter.convention("Greeter")
        }
    }
}
```

```

    }
}

apply plugin: GreetingPlugin

tasks.withType(GreetingTask).configureEach {
    // setting convention from build script
    guest.convention("Guest")
}

abstract class GreetingTask extends DefaultTask {
    // setting convention from constructor
    @Input
    abstract Property<String> getGuest()

    GreetingTask() {
        guest.convention("person2")
    }

    // setting convention from declaration
    @Input
    final Property<String> greeter = project.objects.property(String)
        .convention("person1")

    @TaskAction
    void greet() {
        println("hello, ${guest.get()}, from ${greeter.get()}")
    }
}

```

From a plugin's `apply()` method

Plugin authors may configure a convention on a lazy property from a plugin's `apply()` method, while performing preliminary configuration of the task or extension defining the property. This works well for regular plugins (meant to be distributed and used in the wild), and internal [convention plugins](#) (which often configure properties defined by third party plugins in a uniform way for the entire build).

build.gradle.kts

```

// setting convention when registering a task from plugin
class GreetingPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        project.getTasks().register<GreetingTask>("hello") {
            greeter.convention("Greeter")
        }
    }
}

```

```
}
```

build.gradle

```
// setting convention when registering a task from plugin
class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        project.getTasks().register("hello", GreetingTask) {
            greeter.convention("Greeter")
        }
    }
}
```

From a build script

Build engineers may configure a convention on a lazy property from shared build logic that is configuring tasks (for instance, from third-party plugins) in a standard way for the entire build.

build.gradle.kts

```
apply<GreetingPlugin>()

tasks.withType<GreetingTask>().configureEach {
    // setting convention from build script
    guest.convention("Guest")
}
```

build.gradle

```
tasks.withType(GreetingTask).configureEach {
    // setting convention from build script
    guest.convention("Guest")
}
```

Note that for project-specific values, instead of conventions, you should prefer setting explicit values (using `Property.set(...)` or `ConfigurableFileCollection.setFrom(...)`, for instance), as conventions are only meant to define defaults.

From the task initialization

A task author may configure a convention on a lazy property from the task constructor or (if in Kotlin) initializer block. This approach works for properties with trivial defaults, but it is not appropriate if additional context (external to the task implementation) is required in order to set a suitable default.

build.gradle.kts

```
// setting convention from constructor
@get:Input
abstract val guest: Property<String>

init {
    guest.convention("person2")
}
```

build.gradle

```
// setting convention from constructor
@Input
abstract Property<String> getGuest()

GreetingTask() {
    guest.convention("person2")
}
```

Next to the property declaration

You may configure a convention on a lazy property next to the place where the property is declared. Note this option is not available for [managed properties](#), and has the same caveats as configuring a convention from the task constructor.

build.gradle.kts

```
// setting convention from declaration
@Input
val greeter = project.objects.property<String>().convention("person1")
```

build.gradle

```
// setting convention from declaration
@Input
final Property<String> greeter = project.objects.property(String).convention
("person1")
```

Making a property unmodifiable

Most properties of a task or project are intended to be configured by plugins or build scripts so that they can use specific values for that build.

For example, a property that specifies the output directory for a compilation task may start with a value specified by a plugin. Then a build script might change the value to some custom location, then this value is used by the task when it runs. However, once the task starts to run, we want to prevent further property changes. This way we avoid errors that result from different consumers, such as the task action, Gradle's up-to-date checks, build caching, or other tasks, using different values for the property.

Lazy properties provide several methods that you can use to disallow changes to their value once the value has been configured. The `finalizeValue()` method calculates the *final* value for the property and prevents further changes to the property.

```
libVersioning.version.finalizeValue()
```

When the property's value comes from a `Provider`, the provider is queried for its current value, and the result becomes the final value for the property. This final value replaces the provider and the property no longer tracks the value of the provider. Calling this method also makes a property instance unmodifiable and any further attempts to change the value of the property will fail. Gradle automatically makes the properties of a task final when the task starts execution.

The `finalizeValueOnRead()` method is similar, except that the property's final value is not calculated until the value of the property is queried.

```
modifiedFiles.finalizeValueOnRead()
```

In other words, this method calculates the final value lazily as required, whereas `finalizeValue()` calculates the final value eagerly. This method can be used when the value may be expensive to calculate or may not have been configured yet. You also want to ensure that all consumers of the property see the same value when they query the value.

Using the Provider API

Guidelines to be successful with the Provider API:

1. The [Property](#) and [Provider](#) types have all of the overloads you need to query or configure a value. For this reason, you should follow the following guidelines:
 - For configurable properties, expose the [Property](#) directly through a single getter.
 - For non-configurable properties, expose an [Provider](#) directly through a single getter.
2. Avoid simplifying calls like `obj.getProperty().get()` and `obj.getProperty().set(T)` in your code by introducing additional getters and setters.
3. When migrating your plugin to use providers, follow these guidelines:
 - If it's a new property, expose it as a [Property](#) or [Provider](#) using a single getter.
 - If it's incubating, change it to use a [Property](#) or [Provider](#) using a single getter.
 - If it's a stable property, add a new [Property](#) or [Provider](#) and deprecate the old one. You should wire the old getter/setters into the new property as appropriate.

Provider Files API Reference

Use these types for *read-only* values:

[Provider](#)<[RegularFile](#)>

File on disk

Factories

- [Provider.map\(Transformer\)](#).
- [Provider.flatMap\(Transformer\)](#).
- [DirectoryProperty.file\(String\)](#)

[Provider](#)<[Directory](#)>

Directory on disk

Factories

- [Provider.map\(Transformer\)](#).
- [Provider.flatMap\(Transformer\)](#).
- [DirectoryProperty.dir\(String\)](#)

[FileCollection](#)

Unstructured collection of files

Factories

- [Project.files\(Object\[\]\)](#)
- [ProjectLayout.files\(Object...\)](#)
- [DirectoryProperty.files\(Object...\)](#)

[FileTree](#)

Hierarchy of files

Factories

- [Project.fileTree\(Object\)](#) will produce a [ConfigurableFileTree](#), or you can use [Project.zipTree\(Object\)](#) and [Project.tarTree\(Object\)](#)
- [DirectoryProperty.getAsFileTree\(\)](#)

Property Files API Reference

Use these types for *mutable* values:

RegularFileProperty

File on disk

Factories

- [ObjectFactory.fileProperty\(\)](#)

DirectoryProperty

Directory on disk

Factories

- [ObjectFactory.directoryProperty\(\)](#)

ConfigurableFileCollection

Unstructured collection of files

Factories

- [ObjectFactory.fileCollection\(\)](#)

ConfigurableFileTree

Hierarchy of files

Factories

- [ObjectFactory.fileTree\(\)](#)

SourceDirectorySet

Hierarchy of source directories

Factories

- [ObjectFactory.sourceDirectorySet\(String, String\)](#)

Lazy Collections API Reference

Use these types for *mutable* values:

ListProperty<T>

a property whose value is [List<T>](#)

Factories

- [ObjectFactory.listProperty\(Class\)](#)

SetProperty<T>

a property whose value is `Set<T>`

Factories

- `ObjectFactory.setProperty(Class)`

Lazy Objects API Reference

Use these types for *read only* values:

Provider<T>

a property whose value is an instance of `T`

Factories

- `Provider.map(Transformer)`.
- `Provider.flatMap(Transformer)`.
- `ProviderFactory.provider(Callable)`. Always prefer one of the other factory methods over this method.

Use these types for *mutable* values:

Property<T>

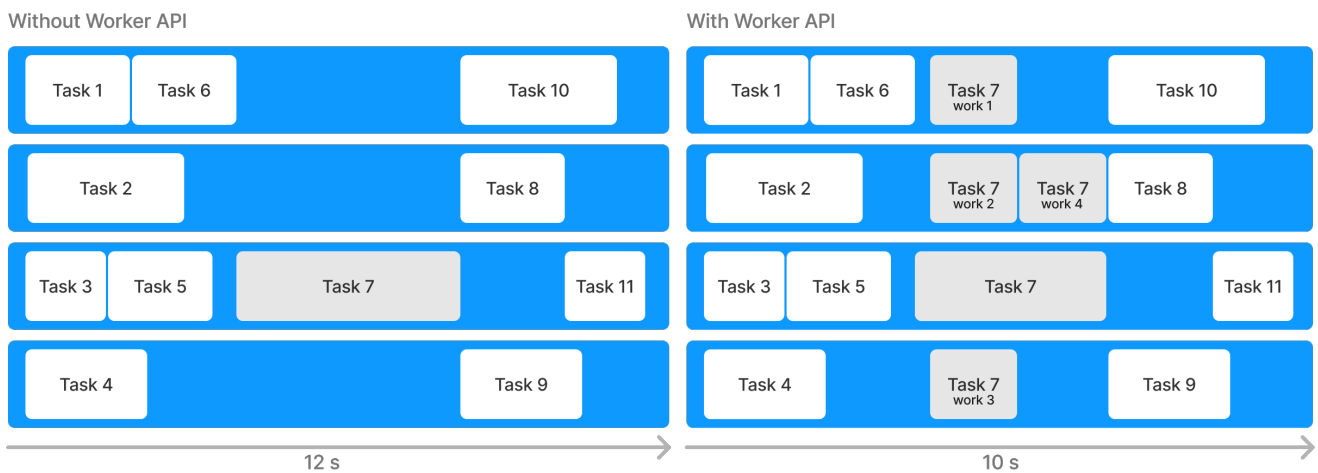
a property whose value is an instance of `T`

Factories

- `ObjectFactory.property(Class)`

Developing Parallel Tasks

Gradle provides an API that can split tasks into sections that can be executed in parallel.



This allows Gradle to fully utilize the resources available and complete builds faster.

The Worker API

The Worker API provides the ability to break up the execution of a task action into discrete units of work and then execute that work concurrently and asynchronously.

Worker API example

The best way to understand how to use the API is to go through the process of converting an existing custom task to use the Worker API:

1. You'll start by creating a custom task class that generates MD5 hashes for a configurable set of files.
2. Then, you'll convert this custom task to use the Worker API.
3. Then, we'll explore running the task with different levels of isolation.

In the process, you'll learn about the basics of the Worker API and the capabilities it provides.

Step 1. Create a custom task class

First, create a custom task that generates MD5 hashes of a configurable set of files.

In a new directory, create a `buildSrc/build.gradle(.kts)` file:

buildSrc/build.gradle.kts

```
repositories {
    mavenCentral()
}

dependencies {
    implementation("commons-io:commons-io:2.5")
    implementation("commons-codec:commons-codec:1.9") ①
}
```

buildSrc/build.gradle

```
repositories {
    mavenCentral()
}

dependencies {
    implementation 'commons-io:commons-io:2.5'
    implementation 'commons-codec:commons-codec:1.9' ①
}
```

- ① Your custom task class will use [Apache Commons Codec](#) to generate MD5 hashes.

Next, create a custom task class in your `buildSrc/src/main/java` directory. You should name this class `CreateMD5`:

buildSrc/src/main/java/CreateMD5.java

```
import org.apache.commons.codec.digest.DigestUtils;
import org.apache.commons.io.FileUtils;
import org.gradle.api.file.DirectoryProperty;
import org.gradle.api.file.RegularFile;
import org.gradle.api.provider.Provider;
import org.gradle.api.tasks.OutputDirectory;
import org.gradle.api.tasks.SourceTask;
import org.gradle.api.tasks.TaskAction;
import org.gradle.workers.WorkerExecutor;

import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;

abstract public class CreateMD5 extends SourceTask { ①

    @OutputDirectory
    abstract public DirectoryProperty getDestinationDirectory(); ②

    @TaskAction
    public void createHashes() {
        for (File sourceFile : getSource().getFiles()) { ③
            try {
                InputStream stream = new FileInputStream(sourceFile);
                System.out.println("Generating MD5 for " + sourceFile.getName() + "
...");
                // Artificially make this task slower.
                Thread.sleep(3000); ④
                Provider<RegularFile> md5File = getDestinationDirectory().file
(sourceFile.getName() + ".md5"); ⑤
                FileUtils.writeStringToFile(md5File.get().getAsFile(), DigestUtils
.md5Hex(stream), (String) null);
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

- ① `SourceTask` is a convenience type for tasks that operate on a set of source files.
- ② The task output will go into a configured directory.
- ③ The task iterates over all the files defined as "source files" and creates an MD5 hash of each.

- ④ Insert an artificial sleep to simulate hashing a large file (the sample files won't be that large).
- ⑤ The MD5 hash of each file is written to the output directory into a file of the same name with an "md5" extension.

Next, create a `build.gradle(.kts)` that registers your new `CreateMD5` task:

build.gradle.kts

```
plugins { id("base") } ①

tasks.register<CreateMD5>("md5") {
    destinationDirectory = project.layout.buildDirectory.dir("md5") ②
    source(project.layout.projectDirectory.file("src")) ③
}
```

build.gradle

```
plugins { id 'base' } ①

tasks.register("md5", CreateMD5) {
    destinationDirectory = project.layout.buildDirectory.dir("md5") ②
    source(project.layout.projectDirectory.file('src')) ③
}
```

- ① Apply the `base` plugin so that you'll have a `clean` task to use to remove the output.
- ② MD5 hash files will be written to `build/md5`.
- ③ This task will generate MD5 hash files for every file in the `src` directory.

You will need some source to generate MD5 hashes from. Create three files in the `src` directory:

src/einstein.txt

Intellectual growth should commence at birth and cease only at death.

src/feynman.txt

I was born not knowing and have had only a little time to change that here and there.

src/hawking.txt

Intelligence is the ability to adapt to change.

At this point, you can test your task by running it `./gradlew md5`:

```
$ gradle md5
```

The output should look similar to:

```
> Task :md5
Generating MD5 for einstein.txt...
Generating MD5 for feynman.txt...
Generating MD5 for hawking.txt...

BUILD SUCCESSFUL in 9s
3 actionable tasks: 3 executed
```

In the `build/md5` directory, you should now see corresponding files with an `md5` extension containing MD5 hashes of the files from the `src` directory. Notice that the task takes at least 9 seconds to run because it hashes each file one at a time (i.e., three files at ~3 seconds apiece).

Step 2. Convert to the Worker API

Although this task processes each file in sequence, the processing of each file is independent of any other file. This work can be done in parallel and take advantage of multiple processors. This is where the Worker API can help.

To use the Worker API, you need to define an interface that represents the parameters of each unit of work and extends `org.gradle.workers.WorkParameters`.

For the generation of MD5 hash files, the unit of work will require two parameters:

1. the file to be hashed and,
2. the file to write the hash to.

There is no need to create a concrete implementation because Gradle will generate one for us at runtime.

buildSrc/src/main/java/MD5WorkParameters.java

```
import org.gradle.api.file.RegularFileProperty;
import org.gradle.workers.WorkParameters;

public interface MD5WorkParameters extends WorkParameters {
    RegularFileProperty getSourceFile(); ①
    RegularFileProperty getMD5File();
}
```

① Use `Property` objects to represent the source and MD5 hash files.

Then, you need to refactor the part of your custom task that does the work for each individual file

into a separate class. This class is your "unit of work" implementation, and it should be an abstract class that extends `org.gradle.workers.WorkAction`:

buildSrc/src/main/java/GenerateMD5.java

```
import org.apache.commons.codec.digest.DigestUtils;
import org.apache.commons.io.FileUtils;
import org.gradle.workers.WorkAction;

import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;

public abstract class GenerateMD5 implements WorkAction<MD5WorkParameters> { ❶
    @Override
    public void execute() {
        try {
            File sourceFile = getParameters().getSourceFile().getAsFile().get();
            File md5File = getParameters().getMD5File().getAsFile().get();
            InputStream stream = new FileInputStream(sourceFile);
            System.out.println("Generating MD5 for " + sourceFile.getName() + "...");
            // Artificially make this task slower.
            Thread.sleep(3000);
            FileUtils.writeStringToFile(md5File, DigestUtils.md5Hex(stream), (String)
null);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

❶ Do not implement the `getParameters()` method - Gradle will inject this at runtime.

Now, change your custom task class to submit work to the `WorkerExecutor` instead of doing the work itself.

buildSrc/src/main/java/CreateMD5.java

```
import org.gradle.api.Action;
import org.gradle.api.file.RegularFile;
import org.gradle.api.provider.Provider;
import org.gradle.api.tasks.*;
import org.gradle.workers.*;
import org.gradle.api.file.DirectoryProperty;

import javax.inject.Inject;
import java.io.File;

abstract public class CreateMD5 extends SourceTask {

    @OutputDirectory
```

```

abstract public DirectoryProperty getDestinationDirectory();

@Inject
abstract public WorkerExecutor getWorkerExecutor(); ①

@TaskAction
public void createHashes() {
    WorkQueue workQueue = getWorkerExecutor().noIsolation(); ②

    for (File sourceFile : getSource().getFiles()) {
        Provider<RegularFile> md5File = getDestinationDirectory().file(sourceFile
        .getName() + ".md5");
        workQueue.submit(GenerateMD5.class, parameters -> { ③
            parameters.getSourceFile().set(sourceFile);
            parameters.getMD5File().set(md5File);
        });
    }
}

```

- ① The `WorkerExecutor` service is required in order to submit your work. Create an abstract getter method annotated `javax.inject.Inject`, and Gradle will inject the service at runtime when the task is created.
- ② Before submitting work, get a `WorkQueue` object with the desired isolation mode (described below).
- ③ When submitting the unit of work, specify the unit of work implementation, in this case `GenerateMD5`, and configure its parameters.

At this point, you should be able to rerun your task:

```

$ gradle clean md5

> Task :md5
Generating MD5 for einstein.txt...
Generating MD5 for feynman.txt...
Generating MD5 for hawking.txt...

BUILD SUCCESSFUL in 3s
3 actionable tasks: 3 executed

```

The results should look the same as before, although the MD5 hash files may be generated in a different order since the units of work are executed in parallel. This time, however, the task runs much faster. This is because the Worker API executes the MD5 calculation for each file in parallel rather than in sequence.

Step 3. Change the isolation mode

The isolation mode controls how strongly Gradle will isolate items of work from each other and the

rest of the Gradle runtime.

There are three methods on `WorkerExecutor` that control this:

1. `noIsolation()`
2. `classLoaderIsolation()`
3. `processIsolation()`

The `noIsolation()` mode is the lowest level of isolation and will prevent a unit of work from changing the project state. This is the fastest isolation mode because it requires the least overhead to set up and execute the work item. However, it will use a single shared classloader for all units of work. This means that each unit of work can affect one another through static class state. It also means that every unit of work uses the same version of libraries on the buildscript classpath. If you wanted the user to be able to configure the task to run with a different (but compatible) version of the [Apache Commons Codec](#) library, you would need to use a different isolation mode.

First, you must change the dependency in `buildSrc/build.gradle` to be `compileOnly`. This tells Gradle that it should use this dependency when building the classes, but should not put it on the build script classpath:

buildSrc/build.gradle.kts

```
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    implementation("commons-io:commons-io:2.5")  
    compileOnly("commons-codec:commons-codec:1.9")  
}
```

buildSrc/build.gradle

```
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    implementation 'commons-io:commons-io:2.5'  
    compileOnly 'commons-codec:commons-codec:1.9'  
}
```

Next, change the `CreateMD5` task to allow the user to configure the version of the codec library that they want to use. It will resolve the appropriate version of the library at runtime and configure the

workers to use this version.

The `classLoaderIsolation()` method tells Gradle to run this work in a thread with an isolated classloader:

buildSrc/src/main/java/CreateMD5.java

```
import org.gradle.api.Action;
import org.gradle.api.file.ConfigurableFileCollection;
import org.gradle.api.file.DirectoryProperty;
import org.gradle.api.file.RegularFile;
import org.gradle.api.provider.Provider;
import org.gradle.api.tasks.*;
import org.gradle.process.JavaForkOptions;
import org.gradle.workers.*;

import javax.inject.Inject;
import java.io.File;
import java.util.Set;

abstract public class CreateMD5 extends SourceTask {

    @InputFiles
    abstract public ConfigurableFileCollection getCodecClasspath(); ①

    @OutputDirectory
    abstract public DirectoryProperty getDestinationDirectory();

    @Inject
    abstract public WorkerExecutor getWorkerExecutor();

    @TaskAction
    public void createHashes() {
        WorkQueue workQueue = getWorkerExecutor().classLoaderIsolation(workerSpec -> {
            workerSpec.getClasspath().from(getCodecClasspath()); ②
        });

        for (File sourceFile : getSource().getFiles()) {
            Provider<RegularFile> md5File = getDestinationDirectory().file(sourceFile
                .getName() + ".md5");
            workQueue.submit(GenerateMD5.class, parameters -> {
                parameters.getSourceFile().set(sourceFile);
                parameters.getMD5File().set(md5File);
            });
        }
    }
}
```

① Expose an input property for the codec library classpath.

② Configure the classpath on the `ClassLoaderWorkerSpec` when creating the work queue.

Next, you need to configure your build so that it has a repository to look up the codec version at task execution time. We also create a dependency to resolve our codec library from this repository:

build.gradle.kts

```
plugins { id("base") }

repositories {
    mavenCentral() ①
}

val codec = configurations.create("codec") { ②
    attributes {
        attribute(Usage.USAGE_ATTRIBUTE, objects.named(Usage.JAVA_RUNTIME))
    }
    isVisible = false
    isCanBeConsumed = false
}

dependencies {
    codec("commons-codec:commons-codec:1.10") ③
}

tasks.register<CreateMD5>("md5") {
    codecClasspath.from(codec) ④
    destinationDirectory = project.layout.buildDirectory.dir("md5")
    source(project.layout.projectDirectory.file("src"))
}
```

build.gradle

```
plugins { id 'base' }

repositories {
    mavenCentral() ①
}

configurations.create('codec') { ②
    attributes {
        attribute(Usage.USAGE_ATTRIBUTE, objects.named(Usage, Usage
        .JAVA_RUNTIME))
    }
    visible = false
    canBeConsumed = false
}

dependencies {
```

```

        codec 'commons-codec:commons-codec:1.10' ③
    }

    tasks.register('md5', CreateMD5) {
        codecClasspath.from(configurations.codec) ④
        destinationDirectory = project.layout.buildDirectory.dir('md5')
        source(project.layout.projectDirectory.file('src'))
    }

```

- ① Add a repository to resolve the codec library - this can be a different repository than the one used to build the `CreateMD5` task class.
- ② Add a *configuration* to resolve our codec library version.
- ③ Configure an alternate, compatible version of [Apache Commons Codec](#).
- ④ Configure the `md5` task to use the configuration as its classpath. Note that the configuration will not be resolved until the task is executed.

Now, if you run your task, it should work as expected using the configured version of the codec library:

```

$ gradle clean md5

> Task :md5
Generating MD5 for einstein.txt...
Generating MD5 for feynman.txt...
Generating MD5 for hawking.txt...

BUILD SUCCESSFUL in 3s
3 actionable tasks: 3 executed

```

Step 4. Create a Worker Daemon

Sometimes, it is desirable to utilize even greater levels of isolation when executing items of work. For instance, external libraries may rely on certain system properties to be set, which may conflict between work items. Or a library might not be compatible with the version of JDK that Gradle is running with and may need to be run with a different version.

The Worker API can accommodate this using the `processIsolation()` method that causes the work to execute in a separate "worker daemon". These worker processes will be session-scoped and can be reused within the same build session, but they won't persist across builds. However, if system resources get low, Gradle will stop unused worker daemons.

To utilize a worker daemon, use the `processIsolation()` method when creating the `WorkQueue`. You may also want to configure custom settings for the new process:

```
import org.gradle.api.Action;
import org.gradle.api.file.ConfigurableFileCollection;
import org.gradle.api.file.DirectoryProperty;
import org.gradle.api.file.RegularFile;
import org.gradle.api.provider.Provider;
import org.gradle.api.tasks.*;
import org.gradle.process.JavaForkOptions;
import org.gradle.workers.*;

import javax.inject.Inject;
import java.io.File;
import java.util.Set;

abstract public class CreateMD5 extends SourceTask {

    @InputFiles
    abstract public ConfigurableFileCollection getCodecClasspath(); ①

    @OutputDirectory
    abstract public DirectoryProperty getDestinationDirectory();

    @Inject
    abstract public WorkerExecutor getWorkerExecutor();

    @TaskAction
    public void createHashes() {
        ①
        WorkQueue workQueue = getWorkerExecutor().processIsolation(workerSpec -> {
            workerSpec.getClasspath().from(getCodecClasspath());
            workerSpec.forkOptions(options -> {
                options.setMaxHeapSize("64m"); ②
            });
        });

        for (File sourceFile : getSource().getFiles()) {
            Provider<RegularFile> md5File = getDestinationDirectory().file(sourceFile
                .getName() + ".md5");
            workQueue.submit(GenerateMD5.class, parameters -> {
                parameters.getSourceFile().set(sourceFile);
                parameters.getMD5File().set(md5File);
            });
        }
    }
}
```

① Change the isolation mode to **PROCESS**.

② Set up the **JavaForkOptions** for the new process.

Now, you should be able to run your task, and it will work as expected but using worker daemons instead:

```
$ gradle clean md5

> Task :md5
Generating MD5 for einstein.txt...
Generating MD5 for feynman.txt...
Generating MD5 for hawking.txt...

BUILD SUCCESSFUL in 3s
3 actionable tasks: 3 executed
```

Note that the execution time may be high. This is because Gradle has to start a new process for each worker daemon, which is expensive.

However, if you run your task a second time, you will see that it runs much faster. This is because the worker daemon(s) started during the initial build have persisted and are available for use immediately during subsequent builds:

```
$ gradle clean md5

> Task :md5
Generating MD5 for einstein.txt...
Generating MD5 for feynman.txt...
Generating MD5 for hawking.txt...

BUILD SUCCESSFUL in 1s
3 actionable tasks: 3 executed
```

Isolation modes

Gradle provides three isolation modes that can be configured when creating a [WorkQueue](#) and are specified using one of the following methods on [WorkerExecutor](#):

`WorkerExecutor.noIsolation()`

This states that the work should be run in a thread with minimal isolation.

For instance, it will share the same classloader that the task is loaded from. This is the fastest level of isolation.

`WorkerExecutor.classLoaderIsolation()`

This states that the work should be run in a thread with an isolated classloader.

The classloader will have the classpath from the classloader that the unit of work implementation class was loaded from as well as any additional classpath entries added through `ClassLoaderWorkerSpec.getClasspath()`.

WorkerExecutor.processIsolation()

This states that the work should be run with a maximum isolation level by executing the work in a separate process.

The classloader of the process will use the classpath from the classloader that the unit of work was loaded from as well as any additional classpath entries added through `ClassLoaderWorkerSpec.getClasspath()`. Furthermore, the process will be a *worker daemon* that will stay alive and can be reused for future work items with the same requirements. This process can be configured with different settings than the Gradle JVM using `ProcessWorkerSpec.forkOptions(org.gradle.api.Action)`.

Worker Daemons

When using `processIsolation()`, Gradle will start a long-lived *worker daemon* process that can be reused for future work items.

build.gradle.kts

```
// Create a WorkQueue with process isolation
val workQueue = workerExecutor.processIsolation() {
    // Configure the options for the forked process
    forkOptions {
        maxHeapSize = "512m"
        systemProperty("org.gradle.sample.showFileSize", "true")
    }
}

// Create and submit a unit of work for each file
source.forEach { file ->
    workQueue.submit(ReverseFile::class) {
        fileToReverse = file
        destinationDir = outputDir
    }
}
```

build.gradle

```
// Create a WorkQueue with process isolation
WorkQueue workQueue = workerExecutor.processIsolation() { ProcessWorkerSpec
spec ->
    // Configure the options for the forked process
    forkOptions { JavaForkOptions options ->
        options.maxHeapSize = "512m"
        options.systemProperty "org.gradle.sample.showFileSize", "true"
    }
}
```

```
// Create and submit a unit of work for each file
source.each { file ->
    workQueue.submit(ReverseFile.class) { ReverseParameters parameters ->
        parameters.fileToReverse = file
        parameters.destinationDir = outputDir
    }
}
```

When a unit of work for a worker daemon is submitted, Gradle will first look to see if a compatible, idle daemon already exists. If so, it will send the unit of work to the idle daemon, marking it as busy. If not, it will start a new daemon. When evaluating compatibility, Gradle looks at a number of criteria, all of which can be controlled through [ProcessWorkerSpec.forkOptions\(org.gradle.api.Action\)](#).

By default, a worker daemon starts with a maximum heap of 512MB. This can be changed by adjusting the workers' fork options.

executable

A daemon is considered compatible only if it uses the same Java executable.

classpath

A daemon is considered compatible if its classpath contains all the classpath entries requested.

Note that a daemon is considered compatible only if the classpath exactly matches the requested classpath.

heap settings

A daemon is considered compatible if it has at least the same heap size settings as requested.

In other words, a daemon that has higher heap settings than requested would be considered compatible.

jvm arguments

A daemon is compatible if it has set all the JVM arguments requested.

Note that a daemon is compatible if it has additional JVM arguments beyond those requested (except for those treated especially, such as heap settings, assertions, debug, etc.).

system properties

A daemon is considered compatible if it has set all the system properties requested with the same values.

Note that a daemon is compatible if it has additional system properties beyond those requested.

environment variables

A daemon is considered compatible if it has set all the environment variables requested with the same values.

Note that a daemon is compatible if it has more environment variables than requested.

bootstrap classpath

A daemon is considered compatible if it contains all the bootstrap classpath entries requested.

Note that a daemon is compatible if it has more bootstrap classpath entries than requested.

debug

A daemon is considered compatible only if debug is set to the same value as requested (**true** or **false**).

enable assertions

A daemon is considered compatible only if enable assertions are set to the same value as requested (**true** or **false**).

default character encoding

A daemon is considered compatible only if the default character encoding is set to the same value as requested.

Worker daemons will remain running until the build daemon that started them is stopped or system memory becomes scarce. When system memory is low, Gradle will stop worker daemons to minimize memory consumption.

NOTE

A step-by-step description of converting a normal task action to use the worker API can be found in the section on [developing parallel tasks](#).

Cancellation and timeouts

To support cancellation (e.g., when the user stops the build with CTRL+C) and task timeouts, custom tasks should react to interrupting their executing thread. The same is true for work items submitted via the worker API. If a task does not respond to an interrupt within 10s, the daemon will shut down to free up system resources.

Advanced Tasks

Incremental tasks

In Gradle, implementing a task that skips execution when its inputs and outputs are already **UP-TO-DATE** is simple and efficient, thanks to the [Incremental Build](#) feature.

However, there are times when only a few input files have changed since the last execution, and it is best to avoid reprocessing all the unchanged inputs. This situation is common in tasks that transform input files into output files on a one-to-one basis.

To optimize your build process you can use an incremental task. This approach ensures that only out-of-date input files are processed, improving build performance.

Implementing an incremental task

For a task to process inputs incrementally, that task must contain an *incremental* task action.

This is a task action method that has a single [InputChanges](#) parameter. That parameter tells Gradle that the action only wants to process the changed inputs.

In addition, the task needs to declare at least one incremental file input property by using either `@Incremental` or `@SkipWhenEmpty`:

build.gradle.kts

```
public class IncrementalReverseTask : DefaultTask() {

    @get:Incremental
    @get:InputDirectory
    val inputDir: DirectoryProperty = project.objects.directoryProperty()

    @get:OutputDirectory
    val outputDir: DirectoryProperty = project.objects.directoryProperty()

    @get:Input
    val inputProperty: RegularFileProperty = project.objects.fileProperty()
    // File input property

    @TaskAction
    fun execute(inputs: InputChanges) { // InputChanges parameter
        val msg = if (inputs.isIncremental) "CHANGED inputs are out of date"
            else "ALL inputs are out of date"
        println(msg)
    }
}
```

build.gradle

```
class IncrementalReverseTask extends DefaultTask {

    @Incremental
    @InputDirectory
    def File inputDir

    @OutputDirectory
    def File outputDir

    @Input
    def inputProperty // File input property

    @TaskAction
    void execute(InputChanges inputs) { // InputChanges parameter
        println inputs.incremental ? "CHANGED inputs are out of date"
            : "ALL inputs are out of date"
    }
}
```

IMPORTANT

To query incremental changes for an input file property, that property must always return the same instance. The easiest way to accomplish this is to use one of the following property types: `RegularFileProperty`, `DirectoryProperty` or `ConfigurableFileCollection`.

You can learn more about `RegularFileProperty` and `DirectoryProperty` in [Lazy Configuration](#).

The incremental task action can use `InputChanges.getFileChanges()` to find out what files have changed for a given file-based input property, be it of type `RegularFileProperty`, `DirectoryProperty` or `ConfigurableFileCollection`.

The method returns an `Iterable` of type `FileChanges`, which in turn can be queried for the following:

- the `affected file`
- the `change type` (`ADDED`, `REMOVED` or `MODIFIED`)
- the `normalized path` of the changed file
- the `file type` of the changed file

The following example demonstrates an incremental task that has a directory input. It assumes that the directory contains a collection of text files and copies them to an output directory, reversing the text within each file:

build.gradle.kts

```
abstract class IncrementalReverseTask : DefaultTask() {
    @get:Incremental
    @get:PathSensitive(PathSensitivity.NAME_ONLY)
    @get:InputDirectory
    abstract val inputDir: DirectoryProperty

    @get:OutputDirectory
    abstract val outputDir: DirectoryProperty

    @get:Input
    abstract val inputProperty: Property<String>

    @TaskAction
    fun execute(inputChanges: InputChanges) {
        println(
            if (inputChanges.isIncremental) "Executing incrementally"
            else "Executing non-incrementally"
        )

        inputChanges.getFileChanges(inputDir).forEach { change ->
            if (change.fileType == FileType.DIRECTORY) return@forEach
        }
    }
}
```

```

        println("${change.changeType}: ${change.normalizedPath}")
        val targetFile =
outputDir.file(change.normalizedPath).get().asFile
        if (change.changeType == ChangeType.REMOVED) {
            targetFile.delete()
        } else {
            targetFile.writeText(change.file.readText().reversed())
        }
    }
}
}

```

build.gradle

```

abstract class IncrementalReverseTask extends DefaultTask {
    @Incremental
    @PathSensitive(PathSensitivity.NAME_ONLY)
    @InputDirectory
    abstract DirectoryProperty getInputDir()

    @OutputDirectory
    abstract DirectoryProperty getOutputDir()

    @Input
    abstract Property<String> getInputProperty()

    @TaskAction
    void execute(InputChanges inputChanges) {
        println(inputChanges.incremental
            ? 'Executing incrementally'
            : 'Executing non-incrementally'
        )

        inputChanges.getFileChanges(inputDir).each { change ->
            if (change.fileType == FileType.DIRECTORY) return

            println "${change.changeType}: ${change.normalizedPath}"
            def targetFile = outputDir.file(change.normalizedPath).get()
.asFile
            if (change.changeType == ChangeType.REMOVED) {
                targetFile.delete()
            } else {
                targetFile.text = change.file.text.reverse()
            }
        }
    }
}

```

NOTE

The type of the `inputDir` property, its annotations, and the `execute()` action use `getFileChanges()` to process the subset of files that have changed since the last build. The action deletes a target file if the corresponding input file has been removed.

If, for some reason, the task is executed non-incrementally (by running with `--rerun-tasks`, for example), all files are reported as `ADDED`, irrespective of the previous state. In this case, Gradle automatically removes the previous outputs, so the incremental task must only process the given files.

For a simple transformer task like the above example, the task action must generate output files for any out-of-date inputs and delete output files for any removed inputs.

IMPORTANT

A task may only contain a single incremental task action.

Which inputs are considered out of date?

When a task has been previously executed, and the only changes since that execution are to incremental input file properties, Gradle can intelligently determine which input files need to be processed, a concept known as incremental execution.

In this scenario, the `InputChanges.getFileChanges()` method, available in the `org.gradle.work.InputChanges` class, provides details for all input files associated with the given property that have been `ADDED`, `REMOVED` or `MODIFIED`.

However, there are many cases where Gradle cannot determine which input files need to be processed (i.e., non-incremental execution). Examples include:

- There is no history available from a previous execution.
- You are building with a different version of Gradle. Currently, Gradle does not use task history from a different version.
- An `upToDateWhen` criterion added to the task returns `false`.
- An input property has changed since the previous execution.
- A non-incremental input file property has changed since the previous execution.
- One or more output files have changed since the previous execution.

In these cases, Gradle will report all input files as `ADDED`, and the `getFileChanges()` method will return details for all the files that comprise the given input property.

You can check if the task execution is incremental or not with the `InputChanges.isIncremental()` method.

An incremental task in action

Consider an instance of `IncrementalReverseTask` executed against a set of inputs for the first time.

In this case, all inputs will be considered `ADDED`, as shown here:

build.gradle.kts

```
tasks.register<IncrementalReverseTask>("incrementalReverse") {  
    inputDir = file("inputs")  
    outputDir = layout.buildDirectory.dir("outputs")  
    inputProperty = project.findProperty("taskInputProperty") as String? ?:  
    "original"  
}
```

build.gradle

```
tasks.register('incrementalReverse', IncrementalReverseTask) {  
    inputDir = file('inputs')  
    outputDir = layout.buildDirectory.dir("outputs")  
    inputProperty = project.properties['taskInputProperty'] ?: 'original'  
}
```

The build layout:

```
.  
├── build.gradle  
└── inputs  
    ├── 1.txt  
    ├── 2.txt  
    └── 3.txt
```

```
$ gradle -q incrementalReverse  
Executing non-incrementally  
ADDED: 1.txt  
ADDED: 2.txt  
ADDED: 3.txt
```

Naturally, when the task is executed again with no changes, then the entire task is **UP-TO-DATE**, and the task action is not executed:

```
$ gradle incrementalReverse  
> Task :incrementalReverse UP-TO-DATE  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 up-to-date
```

When an input file is modified in some way or a new input file is added, then re-executing the task results in those files being returned by `InputChanges.getFileChanges()`.

The following example modifies the content of one file and adds another before running the incremental task:

build.gradle.kts

```
tasks.register("updateInputs") {
    val inputsDir = layout.projectDirectory.dir("inputs")
    outputs.dir(inputsDir)
    doLast {
        inputsDir.file("1.txt").asFile.writeText("Changed content for
existing file 1.")
        inputsDir.file("4.txt").asFile.writeText("Content for new file 4.")
    }
}
```

build.gradle

```
tasks.register('updateInputs') {
    def inputsDir = layout.projectDirectory.dir('inputs')
    outputs.dir(inputsDir)
    doLast {
        inputsDir.file('1.txt').asFile.text = 'Changed content for existing
file 1.'
        inputsDir.file('4.txt').asFile.text = 'Content for new file 4.'
    }
}
```

```
$ gradle -q updateInputs incrementalReverse
Executing incrementally
MODIFIED: 1.txt
ADDED: 4.txt
```

NOTE

The various mutation tasks (`updateInputs`, `removeInput`, etc) are only present to demonstrate the behavior of incremental tasks. They should not be viewed as the kinds of tasks or task implementations you should have in your own build scripts.

When an existing input file is removed, then re-executing the task results in that file being returned by `InputChanges.getFileChanges()` as `REMOVED`.

The following example removes one of the existing files before executing the incremental task:

build.gradle.kts

```
tasks.register<Delete>("removeInput") {  
    delete("inputs/3.txt")  
}
```

build.gradle

```
tasks.register('removeInput', Delete) {  
    delete 'inputs/3.txt'  
}
```

```
$ gradle -q removeInput incrementalReverse  
Executing incrementally  
REMOVED: 3.txt
```

Gradle cannot determine which input files are out-of-date when an *output* file is deleted (or modified). In this case, details for *all* the input files for the given property are returned by `InputChanges.getFileChanges()`.

The following example removes one of the output files from the build directory. However, all the input files are considered to be **ADDED**:

build.gradle.kts

```
tasks.register<Delete>("removeOutput") {  
    delete(layout.buildDirectory.file("outputs/1.txt"))  
}
```

build.gradle

```
tasks.register('removeOutput', Delete) {  
    delete layout.buildDirectory.file("outputs/1.txt")  
}
```

```
$ gradle -q removeOutput incrementalReverse  
Executing non-incrementally  
ADDED: 1.txt
```

```
ADDED: 2.txt  
ADDED: 3.txt
```

The last scenario we want to cover concerns what happens when a non-file-based input property is modified. In such cases, Gradle cannot determine how the property impacts the task outputs, so the task is executed non-incrementally. This means that *all* input files for the given property are returned by `InputChanges.getFileChanges()` and they are all treated as **ADDED**.

The following example sets the project property `taskInputProperty` to a new value when running the `incrementalReverse` task. That project property is used to initialize the task's `inputProperty` property, as you can see in the [first example of this section](#).

Here is the expected output in this case:

```
$ gradle -q -PtaskInputProperty=changed incrementalReverse  
Executing non-incrementally  
ADDED: 1.txt  
ADDED: 2.txt  
ADDED: 3.txt
```

Command Line options

Sometimes, a user wants to declare the value of an exposed task property on the command line instead of the build script. Passing property values on the command line is particularly helpful if they change more frequently.

The task API supports a mechanism for marking a property to automatically generate a corresponding command line parameter with a specific name at runtime.

Step 1. Declare a command-line option

To expose a new command line option for a task property, annotate the corresponding setter method of a property with [Option](#):

```
@Option(option = "flag", description = "Sets the flag")
```

An option requires a mandatory identifier. You can provide an optional description.

A task can expose as many command line options as properties available in the class.

Options may be declared in superinterfaces of the task class as well. If multiple interfaces declare the same property but with different option flags, they will both work to set the property.

In the example below, the custom task `UrlVerify` verifies whether a URL can be resolved by making an HTTP call and checking the response code. The URL to be verified is configurable through the

property `url`. The setter method for the property is annotated with `@Option`:

UrlVerify.java

```
import org.gradle.api.tasks.options.Option;

public class UrlVerify extends DefaultTask {
    private String url;

    @Option(option = "url", description = "Configures the URL to be verified.")
    public void setUrl(String url) {
        this.url = url;
    }

    @Input
    public String getUrl() {
        return url;
    }

    @TaskAction
    public void verify() {
        getLogger().quiet("Verifying URL '{}'", url);

        // verify URL by making a HTTP call
    }
}
```

All options declared for a task can be [rendered as console output](#) by running the `help` task and the `--task` option.

Step 2. Use an option on the command line

There are a few rules for options on the command line:

- The option uses a double-dash as a prefix, e.g., `--url`. A single dash does not qualify as valid syntax for a task option.
- The option argument follows directly after the task declaration, e.g., `verifyUrl --url=http://www.google.com/`.
- Multiple task options can be declared in any order on the command line following the task name.

Building upon the earlier example, the build script creates a task instance of type `UrlVerify` and provides a value from the command line through the exposed option:

build.gradle.kts

```
tasks.register<UrlVerify>("verifyUrl")
```

```
build.gradle
```

```
tasks.register('verifyUrl', UrlVerify)
```

```
$ gradle -q verifyUrl --url=http://www.google.com/  
Verifying URL 'http://www.google.com/'
```

Supported data types for options

Gradle limits the data types that can be used for declaring command line options.

The use of the command line differs per type:

boolean, Boolean, Property<Boolean>

Describes an option with the value `true` or `false`.

Passing the option on the command line treats the value as `true`. For example, `--foo` equates to `true`.

The absence of the option uses the default value of the property. For each boolean option, an opposite option is created automatically. For example, `--no-foo` is created for the provided option `--foo` and `--bar` is created for `--no-bar`. Options whose name starts with `--no` are disabled options and set the option value to `false`. An opposite option is only created if no option with the same name already exists for the task.

Double, Property<Double>

Describes an option with a double value.

Passing the option on the command line also requires a value, e.g., `--factor=2.2` or `--factor 2.2`.

Integer, Property<Integer>

Describes an option with an integer value.

Passing the option on the command line also requires a value, e.g., `--network-timeout=5000` or `--network-timeout 5000`.

Long, Property<Long>

Describes an option with a long value.

Passing the option on the command line also requires a value, e.g., `--threshold=2147483648` or `--threshold 2147483648`.

String, Property<String>

Describes an option with an arbitrary String value.

Passing the option on the command line also requires a value, e.g., `--container-id=2x94held` or `--container-id 2x94held`.

enum, Property<enum>

Describes an option as an enumerated type.

Passing the option on the command line also requires a value e.g., `--log-level=DEBUG` or `--log`

-level debug.

The value is not case-sensitive.

List<T> where T is Double, Integer, Long, String, enum

Describes an option that can take multiple values of a given type.

The values for the option have to be provided as multiple declarations, e.g., `--image-id=123`
`--image-id=456`.

Other notations, such as comma-separated lists or multiple values separated by a space character, are currently not supported.

ListProperty<T>, SetProperty<T> where T is Double, Integer, Long, String, enum

Describes an option that can take multiple values of a given type.

The values for the option have to be provided as multiple declarations, e.g., `--image-id=123`
`--image-id=456`.

Other notations, such as comma-separated lists or multiple values separated by a space character, are currently not supported.

DirectoryProperty, RegularFileProperty

Describes an option with a file system element.

Passing the option on the command line also requires a value representing a path, e.g., `--output`
`-file=file.txt` or `--output-dir outputDir`.

Relative paths are resolved relative to the project directory of the project that owns this property instance. See `FileSystemLocationProperty.set()`.

Documenting available values for an option

Theoretically, an option for a property type `String` or `List<String>` can accept any arbitrary value. Accepted values for such an option can be documented programmatically with the help of the annotation `OptionValues`:

```
@OptionValues('file')
```

This annotation may be assigned to any method that returns a `List` of one of the supported data types. You need to specify an option identifier to indicate the relationship between the option and available values.

NOTE

Passing a value on the command line not supported by the option does not fail the build or throw an exception. You must implement custom logic for such behavior in the task action.

The example below demonstrates the use of multiple options for a single task. The task implementation provides a list of available values for the option `output-type`:

UrlProcess.java

```
import org.gradle.api.tasks.options.Option;  
import org.gradle.api.tasks.options.OptionValues;
```

```

public abstract class UrlProcess extends DefaultTask {
    private String url;
    private OutputType outputType;

    @Input
    @Option(option = "http", description = "Configures the http protocol to be
allowed.")
    public abstract Property<Boolean> getHttp();

    @Option(option = "url", description = "Configures the URL to send the request to.
")
    public void setUrl(String url) {
        if (!getHttp().getOrElse(true) && url.startsWith("http://")) {
            throw new IllegalArgumentException("HTTP is not allowed");
        } else {
            this.url = url;
        }
    }

    @Input
    public String getUrl() {
        return url;
    }

    @Option(option = "output-type", description = "Configures the output type.")
    public void setOutputType(OutputType outputType) {
        this.outputType = outputType;
    }

    @OptionValues("output-type")
    public List<OutputType> getAvailableOutputTypes() {
        return new ArrayList<OutputType>(Arrays.asList(OutputType.values()));
    }

    @Input
    public OutputType getOutputType() {
        return outputType;
    }

    @TaskAction
    public void process() {
        getLogger().quiet("Writing out the URL response from '{}{}' to '{}{}'", url,
outputType);

        // retrieve content from URL and write to output
    }

    private static enum OutputType {
        CONSOLE, FILE
    }
}

```

```
}
```

Listing command line options

Command line options using the annotations [Option](#) and [OptionValues](#) are self-documenting.

You will see [declared options](#) and their [available values](#) reflected in the console output of the [help](#) task. The output renders options alphabetically, except for boolean disable options, which appear following the enable option:

```
$ gradle -q help --task processUrl
Detailed task information for processUrl

Path
    :processUrl

Type
    UrlProcess (UrlProcess)

Options
    --http      Configures the http protocol to be allowed.

    --no-http   Disables option --http.

    --output-type  Configures the output type.
                   Available values are:
                   CONSOLE
                   FILE

    --url       Configures the URL to send the request to.

    --rerun     Causes the task to be re-run even if up-to-date.

Description
    -

Group
    -
```

Limitations

Support for declaring command line options currently comes with a few limitations.

- Command line options can only be declared for custom tasks via annotation. There's no programmatic equivalent for defining options.
- Options cannot be declared globally, e.g., on a project level or as part of a plugin.
- When assigning an option on the command line, the task exposing the option needs to be spelled out explicitly, e.g., `gradle check --tests abc` does not work even though the `check` task

depends on the `test` task.

- If you specify a task option name that conflicts with the name of a built-in Gradle option, use the `--` delimiter before calling your task to reference that option. For more information, see [Disambiguate Task Options from Built-in Options](#).

Verification failures

Normally, exceptions thrown during task execution result in a failure that immediately terminates a build. The outcome of the task will be `FAILED`, the result of the build will be `FAILED`, and no further tasks will be executed. When [running with the `--continue` flag](#), Gradle will continue to run other requested tasks in the build after encountering a task failure. However, any tasks that depend on a failed task will not be executed.

There is a special type of exception that behaves differently when downstream tasks only rely on the outputs of a failing task. A task can throw a subtype of `VerificationException` to indicate that it has failed in a controlled manner such that its output is still valid for consumers. A task depends on the **outcome** of another task when it directly depends on it using `dependsOn`. When Gradle is run with `--continue`, consumer tasks that depend on a producer task's output (via a relationship between task inputs and outputs) can still run after the producer fails.

A failed unit test, for instance, will cause a failing outcome for the test task. However, this doesn't prevent another task from reading and processing the (valid) test results the task produced. Verification failures are used in exactly this manner by the [Test Report Aggregation Plugin](#).

Verification failures are also useful for tasks that need to report a failure even after producing useful output consumable by other tasks.

build.gradle.kts

```
val process = tasks.register("process") {
    val outputFile = layout.buildDirectory.file("processed.log")
    outputs.files(outputFile) ❶

    doLast {
        val logFile = outputFile.get().asFile
        logFile.appendText("Step 1 Complete.") ❷
        throw VerificationException("Process failed!") ❸
        logFile.appendText("Step 2 Complete.") ❹
    }
}

tasks.register("postProcess") {
    inputs.files(process) ❺

    doLast {
        println("Results: ${inputs.files.singleFile.readText()}") ❻
    }
}
```

```
}
```

build.gradle

```
tasks.register("process") {
    def outputFile = layout.buildDirectory.file("processed.log")
    outputs.files(outputFile) ①

    doLast {
        def logFile = outputFile.get().asFile
        logFile << "Step 1 Complete." ②
        throw new VerificationException("Process failed!") ③
        logFile << "Step 2 Complete." ④
    }
}

tasks.register("postProcess") {
    inputs.files(tasks.named("process")) ⑤

    doLast {
        println("Results: ${inputs.files.singleFile.text}") ⑥
    }
}
```

```
$ gradle postProcess --continue
> Task :process FAILED

> Task :postProcess
Results: Step 1 Complete.
2 actionable tasks: 2 executed

FAILURE: Build failed with an exception.
```

- ① **Register Output:** The `process` task writes its output to a log file.
- ② **Modify Output:** The task writes to its output file as it executes.
- ③ **Task Failure:** The task throws a `VerificationException` and fails at this point.
- ④ **Continue to Modify Output:** This line never runs due to the exception stopping the task.
- ⑤ **Consume Output:** The `postProcess` task depends on the output of the `process` task due to using that task's outputs as its own inputs.
- ⑥ **Use Partial Result:** With the `--continue` flag set, Gradle still runs the requested `postProcess` task despite the `process` task's failure. `postProcess` can read and display the partial (though still valid) result.

Using Shared Build Services

Shared build services allow tasks to share state or resources. For example, tasks might share a cache of pre-computed values or use a web service or database instance.

A build service is an object that holds the state for tasks to use. It provides an alternative mechanism for hooking into a Gradle build and receiving information about task execution and operation completion.

Build services are configuration cacheable.

Gradle manages the service lifecycle, creating the service instance only when required and cleaning it up when no longer needed. Gradle can also coordinate access to the build service, ensuring that no more than a specified number of tasks use the service concurrently.

Implementing a build service

To implement a build service, create an abstract class that implements [BuildService](#). Then, define methods you want the tasks to use on this type.

```
abstract class BaseCountingService implements BuildService<CountingParams>,
    AutoCloseable {

}
```

A build service implementation is treated as a [custom Gradle type](#) and can use any of the features available to custom Gradle types.

A build service can optionally take parameters, which Gradle injects into the service instance when creating it. To provide parameters, you define an abstract class (or interface) that holds the parameters. The parameters type must implement (or extend) [BuildServiceParameters](#). The service implementation can access the parameters using `this.getParameters()`. The parameters type is also a [custom Gradle type](#).

When the build service does not require any parameters, you can use [BuildServiceParameters.None](#) as the type of parameter.

```
interface CountingParams extends BuildServiceParameters {
    Property<Integer> getInitial()
}
```

A build service implementation can also optionally implement [AutoCloseable](#), in which case Gradle will call the build service instance's `close()` method when it discards the service instance. This happens sometime between the completion of the last task that uses the build service and the end of the build.

Here is an example of a service that takes parameters and is closeable:

WebServer.java

```
import org.gradle.api.file.DirectoryProperty;
import org.gradle.api.provider.Property;
import org.gradle.api.services.BuildService;
import org.gradle.api.services.BuildServiceParameters;

import java.net.URI;
import java.net.URISyntaxException;

public abstract class WebServer implements BuildService<WebServer.Params>,
    AutoCloseable {

    // Some parameters for the web server
    interface Params extends BuildServiceParameters {
        Property<Integer> getPort();

        DirectoryProperty getResources();
    }

    private final URI uri;

    public WebServer() throws URISyntaxException {
        // Use the parameters
        int port = getParameters().getPort().get();
        uri = new URI(String.format("https://localhost:%d/", port));

        // Start the server ...

        System.out.println(String.format("Server is running at %s", uri));
    }

    // A public method for tasks to use
    public URI getUri() {
        return uri;
    }

    @Override
    public void close() {
        // Stop the server ...
    }
}
```

Note that you should **not** implement the `BuildService.getParameters()` method, as Gradle will provide an implementation of this.

A build service implementation must be thread-safe, as it will potentially be used by multiple tasks concurrently.

Registering a build service and connecting it to a task

To create a build service, you register the service instance using the `BuildServiceRegistry.registerIfAbsent()` method.

Registering the service does not create the service instance. This happens on demand when a task first uses the service. The service instance will not be created if no task uses the service during a build.

Currently, build services are scoped to a build, rather than a project, and these services are available to be shared by the tasks of all projects. You can access the registry of shared build services via `Project.getGradle().getSharedServices()`.

Registering a build service to be consumed via `@ServiceReference` task properties

Here is an example of a plugin that registers the previous service when the task property consuming the service is annotated with `@ServiceReference`:

DownloadPlugin.java

```
import org.gradle.api.Plugin;
import org.gradle.api.Project;
import org.gradle.api.provider.Provider;

public class DownloadPlugin implements Plugin<Project> {
    public void apply(Project project) {
        // Register the service
        project.getGradle().getSharedServices().registerIfAbsent("web", WebServer
.class, spec -> {
            // Provide some parameters
            spec.getParameters().getPort().set(5005);
        });

        project.getTasks().register("download", Download.class, task -> {
            task.getOutputFile().set(project.getLayout().getBuildDirectory().file
("result.zip"));
        });
    }
}
```

As you can see, there is no need to assign the build service provider returned by `registerIfAbsent()` to the task, the service is automatically injected into all matching properties that were annotated with `@ServiceReference`.

Here is an example of a task that consumes the previous service via a property annotated with `@ServiceReference`:

Download.java

```
import org.gradle.api.DefaultTask;
import org.gradle.api.file.RegularFileProperty;
import org.gradle.api.provider.Property;
import org.gradle.api.services.ServiceReference;
import org.gradle.api.tasks.OutputFile;
import org.gradle.api.tasks.TaskAction;

import java.net.URI;

public abstract class Download extends DefaultTask {
    // This property provides access to the service instance
    @ServiceReference("web")
    abstract Property<WebServer> getServer();

    @OutputFile
    abstract RegularFileProperty getOutputFile();

    @TaskAction
    public void download() {
        // Use the server to download a file
        WebServer server = getServer().get();
        URI uri = server.getUri().resolve("somefile.zip");
        System.out.println(String.format("Downloading %s", uri));
    }
}
```

Automatic matching of registered build services with service reference properties is done by type and (optionally) by name (for properties that declare the name of the service they expect). In case multiple services would match the requested service type (i.e. multiple services were registered for the same type, and a service name was not provided in the `@ServiceReference` annotation), you will need also to assign the shared build service provider manually to the task property.

Read on to compare that to when the task property consuming the service is instead annotated with `@Internal`.

Registering a build service to be consumed via `@Internal` task properties

DownloadPlugin.java

```
import org.gradle.api.Plugin;
import org.gradle.api.Project;
import org.gradle.api.provider.Provider;

public class DownloadPlugin implements Plugin<Project> {
    public void apply(Project project) {
        // Register the service
    }
}
```

```

    Provider<WebServer> serviceProvider = project.getGradle()
        .getSharedServices().registerIfAbsent("web", WebServer.class, spec -> {
            // Provide some parameters
            spec.getParameters().getPort().set(5005);
        });

    project.getTasks().register("download", Download.class, task -> {
        // Connect the service provider to the task
        task.getServer().set(serviceProvider);
        // Declare the association between the task and the service
        task.useService(serviceProvider);
        task.getOutputFile().set(project.getLayout().getBuildDirectory().file
            ("result.zip"));
    });
}
}

```

In this case, the plugin registers the service and receives a `Provider<WebService>` back. This provider can be connected to task properties to pass the service to the task. Note that for a task property annotated with `@Internal`, the task property needs to (1) be explicitly assigned with the provider obtained during registration, and (2) you must tell Gradle the task uses the service via `Task.useService`. None of that is needed when the task property consuming the service is annotated with `@ServiceReference`.

Here is an example of a task that consumes the previous service via a property annotated with `@Internal`:

Download.java

```

import org.gradle.api.DefaultTask;
import org.gradle.api.file.RegularFileProperty;
import org.gradle.api.provider.Property;
import org.gradle.api.tasks.Internal;
import org.gradle.api.tasks.OutputFile;
import org.gradle.api.tasks.TaskAction;

import java.net.URI;

public abstract class Download extends DefaultTask {
    // This property provides access to the service instance
    @Internal
    abstract Property<WebServer> getServer();

    @OutputFile
    abstract RegularFileProperty getOutputFile();

    @TaskAction
    public void download() {
        // Use the server to download a file
    }
}

```

```

        WebServer server = getServer().get();
        URI uri = server.getUri().resolve("somefile.zip");
        System.out.println(String.format("Downloading %s", uri));
    }
}

```

Note that using a service with any annotation other than `@ServiceReference` or `@Internal` is currently not supported. For example, it is currently impossible to mark a service as an input to a task.

Using shared build services from configuration actions

Generally, build services are intended to be used by tasks, and as they usually represent some potentially expensive state to create, you should avoid using them at configuration time. However, sometimes, using the service at configuration time can make sense. This is possible; call `get()` on the provider.

Using a build service with the Worker API

In addition to using a build service from a task, you can use a build service from a [Worker API action](#), an [artifact transform](#) or another build service. To do this, pass the build service `Provider` as a parameter of the consuming action or service, in the same way you pass other parameters to the action or service.

For example, to pass a `MyServiceType` service to Worker API action, you might add a property of type `Property<MyServiceType>` to the action's parameters object and then connect the `Provider<MyServiceType>` that you receive when registering the service to this property:

Download.java

```

import org.gradle.api.DefaultTask;
import org.gradle.api.provider.Property;
import org.gradle.api.services.ServiceReference;
import org.gradle.api.tasks.TaskAction;
import org.gradle.workers.WorkAction;
import org.gradle.workers.WorkParameters;
import org.gradle.workers.WorkQueue;
import org.gradle.workers.WorkerExecutor;

import javax.inject.Inject;
import java.net.URI;

public abstract class Download extends DefaultTask {

    public static abstract class DownloadWorkAction implements WorkAction
    <DownloadWorkAction.Parameters> {
        interface Parameters extends WorkParameters {
            // This property provides access to the service instance from the work
            action

```

```

        abstract Property<WebServer> getServer();
    }

    @Override
    public void execute() {
        // Use the server to download a file
        WebServer server = getParameters().getServer().get();
        URI uri = server.getUri().resolve("somefile.zip");
        System.out.println(String.format("Downloading %s", uri));
    }
}

@Inject
abstract public WorkerExecutor getWorkerExecutor();

// This property provides access to the service instance from the task
@ServiceReference("web")
abstract Property<WebServer> getServer();

@TaskAction
public void download() {
    WorkQueue workQueue = getWorkerExecutor().noIsolation();
    workQueue.submit(DownloadWorkAction.class, parameter -> {
        parameter.getServer().set(getServer());
    });
}
}

```

Currently, it is impossible to use a build service with a worker API action that uses `ClassLoader` or process isolation modes.

Accessing the build service concurrently

You can constrain concurrent execution when you register the service, by using the `Property` object returned from `BuildServiceSpec.getMaxParallelUsages()`. When this property has no value, which is the default, Gradle does not constrain access to the service. When this property has a value > 0 , Gradle will allow no more than the specified number of tasks to use the service concurrently.

IMPORTANT

When the consuming task property is annotated with `@Internal`, for the constraint to take effect, the build service **must** be registered with the consuming task via `Task.usesService`. NOTE: at this time, Gradle cannot discover indirect usage of services (for instance, if an additional service is used only by a service that the task uses directly). As a workaround, indirect usage may be declared explicitly to Gradle by either adding a `@ServiceReference` property to the task and assigning the service that is only used indirectly to it (making it a direct reference), or invoking `Task.usesService`.

Receiving information about task execution

A build service can be used to receive events as tasks are executed. To do this, create and register a build service that implements [OperationCompletionListener](#):

TaskEventsService.java

```
import org.gradle.api.services.BuildService;
import org.gradle.api.services.BuildServiceParameters;
import org.gradle.tooling.events.FinishEvent;
import org.gradle.tooling.events.OperationCompletionListener;
import org.gradle.tooling.events.task.TaskFinishEvent;

public abstract class TaskEventsService implements BuildService
<BuildServiceParameters.None>,
    OperationCompletionListener { ①

    @Override
    public void onFinish(FinishEvent finishEvent) {
        if (finishEvent instanceof TaskFinishEvent) { ②
            // Handle task finish event...
        }
    }
}
```

① Implement the [OperationCompletionListener](#) interface and the [BuildService](#) interface.

② Check if the finish event is a [TaskFinishEvent](#).

Then, in the plugin, you can use the methods on the [BuildEventsListenerRegistry](#) service to start receiving events:

TaskEventsPlugin.java

```
import org.gradle.api.Plugin;
import org.gradle.api.Project;
import org.gradle.api.provider.Provider;
import org.gradle.build.event.BuildEventsListenerRegistry;

import javax.inject.Inject;

public abstract class TaskEventsPlugin implements Plugin<Project> {
    @Inject
    public abstract BuildEventsListenerRegistry getEventsListenerRegistry(); ①

    @Override
    public void apply(Project project) {
        Provider<TaskEventsService> serviceProvider =
            project.getGradle().getSharedServices().registerIfAbsent(
```

```
        "taskEvents", TaskEventsService.class, spec -> {}); ②  
        getEventsListenerRegistry().onTaskCompletion(serviceProvider); ③  
    }  
}
```

- ① Use [service injection](#) to obtain an instance of the `BuildEventsListenerRegistry`.
- ② Register the build service as usual.
- ③ Use the service `Provider` to subscribe to the build service to build events.

[1] You might be wondering why there is neither an import for the `StopExecutionException` nor do we access it via its fully qualified name. The reason is that Gradle adds a set of default imports to your script (see [Default imports](#)).

DEVELOPING PLUGINS

Understanding Plugins

Gradle comes with a set of powerful core systems such as dependency management, task execution, and project configuration. But everything else it can do is supplied by plugins.

Plugins encapsulate logic for specific tasks or integrations, such as compiling code, running tests, or deploying artifacts. By applying plugins, users can easily add new features to their build process without having to write complex code from scratch.

This plugin-based approach allows Gradle to be lightweight and modular. It also promotes code reuse and maintainability, as plugins can be shared across projects or within an organization.

Before reading this chapter, it's recommended that you first read [Learning The Basics](#) and complete the [Tutorial](#).

Plugins Introduction

Plugins can be sourced from Gradle or the Gradle community. But when users want to organize their build logic or need specific build capabilities not provided by existing plugins, they can develop their own.

As such, we distinguish between three different kinds of plugins:

1. **Core Plugins** - plugins that come from Gradle.
2. **Community Plugins** - plugins that come from [Gradle Plugin Portal](#) or a public repository.
3. **Local or Custom Plugins** - plugins that you develop yourself.

Core Plugins

The term **core plugin** refers to a plugin that is part of the Gradle distribution such as the [Java Library Plugin](#). They are always available.

Community Plugins

The term **community plugin** refers to a plugin published to the Gradle Plugin Portal (or another public repository) such as the [Spotless Plugin](#).

Local or Custom Plugins

The term **local or custom plugin** refers to a plugin you write yourself for your own build.

Custom plugins

There are three types of **custom plugins**:

#	Type	Location:	Most likely:	Benefit:
1	Script plugins	A <code>.gradle(.kts)</code> script file	A local plugin	Plugin is automatically compiled and included in the classpath of the build script.
2	Precompiled script plugins	<code>buildSrc</code> folder or <code>composite</code> build	A convention plugin	Plugin is automatically compiled, tested, and available on the classpath of the build script. The plugin is visible to every build script used by the build.
3	Binary plugins	Standalone project	A shared plugin	Plugin JAR is produced and published. The plugin can be used in multiple builds and shared with others.

Script plugins

Script plugins are typically small, local plugins written in script files for tasks specific to a single build or project. They do not need to be reused across multiple projects. Script plugins **are not recommended** but many other forms of plugins evolve from script plugins.

To create a plugin, you need to write a class that implements the [Plugin](#) interface.

The following sample creates a `GreetingPlugin`, which adds a `hello` task to a project when applied:

build.gradle.kts

```
class GreetingPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        project.task("hello") {
            doLast {
                println("Hello from the GreetingPlugin")
            }
        }
    }
}
```

```
// Apply the plugin
apply<GreetingPlugin>()
```

build.gradle

```
class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        project.task('hello') {
            doLast {
                println 'Hello from the GreetingPlugin'
            }
        }
    }
}

// Apply the plugin
apply plugin: GreetingPlugin
```

```
$ gradle -q hello
Hello from the GreetingPlugin
```

The **Project** object is passed as a parameter in **apply()**, which the plugin can use to configure the project however it needs to (such as adding tasks, configuring dependencies, etc.). In this example, the plugin is written directly in the build file which is **not a recommended practice**.

When the plugin is written in a separate script file, it can be applied using **apply(from = "file_name.gradle.kts")** or **apply from: 'file_name.gradle'**. In the example below, the plugin is coded in the **other.gradle(.kts)** script file. Then, the **other.gradle(.kts)** is applied to **build.gradle(.kts)** using **apply from:**

other.gradle.kts

```
class GreetingScriptPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        project.task("hi") {
            doLast {
                println("Hi from the GreetingScriptPlugin")
            }
        }
    }
}
```

```
// Apply the plugin
apply<GreetingScriptPlugin>()
```

other.gradle

```
class GreetingScriptPlugin implements Plugin<Project> {
    void apply(Project project) {
        project.task('hi') {
            doLast {
                println 'Hi from the GreetingScriptPlugin'
            }
        }
    }
}

// Apply the plugin
apply plugin: GreetingScriptPlugin
```

build.gradle.kts

```
apply(from = "other.gradle.kts")
```

build.gradle

```
apply from: 'other.gradle'
```

```
$ gradle -q hi
Hi from the GreetingScriptPlugin
```

Script plugins should be avoided.

Precompiled script plugins

Precompiled script plugins are compiled into class files and packaged into a JAR before they are executed. These plugins use the Groovy DSL or Kotlin DSL instead of pure Java, Kotlin, or Groovy. They are best used as **convention plugins** that share build logic across projects or as a way to neatly organize build logic.

To create a precompiled script plugin, you can:

1. Use Gradle's Kotlin DSL - The plugin is a `.gradle.kts` file, and apply `kotlin-dsl`.
2. Use Gradle's Groovy DSL - The plugin is a `.gradle` file, and apply `id("groovy-gradle-plugin")`.

To apply a precompiled script plugin, you need to know its ID. The ID is derived from the plugin script's filename and its (optional) package declaration.

For example, the script `src/main/*/some-java-library.gradle(.kts)` has a plugin ID of `some-java-library` (assuming it has no package declaration). Likewise, `src/main/*/my/some-java-library.gradle(.kts)` has a plugin ID of `my.some-java-library` as long as it has a package declaration of `my`.

Precompiled script plugin names have two important limitations:

- They cannot start with `org.gradle`.
- They cannot have the same name as a `core plugin`.

When the plugin is applied to a project, Gradle creates an instance of the plugin class and calls the instance's `Plugin.apply()` method.

NOTE | A new instance of a `Plugin` is created within each project applying that plugin.

Let's rewrite the `GreetingPlugin` script plugin as a precompiled script plugin. Since we are using the Groovy or Kotlin DSL, the file essentially becomes the plugin. The original script plugin simply created a `hello` task which printed a greeting, this is what we will do in the pre-compiled script plugin:

buildSrc/src/main/kotlin/GreetingPlugin.gradle.kts

```
tasks.register("hello") {
    doLast {
        println("Hello from the convention GreetingPlugin")
    }
}
```

buildSrc/src/main/groovy/GreetingPlugin.gradle

```
tasks.register("hello") {
    doLast {
        println("Hello from the convention GreetingPlugin")
    }
}
```

The `GreetingPlugin` can now be applied in other subprojects' builds by using its ID:

app/build.gradle.kts

```
plugins {  
    application  
    id("GreetingPlugin")  
}
```

app/build.gradle

```
plugins {  
    id 'application'  
    id('GreetingPlugin')  
}
```

```
$ gradle -q hello  
Hello from the convention GreetingPlugin
```

Convention plugins

A **convention plugin** is typically a precompiled script plugin that configures existing core and community plugins with your own conventions (i.e. default values) such as setting the Java version by using `java.toolchain.languageVersion = JavaLanguageVersion.of(17)`. Convention plugins are also used to enforce project standards and help streamline the build process. They can apply and configure plugins, create new tasks and extensions, set dependencies, and much more.

Let's take an example build with three subprojects: one for `data-model`, one for `database-logic` and one for `app` code. The project has the following structure:

```
.  
├── buildSrc  
│   ├── src  
│   │   └── ...  
│   └── build.gradle.kts  
├── data-model  
│   ├── src  
│   │   └── ...  
│   └── build.gradle.kts  
├── database-logic  
│   ├── src  
│   │   └── ...  
│   └── build.gradle.kts  
└── app
```



The build file of the **database-logic** subproject is as follows:

database-logic/build.gradle.kts

```
plugins {
    id("java-library")
    id("org.jetbrains.kotlin.jvm") version "2.0.21"
}

repositories {
    mavenCentral()
}

java {
    toolchain.languageVersion.set(JavaLanguageVersion.of(11))
}

tasks.test {
    useJUnitPlatform()
}

kotlin {
    jvmToolchain(11)
}

// More build logic
```

database-logic/build.gradle

```
plugins {
    id 'java-library'
    id 'org.jetbrains.kotlin.jvm' version '2.0.21'
}

repositories {
    mavenCentral()
}

java {
    toolchain.languageVersion.set(JavaLanguageVersion.of(11))
}
```

```

tasks.test {
    useJUnitPlatform()
}

kotlin {
    jvmToolchain {
        languageVersion.set(JavaLanguageVersion.of(11))
    }
}

// More build logic

```

We apply the `java-library` plugin and add the `org.jetbrains.kotlin.jvm` plugin for Kotlin support. We also configure Kotlin, Java, tests and more.

Our build file is beginning to grow...

The more plugins we apply and the more plugins we configure, the larger it gets. There's also repetition in the build files of the `app` and `data-model` subprojects, especially when configuring common extensions like setting the Java version and Kotlin support.

To address this, we use convention plugins. This allows us to avoid repeating configuration in each build file and keeps our build scripts more concise and maintainable. In convention plugins, we can encapsulate arbitrary build configuration or custom build logic.

To develop a convention plugin, we recommend using `buildSrc` – which represents a completely separate Gradle build. `buildSrc` has its own settings file to define where dependencies of this build are located.

We add a Kotlin script called `my-java-library.gradle.kts` inside the `buildSrc/src/main/kotlin` directory. Or conversely, a Groovy script called `my-java-library.gradle` inside the `buildSrc/src/main/groovy` directory. We put all the plugin application and configuration from the `database-logic` build file into it:

buildSrc/src/main/kotlin/my-java-library.gradle.kts

```

plugins {
    id("java-library")
    id("org.jetbrains.kotlin.jvm")
}

repositories {
    mavenCentral()
}

java {
    toolchain.languageVersion.set(JavaLanguageVersion.of(11))
}

```

```

}

tasks.test {
    useJUnitPlatform()
}

kotlin {
    jvmToolchain(11)
}

```

buildSrc/src/main/groovy/my-java-library.gradle

```

plugins {
    id 'java-library'
    id 'org.jetbrains.kotlin.jvm'
}

repositories {
    mavenCentral()
}

java {
    toolchain.languageVersion.set(JavaLanguageVersion.of(11))
}

tasks.test {
    useJUnitPlatform()
}

kotlin {
    jvmToolchain {
        languageVersion.set(JavaLanguageVersion.of(11))
    }
}

```

The name of the file `my-java-library` is the ID of our brand-new plugin, which we can now use in all of our subprojects.

TIP

Why is the version of `id 'org.jetbrains.kotlin.jvm'` missing? See [Applying External Plugins to Pre-Compiled Script Plugins](#).

The `database-logic` build file becomes much simpler by removing all the redundant build logic and applying our convention `my-java-library` plugin instead:

database-logic/build.gradle.kts

```
plugins {  
    id("my-java-library")  
}
```

database-logic/build.gradle

```
plugins {  
    id('my-java-library')  
}
```

This convention plugin enables us to easily share common configurations across all our build files. Any modifications can be made in one place, simplifying maintenance.

Binary plugins

Binary plugins in Gradle are plugins that are built as standalone JAR files and applied to a project using the `plugins{}` block in the build script.

Let's move our `GreetingPlugin` to a standalone project so that we can publish it and share it with others. The plugin is essentially moved from the `buildSrc` folder to its own build called `greeting-plugin`.

NOTE

You can publish the plugin from `buildSrc`, but this is not recommended practice. Plugins that are ready for publication should be in their own build.

`greeting-plugin` is simply a Java project that produces a JAR containing the plugin classes.

The easiest way to package and publish a plugin to a repository is to use the [Gradle Plugin Development Plugin](#). This plugin provides the necessary tasks and configurations (including the plugin metadata) to compile your script into a plugin that can be applied in other builds.

Here is a simple build script for the `greeting-plugin` project using the Gradle Plugin Development Plugin:

build.gradle.kts

```
plugins {  
    `java-gradle-plugin`  
}  
  
gradlePlugin {
```

```
plugins {  
    create("simplePlugin") {  
        id = "org.example.greeting"  
        implementationClass = "org.example.GreetingPlugin"  
    }  
}
```

build.gradle

```
plugins {  
    id 'java-gradle-plugin'  
}  
  
gradlePlugin {  
    plugins {  
        simplePlugin {  
            id = 'org.example.greeting'  
            implementationClass = 'org.example.GreetingPlugin'  
        }  
    }  
}
```

For more on publishing plugins, see [Publishing Plugins](#).

Project vs Settings vs Init plugins

In the example used through this section, the plugin accepts the [Project](#) type as a type parameter. Alternatively, the plugin can accept a parameter of type [Settings](#) to be applied in a settings script, or a parameter of type [Gradle](#) to be applied in an initialization script.

The difference between these types of plugins lies in the scope of their application:

Project Plugin

A project plugin is a plugin that is applied to a specific project in a build. It can customize the build logic, add tasks, and configure the project-specific settings.

Settings Plugin

A settings plugin is a plugin that is applied in the `settings.gradle` or `settings.gradle.kts` file. It can configure settings that apply to the entire build, such as defining which projects are included in the build, configuring build script repositories, and applying common configurations to all projects.

Init Plugin

An init plugin is a plugin that is applied in the `init.gradle` or `init.gradle.kts` file. It can

configure settings that apply globally to all Gradle builds on a machine, such as configuring the Gradle version, setting up default repositories, or applying common plugins to all builds.

Understanding Implementation Options for Plugins

The choice between script, precompiled script, or binary plugins depends on your specific requirements and preferences.

Script Plugins are simple and easy to write. They are written in Kotlin DSL or Groovy DSL. They are suitable for small, one-off tasks or for quick experimentation. However, they can become hard to maintain as the build script grows in size and complexity.

Precompiled Script Plugins are Kotlin or Groovy DSL scripts compiled into Java class files packaged in a library. They offer better performance and maintainability compared to script plugins, and they can be reused across different projects. You can also write them in Groovy DSL but that is not recommended.

Binary Plugins are full-fledged plugins written in Java, Groovy, or Kotlin, compiled into JAR files, and published to a repository. They offer the best performance, maintainability, and reusability. They are suitable for complex build logic that needs to be shared across projects, builds, and teams. You can also write them in Scala or Groovy but that is not recommended.

Here is a breakdown of all options for implementing Gradle plugins:

#	Using:	Type:	The Plugin is:	Recommended?
1	Kotlin DSL	Script plugin	in a <code>.gradle.kts</code> file as an abstract class that implements the <code>apply(Project project)</code> method of the <code>Plugin<Project></code> interface.	No ^[1]
2	Groovy DSL	Script plugin	in a <code>.gradle</code> file as an abstract class that implements the <code>apply(Project project)</code> method of the <code>Plugin<Project></code> interface.	No ^[1]
3	Kotlin DSL	Pre-compiled script plugin	a <code>.gradle.kts</code> file.	Yes
4	Groovy DSL	Pre-compiled script plugin	a <code>.gradle</code> file.	Ok ^[2]

#	Using:	Type:	The Plugin is:	Recommended?
5	Java	Binary plugin	an abstract class that implements the <code>apply(Project project)</code> method of the <code>Plugin<Project></code> interface in Java.	Yes
6	Kotlin / Kotlin DSL	Binary plugin	an abstract class that implements the <code>apply(Project project)</code> method of the <code>Plugin<Project></code> interface in Kotlin and/or Kotlin DSL.	Yes
7	Groovy / Groovy DSL	Binary plugin	an abstract class that implements the <code>apply(Project project)</code> method of the <code>Plugin<Project></code> interface in Groovy and/or Groovy DSL.	Ok ^[2]
8	Scala	Binary plugin	an abstract class that implements the <code>apply(Project project)</code> method of the <code>Plugin<Project></code> interface in Scala.	No ^[2]

If you suspect issues with your plugin code, try creating a [Build Scan](#) to identify bottlenecks. The [Gradle profiler](#) can help automate Build Scan generation and gather more low-level information.

Implementing Pre-compiled Script Plugins

A **precompiled script plugin** is typically a Kotlin script that has been compiled and distributed as Java class files packaged in a library. These scripts are intended to be consumed as binary Gradle plugins and are recommended for use as convention plugins.

A **convention plugin** is a plugin that normally configures existing core and community plugins with your own conventions (i.e. default values) such as setting the Java version by using `java.toolchain.languageVersion = JavaLanguageVersion.of(17)`. Convention plugins are also used to enforce project standards and help streamline the build process. They can apply and configure

plugins, create new tasks and extensions, set dependencies, and much more.

Setting the plugin ID

The plugin ID for a precompiled script is derived from its file name and optional package declaration.

For example, a script named `code-quality.gradle(.kts)` located in `src/main/groovy` (or `src/main/kotlin`) without a package declaration would be exposed as the `code-quality` plugin:

buildSrc/build.gradle.kts

```
plugins {  
    'kotlin-dsl'  
}
```

app/build.gradle.kts

```
plugins {  
    id("code-quality")  
}
```

buildSrc/build.gradle

```
plugins {  
    id 'groovy-gradle-plugin'  
}
```

app/build.gradle

```
plugins {  
    id 'code-quality'  
}
```

On the other hand, a script named `code-quality.gradle.kts` located in `src/main/kotlin/my` with the package declaration `my` would be exposed as the `my.code-quality` plugin:

buildSrc/build.gradle.kts

```
plugins {  
    'kotlin-dsl'  
}
```

app/build.gradle.kts

```
plugins {  
    id("my.code-quality")  
}
```

IMPORTANT

Groovy pre-compiled script plugins cannot have packages.

Making a plugin configurable using extensions

Extension objects are commonly used in plugins to expose configuration options and additional functionality to build scripts.

When you apply a plugin that defines an extension, you can access the extension object and configure its properties or call its methods to customize the behavior of the plugin or tasks provided by the plugin.

A [Project](#) has an associated [ExtensionContainer](#) object that contains all the settings and properties for the plugins that have been applied to the project. You can provide configuration for your plugin by adding an extension object to this container.

Let's update our `greetings` example:

buildSrc/src/main/kotlin/greetings.gradle.kts

```
// Create extension object  
interface GreetingPluginExtension {  
    val message: Property<String>  
}  
  
// Add the 'greeting' extension object to project  
val extension =  
project.extensions.create<GreetingPluginExtension>("greeting")
```

buildSrc/src/main/groovy/greetings.gradle

```
// Create extension object  
interface GreetingPluginExtension {  
    Property<String> getMessage()  
}  
  
// Add the 'greeting' extension object to project  
def extension = project.extensions.create("greeting",  
GreetingPluginExtension)
```

You can set the value of the `message` property directly with `extension.message.set("Hi from Gradle,")`.

However, the `GreetingPluginExtension` object becomes available as a project property with the same name as the extension object. You can now access `message` like so:

buildSrc/src/main/kotlin/greetings.gradle.kts

```
// Where the<GreetingPluginExtension>() is equivalent to
project.extensions.getByType(GreetingPluginExtension::class.java)
the<GreetingPluginExtension>().message.set("Hi from Gradle")
```

buildSrc/src/main/groovy/greetings.gradle

```
extensions.findByType(GreetingPluginExtension).message.set("Hi from Gradle")
```

If you apply the `greetings` plugin, you can set the convention in your build script:

app/build.gradle.kts

```
plugins {
    application
    id("greetings")
}

greeting {
    message = "Hello from Gradle"
}
```

app/build.gradle

```
plugins {
    id 'application'
    id 'greetings'
}

configure(greeting) {
    message = "Hello from Gradle"
}
```

Adding default configuration as conventions

In plugins, you can define default values, also known as **conventions**, using the **project** object.

Convention properties are properties that are initialized with default values but can be overridden:

buildSrc/src/main/kotlin/greetings.gradle.kts

```
// Create extension object
interface GreetingPluginExtension {
    val message: Property<String>
}

// Add the 'greeting' extension object to project
val extension =
    project.extensions.create<GreetingPluginExtension>("greeting")

// Set a default value for 'message'
extension.message.convention("Hello from Gradle")
```

buildSrc/src/main/groovy/greetings.gradle

```
// Create extension object
interface GreetingPluginExtension {
    Property<String> getMessage()
}

// Add the 'greeting' extension object to project
def extension = project.extensions.create("greeting",
    GreetingPluginExtension)

// Set a default value for 'message'
extension.message.convention("Hello from Gradle")
```

`extension.message.convention(...)` sets a convention for the `message` property of the extension. This convention specifies that the value of `message` should default to "Hello from Gradle".

If the `message` property is not explicitly set, its value will be automatically set to "Hello from Gradle".

Mapping extension properties to task properties

Using an extension and mapping it to a custom task's input/output properties is common in plugins.

In this example, the `message` property of the `GreetingPluginExtension` is mapped to the `message` property of the `GreetingTask` as an input:

buildSrc/src/main/kotlin/greetings.gradle.kts

```
// Create extension object
interface GreetingPluginExtension {
    val message: Property<String>
}

// Add the 'greeting' extension object to project
val extension =
    project.extensions.create<GreetingPluginExtension>("greeting")

// Set a default value for 'message'
extension.message.convention("Hello from Gradle")

// Create a greeting task
abstract class GreetingTask : DefaultTask() {
    @Input
    val message = project.objects.property<String>()

    @TaskAction
    fun greet() {
        println("Message: ${message.get()}")
    }
}

// Register the task and set the convention
tasks.register<GreetingTask>("hello") {
    message.convention(extension.message)
}
```

buildSrc/src/main/groovy/greetings.gradle

```
// Create extension object
interface GreetingPluginExtension {
    Property<String> getMessage()
}

// Add the 'greeting' extension object to project
def extension = project.extensions.create("greeting",
    GreetingPluginExtension)

// Set a default value for 'message'
extension.message.convention("Hello from Gradle")

// Create a greeting task
abstract class GreetingTask extends DefaultTask {
    @Input
```

```

abstract Property<String> getMessage()

@TaskAction
void greet() {
    println("Message: ${message.get()}")
}

// Register the task and set the convention
tasks.register("hello", GreetingTask) {
    message.convention(extension.message)
}

```

```

$ gradle -q hello
Message: Hello from Gradle

```

This means that changes to the extension's `message` property will trigger the task to be considered out-of-date, ensuring that the task is re-executed with the new message.

You can find out more about types that you can use in task implementations and extensions in [Lazy Configuration](#).

Applying external plugins

In order to apply an external plugin in a precompiled script plugin, it has to be added to the plugin project's implementation classpath in the plugin's build file:

buildSrc/build.gradle.kts

```

plugins {
    `kotlin-dsl`
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("com.bmuschko:gradle-docker-plugin:6.4.0")
}

```

buildSrc/build.gradle

```
plugins {  
    id 'groovy-gradle-plugin'  
}  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    implementation 'com.bmuschko:gradle-docker-plugin:6.4.0'  
}
```

It can then be applied in the precompiled script plugin:

buildSrc/src/main/kotlin/my-plugin.gradle.kts

```
plugins {  
    id("com.bmuschko.docker-remote-api")  
}
```

buildSrc/src/main/groovy/my-plugin.gradle

```
plugins {  
    id 'com.bmuschko.docker-remote-api'  
}
```

The plugin version in this case is defined in the dependency declaration.

Implementing Binary Plugins

Binary plugins refer to plugins that are compiled and distributed as JAR files. These plugins are usually written in Java or Kotlin and provide custom functionality or tasks to a Gradle build.

Using the Plugin Development plugin

The [Gradle Plugin Development plugin](#) can be used to assist in developing Gradle plugins.

This plugin will automatically apply the [Java Plugin](#), add the `gradleApi()` dependency to the `api` configuration, generate the required plugin descriptors in the resulting JAR file, and configure the

[Plugin Marker Artifact](#) to be used when publishing.

To apply and configure the plugin, add the following code to your build file:

build.gradle.kts

```
plugins {
    `java-gradle-plugin`
}

gradlePlugin {
    plugins {
        create("simplePlugin") {
            id = "org.example.greeting"
            implementationClass = "org.example.GreetingPlugin"
        }
    }
}
```

build.gradle

```
plugins {
    id 'java-gradle-plugin'
}

gradlePlugin {
    plugins {
        simplePlugin {
            id = 'org.example.greeting'
            implementationClass = 'org.example.GreetingPlugin'
        }
    }
}
```

Writing and using [custom task types](#) is recommended when developing plugins as it automatically benefits from [incremental builds](#). As an added benefit of applying the plugin to your project, the task `validatePlugins` automatically checks for an existing input/output annotation for every public property defined in a custom task type implementation.

Creating a plugin ID

Plugin IDs are meant to be globally unique, similar to Java package names (i.e., a reverse domain name). This format helps prevent naming collisions and allows grouping plugins with similar ownership.

An explicit plugin identifier simplifies applying the plugin to a project. Your plugin ID should combine components that reflect the namespace (a reasonable pointer to you or your organization) and the name of the plugin it provides. For example, if your Github account is named `foo` and your plugin is named `bar`, a suitable plugin ID might be `com.github.foo.bar`. Similarly, if the plugin was developed at the `baz` organization, the plugin ID might be `org.baz.bar`.

Plugin IDs should adhere to the following guidelines:

- May contain any alphanumeric character, '.', and '-'.
- Must contain at least one '.' character separating the namespace from the plugin's name.
- Conventionally use a lowercase reverse domain name convention for the namespace.
- Conventionally use only lowercase characters in the name.
- `org.gradle`, `com.gradle`, and `com.gradleware` namespaces may not be used.
- Cannot start or end with a '.' character.
- Cannot contain consecutive '.' characters (i.e., '..').

A namespace that identifies ownership and a name is sufficient for a plugin ID.

When bundling multiple plugins in a single JAR artifact, adhering to the same naming conventions is recommended. This practice helps logically group related plugins.

There is no limit to the number of plugins that can be defined and registered (by different identifiers) within a single project.

The identifiers for plugins written as a class should be defined in the project's build script containing the plugin classes. For this, the `java-gradle-plugin` needs to be applied:

buildSrc/build.gradle.kts

```
plugins {
    id("java-gradle-plugin")
}

gradlePlugin {
    plugins {
        create("androidApplicationPlugin") {
            id = "com.android.application"
            implementationClass = "com.android.AndroidApplicationPlugin"
        }
        create("androidLibraryPlugin") {
            id = "com.android.library"
            implementationClass = "com.android.AndroidLibraryPlugin"
        }
    }
}
```

buildSrc/build.gradle

```
plugins {
    id 'java-gradle-plugin'
}

gradlePlugin {
    plugins {
        androidApplicationPlugin {
            id = 'com.android.application'
            implementationClass = 'com.android.AndroidApplicationPlugin'
        }
        androidLibraryPlugin {
            id = 'com.android.library'
            implementationClass = 'com.android.AndroidLibraryPlugin'
        }
    }
}
```

Working with files

When developing plugins, it's a good idea to be flexible when accepting input configuration for file locations.

It is recommended to use Gradle's [managed properties](#) and `project.layout` to select file or directory locations. This will enable lazy configuration so that the actual location will only be resolved when the file is needed and can be reconfigured at any time during build configuration.

This Gradle build file defines a task `GreetingToFileTask` that writes a greeting to a file. It also registers two tasks: `greet`, which creates the file with the greeting, and `sayGreeting`, which prints the file's contents. The `greetingFile` property is used to specify the file path for the greeting:

build.gradle.kts

```
abstract class GreetingToFileTask : DefaultTask() {

    @get:OutputFile
    abstract val destination: RegularFileProperty

    @TaskAction
    fun greet() {
        val file = destination.get().asFile
        file.parentFile.mkdirs()
        file.writeText("Hello!")
    }
}
```

```

val greetingFile = objects.fileProperty()

tasks.register<GreetingToFileTask>("greet") {
    destination = greetingFile
}

tasks.register("sayGreeting") {
    dependsOn("greet")
    val greetingFile = greetingFile
    doLast {
        val file = greetingFile.get().asFile
        println("${file.readText()} (file: ${file.name})")
    }
}

greetingFile = layout.buildDirectory.file("hello.txt")

```

build.gradle

```

abstract class GreetingToFileTask extends DefaultTask {

    @OutputFile
    abstract RegularFileProperty getDestination()

    @TaskAction
    def greet() {
        def file = getDestination().get().asFile
        file.parentFile.mkdirs()
        file.write 'Hello!'
    }
}

def greetingFile = objects.fileProperty()

tasks.register('greet', GreetingToFileTask) {
    destination = greetingFile
}

tasks.register('sayGreeting') {
    dependsOn greet
    doLast {
        def file = greetingFile.get().asFile
        println "${file.text} (file: ${file.name})"
    }
}

greetingFile = layout.buildDirectory.file('hello.txt')

```

```
$ gradle -q sayGreeting
Hello! (file: hello.txt)
```

In this example, we configure the `greet` task `destination` property as a closure/provider, which is evaluated with the `Project.file(java.lang.Object)` method to turn the return value of the closure/provider into a `File` object at the last minute. Note that we specify the `greetingFile` property value *after* the task configuration. This lazy evaluation is a key benefit of accepting any value when setting a file property and then resolving that value when reading the property.

You can learn more about working with files lazily in [Working with Files](#).

Making a plugin configurable using extensions

Most plugins offer configuration options for build scripts and other plugins to customize how the plugin works. Plugins do this using **extension objects**.

A `Project` has an associated `ExtensionContainer` object that contains all the settings and properties for the plugins that have been applied to the project. You can provide configuration for your plugin by adding an extension object to this container.

An extension object is simply an object with Java Bean properties representing the configuration.

Let's add a `greeting` extension object to the project, which allows you to configure the greeting:

build.gradle.kts

```
interface GreetingPluginExtension {
    val message: Property<String>
}

class GreetingPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        // Add the 'greeting' extension object
        val extension =
            project.extensions.create<GreetingPluginExtension>("greeting")
        // Add a task that uses configuration from the extension object
        project.task("hello") {
            doLast {
                println(extension.message.get())
            }
        }
    }
}

apply<GreetingPlugin>()

// Configure the extension
```

```
the<GreetingPluginExtension>().message = "Hi from Gradle"
```

build.gradle

```
interface GreetingPluginExtension {
    Property<String> getMessage()
}

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        // Add the 'greeting' extension object
        def extension = project.extensions.create('greeting',
GreetingPluginExtension)
        // Add a task that uses configuration from the extension object
        project.task('hello') {
            doLast {
                println extension.message.get()
            }
        }
    }
}

apply plugin: GreetingPlugin

// Configure the extension
greeting.message = 'Hi from Gradle'
```

```
$ gradle -q hello
Hi from Gradle
```

In this example, `GreetingPluginExtension` is an object with a property called `message`. The extension object is added to the project with the name `greeting`. This object becomes available as a project property with the same name as the extension object. `the<GreetingPluginExtension>()` is equivalent to `project.extensions.getByType(GreetingPluginExtension::class.java)`.

Often, you have several related properties you need to specify on a single plugin. Gradle adds a configuration block for each extension object, so you can group settings:

build.gradle.kts

```
interface GreetingPluginExtension {
    val message: Property<String>
    val greeter: Property<String>
}
```

```

class GreetingPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        val extension =
        project.extensions.create<GreetingPluginExtension>("greeting")
        project.task("hello") {
            doLast {
                println("${extension.message.get()} from
${extension.greeter.get()}")
            }
        }
    }
}

apply<GreetingPlugin>()

// Configure the extension using a DSL block
configure<GreetingPluginExtension> {
    message = "Hi"
    greeter = "Gradle"
}

```

build.gradle

```

interface GreetingPluginExtension {
    Property<String> getMessage()
    Property<String> getGreeter()
}

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        def extension = project.extensions.create('greeting',
GreetingPluginExtension)
        project.task('hello') {
            doLast {
                println "${extension.message.get()} from ${extension.greeter
.get()}"
            }
        }
    }
}

apply plugin: GreetingPlugin

// Configure the extension using a DSL block
greeting {
    message = 'Hi'
    greeter = 'Gradle'
}

```

```
}
```

```
$ gradle -q hello  
Hi from Gradle
```

In this example, several settings can be grouped within the `configure<GreetingPluginExtension>` block. The `configure` function is used to configure an extension object. It provides a convenient way to set properties or apply configurations to these objects. The type used in the build script's `configure` function (`GreetingPluginExtension`) must match the extension type. Then, when the block is executed, the receiver of the block is the extension.

In this example, several settings can be grouped within the `greeting` closure. The name of the closure block in the build script (`greeting`) must match the extension object name. Then, when the closure is executed, the fields on the extension object will be mapped to the variables within the closure based on the standard Groovy closure delegate feature.

Declaring a DSL configuration container

Using an extension object *extends* the Gradle DSL to add a project property and DSL block for the plugin. Because an extension object is a regular object, you can provide your own DSL nested inside the plugin block by adding properties and methods to the extension object.

Let's consider the following build script for illustration purposes.

build.gradle.kts

```
plugins {  
    id("org.myorg.server-env")  
}  
  
environments {  
    create("dev") {  
        url = "http://localhost:8080"  
    }  
  
    create("staging") {  
        url = "http://staging.enterprise.com"  
    }  
  
    create("production") {  
        url = "http://prod.enterprise.com"  
    }  
}
```

build.gradle

```
plugins {  
    id 'org.myorg.server-env'  
}  
  
environments {  
    dev {  
        url = 'http://localhost:8080'  
    }  
  
    staging {  
        url = 'http://staging.enterprise.com'  
    }  
  
    production {  
        url = 'http://prod.enterprise.com'  
    }  
}
```

The DSL exposed by the plugin exposes a container for defining a set of environments. Each environment the user configures has an arbitrary but declarative name and is represented with its own DSL configuration block. The example above instantiates a development, staging, and production environment, including its respective URL.

Each environment must have a data representation in code to capture the values. The name of an environment is immutable and can be passed in as a constructor parameter. Currently, the only other parameter the data object stores is a URL.

The following `ServerEnvironment` object fulfills those requirements:

ServerEnvironment.java

```
abstract public class ServerEnvironment {  
    private final String name;  
  
    @javax.inject.Inject  
    public ServerEnvironment(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    abstract public Property<String> getUrl();  
}
```

Gradle exposes the factory method `ObjectFactory.domainObjectContainer(Class, NamedDomainObjectFactory)` to create a container of data objects. The parameter the method takes is the class representing the data. The created instance of type `NamedDomainObjectContainer` can be exposed to the end user by adding it to the extension container with a specific name.

It's common for a plugin to post-process the captured values within the plugin implementation, e.g., to configure tasks:

ServerEnvironmentPlugin.java

```
public class ServerEnvironmentPlugin implements Plugin<Project> {
    @Override
    public void apply(final Project project) {
        ObjectFactory objects = project.getObjects();

        NamedDomainObjectContainer<ServerEnvironment> serverEnvironmentContainer =
            objects.domainObjectContainer(ServerEnvironment.class, name -> objects
                .newInstance(ServerEnvironment.class, name));
        project.getExtensions().add("environments", serverEnvironmentContainer);

        serverEnvironmentContainer.all(serverEnvironment -> {
            String env = serverEnvironment.getName();
            String capitalizedServerEnv = env.substring(0, 1).toUpperCase() + env
                .substring(1);
            String taskName = "deployTo" + capitalizedServerEnv;
            project.getTasks().register(taskName, Deploy.class, task -> task.getUrl()
                .set(serverEnvironment.getUrl()));
        });
    }
}
```

In the example above, a deployment task is created dynamically for every user-configured environment.

You can find out more about implementing project extensions in [Developing Custom Gradle Types](#).

Modeling DSL-like APIs

DSLs exposed by plugins should be readable and easy to understand.

For example, let's consider the following extension provided by a plugin. In its current form, it offers a "flat" list of properties for configuring the creation of a website:

build-flat.gradle.kts

```
plugins {
    id("org.myorg.site")
}
```

```

site {
    outputDir = layout.buildDirectory.file("mysite")
    websiteUrl = "https://gradle.org"
    vcsUrl = "https://github.com/gradle/gradle-site-plugin"
}

```

build-flat.gradle

```

plugins {
    id 'org.myorg.site'
}

site {
    outputDir = layout.buildDirectory.file("mysite")
    websiteUrl = 'https://gradle.org'
    vcsUrl = 'https://github.com/gradle/gradle-site-plugin'
}

```

As the number of exposed properties grows, you should introduce a nested, more expressive structure.

The following code snippet adds a new configuration block named `siteInfo` as part of the extension. This provides a stronger indication of what those properties mean:

build.gradle.kts

```

plugins {
    id("org.myorg.site")
}

site {
    outputDir = layout.buildDirectory.file("mysite")

    siteInfo {
        websiteUrl = "https://gradle.org"
        vcsUrl = "https://github.com/gradle/gradle-site-plugin"
    }
}

```

build.gradle

```

plugins {
    id 'org.myorg.site'
}

```

```

}

site {
    outputDir = layout.buildDirectory.file("mysite")

    siteInfo {
        websiteUrl = 'https://gradle.org'
        vcsUrl = 'https://github.com/gradle/gradle-site-plugin'
    }
}

```

Implementing the backing objects for such an extension is simple. First, introduce a new data object for managing the properties `websiteUrl` and `vcsUrl`:

SiteInfo.java

```

abstract public class SiteInfo {

    abstract public Property<String> getWebsiteUrl();

    abstract public Property<String> getVcsUrl();
}

```

In the extension, create an instance of the `siteInfo` class and a method to delegate the captured values to the data instance.

To configure underlying data objects, define a parameter of type `Action`.

The following example demonstrates the use of `Action` in an extension definition:

SiteExtension.java

```

abstract public class SiteExtension {

    abstract public RegularFileProperty getOutputDir();

    @Nested
    abstract public SiteInfo getSiteInfo();

    public void siteInfo(Action<? super SiteInfo> action) {
        action.execute(getSiteInfo());
    }
}

```

Mapping extension properties to task properties

Plugins commonly use an extension to capture user input from the build script and map it to a

custom task's input/output properties. The build script author interacts with the extension's DSL, while the plugin implementation handles the underlying logic:

app/build.gradle.kts

```
// Extension class to capture user input
class MyExtension {
    @Input
    var inputParameter: String? = null
}

// Custom task that uses the input from the extension
class MyCustomTask : org.gradle.api.DefaultTask() {
    @Input
    var inputParameter: String? = null

    @TaskAction
    fun executeTask() {
        println("Input parameter: $inputParameter")
    }
}

// Plugin class that configures the extension and task
class MyPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        // Create and configure the extension
        val extension = project.extensions.create("myExtension",
MyExtension::class.java)
        // Create and configure the custom task
        project.tasks.register("myTask", MyCustomTask::class.java) {
            group = "custom"
            inputParameter = extension.inputParameter
        }
    }
}
```

app/build.gradle

```
// Extension class to capture user input
class MyExtension {
    @Input
    String inputParameter = null
}

// Custom task that uses the input from the extension
class MyCustomTask extends DefaultTask {
    @Input
```

```

String inputParameter = null

@TaskAction
def executeTask() {
    println("Input parameter: $inputParameter")
}

// Plugin class that configures the extension and task
class MyPlugin implements Plugin<Project> {
    void apply(Project project) {
        // Create and configure the extension
        def extension = project.extensions.create("myExtension", MyExtension)
        // Create and configure the custom task
        project.tasks.register("myTask", MyCustomTask) {
            group = "custom"
            inputParameter = extension.inputParameter
        }
    }
}

```

In this example, the `MyExtension` class defines an `inputParameter` property that can be set in the build script. The `MyPlugin` class configures this extension and uses its `inputParameter` value to configure the `MyCustomTask` task. The `MyCustomTask` task then uses this input parameter in its logic.

You can learn more about types you can use in task implementations and extensions in [Lazy Configuration](#).

Adding default configuration with conventions

Plugins should provide sensible defaults and standards in a specific context, reducing the number of decisions users need to make. Using the `project` object, you can define default values. These are known as **conventions**.

Conventions are properties that are initialized with default values and can be overridden by the user in their build script. For example:

build.gradle.kts

```

interface GreetingPluginExtension {
    val message: Property<String>
}

class GreetingPlugin : Plugin<Project> {
    override fun apply(project: Project) {
        // Add the 'greeting' extension object
        val extension =

```

```

project.extensions.create<GreetingPluginExtension>("greeting")
    extension.message.convention("Hello from GreetingPlugin")
    // Add a task that uses configuration from the extension object
    project.task("hello") {
        doLast {
            println(extension.message.get())
        }
    }
}

apply<GreetingPlugin>()

```

build.gradle

```

interface GreetingPluginExtension {
    Property<String> getMessage()
}

class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        // Add the 'greeting' extension object
        def extension = project.extensions.create('greeting',
GreetingPluginExtension)
        extension.message.convention('Hello from GreetingPlugin')
        // Add a task that uses configuration from the extension object
        project.task('hello') {
            doLast {
                println extension.message.get()
            }
        }
    }
}

apply plugin: GreetingPlugin

```

```

$ gradle -q hello
Hello from GreetingPlugin

```

In this example, `GreetingPluginExtension` is a class that represents the convention. The message property is the convention property with a default value of 'Hello from GreetingPlugin'.

Users can override this value in their build script:

build.gradle.kts

```
GreetingPluginExtension {  
    message = "Custom message"  
}
```

build.gradle

```
GreetingPluginExtension {  
    message = 'Custom message'  
}
```

```
$ gradle -q hello  
Custom message
```

Separating capabilities from conventions

Separating capabilities from conventions in plugins allows users to choose which tasks and conventions to apply.

For example, the Java Base plugin provides un-opinionated (i.e., generic) functionality like `SourceSets`, while the Java plugin adds tasks and conventions familiar to Java developers like `classes`, `jar` or `javadoc`.

When designing your own plugins, consider developing two plugins — one for capabilities and another for conventions — to offer flexibility to users.

In the example below, `MyPlugin` contains conventions, and `MyBasePlugin` defines capabilities. Then, `MyPlugin` applies `MyBasePlugin`, this is called *plugin composition*. To apply a plugin from another one:

MyBasePlugin.java

```
import org.gradle.api.Plugin;  
import org.gradle.api.Project;  
  
public class MyBasePlugin implements Plugin<Project> {  
    public void apply(Project project) {  
        // define capabilities  
    }  
}
```

MyPlugin.java

```
import org.gradle.api.Plugin;
import org.gradle.api.Project;

public class MyPlugin implements Plugin<Project> {
    public void apply(Project project) {
        project.getPluginManager().apply(MyBasePlugin.class);

        // define conventions
    }
}
```

Reacting to plugins

A common pattern in Gradle plugin implementations is configuring the runtime behavior of existing plugins and tasks in a build.

For example, a plugin could assume that it is applied to a Java-based project and automatically reconfigure the standard source directory:

InhouseStrongOpinionConventionJavaPlugin.java

```
public class InhouseStrongOpinionConventionJavaPlugin implements Plugin<Project> {
    public void apply(Project project) {
        // Careful! Eagerly applying plugins has downsides, and is not always
        recommended.
        project.getPluginManager().apply(JavaPlugin.class);
        SourceSetContainer sourceSets = project.getExtensions().getByType
        (SourceSetContainer.class);
        SourceSet main = sourceSets.getByName(SourceSet.MAIN_SOURCE_SET_NAME);
        main.getJava().setSrcDirs(Arrays.asList("src"));
    }
}
```

The drawback to this approach is that it automatically forces the project to apply the Java plugin, imposing a strong opinion on it (i.e., reducing flexibility and generality). In practice, the project applying the plugin might not even deal with Java code.

Instead of automatically applying the Java plugin, the plugin could react to the fact that the consuming project applies the Java plugin. Only if that is the case, then a certain configuration is applied:

InhouseConventionJavaPlugin.java

```
public class InhouseConventionJavaPlugin implements Plugin<Project> {
    public void apply(Project project) {
        project.getPluginManager().withPlugin("java", javaPlugin -> {
            SourceSetContainer sourceSets = project.getExtensions().getByType
            (SourceSetContainer.class);
```

```

        SourceSet main = sourceSets.getByName(SourceSet.MAIN_SOURCE_SET_NAME);
        main.getJava().setSrcDirs(Arrays.asList("src"));
    });
}

```

Reacting to plugins is preferred over applying plugins if there is no good reason to assume that the consuming project has the expected setup.

The same concept applies to task types:

InhouseConventionWarPlugin.java

```

public class InhouseConventionWarPlugin implements Plugin<Project> {
    public void apply(Project project) {
        project.getTasks().withType(War.class).configureEach(war ->
            war.setWebXml(project.file("src/someWeb.xml")));
    }
}

```

Reacting to build features

Plugins can access the status of build features in the build. The [Build Features API](#) allows checking whether the user requested a particular Gradle feature and if it is active in the current build. An example of a build feature is the [configuration cache](#).

There are two main use cases:

- Using the status of build features in reports or statistics.
- Incrementally adopting experimental Gradle features by disabling incompatible plugin functionality.

Below is an example of a plugin that utilizes both of the cases.

Reacting to build features

```

public abstract class MyPlugin implements Plugin<Project> {

    @Inject
    protected abstract BuildFeatures getBuildFeatures(); ①

    @Override
    public void apply(Project p) {
        BuildFeatures buildFeatures = getBuildFeatures();

        Boolean configCacheRequested = buildFeatures.getConfigurationCache()
            .getRequested() ②
            .getOrNull(); // could be null if user did not opt in nor opt out
        String configCacheUsage = describeFeatureUsage(configCacheRequested);
    }
}

```

```

MyReport myReport = new MyReport();
myReport.setConfigurationCacheUsage(configCacheUsage);

boolean isolatedProjectsActive = buildFeatures.getIsolatedProjects().
getActive() ③
    .get(); // the active state is always defined
if (!isolatedProjectsActive) {
    myOptionalPluginLogicIncompatibleWithIsolatedProjects();
}

private String describeFeatureUsage(Boolean requested) {
    return requested == null ? "no preference" : requested ? "opt-in" : "opt-out";
}

private void myOptionalPluginLogicIncompatibleWithIsolatedProjects() {
}
}

```

- ① The **BuildFeatures** service can be injected into plugins, tasks, and other managed types.
- ② Accessing the **requested** status of a feature for reporting.
- ③ Using the **active** status of a feature to disable incompatible functionality.

Build feature properties

A **BuildFeature** status properties are represented with **Provider<Boolean>** types.

The **BuildFeature.getRequest()** status of a build feature determines if the user requested to enable or disable the feature.

When the **requested** provider value is:

- **true** — the user opted in for using the feature
- **false** — the user opted out from using the feature
- **undefined** — the user neither opted in nor opted out from using the feature

The **BuildFeature.getActive()** status of a build feature is always defined. It represents the effective state of the feature in the build.

When the **active** provider value is:

- **true** — the feature *may* affect the build behavior in a way specific to the feature
- **false** — the feature will not affect the build behavior

Note that the **active** status does not depend on the **requested** status. Even if the user requests a feature, it may still not be active due to other build options being used in the build. Gradle can also activate a feature by default, even if the user did not specify a preference.

Using a custom `dependencies` block

A plugin can provide dependency declarations in custom blocks that allow users to declare dependencies in a type-safe and context-aware way.

For instance, instead of users needing to know and use the underlying `Configuration` name to add dependencies, a custom `dependencies` block lets the plugin pick a meaningful name that can be used consistently.

Adding a custom `dependencies` block

To add a custom `dependencies` block, you need to create a new `type` that will represent the set of dependency scopes available to users. That new type needs to be accessible from a part of your plugin (from a domain object or extension). Finally, the dependency scopes need to be wired back to underlying `Configuration` objects that will be used during dependency resolution.

See `JvmComponentDependencies` and `JvmTestSuite` for an example of how this is used in a Gradle core plugin.

1. Create an interface that extends `Dependencies`

NOTE

You can also extend `GradleDependencies` to get access to Gradle-provided dependencies like `gradleApi()`.

ExampleDependencies.java

```
/**
 * Custom dependencies block for the example plugin.
 */
public interface ExampleDependencies extends Dependencies {
```

2. Add accessors for dependency scopes

For each dependency scope your plugin wants to support, add a getter method that returns a `DependencyCollector`.

ExampleDependencies.java

```
/**
 * Dependency scope called "implementation"
 */
DependencyCollector getImplementation();
```

3. Add accessors for custom `dependencies` block

To make the custom `dependencies` block configurable, the plugin needs to add a `getDependencies`

method that returns the new type from above and a configurable block method named `dependencies`.

By convention, the accessors for your custom `dependencies` block should be called `getDependencies()/dependencies(Action)`. This method could be named something else, but users would need to know that a different block can behave like a `dependencies` block.

ExampleExtension.java

```
/**
 * Custom dependencies for this extension.
 */
@Nested
ExampleDependencies getDependencies();

/**
 * Configurable block
 */
default void dependencies(Action<? super ExampleDependencies> action) {
    action.execute(getDependencies());
}
```

4. Wire dependency scope to `Configuration`

Finally, the plugin needs to wire the custom `dependencies` block to some underlying `Configuration` objects. If this is not done, none of the dependencies declared in the custom block will be available to dependency resolution.

ExamplePlugin.java

```
project.getConfigurations().dependencyScope("exampleImplementation", conf
-> {
    conf.fromDependencyCollector(example.getDependencies()
    .getImplementation());
});
```

NOTE

In this example, the name users will use to add dependencies is "implementation", but the underlying `Configuration` is named `exampleImplementation`.

build.gradle.kts

```
example {
    dependencies {
        implementation("junit:junit:4.13")
    }
}
```

```
}
```

build.gradle

```
example {  
    dependencies {  
        implementation("junit:junit:4.13")  
    }  
}
```

Differences between the custom `dependencies` and the top-level `dependencies` blocks

Each dependency scope returns a `DependencyCollector` that provides strongly-typed methods to add and configure dependencies.

There is also a `DependencyFactory` with factory methods to create new dependencies from different notations. Dependencies can be created lazily using these factory methods, as shown below.

A custom `dependencies` block differs from the top-level `dependencies` block in the following ways:

- Dependencies must be declared using a `String`, an instance of `Dependency`, a `FileCollection`, a `Provider` of `Dependency`, or a `ProviderConvertible` of `MinimalExternalModuleDependency`.
- Outside of Gradle build scripts, you must explicitly call a getter for the `DependencyCollector` and `add`.
 - `dependencies.add("implementation", x)` becomes `getImplementation().add(x)`
- You cannot declare dependencies with the `Map` notation from Kotlin and Java. Use multi-argument methods instead in Kotlin and Java.
 - Kotlin: `compileOnly(mapOf("group" to "foo", "name" to "bar"))` becomes `compileOnly(module(group = "foo", name = "bar"))`
 - Java: `compileOnly(Map.of("group", "foo", "name", "bar"))` becomes `getCompileOnly().add(module("foo", "bar", null))`
- You cannot add a dependency with an instance of `Project`. You must turn it into a `ProjectDependency` first.
- You cannot add version catalog bundles directly. Instead, use the `bundle` method on each configuration.
 - Kotlin and Groovy: `implementation(libs.bundles.testing)` becomes `implementation.bundle(libs.bundles.testing)`
- You cannot use providers for non-`Dependency` types directly. Instead, map them to a `Dependency` using the `DependencyFactory`.
 - Kotlin and Groovy: `implementation(myStringProvider)` becomes `implementation(myStringProvider.map { dependencyFactory.create(it) })`
 - Java: `implementation(myStringProvider)` becomes `getImplementation().add(myStringProvider.map(getDependencyFactory()::create))`

- Unlike the top-level `dependencies` block, constraints are not in a separate block.
 - Instead, constraints are added by decorating a dependency with `constraint(...)` like `implementation(constraint("org:foo:1.0"))`.

Keep in mind that the `dependencies` block may not provide access to the same methods as the [top-level dependencies block](#).

NOTE | Plugins should prefer adding dependencies via their own `dependencies` block.

Providing default dependencies

The implementation of a plugin sometimes requires the use of an external dependency.

You might want to automatically download an artifact using Gradle's dependency management mechanism and later use it in the action of a task type declared in the plugin. Ideally, the plugin implementation does not need to ask the user for the coordinates of that dependency - it can simply predefine a sensible default version.

Let's look at an example of a plugin that downloads files containing data for further processing. The plugin implementation declares a custom configuration that allows for [assigning those external dependencies with dependency coordinates](#):

DataProcessingPlugin.java

```
public class DataProcessingPlugin implements Plugin<Project> {
    public void apply(Project project) {
        Configuration dataFiles = project.getConfigurations().create("dataFiles", c ->
    {
        c.setVisible(false);
        c.setCanBeConsumed(false);
        c.setCanBeResolved(true);
        c.setDescription("The data artifacts to be processed for this plugin.");
        c.defaultDependencies(d -> d.add(project.getDependencies().create(
"org.myorg:data:1.4.6"))));
    });

    project.getTasks().withType(DataProcessing.class).configureEach(
        dataProcessing -> dataProcessing.getDataFiles().from(dataFiles));
    }
}
```

DataProcessing.java

```
abstract public class DataProcessing extends DefaultTask {

    @InputFiles
    abstract public ConfigurableFileCollection getDataFiles();

    @TaskAction
```

```
public void process() {  
    System.out.println(getDataFiles().getFiles());  
}  
}
```

This approach is convenient for the end user as there is no need to actively declare a dependency. The plugin already provides all the details about this implementation.

But what if the user wants to redefine the default dependency?

No problem. The plugin also exposes the custom configuration that can be used to assign a different dependency. Effectively, the default dependency is overwritten:

build.gradle.kts

```
plugins {  
    id("org.myorg.data-processing")  
}  
  
dependencies {  
    dataFiles("org.myorg:more-data:2.6")  
}
```

build.gradle

```
plugins {  
    id 'org.myorg.data-processing'  
}  
  
dependencies {  
    dataFiles 'org.myorg:more-data:2.6'  
}
```

You will find that this pattern works well for tasks that require an external dependency when the task's action is executed. You can go further and abstract the version to be used for the external dependency by exposing an extension property (e.g. `toolVersion` in [the JaCoCo plugin](#)).

Minimizing the use of external libraries

Using external libraries in your Gradle projects can bring great convenience, but be aware that they can introduce complex dependency graphs. Gradle's `buildEnvironment` task can help you visualize these dependencies, including those of your plugins. Keep in mind that plugins share the same classloader, so conflicts may arise with different versions of the same library.

To demonstrate let's assume the following build script:

build.gradle.kts

```
plugins {  
    id("org.asciidoctor.jvm.convert") version "4.0.2"  
}
```

build.gradle

```
plugins {  
    id 'org.asciidoctor.jvm.convert' version '4.0.2'  
}
```

The output of the task clearly indicates the classpath of the **classpath** configuration:

```
$ gradle buildEnvironment
```

```
> Task :buildEnvironment
```

```
-----  
Root project 'external-libraries'  
-----
```

```
classpath
```

```
\--- org.asciidoctor.jvm.convert:org.asciidoctor.jvm.convert.gradle.plugin:4.0.2  
    \--- org.asciidoctor:asciidoctor-gradle-jvm:4.0.2  
        +--- org.ysb33r.gradle:grolifant-rawhide:3.0.0  
            |    \--- org.tukaani:xz:1.6  
        +--- org.ysb33r.gradle:grolifant-herd:3.0.0  
            |    +--- org.tukaani:xz:1.6  
            |    +--- org.ysb33r.gradle:grolifant40:3.0.0  
            |        |    +--- org.tukaani:xz:1.6  
            |        |    +--- org.apache.commons:commons-collections4:4.4  
            |        |    +--- org.ysb33r.gradle:grolifant-core:3.0.0  
            |        |        |    +--- org.tukaani:xz:1.6  
            |        |        |    +--- org.apache.commons:commons-collections4:4.4  
            |        |        |    \--- org.ysb33r.gradle:grolifant-rawhide:3.0.0 (*)  
            |        |    \--- org.ysb33r.gradle:grolifant-rawhide:3.0.0 (*)  
        +--- org.ysb33r.gradle:grolifant50:3.0.0  
            |    +--- org.tukaani:xz:1.6  
            |    +--- org.ysb33r.gradle:grolifant40:3.0.0 (*)  
            |    +--- org.ysb33r.gradle:grolifant-core:3.0.0 (*)  
            |    \--- org.ysb33r.gradle:grolifant40-legacy-api:3.0.0
```

```

|      |      +--- org.tukaani:xz:1.6
|      |      +--- org.apache.commons:commons-collections4:4.4
|      |      +--- org.ysb33r.gradle:grolifant-core:3.0.0 (*)
|      |      \--- org.ysb33r.gradle:grolifant40:3.0.0 (*)
+---- org.ysb33r.gradle:grolifant60:3.0.0
|      |      +--- org.tukaani:xz:1.6
|      |      +--- org.ysb33r.gradle:grolifant40:3.0.0 (*)
|      |      +--- org.ysb33r.gradle:grolifant50:3.0.0 (*)
|      |      +--- org.ysb33r.gradle:grolifant-core:3.0.0 (*)
|      |      \--- org.ysb33r.gradle:grolifant-rawhide:3.0.0 (*)
+---- org.ysb33r.gradle:grolifant70:3.0.0
|      |      +--- org.tukaani:xz:1.6
|      |      +--- org.ysb33r.gradle:grolifant40:3.0.0 (*)
|      |      +--- org.ysb33r.gradle:grolifant50:3.0.0 (*)
|      |      +--- org.ysb33r.gradle:grolifant60:3.0.0 (*)
|      |      \--- org.ysb33r.gradle:grolifant-core:3.0.0 (*)
+---- org.ysb33r.gradle:grolifant80:3.0.0
|      |      +--- org.tukaani:xz:1.6
|      |      +--- org.ysb33r.gradle:grolifant40:3.0.0 (*)
|      |      +--- org.ysb33r.gradle:grolifant50:3.0.0 (*)
|      |      +--- org.ysb33r.gradle:grolifant60:3.0.0 (*)
|      |      +--- org.ysb33r.gradle:grolifant70:3.0.0 (*)
|      |      \--- org.ysb33r.gradle:grolifant-core:3.0.0 (*)
+---- org.ysb33r.gradle:grolifant-core:3.0.0 (*)
|      \--- org.ysb33r.gradle:grolifant-rawhide:3.0.0 (*)
+---- org.asciidoctor:asciidoctor-gradle-base:4.0.2
|      \--- org.ysb33r.gradle:grolifant-herd:3.0.0 (*)
\--- org.asciidoctor:asciidoctorj-api:2.5.7

```

(*) - Indicates repeated occurrences of a transitive dependency subtree. Gradle expands transitive dependency subtrees only once per project; repeat occurrences only display the root of the subtree, followed by this annotation.

A web-based, searchable dependency report is available by adding the `--scan` option.

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed

A Gradle plugin does not run in its own, isolated classloader, so you must consider whether you truly need a library or if a simpler solution suffices.

For logic that is executed as part of task execution, use the [Worker API](#) that allows you to isolate libraries.

Providing multiple variants of a plugin

Variants of a plugin refer to different flavors or configurations of the plugin that are tailored to specific needs or use cases. These variants can include different implementations, extensions, or configurations of the base plugin.

The most convenient way to configure additional plugin variants is to use [feature variants](#), a concept available in all Gradle projects that apply one of the Java plugins:

```
dependencies {
    implementation 'com.google.guava:guava:30.1-jre'           // Regular dependency
    featureVariant 'com.google.guava:guava-gwt:30.1-jre'      // Feature variant
    dependency
}
```

In the following example, each plugin variant is developed in isolation. A separate source set is compiled and packaged in a separate jar for each variant.

The following sample demonstrates how to add a variant that is compatible with Gradle 7.0+ while the "main" variant is compatible with older versions:

build.gradle.kts

```
val gradle7 = sourceSets.create("gradle7")

java {
    registerFeature(gradle7.name) {
        usingSourceSet(gradle7)
        capability(project.group.toString(), project.name,
project.version.toString()) ①
    }
}

configurations.configureEach {
    if (isCanBeConsumed && name.startsWith(gradle7.name)) {
        attributes {

attribute(GradlePluginApiVersion.GRADLE_PLUGIN_API_VERSION_ATTRIBUTE, ②
            objects.named("7.0"))

        }
    }
}

tasks.named<Copy>(gradle7.processResourcesTaskName) { ③
    val copyPluginDescriptors = rootSpec.addChild()
    copyPluginDescriptors.into("META-INF/gradle-plugins")
    copyPluginDescriptors.from(tasks.pluginDescriptors)
}

dependencies {
    "gradle7CompileOnly"(gradleApi()) ④
}
```

build.gradle

```
def gradle7 = sourceSets.create('gradle7')

java {
    registerFeature(gradle7.name) {
        usingSourceSet(gradle7)
        capability(project.group.toString(), project.name, project.version
.toString()) ❶
    }
}

configurations.configureEach {
    if (canBeConsumed && name.startsWith(gradle7.name)) {
        attributes {
            attribute(GradlePluginApiVersion
.GRADLE_PLUGIN_API_VERSION_ATTRIBUTE, ❷
                objects.named(GradlePluginApiVersion, '7.0'))
        }
    }
}

tasks.named(gradle7.processResourcesTaskName) { ❸
    def copyPluginDescriptors = rootSpec.addChild()
    copyPluginDescriptors.into('META-INF/gradle-plugins')
    copyPluginDescriptors.from(tasks.pluginDescriptors)
}

dependencies {
    gradle7CompileOnly(gradleApi()) ❹
}
```

NOTE

Only Gradle versions 7 or higher can be explicitly targeted by a variant, as support for this was only added in Gradle 7.

First, we declare a separate *source set* and a *feature variant* for our Gradle 7 plugin variant. Then, we do some specific wiring to turn the feature into a proper Gradle plugin variant:

- ❶ Assign the [implicit capability that corresponds to the components GAV](#) to the variant.
- ❷ Assign the [Gradle API version attribute](#) to all [consumable configurations](#) of our Gradle7 variant. Gradle uses this information to determine which variant to select during plugin resolution.
- ❸ Configure the [processGradle7Resources](#) task to ensure the plugin descriptor file is added to the Gradle7 variant Jar.
- ❹ Add a dependency to the [gradleApi\(\)](#) for our new variant so that the API is visible during compilation time.

Note that there is currently no convenient way to access the API of other Gradle versions as the one you are building the plugin with. Ideally, every variant should be able to declare a dependency on the API of the minimal Gradle version it supports. This will be improved in the future.

The above snippet assumes that all variants of your plugin have the plugin class at the same location. That is, if your plugin class is `org.example.GreetingPlugin`, you need to create a second variant of that class in `src/gradle7/java/org/example`.

Using version-specific variants of multi-variant plugins

Given a dependency on a multi-variant plugin, Gradle will automatically choose its variant that best matches the current Gradle version when it resolves any of:

- plugins specified in the `plugins {}` block;
- `buildscript` classpath dependencies;
- dependencies in the root project of the `build source (buildSrc)` that appear on the compile or runtime classpath;
- dependencies in a project that applies the [Java Gradle Plugin Development plugin](#) or the [Kotlin DSL plugin](#), appearing on the compile or runtime classpath.

The best matching variant is the variant that targets the highest Gradle API version and does not exceed the current build's Gradle version.

In all other cases, a plugin variant that does not specify the supported Gradle API version is preferred if such a variant is present.

In projects that use plugins as dependencies, requesting the variants of plugin dependencies that support a different Gradle version is possible. This allows a multi-variant plugin that depends on other plugins to access their APIs, which are exclusively provided in their version-specific variants.

This snippet makes the `plugin variant gradle7` defined above consume the matching variants of its dependencies on other multi-variant plugins:

build.gradle.kts

```
configurations.configureEach {
    if (isCanBeResolved && name.startsWith(gradle7.name)) {
        attributes {

            attribute(GradlePluginApiVersion.GRADLE_PLUGIN_API_VERSION_ATTRIBUTE,
                objects.named("7.0"))
        }
    }
}
```

build.gradle

```
configurations.configureEach {
    if (canBeResolved && name.startsWith(gradle7.name)) {
        attributes {
            attribute(GradlePluginApiVersion
                .GRADLE_PLUGIN_API_VERSION_ATTRIBUTE,
                objects.named(GradlePluginApiVersion, '7.0'))
        }
    }
}
```

Testing Gradle plugins

Testing plays a crucial role in the development process by ensuring reliable and high-quality software. This principle applies to build code, including Gradle plugins.

The sample project

This section revolves around a sample project called the "URL verifier plugin". This plugin creates a task named `verifyUrl` that checks whether a given URL can be resolved via HTTP GET. The end user can provide the URL via an extension named `verification`.

The following build script assumes that the plugin JAR file has been published to a binary repository. The script demonstrates how to apply the plugin to the project and configure its exposed extension:

build.gradle.kts

```
plugins {
    id("org.myorg.url-verifier") ①
}

verification {
    url = "https://www.google.com/" ②
}
```

build.gradle

```
plugins {
    id 'org.myorg.url-verifier' ①
}
```

```
verification {  
    url = 'https://www.google.com/' ②  
}
```

- ① Applies the plugin to the project
- ② Configures the URL to be verified through the exposed extension

Executing the `verifyUrl` task renders a success message if the HTTP GET call to the configured URL returns with a 200 response code:

```
$ gradle verifyUrl  
  
> Task :verifyUrl  
Successfully resolved URL 'https://www.google.com/'  
  
BUILD SUCCESSFUL in 0s  
5 actionable tasks: 5 executed
```

Before diving into the code, let's first revisit the different types of tests and the tooling that supports implementing them.

The importance of testing

Testing is a crucial part of the software development life cycle, ensuring that software functions correctly and meets quality standards before release. Automated testing allows developers to refactor and improve code with confidence.

The testing pyramid

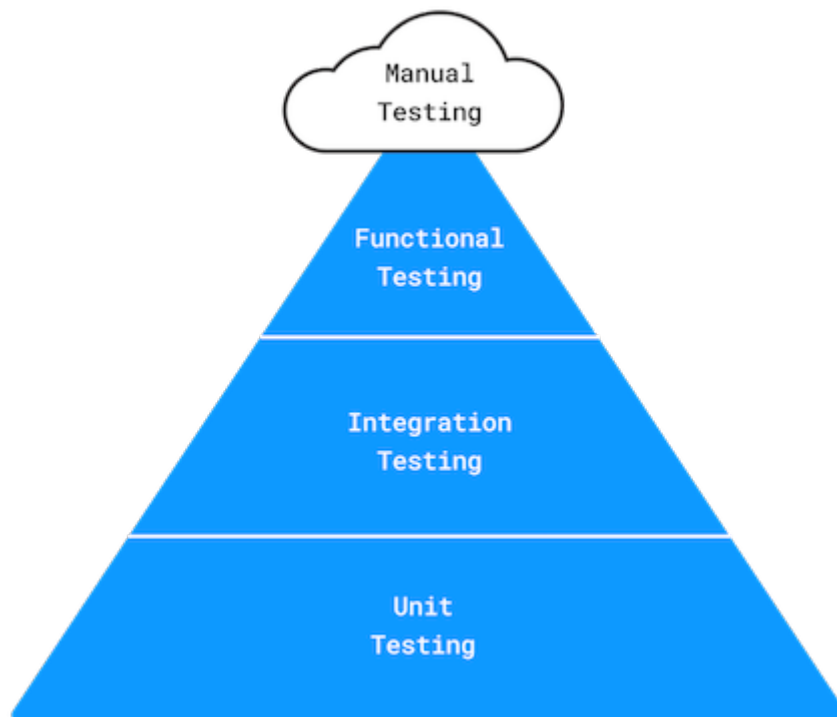
Manual Testing

While manual testing is straightforward, it is error-prone and requires human effort. For Gradle plugins, manual testing involves using the plugin in a build script.

Automated Testing

Automated testing includes unit, integration, and functional testing.

The testing pyramid introduced by Mike Cohen in his book [Succeeding with Agile: Software Development Using Scrum](#) describes three types of automated tests:



1. **Unit Testing:** Verifies the smallest units of code, typically methods, in isolation. It uses Stubs or Mocks to isolate code from external dependencies.
2. **Integration Testing:** Validates that multiple units or components work together.
3. **Functional Testing:** Tests the system from the end user's perspective, ensuring correct functionality. End-to-end tests for Gradle plugins simulate a build, apply the plugin, and execute specific tasks to verify functionality.

Tooling support

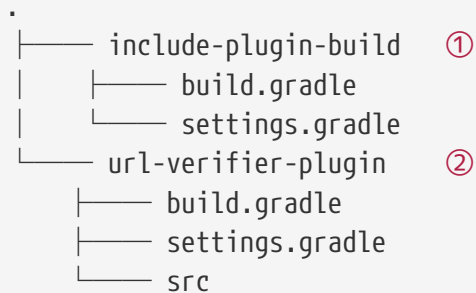
Testing Gradle plugins, both manually and automatically, is simplified with the appropriate tools. The table below provides a summary of each testing approach. You can choose any test framework you're comfortable with.

For detailed explanations and code examples, refer to the specific sections below:

Test type	Tooling support
Manual tests	Gradle composite builds
Unit tests	Any JVM-based test framework
Integration tests	Any JVM-based test framework
Functional tests	Any JVM-based test framework and Gradle TestKit

Setting up manual tests

The [composite builds](#) feature of Gradle makes it easy to test a plugin manually. The standalone plugin project and the consuming project can be combined into a single unit, making it straightforward to try out or debug changes without re-publishing the binary file:



① Consuming project that includes the plugin project

② The plugin project

There are two ways to include a plugin project in a consuming project:

1. By using the command line option `--include-build`.
2. By using the method `includeBuild` in `settings.gradle`.

The following code snippet demonstrates the use of the settings file:

settings.gradle.kts

```
pluginManagement {
    includeBuild("../url-verifier-plugin")
}
```

settings.gradle

```
pluginManagement {
    includeBuild '../url-verifier-plugin'
}
```

The command line output of the `verifyUrl` task from the project `include-plugin-build` looks exactly the same as shown in the introduction, except that it now executes as part of a composite build.

Manual testing has its place in the development process, but it is not a replacement for automated testing.

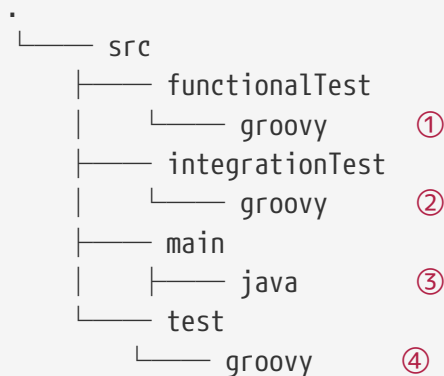
Setting up automated tests

Setting up a suite of tests early on is crucial to the success of your plugin. Automated tests become an invaluable safety net when upgrading the plugin to a new Gradle version or enhancing/refactoring the code.

Organizing test source code

We recommend implementing a good distribution of unit, integration, and functional tests to cover the most important use cases. Separating the source code for each test type automatically results in a project that is more maintainable and manageable.

By default, the Java project creates a convention for organizing unit tests in the directory `src/test/java`. Additionally, if you apply the Groovy plugin, source code under the directory `src/test/groovy` is considered for compilation (with the same standard for Kotlin under the directory `src/test/kotlin`). Consequently, source code directories for other test types should follow a similar pattern:



- ① Source directory containing functional tests
- ② Source directory containing integration tests
- ③ Source directory containing production source code
- ④ Source directory containing unit tests

NOTE

The directories `src/integrationTest/groovy` and `src/functionalTest/groovy` are not based on an existing standard convention for Gradle projects. You are free to choose any project layout that works best for you.

You can configure the source directories for compilation and test execution.

The [Test Suite plugin](#) provides a DSL and API to model multiple groups of automated tests into test suites in JVM-based projects. You can also rely on third-party plugins for convenience, such as the [Nebula Facet plugin](#) or the [TestSets plugin](#).

Modeling test types

NOTE

A new configuration DSL for modeling the below `integrationTest` suite is available via the incubating [JVM Test Suite](#) plugin.

In Gradle, source code directories are represented using the concept of [source sets](#). A source set is configured to point to one or more directories containing source code. When you define a source set, Gradle automatically sets up compilation tasks for the specified directories.

A pre-configured source set can be created with one line of build script code. The source set

automatically registers configurations to define dependencies for the sources of the source set:

```
// Define a source set named 'test' for test sources
sourceSets {
    test {
        java {
            srcDirs = ['src/test/java']
        }
    }
}
// Specify a test implementation dependency on JUnit
dependencies {
    testImplementation 'junit:junit:4.12'
}
```

We use that to define an `integrationTestImplementation` dependency to the project itself, which represents the "main" variant of our project (i.e., the compiled plugin code):

build.gradle.kts

```
val integrationTest by sourceSets.createing

dependencies {
    "integrationTestImplementation"(project)
}
```

build.gradle

```
def integrationTest = sourceSets.create("integrationTest")

dependencies {
    integrationTestImplementation(project)
}
```

Source sets are responsible for compiling source code, but they do not deal with executing the bytecode. For test execution, a corresponding task of type `Test` needs to be established. The following setup shows the execution of integration tests, referencing the classes and runtime classpath of the integration test source set:

build.gradle.kts

```
val integrationTestTask = tasks.register<Test>("integrationTest") {
```

```

        description = "Runs the integration tests."
        group = "verification"
        testClassesDirs = integrationTest.output.classesDirs
        classpath = integrationTest.runtimeClasspath
        mustRunAfter(tasks.test)
    }
    tasks.check {
        dependsOn(integrationTestTask)
    }
}

```

build.gradle

```

def integrationTestTask = tasks.register("integrationTest", Test) {
    description = 'Runs the integration tests.'
    group = "verification"
    testClassesDirs = integrationTest.output.classesDirs
    classpath = integrationTest.runtimeClasspath
    mustRunAfter(tasks.named('test'))
}
tasks.named('check') {
    dependsOn(integrationTestTask)
}

```

Configuring a test framework

Gradle does not dictate the use of a specific test framework. Popular choices include [JUnit](#), [TestNG](#) and [Spock](#). Once you choose an option, you have to add its dependency to the compile classpath for your tests.

The following code snippet shows how to use Spock for implementing tests:

build.gradle.kts

```

repositories {
    mavenCentral()
}

dependencies {
    testImplementation(platform("org.spockframework:spock-bom:2.2-groovy-3.0"))
    testImplementation("org.spockframework:spock-core")
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")

    "integrationTestImplementation"(platform("org.spockframework:spock-bom:2.2-groovy-3.0"))
}

```

```

    "integrationTestImplementation"("org.spockframework:spock-core")
    "integrationTestRuntimeOnly"("org.junit.platform:junit-platform-
launcher")

    "functionalTestImplementation"(platform("org.spockframework:spock-
bom:2.2-groovy-3.0"))
    "functionalTestImplementation"("org.spockframework:spock-core")
    "functionalTestRuntimeOnly"("org.junit.platform:junit-platform-launcher")
}

tasks.withType<Test>().configureEach {
    // Using JUnitPlatform for running tests
    useJUnitPlatform()
}

```

build.gradle

```

repositories {
    mavenCentral()
}

dependencies {
    testImplementation platform("org.spockframework:spock-bom:2.2-groovy-3.0
")
    testImplementation 'org.spockframework:spock-core'
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'

    integrationTestImplementation platform("org.spockframework:spock-bom:2.2-
groovy-3.0")
    integrationTestImplementation 'org.spockframework:spock-core'
    integrationTestRuntimeOnly 'org.junit.platform:junit-platform-launcher'

    functionalTestImplementation platform("org.spockframework:spock-bom:2.2-
groovy-3.0")
    functionalTestImplementation 'org.spockframework:spock-core'
    functionalTestRuntimeOnly 'org.junit.platform:junit-platform-launcher'
}

tasks.withType(Test).configureEach {
    // Using JUnitPlatform for running tests
    useJUnitPlatform()
}

```

NOTE

Spock is a Groovy-based BDD test framework that even includes APIs for creating Stubs and Mocks. The Gradle team prefers Spock over other options for its expressiveness and conciseness.

Implementing automated tests

This section discusses representative implementation examples for unit, integration, and functional tests. All test classes are based on the use of Spock, though it should be relatively easy to adapt the code to a different test framework.

Implementing unit tests

The URL verifier plugin emits HTTP GET calls to check if a URL can be resolved successfully. The method `DefaultHttpClient.get(String)` is responsible for calling a given URL and returns an instance of type `HttpResponse`. `HttpResponse` is a POJO containing information about the HTTP response code and message:

HttpResponse.java

```
package org.myorg.http;

public class HttpResponse {
    private int code;
    private String message;

    public HttpResponse(int code, String message) {
        this.code = code;
        this.message = message;
    }

    public int getCode() {
        return code;
    }

    public String getMessage() {
        return message;
    }

    @Override
    public String toString() {
        return "HTTP " + code + ", Reason: " + message;
    }
}
```

The class `HttpResponse` represents a good candidate for a unit test. It does not reach out to any other classes nor does it use the Gradle API.

HttpResponseTest.groovy

```
package org.myorg.http

import spock.lang.Specification

class HttpResponseTest extends Specification {
```

```

private static final int OK_HTTP_CODE = 200
private static final String OK_HTTP_MESSAGE = 'OK'

def "can access information"() {
    when:
    def httpResponse = new HttpResponse(OK_HTTP_CODE, OK_HTTP_MESSAGE)

    then:
    httpResponse.code == OK_HTTP_CODE
    httpResponse.message == OK_HTTP_MESSAGE
}

def "can get String representation"() {
    when:
    def httpResponse = new HttpResponse(OK_HTTP_CODE, OK_HTTP_MESSAGE)

    then:
    httpResponse.toString() == "HTTP $OK_HTTP_CODE, Reason: $OK_HTTP_MESSAGE"
}
}

```

IMPORTANT

When writing unit tests, it's important to test boundary conditions and various forms of invalid input. Try to extract as much logic as possible from classes that use the Gradle API to make it testable as unit tests. It will result in maintainable code and faster test execution.

You can use the [ProjectBuilder](#) class to create [Project](#) instances to use when you test your plugin implementation.

src/test/java/org/example/GreetingPluginTest.java

```

public class GreetingPluginTest {
    @Test
    public void greeterPluginAddsGreetingTaskToProject() {
        Project project = ProjectBuilder.builder().build();
        project.getPluginManager().apply("org.example.greeting");

        assertTrue(project.getTasks().getByName("hello") instanceof GreetingTask);
    }
}

```

Implementing integration tests

Let's look at a class that reaches out to another system, the piece of code that emits the HTTP calls. At the time of executing a test for the class [DefaultHttpClient](#), the runtime environment needs to be able to reach out to the internet:

DefaultHttpClient.java

```
package org.myorg.http;

import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.URI;
import java.net.URISyntaxException;

public class DefaultHttpClient implements HttpClient {
    @Override
    public HttpResponse get(String url) {
        try {
            HttpURLConnection connection = (HttpURLConnection) new URI(url).toURL()
                .openConnection();
            connection.setConnectTimeout(5000);
            connection.setRequestMethod("GET");
            connection.connect();

            int code = connection.getResponseCode();
            String message = connection.getResponseMessage();
            return new HttpResponse(code, message);
        } catch (IOException e) {
            throw new HttpCallException(String.format("Failed to call URL '%s' via
HTTP GET", url), e);
        } catch (URISyntaxException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Implementing an integration test for `DefaultHttpClient` doesn't look much different from the unit test shown in the previous section:

DefaultHttpClientIntegrationTest.groovy

```
package org.myorg.http

import spock.lang.Specification
import spock.lang.Subject

class DefaultHttpClientIntegrationTest extends Specification {
    @Subject HttpClient httpClient = new DefaultHttpClient()

    def "can make successful HTTP GET call"() {
        when:
        def httpResponse = httpClient.get('https://www.google.com/')

        then:
        httpResponse.code == 200
    }
}
```

```

        httpResponse.message == 'OK'
    }

    def "throws exception when calling unknown host via HTTP GET"() {
        when:
        httpCaller.get('https://www.wedonotknowyou123.com/')

        then:
        def t = thrown(HttpCallException)
        t.message == "Failed to call URL 'https://www.wedonotknowyou123.com/' via HTTP
GET"
        t.cause instanceof UnknownHostException
    }
}

```

Implementing functional tests

Functional tests verify the correctness of the plugin end-to-end. In practice, this means applying, configuring, and executing the functionality of the plugin implementation. The `UrlVerifierPlugin` class exposes an extension and a task instance that uses the URL value configured by the end user:

UrlVerifierPlugin.java

```

package org.myorg;

import org.gradle.api.Plugin;
import org.gradle.api.Project;
import org.myorg.tasks.UrlVerify;

public class UrlVerifierPlugin implements Plugin<Project> {
    @Override
    public void apply(Project project) {
        UrlVerifierExtension extension = project.getExtensions().create("verification",
        UrlVerifierExtension.class);
        UrlVerify verifyUrlTask = project.getTasks().create("verifyUrl", UrlVerify
        .class);
        verifyUrlTask.getUrl().set(extension.getUrl());
    }
}

```

Every Gradle plugin project should apply the [plugin development plugin](#) to reduce boilerplate code. By applying the plugin development plugin, the test source set is preconfigured for the use with TestKit. If we want to use a custom source set for functional tests and leave the default test source set for only unit tests, we can configure the plugin development plugin to look for TestKit tests elsewhere.

build.gradle.kts

```
gradlePlugin {  
    testSourceSets(functionalTest)  
}
```

build.gradle

```
gradlePlugin {  
    testSourceSets(sourceSets.functionalTest)  
}
```

Functional tests for Gradle plugins use an instance of `GradleRunner` to execute the build under test. `GradleRunner` is an API provided by TestKit, which internally uses the Tooling API to execute the build.

The following example applies the plugin to the build script under test, configures the extension and executes the build with the task `verifyUrl`. Please see the [TestKit documentation](#) to get more familiar with the functionality of TestKit.

UrlVerifierPluginFunctionalTest.groovy

```
package org.myorg  
  
import org.gradle.testkit.runner.GradleRunner  
import spock.lang.Specification  
import spock.lang.TempDir  
  
import static org.gradle.testkit.runner.TaskOutcome.SUCCESS  
  
class UrlVerifierPluginFunctionalTest extends Specification {  
    @TempDir File testProjectDir  
    File buildFile  
  
    def setup() {  
        buildFile = new File(testProjectDir, 'build.gradle')  
        buildFile << """  
            plugins {  
                id 'org.myorg.url-verifier'  
            }  
        """  
    }  
  
    def "can successfully configure URL through extension and verify it"() {  
        buildFile << """
```

```

        verification {
            url = 'https://www.google.com/'
        }
    """

    when:
    def result = GradleRunner.create()
        .withProjectDir(testProjectDir)
        .withArguments('verifyUrl')
        .withPluginClasspath()
        .build()

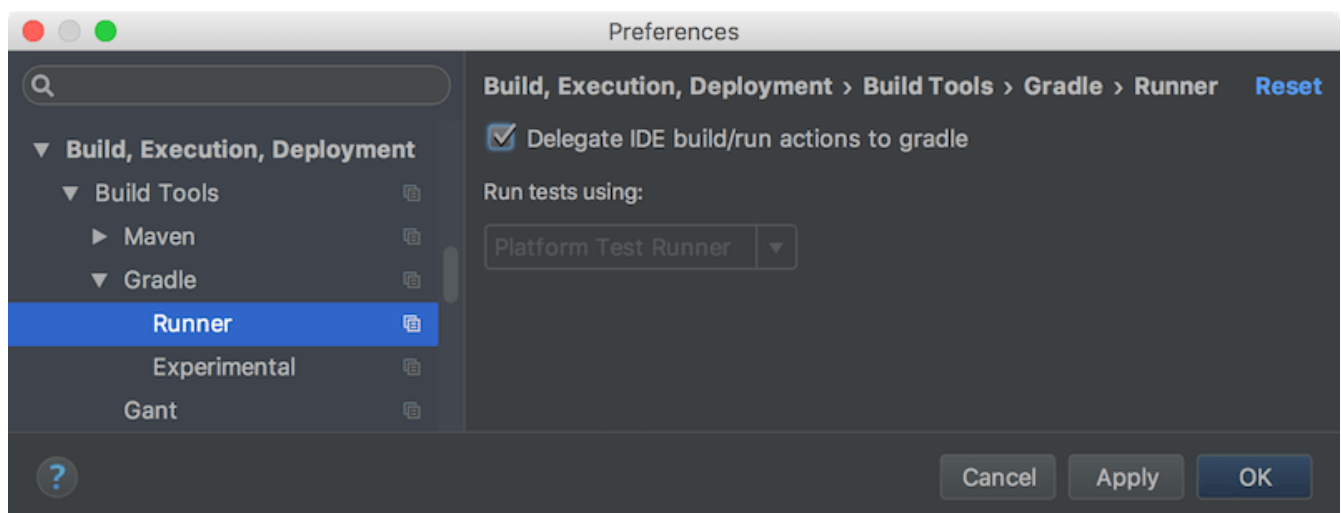
    then:
    result.output.contains("Successfully resolved URL 'https://www.google.com/'")
    result.task(":verifyUrl").outcome == SUCCESS
}
}

```

IDE integration

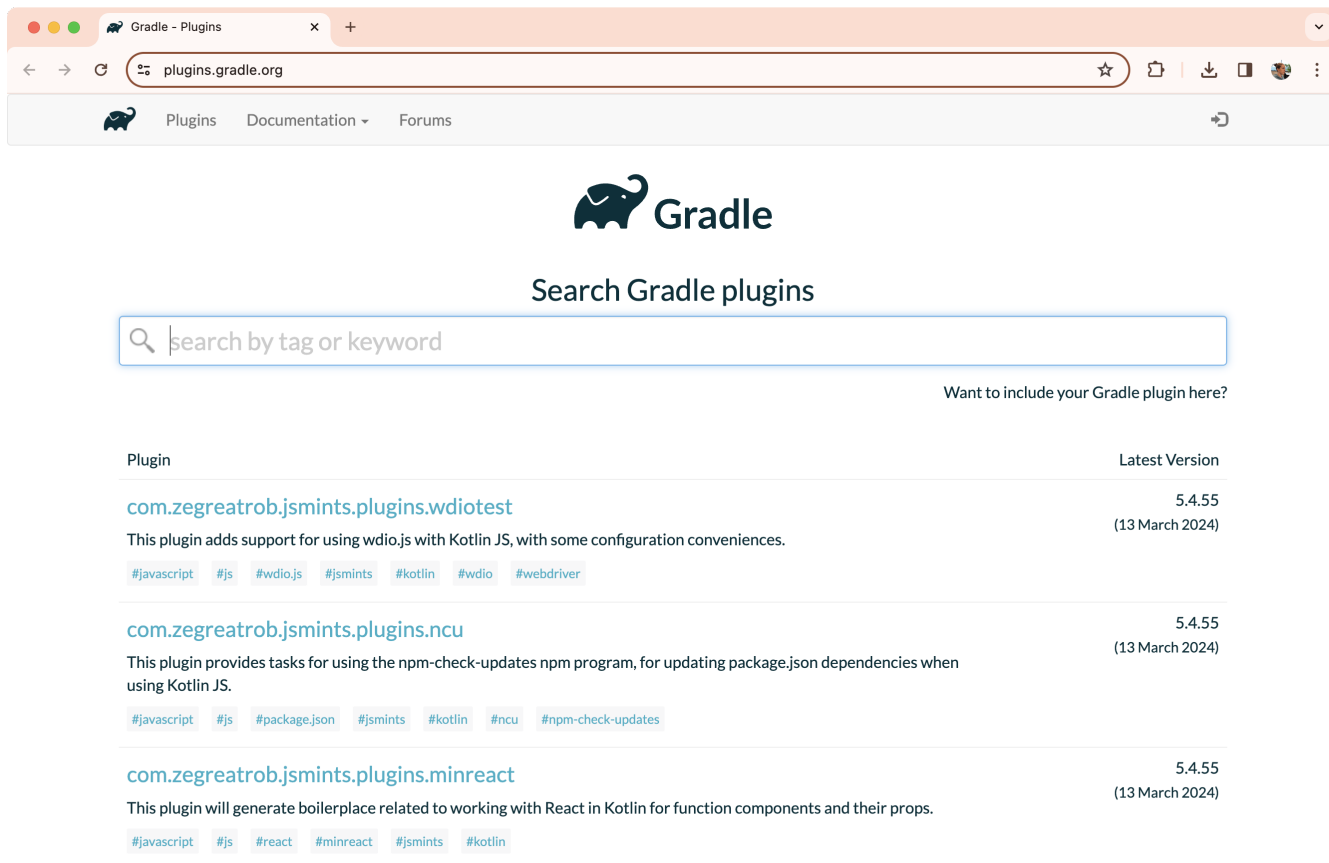
TestKit determines the plugin classpath by running a specific Gradle task. You will need to execute the `assemble` task to initially generate the plugin classpath or to reflect changes to it even when running TestKit-based functional tests from the IDE.

Some IDEs provide a convenience option to delegate the "test classpath generation and execution" to the build. In IntelliJ, you can find this option under Preferences... > Build, Execution, Deployment > Build Tools > Gradle > Runner > Delegate IDE build/run actions to Gradle.



Publishing Plugins to the Gradle Plugin Portal

Publishing a plugin is the primary way to make it available for others to use. While you can publish to a private repository to restrict access, publishing to the [Gradle Plugin Portal](https://plugins.gradle.org/) makes your plugin available to anyone in the world.



The screenshot shows the Gradle Plugins website in a browser. The address bar is 'plugins.gradle.org'. The page has a navigation bar with 'Plugins', 'Documentation', and 'Forums'. The main heading is 'Search Gradle plugins' with a search input field containing 'search by tag or keyword'. Below the search bar, there's a link 'Want to include your Gradle plugin here?'. A table lists three plugins:

Plugin	Latest Version
com.zegreatrob.jsmints.plugins.wdiotest This plugin adds support for using wdio.js with Kotlin JS, with some configuration conveniences. #javascript #js #wdio.js #jsmints #kotlin #wdio #webdriver	5.4.55 (13 March 2024)
com.zegreatrob.jsmints.plugins.ncu This plugin provides tasks for using the npm-check-updates npm program, for updating package.json dependencies when using Kotlin JS. #javascript #js #package.json #jsmints #kotlin #ncu #npm-check-updates	5.4.55 (13 March 2024)
com.zegreatrob.jsmints.plugins.minreact This plugin will generate boilerplate related to working with React in Kotlin for function components and their props. #javascript #js #react #minreact #jsmints #kotlin	5.4.55 (13 March 2024)

This guide shows you how to use the `com.gradle.plugin-publish` plugin to publish plugins to the [Gradle Plugin Portal](#) using a convenient DSL. This approach streamlines configuration steps and provides validation checks to ensure your plugin meets the Gradle Plugin Portal's criteria.

Prerequisites

You'll need an existing Gradle plugin project for this tutorial. If you don't have one, use the [Greeting plugin sample](#).

Attempting to publish this plugin will safely fail with a permission error, so don't worry about cluttering up the Gradle Plugin Portal with a trivial example plugin.

Account setup

Before publishing your plugin, you must create an account on the Gradle Plugin Portal. Follow the instructions on the [registration page](#) to create an account and obtain an API key from your profile page's "API Keys" tab.

To publish your plugin, add the `com.gradle.plugin-publish` plugin to your project's `build.gradle` or `build.gradle.kts` file:

build.gradle.kts

```
plugins {  
    id("com.gradle.plugin-publish") version "1.2.1"  
}
```

build.gradle

```
plugins {  
    id 'com.gradle.plugin-publish' version '1.2.1'  
}
```

The latest version of the Plugin Publishing Plugin can be found on the [Gradle Plugin Portal](#).

NOTE

Since version 1.0.0 the Plugin Publish Plugin automatically applies the [Java Gradle Plugin Development Plugin](#) (assists with developing Gradle plugins) and the [Maven Publish Plugin](#) (generates plugin publication metadata). If using older versions of the Plugin Publish Plugin, these helper plugins must be applied explicitly.

Configuring the Plugin Publishing Plugin

Configure the `com.gradle.plugin-publish` plugin in your `build.gradle` or `build.gradle.kts` file.

build.gradle.kts

```
group = "io.github.johndoe" ①  
version = "1.0" ②  
  
gradlePlugin { ③  
    website = "<substitute your project website>" ④  
    vcsUrl = "<uri to project source repository>" ⑤  
  
    // ... ⑥  
}
```

build.gradle

```
group = 'io.github.johndoe' ①  
version = '1.0' ②
```

```
gradlePlugin { ❸
    website = '<substitute your project website>' ❹
    vcsUrl = '<uri to project source repository>' ❺

    // ... ❻
}
```

- ❶ Make sure your project has a **group** set which is used to identify the artifacts (jar and metadata) you publish for your plugins in the repository of the Gradle Plugin Portal and which is descriptive of the plugin author or the organization the plugins belong too.
- ❷ Set the version of your project, which will also be used as the version of your plugins.
- ❸ Use the **gradlePlugin** block provided by the [Java Gradle Plugin Development Plugin](#) to configure further options for your plugin publication.
- ❹ Set the website for your plugin's project.
- ❺ Provide the source repository URI so that others can find it, if they want to contribute.
- ❻ Set specific properties for each plugin you want to publish; see next section.

Define common properties for all plugins, such as group, version, website, and source repository, using the **gradlePlugin{}** block:

build.gradle.kts

```
gradlePlugin { ❶
    // ... ❷

    plugins { ❸
        create("greetingsPlugin") { ❹
            id = "<your plugin identifier>" ❺
            displayName = "<short displayable name for plugin>" ❻
            description = "<human-readable description of what your plugin is
about>" ❼
            tags = listOf("tags", "for", "your", "plugins") ❽
            implementationClass = "<your plugin class>"
        }
    }
}
```

build.gradle

```
gradlePlugin { ❶
    // ... ❷
```

```

    plugins { ③
        greetingsPlugin { ④
            id = '<your plugin identifier>' ⑤
            displayName = '<short displayable name for plugin>' ⑥
            description = '<human-readable description of what your plugin is
about>' ⑦
            tags.set(['tags', 'for', 'your', 'plugins']) ⑧
            implementationClass = '<your plugin class>'
        }
    }
}

```

- ① Plugin specific configuration also goes into the `gradlePlugin` block.
- ② This is where we previously added global properties.
- ③ Each plugin you publish will have its own block inside `plugins`.
- ④ The name of a plugin block must be unique for each plugin you publish; this is a property used only locally by your build and will not be part of the publication.
- ⑤ Set the unique `id` of the plugin, as it will be identified in the publication.
- ⑥ Set the plugin name in human-readable form.
- ⑦ Set a description to be displayed on the portal. It provides useful information to people who want to use your plugin.
- ⑧ Specifies the categories your plugin covers. It makes the plugin more likely to be discovered by people needing its functionality.

For example, consider the configuration for the [GradleTest plugin](#), already published to the Gradle Plugin Portal.

build.gradle.kts

```

gradlePlugin {
    website = "https://github.com/ysb33r/gradleTest"
    vcsUrl = "https://github.com/ysb33r/gradleTest.git"
    plugins {
        create("gradleTestPlugin") {
            id = "org.ysb33r.gradleTest"
            displayName = "Plugin for compatibility testing of Gradle
plugins"
            description = "A plugin that helps you test your plugin against a
variety of Gradle versions"
            tags = listOf("testing", "integrationTesting", "compatibility")
            implementationClass =
"org.ysb33r.gradle.gradleTest.GradleTestPlugin"
        }
    }
}

```

```
}
```

build.gradle

```
gradlePlugin {
    website = 'https://github.com/ysb33r/gradleTest'
    vcsUrl = 'https://github.com/ysb33r/gradleTest.git'
    plugins {
        gradleTestPlugin {
            id = 'org.ysb33r.gradleTest'
            displayName = 'Plugin for compatibility testing of Gradle
plugins'
            description = 'A plugin that helps you test your plugin against a
variety of Gradle versions'
            tags.addAll('testing', 'integrationTesting', 'compatibility')
            implementationClass =
'org.ysb33r.gradle.gradleTest.GradleTestPlugin'
        }
    }
}
```

If you browse the associated page on the Gradle Plugin Portal for the [GradleTest plugin](#), you will see how the specified metadata is displayed.

org.ysb33r.gradleTest

Owner:  Schalk Cronjé

A plugin that helps you test your plugin against a variety of Gradle versions

<https://gitlab.com/ysb33rOrg/gradleTest>

Sources: <https://github.com/ysb33r/gradleTest.git>

[#testing](#) [#integrationtesting](#) [#compatibility](#)

gradlePlugin.plugins.gradleTestPlugin.id

gradlePlugin.plugins.gradleTestPlugin.description

gradlePlugin.website

gradlePlugin.plugins.gradleTestPlugin.tags

Version 3.0.0-alpha.4 (latest)

[Other versions](#) ▾

Created 19 September 2023.

A plugin that helps you test your plugin against a variety of Gradle versions

Sources & Javadoc

The Plugin Publish Plugin automatically generates and publishes the [Javadoc](#), and [sources JARs](#) for your plugin publication.

Sign artifacts

Starting from version 1.0.0 of Plugin Publish Plugin, the signing of published plugin artifacts has been made automatic. To enable it, all that's needed is to apply the [signing](#) plugin in your build.

Shadow dependencies

Starting from version 1.0.0 of Plugin Publish Plugin, shadowing your plugin's dependencies (ie, publishing it as a fat jar) has been made automatic. To enable it, all that's needed is to apply the `com.gradleup.shadow` plugin in your build.

Publishing the plugin

If you publish your plugin internally for use within your organization, you can publish it like any other code artifact. See the [Ivy](#) and [Maven](#) chapters on publishing artifacts.

If you are interested in publishing your plugin to be used by the wider Gradle community, you can publish it to [Gradle Plugin Portal](#). This site provides the ability to search for and gather information about plugins contributed by the Gradle community. Please refer to the corresponding [section](#) on making your plugin available on this site.

Publish locally

To check how the artifacts of your published plugin look or to use it only locally or internally in your company, you can publish it to any Maven repository, including a local folder. You only need to [configure repositories for publishing](#). Then, you can run the `publish` task to publish your plugin to all repositories you have defined (but not the Gradle Plugin Portal).

build.gradle.kts

```
publishing {
    repositories {
        maven {
            name = "localPluginRepository"
            url = uri("../local-plugin-repository")
        }
    }
}
```

build.gradle

```
publishing {
    repositories {
        maven {
            name = 'localPluginRepository'
            url = file('../local-plugin-repository')
        }
    }
}
```

To use the repository in another build, add it to [the repositories of the `pluginManagement {}` block](#) in your `settings.gradle(.kts)` file.

Publish to the Plugin Portal

Publish the plugin by using the `publishPlugin` task:

```
$ ./gradlew publishPlugins
```

You can validate your plugins before publishing using the `--validate-only` flag:

```
$ ./gradlew publishPlugins --validate-only
```

If you have not configured your `gradle.properties` for the Gradle Plugin Portal, you can specify them on the command-line:

```
$ ./gradlew publishPlugins -Pgradle.publish.key=<key> -Pgradle.publish.secret=<secret>
```

NOTE

You will encounter a permission failure if you attempt to publish the example Greeting Plugin with the ID used in this section. That's expected and ensures the portal won't be overrun with multiple experimental and duplicate greeting-type plugins.

After approval, your plugin will be available on the Gradle Plugin Portal for others to discover and use.

Consume the published plugin

Once you successfully publish a plugin, it won't immediately appear on the Portal. It also needs to pass an approval process, which is manual and relatively slow for the initial version of your plugin, but is fully automatic for subsequent versions. For further details, see [here](#).

Once your plugin is approved, you can find instructions for its use at a URL of the form <https://plugins.gradle.org/plugin/<your-plugin-id>>. For example, the Greeting Plugin example is already on the portal at <https://plugins.gradle.org/plugin/org.example.greeting>.

Plugins published without Gradle Plugin Portal

If your plugin was published without using the [Java Gradle Plugin Development Plugin](#), the publication will be lacking [Plugin Marker Artifact](#), which is needed for [plugins DSL](#) to locate the plugin. In this case, the recommended way to resolve the plugin in another project is to add a `resolutionStrategy` section to the `pluginManagement {}` block of the project's settings file, as shown below.

settings.gradle.kts

```
resolutionStrategy {
    eachPlugin {
        if (requested.id.namespace == "org.example") {
            useModule("org.example:custom-plugin:${requested.version}")
        }
    }
}
```

settings.gradle

```
resolutionStrategy {
    eachPlugin {
        if (requested.id.namespace == 'org.example') {
            useModule("org.example:custom-plugin:${requested.version}")
        }
    }
}
```

Reporting problems

Plugins can report problems through Gradle's problems-reporting APIs. The APIs report rich, structured information about problems happening during the build. This information can be used by different user interfaces such as Gradle's console output, Build Scans, or IDEs to communicate problems to the user in the most appropriate way.

The following example shows an issue reported from a plugin:

ProblemReportingPlugin.java

```
public class ProblemReportingPlugin implements Plugin<Project> {

    public static final ProblemGroup PROBLEM_GROUP = ProblemGroup.create(
        "sample-group", "Sample Group");

    private final ProblemReporter problemReporter;

    @Inject
    public ProblemReportingPlugin(Problems problems) { ①
        this.problemReporter = problems.getReporter(); ②
    }
}
```

```

public void apply(Project project) {
    ProblemId problemId = ProblemId.create("adhoc-deprecation", "Plugin
    'x' is deprecated", PROBLEM_GROUP);
    this.problemReporter.report(problemId, builder -> builder ③
        .details("The plugin 'x' is deprecated since version 2.5")
        .solution("Please use plugin 'y'")
        .severity(Severity.WARNING)
    );
}
}

```

- ① The **Problem** service is injected into the plugin.
- ② A problem reporter, is created for the plugin. While the namespace is up to the plugin author, it is recommended that the plugin ID be used.
- ③ A problem is reported. This problem is recoverable so that the build will continue.

For a full example, see our [end-to-end sample](#).

Problem building

When reporting a problem, a wide variety of information can be provided. The [ProblemSpec](#) describes all the information that can be provided.

Reporting problems

When it comes to reporting problems, we support two different modes:

- **Reporting** a problem is used for reporting problems that are recoverable, and the build should continue.
- **Throwing** a problem is used for reporting problems that are not recoverable, and the build should fail.

For more details, see the [ProblemReporter](#) documentation.

Problem summarization

When reporting issues, Gradle ensures that the reports are concise and free of unnecessary redundancy. Specifically, it prevents reporting the same problem repeatedly once a certain threshold is reached.

- During the build, the *first* few instances of a problem are reported as a [Problem](#), providing all information available for that problem.
- At the end of the build, subsequent occurrences of the same problem are grouped and summarized as a [ProblemSummary](#). This summary is delivered as with a [ProblemSummariesEvent](#), which provides the total count of occurrences.

Build failures

The standard approach for indicating a build failure is to throw an exception. [ProblemReporter](#) provides enhanced support for this by allowing exceptions to be thrown with associated problem reports.

FailingTask.java

```
ProblemId id = ProblemId.create("sample-error", "Sample Error", StandardPlugin
.PROBLEM_GROUP);
throw getProblems().getReporter().throwing((new RuntimeException("Message from
runtime exception")), id, problemSpec -> {
    problemSpec.contextualLabel("This happened because ProblemReporter.throwing()
was called");
    problemSpec.details("This is a demonstration of how to add\ndetailed
information to a build failure");
    problemSpec.documentedAt("https://example.com/docs");
    problemSpec.solution("Remove the Problems.throwing() method call from the task
action");
});
```

This ensures that build failures are clearly linked to the underlying issues and that these problems are properly communicated to various clients. When a build failure is reported using the Problems API, all clients (Tooling API, CLI, Build Scans, etc.) will have access to the association.

Command-line interface

The CLI build failure output will include detailed information about the problem. Error messages and descriptions are sourced directly from the problem report. If the problem report includes a solution or recommended actions, these will be displayed in place of generic resolutions.

FAILURE: Build failed with an exception.

* What went wrong:

Execution failed for task ':sample-project:myFailingTask'.

> Message from runtime exception

 This happened because ProblemReporter.throwing() was called

 This is a demonstration of how to add
 detailed information to a build failure

* Try:

> Remove the Problems.throwing() method call from the task action

> Run with --scan to get full insights.

BUILD FAILED in 0ms

Tooling API clients

Tooling API clients can access detailed problem reports associated with build failures via the `Failure` object on the root build operation. To receive these reports, the clients must register a progress listener for the `OperationType.ROOT` operation type. The progress listener callback should then check if the operation result is of type `FailureResult`, and then it can access the associated problems via `Failure.getProblems()`.

In addition, there's a more convenient way to access the failure details. If clients configure the project connection with `LongRunningOperation.withFailureDetails()`, the Tooling API implicitly subscribes to the `ROOT` operation type and provides failure details via the `GradleConnectionException.getFailures()` method.

Generated HTML report

The output of the problems generated by the Problems API is also provided as a rich HTML report generated at the end of the build. This report serves as a central location for users to review problems that occurred during a build.

Plugin authors can use the Problems API to log events specific to their plugins, adding to the Gradle-generated ones.

The report is not generated if no issues have been reported. Also, if you do not want to generate this report, you can disable it with the `--no-problems-report` flag. The console output provides a link to this report, as shown below:

```
[Incubating] Problem report is available at: <project-dir>/build/reports/problems/problems-report.html
```

```
BUILD SUCCESSFUL in 1s
```

The rendered report link directs you to a detailed HTML view of the problems:



Problems Report

1 problems have been reported during the execution for the following tasks: sC

Messages **1**

Group **1**

Locations **1**

▼ **Deprecation** deprecation

> **Executing Gradle on JVM versions 16 and lower has been deprecated.**

– Ungrouped

[1] **Script plugins** are hard to maintain. Do not use script plugins **apply from:**, they are not recommended.

[2] It is recommended to use a statically-typed language like **Java** or **Kotlin** for implementing plugins to reduce the likelihood of binary incompatibilities. If using Groovy, consider using **statically compiled Groovy**.

OTHER TOPICS

Working With Files

File operations are fundamental to nearly every Gradle build. They involve handling source files, managing file dependencies, and generating reports. Gradle provides a robust API that simplifies these operations, enabling developers to perform necessary file tasks easily.

Hardcoded paths and laziness

It is best practice to **avoid** hardcoded paths in build scripts.

In addition to avoiding hardcoded paths, Gradle encourages laziness in its build scripts. This means that tasks and operations should be deferred until they are actually needed rather than executed eagerly.

Many examples in this chapter use hard-coded paths as string literals. This makes them easy to understand, but it is not good practice. The problem is that paths often change, and the more places you need to change them, the more likely you will miss one and break the build.

Where possible, you should use tasks, task properties, and [project properties](#) — in that order of preference — to configure file paths.

For example, if you create a task that packages the compiled classes of a Java application, you should use an implementation similar to this:

build.gradle.kts

```
val archivesDirPath = layout.buildDirectory.dir("archives")

tasks.register<Zip>("packageClasses") {
    archiveAppendix = "classes"
    destinationDirectory = archivesDirPath

    from(tasks.compileJava)
}
```

build.gradle

```
def archivesDirPath = layout.buildDirectory.dir('archives')

tasks.register('packageClasses', Zip) {
    archiveAppendix = "classes"
    destinationDirectory = archivesDirPath
}
```

```
from compileJava
}
```

The `compileJava` task is the source of the files to package, and the project property `archivesDirPath` stores the location of the archives, as we are likely to use it elsewhere in the build.

Using a task directly as an argument like this relies on it having [defined outputs](#), so it won't always be possible. This example could be further improved by relying on the Java plugin's convention for `destinationDirectory` rather than overriding it, but it does demonstrate the use of project properties.

Locating files

To perform some action on a file, you need to know where it is, and that's the information provided by file paths. Gradle builds on the standard Java `File` class, which represents the location of a single file and provides APIs for dealing with collections of paths.

Using `ProjectLayout`

The `ProjectLayout` class is used to access various directories and files within a project. It provides methods to retrieve paths to the project directory, build directory, settings file, and other important locations within the project's file structure. This class is particularly useful when you need to work with files in a build script or plugin in different project paths:

build.gradle.kts

```
val archivesDirPath = layout.buildDirectory.dir("archives")
```

build.gradle

```
def archivesDirPath = layout.buildDirectory.dir('archives')
```

You can learn more about the `ProjectLayout` class in [Services](#).

Using `Project.file()`

Gradle provides the `Project.file(java.lang.Object)` method for specifying the location of a single file or directory.

Relative paths are resolved relative to the project directory, while absolute paths remain unchanged.

CAUTION

Never use `new File(relative path)` unless passed to `file()` or `files()` or `from()` or other methods defined in terms of `file()` or `files()`. Otherwise, this creates a path relative to the current working directory (CWD). Gradle can make no guarantees about the location of the CWD, which means builds that rely on it may break at any time.

Here are some examples of using the `file()` method with different types of arguments:

build.gradle.kts

```
// Using a relative path
var configFile = file("src/config.xml")

// Using an absolute path
configFile = file(configFile.absolutePath)

// Using a File object with a relative path
configFile = file(File("src/config.xml"))

// Using a java.nio.file.Path object with a relative path
configFile = file(Paths.get("src", "config.xml"))

// Using an absolute java.nio.file.Path object
configFile = file(Paths.get(System.getProperty("user.home")).resolve("global-
config.xml"))
```

build.gradle

```
// Using a relative path
File configFile = file('src/config.xml')

// Using an absolute path
configFile = file(configFile.absolutePath)

// Using a File object with a relative path
configFile = file(new File('src/config.xml'))

// Using a java.nio.file.Path object with a relative path
configFile = file(Paths.get('src', 'config.xml'))

// Using an absolute java.nio.file.Path object
configFile = file(Paths.get(System.getProperty('user.home')).resolve('global-
config.xml'))
```

As you can see, you can pass strings, `File` instances and `Path` instances to the `file()` method, all of which result in an absolute `File` object.

In the case of multi-project builds, the `file()` method will always turn relative paths into paths relative to the current project directory, which may be a child project.

Using `ProjectLayout.settingsDirectory()`

To use a path relative to the `settings` directory, access the `Project.layout`, retrieve the `settingsDirectory` from it, and construct an absolute path.

For example:

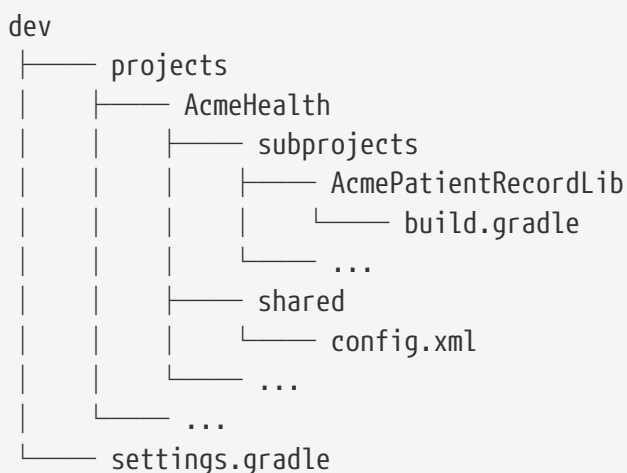
build.gradle.kts

```
val configFile = layout.settingsDirectory.file("shared/config.xml").asFile
```

build.gradle

```
File configFile = layout.settingsDirectory.file("shared/config.xml").asFile
```

Let's say you're working on a multi-project build in the directory: `dev/projects/AcmeHealth`. The build script above is at: `AcmeHealth/subprojects/AcmePatientRecordLib/build.gradle`. The absolute file path will resolve to: `dev/projects/AcmeHealth/shared/config.xml`:



Note that `Project` also provides `Project.getRootProject()` for multi-project builds which, in the example, would resolve to: `dev/projects/AcmeHealth/subprojects/AcmePatientRecordLib`.

Using `FileCollection`

A *file collection* is simply a set of file paths represented by the `FileCollection` interface.

The set of paths can be *any* file path. The file paths don't have to be related in any way, so they don't have to be in the same directory or have a shared parent directory.

The recommended way to specify a collection of files is to use the `ProjectLayout.files(java.lang.Object...)` method, which returns a `FileCollection` instance. This flexible method allows you to pass multiple strings, `File` instances, collections of strings, collections of `Files`, and more. You can also pass in tasks as arguments if they have [defined outputs](#).

CAUTION

`files()` properly handles relative paths and `File(relative path)` instances, resolving them relative to the project directory.

As with the `Project.file(java.lang.Object)` method covered in the [previous section](#), all relative paths are evaluated relative to the current project directory. The following example demonstrates some of the variety of argument types you can use — strings, `File` instances, lists, or `Paths`:

build.gradle.kts

```
val collection: FileCollection = layout.files(  
    "src/file1.txt",  
    File("src/file2.txt"),  
    listOf("src/file3.csv", "src/file4.csv"),  
    Paths.get("src", "file5.txt")  
)
```

build.gradle

```
FileCollection collection = layout.files('src/file1.txt',  
                                         new File('src/file2.txt'),  
                                         ['src/file3.csv', 'src/file4.csv'],  
                                         Paths.get('src', 'file5.txt'))
```

File collections have important attributes in Gradle. They can be:

- created lazily
- iterated over
- filtered
- combined

Lazy creation of a file collection is useful when evaluating the files that make up a collection when a build runs. In the following example, we query the file system to find out what files exist in a particular directory and then make those into a file collection:

build.gradle.kts

```
tasks.register("list") {
    val projectDirectory = layout.projectDirectory
    doLast {
        var srcDir: File? = null

        val collection = projectDirectory.files({
            srcDir?.listFiles()
        })

        srcDir = projectDirectory.file("src").asFile
        println("Contents of ${srcDir.name}")
        collection.map { it.relativeTo(projectDirectory.asFile) }
        }.sorted().forEach { println(it) }

        srcDir = projectDirectory.file("src2").asFile
        println("Contents of ${srcDir.name}")
        collection.map { it.relativeTo(projectDirectory.asFile) }
        }.sorted().forEach { println(it) }
    }
}
```

build.gradle

```
tasks.register('list') {
    Directory projectDirectory = layout.projectDirectory
    doLast {
        File srcDir

        // Create a file collection using a closure
        collection = projectDirectory.files { srcDir.listFiles() }

        srcDir = projectDirectory.file('src').asFile
        println "Contents of $srcDir.name"
        collection.collect { projectDirectory.asFile.relativePath(it) }.sort
        ().each { println it }

        srcDir = projectDirectory.file('src2').asFile
        println "Contents of $srcDir.name"
        collection.collect { projectDirectory.asFile.relativePath(it) }.sort
        ().each { println it }
    }
}
```

```
$ gradle -q list
Contents of src
src/dir1
src/file1.txt
Contents of src2
src2/dir1
src2/dir2
```

The key to lazy creation is passing a closure (in Groovy) or a `Provider` (in Kotlin) to the `files()` method. Your closure or provider must return a value of a type accepted by `files()`, such as `List<File>`, `String`, or `FileCollection`.

Iterating over a file collection can be done through the `each()` method (in Groovy) or `forEach` method (in Kotlin) on the collection or using the collection in a `for` loop. In both approaches, the file collection is treated as a set of `File` instances, i.e., your iteration variable will be of type `File`.

The following example demonstrates such iteration. It also demonstrates how you can convert file collections to other types using the `as` operator (or supported properties):

build.gradle.kts

```
// Iterate over the files in the collection
collection.forEach { file: File ->
    println(file.name)
}

// Convert the collection to various types
val set: Set<File> = collection.files
val list: List<File> = collection.toList()
val path: String = collection.asPath
val file: File = collection.singleFile

// Add and subtract collections
val union = collection + projectLayout.files("src/file2.txt")
val difference = collection - projectLayout.files("src/file2.txt")
```

build.gradle

```
// Iterate over the files in the collection
collection.each { File file ->
    println file.name
}

// Convert the collection to various types
Set set = collection.files
Set set2 = collection as Set
```

```
List list = collection as List
String path = collection.asPath
File file = collection.singleFile

// Add and subtract collections
def union = collection + projectLayout.files('src/file2.txt')
def difference = collection - projectLayout.files('src/file2.txt')
```

You can also see at the end of the example *how to combine file collections* using the `+` and `-` operators to merge and subtract them. An important feature of the resulting file collections is that they are *live*. In other words, when you combine file collections this way, the result always reflects what's currently in the source file collections, even if they change during the build.

For example, imagine `collection` in the above example gains an extra file or two after `union` is created. As long as you use `union` after those files are added to `collection`, `union` will also contain those additional files. The same goes for the `different` file collection.

Live collections are also important when it comes to *filtering*. Suppose you want to use a subset of a file collection. In that case, you can take advantage of the `FileCollection.filter(org.gradle.api.specs.Spec)` method to determine which files to "keep". In the following example, we create a new collection that consists of only the files that end with `.txt` in the source collection:

build.gradle.kts

```
val textFiles: FileCollection = collection.filter { f: File ->
    f.name.endsWith(".txt")
}
```

build.gradle

```
FileCollection textFiles = collection.filter { File f ->
    f.name.endsWith(".txt")
}
```

```
$ gradle -q filterTextFiles
src/file1.txt
src/file2.txt
src/file5.txt
```

If `collection` changes at any time, either by adding or removing files from itself, then `textFiles` will

immediately reflect the change because it is also a live collection. Note that the closure you pass to `filter()` takes a `File` as an argument and should return a boolean.

Understanding implicit conversion to file collections

Many objects in Gradle have properties which accept a set of input files. For example, the `JavaCompile` task has a `source` property that defines the source files to compile. You can set the value of this property using any of the types supported by the `files()` method, as mentioned in the API docs. This means you can, for example, set the property to a `File`, `String`, collection, `FileCollection` or even a closure or `Provider`.

This is a feature of specific tasks! That means implicit conversion will not happen for just any task that has a `FileCollection` or `FileTree` property. If you want to know whether implicit conversion happens in a particular situation, you will need to read the relevant documentation, such as the corresponding task's API docs. Alternatively, you can remove all doubt by explicitly using `ProjectLayout.files(java.lang.Object...)` in your build.

Here are some examples of the different types of arguments that the `source` property can take:

build.gradle.kts

```
tasks.register<JavaCompile>("compile") {
    // Use a File object to specify the source directory
    source = fileTree(file("src/main/java"))

    // Use a String path to specify the source directory
    source = fileTree("src/main/java")

    // Use a collection to specify multiple source directories
    source = fileTree(listOf("src/main/java", "../shared/java"))

    // Use a FileCollection (or FileTree in this case) to specify the source
    files
    source = fileTree("src/main/java").matching {
        include("org/gradle/api/**") }

    // Using a closure to specify the source files.
    setSource({
        // Use the contents of each zip file in the src dir
        file("src").listFiles().filter { it.name.endsWith(".zip") }.map {
            zipTree(it) }
    })
}
```

build.gradle

```
tasks.register('compile', JavaCompile) {
```

```

// Use a File object to specify the source directory
source = file('src/main/java')

// Use a String path to specify the source directory
source = 'src/main/java'

// Use a collection to specify multiple source directories
source = ['src/main/java', '../shared/java']

// Use a FileCollection (or FileTree in this case) to specify the source
files
source = fileTree(dir: 'src/main/java').matching { include
'org/gradle/api/**' }

// Using a closure to specify the source files.
source = {
    // Use the contents of each zip file in the src dir
    file('src').listFiles().findAll {it.name.endsWith('.zip')}.collect {
zipTree(it) }
}
}

```

One other thing to note is that properties like `source` have corresponding methods in core Gradle tasks. Those methods follow the convention of *appending* to collections of values rather than replacing them. Again, this method accepts any of the types supported by the `files()` method, as shown here:

build.gradle.kts

```

tasks.named<JavaCompile>("compile") {
    // Add some source directories use String paths
    source("src/main/java", "src/main/groovy")

    // Add a source directory using a File object
    source(file("../shared/java"))

    // Add some source directories using a closure
    setSource({ file("src/test/").listFiles() })
}

```

build.gradle

```

compile {
    // Add some source directories use String paths
}

```

```

source 'src/main/java', 'src/main/groovy'

// Add a source directory using a File object
source file('../shared/java')

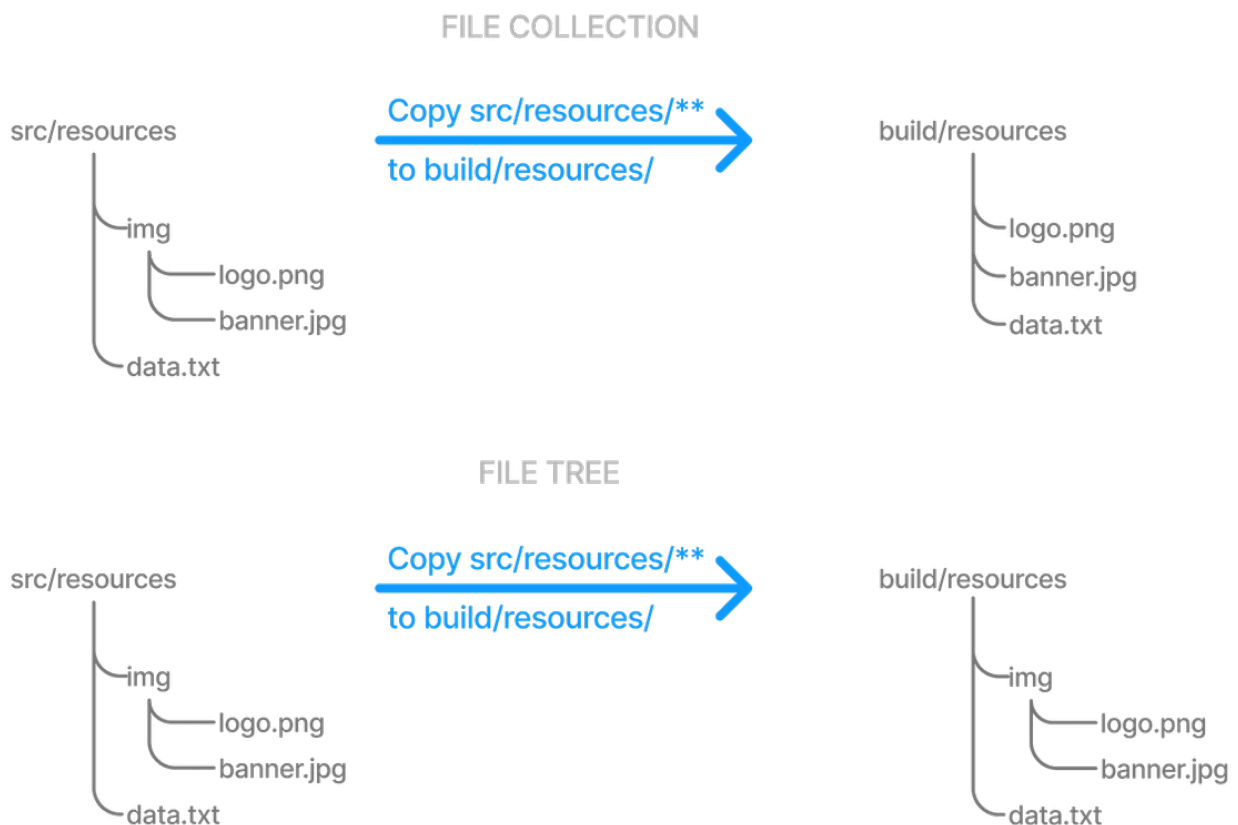
// Add some source directories using a closure
source { file('src/test/').listFiles() }
}

```

As this is a common convention, we recommend that you follow it in your own custom tasks. Specifically, if you plan to add a method to configure a collection-based property, make sure the method appends rather than replaces values.

Using **FileTree**

A *file tree* is a file collection that retains the directory structure of the files it contains and has the type **FileTree**. This means all the paths in a file tree must have a shared parent directory. The following diagram highlights the distinction between file trees and file collections in the typical case of copying files:



NOTE

Although **FileTree** extends **FileCollection** (an is-a relationship), their behaviors differ. In other words, you can use a file tree wherever a file collection is required, but remember that a file collection is a flat list/set of files, while a file tree is a file and directory hierarchy. To convert a file tree to a flat collection, use the **FileTree.GetFiles()** property.

The simplest way to create a file tree is to pass a file or directory path to the [Project.fileTree\(java.lang.Object\)](#) method. This will create a tree of all the files and directories in that base directory (but not the base directory itself). The following example demonstrates how to use this method and how to filter the files and directories using Ant-style patterns:

build.gradle.kts

```
// Create a file tree with a base directory
var tree: ConfigurableFileTree = fileTree("src/main")

// Add include and exclude patterns to the tree
tree.include("**/*.java")
tree.exclude("**/Abstract*")

// Create a tree using closure
tree = fileTree("src") {
    include("**/*.java")
}

// Create a tree using a map
tree = fileTree("dir" to "src", "include" to "**/*.java")
tree = fileTree("dir" to "src", "includes" to listOf("**/*.java",
"**/*.xml"))
tree = fileTree("dir" to "src", "include" to "**/*.java", "exclude" to
"**/*test*/**")
```

build.gradle

```
// Create a file tree with a base directory
ConfigurableFileTree tree = fileTree(dir: 'src/main')

// Add include and exclude patterns to the tree
tree.include '**/*.java'
tree.exclude '**/Abstract*'

// Create a tree using closure
tree = fileTree('src') {
    include '**/*.java'
}

// Create a tree using a map
tree = fileTree(dir: 'src', include: '**/*.java')
tree = fileTree(dir: 'src', includes: ['**/*.java', '**/*.xml'])
tree = fileTree(dir: 'src', include: '**/*.java', exclude: '**/*test*/**')
```

You can see more examples of supported patterns in the API docs for [PatternFilterable](#).

By default, `fileTree()` returns a `FileTree` instance that applies some default exclude patterns for convenience — the same defaults as Ant. For the complete default exclude list, see [the Ant manual](#).

If those default excludes prove problematic, you can work around the issue by changing the default excludes in the settings script:

settings.gradle.kts

```
import org.apache.tools.ant.DirectoryScanner

DirectoryScanner.removeDefaultExclude("**/.git")
DirectoryScanner.removeDefaultExclude("**/.git/**")
```

settings.gradle

```
import org.apache.tools.ant.DirectoryScanner

DirectoryScanner.removeDefaultExclude('**/.git')
DirectoryScanner.removeDefaultExclude('**/.git/**')
```

IMPORTANT

Gradle does not support changing default excludes during the execution phase.

You can do many of the same things with file trees that you can with file collections:

- iterate over them (depth first)
- filter them (using [FileTree.matching\(org.gradle.api.Action\)](#) and Ant-style patterns)
- merge them

You can also traverse file trees using the [FileTree.visit\(org.gradle.api.Action\)](#) method. All of these techniques are demonstrated in the following example:

build.gradle.kts

```
// Iterate over the contents of a tree
tree.forEach{ file: File ->
    println(file)
}

// Filter a tree
```

```

val filtered: FileTree = tree.matching {
    include("org/gradle/api/**")
}

// Add trees together
val sum: FileTree = tree + fileTree("src/test")

// Visit the elements of the tree
tree.visit {
    println("${this.relativePath} => ${this.file}")
}

```

build.gradle

```

// Iterate over the contents of a tree
tree.each {File file ->
    println file
}

// Filter a tree
FileTree filtered = tree.matching {
    include 'org/gradle/api/**'
}

// Add trees together
FileTree sum = tree + fileTree(dir: 'src/test')

// Visit the elements of the tree
tree.visit {element ->
    println "$element.relativePath => $element.file"
}

```

Copying files

Copying files in Gradle primarily uses **CopySpec**, a mechanism that makes it easy to manage resources such as source code, configuration files, and other assets in your project build process.

Understanding **CopySpec**

CopySpec is a copy specification that allows you to define what files to copy, where to copy them from, and where to copy them. It provides a flexible and expressive way to specify complex file copying operations, including filtering files based on patterns, renaming files, and including/excluding files based on various criteria.

CopySpec instances are used in the **Copy** task to specify the files and directories to be copied.

CopySpec has two important attributes:

1. It is independent of tasks, allowing you to *share copy specs within a build*.
2. It is hierarchical, providing *fine-grained control* within the overall copy specification.

1. Sharing copy specs

Consider a build with several tasks that copy a project's static website resources or add them to an archive. One task might copy the resources to a folder for a local HTTP server, and another might package them into a distribution. You could manually specify the file locations and appropriate inclusions each time they are needed, but human error is more likely to creep in, resulting in inconsistencies between tasks.

One solution is the [Project.copySpec\(org.gradle.api.Action\)](#) method. This allows you to create a copy spec outside a task, which can then be attached to an appropriate task using the [CopySpec.with\(org.gradle.api.file.CopySpec...\)](#) method. The following example demonstrates how this is done:

build.gradle.kts

```
val webAssetsSpec: CopySpec = copySpec {
    from("src/main/webapp")
    include("**/*.html", "**/*.png", "**/*.jpg")
    rename("(.)-staging(.)", "$1$2")
}

tasks.register<Copy>("copyAssets") {
    into(layout.buildDirectory.dir("inPlaceApp"))
    with(webAssetsSpec)
}

tasks.register<Zip>("distApp") {
    archiveFileName = "my-app-dist.zip"
    destinationDirectory = layout.buildDirectory.dir("dists")

    from(appClasses)
    with(webAssetsSpec)
}
```

build.gradle

```
CopySpec webAssetsSpec = copySpec {
    from 'src/main/webapp'
    include '**/*.html', '**/*.png', '**/*.jpg'
    rename '(.)-staging(.)', '$1$2'
}
```

```

tasks.register('copyAssets', Copy) {
    into layout.buildDirectory.dir("inPlaceApp")
    with webAssetsSpec
}

tasks.register('distApp', Zip) {
    archiveFileName = 'my-app-dist.zip'
    destinationDirectory = layout.buildDirectory.dir('dists')

    from appClasses
    with webAssetsSpec
}

```

Both the `copyAssets` and `distApp` tasks will process the static resources under `src/main/webapp`, as specified by `webAssetsSpec`.

NOTE

The configuration defined by `webAssetsSpec` will *not* apply to the app classes included by the `distApp` task. That's because `from appClasses` is its own child specification independent of `with webAssetsSpec`.

This can be confusing, so it's probably best to treat `with()` as an extra `from()` specification in the task. Hence, it doesn't make sense to define a standalone copy spec without at least one `from()` defined.

Suppose you encounter a scenario in which you want to apply the same copy configuration to *different* sets of files. In that case, you can share the configuration block directly without using `copySpec()`. Here's an example that has two independent tasks that happen to want to process image files only:

build.gradle.kts

```

val webAssetPatterns = Action<CopySpec> {
    include("**/*.html", "**/*.png", "**/*.jpg")
}

tasks.register<Copy>("copyAppAssets") {
    into(layout.buildDirectory.dir("inPlaceApp"))
    from("src/main/webapp", webAssetPatterns)
}

tasks.register<Zip>("archiveDistAssets") {
    archiveFileName = "distribution-assets.zip"
    destinationDirectory = layout.buildDirectory.dir("dists")

    from("distResources", webAssetPatterns)
}

```

build.gradle

```
def webAssetPatterns = {
    include '**/*.html', '**/*.png', '**/*.jpg'
}

tasks.register('copyAppAssets', Copy) {
    into layout.buildDirectory.dir("inPlaceApp")
    from 'src/main/webapp', webAssetPatterns
}

tasks.register('archiveDistAssets', Zip) {
    archiveFileName = 'distribution-assets.zip'
    destinationDirectory = layout.buildDirectory.dir('dists')

    from 'distResources', webAssetPatterns
}
```

In this case, we assign the copy configuration to its own variable and apply it to whatever `from()` specification we want. This doesn't just work for inclusions but also exclusions, file renaming, and file content filtering.

2. Using child specifications

If you only use a single copy spec, the file filtering and renaming will apply to *all* files copied. Sometimes, this is what you want, but not always. Consider the following example that copies files into a directory structure that a Java Servlet container can use to deliver a website:



This is not a straightforward copy as the `WEB-INF` directory and its subdirectories don't exist within the project, so they must be created during the copy. In addition, we only want HTML and image files going directly into the root folder — `build/explodedWar` — and only JavaScript files going into the `js` directory. We need separate filter patterns for those two sets of files.

The solution is to use *child specifications*, which can be applied to both `from()` and `into()` declarations. The following task definition does the necessary work:

build.gradle.kts

```
tasks.register<Copy>("nestedSpecs") {
    into(layout.buildDirectory.dir("explodedWar"))
    exclude("**/*staging*")
    from("src/dist") {
        include("**/*.html", "**/*.png", "**/*.jpg")
    }
    from(sourceSets.main.get().output) {
        into("WEB-INF/classes")
    }
    into("WEB-INF/lib") {
        from(configurations.runtimeClasspath)
    }
}
```

build.gradle

```
tasks.register('nestedSpecs', Copy) {
    into layout.buildDirectory.dir("explodedWar")
    exclude '**/*staging*'
    from('src/dist') {
        include '**/*.html', '**/*.png', '**/*.jpg'
    }
    from(sourceSets.main.output) {
        into 'WEB-INF/classes'
    }
    into('WEB-INF/lib') {
        from configurations.runtimeClasspath
    }
}
```

Notice how the `src/dist` configuration has a nested inclusion specification; it is the child copy spec. You can, of course, add content filtering and renaming here as required. A child copy spec is still a copy spec.

The above example also demonstrates how you can copy files into a subdirectory of the destination either by using a child `into()` on a `from()` or a child `from()` on an `into()`. Both approaches are acceptable, but you should create and follow a convention to ensure consistency across your build files.

NOTE

Don't get your `into()` specifications mixed up. For a normal copy, one to the filesystem rather than an archive, there should always be *one* "root" `into()` that specifies the overall destination directory of the copy. Any other `into()` should have

a child spec attached, and its path will be relative to the root `into()`.

One final thing to be aware of is that a child copy spec inherits its destination path, include patterns, exclude patterns, copy actions, name mappings, and filters from its parent. So, be careful where you place your configuration.

Using the `Sync` task

The `Sync` task, which extends the `Copy` task, copies the source files into the destination directory and then removes any files from the destination directory which it did not copy. It synchronizes the contents of a directory with its source.

This can be useful for doing things such as installing your application, creating an exploded copy of your archives, or maintaining a copy of the project's dependencies.

Here is an example that maintains a copy of the project's runtime dependencies in the `build/libs` directory:

build.gradle.kts

```
tasks.register<Sync>("libs") {  
    from(configurations["runtime"])  
    into(layout.buildDirectory.dir("libs"))  
}
```

build.gradle

```
tasks.register('libs', Sync) {  
    from configurations.runtime  
    into layout.buildDirectory.dir('libs')  
}
```

You can also perform the same function in your own tasks with the `Project.sync(org.gradle.api.Action)` method.

Using the `Copy` task

You can copy a file by creating an instance of Gradle's builtin `Copy` task and configuring it with the location of the file and where you want to put it.

This example mimics copying a generated report into a directory that will be packed into an archive, such as a ZIP or TAR:

build.gradle.kts

```
tasks.register<Copy>("copyReport") {  
    from(layout.buildDirectory.file("reports/my-report.pdf"))  
    into(layout.buildDirectory.dir("toArchive"))  
}
```

build.gradle

```
tasks.register('copyReport', Copy) {  
    from layout.buildDirectory.file("reports/my-report.pdf")  
    into layout.buildDirectory.dir("toArchive")  
}
```

The file and directory paths are then used to specify what file to copy using [Copy.from\(java.lang.Object...\)](#) and which directory to copy it to using [Copy.into\(java.lang.Object\)](#).

Although hard-coded paths make for simple examples, they make the build brittle. Using a reliable, single source of truth, such as a task or shared project property, is better. In the following modified example, we use a report task defined elsewhere that has the report's location stored in its `outputFile` property:

build.gradle.kts

```
tasks.register<Copy>("copyReport2") {  
    from(myReportTask.flatMap { it.outputFile })  
    into(archiveReportsTask.flatMap { it.dirToArchive })  
}
```

build.gradle

```
tasks.register('copyReport2', Copy) {  
    from myReportTask.outputFile  
    into archiveReportsTask.dirToArchive  
}
```

We have also assumed that the reports will be archived by `archiveReportsTask`, which provides us with the directory that will be archived and hence where we want to put the copies of the reports.

Copying multiple files

You can extend the previous examples to multiple files very easily by providing multiple arguments to `from()`:

build.gradle.kts

```
tasks.register<Copy>("copyReportsForArchiving") {  
    from(layout.buildDirectory.file("reports/my-report.pdf"),  
        layout.projectDirectory.file("src/docs/manual.pdf"))  
    into(layout.buildDirectory.dir("toArchive"))  
}
```

build.gradle

```
tasks.register('copyReportsForArchiving', Copy) {  
    from layout.buildDirectory.file("reports/my-report.pdf"), layout.  
        .projectDirectory.file("src/docs/manual.pdf")  
    into layout.buildDirectory.dir("toArchive")  
}
```

Two files are now copied into the archive directory.

You can also use multiple `from()` statements to do the same thing, as shown in the first example of the section [File copying in depth](#).

But what if you want to copy all the PDFs in a directory without specifying each one? To do this, attach inclusion and/or exclusion patterns to the copy specification. Here, we use a string pattern to include PDFs only:

build.gradle.kts

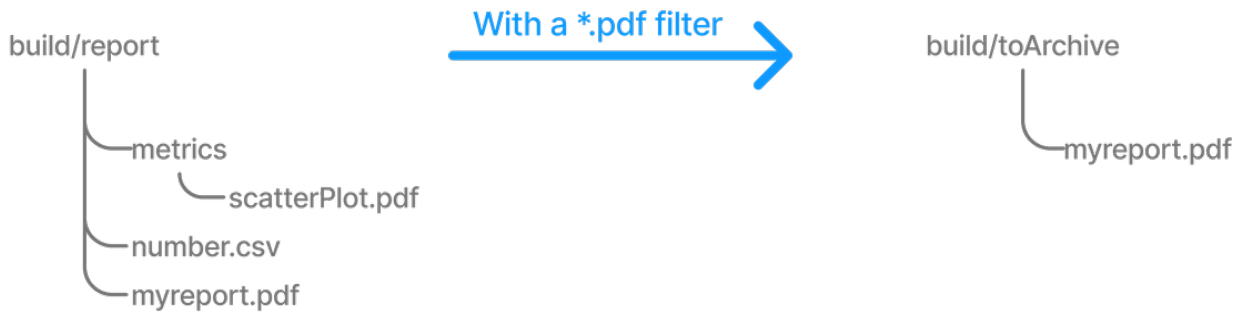
```
tasks.register<Copy>("copyPdfReportsForArchiving") {  
    from(layout.buildDirectory.dir("reports"))  
    include("*.pdf")  
    into(layout.buildDirectory.dir("toArchive"))  
}
```

build.gradle

```
tasks.register('copyPdfReportsForArchiving', Copy) {  
    from layout.buildDirectory.dir("reports")
```

```
include "*.pdf"
into layout.buildDirectory.dir("toArchive")
}
```

One thing to note, as demonstrated in the following diagram, is that only the PDFs that reside directly in the `reports` directory are copied:



You can include files in subdirectories by using an Ant-style glob pattern (`**/*`), as done in this updated example:

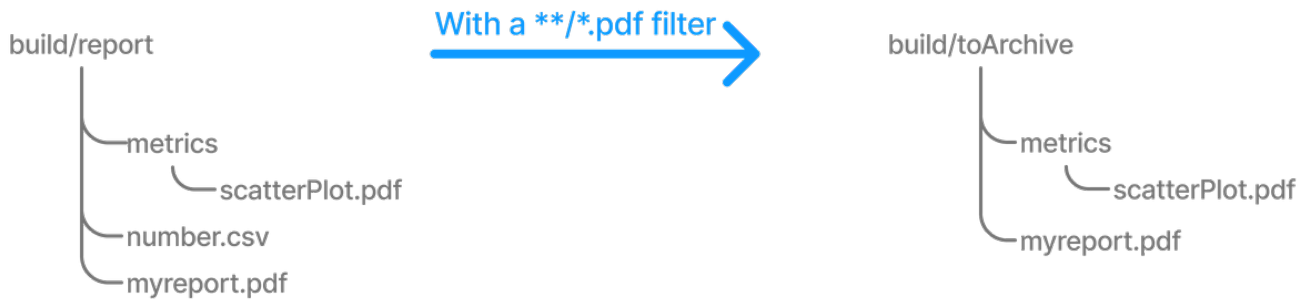
build.gradle.kts

```
tasks.register<Copy>("copyAllPdfReportsForArchiving") {
    from(layout.buildDirectory.dir("reports"))
    include("**/*.pdf")
    into(layout.buildDirectory.dir("toArchive"))
}
```

build.gradle

```
tasks.register('copyAllPdfReportsForArchiving', Copy) {
    from layout.buildDirectory.dir("reports")
    include "**/*.pdf"
    into layout.buildDirectory.dir("toArchive")
}
```

This task has the following effect:



Remember that a deep filter like this has the side effect of copying the directory structure below **reports** and the files. If you want to copy the files without the directory structure, you must use an explicit `fileTree(dir) { includes }.files` expression.

Copying directory hierarchies

You may need to copy files as well as the directory structure in which they reside. This is the default behavior when you specify a directory as the `from()` argument, as demonstrated by the following example that copies everything in the **reports** directory, including all its subdirectories, to the destination:

build.gradle.kts

```
tasks.register<Copy>("copyReportsDirForArchiving") {
    from(layout.buildDirectory.dir("reports"))
    into(layout.buildDirectory.dir("toArchive"))
}
```

build.gradle

```
tasks.register('copyReportsDirForArchiving', Copy) {
    from layout.buildDirectory.dir("reports")
    into layout.buildDirectory.dir("toArchive")
}
```

The key aspect that users need help with is controlling how much of the directory structure goes to the destination. In the above example, do you get a **toArchive/reports** directory, or does everything in **reports** go straight into **toArchive**? The answer is the latter. If a directory is part of the `from()` path, then it *won't* appear in the destination.

So how do you ensure that **reports** itself is copied across, but not any other directory in `${layout.buildDirectory}`? The answer is to add it as an include pattern:

build.gradle.kts

```
tasks.register<Copy>("copyReportsDirForArchiving2") {  
    from(layout.buildDirectory) {  
        include("reports/**")  
    }  
    into(layout.buildDirectory.dir("toArchive"))  
}
```

build.gradle

```
tasks.register('copyReportsDirForArchiving2', Copy) {  
    from(layout.buildDirectory) {  
        include "reports/**"  
    }  
    into layout.buildDirectory.dir("toArchive")  
}
```

You'll get the same behavior as before except with one extra directory level in the destination, i.e., `toArchive/reports`.

One thing to note is how the `include()` directive applies only to the `from()`, whereas the directive in the previous section applied to the whole task. These different [levels of granularity](#) in the copy specification allow you to handle most requirements that you will come across easily.

Understanding file copying

The basic process of copying files in Gradle is a simple one:

- Define a task of type [Copy](#)
- Specify which files (and potentially directories) to copy
- Specify a destination for the copied files

But this apparent simplicity hides a rich API that allows fine-grained control of which files are copied, where they go, and what happens to them as they are copied — renaming of the files and token substitution of file content are both possibilities, for example.

Let's start with the last two items on the list, which involve [CopySpec](#). The [CopySpec](#) interface, which the [Copy](#) task implements, offers:

- A [CopySpec.from\(java.lang.Object...\)](#) method to define what to copy
- An [CopySpec.into\(java.lang.Object\)](#) method to define the destination

[CopySpec](#) has several additional methods that allow you to control the copying process, but these

two are the only required ones. `into()` is straightforward, requiring a directory path as its argument in any form supported by the `Project.file(java.lang.Object)` method. The `from()` configuration is far more flexible.

Not only does `from()` accept multiple arguments, it also allows several different types of argument. For example, some of the most common types are:

- A `String` — treated as a file path or, if it starts with "file://", a file URI
- A `File` — used as a file path
- A `FileCollection` or `FileTree` — all files in the collection are included in the copy
- A task — the files or directories that form a task's `defined outputs` are included

In fact, `from()` accepts all the same arguments as `Project.files(java.lang.Object...)` so see that method for a more detailed list of acceptable types.

Something else to consider is what type of thing a file path refers to:

- A file — the file is copied as is
- A directory — this is effectively treated as a file tree: everything in it, including subdirectories, is copied. However, the directory itself is not included in the copy.
- A non-existent file — the path is ignored

Here is an example that uses multiple `from()` specifications, each with a different argument type. You will probably also notice that `into()` is configured lazily using a closure (in Groovy) or a Provider (in Kotlin) — a technique that also works with `from()`:

build.gradle.kts

```
tasks.register<Copy>("anotherCopyTask") {
    // Copy everything under src/main/webapp
    from("src/main/webapp")
    // Copy a single file
    from("src/staging/index.html")
    // Copy the output of a task
    from(copyTask)
    // Copy the output of a task using Task outputs explicitly.
    from(tasks["copyTaskWithPatterns"].outputs)
    // Copy the contents of a Zip file
    from(zipTree("src/main/assets.zip"))
    // Determine the destination directory later
    into({ getDestDir() })
}
```

build.gradle

```
tasks.register('anotherCopyTask', Copy) {
    // Copy everything under src/main/webapp
    from 'src/main/webapp'
    // Copy a single file
    from 'src/staging/index.html'
    // Copy the output of a task
    from copyTask
    // Copy the output of a task using Task outputs explicitly.
    from copyTaskWithPatterns.outputs
    // Copy the contents of a Zip file
    from zipTree('src/main/assets.zip')
    // Determine the destination directory later
    into { getDestDir() }
}
```

Note that the lazy configuration of `into()` is different from a [child specification](#), even though the syntax is similar. Keep an eye on the number of arguments to distinguish between them.

Copying files in your own tasks

WARNING

Using the `Project.copy` method at execution time, as described here, is not compatible with the [configuration cache](#). A possible solution is to implement the task as a proper class and use `FileSystemOperations.copy` method instead, as described in the [configuration cache chapter](#).

Occasionally, you want to copy files or directories as *part* of a task. For example, a custom archiving task based on an unsupported archive format might want to copy files to a temporary directory before they are archived. You still want to take advantage of Gradle's copy API without introducing an extra `Copy` task.

The solution is to use the `Project.copy(org.gradle.api.Action)` method. Configuring it with a copy spec works like the `Copy` task. Here's a trivial example:

build.gradle.kts

```
tasks.register("copyMethod") {
    doLast {
        copy {
            from("src/main/webapp")
            into(layout.buildDirectory.dir("explodedWar"))
            include("**/*.html")
            include("**/*.jsp")
        }
    }
}
```

```
}
```

build.gradle

```
tasks.register('copyMethod') {
    doLast {
        copy {
            from 'src/main/webapp'
            into layout.buildDirectory.dir('explodedWar')
            include '**/*.html'
            include '**/*.jsp'
        }
    }
}
```

The above example demonstrates the basic syntax and also highlights two major limitations of using the `copy()` method:

1. The `copy()` method is not [incremental](#). The example's `copyMethod` task will *always* execute because it has no information about what files make up the task's inputs. You have to define the task inputs and outputs manually.
2. Using a task as a copy source, i.e., as an argument to `from()`, won't create an automatic task dependency between your task and that copy source. As such, if you use the `copy()` method as part of a task action, you must explicitly declare all inputs and outputs to get the correct behavior.

The following example shows how to work around these limitations using the [dynamic API for task inputs and outputs](#):

build.gradle.kts

```
tasks.register("copyMethodWithExplicitDependencies") {
    // up-to-date check for inputs, plus add copyTask as dependency
    inputs.files(copyTask)
        .withPropertyName("inputs")
        .withPathSensitivity(PathSensitivity.RELATIVE)
    outputs.dir("some-dir") // up-to-date check for outputs
        .withPropertyName("outputDir")
    doLast {
        copy {
            // Copy the output of copyTask
            from(copyTask)
            into("some-dir")
        }
    }
}
```

```
}  
}
```

build.gradle

```
tasks.register('copyMethodWithExplicitDependencies') {  
    // up-to-date check for inputs, plus add copyTask as dependency  
    inputs.files(copyTask)  
        .withPropertyName("inputs")  
        .withPathSensitivity(PathSensitivity.RELATIVE)  
    outputs.dir('some-dir') // up-to-date check for outputs  
        .withPropertyName("outputDir")  
    doLast {  
        copy {  
            // Copy the output of copyTask  
            from copyTask  
            into 'some-dir'  
        }  
    }  
}
```

These limitations make it preferable to use the `Copy` task wherever possible because of its built-in support for incremental building and task dependency inference. That is why the `copy()` method is intended for use by `custom tasks` that need to copy files as part of their function. Custom tasks that use the `copy()` method should declare the necessary inputs and outputs relevant to the copy action.

Renaming files

Renaming files in Gradle can be done using the `CopySpec` API, which provides methods for renaming files as they are copied.

Using `Copy.rename()`

If the files used and generated by your builds sometimes don't have names that suit, you can rename those files as you copy them. Gradle allows you to do this as part of a copy specification using the `rename()` configuration.

The following example removes the "-staging" marker from the names of any files that have it:

build.gradle.kts

```
tasks.register<Copy>("copyFromStaging") {  
    from("src/main/webapp")  
    into(layout.buildDirectory.dir("explodedWar"))  
}
```

```
    rename("(.)-staging(.+)", "$1$2")
}
```

build.gradle

```
tasks.register('copyFromStaging', Copy) {
    from "src/main/webapp"
    into layout.buildDirectory.dir('explodedWar')

    rename '(.)-staging(.+)', '$1$2'
}
```

As in the above example, you can use regular expressions for this or closures that use more complex logic to determine the target filename. For example, the following task truncates filenames:

build.gradle.kts

```
tasks.register<Copy>("copyWithTruncate") {
    from(layout.buildDirectory.dir("reports"))
    rename { filename: String ->
        if (filename.length > 10) {
            filename.slice(0..7) + "~" + filename.length
        }
        else filename
    }
    into(layout.buildDirectory.dir("toArchive"))
}
```

build.gradle

```
tasks.register('copyWithTruncate', Copy) {
    from layout.buildDirectory.dir("reports")
    rename { String filename ->
        if (filename.size() > 10) {
            return filename[0..7] + "~" + filename.size()
        }
        else return filename
    }
    into layout.buildDirectory.dir("toArchive")
}
```

As with filtering, you can also rename a subset of files by configuring it as part of a child specification on a `from()`.

Using `Copyspec.rename{}`

The [example of how to rename files on copy](#) gives you most of the information you need to perform this operation. It demonstrates the two options for renaming:

1. Using a regular expression
2. Using a closure

Regular expressions are a flexible approach to renaming, particularly as Gradle supports regex groups that allow you to remove and replace parts of the source filename. The following example shows how you can remove the string "-staging" from any filename that contains it using a simple regular expression:

build.gradle.kts

```
tasks.register<Copy>("rename") {
    from("src/main/webapp")
    into(layout.buildDirectory.dir("explodedWar"))
    // Use a regular expression to map the file name
    rename("(.)-staging(.+)", "$1$2")
    rename("(.)-staging(.+)".toRegex().pattern, "$1$2")
    // Use a closure to convert all file names to upper case
    rename { fileName: String ->
        fileName.toUpperCase()
    }
}
```

build.gradle

```
tasks.register('rename', Copy) {
    from 'src/main/webapp'
    into layout.buildDirectory.dir('explodedWar')
    // Use a regular expression to map the file name
    rename '(.)-staging(.+)', '$1$2'
    rename /(.)-staging(.+)/, '$1$2'
    // Use a closure to convert all file names to upper case
    rename { String fileName ->
        fileName.toUpperCase()
    }
}
```

You can use any regular expression supported by the Java [Pattern](#) class and the substitution string. The second argument of `rename()` works on the same principles as the `Matcher.appendReplacement()` method.

Regular expressions in Groovy build scripts

There are two common issues people come across when using regular expressions in this context:

1. If you use a slashy string (those delimited by '/') for the first argument, you *must* include the parentheses for `rename()` as shown in the above example.
2. It's safest to use single quotes for the second argument, otherwise you need to escape the '\$' in group substitutions, i.e. `"\${1}\${2}"`.

The first is a minor inconvenience, but slashy strings have the advantage that you don't have to escape backslash ('\') characters in the regular expression. The second issue stems from Groovy's support for embedded expressions using `${ }` syntax in double-quoted and slashy strings.

The closure syntax for `rename()` is straightforward and can be used for any requirements that simple regular expressions can't handle. You're given a file's name, and you return a new name for that file or `null` if you don't want to change the name. Be aware that the closure will be executed for every file copied, so try to avoid expensive operations where possible.

Filtering files

Filtering files in Gradle involves selectively including or excluding files based on certain criteria.

Using `CopySpec.include()` and `CopySpec.exclude()`

You can apply filtering in any copy specification through the `CopySpec.include(java.lang.String...)` and `CopySpec.exclude(java.lang.String...)` methods.

These methods are typically used with Ant-style include or exclude patterns, as described in [PatternFilterable](#).

You can also perform more complex logic by using a closure that takes a [FileTreeElement](#) and returns `true` if the file should be included or `false` otherwise. The following example demonstrates both forms, ensuring that only `.html` and `.jsp` files are copied, except for those `.html` files with the word "DRAFT" in their content:

build.gradle.kts

```
tasks.register<Copy>("copyTaskWithPatterns") {
    from("src/main/webapp")
    into(layout.buildDirectory.dir("explodedWar"))
    include("**/*.html")
    include("**/*.jsp")
    exclude { details: FileTreeElement ->
        details.file.name.endsWith(".html") &&
        details.file.readText().contains("DRAFT")
    }
}
```

```
}  
}
```

build.gradle

```
tasks.register('copyTaskWithPatterns', Copy) {  
    from 'src/main/webapp'  
    into layout.buildDirectory.dir('explodedWar')  
    include '**/*.html'  
    include '**/*.jsp'  
    exclude { FileTreeElement details ->  
        details.file.name.endsWith('.html') &&  
            details.file.text.contains('DRAFT')  
    }  
}
```

A question you may ask yourself at this point is what happens when inclusion and exclusion patterns overlap? Which pattern wins? Here are the basic rules:

- If there are no explicit inclusions or exclusions, everything is included
- If at least one inclusion is specified, only files and directories matching the patterns are included
- Any exclusion pattern overrides any inclusions, so if a file or directory matches at least one exclusion pattern, it won't be included, regardless of the inclusion patterns

Bear these rules in mind when creating combined inclusion and exclusion specifications so that you end up with the exact behavior you want.

Note that the inclusions and exclusions in the above example will apply to *all* `from()` configurations. If you want to apply filtering to a subset of the copied files, you'll need to use [child specifications](#).

Filtering file content

Filtering file content in Gradle involves replacing placeholders or tokens in files with dynamic values.

Using `CopySpec.filter()`

Transforming the content of files while they are being copied involves basic templating that uses token substitution, removal of lines of text, or even more complex filtering using a full-blown template engine.

The following example demonstrates several forms of filtering, including token substitution using the `CopySpec.expand(java.util.Map)` method and another using `CopySpec.filter(java.lang.Class)` with an [Ant filter](#):

build.gradle.kts

```
import org.apache.tools.ant.filters.FixCrLfFilter
import org.apache.tools.ant.filters.ReplaceTokens
tasks.register<Copy>("filter") {
    from("src/main/webapp")
    into(layout.buildDirectory.dir("explodedWar"))
    // Substitute property tokens in files
    expand("copyright" to "2009", "version" to "2.3.1")
    // Use some of the filters provided by Ant
    filter(FixCrLfFilter::class)
    filter(ReplaceTokens::class, "tokens" to mapOf("copyright" to "2009",
"version" to "2.3.1"))
    // Use a closure to filter each line
    filter { line: String ->
        "[$line]"
    }
    // Use a closure to remove lines
    filter { line: String ->
        if (line.startsWith('-')) null else line
    }
    filteringCharset = "UTF-8"
}
```

build.gradle

```
import org.apache.tools.ant.filters.FixCrLfFilter
import org.apache.tools.ant.filters.ReplaceTokens

tasks.register('filter', Copy) {
    from 'src/main/webapp'
    into layout.buildDirectory.dir('explodedWar')
    // Substitute property tokens in files
    expand(copyright: '2009', version: '2.3.1')
    // Use some of the filters provided by Ant
    filter(FixCrLfFilter)
    filter(ReplaceTokens, tokens: [copyright: '2009', version: '2.3.1'])
    // Use a closure to filter each line
    filter { String line ->
        "[$line]"
    }
    // Use a closure to remove lines
    filter { String line ->
        line.startsWith('-') ? null : line
    }
    filteringCharset = 'UTF-8'
}
```

```
}
```

The `filter()` method has two variants, which behave differently:

- one takes a `FilterReader` and is designed to work with Ant filters, such as `ReplaceTokens`
- one takes a closure or `Transformer` that defines the transformation for each line of the source file

Note that both variants assume the source files are text-based. When you use the `ReplaceTokens` class with `filter()`, you create a template engine that replaces tokens of the form `@tokenName@` (the Ant-style token) with values you define.

Using `CopySpec.expand()`

The `expand()` method treats the source files as `Groovy templates`, which evaluates and expands expressions of the form `${expression}`.

You can pass in property names and values that are then expanded in the source files. `expand()` allows for more than basic token substitution as the embedded expressions are full-blown Groovy expressions.

NOTE

Specifying the character set when reading and writing the file is good practice. Otherwise, the transformations won't work properly for non-ASCII text. You configure the character set with the `CopySpec.setFilteringCharset(String)` property. If it's not specified, the JVM default character set is used, which will likely differ from the one you want.

Setting file permissions

Setting file permissions in Gradle involves specifying the permissions for files or directories created or modified during the build process.

Using `CopySpec.filePermissions{}`

For any `CopySpec` involved in copying files, may it be the `Copy` task itself, or any child specifications, you can explicitly set the permissions the destination files will have via the `CopySpec.filePermissions {}` configurations block.

Using `CopySpec.dirPermissions{}`

You can do the same for directories too, independently of files, via the `CopySpec.dirPermissions {}` configurations block.

NOTE

Not setting permissions explicitly will preserve the permissions of the original files or directories.

build.gradle.kts

```
tasks.register<Copy>("permissions") {
    from("src/main/webapp")
    into(layout.buildDirectory.dir("explodedWar"))
    filePermissions {
        user {
            read = true
            execute = true
        }
        other.execute = false
    }
    dirPermissions {
        unix("r-xr-x---")
    }
}
```

build.gradle

```
tasks.register('permissions', Copy) {
    from 'src/main/webapp'
    into layout.buildDirectory.dir('explodedWar')
    filePermissions {
        user {
            read = true
            execute = true
        }
        other.execute = false
    }
    dirPermissions {
        unix('r-xr-x---')
    }
}
```

For a detailed description of file permissions, see [FilePermissions](#) and [UserClassFilePermissions](#). For details on the convenience method used in the samples, see [ConfigurableFilePermissions.unix\(String\)](#).

Using empty configuration blocks for file or directory permissions still sets them explicitly, just to fixed default values. Everything inside one of these configuration blocks is relative to the default values. Default permissions differ for files and directories:

- **file**: read & write for **owner**, read for **group**, read for **other** (0644, **rw-r--r--**)
- **directory**: read, write & execute for **owner**, read & execute for **group**, read & execute for **other**

(0755, **rwxr-xr-x**)

Moving files and directories

Moving files and directories in Gradle is a straightforward process that can be accomplished using several APIs. When implementing file-moving logic in your build scripts, it's important to consider file paths, conflicts, and task dependencies.

Using `File.renameTo()`

`File.renameTo()` is a method in Java (and by extension, in Gradle's Groovy DSL) used to rename or move a file or directory. When you call `renameTo()` on a `File` object, you provide another `File` object representing the new name or location. If the operation is successful, `renameTo()` returns `true`; otherwise, it returns `false`.

It's important to note that `renameTo()` has some limitations and platform-specific behavior.

In this example, the `moveFile` task uses the `Copy` task type to specify the source and destination directories. Inside the `doLast` closure, it uses `File.renameTo()` to move the file from the source directory to the destination directory:

```
task moveFile {
    doLast {
        def sourceFile = file('source.txt')
        def destFile = file('destination/new_name.txt')

        if (sourceFile.renameTo(destFile)) {
            println "File moved successfully."
        }
    }
}
```

Using the `Copy` task

In this example, the `moveFile` task copies the file `source.txt` to the destination directory and renames it to `new_name.txt` in the process. This achieves a similar effect to moving a file.

```
task moveFile(type: Copy) {
    from 'source.txt'
    into 'destination'
    rename { fileName ->
        'new_name.txt'
    }
}
```

Deleting files and directories

Deleting files and directories in Gradle involves removing them from the file system.

Using the **Delete** task

You can easily delete files and directories using the [Delete](#) task. You must specify which files and directories to delete in a way supported by the [Project.files\(java.lang.Object...\)](#) method.

For example, the following task deletes the entire contents of a build's output directory:

build.gradle.kts

```
tasks.register<Delete>("myClean") {  
    delete(buildDir)  
}
```

build.gradle

```
tasks.register('myClean', Delete) {  
    delete buildDir  
}
```

If you want more control over which files are deleted, you can't use inclusions and exclusions the same way you use them for copying files. Instead, you use the built-in filtering mechanisms of [FileCollection](#) and [FileTree](#). The following example does just that to clear out temporary files from a source directory:

build.gradle.kts

```
tasks.register<Delete>("cleanTempFiles") {  
    delete(fileTree("src").matching {  
        include("**/*.tmp")  
    })  
}
```

build.gradle

```
tasks.register('cleanTempFiles', Delete) {  
    delete fileTree("src").matching {  
        include "**/*.tmp"  
    }  
}
```

Using `Project.delete()`

The `Project.delete(org.gradle.api.Action)` method can delete files and directories.

This method takes one or more arguments representing the files or directories to be deleted.

For example, the following task deletes the entire contents of a build's output directory:

build.gradle.kts

```
tasks.register<Delete>("myClean") {  
    delete(buildDir)  
}
```

build.gradle

```
tasks.register('myClean', Delete) {  
    delete buildDir  
}
```

If you want more control over which files are deleted, you can't use inclusions and exclusions the same way you use them for copying files. Instead, you use the built-in filtering mechanisms of `FileCollection` and `FileTree`. The following example does just that to clear out temporary files from a source directory:

build.gradle.kts

```
tasks.register<Delete>("cleanTempFiles") {  
    delete(fileTree("src").matching {  
        include("**/*.tmp")  
    })  
}
```

build.gradle

```
tasks.register('cleanTempFiles', Delete) {  
    delete fileTree("src").matching {  
        include "**/*.tmp"  
    }  
}
```

Creating archives

From the perspective of Gradle, packing files into an archive is effectively a copy in which the destination is the archive file rather than a directory on the file system. Creating archives looks a lot like copying, with all the same features.

Using the `Zip`, `Tar`, or `Jar` task

The simplest case involves archiving the entire contents of a directory, which this example demonstrates by creating a ZIP of the `toArchive` directory:

build.gradle.kts

```
tasks.register<Zip>("packageDistribution") {
    archiveFileName = "my-distribution.zip"
    destinationDirectory = layout.buildDirectory.dir("dist")

    from(layout.buildDirectory.dir("toArchive"))
}
```

build.gradle

```
tasks.register('packageDistribution', Zip) {
    archiveFileName = "my-distribution.zip"
    destinationDirectory = layout.buildDirectory.dir('dist')

    from layout.buildDirectory.dir("toArchive")
}
```

Notice how we specify the destination and name of the archive instead of an `into()`: both are required. You often won't see them explicitly set because most projects apply the [Base Plugin](#). It provides some conventional values for those properties.

The following example demonstrates this; you can learn more about the conventions in the [archive naming](#) section.

Each type of archive has its own task type, the most common ones being [Zip](#), [Tar](#) and [Jar](#). They all share most of the configuration options of `Copy`, including filtering and renaming.

One of the most common scenarios involves copying files into specified archive subdirectories. For example, let's say you want to package all PDFs into a `docs` directory in the archive's root. This `docs` directory doesn't exist in the source location, so you must create it as part of the archive. You do this by adding an `into()` declaration for just the PDFs:

build.gradle.kts

```
plugins {  
    base  
}  
  
version = "1.0.0"  
  
tasks.register<Zip>("packageDistribution") {  
    from(layout.buildDirectory.dir("toArchive")) {  
        exclude("**/*.pdf")  
    }  
  
    from(layout.buildDirectory.dir("toArchive")) {  
        include("**/*.pdf")  
        into("docs")  
    }  
}
```

build.gradle

```
plugins {  
    id 'base'  
}  
  
version = "1.0.0"  
  
tasks.register('packageDistribution', Zip) {  
    from(layout.buildDirectory.dir("toArchive")) {  
        exclude "**/*.pdf"  
    }  
  
    from(layout.buildDirectory.dir("toArchive")) {  
        include "**/*.pdf"  
        into "docs"  
    }  
}
```

As you can see, you can have multiple `from()` declarations in a copy specification, each with its own configuration. See [Using child copy specifications](#) for more information on this feature.

Understanding archive creation

Archives are essentially self-contained file systems, and Gradle treats them as such. This is why working with archives is similar to working with files and directories.

Out of the box, Gradle supports the creation of ZIP and TAR archives and, by extension, Java's JAR, WAR, and EAR formats—Java's archive formats are all ZIPs. Each of these formats has a corresponding task type to create them: [Zip](#), [Tar](#), [Jar](#), [War](#), and [Ear](#). These all work the same way and are based on copy specifications, just like the [Copy](#) task.

Creating an archive file is essentially a file copy in which the destination is implicit, i.e., the archive file itself. Here is a basic example that specifies the path and name of the target archive file:

build.gradle.kts

```
tasks.register<Zip>("packageDistribution") {  
    archiveFileName = "my-distribution.zip"  
    destinationDirectory = layout.buildDirectory.dir("dist")  
  
    from(layout.buildDirectory.dir("toArchive"))  
}
```

build.gradle

```
tasks.register('packageDistribution', Zip) {  
    archiveFileName = "my-distribution.zip"  
    destinationDirectory = layout.buildDirectory.dir('dist')  
  
    from layout.buildDirectory.dir("toArchive")  
}
```

The full power of copy specifications is available to you when creating archives, which means you can do content filtering, file renaming, or anything else covered in the previous section. A common requirement is copying files into subdirectories of the archive that don't exist in the source folders, something that can be achieved with [into\(\)](#) [child specifications](#).

Gradle allows you to create as many archive tasks as you want, but it's worth considering that many convention-based plugins provide their own. For example, the Java plugin adds a [jar](#) task for packaging a project's compiled classes and resources in a JAR. Many of these plugins provide sensible conventions for the names of archives and the copy specifications used. We recommend you use these tasks wherever you can rather than overriding them with your own.

Naming archives

Gradle has several conventions around the naming of archives and where they are created based on the plugins your project uses. The main convention is provided by the [Base Plugin](#), which defaults to creating archives in the `layout.buildDirectory.dir("distributions")` directory and typically uses archive names of the form `[projectName]-[version].[type]`.

The following example comes from a project named `archive-naming`, hence the `myZip` task creates an archive named `archive-naming-1.0.zip`:

build.gradle.kts

```
plugins {
    base
}

version = "1.0"

tasks.register<Zip>("myZip") {
    from("somedir")
    val projectDir = layout.projectDirectory.asFile
    doLast {
        println(archiveFileName.get())
        println(destinationDirectory.get().asFile.relativeTo(projectDir))
        println(archiveFile.get().asFile.relativeTo(projectDir))
    }
}
```

build.gradle

```
plugins {
    id 'base'
}

version = 1.0

tasks.register('myZip', Zip) {
    from 'somedir'
    File projectDir = layout.projectDirectory.asFile
    doLast {
        println archiveFileName.get()
        println projectDir.relativePath(destinationDirectory.get().asFile)
        println projectDir.relativePath(archiveFile.get().asFile)
    }
}
```

```
$ gradle -q myZip
archive-naming-1.0.zip
build/distributions
build/distributions/archive-naming-1.0.zip
```

Note that the archive name does *not* derive from the task's name that creates it.

If you want to change the name and location of a generated archive file, you can provide values for the corresponding task's `archiveFileName` and `destinationDirectory` properties. These override any conventions that would otherwise apply.

Alternatively, you can make use of the default archive name pattern provided by `AbstractArchiveTask.getArchiveFileName()`: `[archiveBaseName]-[archiveAppendix]-[archiveVersion]-[archiveClassifier].[archiveExtension]`. You can set each of these properties on the task separately. Note that the Base Plugin uses the convention of the project name for `archiveBaseName`, project version for `archiveVersion`, and the archive type for `archiveExtension`. It does not provide values for the other properties.

This example — from the same project as the one above — configures just the `archiveBaseName` property, overriding the default value of the project name:

build.gradle.kts

```
tasks.register<Zip>("myCustomZip") {
    archiveBaseName = "customName"
    from("somedir")

    doLast {
        println(archiveFileName.get())
    }
}
```

build.gradle

```
tasks.register('myCustomZip', Zip) {
    archiveBaseName = 'customName'
    from 'somedir'

    doLast {
        println archiveFileName.get()
    }
}
```

```
$ gradle -q myCustomZip
customName-1.0.zip
```

You can also override the default `archiveBaseName` value for *all* the archive tasks in your build by using the *project* property `archivesBaseName`, as demonstrated by the following example:

build.gradle.kts

```
plugins {
    base
}

version = "1.0"

base {
    archivesName = "gradle"
    distsDirectory = layout.buildDirectory.dir("custom-dist")
    libsDirectory = layout.buildDirectory.dir("custom-libs")
}

val myZip by tasks.registering(Zip::class) {
    from("somedir")
}

val myOtherZip by tasks.registering(Zip::class) {
    archiveAppendix = "wrapper"
    archiveClassifier = "src"
    from("somedir")
}

tasks.register("echoNames") {
    val projectNameString = project.name
    val archiveFileName = myZip.flatMap { it.archiveFileName }
    val myOtherArchiveFileName = myOtherZip.flatMap { it.archiveFileName }
    doLast {
        println("Project name: $projectNameString")
        println(archiveFileName.get())
        println(myOtherArchiveFileName.get())
    }
}
```

build.gradle

```
plugins {
    id 'base'
}

version = 1.0
base {
    archivesName = "gradle"
    distsDirectory = layout.buildDirectory.dir('custom-dist')
    libsDirectory = layout.buildDirectory.dir('custom-libs')
}
```

```

def myZip = tasks.register('myZip', Zip) {
    from 'somedir'
}

def myOtherZip = tasks.register('myOtherZip', Zip) {
    archiveAppendix = 'wrapper'
    archiveClassifier = 'src'
    from 'somedir'
}

tasks.register('echoNames') {
    def projectNameString = project.name
    def archiveFileName = myZip.flatMap { it.archiveFileName }
    def myOtherArchiveFileName = myOtherZip.flatMap { it.archiveFileName }
    doLast {
        println "Project name: $projectNameString"
        println archiveFileName.get()
        println myOtherArchiveFileName.get()
    }
}

```

```

$ gradle -q echoNames
Project name: archives-changed-base-name
gradle-1.0.zip
gradle-wrapper-1.0-src.zip

```

You can find all the possible archive task properties in the API documentation for [AbstractArchiveTask](#). Still, we have also summarized the main ones here:

archiveFileName — **Property<String>**, **default:** **archiveBaseName-archiveAppendix-archiveVersion-archiveClassifier.archiveExtension**

The complete file name of the generated archive. If any of the properties in the default value are empty, their '-' separator is dropped.

archiveFile — **Provider<RegularFile>**, **read-only**, **default:** **destinationDirectory/archiveFileName**

The absolute file path of the generated archive.

destinationDirectory — **DirectoryProperty**, **default:** **depends on archive type**

The target directory in which to put the generated archive. By default, JARs and WARs go into `layout.buildDirectory.dir("libs")`. ZIPs and TARs go into `layout.buildDirectory.dir("distributions")`.

archiveBaseName — **Property<String>**, **default:** **project.name**

The base name portion of the archive file name, typically a project name or some other descriptive name for what it contains.

archiveAppendix — **Property<String>**, **default: null**

The appendix portion of the archive file name that comes immediately after the base name. It is typically used to distinguish between different forms of content, such as code and docs, or a minimal distribution versus a full or complete one.

archiveVersion — **Property<String>**, **default: project.version**

The version portion of the archive file name, typically in the form of a normal project or product version.

archiveClassifier — **Property<String>**, **default: null**

The classifier portion of the archive file name. Often used to distinguish between archives that target different platforms.

archiveExtension — **Property<String>**, **default: depends on archive type and compression type**

The filename extension for the archive. By default, this is set based on the archive task type and the compression type (if you're creating a TAR). Will be one of: `zip`, `jar`, `war`, `tar`, `tgz` or `tbz2`. You can of course set this to a custom extension if you wish.

Sharing content between multiple archives

As described in the [CopySpec](#) section above, you can use the `Project.copySpec(org.gradle.api.Action)` method to share content between archives.

Using archives as file trees

An archive is a directory and file hierarchy packed into a single file. In other words, it's a special case of a file tree, and that's exactly how Gradle treats archives.

Instead of using the `fileTree()` method, which only works on normal file systems, you use the `Project.zipTree(java.lang.Object)` and `Project.tarTree(java.lang.Object)` methods to wrap archive files of the corresponding type (note that JAR, WAR and EAR files are ZIPs). Both methods return `FileTree` instances that you can then use in the same way as normal file trees. For example, you can extract some or all of the files of an archive by copying its contents to some directory on the file system. Or you can merge one archive into another.

Here are some simple examples of creating archive-based file trees:

build.gradle.kts

```
// Create a ZIP file tree using path
val zip: FileTree = zipTree("someFile.zip")

// Create a TAR file tree using path
val tar: FileTree = tarTree("someFile.tar")

// tar tree attempts to guess the compression based on the file extension
// however if you must specify the compression explicitly you can:
```

```
val someTar: FileTree = tarTree(resources.gzip("someTar.ext"))
```

build.gradle

```
// Create a ZIP file tree using path
FileTree zip = zipTree('someFile.zip')

// Create a TAR file tree using path
FileTree tar = tarTree('someFile.tar')

//tar tree attempts to guess the compression based on the file extension
//however if you must specify the compression explicitly you can:
FileTree someTar = tarTree(resources.gzip('someTar.ext'))
```

You can see a practical example of extracting an archive file [in the unpacking archives section](#) below.

Using **AbstractArchiveTask** for reproducible builds

Sometimes it's desirable to recreate archives exactly the same, byte for byte, on different machines. You want to be sure that building an artifact from source code produces the same result no matter when and where it is built. This is necessary for projects like [reproducible-builds.org](#).

Reproducing the same byte-for-byte archive poses some challenges since the order of the files in an archive is influenced by the underlying file system. Each time a ZIP, TAR, JAR, WAR or EAR is built from source, the order of the files inside the archive may change. Files that only have a different timestamp also causes differences in archives from build to build.

All **AbstractArchiveTask** (e.g. Jar, Zip) tasks shipped with Gradle include support for producing reproducible archives.

For example, to make a **Zip** task reproducible you need to set **Zip.isReproducibleFileOrder()** to **true** and **Zip.isPreserveFileTimestamps()** to **false**. In order to make all archive tasks in your build reproducible, consider adding the following configuration to your build file:

build.gradle.kts

```
tasks.withType<AbstractArchiveTask>().configureEach {
    isPreserveFileTimestamps = false
    isReproducibleFileOrder = true
}
```

build.gradle

```
tasks.withType(AbstractArchiveTask).configureEach {
    preserveFileTimestamps = false
    reproducibleFileOrder = true
}
```

Often you will want to publish an archive, so that it is usable from another project.

Unpacking archives

Archives are effectively self-contained file systems, so unpacking them is a case of copying the files from that file system onto the local file system — or even into another archive. Gradle enables this by providing some wrapper functions that make archives available as hierarchical collections of files ([file trees](#)).

Using `Project.zipTree` and `Project.tarTree`

The two functions of interest are `Project.zipTree(java.lang.Object)` and `Project.tarTree(java.lang.Object)`, which produce a [FileTree](#) from a corresponding archive file.

That file tree can then be used in a `from()` specification, like so:

build.gradle.kts

```
tasks.register<Copy>("unpackFiles") {
    from(zipTree("src/resources/thirdPartyResources.zip"))
    into(layout.buildDirectory.dir("resources"))
}
```

build.gradle

```
tasks.register('unpackFiles', Copy) {
    from zipTree("src/resources/thirdPartyResources.zip")
    into layout.buildDirectory.dir("resources")
}
```

As with a normal copy, you can control which files are unpacked via [filters](#) and even [rename files](#) as they are unpacked.

More advanced processing can be handled by the `eachFile()` method. For example, you might need to extract different subtrees of the archive into different paths within the destination directory. The

following sample uses the method to extract the files within the archive's **libs** directory into the root destination directory, rather than into a **libs** subdirectory:

build.gradle.kts

```
tasks.register<Copy>("unpackLibsDirectory") {
    from(zipTree("src/resources/thirdPartyResources.zip")) {
        include("libs/**") ①
        eachFile {
            relativePath = RelativePath(true,
            *relativePath.segments.drop(1).toTypedArray()) ②
        }
        includeEmptyDirs = false ③
    }
    into(layout.buildDirectory.dir("resources"))
}
```

build.gradle

```
tasks.register('unpackLibsDirectory', Copy) {
    from(zipTree("src/resources/thirdPartyResources.zip")) {
        include "libs/**" ①
        eachFile { fcd ->
            fcd.relativePath = new RelativePath(true, fcd.relativePath
            .segments.drop(1)) ②
        }
        includeEmptyDirs = false ③
    }
    into layout.buildDirectory.dir("resources")
}
```

- ① Extracts only the subset of files that reside in the **libs** directory
- ② Remaps the path of the extracting files into the destination directory by dropping the **libs** segment from the file path
- ③ Ignores the empty directories resulting from the remapping, see Caution note below

CAUTION

You can not change the destination path of empty directories with this technique. You can learn more in [this issue](#).

If you're a Java developer wondering why there is no **jarTree()** method, that's because **zipTree()** works perfectly well for JARs, WARs, and EARs.

Creating "uber" or "fat" JARs

In Java, applications and their dependencies were typically packaged as separate JARs within a single distribution archive. That still happens, but another approach that is now common is placing the classes and resources of the dependencies directly into the application JAR, creating what is known as an Uber or fat JAR.

Creating "uber" or "fat" JARs in Gradle involves packaging all dependencies into a single JAR file, making it easier to distribute and run the application.

Using the Shadow Plugin

Gradle does not have full built-in support for creating uber JARs, but you can use third-party plugins like the [Shadow plugin](#) (`com.gradleup.shadow`) to achieve this. This plugin packages your project classes and dependencies into a single JAR file.

Using `Project.zipTree()` and the `Jar` task

To copy the contents of other JAR files into the application JAR, use the [Project.zipTree\(java.lang.Object\)](#) method and the `Jar` task. This is demonstrated by the `uberJar` task in the following example:

build.gradle.kts

```
plugins {
    java
}

version = "1.0.0"

repositories {
    mavenCentral()
}

dependencies {
    implementation("commons-io:commons-io:2.6")
}

tasks.register<Jar>("uberJar") {
    archiveClassifier = "uber"

    from(sourceSets.main.get().output)

    dependsOn(configurations.runtimeClasspath)
    from({
        configurations.runtimeClasspath.get().filter {
            it.name.endsWith("jar")
        }.map { zipTree(it) }
    })
}
```

```
}
```

build.gradle

```
plugins {  
    id 'java'  
}  
  
version = '1.0.0'  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    implementation 'commons-io:commons-io:2.6'  
}  
  
tasks.register('uberJar', Jar) {  
    archiveClassifier = 'uber'  
  
    from sourceSets.main.output  
  
    dependsOn configurations.runtimeClasspath  
    from {  
        configurations.runtimeClasspath.findAll { it.name.endsWith('jar') }  
    }.collect { zipTree(it) }  
}
```

In this case, we're taking the runtime dependencies of the project — `configurations.runtimeClasspath.files` — and wrapping each of the JAR files with the `zipTree()` method. The result is a collection of ZIP file trees, the contents of which are copied into the uber JAR alongside the application classes.

Creating directories

Many tasks need to create directories to store the files they generate, which is why Gradle [automatically manages](#) this aspect of tasks when they explicitly define file and directory outputs. All core Gradle tasks ensure that any output directories they need are created, if necessary, using this mechanism.

Using `File.mkdirs` and `Files.createDirectories`

In cases where you need to create a directory manually, you can use the standard `Files.createDirectories` or `File.mkdirs` methods from within your build scripts or custom task

implementations.

Here is a simple example that creates a single `images` directory in the project folder:

build.gradle.kts

```
tasks.register("ensureDirectory") {
    // Store target directory into a variable to avoid project reference in
    the configuration cache
    val directory = file("images")

    doLast {
        Files.createDirectories(directory.toPath())
    }
}
```

build.gradle

```
tasks.register('ensureDirectory') {
    // Store target directory into a variable to avoid project reference in
    the configuration cache
    def directory = file("images")

    doLast {
        Files.createDirectories(directory.toPath())
    }
}
```

As described in the [Apache Ant manual](#), the `mkdir` task will automatically create all necessary directories in the given path. It will do nothing if the directory already exists.

Using `Project.mkdir`

You can create directories in Gradle using the `mkdir` method, which is available in the `Project` object. This method takes a `File` object or a `String` representing the path of the directory to be created:

```
tasks.register('createDirs') {
    doLast {
        mkdir 'src/main/resources'
        mkdir file('build/generated')

        // Create multiple dirs
        mkdir files(['src/main/resources', 'src/test/resources'])
    }
}
```

```

    // Check dir existence
    def dir = file('src/main/resources')
    if (!dir.exists()) {
        mkdir dir
    }
}
}

```

Installing executables

When you are building a standalone executable, you may want to install this file on your system, so it ends up in your path.

Using the **Copy** task

You can use a **Copy** task to install the executable into shared directories like `/usr/local/bin`. The installation directory probably contains many other executables, some of which may even be unreadable by Gradle. To support the unreadable files in the **Copy** task's destination directory and to avoid time consuming up-to-date checks, you can use `Task.doNotTrackState()`:

build.gradle.kts

```

tasks.register<Copy>("installExecutable") {
    from("build/my-binary")
    into("/usr/local/bin")
    doNotTrackState("Installation directory contains unrelated files")
}

```

build.gradle

```

tasks.register("installExecutable", Copy) {
    from "build/my-binary"
    into "/usr/local/bin"
    doNotTrackState("Installation directory contains unrelated files")
}

```

Deploying single files into application servers

Deploying a single file to an application server typically refers to the process of transferring a packaged application artifact, such as a WAR file, to the application server's deployment directory.

Using the **Copy** task

When working with application servers, you can use a **Copy** task to deploy the application archive (e.g. a WAR file). Since you are deploying a single file, the destination directory of the **Copy** is the whole deployment directory. The deployment directory sometimes does contain unreadable files like named pipes, so Gradle may have problems doing up-to-date checks. In order to support this use-case, you can use [Task.doNotTrackState\(\)](#):

build.gradle.kts

```
plugins {
    war
}

tasks.register<Copy>("deployToTomcat") {
    from(tasks.war)
    into(layout.projectDirectory.dir("tomcat/webapps"))
    doNotTrackState("Deployment directory contains unreadable files")
}
```

build.gradle

```
plugins {
    id 'war'
}

tasks.register("deployToTomcat", Copy) {
    from war
    into layout.projectDirectory.dir('tomcat/webapps')
    doNotTrackState("Deployment directory contains unreadable files")
}
```

Initialization Scripts

Initialization scripts are scripts that run before the build script is executed. They allow you to customize the build environment or configure settings early in the build.

Initialization scripts can be useful for setting up common configurations, such as repositories, plugins, or custom tasks, across multiple projects.

Using an init script

Initialization scripts, also called *init scripts*, are similar to other scripts in Gradle. Initialization scripts run before the build starts.

They are useful for various purposes:

- Setting up enterprise-wide configurations (e.g., custom plugin locations)
- Configuring properties based on the environment (e.g., developer's machine vs. CI server)
- Providing user-specific information (e.g., authentication credentials)
- Defining machine-specific details (e.g., JDK locations)
- Registering build listeners (e.g., external tools that wish to listen to Gradle events might find this helpful)
- Registering loggers (e.g., customize how Gradle logs the events that it generates)

One main [limitation of init scripts](#) is that they cannot access classes in the `buildSrc` project.

Invoking an init script

There are several ways to invoke an init script (in order of priority):

1. **Specify a file on the command line** with the option `-I` or `--init-script` followed by the path to the script.

The command line option can appear more than once, each time adding another init script. The build will fail if any files specified on the command line do not exist.

2. **Put a file called `init.gradle(.kts)`** in the `$GRADLE_USER_HOME/` directory.
3. **Put a file called `yourfilename.init.gradle(.kts)`** in the `$GRADLE_USER_HOME/init.d/` directory.
4. **Put a file called `yourfilename.init.gradle(.kts)`** in the `$GRADLE_HOME/init.d/` directory. Entries will be evaluated in alphabetic order.

This lets you package a custom Gradle distribution containing custom build logic and plugins. You can combine this with the [Gradle wrapper](#) to make custom logic available to all builds in your enterprise.

If more than one init script is found, they will all be executed in the order specified above.

Scripts in a given directory are executed in alphabetical order. For example, a tool can specify an init script on the command line and another in the home directory to define the environment. Both scripts will run when Gradle is executed.

Writing an init script

Like a Gradle build script, an init script is a Groovy or Kotlin script. Each init script has a [Gradle](#) instance associated with it. Any property reference and method call in the init script will be delegated to this `Gradle` instance.

Each init script implements the [Script](#) interface.

NOTE

When writing init scripts, pay attention to the scope of the reference you are trying to access. For example, properties loaded from `gradle.properties` are available on

Settings or **Project** instances but not on the **Gradle** one.

Configuring projects from an init script

You can use an init script to configure the projects in the build. This works similarly to configuring projects in a multi-project build.

The following sample shows how to perform extra configuration from an init script *before* the projects are evaluated:

build.gradle

```
repositories {
    mavenCentral()
}

tasks.register('showRepos') {
    def repositoryNames = repositories.collect { it.name }
    doLast {
        println "All repos:"
        println repositoryNames
    }
}
```

init.gradle

```
allprojects {
    repositories {
        mavenLocal()
    }
}
```

build.gradle.kts

```
repositories {
    mavenCentral()
}

tasks.register("showRepos") {
    val repositoryNames = repositories.map { it.name }
    doLast {
        println("All repos:")
        println(repositoryNames)
    }
}
```

init.gradle.kts

```
allprojects {  
    repositories {  
        mavenLocal()  
    }  
}
```

This sample uses this feature to configure an additional repository to be used only for specific environments.

```
> gradle --init-script init.gradle.kts -q showRepos  
All repos:  
[MavenLocal, MavenRepo]
```

```
> gradle --init-script init.gradle -q showRepos  
All repos:  
[MavenLocal, MavenRepo]
```

Adding external dependencies

Init scripts can also declare dependencies with the `initscript()` method, passing in a closure that declares the init script classpath.

Declaring external dependencies for an init script:

init.gradle.kts

```
initscript {  
    repositories {  
        mavenCentral()  
    }  
    dependencies {  
        classpath("org.apache.commons:commons-math:2.0")  
    }  
}
```

init.gradle

```
initscript {  
    repositories {  
        mavenCentral()  
    }  
}
```

```

    }
    dependencies {
        classpath 'org.apache.commons:commons-math:2.0'
    }
}

```

The closure passed to the `initscript()` method configures a `ScriptHandler` instance. You declare the init script classpath by adding dependencies to the `classpath` configuration.

This is the same way you declare, for example, the Java compilation classpath. You can use any of the dependency types described in [Declaring Dependencies](#), except project dependencies.

Having declared the init script classpath, you can use the classes in your init script as you would any other classes on the classpath. The following example adds to the previous example and uses classes from the init script classpath.

An init script with external dependencies:

init.gradle.kts

```

import org.apache.commons.math.fraction.Fraction

initscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.apache.commons:commons-math:2.0")
    }
}

println(Fraction.ONE_FIFTH.multiply(2))

```

build.gradle.kts

```

tasks.register("doNothing")

```

init.gradle

```

import org.apache.commons.math.fraction.Fraction

initscript {
    repositories {
        mavenCentral()
    }
}

```

```
dependencies {  
    classpath 'org.apache.commons:commons-math:2.0'  
}  
}  
  
println Fraction.ONE_FIFTH.multiply(2)
```

build.gradle

```
tasks.register('doNothing')
```

```
> gradle --init-script init.gradle.kts -q doNothing  
2 / 5
```

```
> gradle --init-script init.gradle -q doNothing  
2 / 5
```

Applying plugins

Plugins can be applied to init scripts like a Gradle build script or a Gradle settings file.

Using plugins in init scripts:

init.gradle.kts

```
apply<EnterpriseRepositoryPlugin>()  
  
class EnterpriseRepositoryPlugin : Plugin<Gradle> {  
    companion object {  
        const val ENTERPRISE_REPOSITORY_URL =  
            "https://repo.gradle.org/gradle/repo"  
    }  
  
    override fun apply(gradle: Gradle) {  
        // ONLY USE ENTERPRISE REPO FOR DEPENDENCIES  
        gradle.allprojects {  
            repositories {  
  
                // Remove all repositories not pointing to the enterprise  
                repository url  
                all {  
                    if (this !is MavenArtifactRepository || url.toString() !=  
                        ENTERPRISE_REPOSITORY_URL) {  
                        project.logger.lifecycle("Repository ${(this as?)}
```

```
build.gradle.kts
```

init.gradle

```
apply plugin: EnterpriseRepositoryPlugin

class EnterpriseRepositoryPlugin implements Plugin<Gradle> {

    private static String ENTERPRISE_REPOSITORY_URL =
        "https://repo.gradle.org/gradle/repo"

    void apply(Gradle gradle) {
        // ONLY USE ENTERPRISE REPO FOR DEPENDENCIES
        gradle.allprojects { project ->
            project.repositories {
```

```

        // Remove all repositories not pointing to the enterprise
        repository url
        all { ArtifactRepository repo ->
            if (!(repo instanceof MavenArtifactRepository) ||
                repo.url.toString() != ENTERPRISE_REPOSITORY_URL) {
                project.logger.lifecycle "Repository ${repo.url}
removed. Only $ENTERPRISE_REPOSITORY_URL is allowed"
                remove repo
            }
        }

        // add the enterprise repository
        maven {
            name = "STANDARD_ENTERPRISE_REPO"
            url = ENTERPRISE_REPOSITORY_URL
        }
    }
}
}

```

build.gradle

```

repositories{
    mavenCentral()
}

@Immutable
class RepositoryData {
    String name
    URI url
}

tasks.register('showRepositories') {
    def repositoryData = repositories.collect { new RepositoryData(it.name,
it.url) }
    doLast {
        repositoryData.each {
            println "repository: ${it.name} ('${it.url}')"
        }
    }
}

```

```

> gradle --init-script init.gradle.kts -q showRepositories
repository: STANDARD_ENTERPRISE_REPO ('https://repo.gradle.org/gradle/repo')

```

```
> gradle --init-script init.gradle -q showRepositories
repository: STANDARD_ENTERPRISE_REPO ('https://repo.gradle.org/gradle/repo')
```

The plugin in the init script ensures that only a specified repository is used when running the build.

When applying plugins within the init script, Gradle instantiates the plugin and calls the plugin instance's `Plugin.apply(T)` method.

The `gradle` object is passed as a parameter, which can be used to configure all aspects of a build. Of course, the applied plugin can be resolved as an external dependency as described in [External dependencies for the init script](#)

Dataflow Actions

NOTE The dataflow actions support is an [incubating](#) feature and is subject to change.

A preferred way of executing work in a Gradle build is using a task. However, some kinds of work do not fit tasks well, such as custom handling of the build failure.

What if you want to play a cheerful sound when the build succeeds and a sad one when it fails? This work piece has to process the task execution result, so it cannot be a task itself.

The Dataflow Actions API provides a way to schedule this type of work. A dataflow action is a parameterized isolated piece of work that becomes eligible for execution as soon as all input parameters become available.

Implementing a dataflow action

The first step is to implement the action itself. You must create a class implementing the [FlowAction](#) interface:

```
import org.gradle.api.flow.FlowAction
import org.gradle.api.flow.FlowParameters

abstract class ReportConsumption : FlowAction<ReportConsumption.Params> {

    interface Params : FlowParameters {

    }

    override fun execute(parameters: Params) {

    }

}
```

The `execute` method must be implemented because this is where the work happens. An action implementation is treated as a [custom Gradle type](#) and can use any of the features available to

custom Gradle types. In particular, some Gradle services can be injected into the implementation.

A dataflow action may accept parameters. To provide parameters, you define an abstract class (or interface) to hold the parameters:

- The parameters type must implement (or extend) [FlowParameters](#).
- The parameters type is also a [custom Gradle type](#).
- The action implementation gets the parameters as an argument of the `execute` method.

When the action requires no parameters, you can use [FlowParameters.None](#) as the type of parameter.

Here is an example of a dataflow action that takes a shared build service and a file path as parameters:

SoundPlay.java

```
package org.gradle.sample.sound;

import org.gradle.api.flow.FlowAction;
import org.gradle.api.flow.FlowParameters;
import org.gradle.api.provider.Property;
import org.gradle.api.services.ServiceReference;
import org.gradle.api.tasks.Input;

import java.io.File;

public abstract class SoundPlay implements FlowAction<SoundPlay.Parameters> {
    interface Parameters extends FlowParameters {
        @ServiceReference ①
        Property<SoundService> getSoundService();

        @Input ②
        Property<File> getMediaFile();
    }

    @Override
    public void execute(Parameters parameters) {
        parameters.getSoundService().get().playSoundFile(parameters.getMediaFile(
        ).get());
    }
}
```

- ① Parameters in the parameter type must be annotated. If a parameter is annotated with `@ServiceReference`, then a suitable shared build service implementation is automatically assigned to the parameter when the action is created, according to the [usual rules](#).
- ② All other parameters must be annotated with `@Input`.

Using lifecycle event providers

Besides the usual value providers, Gradle provides dedicated providers for build lifecycle events, like build completion. These providers are intended for dataflow actions and provide additional ordering guarantees when used as inputs. The ordering also applies if you derive a provider from the event provider by, for example, calling `map` or `flatMap`. You can obtain these providers from the `FlowProviders` class.

```
flowProviders.buildWorkResult.map {  
    [  
        buildInvocationId: scopeIdsService.buildInvocationId,  
        workspaceId: scopeIdsService.workspaceId,  
        userId: scopeIdsService.userId  
    ]  
}
```

WARNING

If you're not using a lifecycle event provider as an input to the dataflow action, then the exact timing when the action is executed is not defined and may change in the next version of Gradle.

Supplying the action for execution

You should not create `FlowAction` objects manually. Instead, you request to execute them in the appropriate scope of `FlowScope`. In doing so, you can configure the parameters for the task:

SoundFeedbackPlugin.java

```
package org.gradle.sample.sound;  
  
import org.gradle.api.Plugin;  
import org.gradle.api.flow.FlowProviders;  
import org.gradle.api.flow.FlowScope;  
import org.gradle.api.initialization.Settings;  
  
import javax.inject.Inject;  
import java.io.File;  
  
public abstract class SoundFeedbackPlugin implements Plugin<Settings> {  
    @Inject  
    protected abstract FlowScope getFlowScope(); ①  
  
    @Inject  
    protected abstract FlowProviders getFlowProviders(); ①  
  
    @Override  
    public void apply(Settings settings) {  
        final File soundsDir = new File(settings.getSettingsDir(), "sounds");  
        getFlowScope().always( ②
```

```

    SoundPlay.class, ③
    spec -> ④
        spec.getParameters().getMediaFile().set(
            getFlowProviders().getBuildWorkResult().map(result -> ⑤
                new File(
                    soundsDir,
                    result.getFailure().isPresent() ? "sad-trombone.mp3" :
"tada.mp3"
                )
            )
        )
    );
}
}

```

- ① Use service injection to obtain `FlowScope` and `FlowProviders` instances. They are available for project and settings plugins.
- ② Use an appropriate scope to run your actions. As the name suggests, actions in the `always` scope are executed every time the build runs.
- ③ Specify the class that implements the action.
- ④ Use the `spec` argument to configure the action parameters.
- ⑤ A lifecycle event provider can be mapped into something else while preserving the action order.

As a result, when you run the build, and it completes successfully, the action will play the "tada" sound. If the build fails at configuration or execution time, you'll hear "sad-trombone" sound — assuming that build configuration proceeds far enough for the action to be registered.

Testing Build Logic with TestKit

The Gradle TestKit (a.k.a. just TestKit) is a library that aids in testing Gradle plugins and build logic generally. At this time, it is focused on *functional* testing. That is, testing build logic by exercising it as part of a programmatically executed build. Over time, the TestKit will likely expand to facilitate other kinds of tests.

Using TestKit

To use the TestKit, include the following in your plugin's build:

Example 1. Declaring the TestKit dependency

build.gradle.kts

```

dependencies {
    testImplementation(gradleTestKit())
}

```

build.gradle

```
dependencies {  
    testImplementation gradleTestKit()  
}
```

The `gradleTestKit()` encompasses the classes of the TestKit, as well as the [Gradle Tooling API client](#). It does not include a version of [JUnit](#), [TestNG](#), or any other test execution framework. Such a dependency must be explicitly declared.

Example 2. [Declaring the JUnit dependency](#)

build.gradle.kts

```
dependencies {  
    testImplementation("org.junit.jupiter:junit-jupiter:5.7.1")  
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")  
}  
  
tasks.named<Test>("test") {  
    useJUnitPlatform()  
}
```

build.gradle

```
dependencies {  
    testImplementation("org.junit.jupiter:junit-jupiter:5.7.1")  
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")  
}  
  
tasks.named('test', Test) {  
    useJUnitPlatform()  
}
```

Functional testing with the Gradle runner

The [GradleRunner](#) facilitates programmatically executing Gradle builds, and inspecting the result.

A contrived build can be created (e.g. programmatically, or from a template) that exercises the "logic under test". The build can then be executed, potentially in a variety of ways (e.g. different combinations of tasks and arguments). The correctness of the logic can then be verified by asserting the following, potentially in combination:

- The build's output;
- The build's logging (i.e. console output);
- The set of tasks executed by the build and their results (e.g. FAILED, UP-TO-DATE etc.).

After creating and configuring a runner instance, the build can be executed via the `GradleRunner.build()` or `GradleRunner.buildAndFail()` methods depending on the anticipated outcome.

The following demonstrates the usage of the Gradle runner in a Java JUnit test:

Example: Using GradleRunner with Java and JUnit

BuildLogicFunctionalTest.java

```
import org.gradle.testkit.runner.BuildResult;
import org.gradle.testkit.runner.GradleRunner;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.io.TempDir;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

import static org.gradle.testkit.runner.TaskOutcome.SUCCESS;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;

public class BuildLogicFunctionalTest {

    @TempDir File testProjectDir;
    private File settingsFile;
    private File buildFile;

    @BeforeEach
    public void setup() {
        settingsFile = new File(testProjectDir, "settings.gradle");
        buildFile = new File(testProjectDir, "build.gradle");
    }

    @Test
    public void testHelloWorldTask() throws IOException {
        writeFile(settingsFile, "rootProject.name = 'hello-world'");
        String buildFileContent = "task helloWorld {" +
                                   "    doLast {" +
                                   "        println 'Hello world!'" +
                                   "    }" +
                                   "}";
        writeFile(buildFile, buildFileContent);
    }
}
```

```

        BuildResult result = GradleRunner.create()
            .withProjectDir(testProjectDir)
            .withArguments("helloWorld")
            .build();

        assertTrue(result.getOutput().contains("Hello world!"));
        assertEquals(SUCCESS, result.task(":helloWorld").getOutcome());
    }

    private void writeFile(File destination, String content) throws IOException {
        BufferedWriter output = null;
        try {
            output = new BufferedWriter(new FileWriter(destination));
            output.write(content);
        } finally {
            if (output != null) {
                output.close();
            }
        }
    }
}

```

Any test execution framework can be used.

As Gradle build scripts can also be written in the Groovy programming language, it is often a productive choice to write Gradle functional tests in Groovy. Furthermore, it is recommended to use the (Groovy based) [Spock test execution framework](#) as it offers many compelling features over the use of JUnit.

The following demonstrates the usage of the Gradle runner in a Groovy Spock test:

Example: Using GradleRunner with Groovy and Spock

BuildLogicFunctionalTest.groovy

```

import org.gradle.testkit.runner.GradleRunner
import static org.gradle.testkit.runner.TaskOutcome.*
import spock.lang.TempDir
import spock.lang.Specification

class BuildLogicFunctionalTest extends Specification {
    @TempDir File testProjectDir
    File settingsFile
    File buildFile

    def setup() {
        settingsFile = new File(testProjectDir, 'settings.gradle')
        buildFile = new File(testProjectDir, 'build.gradle')
    }
}

```

```

def "hello world task prints hello world"() {
    given:
    settingsFile << "rootProject.name = 'hello-world'"
    buildFile << """
        task helloWorld {
            doLast {
                println 'Hello world!'
            }
        }
    """

    when:
    def result = GradleRunner.create()
        .withProjectDir(testProjectDir)
        .withArguments('helloWorld')
        .build()

    then:
    result.output.contains('Hello world!')
    result.task(":helloWorld").outcome == SUCCESS
}

```

It is a common practice to implement any custom build logic (like plugins and task types) that is more complex in nature as external classes in a standalone project. The main driver behind this approach is bundle the compiled code into a JAR file, publish it to a binary repository and reuse it across various projects.

Getting the plugin-under-test into the test build

The GradleRunner uses the [Tooling API](#) to execute builds. An implication of this is that the builds are executed in a separate process (i.e. not the same process executing the tests). Therefore, the test build does not share the same classpath or classloaders as the test process and the code under test is not implicitly available to the test build.

NOTE

GradleRunner supports the same range of Gradle versions as the Tooling API. The supported versions are defined in the [compatibility matrix](#).

Builds with older Gradle versions *may* still work but there are no guarantees.

Starting with version 2.13, Gradle provides a conventional mechanism to inject the code under test into the test build.

Automatic injection with the Java Gradle Plugin Development plugin

The [Java Gradle Plugin development plugin](#) can be used to assist in the development of Gradle plugins. Starting with Gradle version 2.13, the plugin provides a direct integration with TestKit. When applied to a project, the plugin automatically adds the `gradleTestKit()` dependency to the `testApi` configuration. Furthermore, it automatically generates the classpath for the code under test

and injects it via `GradleRunner.withPluginClasspath()` for any `GradleRunner` instance created by the user. It's important to note that the mechanism currently *only* works if the plugin under test is applied using the `plugins DSL`. If the `target Gradle version` is prior to 2.8, automatic plugin classpath injection is not performed.

The plugin uses the following conventions for applying the TestKit dependency and injecting the classpath:

- Source set containing code under test: `sourceSets.main`
- Source set used for injecting the plugin classpath: `sourceSets.test`

Any of these conventions can be reconfigured with the help of the class `GradlePluginDevelopmentExtension`.

The following Groovy-based sample demonstrates how to automatically inject the plugin classpath by using the standard conventions applied by the Java Gradle Plugin Development plugin.

Example 3. Using the Java Gradle Development plugin for generating the plugin metadata

build.gradle.kts

```
plugins {
    groovy
    `java-gradle-plugin`
}

dependencies {
    testImplementation("org.spockframework:spock-core:2.2-groovy-3.0") {
        exclude(group = "org.codehaus.groovy")
    }
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")
}
```

build.gradle

```
plugins {
    id 'groovy'
    id 'java-gradle-plugin'
}

dependencies {
    testImplementation('org.spockframework:spock-core:2.2-groovy-3.0') {
        exclude group: 'org.codehaus.groovy'
    }
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
}
```

Example: Automatically injecting the code under test classes into test builds

src/test/groovy/org/gradle/sample/BuildLogicFunctionalTest.groovy

```
def "hello world task prints hello world"() {
    given:
    settingsFile << "rootProject.name = 'hello-world'"
    buildFile << """
        plugins {
            id 'org.gradle.sample.helloworld'
        }
    """

    when:
    def result = GradleRunner.create()
        .withProjectDir(testProjectDir)
        .withArguments('helloWorld')
        .withPluginClasspath()
        .build()

    then:
    result.output.contains('Hello world!')
    result.task(":helloWorld").outcome == SUCCESS
}
```

The following build script demonstrates how to reconfigure the conventions provided by the Java Gradle Plugin Development plugin for a project that uses a custom **Test** source set.

NOTE

A new configuration DSL for modeling the below **functionalTest** suite is available via the incubating **JVM Test Suite** plugin.

Example 4. Reconfiguring the classpath generation conventions of the Java Gradle Development plugin

build.gradle.kts

```
plugins {
    groovy
    `java-gradle-plugin`
}

val functionalTest = sourceSets.create("functionalTest")
val functionalTestTask = tasks.register<Test>("functionalTest") {
    group = "verification"
    testClassesDirs = functionalTest.output.classesDirs
    classpath = functionalTest.runtimeClasspath
    useJUnitPlatform()
}

tasks.check {
```

```

        dependsOn(functionalTestTask)
    }

    gradlePlugin {
        testSourceSets(functionalTest)
    }

    dependencies {
        "functionalTestImplementation"("org.spockframework:spock-core:2.2-groovy-3.0") {
            exclude(group = "org.codehaus.groovy")
        }
        "functionalTestRuntimeOnly"("org.junit.platform:junit-platform-launcher")
    }

```

build.gradle

```

plugins {
    id 'groovy'
    id 'java-gradle-plugin'
}

def functionalTest = sourceSets.create('functionalTest')
def functionalTestTask = tasks.register('functionalTest', Test) {
    group = 'verification'
    testClassesDirs = sourceSets.functionalTest.output.classesDirs
    classpath = sourceSets.functionalTest.runtimeClasspath
    useJUnitPlatform()
}

tasks.named("check") {
    dependsOn functionalTestTask
}

gradlePlugin {
    testSourceSets sourceSets.functionalTest
}

dependencies {
    functionalTestImplementation('org.spockframework:spock-core:2.2-groovy-3.0') {
        exclude group: 'org.codehaus.groovy'
    }
    functionalTestRuntimeOnly 'org.junit.platform:junit-platform-launcher'
}

```

Controlling the build environment

The runner executes the test builds in an isolated environment by specifying a dedicated "working directory" in a directory inside the JVM's temp directory (i.e. the location specified by the `java.io.tmpdir` system property, typically `/tmp`). Any configuration in the default Gradle User Home (e.g. `~/.gradle/gradle.properties`) is not used for test execution. The TestKit does not expose a mechanism for fine grained control of all aspects of the environment (e.g., JDK). Future versions of the TestKit will provide improved configuration options.

The TestKit uses dedicated daemon processes that are automatically shut down after test execution.

The dedicated working directory is not deleted by the runner after the build. The TestKit provides two ways to specify a location that is regularly cleaned, such as the project's build folder:

- The `org.gradle.testkit.dir` system property;
- The `GradleRunner.withTestKitDir(file testKitDir)` method.

Setting the Gradle version used to test

The Gradle runner requires a Gradle distribution in order to execute the build. The TestKit does not depend on all of Gradle's implementation.

By default, the runner will attempt to find a Gradle distribution based on where the `GradleRunner` class was loaded from. That is, it is expected that the class was loaded from a Gradle distribution, as is the case when using the `gradleTestKit()` dependency declaration.

When using the runner as part of tests *being executed by Gradle* (e.g. executing the `test` task of a plugin project), the same distribution used to execute the tests will be used by the runner. When using the runner as part of tests *being executed by an IDE*, the same distribution of Gradle that was used when importing the project will be used. This means that the plugin will effectively be tested with the same version of Gradle that it is being built with.

Alternatively, a different and specific version of Gradle to use can be specified by the any of the following `GradleRunner` methods:

- `GradleRunner.withGradleVersion(java.lang.String)`
- `GradleRunner.withGradleInstallation(java.io.File)`
- `GradleRunner.withGradleDistribution(java.net.URI)`

This can potentially be used to test build logic across Gradle versions. The following demonstrates a cross-version compatibility test written as Groovy Spock test:

Example: Specifying a Gradle version for test execution

BuildLogicFunctionalTest.groovy

```
import org.gradle.testkit.runner.GradleRunner
import static org.gradle.testkit.runner.TaskOutcome.*
import spock.lang.TempDir
```

```

import spock.lang.Specification

class BuildLogicFunctionalTest extends Specification {
    @TempDir File testProjectDir
    File settingsFile
    File buildFile

    def setup() {
        settingsFile = new File(testProjectDir, 'settings.gradle')
        buildFile = new File(testProjectDir, 'build.gradle')
    }

    def "can execute hello world task with Gradle version #gradleVersion"() {
        given:
        buildFile << """
            task helloWorld {
                doLast {
                    logger.quiet 'Hello world!'
                }
            }
        """
        settingsFile << ""

        when:
        def result = GradleRunner.create()
            .withGradleVersion(gradleVersion)
            .withProjectDir(testProjectDir)
            .withArguments('helloWorld')
            .build()

        then:
        result.output.contains('Hello world!')
        result.task(":helloWorld").outcome == SUCCESS

        where:
        gradleVersion << ['5.0', '6.0.1']
    }
}

```

Feature support when testing with different Gradle versions

It is possible to use the GradleRunner to execute builds with Gradle 1.0 and later. However, some runner features are not supported on earlier versions. In such cases, the runner will throw an exception when attempting to use the feature.

The following table lists the features that are sensitive to the Gradle version being used.

Table 5. Gradle version compatibility

Feature	Minimum Version	Description
Inspecting executed tasks	2.5	Inspecting the executed tasks, using BuildResult.getTasks() and similar methods.
Plugin classpath injection	2.8	Injecting the code under test via GradleRunner.withPluginClasspath(java.lang.Iterable) .
Inspecting build output in debug mode	2.9	Inspecting the build's text output when run in debug mode, using BuildResult.getOutput() .
Automatic plugin classpath injection	2.13	Injecting the code under test automatically via GradleRunner.withPluginClasspath() by applying the Java Gradle Plugin Development plugin.
Setting environment variables to be used by the build.	3.5	The Gradle Tooling API only supports setting environment variables in later versions.

Debugging build logic

The runner uses the [Tooling API](#) to execute builds. An implication of this is that the builds are executed in a separate process (i.e. not the same process executing the tests). Therefore, executing your *tests* in debug mode does not allow you to debug your build logic as you may expect. Any breakpoints set in your IDE will be not be tripped by the code being exercised by the test build.

The TestKit provides two different ways to enable the debug mode:

- Setting “org.gradle.testkit.debug” system property to `true` for the JVM *using* the `GradleRunner` (i.e. not the build being executed with the runner);
- Calling the [GradleRunner.withDebug\(boolean\)](#) method.

The system property approach can be used when it is desirable to enable debugging support without making an adhoc change to the runner configuration. Most IDEs offer the capability to set JVM system properties for test execution, and such a feature can be used to set this system property.

Testing with the Build Cache

To enable the [Build Cache](#) in your tests, you can pass the `--build-cache` argument to [GradleRunner](#) or use one of the other methods described in [Enable the build cache](#). You can then check for the task outcome [TaskOutcome.FROM_CACHE](#) when your plugin's custom task is cached. This outcome is only valid for Gradle 3.5 and newer.

Example: Testing cacheable tasks

BuildLogicFunctionalTest.groovy

```
def "cacheableTask is loaded from cache"() {
    given:
    buildFile << """
```

```

        plugins {
            id 'org.gradle.sample.helloworld'
        }
    """

    when:
    def result = runner()
        .withArguments( '--build-cache', 'cacheableTask')
        .build()

    then:
    result.task(":cacheableTask").outcome == SUCCESS

    when:
    new File(testProjectDir, 'build').deleteDir()
    result = runner()
        .withArguments( '--build-cache', 'cacheableTask')
        .build()

    then:
    result.task(":cacheableTask").outcome == FROM_CACHE
}

```

Note that TestKit re-uses a Gradle User Home between tests (see [GradleRunner.withTestKitDir\(java.io.File\)](#)) which contains the default location for the local build cache. For testing with the build cache, the build cache directory should be cleaned between tests. The easiest way to accomplish this is to configure the local build cache to use a temporary directory.

Example: Clean build cache between tests

BuildLogicFunctionalTest.groovy

```

@TempDir File testProjectDir
File buildFile
File localBuildCacheDirectory

def setup() {
    localBuildCacheDirectory = new File(testProjectDir, 'local-cache')
    buildFile = new File(testProjectDir, 'settings.gradle') << """
        buildCache {
            local {
                directory = '${localBuildCacheDirectory.toURI()}'
            }
        }
    """

    buildFile = new File(testProjectDir, 'build.gradle')
}

```

Using Ant from Gradle

Gradle provides integration with Ant.

Gradle integrates with Ant, allowing you to use individual Ant tasks or entire Ant builds in your Gradle builds. Using Ant tasks in a Gradle build script is often easier and more powerful than using Ant's XML format. Gradle can also be used as a powerful Ant task scripting tool.

Ant can be divided into two layers:

1. **Layer 1: The Ant language.** It provides the syntax for the `build.xml` file, the handling of the targets, special constructs like macrodefs, and more. In other words, this layer includes everything except the Ant tasks and types. Gradle understands this language and lets you import your Ant `build.xml` directly into a Gradle project. You can then use the targets of your Ant build as if they were Gradle tasks.
2. **Layer 2: The Ant tasks and types,** like `javac`, `copy` or `jar`. For this layer, Gradle provides integration using Groovy and the `AntBuilder`.

Since build scripts are Kotlin or Groovy scripts, you can execute an Ant build as an external process. Your build script may contain statements like: `"ant clean compile".execute().[1]`

Gradle's Ant integration allows you to migrate your build from Ant to Gradle smoothly:

1. Begin by importing your existing Ant build.
2. Then, transition your dependency declarations from the Ant script to your build file.
3. Finally, move your tasks to your build file or replace them with Gradle's plugins.

This migration process can be performed incrementally, and you can maintain a functional Gradle build throughout the transition.

WARNING

Ant integration is not fully compatible with the [configuration cache](#). Using `Task.ant` to run Ant task in the task action may work, but importing the Ant build is not supported.

The Ant integration is provided by the [AntBuilder API](#).

Using Ant tasks and types

Gradle provides a property called `ant` in your build script. This is a reference to an [AntBuilder](#) instance.

`AntBuilder` is used to access Ant tasks, types, and properties from your build script.

You execute an Ant task by calling a method on the `AntBuilder` instance. You use the task name as the method name:

build.gradle

```
ant.mkdir(dir: "$STAGE")
```

```
ant.copy(todir: "$STAGE/bin") {
    ant.fileset(dir: 'bin', includes: "**")
}
ant.gzip(destfile:"build/file-${VERSION}.tar.gz", src: "build/file-${VERSION}.tar")
```

For example, you execute the Ant `echo` task using the `ant.echo()` method.

The attributes of the Ant task are passed as Map parameters to the method. Below is an example of the `echo` task:

build.gradle.kts

```
tasks.register("hello") {
    doLast {
        val greeting = "hello from Ant"
        ant.withGroovyBuilder {
            "echo"("message" to greeting)
        }
    }
}
```

build.gradle

```
tasks.register('hello') {
    doLast {
        String greeting = 'hello from Ant'
        ant.echo(message: greeting)
    }
}
```

```
$ gradle hello

> Task :hello
[ant:echo] hello from Ant

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

TIP

You can mix Groovy/Kotlin code and the Ant task markup. This can be extremely powerful.

You pass nested text to an Ant task as a parameter of the task method call. In this example, we pass the message for the `echo` task as nested text:

build.gradle.kts

```
tasks.register("hello") {
    doLast {
        ant.withGroovyBuilder {
            "echo"("message" to "hello from Ant")
        }
    }
}
```

build.gradle

```
tasks.register('hello') {
    doLast {
        ant.echo('hello from Ant')
    }
}
```

```
$ gradle hello
```

```
> Task :hello
[ant:echo] hello from Ant
```

```
BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

You pass nested elements to an Ant task inside a closure. Nested elements are defined in the same way as tasks by calling a method with the same name as the element we want to define:

build.gradle.kts

```
tasks.register("zip") {
    doLast {
        ant.withGroovyBuilder {
            "zip"("destfile" to "archive.zip") {
                "fileset"("dir" to "src") {
                    "include"("name" to "**.xml")
                    "exclude"("name" to "**.java")
                }
            }
        }
    }
}
```

```
}
```

build.gradle

```
tasks.register('zip') {
    doLast {
        ant.zip(destfile: 'archive.zip') {
            fileset(dir: 'src') {
                include(name: '**.xml')
                exclude(name: '**.java')
            }
        }
    }
}
```

You can access Ant types the same way you access tasks, using the name of the type as the method name. The method call returns the Ant data type, which you can use directly in your build script. In the following example, we create an Ant `path` object, then iterate over the contents of it:

build.gradle.kts

```
import org.apache.tools.ant.types.Path

tasks.register("list") {
    doLast {
        val path = ant.withGroovyBuilder {
            "path" {
                "fileset"("dir" to "libs", "includes" to "*.jar")
            }
        } as Path
        path.list().forEach {
            println(it)
        }
    }
}
```

build.gradle

```
tasks.register('list') {
    doLast {
        def path = ant.path {
            fileset(dir: 'libs', includes: '*.jar')
        }
    }
}
```

```

        path.list().each {
            println it
        }
    }
}

```

Using custom Ant tasks

To make custom tasks available in your build, use the `taskdef` (usually easier) or `typedef` Ant task, just as you would in a `build.xml` file. You can then refer to the custom Ant task as you would a built-in Ant task:

build.gradle.kts

```

tasks.register("check") {
    val checkstyleConfig = file("checkstyle.xml")
    doLast {
        ant.withGroovyBuilder {
            "taskdef"("resource" to
"com/puppycrawl/tools/checkstyle/ant/checkstyle-ant-task.properties") {
                "classpath" {
                    "fileset"("dir" to "libs", "includes" to "*.jar")
                }
            }
            "checkstyle"("config" to checkstyleConfig) {
                "fileset"("dir" to "src")
            }
        }
    }
}

```

build.gradle

```

tasks.register('check') {
    def checkstyleConfig = file('checkstyle.xml')
    doLast {
        ant.taskdef(resource:
'com/puppycrawl/tools/checkstyle/ant/checkstyle-ant-task.properties') {
            classpath {
                fileset(dir: 'libs', includes: '*.jar')
            }
        }
        ant.checkstyle(config: checkstyleConfig) {
            fileset(dir: 'src')
        }
    }
}

```

```
}  
}
```

You can use Gradle's dependency management to assemble the classpath for the custom tasks. To do this, you need to define a custom configuration for the classpath and add some dependencies to it. This is described in more detail in [Declaring Dependencies](#):

build.gradle.kts

```
val pmd = configurations.create("pmd")  
  
dependencies {  
    pmd(group = "pmd", name = "pmd", version = "4.2.5")  
}
```

build.gradle

```
configurations {  
    pmd  
}  
  
dependencies {  
    pmd group: 'pmd', name: 'pmd', version: '4.2.5'  
}
```

To use the classpath configuration, use the `asPath` property of the custom configuration:

build.gradle.kts

```
tasks.register("check") {  
    doLast {  
        ant.withGroovyBuilder {  
            "taskdef"("name" to "pmd",  
                    "classname" to "net.sourceforge.pmd.ant.PMDTask",  
                    "classpath" to pmd.asPath)  
            "pmd"("shortFileNames" to true,  
                "failonruleviolation" to true,  
                "rulesetfiles" to file("pmd-rules.xml").toURI().toString())  
        }  
        "formatter"("type" to "text", "toConsole" to "true")  
        "fileset"("dir" to "src")  
    }  
}
```

```

    }
  }
}

```

build.gradle

```

tasks.register('check') {
    doLast {
        ant.taskdef(name: 'pmd',
                    classname: 'net.sourceforge.pmd.ant.PMDTask',
                    classpath: configurations.pmd.asPath)
        ant.pmd(shortFileNames: 'true',
                failonruleviolation: 'true',
                rulesetfiles: file('pmd-rules.xml').toURI().toString()) {
            formatter(type: 'text', toConsole: 'true')
            fileset(dir: 'src')
        }
    }
}

```

Importing an Ant build

You can use the `ant.importBuild()` method to import an Ant build into your Gradle project.

When you import an Ant build, each Ant target is treated as a Gradle task. This means you can manipulate and execute the Ant targets in the same way as Gradle tasks:

build.gradle.kts

```
ant.importBuild("build.xml")
```

build.gradle

```
ant.importBuild 'build.xml'
```

build.xml

```
<project>
```

```
<target name="hello">
    <echo>Hello, from Ant</echo>
</target>
</project>
```

```
$ gradle hello

> Task :hello
[ant:echo] Hello, from Ant

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

You can add a task that depends on an Ant target:

build.gradle.kts

```
ant.importBuild("build.xml")

tasks.register("intro") {
    dependsOn("hello")
    doLast {
        println("Hello, from Gradle")
    }
}
```

build.gradle

```
ant.importBuild 'build.xml'

tasks.register('intro') {
    dependsOn("hello")
    doLast {
        println 'Hello, from Gradle'
    }
}
```

```
$ gradle intro

> Task :hello
[ant:echo] Hello, from Ant
```

```
> Task :intro
Hello, from Gradle

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

Or, you can add behavior to an Ant target:

build.gradle.kts

```
ant.importBuild("build.xml")

tasks.named("hello") {
    doLast {
        println("Hello, from Gradle")
    }
}
```

build.gradle

```
ant.importBuild 'build.xml'

hello {
    doLast {
        println 'Hello, from Gradle'
    }
}
```

```
$ gradle hello

> Task :hello
[ant:echo] Hello, from Ant
Hello, from Gradle

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

It is also possible for an Ant target to depend on a Gradle task:

build.gradle.kts

```
ant.importBuild("build.xml")

tasks.register("intro") {
    doLast {
        println("Hello, from Gradle")
    }
}
```

build.gradle

```
ant.importBuild 'build.xml'

tasks.register('intro') {
    doLast {
        println 'Hello, from Gradle'
    }
}
```

build.xml

```
<project>
  <target name="hello" depends="intro">
    <echo>Hello, from Ant</echo>
  </target>
</project>
```

```
$ gradle hello
```

```
> Task :intro
Hello, from Gradle
```

```
> Task :hello
[ant:echo] Hello, from Ant
```

```
BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

Sometimes, it may be necessary to "rename" the task generated for an Ant target to avoid a naming

collision with existing Gradle tasks. To do this, use the [AntBuilder.importBuild\(java.lang.Object, org.gradle.api.Transformer\)](#) method:

build.gradle.kts

```
ant.importBuild("build.xml") { antTargetName ->
    "a-" + antTargetName
}
```

build.gradle

```
ant.importBuild('build.xml') { antTargetName ->
    'a-' + antTargetName
}
```

build.xml

```
<project>
  <target name="hello">
    <echo>Hello, from Ant</echo>
  </target>
</project>
```

```
$ gradle a-hello
```

```
> Task :a-hello
[ant:echo] Hello, from Ant
```

```
BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

NOTE

While the second argument to this method should be a [Transformer](#), when programming in Groovy you can use a closure instead of an anonymous inner class (or similar) due to [Groovy's support for automatically coercing closures to single-abstract-method types](#).

Using Ant properties and references

There are several ways to set an Ant property so that the property can be used by Ant tasks.

You can set the property directly on the `AntBuilder` instance. The Ant properties are also available as a Map, which you can change.

You can also use the Ant `property` task:

build.gradle.kts

```
ant.setProperty("buildDir", buildDir)
ant.properties.set("buildDir", buildDir)
ant.properties["buildDir"] = buildDir
ant.withGroovyBuilder {
    "property"("name" to "buildDir", "location" to "buildDir")
}
```

build.gradle

```
ant.buildDir = buildDir
ant.properties.buildDir = buildDir
ant.properties['buildDir'] = buildDir
ant.property(name: 'buildDir', location: buildDir)
```

Many Ant tasks set properties when they execute. There are several ways to get the value of these properties. You can get the property directly from the `AntBuilder` instance. The Ant properties are also available as a Map:

build.xml

```
<property name="antProp" value="a property defined in an Ant build"/>
```

build.gradle.kts

```
println(ant.getProperty("antProp"))
println(ant.properties.get("antProp"))
println(ant.properties["antProp"])
```

build.gradle

```
println ant.antProp
println ant.properties.antProp
println ant.properties['antProp']
```

There are several ways to set an Ant reference:

build.gradle.kts

```
ant.withGroovyBuilder { "path"("id" to "classpath", "location" to "libs") }
ant.references.set("classpath", ant.withGroovyBuilder { "path"("location" to
"libs") })
ant.references["classpath"] = ant.withGroovyBuilder { "path"("location" to
"libs") }
```

build.gradle

```
ant.path(id: 'classpath', location: 'libs')
ant.references.classpath = ant.path(location: 'libs')
ant.references['classpath'] = ant.path(location: 'libs')
```

build.xml

```
<path refid="classpath"/>
```

There are several ways to get an Ant reference:

build.xml

```
<path id="antPath" location="libs"/>
```

build.gradle.kts

```
println(ant.references.get("antPath"))  
println(ant.references["antPath"])
```

build.gradle

```
println ant.references.antPath  
println ant.references['antPath']
```

Using Ant logging

Gradle maps Ant message priorities to Gradle log levels so that messages logged from Ant appear in the Gradle output. By default, these are mapped as follows:

Ant Message Priority	Gradle Log Level
<i>VERBOSE</i>	DEBUG
<i>DEBUG</i>	DEBUG
<i>INFO</i>	INFO
<i>WARN</i>	WARN
<i>ERROR</i>	ERROR

Fine-tuning Ant logging

The default mapping of Ant message priority to the Gradle log level can sometimes be problematic. For example, no message priority maps directly to the **LIFECYCLE** log level, which is the default for Gradle. Many Ant tasks log messages at the *INFO* priority, which means to expose those messages from Gradle, a build would have to be run with the log level set to **INFO**, potentially logging much more output than is desired.

Conversely, if an Ant task logs messages at too high of a level, suppressing those messages would require the build to be run at a higher log level, such as **QUIET**. However, this could result in other desirable outputs being suppressed.

To help with this, Gradle allows the user to fine-tune the Ant logging and control the mapping of message priority to the Gradle log level. This is done by setting the priority that should map to the default Gradle **LIFECYCLE** log level using the [AntBuilder.setLifecycleLogLevel\(java.lang.String\)](#) method. When this value is set, any Ant message logged at the configured priority or above will be logged at least at **LIFECYCLE**. Any Ant message logged below this priority will be logged at **INFO** at most.

For example, the following changes the mapping such that Ant *INFO* priority messages are exposed at the **LIFECYCLE** log level:

build.gradle.kts

```
ant.lifecycleLogLevel = AntBuilder.AntMessagePriority.INFO

tasks.register("hello") {
    doLast {
        ant.withGroovyBuilder {
            "echo"("level" to "info", "message" to "hello from info
priority!")
        }
    }
}
```

build.gradle

```
ant.lifecycleLogLevel = "INFO"

tasks.register('hello') {
    doLast {
        ant.echo(level: "info", message: "hello from info priority!")
    }
}
```

```
$ gradle hello
```

```
> Task :hello
[ant:echo] hello from info priority!
```

```
BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

On the other hand, if the **lifecycleLogLevel** was set to *ERROR*, Ant messages logged at the *WARN* priority would no longer be logged at the **WARN** log level. They would now be logged at the **INFO** level and suppressed by default.

[1] In Groovy you can execute Strings.

OPTIMIZING BUILD PERFORMANCE

Configuring the Build Environment

Configuring the build environment is a powerful way to customize the build process. There are many mechanisms available. By leveraging these mechanisms, you can make your Gradle builds more flexible and adaptable to different environments and requirements.

Available mechanisms

Gradle provides multiple mechanisms for configuring the behavior of Gradle itself and specific projects:

Mechanism	Information	Example
Command line interface	Flags that configure build behavior and Gradle features	<code>--rerun</code>
Project properties	Properties specific to your Gradle project	<code>TestFilter::isFailOnNoMatchingTests=false</code>
System properties	Properties that are passed to the Gradle runtime (JVM)	<code>http.proxyHost=somehost.org</code>
Gradle properties	Properties that configure Gradle settings and the Java process that executes your build	<code>org.gradle.logging.level=quiet</code>
Environment variables	Properties that configure build behavior based on the environment	<code>JAVA_HOME</code>

Priority for configurations

When configuring Gradle behavior, you can use these methods, but you must consider their priority.

The following table lists these methods in order of highest to lowest precedence (the first one wins):

Priority	Method	Location	Notes
1	Command-line	> Command line	Flags have precedence over properties and environment variables
2	System properties	> Project Root Dir	Stored in a <code>gradle.properties</code> file
3	Gradle properties	> <code>GRADLE_USER_HOME</code> > Project Root Dir > <code>GRADLE_HOME</code>	Stored in a <code>gradle.properties</code> file

Priority	Method	Location	Notes
4	Environment variables	> Environment	Sourced by the environment that executes Gradle

Here are all possible configurations of specifying the JDK installation directory in order of priority:

1. Command Line

```
$ ./gradlew exampleTask -Dorg.gradle.java.home=/path/to/your/java/home --scan
```

2. Gradle Properties File

gradle.properties

```
org.gradle.java.home=/path/to/your/java/home
```

3. Environment Variable

```
$ export JAVA_HOME=/path/to/your/java/home
```

The **gradle.properties** file

Gradle properties, system properties, and project properties can be found in the **gradle.properties** file:

gradle.properties

```
# Gradle properties
org.gradle.parallel=true
org.gradle.caching=true
org.gradle.jvmargs=-Duser.language=en -Duser.country=US -Dfile.encoding=UTF-8

# System properties
systemProp.pts.enabled=true
systemProp.log4j2.disableJmx=true
systemProp.file.encoding = UTF-8

# Project properties
kotlin.code.style=official
android.nonTransitiveRClass=false
spring-boot.version = 2.2.1.RELEASE
```

You can place the **gradle.properties** file in the root directory of your project, the Gradle user home directory (**GRADLE_USER_HOME**), or the directory where Gradle is optionally installed (**GRADLE_HOME**).

When resolving properties, Gradle first looks in the project-level `gradle.properties` file, then in the user-level `gradle.properties` file located in `GRADLE_USER_HOME`, and finally in the `gradle.properties` file located in `GRADLE_HOME`, with project-level properties taking precedence over user-level and installation-level properties.

Project properties

Project properties are specific to your Gradle project, they can be used to customize your build. Project properties can be accessed in your build files and get passed in from an external source when your build is executed. Project properties can be retrieved lazily using `providers.gradleProperty()`.

Setting a project property

You have four options to add project properties, listed in order of priority:

1. **Command Line:** You can add project properties directly to your `Project` object via the `-P` command line option.

```
$ ./gradlew build -PmyProperty='Hi, world'
```

2. **System Property:** Gradle creates specially-named system properties for project properties which you can set using the `-D` command line flag or `gradle.properties` file. For the project property `myProperty`, the system property created is called `org.gradle.project.myProperty`.

```
$ ./gradlew build -Dorg.gradle.project.myProperty='Hi, world'
```

gradle.properties

```
org.gradle.project.myProperty='Hi, world'
```

3. **Gradle Properties File:** You can also set project properties in `gradle.properties` files.

gradle.properties

```
myProperty='Hi, world'
```

4. **Environment Variables:** You can set project properties with environment variables. If the environment variable name looks like `ORG_GRADLE_PROJECT_myProperty='Hi, world'`, then Gradle will set a `myProperty` property on your project object, with the value of `Hi, world`.

```
$ export ORG_GRADLE_PROJECT_myProperty='Hi, world'
```

This is typically the preferred method for supplying project properties, especially secrets, to unattended builds like those running on CI servers.

It is possible to change the behavior of a task based on project properties specified at invocation time. Suppose you'd like to ensure release builds are only triggered by CI. A simple way to handle this is through an `isCI` project property:

build.gradle.kts

```
tasks.register("performRelease") {
    val isCI = providers.gradleProperty("isCI")
    doLast {
        if (isCI.isPresent) {
            println("Performing release actions")
        } else {
            throw InvalidUserDataException("Cannot perform release outside of
CI")
        }
    }
}
```

build.gradle

```
tasks.register('performRelease') {
    def isCI = providers.gradleProperty("isCI")
    doLast {
        if (isCI.present) {
            println("Performing release actions")
        } else {
            throw new InvalidUserDataException("Cannot perform release
outside of CI")
        }
    }
}
```

```
$ ./gradlew performRelease -PisCI=true --quiet
Performing release actions
```

Note that running `./gradlew performRelease` yields the same results as long as your `gradle.properties` file includes `isCI=true`:

gradle.properties

```
isCI=true
```

```
$ ./gradlew performRelease --quiet
Performing release actions
```

Command-line flags

The command line interface and the available flags are described in [its own section](#).

System properties

System properties are variables set at the JVM level and accessible to the Gradle build process. System properties can be retrieved lazily using `providers.systemProperty()`.

Setting a system property

You have two options to add system properties listed in order of priority:

1. **Command Line:** Using the `-D` command-line option, you can pass a system property to the JVM, which runs Gradle. The `-D` option of the `gradle` command has the same effect as the `-D` option of the `java` command.

```
$ ./gradlew build -Dgradle.wrapperUser=myuser
```

2. **Gradle Properties File:** You can also set system properties in `gradle.properties` files with the prefix `systemProp`.

gradle.properties

```
systemProp.gradle.wrapperUser=myuser
```

System properties reference

For a quick reference, the following are *common* system properties:

`gradle.wrapperUser=(myuser)`

Specify username to download Gradle distributions [from servers using HTTP Basic Authentication](#).

`gradle.wrapperPassword=(mypassword)`

Specify password for downloading a Gradle distribution using the Gradle wrapper.

`gradle.user.home=(path to directory)`

Specify the `GRADLE_USER_HOME` directory.

`https.protocols`

Specify the supported TLS versions in a comma-separated format. e.g., `TLSv1.2,TLSv1.3`.

Additional Java system properties are listed [here](#).

In a multi-project build, `systemProp` properties set in any project except the root will be **ignored**. Only the root project's `gradle.properties` file will be checked for properties that begin with `systemProp`.

Gradle properties

Gradle properties configure Gradle itself and usually have the name `org.gradle.*`. Gradle properties should not be used in build logic, their values should not be read/retrieved.

Setting a Gradle property

You have two options to add Gradle properties listed in order of priority:

1. **Command Line:** Using the `-D` command-line option, you can pass a Gradle property:

```
$ ./gradlew build -Dorg.gradle.caching.debug=false
```

2. **Gradle Properties File:** Place these settings into a `gradle.properties` file and commit it to your version control system.

gradle.properties

```
org.gradle.caching.debug=false
```

The final configuration considered by Gradle is a combination of all Gradle properties set on the command line and your `gradle.properties` files. If an option is configured in multiple locations, the *first one* found in any of these locations wins:

Priority	Method	Location	Details
1	Command line interface	.	In the command line using <code>-D</code> .
2	<code>gradle.properties</code> file	<code>GRADLE_USER_HOME</code>	Stored in a <code>gradle.properties</code> file in the <code>GRADLE_USER_HOME</code> .
3	<code>gradle.properties</code> file	Project Root Dir	Stored in a <code>gradle.properties</code> file in a project directory, then its parent project's directory up to the project's root directory.
4	<code>gradle.properties</code> file	<code>GRADLE_HOME</code>	Stored in a <code>gradle.properties</code> file in the <code>GRADLE_HOME</code> , the optional Gradle installation directory.

NOTE

The location of the `GRADLE_USER_HOME` may have been changed beforehand via the `-Dgradle.user.home` system property passed on the command line.

Gradle properties reference

For reference, the following properties are common Gradle properties:

`org.gradle.caching=(true,false)`

When set to `true`, Gradle will reuse task outputs from any previous build when possible, resulting in much faster builds.

Default is `false`; the build cache is **not** enabled.

`org.gradle.caching.debug=(true,false)`

When set to `true`, individual input property hashes and the build cache key for each task are logged on the console.

Default is `false`.

`org.gradle.configuration-cache=(true,false)`

Enables configuration caching. Gradle will try to reuse the build configuration from previous builds.

Default is `false`.

`org.gradle.configureondemand=(true,false)`

Enables incubating configuration-on-demand, where Gradle will attempt to configure only necessary projects.

Default is `false`.

`org.gradle.console=(auto,plain,rich,verbose)`

Customize console output coloring or verbosity.

Default depends on how Gradle is invoked.

`org.gradle.continue=(true,false)`

If enabled, continue task execution after a task failure, else stop task execution after a task failure.

Default is `false`.

`org.gradle.daemon=(true,false)`

When set to `true` the Gradle Daemon is used to run the build.

Default is `true`.

`org.gradle.daemon.idletimeout=(# of idle millis)`

Gradle Daemon will terminate itself after a specified number of idle milliseconds.

Default is 10800000 (3 hours).

org.gradle.debug=(true,false)

When set to `true`, Gradle will run the build with remote debugging enabled, listening on port 5005. Note that this is equivalent to adding `-agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005` to the JVM command line and will suspend the virtual machine until a debugger is attached.

Default is `false`.

org.gradle.java.home=(path to JDK home)

Specifies the Java home for the Gradle build process. The value can be set to either a `jdk` or `jre` location; however, using a JDK is safer depending on what your build does. This does not affect the version of Java used to launch the [Gradle client VM](#).

You can also control the JVM used to run Gradle itself using the [Daemon JVM criteria](#).

Default is derived from your environment (`JAVA_HOME` or the path to `java`) if the setting is unspecified.

org.gradle.jvmargs=(JVM arguments)

Specifies the JVM arguments used for the Gradle Daemon. The setting is particularly useful for [configuring JVM memory settings](#) for build performance. This does not affect the JVM settings for the Gradle client VM.

Default is `-Xmx512m "-XX:MaxMetaspaceSize=384m"`.

org.gradle.logging.level=(quiet,warn,lifecycle,info,debug)

When set to `quiet`, `warn`, `info`, or `debug`, Gradle will use this [log level](#). The values are not case-sensitive.

Default is `lifecycle` level.

org.gradle.parallel=(true,false)

When configured, Gradle will fork up to `org.gradle.workers.max` JVMs to execute [projects in parallel](#).

Default is `false`.

org.gradle.priority=(low,normal)

Specifies the [scheduling priority](#) for the Gradle daemon and all processes launched by it.

Default is `normal`.

org.gradle.projectcachedir=(directory)

Specify the project-specific cache directory. Defaults to `.gradle` in the root project directory."

Default is `.gradle`.

org.gradle.problems.report=(true|false)

Enable (**true**) or disable (**false**) the generation of `build/reports/problems-report.html`. **true** is the default. The report is generated with problems provided to the [Problems API](#).

org.gradle.unsafe.isolated-projects=(true,false)

Enables project isolation, which enables configuration caching.

*Default is **false**.*

org.gradle.vfs.verbose=(true,false)

Configures verbose logging when [watching the file system](#).

*Default is **false**.*

org.gradle.vfs.watch=(true,false)

Toggles [watching the file system](#). When enabled, Gradle reuses information it collects about the file system between builds.

*Default is **true** on operating systems where Gradle supports this feature.*

org.gradle.warning.mode=(all,fail,summary,none)

When set to **all**, **summary**, or **none**, Gradle will use [different warning type display](#).

*Default is **summary**.*

org.gradle.workers.max=(max # of worker processes)

When configured, Gradle will use a maximum of the [given number of workers](#).

Default is the number of CPU processors.

Environment variables

Gradle provides a number of environment variables, which are listed below. Environment variables can be retrieved lazily using `providers.environmentVariable()`.

Setting environment variables

Let's take an example that sets the `$JAVA_HOME` environment variable:

```
$ set JAVA_HOME=C:\Path\To\Your\Java\Home // Windows
$ export JAVA_HOME=/path/to/your/java/home // Mac/Linux
```

You can access environment variables as properties in the build script using the `System.getenv()` method:

```
task printEnvVariables {
    doLast {
        println "JAVA_HOME: ${System.getenv('JAVA_HOME')}}"
    }
}
```

```
}  
}
```

Environment variables reference

The following environment variables are available for the `gradle` command:

GRADLE_HOME

Installation directory for Gradle.

Can be used to specify a local Gradle version instead of using the wrapper.

You can add `GRADLE_HOME/bin` to your `PATH` for specific applications and use cases (such as testing an early release for Gradle).

JAVA_OPTS

Used to pass JVM options and custom settings to the JVM.

```
export JAVA_OPTS="-Xmx18928m -XX:+HeapDumpOnOutOfMemoryError -Dfile.encoding=UTF-8  
-Djava.awt.headless=true -Dkotlin.daemon.jvm.options=-Xmx6309m"
```

GRADLE_OPTS

Specifies JVM arguments to use when starting the Gradle client VM.

The client VM only handles command line input/output, so one would rarely need to change its VM options.

The actual build is run by the Gradle daemon, which is not affected by this environment variable.

GRADLE_USER_HOME

Specifies the `GRADLE_USER_HOME` directory for Gradle to store its global configuration properties, initialization scripts, caches, log files and more.

Defaults to `USER_HOME/.gradle` if not set.

JAVA_HOME

Specifies the JDK installation directory to use for the client VM.

This VM is also used for the daemon unless a different one is specified in a Gradle properties file with `org.gradle.java.home` or using the [Daemon JVM criteria](#).

GRADLE_LIBS_REPO_OVERRIDE

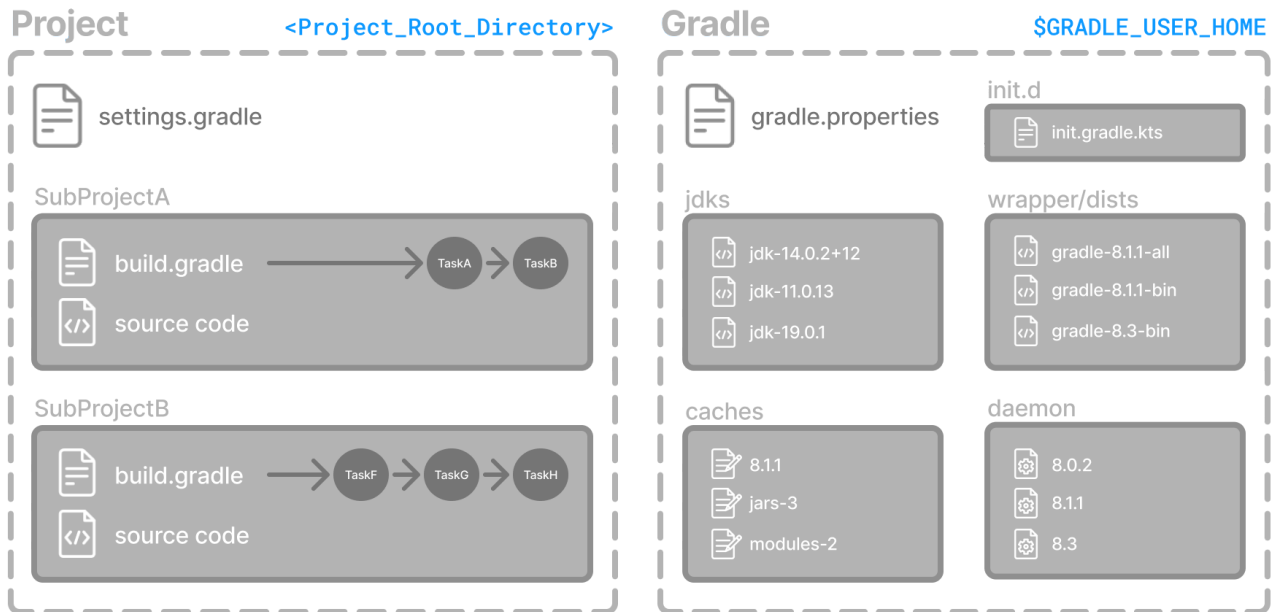
Overrides for the default Gradle library repository.

Can be used to specify a default Gradle repository URL in `org.gradle.plugins.ide.internal.resolver`.

Useful override to specify an internally hosted repository if your company uses a firewall/proxy.

Gradle-managed Directories

Gradle uses two main directories to perform and manage its work: the [Gradle User Home directory](#) and the [Project Root directory](#).



Gradle User Home directory

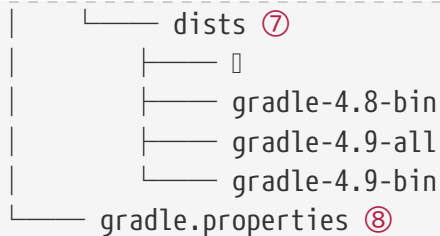
By default, the Gradle User Home (`~/.gradle` or `C:\Users\<USERNAME>\.gradle`) stores global configuration properties, initialization scripts, caches, and log files.

It can be set with the environment variable `GRADLE_USER_HOME`.

TIP Not to be confused with the `GRADLE_HOME`, the optional installation directory for Gradle.

It is roughly structured as follows:

```
├── caches ①
│   ├── 4.8 ②
│   ├── 4.9 ②
│   └── 
│       ├── jars-3 ③
│       └── modules-2 ③
├── daemon ④
│   ├── 
│   ├── 4.8
│   └── 4.9
├── init.d ⑤
│   └── my-setup.gradle
├── jdks ⑥
│   └── 
│       └── jdk-14.0.2+12
└── wrapper
```



① Global cache directory (for everything that is not project-specific).

② Version-specific caches (e.g., to support incremental builds).

③ Shared caches (e.g., for artifacts of dependencies).

④ Registry and logs of the [Gradle Daemon](#).

⑤ Global [initialization scripts](#).

⑥ JDKs downloaded by the [toolchain support](#).

⑦ Distributions downloaded by the [Gradle Wrapper](#).

⑧ Global [Gradle configuration properties](#).

Cleanup of caches and distributions

Gradle automatically cleans its user home directory.

By default, the cleanup runs in the background when the Gradle daemon is stopped or shut down.

If using `--no-daemon`, it runs in the foreground after the build session.

The following cleanup strategies are applied periodically (by default, once every 24 hours):

- Version-specific caches in all `caches/<GRADLE_VERSION>/` directories are checked for whether they are still in use.

If not, directories for release versions are deleted after 30 days of inactivity, and snapshot versions after 7 days.

- Shared caches in `caches/` (e.g., `jars-*`) are checked for whether they are still in use.

If no Gradle version still uses them, they are deleted.

- Files in shared caches used by the current Gradle version in `caches/` (e.g., `jars-3` or `modules-2`) are checked for when they were last accessed.

Depending on whether the file can be recreated locally or downloaded from a remote repository, it will be deleted after 7 or 30 days, respectively.

- Gradle distributions in `wrapper/dists/` are checked for whether they are still in use, i.e., whether there's a corresponding version-specific cache directory.

Unused distributions are deleted.

Configuring cleanup of caches and distributions

The retention periods of the various caches can be configured.

Caches are classified into five categories:

1. **Released wrapper distributions:** Distributions and related version-specific caches corresponding to released versions (e.g., **4.6.2** or **8.0**).

Default retention for unused versions is 30 days.

2. **Snapshot wrapper distributions:** Distributions and related version-specific caches corresponding to snapshot versions (e.g. **7.6-20221130141522+0000**).

Default retention for unused versions is 7 days.

3. **Downloaded resources:** Shared caches downloaded from a remote repository (e.g., cached dependencies).

Default retention for unused resources is 30 days.

4. **Created resources:** Shared caches that Gradle creates during a build (e.g., artifact transforms).

Default retention for unused resources is 7 days.

5. **Build cache:** The local build cache (e.g., build-cache-1).

Default retention for unused build cache entries is 7 days.

The retention period for each category can be configured independently via an [init script](#) in the Gradle User Home:

gradleUserHome/init.d/cache-settings.gradle.kts

```
beforeSettings {  
    caches {  
        releasedWrappers.setRemoveUnusedEntriesAfterDays(45)  
        snapshotWrappers.setRemoveUnusedEntriesAfterDays(10)  
        downloadedResources.setRemoveUnusedEntriesAfterDays(45)  
        createdResources.setRemoveUnusedEntriesAfterDays(10)  
        buildCache.setRemoveUnusedEntriesAfterDays(5)  
    }  
}
```

gradleUserHome/init.d/cache-settings.gradle

```
beforeSettings { settings ->  
    settings.caches {  
        releasedWrappers.removeUnusedEntriesAfterDays = 45
```

```

        snapshotWrappers.removeUnusedEntriesAfterDays = 10
        downloadedResources.removeUnusedEntriesAfterDays = 45
        createdResources.removeUnusedEntriesAfterDays = 10
        buildCache.removeUnusedEntriesAfterDays = 5
    }
}

```

The frequency at which cache cleanup is invoked is also configurable.

There are three possible settings:

1. **DEFAULT:** Cleanup is performed periodically in the background (currently once every 24 hours).
2. **DISABLED:** Never cleanup Gradle User Home.

This is useful in cases where Gradle User Home is ephemeral or delaying cleanup is desirable until an explicit point.

3. **ALWAYS:** Cleanup is performed at the end of each build session.

This is useful in cases where it's desirable to ensure that cleanup has occurred before proceeding.

However, this performs cache cleanup during the build (rather than in the background), which can be expensive, so this option should only be used when necessary.

To disable cache cleanup:

gradleUserHome/init.d/cache-settings.gradle.kts

```

beforeSettings {
    caches {
        cleanup = Cleanup.DISABLED
    }
}

```

gradleUserHome/init.d/cache-settings.gradle

```

beforeSettings { settings ->
    settings.caches {
        cleanup = Cleanup.DISABLED
    }
}

```

NOTE

Cache cleanup settings can only be configured via init scripts and should be placed under the `init.d` directory in Gradle User Home. This effectively couples the configuration of cache cleanup to the Gradle User Home those settings apply to and limits the possibility of different conflicting settings from different projects being applied to the same directory.

Multiple versions of Gradle sharing a Gradle User Home

It is common to share a single Gradle User Home between multiple versions of Gradle.

As stated above, caches in Gradle User Home are version-specific. Different versions of Gradle will perform maintenance on only the version-specific caches associated with each version.

On the other hand, some caches are shared between versions (e.g., the dependency artifact cache or the artifact transform cache).

Beginning with Gradle version 8.0, the cache cleanup settings can be configured to custom retention periods. However, older versions have fixed retention periods (7 or 30 days, depending on the cache). These shared caches could be accessed by versions of Gradle with different settings to retain cache artifacts.

This means that:

- If the retention period is *not* customized, all versions that perform cleanup will have the same retention periods. There will be no effect due to sharing a Gradle User Home with multiple versions.
- If the retention period is customized for Gradle versions greater than or equal to version 8.0 to use retention periods *shorter* than the previously fixed periods, there will also be no effect.

The versions of Gradle aware of these settings will cleanup artifacts earlier than the previously fixed retention periods, and older versions will effectively not participate in the cleanup of shared caches.

- If the retention period is customized for Gradle versions greater than or equal to version 8.0 to use retention periods *longer* than the previously fixed periods, the older versions of Gradle may clean the shared caches earlier than what is configured.

In this case, if it is desirable to maintain these shared cache entries for newer versions for longer retention periods, they will not be able to share a Gradle User Home with older versions. They will need to use a separate directory.

Another consideration when sharing the Gradle User Home with versions of Gradle before version 8.0 is that the DSL elements to configure the cache retention settings are unavailable in earlier versions, so this must be accounted for in any init script shared between versions. This can easily be handled by conditionally applying a version-compliant script.

NOTE

The version-compliant script should reside somewhere other than the `init.d` directory (such as a sub-directory), so it is not automatically applied.

To configure cache cleanup in a version-safe manner:

gradleUserHome/init.d/cache-settings.gradle.kts

```
if (GradleVersion.current() >= GradleVersion.version("8.0")) {  
    apply(from = "gradle8/cache-settings.gradle.kts")  
}
```

gradleUserHome/init.d/cache-settings.gradle

```
if (GradleVersion.current() >= GradleVersion.version('8.0')) {  
    apply from: "gradle8/cache-settings.gradle"  
}
```

Version-compliant cache configuration script:

gradleUserHome/init.d/gradle8/cache-settings.gradle.kts

```
beforeSettings {  
    caches {  
        releasedWrappers { setRemoveUnusedEntriesAfterDays(45) }  
        snapshotWrappers { setRemoveUnusedEntriesAfterDays(10) }  
        downloadedResources { setRemoveUnusedEntriesAfterDays(45) }  
        createdResources { setRemoveUnusedEntriesAfterDays(10) }  
        buildCache { setRemoveUnusedEntriesAfterDays(5) }  
    }  
}
```

gradleUserHome/init.d/gradle8/cache-settings.gradle

```
beforeSettings { settings ->  
    settings.caches {  
        releasedWrappers.removeUnusedEntriesAfterDays = 45  
        snapshotWrappers.removeUnusedEntriesAfterDays = 10  
        downloadedResources.removeUnusedEntriesAfterDays = 45  
        createdResources.removeUnusedEntriesAfterDays = 10  
        buildCache.removeUnusedEntriesAfterDays = 5  
    }  
}
```

Cache marking

Beginning with Gradle version 8.1, Gradle supports marking caches with a **CACHEDIR.TAG** file.

It follows the format described in [the Cache Directory Tagging Specification](#). The purpose of this file is to allow tools to identify the directories that do not need to be searched or backed up.

By default, the directories **caches**, **wrapper/dists**, **daemon**, and **jdk**s in the Gradle User Home are marked with this file.

Configuring cache marking

The cache marking feature can be configured via an init script in the Gradle User Home:

gradleUserHome/init.d/cache-settings.gradle.kts

```
beforeSettings {
    caches {
        // Disable cache marking for all caches
        markingStrategy = MarkingStrategy.NONE
    }
}
```

gradleUserHome/init.d/cache-settings.gradle

```
beforeSettings { settings ->
    settings.caches {
        // Disable cache marking for all caches
        markingStrategy = MarkingStrategy.NONE
    }
}
```

NOTE

Cache marking settings can only be configured via init scripts and should be placed under the **init.d** directory in Gradle User Home. This effectively couples the configuration of cache marking to the Gradle User Home to which those settings apply and limits the possibility of different conflicting settings from different projects being applied to the same directory.

Project Root directory

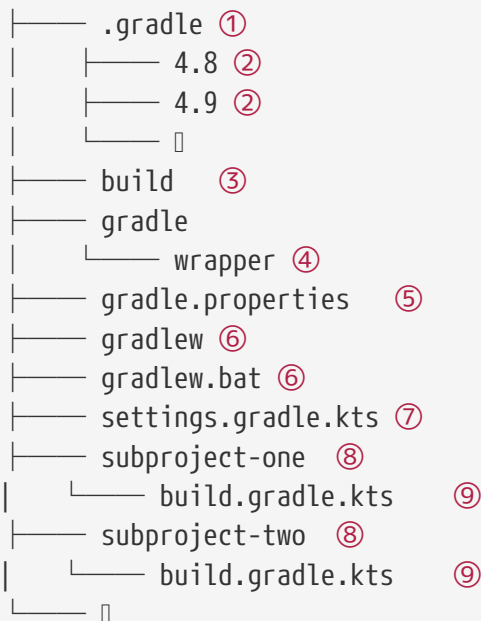
The project root directory contains all source files from your project.

It also contains files and directories Gradle generates, such as **.gradle** and **build**.

While the former are usually checked into source control, the latter are transient files Gradle uses

to support features like incremental builds.

The anatomy of a typical project root directory looks as follows:



- ① Project-specific cache directory generated by Gradle.
- ② Version-specific caches (e.g., to support incremental builds).
- ③ The build directory of this project into which Gradle generates all build artifacts.
- ④ Contains the JAR file and configuration of the [Gradle Wrapper](#).
- ⑤ Project-specific [Gradle configuration properties](#).
- ⑥ Scripts for executing builds using the [Gradle Wrapper](#).
- ⑦ The project's [settings file](#) where the list of subprojects is defined.
- ⑧ Usually, a project is organized into one or multiple subprojects.
- ⑨ Each subproject has its own Gradle build script.

Project cache cleanup

From version 4.10 onwards, Gradle automatically cleans the project-specific cache directory.

After building the project, version-specific cache directories in `.gradle/8.13/` are checked periodically (at most, every 24 hours) to determine whether they are still in use. They are deleted if they haven't been used for 7 days.

Next Step: [Learn about the Gradle Build Lifecycle >>](#)

Logging

The log serves as the primary 'UI' of a build tool. If it becomes overly verbose, important warnings and issues can be obscured. However, it is essential to have relevant information to determine if something has gone wrong.

Gradle defines six log levels, detailed in [Log levels](#). In addition to the standard log levels, Gradle introduces two specific levels: *QUIET* and *LIFECYCLE*. *LIFECYCLE* is the default level used to report build progress.

Understanding Log levels

There are 6 log levels in Gradle:

ERROR Error messages

QUIET Important information messages

WARNING Warning messages

LIFECYCLE Progress information messages

INFO Information messages

DEBUG Debug messages

NOTE

The console's rich components (build status and work-in-progress area) are displayed regardless of the log level used.

Choosing a log level

You can choose different log levels from the command line switches shown in [Log level command-line options](#).

You can also configure the log level using `gradle.properties`.

In [Stacktrace command-line options](#) you can find the command line switches which affect stacktrace logging.

Log level command-line options:

Option	Outputs Log Levels
<code>-q</code> or <code>--quiet</code>	QUIET and higher
<code>-w</code> or <code>--warn</code>	WARN and higher
no logging options	LIFECYCLE and higher
<code>-i</code> or <code>--info</code>	INFO and higher
<code>-d</code> or <code>--debug</code>	DEBUG and higher (that is, all log messages)

CAUTION

The **DEBUG** log level can [expose sensitive security information to the console](#).

Stacktrace command-line options

`-s` or `--stacktrace`

Truncated stacktraces are printed. We recommend this over full stacktraces. Groovy full

stacktraces are extremely verbose due to the underlying dynamic invocation mechanisms. Yet they usually do not contain relevant information about what has gone wrong in *your* code. This option renders stacktraces for deprecation warnings.

-S or --full-stacktrace

The full stacktraces are printed out. This option renders stacktraces for deprecation warnings.

<No stacktrace options>

No stacktraces are printed to the console in case of a build error (e.g., a compile error). Only in case of internal exceptions will stacktraces be printed. If the **DEBUG** log level is chosen, truncated stacktraces are always printed.

Logging Sensitive Information

Running Gradle with the **DEBUG** log level can potentially expose sensitive information to the console and build log.

This information might include:

- Environment variables
- Private repository credentials
- Build cache and Develocity credentials
- Plugin Portal publishing credentials

It's important to avoid using the **DEBUG** log level when running on public Continuous Integration (CI) services. Build logs on these services are accessible to the public and can expose sensitive information. Even on private CI services, logging sensitive credentials may pose a risk depending on your organization's threat model. It's advisable to discuss this with your organization's security team.

Some CI providers attempt to redact sensitive credentials from logs, but this process is not foolproof and typically only redacts exact matches of pre-configured secrets.

If you suspect that a Gradle Plugin may inadvertently expose sensitive information, please contact [our security team](#) for assistance with disclosure.

Writing your own log messages

A simple option for logging in your build file is to write messages to standard output. Gradle redirects anything written to standard output to its logging system at the **QUIET** log level:

```
build.gradle.kts
```

```
println("A message which is logged at QUIET level")
```

build.gradle

```
println 'A message which is logged at QUIET level'
```

Gradle also provides a `logger` property to a build script, which is an instance of `Logger`. This interface extends the SLF4J `Logger` interface and adds a few Gradle-specific methods. Below is an example of how this is used in the build script:

build.gradle.kts

```
logger.quiet("An info log message which is always logged.")
logger.error("An error log message.")
logger.warn("A warning log message.")
logger.lifecycle("A lifecycle info log message.")
logger.info("An info log message.")
logger.debug("A debug log message.")
logger.trace("A trace log message.") // Gradle never logs TRACE level logs
```

build.gradle

```
logger.quiet('An info log message which is always logged.')
logger.error('An error log message.')
logger.warn('A warning log message.')
logger.lifecycle('A lifecycle info log message.')
logger.info('An info log message.')
logger.debug('A debug log message.')
logger.trace('A trace log message.') // Gradle never logs TRACE level logs
```

Use the link [typical SLF4J pattern](#) to replace a placeholder with an actual value in the log message.

build.gradle.kts

```
logger.info("A {} log message", "info")
```

build.gradle

```
logger.info('A {} log message', 'info')
```

You can also hook into Gradle's logging system from within other classes used in the build (classes from the `buildSrc` directory, for example) with an SLF4J logger. You can use this logger the same way as you use the provided logger in the build script.

build.gradle.kts

```
import org.slf4j.LoggerFactory

val slf4jLogger = LoggerFactory.getLogger("some-logger")
slf4jLogger.info("An info log message logged using SLF4j")
```

build.gradle

```
import org.slf4j.LoggerFactory

def slf4jLogger = LoggerFactory.getLogger('some-logger')
slf4jLogger.info('An info log message logged using SLF4j')
```

Logging from external tools and libraries

Internally, Gradle uses Ant and Ivy. Both have their own logging system. Gradle redirects their logging output into the Gradle logging system.

There is a 1:1 mapping from the Ant/Ivy log levels to the Gradle log levels, except the Ant/Ivy `TRACE` log level, which is mapped to the Gradle `DEBUG` log level. This means the default Gradle log level will not show any Ant/Ivy output unless it is an error or a warning.

Many tools out there still use the standard output for logging. By default, Gradle redirects standard output to the `QUIET` log level and standard error to the `ERROR` level. This behavior is configurable.

The `project` object provides a `LoggingManager`, which allows you to change the log levels that standard out or error are redirected to when your build script is evaluated.

build.gradle.kts

```
logging.captureStandardOutput(LogLevel.INFO)
println("A message which is logged at INFO level")
```

build.gradle

```
logging.captureStandardOutput LogLevel.INFO
```

```
println 'A message which is logged at INFO level'
```

To change the log level for standard out or error during task execution, use a [LoggingManager](#).

build.gradle.kts

```
tasks.register("logInfo") {
    logging.captureStandardOutput(LogLevel.INFO)
    doFirst {
        println("A task message which is logged at INFO level")
    }
}
```

build.gradle

```
tasks.register('logInfo') {
    logging.captureStandardOutput LogLevel.INFO
    doFirst {
        println 'A task message which is logged at INFO level'
    }
}
```

Gradle also integrates with the [Java Util Logging](#), Jakarta Commons Logging and [Log4j](#) logging toolkits. Any log messages your build classes write using these logging toolkits will be redirected to Gradle's logging system.

Changing what Gradle logs

WARNING

This feature is deprecated and will be removed in the next major version without a replacement.

The [configuration cache](#) limits the ability to customize Gradle's logging UI. The custom logger can only implement [supported listener interfaces](#). These interfaces do not receive events when the configuration cache entry is reused because the configuration phase is skipped.

You can replace much of Gradle's logging UI with your own. You could do this if you want to customize the UI somehow - to log more or less information or to change the formatting. Simply replace the logging using the [Gradle.useLogger\(java.lang.Object\)](#) method. This is accessible from a build script, an init script, or via the embedding API. Note that this completely disables Gradle's default output. Below is an example init script that changes how task execution and build completion are logged:

customLogger.init.gradle.kts

```
useLogger(CustomEventLogger())

@Suppress("deprecation")
class CustomEventLogger() : BuildAdapter(), TaskExecutionListener {

    override fun beforeExecute(task: Task) {
        println("[${task.name}]")
    }

    override fun afterExecute(task: Task, state: TaskState) {
        println()
    }

    override fun buildFinished(result: BuildResult) {
        println("build completed")
        if (result.failure != null) {
            (result.failure as Throwable).printStackTrace()
        }
    }
}
```

customLogger.init.gradle

```
useLogger(new CustomEventLogger())

@SuppressWarnings("deprecation")
class CustomEventLogger extends BuildAdapter implements TaskExecutionListener
{

    void beforeExecute(Task task) {
        println "[${task.name}]"
    }

    void afterExecute(Task task, TaskState state) {
        println()
    }

    void buildFinished(BuildResult result) {
        println 'build completed'
        if (result.failure != null) {
            result.failure.printStackTrace()
        }
    }
}
```

```
$ gradle -I customLogger.init.gradle.kts build
```

```
> Task :compile  
[compile]  
compiling source
```

```
> Task :testCompile  
[testCompile]  
compiling test source
```

```
> Task :test  
[test]  
running unit tests
```

```
> Task :build  
[build]
```

```
build completed  
3 actionable tasks: 3 executed
```

```
$ gradle -I customLogger.init.gradle build
```

```
> Task :compile  
[compile]  
compiling source
```

```
> Task :testCompile  
[testCompile]  
compiling test source
```

```
> Task :test  
[test]  
running unit tests
```

```
> Task :build  
[build]
```

```
build completed  
3 actionable tasks: 3 executed
```

Your logger can implement any of the listener interfaces listed below. When you register a logger, only the logging for the interfaces it implements is replaced. Logging for the other interfaces is left untouched. You can find out more about the listener interfaces in [Build lifecycle events](#).

- [BuildListener](#)^[1]
- [ProjectEvaluationListener](#)
- [TaskExecutionGraphListener](#)

- [TaskExecutionListener](#)^[1]
- [TaskActionListener](#)^[1]

Configuring Gradle

Configuring JVM memory

The `org.gradle.jvmargs` Gradle property controls the VM running the build. It defaults to `-Xmx512m -XX:MaxMetaspaceSize=384m`

You can adjust JVM options for Gradle in the following ways.

Option 1: Changing JVM settings for the build VM:

```
org.gradle.jvmargs=-Xmx2g -XX:MaxMetaspaceSize=512m
-XX:+HeapDumpOnOutOfMemoryError -Dfile.encoding=UTF-8
```

The `JAVA_OPTS` environment variable controls the command line client, which is only used to display console output. It defaults to `-Xmx64m`

Option 2: Changing JVM settings for the client VM:

```
JAVA_OPTS="-Xmx64m -XX:+HeapDumpOnOutOfMemoryError -Dfile.encoding=UTF-8"
```

NOTE

There is one case where the client VM can also serve as the build VM:

If you deactivate the [Gradle Daemon](#) and the client VM has the same settings as required for the build VM, the client VM will run the build directly.

Otherwise, the client VM will fork a new VM to run the actual build in order to honor the different settings.

Certain tasks, like the `test` task, also fork additional JVM processes. You can configure these through the tasks themselves. They use `-Xmx512m` by default.

Example 1: Set compile options for Java compilation tasks:

build.gradle.kts

```
plugins {
    java
}
```

```
tasks.withType<JavaCompile>().configureEach {
    options.compilerArgs = listOf("-Xdoclint:none", "-Xlint:none", "-nowarn")
}
```

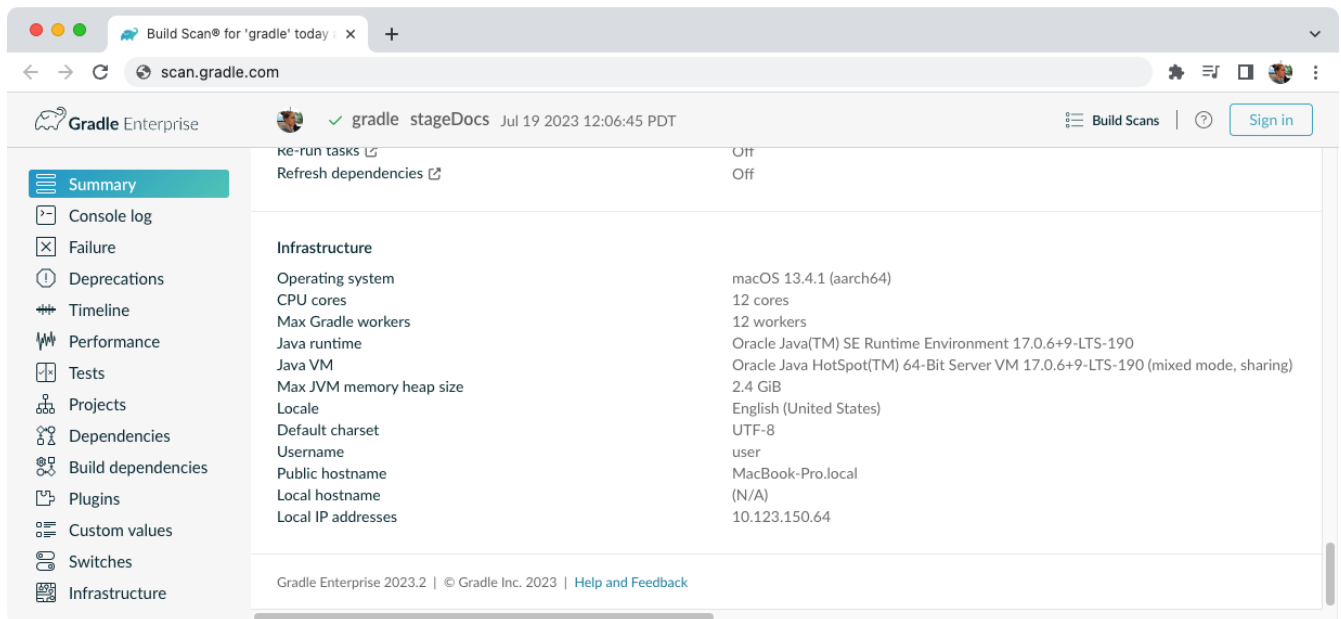
build.gradle

```
plugins {
    id 'java'
}

tasks.withType(JavaCompile).configureEach {
    options.compilerArgs += ['-Xdoclint:none', '-Xlint:none', '-nowarn']
}
```

See other examples in the [Test API documentation](#) and [test execution in the Java plugin reference](#).

Build scans will tell you information about the JVM that executed the build when you use the `--scan` option:



The screenshot shows the Gradle Enterprise Build Scans web interface. The browser address bar shows `scan.gradle.com`. The page header includes the Gradle logo, the text "Gradle Enterprise", a green checkmark, the build ID "gradle stageDocs", the timestamp "Jul 19 2023 12:06:45 PDT", and a "Sign in" button. A left sidebar contains a menu with items: Summary (selected), Console log, Failure, Deprecations, Timeline, Performance, Tests, Projects, Dependencies, Build dependencies, Plugins, Custom values, Switches, and Infrastructure. The main content area displays build details. At the top, it shows "Re-run tasks" and "Refresh dependencies" both set to "Off". Below this, the "Infrastructure" section lists various system properties:

Property	Value
Operating system	macOS 13.4.1 (aarch64)
CPU cores	12 cores
Max Gradle workers	12 workers
Java runtime	Oracle Java(TM) SE Runtime Environment 17.0.6+9-LTS-190
Java VM	Oracle Java HotSpot(TM) 64-Bit Server VM 17.0.6+9-LTS-190 (mixed mode, sharing)
Max JVM memory heap size	2.4 GiB
Locale	English (United States)
Default charset	UTF-8
Username	user
Public hostname	MacBook-Pro.local
Local hostname	(N/A)
Local IP addresses	10.123.150.64

At the bottom of the page, it says "Gradle Enterprise 2023.2 | © Gradle Inc. 2023 | [Help and Feedback](#)".

Configuring a task using project properties

It is possible to change the behavior of a task based on project properties specified at invocation time.

Suppose you would like to ensure release builds are only triggered by CI. A simple way to do this is using the `isCI` project property.

Example 1: Prevent releasing outside of CI:

build.gradle.kts

```
tasks.register("performRelease") {
    val isCI = providers.gradleProperty("isCI")
    doLast {
        if (isCI.isPresent) {
            println("Performing release actions")
        } else {
            throw InvalidUserDataException("Cannot perform release outside of
CI")
        }
    }
}
```

build.gradle

```
tasks.register('performRelease') {
    def isCI = providers.gradleProperty("isCI")
    doLast {
        if (isCI.present) {
            println("Performing release actions")
        } else {
            throw new InvalidUserDataException("Cannot perform release
outside of CI")
        }
    }
}
```

```
$ gradle performRelease -PisCI=true --quiet
Performing release actions
```

Improve the Performance of Gradle Builds

Build performance is critical to productivity. The longer builds take to complete, the more likely they'll disrupt your development flow. Builds run many times a day, so even small waiting periods add up. The same is true for Continuous Integration (CI) builds: the less time they take, the faster you can react to new issues and the more often you can experiment.

All this means that it's worth investing some time and effort into making your build as fast as possible. This section offers several ways to make a build faster. Additionally, you'll find details about what leads to build performance degradation, and how you can avoid it.

TIP

Want faster Gradle Builds? [Register here](#) for our Build Cache training session to learn how Develocity can speed up builds by up to 90%.

Inspect your build

Before you make any changes, [inspect your build](#) with a build scan or profile report. A proper build inspection helps you understand:

- how long it takes to build your project
- which parts of your build are slow

Inspecting provides a comparison point to better understand the impact of the changes recommended on this page.

To best make use of this page:

1. Inspect your build.
2. Make a change.
3. Inspect your build again.

If the change improved build times, make it permanent. If you don't see an improvement, remove the change and try another.

Update versions

Gradle

The Gradle team continuously improves the performance of Gradle builds. If you're using an old version of Gradle, you're missing out on the benefits of that work. Keeping up with Gradle version upgrades is low risk because the Gradle team ensures backwards compatibility between minor versions of Gradle. Staying up-to-date also makes transitioning to the next major version easier, since you'll get early deprecation warnings.

Java

Gradle runs on the Java Virtual Machine (JVM). Java performance improvements often benefit Gradle. For the best Gradle performance, use the latest version of Java.

Plugins

Plugin writers continuously improve the performance of their plugins. If you're using an old version of a plugin, you're missing out on the benefits of that work. The Android, Java, and Kotlin plugins in particular can significantly impact build performance. Update to the latest version of these plugins for performance improvements.

Enable parallel execution

Most projects consist of more than one subproject. Usually, some of those subprojects are

independent of one another; that is, they do not share state. Yet by default, Gradle only runs one task at a time. To execute tasks belonging to different subprojects in parallel, use the `parallel` flag:

```
$ gradle <task> --parallel
```

To execute project tasks in parallel by default, add the following setting to the `gradle.properties` file in the project root or your Gradle home:

gradle.properties

```
org.gradle.parallel=true
```

Parallel builds can significantly improve build times; how much depends on your project structure and how many dependencies you have between subprojects. A build whose execution time is dominated by a single subproject won't benefit much at all. Neither will a project with lots of inter-subproject dependencies. But most multi-subproject builds see a reduction in build times.

Visualize parallelism with build scans

Build scans give you a visual timeline of task execution. In the following example build, you can see long-running tasks at the beginning and end of the build:

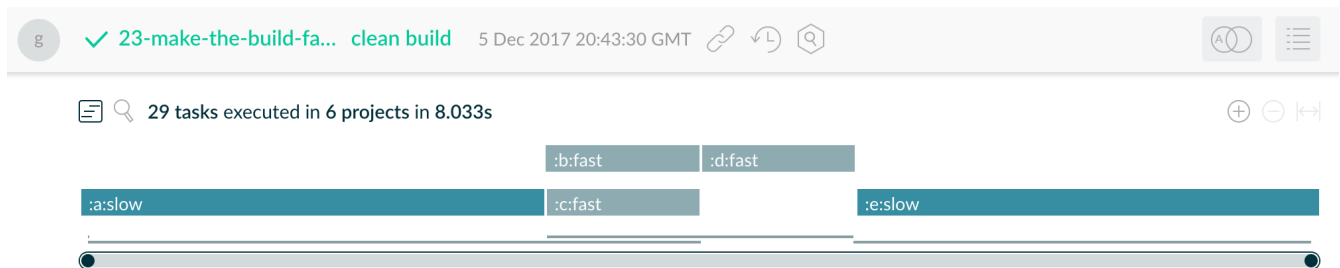


Figure 1. Bottleneck in parallel execution

Tweaking the build configuration to run the two slow tasks early on and in parallel reduces the overall build time from 8 seconds to 5 seconds:

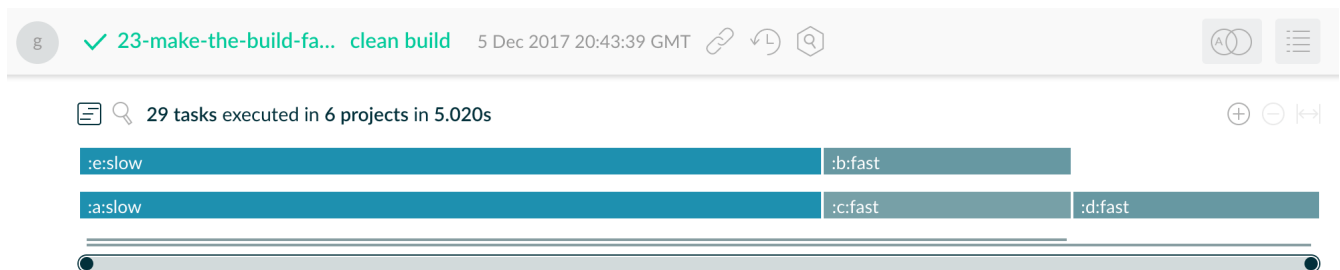


Figure 2. Optimized parallel execution

Re-enable the Gradle Daemon

The Gradle Daemon reduces build times by:

- caching project information across builds
- running in the background so every Gradle build doesn't have to wait for JVM startup
- benefiting from continuous runtime optimization in the JVM
- watching the file system to calculate exactly what needs to be rebuilt before you run a build

Gradle enables the Daemon by default, but some builds override this preference. If your build disables the Daemon, you could see a significant performance improvement from enabling the daemon.

You can enable the Daemon at build time with the **daemon** flag:

```
$ gradle <task> --daemon
```

To enable the Daemon by default in older Gradle versions, add the following setting to the **gradle.properties** file in the project root or your Gradle home:

gradle.properties

```
org.gradle.daemon=true
```

On developer machines, you should see a significant performance improvement. On CI machines, long-lived agents benefit from the Daemon. But short-lived machines don't benefit much. Daemons automatically shut down on memory pressure in Gradle 3.0 and above, so it's always safe to leave the Daemon enabled.

Enable the configuration cache

IMPORTANT

This feature has the following limitations:

- The configuration cache does not support all **core Gradle plugins** and **features**. Full support is a work in progress.
- Your build and the plugins you depend on might require changes to fulfill the **requirements**.
- IDE imports and syncs do not use the configuration cache.

You can cache the result of the configuration phase by enabling the configuration cache. When build configuration inputs remain the same across builds, the configuration cache allows Gradle to skip the configuration phase entirely.

Build configuration inputs include:

- Init scripts
- Settings scripts
- Build scripts
- System properties used during the configuration phase
- Gradle properties used during the configuration phase
- Environment variables used during the configuration phase
- Configuration files accessed using value suppliers such as providers
- `buildSrc` inputs, including build configuration inputs and source files

By default, Gradle does not use the configuration cache. To enable the configuration cache at build time, use the `configuration-cache` flag:

```
$ gradle <task> --configuration-cache
```

To enable the configuration cache by default, add the following setting to the `gradle.properties` file in the project root or your Gradle home:

gradle.properties

```
org.gradle.configuration-cache=true
```

For more information about the configuration cache, check out the [configuration cache documentation](#).

Additional configuration cache benefits

The configuration cache enables additional benefits as well. When enabled, Gradle:

- Executes all tasks in parallel, even those in the same subproject.
- Caches dependency resolution results.

Enable incremental build for custom tasks

Incremental build is a Gradle optimization that skips running tasks that have previously executed with the same inputs. If a task's inputs and its outputs have not changed since the last execution, Gradle skips that task.

Most built-in tasks provided by Gradle work with incremental build. To make a custom task compatible with incremental build, specify the inputs and outputs:

build.gradle.kts

```
tasks.register("processTemplatesAdHoc") {
    inputs.property("engine", TemplateEngineType.FREEMARKER)
    inputs.files(fileTree("src/templates"))
        .withPropertyName("sourceFiles")
        .withPathSensitivity(PathSensitivity.RELATIVE)
    inputs.property("templateData.name", "docs")
    inputs.property("templateData.variables", mapOf("year" to "2013"))
    outputs.dir(layout.buildDirectory.dir("genOutput2"))
        .withPropertyName("outputDir")

    doLast {
        // Process the templates here
    }
}
```

build.gradle

```
tasks.register('processTemplatesAdHoc') {
    inputs.property('engine', TemplateEngineType.FREEMARKER)
    inputs.files(fileTree('src/templates'))
        .withPropertyName('sourceFiles')
        .withPathSensitivity(PathSensitivity.RELATIVE)
    inputs.property('templateData.name', 'docs')
    inputs.property('templateData.variables', [year: '2013'])
    outputs.dir(layout.buildDirectory.dir('genOutput2'))
        .withPropertyName('outputDir')

    doLast {
        // Process the templates here
    }
}
```

For more information about incremental builds, check out the [incremental build documentation](#).

Visualize incremental builds with build scan timelines

Look at the build scan timeline view to identify tasks that could benefit from incremental builds. This can also help you understand why tasks execute when you expect Gradle to skip them.

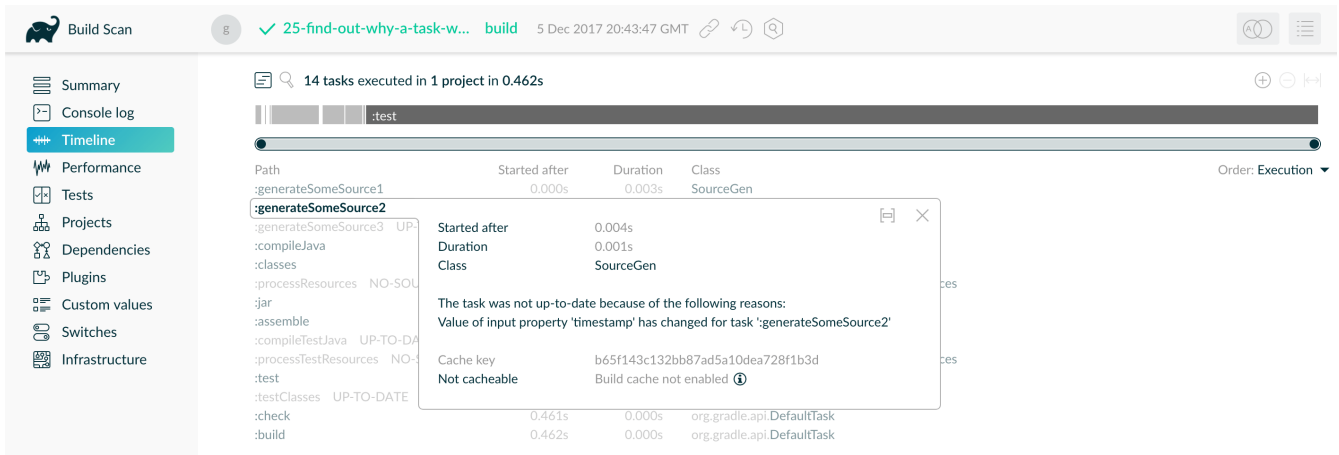


Figure 3. The timeline view can help with incremental build inspection

As you can see in the build scan above, the task was not up-to-date because one of its inputs ("timestamp") changed, forcing the task to re-run.

Sort tasks by duration to find the slowest tasks in your project.

Enable the build cache

The build cache is a Gradle optimization that stores task outputs for specific input. When you later run that same task with the same input, Gradle retrieves the output from the build cache instead of running the task again. By default, Gradle does not use the build cache. To enable the build cache at build time, use the **build-cache** flag:

```
$ gradle <task> --build-cache
```

To enable the build cache by default, add the following setting to the **gradle.properties** file in the project root or your Gradle home:

gradle.properties

```
org.gradle.caching=true
```

You can use a local build cache to speed up repeated builds on a single machine. You can also use a shared build cache to speed up repeated builds across multiple machines. Develcity [provides one](#). Shared build caches can decrease build times for both CI and developer builds.

For more information about the build cache, check out the [build cache documentation](#).

Visualize the build cache with build scans

Build scans can help you investigate build cache effectiveness. In the performance screen, the "Build cache" tab shows you statistics about:

- how many tasks interacted with a cache
- which cache was used
- transfer and pack/unpack rates for these cache entries

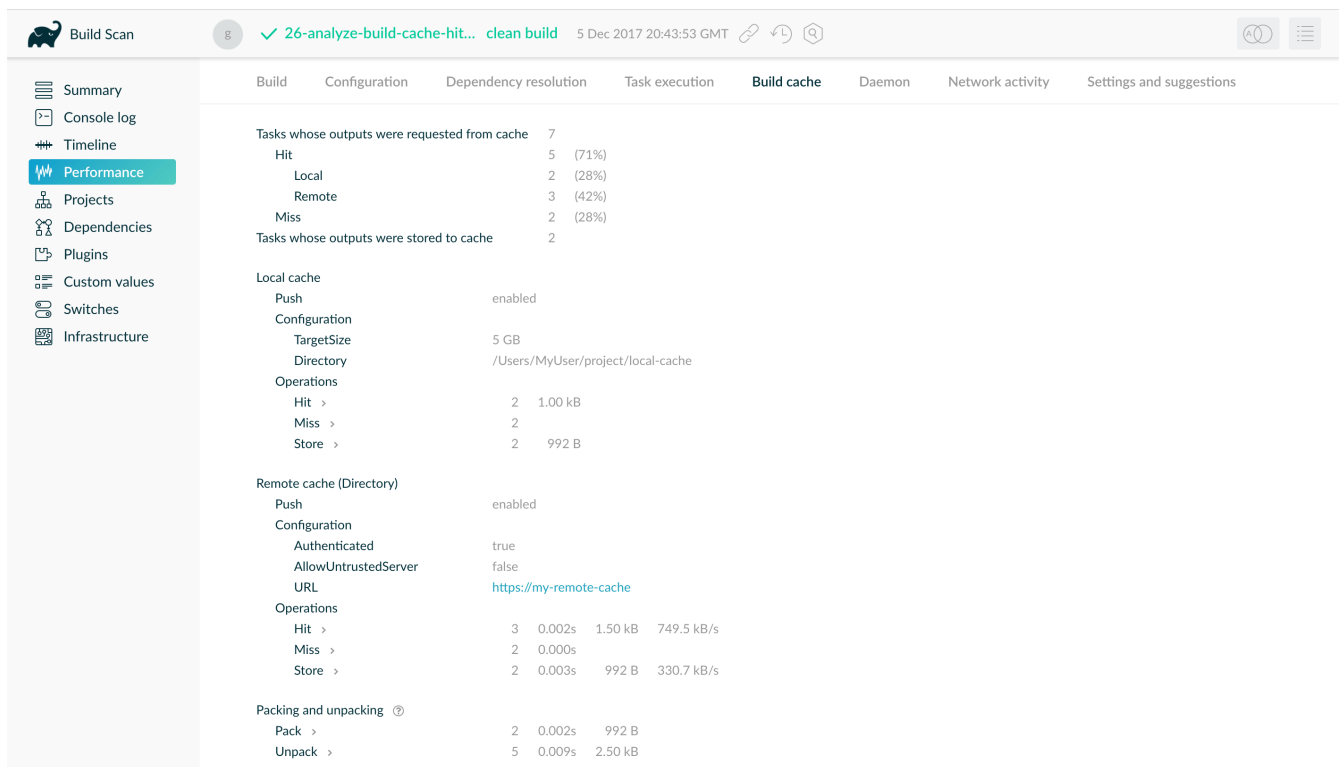


Figure 4. Inspecting the performance of the build cache for a build

The "Task execution" tab shows details about task cacheability. Click on a category to see a timeline screen that highlights tasks of that category.

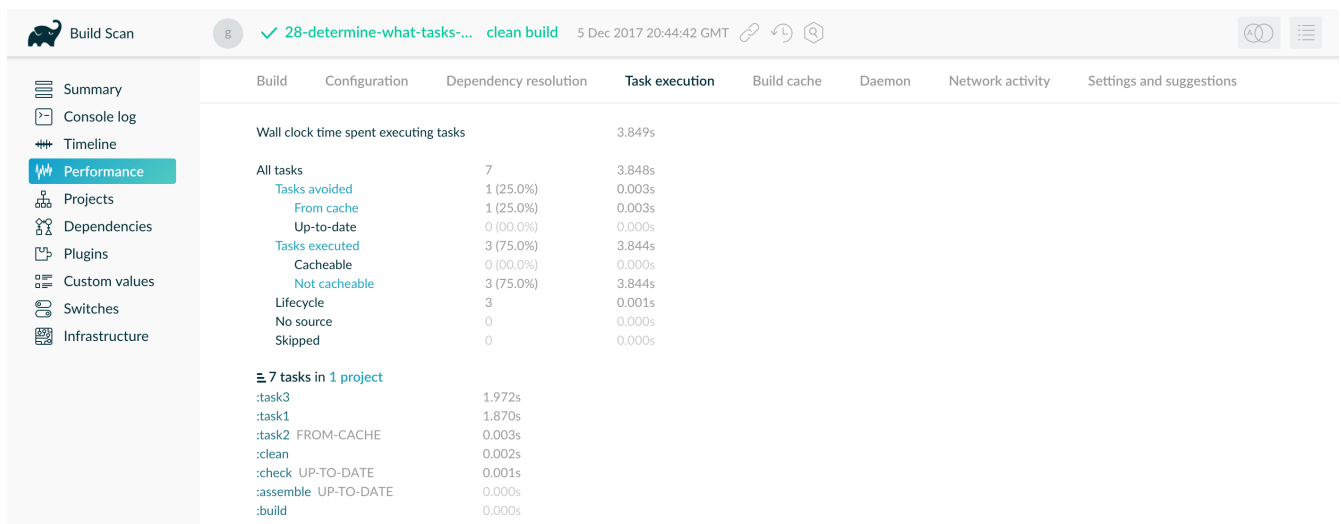


Figure 5. A task oriented view of performance

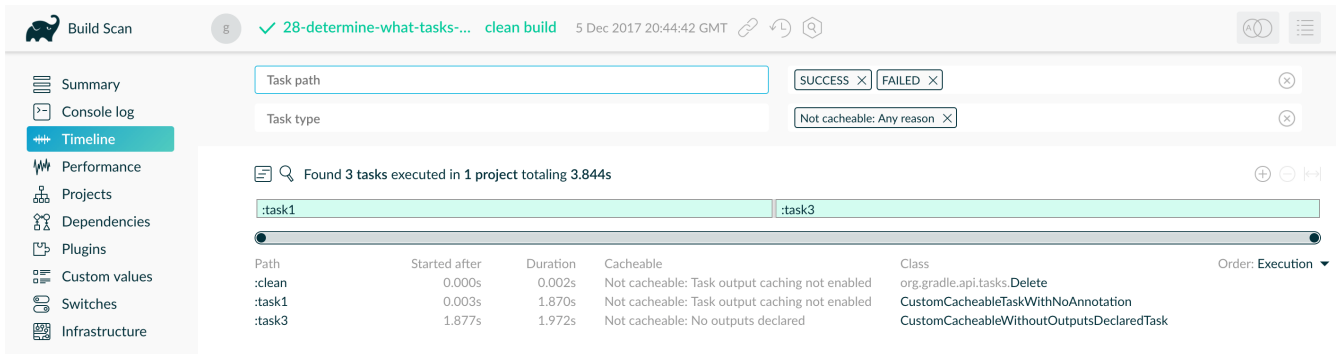


Figure 6. Timeline screen with 'not cacheable' tasks only

Sort by task duration on the timeline screen to highlight tasks with great time saving potential. The build scan above shows that `:task1` and `:task3` could be improved and made cacheable and shows why Gradle didn't cache them.

Create builds for specific developer workflows

The fastest task is one that doesn't execute. If you can find ways to skip tasks you don't need to run, you'll end up with a faster build overall.

If your build includes multiple subprojects, create tasks to build those subprojects independently. This helps you get the most out of caching, since a change to one subproject won't force a rebuild for unrelated subprojects. And this helps reduce build times for teams that work on unrelated subprojects: there's no need for front-end developers to build the back-end subprojects every time they change the front-end. Documentation writers don't need to build front-end or back-end code even if the documentation lives in the same project as that code.

Instead, create tasks that match the needs of developers. You'll still have a single task graph for the whole project. Each group of users suggests a restricted view of the task graph: turn that view into a Gradle workflow that excludes unnecessary tasks.

Gradle provides several features to create these workflows:

- Assign tasks to appropriate [groups](#)
- Create *aggregate tasks*: tasks with no action that only depend on other tasks, such as `assemble`
- Defer configuration via `gradle.taskGraph.whenReady()` and others, so you can perform verification only when it's necessary

Increase the heap size

By default, Gradle reserves 512MB of heap space for your build. This is plenty for most projects. However, some very large builds might need more memory to hold Gradle's model and caches. If this is the case for you, you can specify a larger memory requirement. Specify the following property in the `gradle.properties` file in your project root or your Gradle home:

gradle.properties

```
org.gradle.jvmargs=-Xmx2048M
```

To learn more, check out the [JVM memory configuration documentation](#).

Optimize Configuration

As described in [the build lifecycle chapter](#), a Gradle build goes through 3 phases: initialization, configuration, and execution. Configuration code always executes regardless of the tasks that run. As a result, any expensive work performed during configuration slows down every invocation. Even simple commands like `gradle help` and `gradle tasks`.

The next few subsections introduce techniques that can reduce time spent in the configuration phase.

NOTE

You can also [enable the configuration cache](#) to reduce the impact of a slow configuration phase. But even machines that use the cache still occasionally execute your configuration phase. As a result, you should make the configuration phase as fast as possible with these techniques.

Avoid expensive or blocking work

You should avoid time-intensive work in the configuration phase. But sometimes it can sneak into your build in non-obvious places. It's usually clear when you're encrypting data or calling remote services during configuration if that code is in a build file. But logic like this is more often found in plugins and occasionally custom task classes. Any expensive work in a plugin's `apply()` method or a task's constructor is a red flag.

Only apply plugins where they're needed

Every plugin and script that you apply to a project adds to the overall configuration time. Some plugins have a greater impact than others. That doesn't mean you should avoid using plugins, but you should take care to only apply them where they're needed. For example, it's easy to apply plugins to all subprojects via `allprojects {}` or `subprojects {}` even if not every project needs them.

In the above build scan example, you can see that the root build script applies the `script-a.gradle` script to 3 subprojects inside the build:

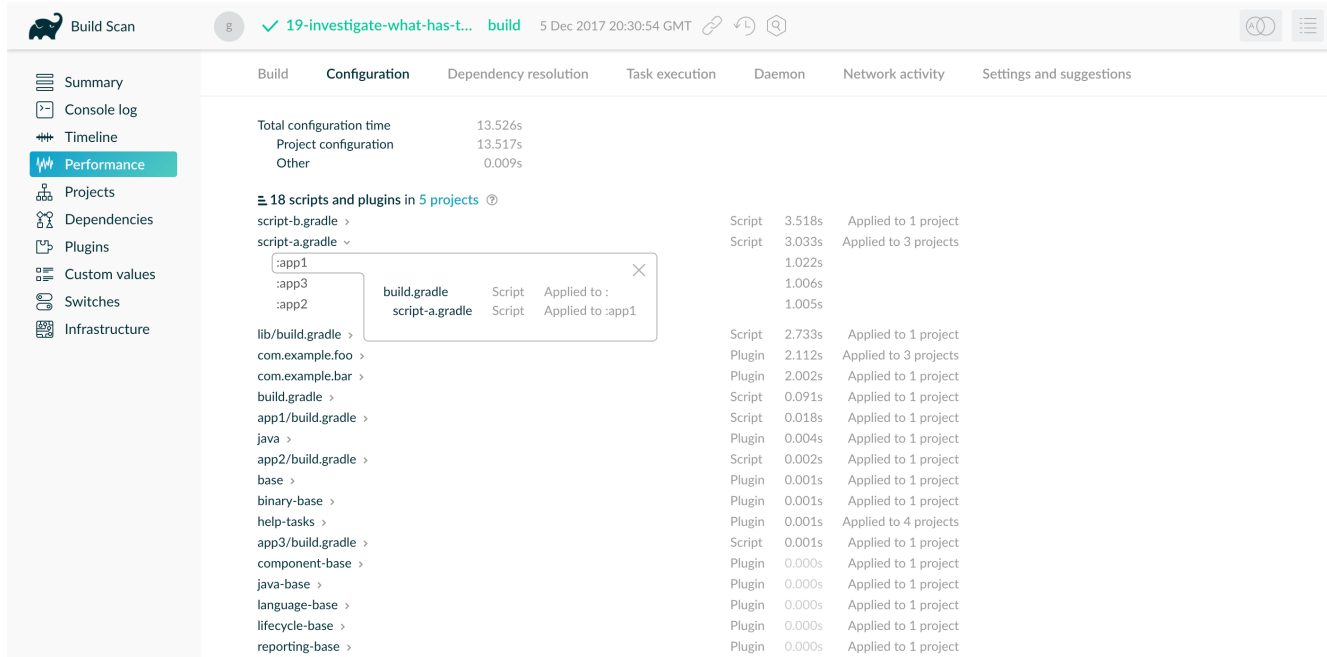


Figure 7. Showing the application of script-a.gradle to the build

This script takes 1 second to run. Since it applies to 3 subprojects, this script cumulatively delays the configuration phase by 3 seconds. In this situation, there are several ways to reduce the delay:

- If only one subproject uses the script, you could remove the script application from the other subprojects. This reduces the configuration delay by two seconds in each Gradle invocation.
- If multiple subprojects, but not all, use the script, you could refactor the script and all surrounding logic into a custom plugin located in `buildSrc`. Apply the custom plugin to only the relevant subprojects, reducing configuration delay and avoiding code duplication.

Statically compile tasks and plugins

Plugin and task authors often write Groovy for its concise syntax, API extensions to the JDK, and functional methods using closures. But Groovy syntax comes with the cost of dynamic interpretation. As a result, method calls in Groovy take more time and use more CPU than method calls in Java or Kotlin.

You can reduce this cost with static Groovy compilation: add the `@CompileStatic` annotation to your Groovy classes when you don't explicitly require dynamic features. If you need dynamic Groovy in a method, add the `@CompileDynamic` annotation to that method.

Alternatively, you can write plugins and tasks in a statically compiled language such as Java or Kotlin.

Warning: Gradle's Groovy DSL relies heavily on Groovy's dynamic features. To use static compilation in your plugins, switch to Java-like syntax.

The following example defines a task that copies files without dynamic features:

```
src/main/groovy/MyPlugin.groovy
```

```
project.tasks.register('copyFiles', Copy) { Task t ->
```

```
t.into(project.layout.buildDirectory.dir('output'))
t.from(project.configurations.getByName('compile'))
}
```

This example uses the `register()` and `getByName()` methods available on all Gradle "domain object containers". Domain object containers include tasks, configurations, dependencies, extensions, and more. Some collections, such as `TaskContainer`, have dedicated types with extra methods like `create`, which accepts a task type.

When you use static compilation, an IDE can:

- quickly show errors related to unrecognised types, properties, and methods
- auto-complete method names

Optimize Dependency resolution

Dependency resolution simplifies integrating third-party libraries and other dependencies into your projects. Gradle contacts remote servers to discover and download dependencies. You can optimize the way you reference dependencies to cut down on these remote server calls.

Avoid unnecessary and unused dependencies

Managing third-party libraries and their transitive dependencies adds a significant cost to project maintenance and build times.

Watch out for unused dependencies: when a third-party library stops being used by isn't removed from the dependency list. This happens frequently during refactors. You can use the [Gradle Lint plugin](#) to identify unused dependencies.

If you only use a small number of methods or classes in a third-party library, consider:

- implementing the required code yourself in your project
- copying the required code from the library (with attribution!) if it is open source

Optimize repository order

When Gradle resolves dependencies, it searches through each repository in the declared order. To reduce the time spent searching for dependencies, declare the repository hosting the largest number of your dependencies first. This minimizes the number of network requests required to resolve all dependencies.

Minimize repository count

Limit the number of declared repositories to the minimum possible for your build to work.

If you're using a custom repository server, create a virtual repository that aggregates several repositories together. Then, add only that repository to your build file.

Minimize dynamic and snapshot versions

Dynamic versions (e.g. "2.+"), and changing versions (snapshots) force Gradle to contact remote repositories to find new releases. By default, Gradle only checks once every 24 hours. But you can change this programmatically with the following settings:

- `cacheDynamicVersionsFor`
- `cacheChangingModulesFor`

If a build file or initialization script lowers these values, Gradle queries repositories more often. When you don't need the absolute latest release of a dependency every time you build, consider removing the custom values for these settings.

Find dynamic and changing versions with build scans

You can find all dependencies with dynamic versions via build scans:

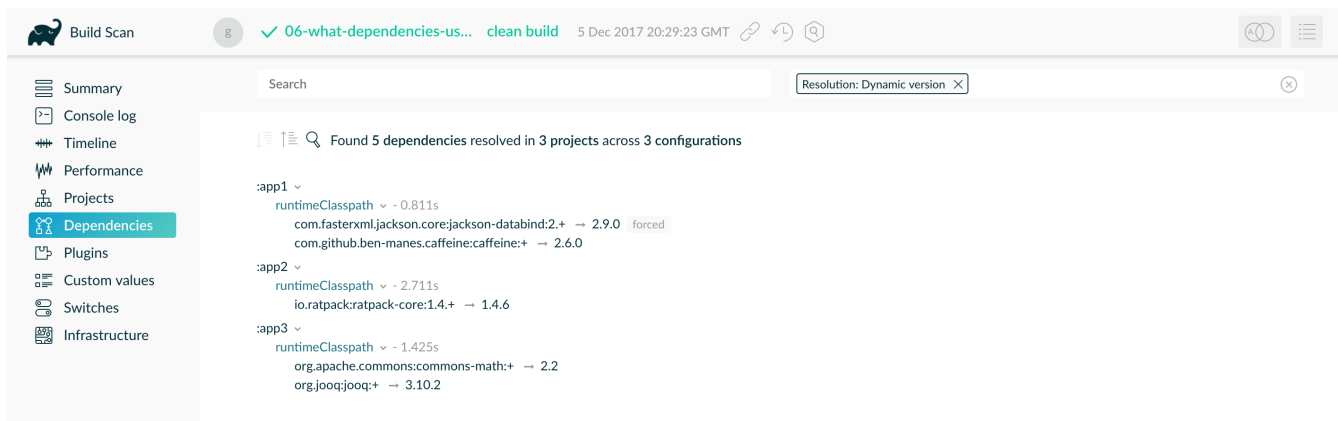


Figure 8. Find dependencies with dynamic versions

You may be able to use fixed versions like "1.2" and "3.0.3.GA" that allow Gradle to cache versions. If you must use dynamic and changing versions, tune the cache settings to best meet your needs.

Avoid dependency resolution during configuration

Dependency resolution is an expensive process, both in terms of I/O and computation. Gradle reduces the required network traffic through caching. But there is still a cost. Gradle runs the configuration phase on every build. If you trigger dependency resolution during the configuration phase, every build pays that cost.

Switch to declarative syntax

If you evaluate a configuration file, your project pays the cost of dependency resolution during configuration. Normally tasks evaluate these files, since you don't need the files until you're ready to do something with them in a task action. Imagine you're doing some debugging and want to display the files that make up a configuration. To implement this, you might inject a print statement:

build.gradle.kts

```
tasks.register<Copy>("copyFiles") {
    println(">> Compilation deps:
${configurations.compileClasspath.get().files.map { it.name }}")
    into(layout.buildDirectory.dir("output"))
    from(configurations.compileClasspath)
}
```

build.gradle

```
tasks.register('copyFiles', Copy) {
    println ">> Compilation deps: ${configurations.compileClasspath.files
.name}"
    into(layout.buildDirectory.dir('output'))
    from(configurations.compileClasspath)
}
```

The `files` property forces Gradle to resolve the dependencies. In this example, that happens during the configuration phase. Because the configuration phase runs on every build, all builds now pay the performance cost of dependency resolution. You can avoid this cost with a `doFirst()` action:

build.gradle.kts

```
tasks.register<Copy>("copyFiles") {
    into(layout.buildDirectory.dir("output"))
    // Store the configuration into a variable because referencing the
    // project from the task action
    // is not compatible with the configuration cache.
    val compileClasspath: FileCollection =
    configurations.compileClasspath.get()
    from(compileClasspath)
    doFirst {
        println(">> Compilation deps: ${compileClasspath.files.map { it.name
    }}")
    }
}
```

build.gradle

```
tasks.register('copyFiles', Copy) {
```

```

into(layout.buildDirectory.dir('output'))
// Store the configuration into a variable because referencing the
project from the task action
// is not compatible with the configuration cache.
FileCollection compileClasspath = configurations.compileClasspath
from(compileClasspath)
doFirst {
    println ">> Compilation deps: ${compileClasspath.files.name}"
}
}

```

Note that the `from()` declaration doesn't resolve the dependencies because you're using the [dependency configuration](#) itself as an argument, not the files. The `Copy` task resolves the configuration itself during task execution.

Visualize dependency resolution with build scans

The "Dependency resolution" tab on the performance page of a build scan shows dependency resolution time during the configuration and execution phases:

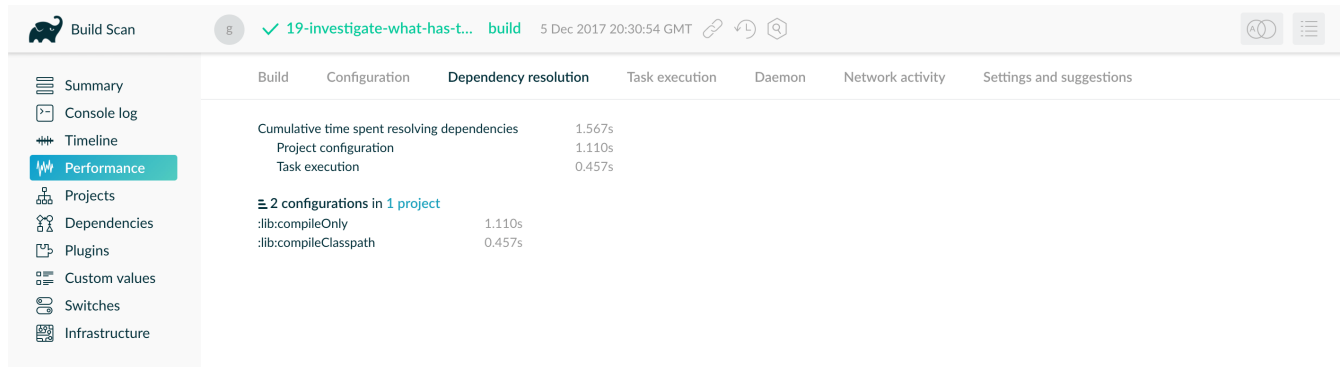


Figure 9. Dependency resolution at configuration time

Build scans provide another means of identifying this issue. Your build should spend 0 seconds resolving dependencies during "project configuration". This example shows the build resolves dependencies too early in the lifecycle. You can also find a "Settings and suggestions" tab on the "Performance" page. This shows dependencies resolved during the configuration phase.

Remove or improve custom dependency resolution logic

Gradle allows users to model dependency resolution in the way that best suits them. Simple customizations, such as forcing specific versions of a dependency or substituting one dependency for another, don't have a big impact on dependency resolution times. More complex customizations, such as custom logic that downloads and parses POMs, can slow down dependency resolution significantly.

Use build scans or profile reports to check that custom dependency resolution logic doesn't adversely affect dependency resolution times. This could be custom logic you have written yourself, or it could be part of a plugin.

Remove slow or unexpected dependency downloads

Slow dependency downloads can impact your overall build performance. Several things could cause this, including a slow internet connection or an overloaded repository server. On the "Performance" page of a build scan, you'll find a "Network Activity" tab. This tab lists information including:

- the time spent downloading dependencies
- the transfer rate of dependency downloads
- a list of downloads sorted by download time

In the following example, two slow dependency downloads took 20 and 40 seconds and slowed down the overall performance of a build:

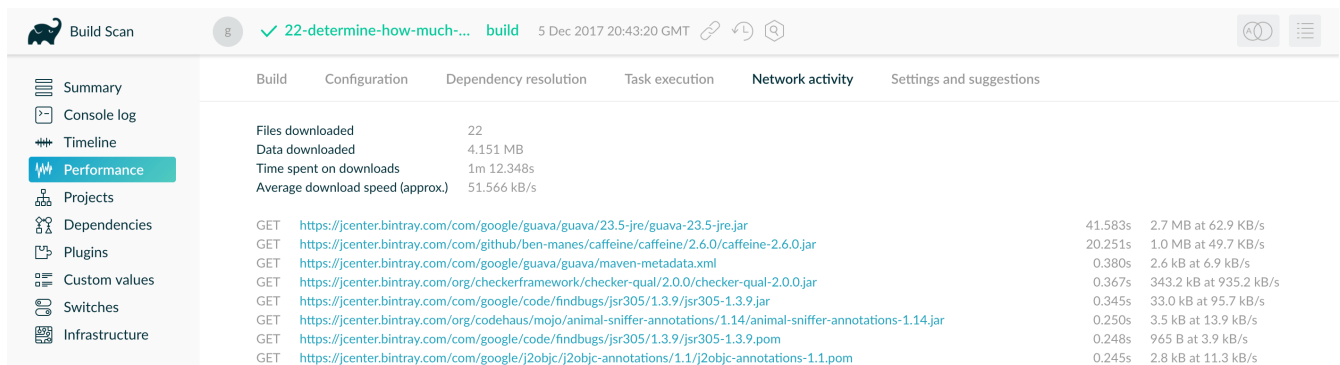


Figure 10. Identify slow dependency downloads

Check the download list for unexpected dependency downloads. For example, you might see a download caused by a dependency using a dynamic version.

Eliminate these slow or unexpected downloads by switching to a different repository or dependency.

Optimize Java projects

The following sections apply only to projects that use the `java` plugin or another JVM language.

Optimize tests

Projects often spend much of their build time testing. These could be a mixture of unit and integration tests. Integration tests usually take longer. Build scans can help you identify the slowest tests. You can then focus on speeding up those tests.

ways.

Fork tests into multiple processes

By default, Gradle runs all tests in a single forked VM. If there are a lot of tests, or some tests that consume lots of memory, your tests may take longer than you expect to run. You can increase the heap size, but garbage collection may slow down your tests.

Alternatively, you can fork a new test VM after a certain number of tests have run with the `forkEvery` setting:

build.gradle.kts

```
tasks.withType<Test>().configureEach {  
    forkEvery = 100  
}
```

build.gradle

```
tasks.withType(Test).configureEach {  
    forkEvery = 100  
}
```

WARNING

Forking a VM is an expensive operation. Setting too small a value here slows down testing.

Disable reports

Gradle automatically creates test reports regardless of whether you want to look at them. That report generation slows down the overall build. You may not need reports if:

- you only care if the tests succeeded (rather than why)
- you use build scans, which provide more information than a local report

To disable test reports, set `reports.html.required` and `reports.junitXml.required` to `false` in the `Test` task:

build.gradle.kts

```
tasks.withType<Test>().configureEach {  
    reports.html.required = false  
    reports.junitXml.required = false  
}
```

```
}
```

build.gradle

```
tasks.withType(Test).configureEach {  
    reports.html.required = false  
    reports.junitXml.required = false  
}
```

Conditionally enable reports

You might want to conditionally enable reports so you don't have to edit the build file to see them. To enable the reports based on a project property, check for the presence of a property before disabling reports:

build.gradle.kts

```
tasks.withType<Test>().configureEach {  
    if (!project.hasProperty("createReports")) {  
        reports.html.required = false  
        reports.junitXml.required = false  
    }  
}
```

build.gradle

```
tasks.withType(Test).configureEach {  
    if (!project.hasProperty("createReports")) {  
        reports.html.required = false  
        reports.junitXml.required = false  
    }  
}
```

Then, pass the property with **-PcreateReports** on the command line to generate the reports.

```
$ gradle <task> -PcreateReports
```

Or configure the property in the **gradle.properties** file in the project root or your Gradle home:

gradle.properties

```
createReports=true
```

Optimize the compiler

The Java compiler is fast. But if you're compiling hundreds of Java classes, even a short compilation time adds up. Gradle offers a several optimizations for Java compilation:

- Run the compiler as a separate process
- Switch internal-only dependencies to implementation visibility

Run the compiler as a separate process

You can run the compiler as a separate process with the following configuration for any `JavaCompile` task:

build.gradle.kts

```
<task>.options.isFork = true
```

build.gradle

```
<task>.options.fork = true
```

To apply the configuration to *all* Java compilation tasks, you can `configureEach` java compilation task:

build.gradle.kts

```
tasks.withType<JavaCompile>().configureEach {  
    options.isFork = true  
}
```

build.gradle

```
tasks.withType(JavaCompile).configureEach {  
    options.fork = true  
}
```

Gradle reuses this process within the duration the build, so the forking overhead is minimal. By forking memory-intensive compilation into a separate process, we minimize garbage collection in the main Gradle process. Less garbage collection means that Gradle's infrastructure can run faster, especially when you also use [parallel builds](#).

Forking compilation rarely impacts the performance of small projects. But you should consider it if a single task compiles more than a thousand source files together.

Switch internal-only dependencies to implementation visibility

NOTE

Only libraries can define **api** dependencies. Use the **java-library** plugin to define API dependencies in your libraries. Projects that use the **java** plugin cannot declare **api** dependencies.

Before Gradle 3.4, projects declared dependencies using the **compile** configuration. This exposed all of those dependencies to downstream projects. In Gradle 3.4 and above, you can separate downstream-facing **api** dependencies from internal-only **implementation** details. Implementation dependencies don't leak into the compile classpath of downstream projects. When implementation details change, Gradle only recompiles **api** dependencies.

build.gradle.kts

```
dependencies {  
    api(project("my-utils"))  
    implementation("com.google.guava:guava:21.0")  
}
```

build.gradle

```
dependencies {  
    api project('my-utils')  
    implementation 'com.google.guava:guava:21.0'  
}
```

This can significantly reduce the "ripple" of recompilations caused by a single change in large multi-project builds.

Improve the performance of older Gradle releases

Some projects cannot easily upgrade to a current Gradle version. While you should always upgrade Gradle to a recent version when possible, we recognize that it isn't always feasible for certain niche situations. In those select cases, check out these recommendations to optimize older versions of Gradle.

Enable the Daemon

Gradle 3.0 and above enable the Daemon by default. If you are using an older version, you should

[update to the latest version of Gradle](#). If you cannot update your Gradle version, you can [enable the Daemon manually](#).

Use incremental compilation

Gradle can analyze dependencies down to the individual class level to recompile only the classes affected by a change. Gradle 4.10 and above enable incremental compilation by default. To enable incremental compilation by default in older Gradle versions, add the following setting to your `build.gradle` file:

build.gradle.kts

```
tasks.withType<JavaCompile>().configureEach {  
    options.isIncremental = true  
}
```

build.gradle

```
tasks.withType(JavaCompile).configureEach {  
    options.incremental = true  
}
```

Use compile avoidance

Often, updates only change internal implementation details of your code, like the body of a method. These updates are known as *ABI-compatible* changes: they have no impact on the binary interface of your project. In Gradle 3.4 and above, ABI-compatible changes no longer trigger recompiles of downstream projects. This especially improves build times in large multi-project builds with deep dependency chains.

Upgrade to a Gradle version above 3.4 to benefit from compile avoidance.

NOTE

If you use annotation processors, you need to explicitly declare them in order for compilation avoidance to work. To learn more, check out the [compile avoidance documentation](#).

Optimize Android projects

Everything on this page applies to Android builds, since Android builds use Gradle. Yet Android introduces unique opportunities for optimization. For more information, check out the [Android team performance guide](#). You can also [watch the accompanying talk](#) from Google IO 2017.

Configuration cache

Introduction

The configuration cache is a feature that significantly improves build performance by caching the result of the [configuration phase](#) and reusing this for subsequent builds. Using the configuration cache, Gradle can skip the configuration phase entirely when nothing that affects the build configuration, such as build scripts, has changed. Gradle also applies performance improvements to task execution as well.

The configuration cache is conceptually similar to the [build cache](#), but caches different information. The build cache takes care of caching the outputs and intermediate files of the build, such as task outputs or artifact transform outputs. The configuration cache takes care of caching the build configuration for a particular set of tasks. In other words, the configuration cache saves the output of the configuration phase, and the build cache saves the outputs of the execution phase.

IMPORTANT

This feature is currently not enabled by default. This feature has the following limitations:

- The configuration cache does not support all [core Gradle plugins](#) and [features](#). Full support is a work in progress.
- Your build and the plugins you depend on might require changes to fulfil the [requirements](#).
- IDE imports and syncs do not yet use the configuration cache.

How does it work?

When the configuration cache is enabled and you run Gradle for a particular set of tasks, for example by running `gradlew check`, Gradle checks whether a configuration cache entry is available for the requested set of tasks. If available, Gradle uses this entry instead of running the configuration phase. The cache entry contains information about the set of tasks to run, along with their configuration and dependency information.

The first time you run a particular set of tasks, there will be no entry in the configuration cache for these tasks and so Gradle will run the configuration phase as normal:

1. Run init scripts.
2. Run the settings script for the build, applying any requested settings plugins.
3. Configure and build the `buildSrc` project, if present.
4. Run the build scripts for the build, applying any requested project plugins.
5. Calculate the task graph for the requested tasks, running any deferred configuration actions.

Following the configuration phase, Gradle writes a snapshot of the task graph to a new configuration cache entry, for later Gradle invocations. Gradle then loads the task graph from the configuration cache, so that it can apply optimizations to the tasks, and then runs the execution phase as normal. Configuration time will still be spent the first time you run a particular set of tasks. However, you should see build performance improvement immediately because [tasks will run in parallel](#).

When you subsequently run Gradle with this same set of tasks, for example by running `gradlew check` again, Gradle will load the tasks and their configuration directly from the configuration cache and skip the configuration phase entirely. Before using a configuration cache entry, Gradle checks that none of the "build configuration inputs", such as build scripts, for the entry have changed. If a build configuration input has changed, Gradle will not use the entry and will run the configuration phase again as above, saving the result for later reuse.

Build configuration inputs include:

- Init scripts
- Settings scripts
- Build scripts
- System properties used during the configuration phase
- Gradle properties used during the configuration phase
- Environment variables used during the configuration phase
- Configuration files accessed using value suppliers such as providers
- `buildSrc` and plugin included build inputs, including build configuration inputs and source files.

Gradle uses its own optimized serialization mechanism and format to store the configuration cache entries. It automatically serializes the state of arbitrary object graphs. If your tasks hold references to objects with simple state or of supported types you don't have anything to do to support the serialization.

As a fallback and to provide some aid in migrating existing tasks, some semantics of [Java Serialization](#) are supported. But it is not recommended relying on it, mostly for performance reasons.

Performance improvements

Apart from skipping the configuration phase, the configuration cache provides some additional performance improvements:

- All tasks run in parallel by default, subject to dependency constraints.
- Dependency resolution is cached.
- Configuration state and dependency resolution state is discarded from heap after writing the task graph. This reduces the peak heap usage required for a given set of tasks.

Configuration caching in action

[running help] | *configuration-cache/running-help.gif*

Using the configuration cache

It is recommended to get started with the simplest task invocation possible. Running `help` with the configuration cache enabled is a good first step:

```
❯ gradle --configuration-cache help
Calculating task graph as no cached configuration is available for tasks: help
...
BUILD SUCCESSFUL in 4s
1 actionable task: 1 executed
Configuration cache entry stored.
```

Running this for the first time, the configuration phase executes, calculating the task graph.

Then, run the same command again. This reuses the cached configuration:

```
❯ gradle --configuration-cache help
Reusing configuration cache.
...
BUILD SUCCESSFUL in 500ms
1 actionable task: 1 executed
Configuration cache entry reused.
```

If it succeeds on your build, congratulations, you can now try with more useful tasks. You should target your development loop. A good example is running tests after making incremental changes.

If any problem is found caching or reusing the configuration, an HTML report is generated to help you diagnose and fix the issues. The report also shows detected build configuration inputs like system properties, environment variables and value suppliers read during the configuration phase. See the [Troubleshooting](#) section below for more information.

Keep reading to learn how to tweak the configuration cache, manually invalidate the state if something goes wrong and use the configuration cache from an IDE.

Enabling the configuration cache

By default, Gradle does not use the configuration cache. To enable the cache at build time, use the `configuration-cache` flag:

```
❯ gradle --configuration-cache
```

You can also enable the cache persistently in a `gradle.properties` file using the `org.gradle.configuration-cache` property:

```
org.gradle.configuration-cache=true
```

If enabled in a `gradle.properties` file, you can override that setting and disable the cache at build time with the `no-configuration-cache` flag:

```
❯ gradle --no-configuration-cache
```

Ignoring problems

By default, Gradle will fail the build if any configuration cache problems are encountered. When gradually improving your plugin or build logic to support the configuration cache it can be useful to temporarily turn problems into warnings, with no guarantee that the build will work.

This can be done from the command line:

```
❯ gradle --configuration-cache-problems=warn
```

or in a `gradle.properties` file:

```
org.gradle.configuration-cache.problems=warn
```

Allowing a maximum number of problems

When configuration cache problems are turned into warnings, Gradle will fail the build if 512 problems are found by default.

This can be adjusted by specifying an allowed maximum number of problems on the command line:

```
❯ gradle -Dorg.gradle.configuration-cache.max-problems=5
```

or in a `gradle.properties` file:

```
org.gradle.configuration-cache.max-problems=5
```

Enabling parallel configuration caching

Configuration cache storing and loading are done sequentially by default. Parallel storing and loading provide better performance, however not all builds are compatible with it.

To enable parallel configuration caching on the command line:

```
❯ gradle -Dorg.gradle.configuration-cache.parallel=true
```

or in a `gradle.properties` file:

```
org.gradle.configuration-cache.parallel=true
```

The parallel configuration caching feature is *incubating*, as not all builds are guaranteed to work correctly. Common symptoms are `ConcurrentModificationException` exceptions during the [configuration phase](#). The feature should work well for [decoupled](#) multi-project builds.

Invalidating the cache

The configuration cache is automatically invalidated when inputs to the configuration phase change. However, certain inputs are not tracked yet, so you may have to manually invalidate the configuration cache when untracked inputs to the configuration phase change. This can happen if you [ignored problems](#). See the [Requirements](#) and [Not yet implemented](#) sections below for more information.

The configuration cache state is stored on disk in a directory named `.gradle/configuration-cache` in the root directory of the Gradle build in use. If you need to invalidate the cache, simply delete that directory:

```
❯ rm -rf .gradle/configuration-cache
```

Configuration cache entries are checked periodically (at most every 24 hours) for whether they are still in use. They are deleted if they haven't been used for 7 days.

Stable configuration cache

Working towards the stabilization of configuration caching we implemented some strictness behind a feature flag when it was considered too disruptive for early adopters.

You can enable that feature flag as follows:

settings.gradle.kts

```
enableFeaturePreview("STABLE_CONFIGURATION_CACHE")
```

settings.gradle

```
enableFeaturePreview "STABLE_CONFIGURATION_CACHE"
```

The `STABLE_CONFIGURATION_CACHE` feature flag enables the following:

Undeclared shared build service usage

When enabled, tasks using a [shared build service](#) without declaring the requirement via the `Task.usesService` method will emit a deprecation warning.

In addition, when the configuration cache is not enabled but the feature flag is present, deprecations for the following [configuration cache requirements](#) are also enabled:

- [Registering build listeners](#)
- [Using task extensions and conventions at execution time](#)

It is recommended to enable it as soon as possible in order to be ready for when we remove the flag and make the linked features the default.

IDE support

If you enable and configure the configuration cache from your `gradle.properties` file, then the configuration cache will be enabled when your IDE delegates to Gradle. There's nothing more to do.

`gradle.properties` is usually checked in to source control. If you don't want to enable the configuration cache for your whole team yet you can also enable the configuration cache from your IDE only as explained below.

Note that syncing a build from an IDE doesn't benefit from the configuration cache, only running tasks does.

IntelliJ based IDEs

In IntelliJ IDEA or Android Studio this can be done in two ways, either globally or per run configuration.

To enable it for the whole build, go to `Run > Edit configurations...`. This will open the IntelliJ IDEA or Android Studio dialog to configure Run/Debug configurations. Select `Templates > Gradle` and add the necessary system properties to the `VM options` field.

For example to enable the configuration cache, turning problems into warnings, add the following:

```
-Dorg.gradle.configuration-cache=true -Dorg.gradle.configuration-cache.problems=warn
```

You can also choose to only enable it for a given run configuration. In this case, leave the `Templates > Gradle` configuration untouched and edit each run configuration as you see fit.

Combining these two ways you can enable globally and disable for certain run configurations, or the opposite.

TIP

You can use the [gradle-idea-ext-plugin](#) to configure IntelliJ run configurations from your build. This is a good way to enable the configuration cache only for the IDE.

Eclipse IDEs

In Eclipse IDEs you can enable and configure the configuration cache through Buildship in two ways, either globally or per run configuration.

To enable it globally, go to `Preferences > Gradle`. You can use the properties described above as system properties. For example to enable the configuration cache, turning problems into warnings, add the following JVM arguments:

- `-Dorg.gradle.configuration-cache=true`
- `-Dorg.gradle.configuration-cache.problems=warn`

To enable it for a given run configuration, go to **Run configurations...**, find the one you want to change, go to **Project Settings**, tick the **Override project settings** checkbox and add the same system properties as a **JVM argument**.

Combining these two ways you can enable globally and disable for certain run configurations, or the opposite.

Supported plugins

The configuration cache is brand new and introduces new requirements for plugin implementations. As a result, both core Gradle plugins, and community plugins need to be adjusted. This section provides information about the current support in **core Gradle plugins** and **community plugins**.

Core Gradle plugins

Not all **core Gradle plugins** support configuration caching yet.

JVM languages and frameworks	Native languages	Packaging and distribution
<ul style="list-style-type: none">JavaJava LibraryJava PlatformGroovyScalaANTLR	<ul style="list-style-type: none">C++ ApplicationC++ LibraryC++ Unit TestSwift ApplicationSwift LibraryXCTest	<ul style="list-style-type: none">ApplicationWAREARMaven PublishIvy PublishDistributionJava Library Distribution
Code analysis	IDE project files generation	Utility

☐ Checkstyle	☐ Eclipse	☐ Base
☐ CodeNarc	☐ IntelliJ IDEA	☐ Build Init
☐ JaCoCo	☐ Visual Studio	☐ Signing
☐ JaCoCo Report Aggregation	☐ Xcode	☐ Java Plugin Development
☐ PMD		☐ Groovy DSL Plugin Development
☐ Test Report Aggregation		☐ Kotlin DSL Plugin Development
		☐ Project Report Plugin

- ☐ Supported plugin
- ☐ Partially supported plugin
- ☐ Unsupported plugin

Community plugins

Please refer to issue [gradle/gradle#13490](#) to learn about the status of community plugins.

Troubleshooting

The following sections will go through some general guidelines on dealing with problems with the configuration cache. This applies to both your build logic and to your Gradle plugins.

Upon failure to serialize the state required to run the tasks, an HTML report of detected problems is generated. The Gradle failure output includes a clickable link to the report. This report is useful and allows you to drill down into problems, understand what is causing them.

Let's look at a simple example build script that contains a couple problems:

build.gradle.kts

```
tasks.register("someTask") {
    val destination = System.getProperty("someDestination") ①
    inputs.dir("source")
    outputs.dir(destination)
    doLast {
        project.copy { ②
            from("source")
            into(destination)
        }
    }
}
```

```
}
```

build.gradle

```
tasks.register('someTask') {  
    def destination = System.getProperty('someDestination') ①  
    inputs.dir('source')  
    outputs.dir(destination)  
    doLast {  
        project.copy { ②  
            from 'source'  
            into destination  
        }  
    }  
}
```

① A system property read at configuration time

② Using the Project object at execution time

Running that task fails and print the following in the console:

```
❯ gradle --configuration-cache someTask -DsomeDestination=dest
```

```
...
```

```
* What went wrong:
```

```
Configuration cache problems found in this build.
```

```
1 problem was found storing the configuration cache.
```

```
- Build file 'build.gradle': line 6: invocation of 'Task.project' at execution time is unsupported.
```

```
See
```

```
https://docs.gradle.org/0.0.0/userguide/configuration\_cache.html#config\_cache:requirements:use\_project\_during\_execution
```

```
See the complete report at
```

```
file:///home/user/gradle/samples/build/reports/configuration-cache/<hash>/configuration-cache-report.html
```

```
> Invocation of 'Task.project' by task ':someTask' at execution time is unsupported.
```

```
* Try:
```

```
> Run with --stacktrace option to get the stack trace.
```

```
> Run with --info or --debug option to get more log output.
```

```
> Run with --scan to get full insights.
```

```
> Get more help at https://help.gradle.org.
```

```
BUILD FAILED in 0s
```

```
1 actionable task: 1 executed
```

Configuration cache entry discarded with 1 problem.

The configuration cache entry was discarded because of the found problem failing the build.

Details can be found in the linked HTML report:



Learn more about the [Gradle Configuration Cache](#).

Storing the configuration cache for `someTask`

1 build configuration input was found and will cause the cache to be discarded when its value change

1 problem was found

Build configuration inputs 1

Problems grouped by message 1

Problems grouped by location 1

> ✗ invocation of `Task.project` at execution time is unsupported. ⓘ

The report displays the set of problems twice. First grouped by problem message, then grouped by task. The former allows you to quickly see what classes of problems your build is facing. The latter allows you to quickly see which tasks are problematic. In both cases you can expand the tree in order to discover where the culprit is in the object graph.

The report also includes a list of detected build configuration inputs, such as environment variables, system properties and value suppliers that were read at configuration phase:



Learn more about the [Gradle Configuration Cache](#).

Storing the configuration cache for `someTask`

1 build configuration input was found and will cause the cache to be discarded when its value change

1 problem was found

Build configuration inputs 1

Problems grouped by message 1

Problems grouped by location 1

> ⚠ system property `someDestination` ⓘ

TIP

Problems displayed in the report have links to the corresponding [requirement](#) where you can find guidance on how to fix the problem or to the corresponding [not yet implemented](#) feature.

When changing your build or plugin to fix the problems you should consider [testing your build logic with TestKit](#).

At this stage, you can decide to either [turn the problems into warnings](#) and continue exploring how your build reacts to the configuration cache, or fix the problems at hand.

Let's ignore the reported problem, and run the same build again twice to see what happens when

reusing the cached problematic configuration:

```
❏ gradle --configuration-cache --configuration-cache-problems=warn someTask
-DsomeDestination=dest
Calculating task graph as no cached configuration is available for tasks: someTask
> Task :someTask

1 problem was found storing the configuration cache.
- Build file 'build.gradle': line 6: invocation of 'Task.project' at execution time is
unsupported.
  See
https://docs.gradle.org/0.0.0/userguide/configuration\_cache.html#config\_cache:requirements:use\_project\_during\_execution

See the complete report at
file:///home/user/gradle/samples/build/reports/configuration-
cache/<hash>/configuration-cache-report.html

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
Configuration cache entry stored with 1 problem.
❏ gradle --configuration-cache --configuration-cache-problems=warn someTask
-DsomeDestination=dest
Reusing configuration cache.
> Task :someTask

1 problem was found reusing the configuration cache.
- Build file 'build.gradle': line 6: invocation of 'Task.project' at execution time is
unsupported.
  See
https://docs.gradle.org/0.0.0/userguide/configuration\_cache.html#config\_cache:requirements:use\_project\_during\_execution

See the complete report at
file:///home/user/gradle/samples/build/reports/configuration-
cache/<hash>/configuration-cache-report.html

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
Configuration cache entry reused with 1 problem.
```

The two builds succeed reporting the observed problem, storing then reusing the configuration cache.

With the help of the links present in the console problem summary and in the HTML report we can fix our problems. Here's a fixed version of the build script:

build.gradle.kts

```
abstract class MyCopyTask : DefaultTask() { ①

    @get:InputDirectory abstract val source: DirectoryProperty ②

    @get:OutputDirectory abstract val destination: DirectoryProperty ②

    @get:Inject abstract val fs: FileSystemOperations ③

    @TaskAction
    fun action() {
        fs.copy { ③
            from(source)
            into(destination)
        }
    }
}

tasks.register<MyCopyTask>("someTask") {
    val projectDir = layout.projectDirectory
    source = projectDir.dir("source")
    destination = projectDir.dir(System.getProperty("someDestination"))
}
```

build.gradle

```
abstract class MyCopyTask extends DefaultTask { ①

    @InputDirectory abstract DirectoryProperty getSource() ②

    @OutputDirectory abstract DirectoryProperty getDestination() ②

    @Inject abstract FileSystemOperations getFs() ③

    @TaskAction
    void action() {
        fs.copy { ③
            from source
            into destination
        }
    }
}

tasks.register('someTask', MyCopyTask) {
    def projectDir = layout.projectDirectory
    source = projectDir.dir('source')
```

```
destination = projectDir.dir(System.getProperty('someDestination'))
}
```

- ① We turned our ad-hoc task into a proper task class,
- ② with inputs and outputs declaration,
- ③ and injected with the `FileSystemOperations` service, a supported replacement for `project.copy` `{}`.

Running the task twice now succeeds without reporting any problem and reuses the configuration cache on the second run:

```
❯ gradle --configuration-cache someTask -DsomeDestination=dest
Calculating task graph as no cached configuration is available for tasks: someTask
> Task :someTask

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
Configuration cache entry stored.
❯ gradle --configuration-cache someTask -DsomeDestination=dest
Reusing configuration cache.
> Task :someTask

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
Configuration cache entry reused.
```

But, what if we change the value of the system property?

```
❯ gradle --configuration-cache someTask -DsomeDestination=another
Calculating task graph as configuration cache cannot be reused because system property
'someDestination' has changed.
> Task :someTask

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
Configuration cache entry stored.
```

The previous configuration cache entry could not be reused, and the task graph had to be calculated and stored again. This is because we read the system property at configuration time, hence requiring Gradle to run the configuration phase again when the value of that property changes. Fixing that is as simple as obtaining the provider of the system property and wiring it to the task input, without reading it at configuration time.

build.gradle.kts

```
tasks.register<MyCopyTask>("someTask") {  
    val projectDir = layout.projectDirectory  
    source = projectDir.dir("source")  
    destination = projectDir.dir(providers.systemProperty("someDestination"))  
    ①  
}
```

build.gradle

```
tasks.register('someTask', MyCopyTask) {  
    def projectDir = layout.projectDirectory  
    source = projectDir.dir('source')  
    destination = projectDir.dir(providers.systemProperty('someDestination'))  
    ①  
}
```

① We wired the system property provider directly, without reading it at configuration time.

With this simple change in place we can run the task any number of times, change the system property value, and reuse the configuration cache:

```
❏ gradle --configuration-cache someTask -DsomeDestination=dest  
Calculating task graph as no cached configuration is available for tasks: someTask  
> Task :someTask  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed  
Configuration cache entry stored.  
❏ gradle --configuration-cache someTask -DsomeDestination=another  
Reusing configuration cache.  
> Task :someTask  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed  
Configuration cache entry reused.
```

We're now done with fixing the problems with this simple task.

Keep reading to learn how to adopt the configuration cache for your build or your plugins.

Declare a task incompatible with the configuration cache

It is possible to declare that a particular task is not compatible with the configuration cache via the `Task.notCompatibleWithConfigurationCache()` method.

Configuration cache problems found in tasks marked incompatible will no longer cause the build to fail.

And, when an incompatible task is scheduled to run, Gradle discards the configuration state at the end of the build. You can use this to help with migration, by temporarily opting out certain tasks that are difficult to change to work with the configuration cache.

Check the method documentation for more details.

Adoption steps

An important prerequisite is to keep your Gradle and plugins versions up to date. The following explores the recommended steps for a successful adoption. It applies both to builds and plugins. While going through these steps, keep in mind the HTML report and the solutions explained in the [requirements](#) chapter below.

Start with `:help`

Always start by trying your build or plugin with the simplest task `:help`. This will exercise the minimal configuration phase of your build or plugin.

Progressively target useful tasks

Don't go with running `build` right away. You can also use `--dry-run` to discover more configuration time problems first.

When working on a build, progressively target your development feedback loop. For example, running tests after making some changes to the source code.

When working on a plugin, progressively target the contributed or configured tasks.

Explore by turning problems into warnings

Don't stop at the first build failure; [turn problems into warnings](#) to discover how your build and plugins behave. If a build fails, use the HTML report to reason about the reported problems related to the failure. Continue running more useful tasks.

This will give you a good overview of the nature of the problems your build and plugins are facing. Remember that when turning problems into warnings you might need to [manually invalidate the cache](#) in case of troubles.

Step back and fix problems iteratively

When you feel you know enough about what needs to be fixed, take a step back and start iteratively fixing the most important problems. Use the HTML report and this documentation to help you in this journey.

Start with problems reported when *storing* the configuration cache. Once fixed, you can rely on a valid cached configuration phase and move on to fixing problems reported when *loading* the

configuration cache if any.

Report encountered issues

If you face a problem with a [Gradle feature](#) or with a [Gradle core plugin](#) that is not covered by this documentation, please report an issue on [gradle/gradle](#).

If you face a problem with a community Gradle plugin, see if it is already listed at [gradle/gradle#13490](#) and consider reporting the issue to the plugin's issue tracker.

A good way to report such issues is by providing information such as:

- a link to this very documentation,
- the plugin version you tried,
- the custom configuration of the plugin if any, or ideally a reproducer build,
- a description of what fails, for example problems with a given task
- a copy of the build failure,
- the self-contained [configuration-cache-report.html](#) file.

Test, test, test

Consider adding tests for your build logic. See the below section on [testing your build logic](#) for the configuration cache. This will help you while iterating on the required changes and prevent future regressions.

Roll it out to your team

Once you have your developer workflow working, for example running tests from the IDE, you can consider enabling it for your team. A faster turnaround when changing code and running tests could be worth it. You'll probably want to do this as an opt-in first.

If needed, turn problems into warnings and set the maximum number of allowed problems in your build [gradle.properties](#) file. Keep the configuration cache disabled by default. Let your team know they can opt-in by, for example, enabling the configuration cache on their IDE run configurations for the supported workflow.

Later on, when more workflows are working, you can flip this around. Enable the configuration cache by default, configure CI to disable it, and if required communicate the unsupported workflow(s) for which the configuration cache needs to be disabled.

Reacting to the configuration cache in the build

Build logic or plugin implementations can detect if the configuration cache is enabled for a given build, and react to it accordingly. The [active](#) status of the configuration cache is provided in the corresponding [build feature](#). You can access it by [injecting](#) the [BuildFeatures](#) service into your code.

You can use this information to configure features of your plugin differently or to disable an optional feature that is not yet compatible. Another example involves providing additional guidance for your users, should they need to adjust their setup or be informed of temporary limitations.

Adopting changes in the configuration cache behavior

Gradle releases bring enhancements to the configuration cache, making it detect more cases of configuration logic interacting with the environment. Those changes improve the correctness of the cache by eliminating potential false cache hits. On the other hand, they impose stricter rules that plugins and build logic need to follow to be cached as often as possible.

Some of those configuration inputs may be considered "benign" if their results do not affect the configured tasks. Having new configuration misses because of them may be undesirable for the build users, and the suggested strategy for eliminating them is:

- Identify the configuration inputs causing the invalidation of the configuration cache with the help of the [configuration cache report](#).
 - Fix undeclared configuration inputs accessed by the build logic of the project.
 - Report issues caused by third-party plugins to the plugin maintainers, and update the plugins once they get fixed.
- For some kinds of configuration inputs, it is possible to use the opt-out options that make Gradle fall back to the earlier behavior, omitting the inputs from detection. This temporary workaround is aimed to mitigate performance issues coming from out-of-date plugins.

It is possible to temporarily opt out of configuration input detection in the following cases:

- Since Gradle 8.1, using many APIs related to the file system is correctly tracked as configuration inputs, including the file system checks, such as `File.exists()` or `File.isFile()`.

For the input tracking to ignore these file system checks on the specific paths, the Gradle property `org.gradle.configuration-cache.inputs.unsafe.ignore.file-system-checks`, with the list of the paths, relative to the root project directory and separated by `;`, can be used. To ignore multiple paths, use `*` to match arbitrary strings within one segment, or `**` across segments. Paths starting with `~/` are based on the user home directory. For example:

gradle.properties

```
org.gradle.configuration-cache.inputs.unsafe.ignore.file-system-checks=\
  ~/.third-party-plugin/*.lock;\
  ../../externalOutputDirectory/**;\
  build/analytics.json
```

- Before Gradle 8.4, some undeclared configuration inputs that were never used in the configuration logic could still be read when the task graph was serialized by the configuration cache. However, their changes would not invalidate the configuration cache afterward. Starting with Gradle 8.4, such undeclared configuration inputs are correctly tracked.

To temporarily revert to the earlier behavior, set the Gradle property `org.gradle.configuration-cache.inputs.unsafe.ignore.in-serialization` to `true`.

Ignore configuration inputs sparingly, and only if they do not affect the tasks produced by the configuration logic. The support for these options will be removed in future releases.

Testing your build logic

The Gradle TestKit (a.k.a. just TestKit) is a library that aids in testing Gradle plugins and build logic generally. For general guidance on how to use TestKit, see the [dedicated chapter](#).

To enable configuration caching in your tests, you can pass the `--configuration-cache` argument to [GradleRunner](#) or use one of the other methods described in [Enabling the configuration cache](#).

You need to run your tasks twice. Once to prime the configuration cache. Once to reuse the configuration cache.

Example 5. Testing the configuration cache

src/test/kotlin/org/example/BuildLogicFunctionalTest.kt

```
@Test
fun `my task can be loaded from the configuration cache`() {

    buildFile.writeText("""
        plugins {
            id 'org.example.my-plugin'
        }
    """)

    runner()
        .withArguments("--configuration-cache", "myTask") ①
        .build()

    val result = runner()
        .withArguments("--configuration-cache", "myTask") ②
        .build()

    require(result.output.contains("Reusing configuration cache. ")) ③
    // ... more assertions on your task behavior
}
```

src/test/groovy/org/example/BuildLogicFunctionalTest.groovy

```
def "my task can be loaded from the configuration cache"() {
    given:
    buildFile << """
        plugins {
            id 'org.example.my-plugin'
        }
    """

    when:
    runner()
```

```

        .withArguments('--configuration-cache', 'myTask') ①
        .build()

    and:
    def result = runner()
        .withArguments('--configuration-cache', 'myTask') ②
        .build()

    then:
    result.output.contains('Reusing configuration cache.') ③
    // ... more assertions on your task behavior
}

```

- ① First run primes the configuration cache.
- ② Second run reuses the configuration cache.
- ③ Assert that the configuration cache gets reused.

If problems with the configuration cache are found then Gradle will fail the build reporting the problems, and the test will fail.

TIP

A good testing strategy for a Gradle plugin is to run its whole test suite with the configuration cache enabled. This requires testing the plugin with a supported Gradle version.

If the plugin already supports a range of Gradle versions it might already have tests for multiple Gradle versions. In that case we recommend enabling the configuration cache starting with the Gradle version that supports it.

If this can't be done right away, using tests that run all tasks contributed by the plugin several times, for e.g. asserting the `UP_TO_DATE` and `FROM_CACHE` behavior, is also a good strategy.

Requirements

In order to capture the state of the task graph to the configuration cache and reload it again in a later build, Gradle applies certain requirements to tasks and other build logic. Each of these requirements is treated as a configuration cache "problem" and fails the build if violations are present.

For the most part these requirements are actually surfacing some undeclared inputs. In other words, using the configuration cache is an opt-in to more strictness, correctness and reliability for all builds.

The following sections describe each of the requirements and how to change your build to fix the problems.

Certain types must not be referenced by tasks

There are a number of types that task instances must not reference from their fields. The same applies to task actions as closures such as `doFirst {}` or `doLast {}`.

These types fall into some categories as follows:

- Live JVM state types
- Gradle model types
- Dependency management types

In all cases the reason these types are disallowed is that their state cannot easily be stored or recreated by the configuration cache.

Live JVM state types (e.g. `ClassLoader`, `Thread`, `OutputStream`, `Socket` etc...) are simply disallowed. These types almost never represent a task input or output. The only exceptions are the standard streams: `System.in`, `System.out`, and `System.err`. These streams can be used, for example, as parameters to `Exec` and `JavaExec` tasks.

Gradle model types (e.g. `Gradle`, `Settings`, `Project`, `SourceSet`, `Configuration` etc...) are usually used to carry some task input that should be explicitly and precisely declared instead.

For example, if you reference a `Project` in order to get the `project.version` at execution time, you should instead directly declare the *project version* as an input to your task using a `Property<String>`. Another example would be to reference a `SourceSet` to later get the source files, the compilation classpath or the outputs of the source set. You should instead declare these as a `FileCollection` input and reference just that.

The same requirement applies to dependency management types with some nuances.

Some types, such as `Configuration` or `SourceDirectorySet`, don't make good task input parameters, as they hold a lot of irrelevant state, and it is better to model these inputs as something more precise. We don't intend to make these types serializable at all. For example, if you reference a `Configuration` to later get the resolved files, you should instead declare a `FileCollection` as an input to your task. In the same vein, if you reference a `SourceDirectorySet` you should instead declare a `FileTree` as an input to your task.

Referencing dependency resolution results is also disallowed (e.g. `ArtifactResolutionQuery`, `ResolvedArtifact`, `ArtifactResult` etc...). For example, if you reference some `ResolvedComponentResult` instances, you should instead declare a `Provider<ResolvedComponentResult>` as an input to your task. Such a provider can be obtained by invoking `ResolutionResult.getRootComponent()`. In the same vein, if you reference some `ResolvedArtifactResult` instances, you should instead use `ArtifactCollection.getResolvedArtifacts()` that returns a `Provider<Set<ResolvedArtifactResult>>` that can be mapped as an input to your task. The rule of thumb is that tasks must not reference *resolved* results, but lazy specifications instead, in order to do the dependency resolution at execution time.

Some types, such as `Publication` or `Dependency` are not serializable, but could be. We may, if necessary, allow these to be used as task inputs directly.

Here's an example of a problematic task type referencing a **SourceSet**:

build.gradle.kts

```
abstract class SomeTask : DefaultTask() {  
  
    @get:Input lateinit var sourceSet: SourceSet ❶  
  
    @TaskAction  
    fun action() {  
        val classpathFiles = sourceSet.compileClasspath.files  
        // ...  
    }  
}
```

build.gradle

```
abstract class SomeTask extends DefaultTask {  
  
    @Input SourceSet sourceSet ❶  
  
    @TaskAction  
    void action() {  
        def classpathFiles = sourceSet.compileClasspath.files  
        // ...  
    }  
}
```

❶ this will be reported as a problem because referencing **SourceSet** is not allowed

The following is how it should be done instead:

build.gradle.kts

```
abstract class SomeTask : DefaultTask() {  
  
    @get:InputFiles @get:Classpath  
    abstract val classpath: ConfigurableFileCollection ❶  
  
    @TaskAction  
    fun action() {  
        val classpathFiles = classpath.files  
        // ...  
    }  
}
```

```
}
```

build.gradle

```
abstract class SomeTask extends DefaultTask {  
  
    @InputFiles @Classpath  
    abstract ConfigurableFileCollection getClasspath() ①  
  
    @TaskAction  
    void action() {  
        def classpathFiles = classpath.files  
        // ...  
    }  
}
```

① no more problems reported, we now reference the supported type `FileCollection`

In the same vein, if you encounter the same problem with an ad-hoc task declared in a script as follows:

build.gradle.kts

```
tasks.register("someTask") {  
    doLast {  
        val classpathFiles = sourceSets.main.get().compileClasspath.files ①  
    }  
}
```

build.gradle

```
tasks.register('someTask') {  
    doLast {  
        def classpathFiles = sourceSets.main.compileClasspath.files ①  
    }  
}
```

① this will be reported as a problem because the `doLast {}` closure is capturing a reference to the `SourceSet`

You still need to fulfil the same requirement, that is not referencing a disallowed type. Here's how

the task declaration above can be fixed:

build.gradle.kts

```
tasks.register("someTask") {  
    val classpath = sourceSets.main.get().compileClasspath ①  
    doLast {  
        val classpathFiles = classpath.files  
    }  
}
```

build.gradle

```
tasks.register('someTask') {  
    def classpath = sourceSets.main.compileClasspath ①  
    doLast {  
        def classpathFiles = classpath.files  
    }  
}
```

① no more problems reported, the `doLast {}` closure now only captures `classpath` which is of the supported `FileCollection` type

Note that sometimes the disallowed type is indirectly referenced. For example, you could have a task reference some type from a plugin that is allowed. That type could reference another allowed type that in turn references a disallowed type. The hierarchical view of the object graph provided in the HTML reports for problems should help you pinpoint the offender.

Using the `Project` object

A task must not use any `Project` objects at execution time. This includes calling `Task.getProject()` while the task is running.

Some cases can be fixed in the same way as for `disallowed types`.

Often, similar things are available on both `Project` and `Task`. For example if you need a `Logger` in your task actions you should use `Task.logger` instead of `Project.logger`.

Otherwise, you can use `injected services` instead of the methods of `Project`.

Here's an example of a problematic task type using the `Project` object at execution time:

build.gradle.kts

```
abstract class SomeTask : DefaultTask() {
    @TaskAction
    fun action() {
        project.copy { ❶
            from("source")
            into("destination")
        }
    }
}
```

build.gradle

```
abstract class SomeTask extends DefaultTask {
    @TaskAction
    void action() {
        project.copy { ❶
            from 'source'
            into 'destination'
        }
    }
}
```

❶ this will be reported as a problem because the task action is using the **Project** object at execution time

The following is how it should be done instead:

build.gradle.kts

```
abstract class SomeTask : DefaultTask() {

    @get:Inject abstract val fs: FileSystemOperations ❶

    @TaskAction
    fun action() {
        fs.copy {
            from("source")
            into("destination")
        }
    }
}
```

build.gradle

```
abstract class SomeTask extends DefaultTask {  
  
    @Inject abstract FileSystemOperations getFs() ①  
  
    @TaskAction  
    void action() {  
        fs.copy {  
            from 'source'  
            into 'destination'  
        }  
    }  
}
```

① no more problem reported, the injected `FileSystemOperations` service is supported as a replacement for `project.copy {}`

In the same vein, if you encounter the same problem with an ad-hoc task declared in a script as follows:

build.gradle.kts

```
tasks.register("someTask") {  
    doLast {  
        project.copy { ①  
            from("source")  
            into("destination")  
        }  
    }  
}
```

build.gradle

```
tasks.register('someTask') {  
    doLast {  
        project.copy { ①  
            from 'source'  
            into 'destination'  
        }  
    }  
}
```

- ① this will be reported as a problem because the task action is using the `Project` object at execution time

Here's how the task declaration above can be fixed:

build.gradle.kts

```
interface Injected {
    @get:Inject val fs: FileSystemOperations ①
}
tasks.register("someTask") {
    val injected = project.objects.newInstance<Injected>() ②
    doLast {
        injected.fs.copy { ③
            from("source")
            into("destination")
        }
    }
}
```

build.gradle

```
interface Injected {
    @Inject FileSystemOperations getFs() ①
}
tasks.register('someTask') {
    def injected = project.objects.newInstance(Injected) ②
    doLast {
        injected.fs.copy { ③
            from 'source'
            into 'destination'
        }
    }
}
```

- ① services can't be injected directly in scripts, we need an extra type to convey the injection point
- ② create an instance of the extra type using `project.object` outside the task action
- ③ no more problem reported, the task action references `injected` that provides the `FileSystemOperations` service, supported as a replacement for `project.copy {}`

As you can see above, fixing ad-hoc tasks declared in scripts requires quite a bit of ceremony. It is a good time to think about extracting your task declaration as a proper task class as shown previously.

The following table shows what APIs or injected service should be used as a replacement for each of the **Project** methods.

Instead of:	Use:
<code>project.rootDir</code>	A task input or output property or a script variable to capture the result of using <code>project.rootDir</code> to calculate the actual parameter.
<code>project.projectDir</code>	A task input or output property or a script variable to capture the result of using <code>project.projectDir</code> to calculate the actual parameter.
<code>project.buildDir</code>	A task input or output property or a script variable to capture the result of using <code>project.buildDir</code> to calculate the actual parameter.
<code>project.name</code>	A task input or output property or a script variable to capture the result of using <code>project.name</code> to calculate the actual parameter.
<code>project.description</code>	A task input or output property or a script variable to capture the result of using <code>project.description</code> to calculate the actual parameter.
<code>project.group</code>	A task input or output property or a script variable to capture the result of using <code>project.group</code> to calculate the actual parameter.
<code>project.version</code>	A task input or output property or a script variable to capture the result of using <code>project.version</code> to calculate the actual parameter.
<code>project.properties,</code> <code>project.hasProperty(name),</code> <code>project.getProperty(name)</code> <code>project.findProperty(name)</code>	<code>project.property(name),</code> or Value providers for Gradle properties.
<code>project.logger</code>	Task.logger
<code>project.provider {}</code>	ProviderFactory.provider {}
<code>project.file(path)</code>	A task input or output property or a script variable to capture the result of using <code>project.file(file)</code> to calculate the actual parameter.

Instead of:	Use:
<code>project.uri(path)</code>	A task input or output property or a script variable to capture the result of using <code>project.uri(path)</code> to calculate the actual parameter. Otherwise, <code>File.toURI()</code> or some other JVM API can be used.
<code>project.relativePath(path)</code>	<code>ProjectLayout.projectDirectory.file(path)</code>
<code>project.files(paths)</code>	<code>ObjectFactory.fileCollection().from(paths)</code>
<code>project.fileTree(paths)</code>	<code>ObjectFactory.fileTree().from(dir)</code>
<code>project.zipTree(path)</code>	<code>ArchiveOperations.zipTree(path)</code>
<code>project.tarTree(path)</code>	<code>ArchiveOperations.tarTree(path)</code>
<code>project.resources</code>	A task input or output property or a script variable to capture the result of using <code>project.resource</code> to calculate the actual parameter.
<code>project.copySpec {}</code>	<code>FileSystemOperations.copySpec {}</code>
<code>project.copy {}</code>	<code>FileSystemOperations.copy {}</code>
<code>project.sync {}</code>	<code>FileSystemOperations.sync {}</code>
<code>project.delete {}</code>	<code>FileSystemOperations.delete {}</code>
<code>project.mkdir(path)</code>	The Kotlin, Groovy or Java API available to your build logic.
<code>project.exec {}</code>	<code>ExecOperations.exec {}</code>
<code>project.javaexec {}</code>	<code>ExecOperations.javaexec {}</code>
<code>project.ant {}</code>	<code>Task.ant</code>
<code>project.createAntBuilder()</code>	<code>Task.ant</code>

Accessing a task instance from another instance

Tasks should not directly access the state of another task instance. Instead, tasks should be connected using [inputs and outputs relationships](#).

Note that this requirement makes it unsupported to write tasks that configure other tasks at execution time.

Sharing mutable objects

When storing a task to the configuration cache, all objects directly or indirectly referenced through the task's fields are serialized. In most cases, deserialization preserves reference equality: if two fields `a` and `b` reference the same instance at configuration time, then upon deserialization they will reference the same instance again, so `a == b` (or `a === b` in Groovy and Kotlin syntax) still holds. However, for performance reasons, some classes, in particular `java.lang.String`, `java.io.File`, and many implementations of `java.util.Collection` interface, are serialized without preserving the

reference equality. Upon deserialization, fields that referred to the object of such a class can refer to different but equal objects.

Let's look at a task that stores a user-defined object and an `ArrayList` in task fields.

build.gradle.kts

```
class StateObject {
    // ...
}

abstract class StatefulTask : DefaultTask() {
    @get:Internal
    var stateObject: StateObject? = null

    @get:Internal
    var strings: List<String>? = null
}

tasks.register<StatefulTask>("checkEquality") {
    val objectValue = StateObject()
    val stringsValue = arrayListOf("a", "b")

    stateObject = objectValue
    strings = stringsValue

    doLast { ①
        println("POJO reference equality: ${stateObject === objectValue}") ②
        println("Collection reference equality: ${strings === stringsValue}")
    } ③

    println("Collection equality: ${strings == stringsValue}") ④
}
```

build.gradle

```
class StateObject {
    // ...
}

abstract class StatefulTask extends DefaultTask {
    @Internal
    StateObject stateObject

    @Internal
    List<String> strings
}
```

```

}

tasks.register("checkEquality", StatefulTask) {
    def objectValue = new StateObject()
    def stringsValue = ["a", "b"] as ArrayList<String>

    stateObject = objectValue
    strings = stringsValue

    doLast { ①
        println("POJO reference equality: ${stateObject === objectValue}") ②
        println("Collection reference equality: ${strings === stringsValue}")
    ③
        println("Collection equality: ${strings == stringsValue}") ④
    }
}

```

- ① `doLast` action captures the references from the enclosing scope. These captured references are also serialized to the configuration cache.
- ② Compare the reference to an object of user-defined class stored in the task field and the reference captured in the `doLast` action.
- ③ Compare the reference to `ArrayList` instance stored in the task field and the reference captured in the `doLast` action.
- ④ Check the equality of stored and captured lists.

Running the build without the configuration cache shows that reference equality is preserved in both cases.

```

❯ gradle --no-configuration-cache checkEquality
> Task :checkEquality
POJO reference equality: true
Collection reference equality: true
Collection equality: true

```

However, with the configuration cache enabled, only the user-defined object references are the same. List references are different, though the referenced lists are equal.

```

❯ gradle --configuration-cache checkEquality
> Task :checkEquality
POJO reference equality: true
Collection reference equality: false
Collection equality: true

```

In general, it isn't recommended to share mutable objects between configuration and execution

phases. If you need to do this, you should always wrap the state in a class you define. There is no guarantee that the reference equality is preserved for standard Java, Groovy, and Kotlin types, or for Gradle-defined types.

Note that no reference equality is preserved between tasks: each task is its own "realm", so it is not possible to share objects between tasks. Instead, you can use a [build service](#) to wrap the shared state.

Accessing task extensions or conventions

Tasks should not access conventions and extensions, including extra properties, at execution time. Instead, any value that's relevant for the execution of the task should be modeled as a task property.

Using build listeners

Plugins and build scripts must not register any build listeners. That is listeners registered at configuration time that get notified at execution time. For example a [BuildListener](#) or a [TaskExecutionListener](#).

These should be replaced by [build services](#), registered to receive information about [task execution](#) if needed. Use [dataflow actions](#) to handle the build result instead of [buildFinished](#) listeners.

Running external processes

Plugin and build scripts should avoid running external processes at configuration time. In general, it is preferred to run external processes in tasks with properly declared inputs and outputs to avoid unnecessary work when the task is up-to-date. However, if needed, you should only use the configuration-cache-compatible APIs described below, instead of Java and Groovy standard APIs, or Gradle-provided methods [Project.exec](#), [Project.javaexec](#), [ExecOperations.exec](#), and [ExecOperations.javaexec](#). The flexibility of these methods prevents Gradle from determining how the calls impact the build configuration, making it difficult to ensure that the configuration cache entry can be safely reused.

For simpler cases, when grabbing the output of the process is enough, [providers.exec\(\)](#) and [providers.javaexec\(\)](#) can be used:

build.gradle.kts

```
val gitVersion = providers.exec {  
    CommandLine("git", "--version")  
}.standardOutput.asText.get()
```

build.gradle

```
def gitVersion = providers.exec {  
    CommandLine("git", "--version")  
}
```

```
}.standardOutput.asText.get()
```

For more complex cases a custom [ValueSource](#) implementation with injected [ExecOperations](#) can be used. This [ExecOperations](#) instance can be used at configuration time without restrictions.

build.gradle.kts

```
abstract class GitVersionValueSource : ValueSource<String,
ValueSourceParameters.None> {
    @get:Inject
    abstract val execOperations: ExecOperations

    override fun obtain(): String {
        val output = ByteArrayOutputStream()
        execOperations.exec {
            commandLine("git", "--version")
            standardOutput = output
        }
        return String(output.toByteArray(), Charset.defaultCharset())
    }
}
```

build.gradle

```
abstract class GitVersionValueSource implements ValueSource<String,
ValueSourceParameters.None> {
    @Inject
    abstract ExecOperations getExecOperations()

    String obtain() {
        ByteArrayOutputStream output = new ByteArrayOutputStream()
        execOperations.exec {
            it.commandLine "git", "--version"
            it.standardOutput = output
        }
        return new String(output.toByteArray(), Charset.defaultCharset())
    }
}
```

You can also use standard Java/Kotlin/Groovy process APIs like [java.lang.ProcessBuilder](#) in the [ValueSource](#).

The [ValueSource](#) implementation can then be used to create a provider with [providers.of](#):

build.gradle.kts

```
val gitVersionProvider = providers.of(GitVersionValueSource::class) {}  
val gitVersion = gitVersionProvider.get()
```

build.gradle

```
def gitVersionProvider = providers.of(GitVersionValueSource.class) {}  
def gitVersion = gitVersionProvider.get()
```

In both approaches, if the value of the provider is used at configuration time then it will become a build configuration input. The external process will be executed for every build to determine if the configuration cache is up-to-date, so it is recommended to only call fast-running processes at configuration time. If the value changes then the cache is invalidated and the process will be run again during this build as part of the configuration phase.

Reading system properties and environment variables

Plugins and build scripts may read system properties and environment variables directly at configuration time with standard Java, Groovy, or Kotlin APIs or with the value supplier APIs. Doing so makes such variable or property a build configuration input, so changing the value invalidates the configuration cache. The configuration cache report includes a list of these build configuration inputs to help track them.

In general, you should avoid reading the value of system properties and environment variables at configuration time, to avoid cache misses when value changes. Instead, you can connect the **Provider** returned by `providers.systemProperty()` or `providers.environmentVariable()` to task properties.

Some access patterns that potentially enumerate all environment variables or system properties (for example, calling `System.getenv().forEach()` or using the iterator of its `keySet()`) are discouraged. In this case, Gradle cannot find out what properties are actual build configuration inputs, so every available property becomes one. Even adding a new property will invalidate the cache if this pattern is used.

Using a custom predicate to filter environment variables is an example of this discouraged pattern:

build.gradle.kts

```
val jdkLocations = System.getenv().filterKeys {  
    it.startsWith("JDK_")  
}
```

build.gradle

```
def jdkLocations = System.getenv().findAll {  
    key, _ -> key.startsWith("JDK_")  
}
```

The logic in the predicate is opaque to the configuration cache, so all environment variables are considered inputs. One way to reduce the number of inputs is to always use methods that query a concrete variable name, such as `getenv(String)`, or `getenv().get()`:

build.gradle.kts

```
val jdkVariables = listOf("JDK_8", "JDK_11", "JDK_17")  
val jdkLocations = jdkVariables.filter { v ->  
    System.getenv(v) != null  
}.associate { v ->  
    v to System.getenv(v)  
}
```

build.gradle

```
def jdkVariables = ["JDK_8", "JDK_11", "JDK_17"]  
def jdkLocations = jdkVariables.findAll { v ->  
    System.getenv(v) != null  
}.collectEntries { v ->  
    [v, System.getenv(v)]  
}
```

The fixed code above, however, is not exactly equivalent to the original as only an explicit list of variables is supported. Prefix-based filtering is a common scenario, so there are provider-based APIs to access [system properties](#) and [environment variables](#):

build.gradle.kts

```
val jdkLocationsProvider = providers.environmentVariablesPrefixedBy("JDK_")
```

build.gradle

```
def jdkLocationsProvider = providers.environmentVariablesPrefixedBy("JDK_")
```

Note that the configuration cache would be invalidated not only when the value of the variable changes or the variable is removed but also when another variable with the matching prefix is added to the environment.

For more complex use cases a custom [ValueSource](#) implementation can be used. System properties and environment variables referenced in the code of the [ValueSource](#) do not become build configuration inputs, so any processing can be applied. Instead, the value of the [ValueSource](#) is recomputed each time the build runs and only if the value changes the configuration cache is invalidated. For example, a [ValueSource](#) can be used to get all environment variables with names containing the substring [JDK](#):

build.gradle.kts

```
abstract class EnvVarsWithSubstringValueSource : ValueSource<Map<String,
String>, EnvVarsWithSubstringValueSource.Parameters> {
    interface Parameters : ValueSourceParameters {
        val substring: Property<String>
    }

    override fun obtain(): Map<String, String> {
        return System.getenv().filterKeys { key ->
            key.contains(parameters.substring.get())
        }
    }
}

val jdkLocationsProvider =
    providers.of(EnvVarsWithSubstringValueSource::class) {
        parameters {
            substring = "JDK"
        }
    }
```

build.gradle

```
abstract class EnvVarsWithSubstringValueSource implements ValueSource<Map
<String, String>, Parameters> {
    interface Parameters extends ValueSourceParameters {
        Property<String> getSubstring()
    }
}
```

```

    Map<String, String> obtain() {
        return System.getenv().findAll { key, _ ->
            key.contains(parameters.substring.get())
        }
    }
}
def jdkLocationsProvider = providers.of(EnvVarsWithSubstringValueSource.
class) {
    parameters {
        substring = "JDK"
    }
}

```

Undeclared reading of files

Plugins and build scripts should not read files directly using the Java, Groovy or Kotlin APIs at configuration time. Instead, declare files as potential build configuration inputs using the value supplier APIs.

This problem is caused by build logic similar to this:

build.gradle.kts

```
val config = file("some.conf").readText()
```

build.gradle

```
def config = file('some.conf').text
```

To fix this problem, read files using [providers.fileContents\(\)](#) instead:

build.gradle.kts

```
val config =
    providers.fileContents(layout.projectDirectory.file("some.conf"))
        .asText
```

build.gradle

```
def config = providers.fileContents(layout.projectDirectory.file('some.conf'))
    .asText
```

In general, you should avoid reading files at configuration time, to avoid invalidating configuration cache entries when the file content changes. Instead, you can connect the `Provider` returned by `providers.fileContents()` to task properties.

Bytecode modifications and Java agent

To detect the configuration inputs, Gradle modifies the bytecode of classes on the build script classpath, like plugins and their dependencies. Gradle uses a Java agent to modify the bytecode. Integrity self-checks of some libraries may fail because of the changed bytecode or the agent's presence.

To work around this, you can use the `Worker API` with classloader or process isolation to encapsulate the library code. The bytecode of the worker's classpath is not modified, so the self-checks should pass. When process isolation is used, the worker action is executed in a separate worker process that doesn't have the Gradle Java agent installed.

In simple cases, when the libraries also provide command-line entry points (`public static void main()` method), you can also use the `JavaExec` task to isolate the library.

Handling of credentials and secrets

The configuration cache has currently no option to prevent storing secrets that are used as inputs, and so they might end up in the serialized configuration cache entry which, by default, is stored under `.gradle/configuration-cache` in your project directory.

To mitigate the risk of accidental exposure, Gradle encrypts the configuration cache. Gradle transparently generates a machine-specific secret key as required, caches it under the `GRADLE_USER_HOME` directory and uses it to encrypt the data in the project specific caches.

To enhance security further, make sure to:

- secure access to configuration cache entries;
- leverage `GRADLE_USER_HOME/gradle.properties` for storing secrets. The content of that file is not part of the configuration cache, only its fingerprint. If you store secrets in that file, care must be taken to protect access to the file content.

See [gradle/gradle#22618](#).

Providing an encryption key via `GRADLE_ENCRYPTION_KEY` environment variable

By default, Gradle automatically generates and manages the encryption key as a Java keystore

stored under the [GRADLE_USER_HOME](#) directory.

For environments where this is undesirable (for instance, when the [GRADLE_USER_HOME](#) directory is shared across machines), you may provide Gradle with the exact encryption key to use when reading or writing the cached configuration data via the [GRADLE_ENCRYPTION_KEY](#) environment variable.

IMPORTANT

You must ensure that the same encryption key is consistently provided across multiple Gradle runs, or else Gradle will not be able to reuse existing cached configurations.

Generating an encryption key that is compatible with [GRADLE_ENCRYPTION_KEY](#)

For Gradle to encrypt the configuration cache using a user-specified encryption key, you must run Gradle while having the [GRADLE_ENCRYPTION_KEY](#) environment variable set with a valid AES key, encoded as a Base64 string.

One way of generating a Base64-encoded AES-compatible key is by using a command like this:

```
❯ openssl rand -base64 16
```

This command should work on Linux, Mac OS, or on Windows, if using a tool like Cygwin.

You can then use the Base64-encoded key produced by that command and set it as the value of the [GRADLE_ENCRYPTION_KEY](#) environment variable.

Not yet implemented

Support for using configuration caching with certain Gradle features is not yet implemented. Support for these features will be added in later Gradle releases.

Sharing the configuration cache

The configuration cache is currently stored locally only. It can be reused by hot or cold local Gradle daemons. But it can't be shared between developers or CI machines.

See [gradle/gradle#13510](#).

Source dependencies

Support for [source dependencies](#) is not yet implemented. With the configuration cache enabled, no problem will be reported and the build will fail.

See [gradle/gradle#13506](#).

Using a Java agent with builds run using TestKit

When running builds using [TestKit](#), the configuration cache can interfere with Java agents, such as the Jacoco agent, that are applied to these builds.

See [gradle/gradle#25979](#).

Fine-grained tracking of Gradle properties as build configuration inputs

Currently, all external sources of Gradle properties (`gradle.properties` in project directories and in the `GRADLE_USER_HOME`, environment variables and system properties that set properties, and properties specified with command-line flags) are considered build configuration inputs regardless of what properties are actually used at configuration time. These sources, however, are not included in the configuration cache report.

See [gradle/gradle#20969](#).

Java Object Serialization

Gradle allows objects that support the [Java Object Serialization](#) protocol to be stored in the configuration cache.

The implementation is currently limited to serializable classes that either implement the `java.io.Externalizable` interface, or implement the `java.io.Serializable` interface and define one of the following combination of methods:

- a `writeObject` method combined with a `readObject` method to control exactly which information to store;
- a `writeObject` method with no corresponding `readObject`; `writeObject` must eventually call `ObjectOutputStream.defaultWriteObject`;
- a `readObject` method with no corresponding `writeObject`; `readObject` must eventually call `ObjectInputStream.defaultReadObject`;
- a `writeReplace` method to allow the class to nominate a replacement to be written;
- a `readResolve` method to allow the class to nominate a replacement for the object just read;

The following *Java Object Serialization* features are **not** supported:

- the `serialPersistentFields` member to explicitly declare which fields are serializable; the member, if present, is ignored; the configuration cache considers all but `transient` fields serializable;
- the following methods of `ObjectOutputStream` are not supported and will throw `UnsupportedOperationException`:
 - `reset()`, `writeFields()`, `putFields()`, `writeChars(String)`, `writeBytes(String)` and `writeUnshared(Any?)`.
- the following methods of `ObjectInputStream` are not supported and will throw `UnsupportedOperationException`:
 - `readLine()`, `readFully(ByteArray)`, `readFully(ByteArray, Int, Int)`, `readUnshared()`, `readFields()`, `transferTo(OutputStream)` and `readAllBytes()`.
- validations registered via `ObjectInputStream.registerValidation` are simply ignored;
- the `readObjectNoData` method, if present, is never invoked;

See [gradle/gradle#13588](#).

Accessing top-level methods and variables of a build script at execution time

A common approach to reuse logic and data in a build script is to extract repeating bits into top-level methods and variables. However, calling such methods at execution time is not currently supported if the configuration cache is enabled.

For builds scripts written in Groovy, the task fails because the method cannot be found. The following snippet uses a top-level method in the `listFiles` task:

build.gradle

```
def dir = file('data')

def listFiles(File dir) {
    dir.listFiles({ file -> file.isFile() } as FileFilter).name.sort()
}

tasks.register('listFiles') {
    doLast {
        println listFiles(dir)
    }
}
```

Running the task with the configuration cache enabled produces the following error:

```
Execution failed for task ':listFiles'.
> Could not find method listFiles() for arguments [/home/user/gradle/samples/data] on
task ':listFiles' of type org.gradle.api.DefaultTask.
```

To prevent the task from failing, convert the referenced top-level method to a static method within a class:

build.gradle

```
def dir = file('data')

class Files {
    static def listFiles(File dir) {
        dir.listFiles({ file -> file.isFile() } as FileFilter).name.sort()
    }
}
```

```
tasks.register('listFilesFixed') {
    doLast {
        println Files.listFiles(dir)
    }
}
```

Build scripts written in Kotlin cannot store tasks that reference top-level methods or variables at execution time in the configuration cache at all. This limitation exists because the captured script object references cannot be serialized. The first run of the Kotlin version of the `listFiles` task fails with the configuration cache problem.

build.gradle.kts

```
val dir = file("data")

fun listFiles(dir: File): List<String> =
    dir.listFiles { file: File -> file.isFile }.map { it.name }.sorted()

tasks.register("listFiles") {
    doLast {
        println(listFiles(dir))
    }
}
```

To make the Kotlin version of this task compatible with the configuration cache, make the following changes:

build.gradle.kts

```
object Files { ❶
    fun listFiles(dir: File): List<String> =
        dir.listFiles { file: File -> file.isFile }.map { it.name }.sorted()
}

tasks.register("listFilesFixed") {
    val dir = file("data") ❷
    doLast {
        println(Files.listFiles(dir))
    }
}
```

- ① Define the method inside an object.
- ② Define the variable in a smaller scope.

See [gradle/gradle#22879](#).

Using build services to invalidate the configuration cache

Currently, it is impossible to use a `BuildServiceProvider` or provider derived from it with `map` or `flatMap` as a parameter for the `ValueSource`, if the value of the `ValueSource` is accessed at configuration time. The same applies when such a `ValueSource` is obtained in a task that executes as part of the configuration phase, for example tasks of the `buildSrc` build or included builds contributing plugins. Note that using a `@ServiceReference` or storing `BuildServiceProvider` in an `@Internal`-annotated property of a task is safe. Generally speaking, this limitation makes it impossible to use a `BuildService` to invalidate the configuration cache.

See [gradle/gradle#24085](#).

Continuous Builds

Continuous Build allows you to automatically re-execute the requested tasks when file inputs change. You can execute the build in this mode using the `-t` or `--continuous` command-line option.

For example, you can continuously run the `test` task and all dependent tasks by running:

```
$ gradle test --continuous
```

Gradle will behave as if you ran `gradle test` after a change to sources or tests that contribute to the requested tasks. This means unrelated changes (such as changes to build scripts) will not trigger a rebuild. To incorporate build logic changes, the continuous build must be restarted manually.

Continuous build uses [file system watching](#) to detect changes to the inputs. If file system watching does not work on your system, then continuous build won't work either. In particular, continuous build does not work when using `--no-daemon`.

When Gradle detects a change to the inputs, it will not trigger the build immediately. Instead, it will wait until no additional changes are detected for a certain period of time - the quiet period. You can configure the quiet period in milliseconds by the Gradle property [org.gradle.continuous.quietperiod](#).

Terminating Continuous Build

If Gradle is attached to an interactive input source, such as a terminal, the continuous build can be exited by pressing `CTRL-D` (On Microsoft Windows, it is required to also press `ENTER` or `RETURN` after `CTRL-D`).

If Gradle is not attached to an interactive input source (e.g. is running as part of a script), the build process must be terminated (e.g. using the `kill` command or similar).

If the build is being executed via the Tooling API, the build can be cancelled using the Tooling API's cancellation mechanism.

Limitations

Under some circumstances, continuous build may not detect changes to inputs.

Creating input directories

Sometimes, creating an input directory that was previously missing does not trigger a build, due to the way file system watching works. For example, creating the `src/main/java` directory may not trigger a build. Similarly, if the input is a [filtered file tree](#) and no files are matching the filter, the creation of matching files may not trigger a build.

Inputs of untracked tasks

Changes to the inputs of [untracked tasks](#) or tasks that have no outputs may not trigger a build.

Changes to files outside of project directories

Gradle only watches for changes to files inside the project directory. Changes to files outside the project directory will go undetected and not trigger a build.

Build cycles

Gradle starts watching for changes just before a task executes. If a task modifies its own inputs while executing, Gradle will detect the change and trigger a new build. If every time the task executes, the inputs are modified again, the build will be triggered again. This isn't unique to continuous build. A task that modifies its own inputs will never be considered up-to-date when run "normally" without continuous build.

If your build enters a build cycle like this, you can track down the task by looking at the list of files reported changed by Gradle. After identifying the file(s) that are changed during each build, you should look for a task that has that file as an input. In some cases, it may be obvious (e.g., a Java file is compiled with `compileJava`). In other cases, you can use `--info` logging to find the task that is out-of-date due to the identified files.

Changes to symbolic links

In general, Gradle will not detect changes to symbolic links or to files referenced via symbolic links.

Changes to build logic are not considered

The current implementation does not recalculate the build model on subsequent builds. This means that changes to task configuration, or any other change to the build model, are effectively ignored.

Inspecting Gradle Builds

```
<div class="badge-wrapper">
  <a class="badge" href="https://dpeuniversity.gradle.com/app/courses/b5069222-cfd0-
```

```
4393-b645-7a2c713853d5/" target="_blank">
    <span class="badge-type button--blue">LEARN</span>
    <span class="badge-text">How to Use Build Scans&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;></span>
  </a>
</div>
```

Gradle provides multiple ways to inspect your build:

- Profile with build scans
- Local profile reports
- Low level profiling

What is a build scan?

Build scans are a persistent, shareable record of what happened when running a build. Build scans provide insights into your build that you can use to identify and fix performance bottlenecks.

In Gradle 4.3 and above, you can create a build scan using the `--scan` command line option:

```
$ gradle build --scan
```

For older Gradle versions, the [Build Scan Plugin User Manual](#) explains how to enable build scans.

At the end of your build, Gradle displays a URL where you can find your build scan:

```
BUILD SUCCESSFUL in 2s
4 actionable tasks: 4 executed

Publishing build scan...
https://gradle.com/s/e6ircx2wjbf7e
```

This section explains how to profile your build with build scans.

Profile with build scans

The performance page can help use build scans to profile a build. To get there, click *"Performance"* in the left hand navigation menu or follow the "Explore performance" link on the build scan home page:

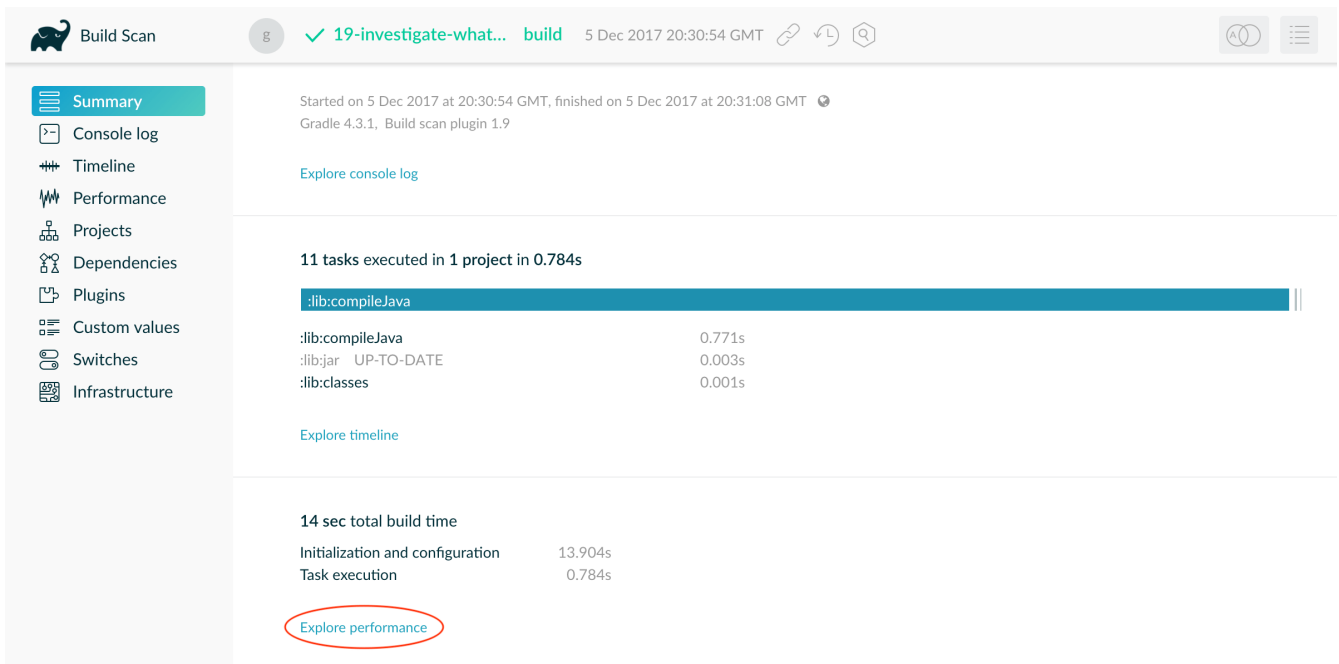


Figure 12. Performance page link on build scan home page

The performance page shows how long it took to complete different stages of a build. This page shows how long it took to:

- start up
- configure the build's projects
- resolve dependencies
- execute tasks

You also get details about environmental properties, such as whether a daemon was used or not.

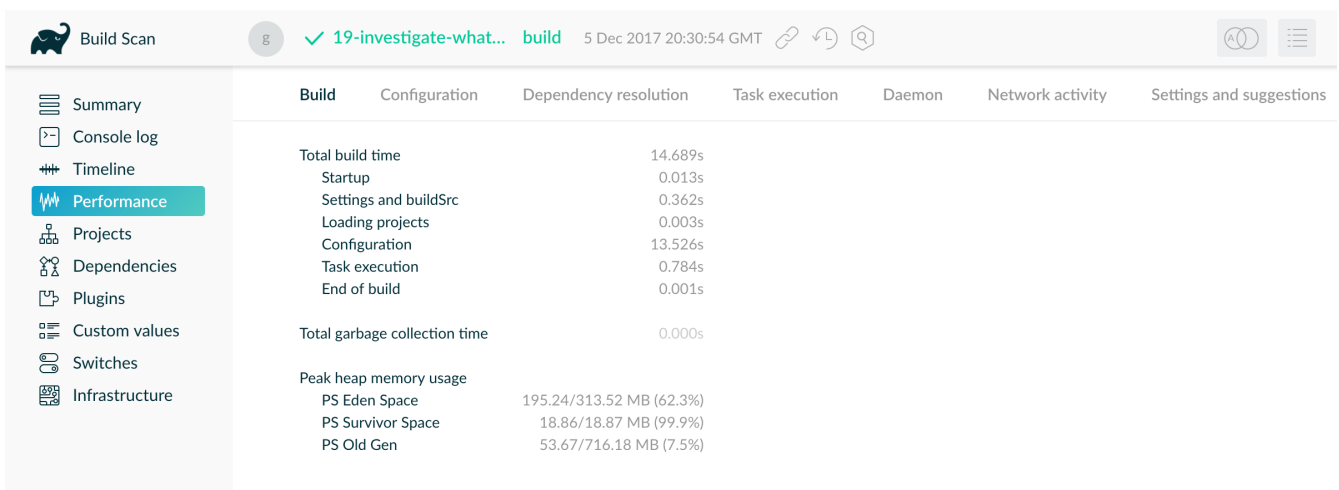


Figure 13. Build scan performance page

In the above build scan, configuration takes over 13 seconds. Click on the "Configuration" tab to break this stage into component parts, exposing the cause of the slowness.

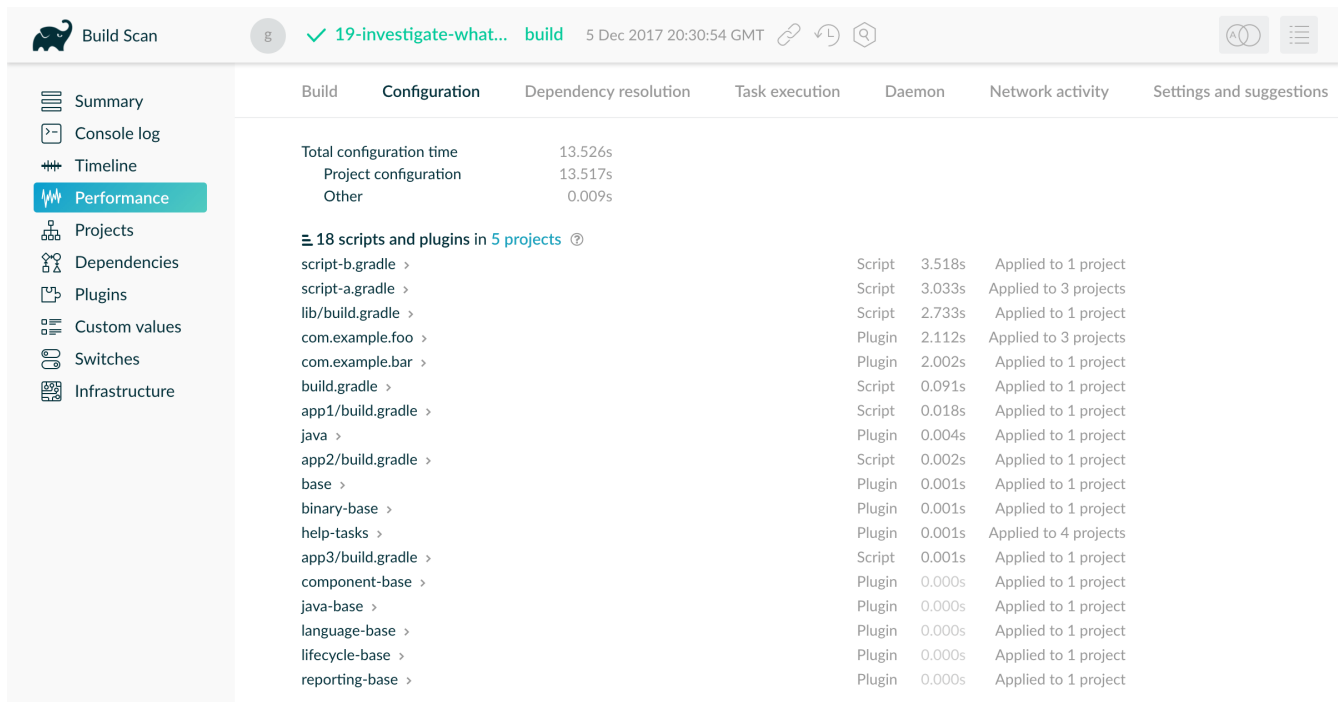


Figure 14. Build scan configuration breakdown

Here you can see the scripts and plugins applied to the project in descending order of how long they took to apply. The slowest plugin and script applications are good candidates for optimization. For example, the script `script-b.gradle` was applied once but took 3 seconds. Expand that row to see where the build applied this script.

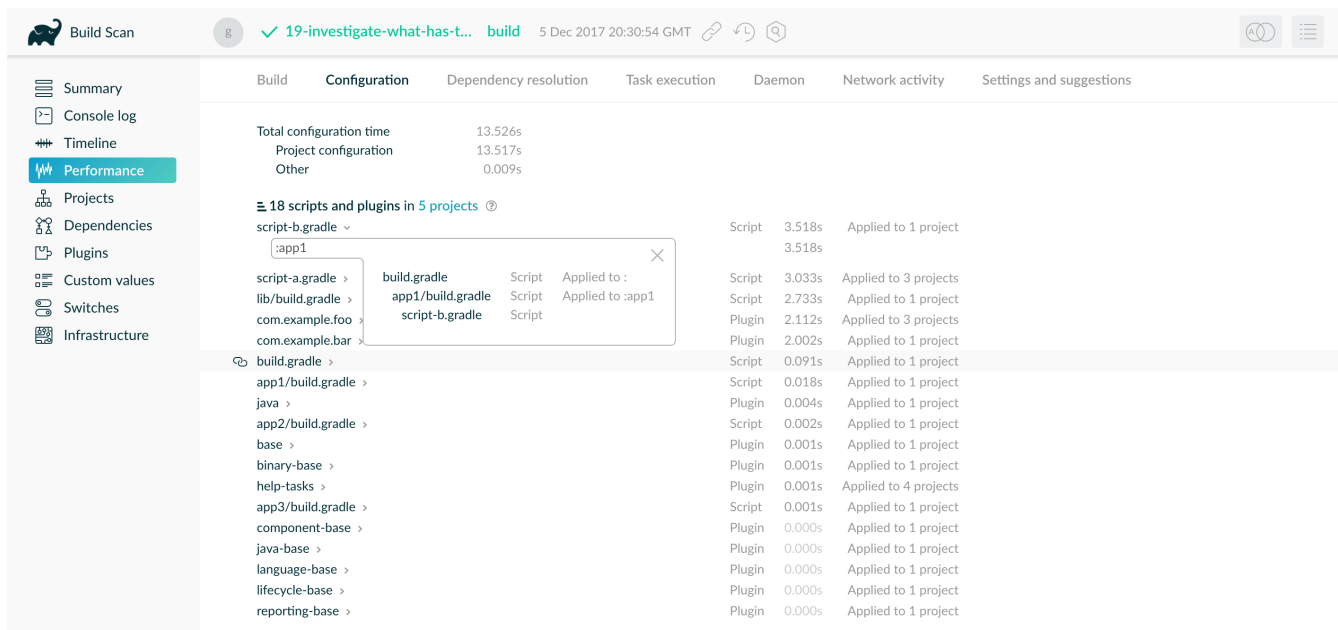


Figure 15. Showing the application of `script-b.gradle` to the build

You can see that subproject `:app1` applied the script once, from inside of that subproject's `build.gradle` file.

Profile report

If you prefer not to use build scans, you can generate an HTML report in the `build/reports/profile` directory of your root project. To generate this report, use the `--profile` command-line option:

```
$ gradle --profile <tasks>
```

Each profile report has a timestamp in its name to avoid overwriting existing ones.

The report displays a breakdown of the time taken to run the build. However, this breakdown is not as detailed as a build scan. The following profile report shows the different categories available:

Profile report

Profiled build: build

Started on: 2017/09/21 - 16:05:15

Summary

Configuration

Dependency Resolution

Task Execution

Description	Duration
Total Build Time	19.226s
Startup	0.678s
Settings and BuildSrc	1.665s
Loading Projects	0.006s
Configuring Projects	16.551s
Task Execution	0.315s

Generated by [Gradle 4.1](#) at 21-Sep-2017 16:05:32

Figure 16. An example profile report

Low level profiling

Sometimes your build can be slow even though your build scripts do everything right. This often comes down to inefficiencies in plugins and custom tasks or constrained resources. Use the [Gradle Profiler](#) to find these kinds of bottlenecks. With the Gradle Profiler, you can define scenarios like "Running 'assemble' after making an ABI-breaking change" and run your build several times to collect profiling data. Use the Profiler to produce build scans. Or combine it with method profilers like JProfiler and YourKit. These profilers can help you find inefficient algorithms in custom plugins. If you find that something in Gradle itself slows down your build, don't hesitate to send a profiler snapshot to performance@gradle.com.

Performance categories

Both build scans and local profile reports break down build execution into the same categories. The following sections explain those categories.

Startup

This reflects Gradle's initialization time, which consists mostly of:

- JVM initialization and class loading
- Downloading the Gradle distribution if you're using the wrapper
- Starting the daemon if a suitable one isn't already running
- Executing Gradle initialization scripts

Even when a build execution has a long startup time, subsequent runs usually see a dramatic drop off in startup time. Persistently slow build startup times are usually the result of problems in your init scripts. Double check that the work you're doing there is necessary and performant.

Settings and *buildSrc*

After startup, Gradle initializes your project. Usually, Gradle only processes your settings file. If you have custom build logic in a *buildSrc* directory, Gradle also processes that logic. After building *buildSrc* once, Gradle considers it up to date. The up-to-date checks take significantly less time than logic processing. If your *buildSrc* phase takes too much time, consider breaking it out into a separate project. You can then add that project's JAR artifact as a dependency.

The settings file rarely contains code with significant I/O or computation. If you find that Gradle takes a long time to process it, use more traditional profiling methods, like the [the Gradle Profiler](#), to determine the cause.

Loading projects

It normally doesn't take a significant amount of time to load projects, nor do you have any control over it. The time spent here is basically a function of the number of projects you have in your build.

Isolated Projects

Isolated Projects is a pre-alpha Gradle feature that extends the configuration cache to further improve performance, particularly the performance of Android Studio and IDEA sync.

When Isolated Projects is enabled, the configuration model of Gradle projects are "isolated" from each other. This means that build logic, such as build scripts or plugins, applied to a project cannot directly access the mutable state of another project. This allows configuration and tooling model creation for each project to safely run in parallel, with the result cached and invalidated independently for each project.

Status as of Gradle 8.11

When Isolated Projects is enabled, Gradle applies two levels of caching during IDE sync:

1. Gradle starts by applying coarse-grained caching.

To do this, Gradle caches the result of the entire sync operation and reuses it when nothing that affects the IDE model has changed. When the cache entry can be reused, Gradle short-circuits

the entire sync operation and returns the cached result to the IDE.

2. Generally, the settings and build scripts affect the IDE model, but the project's source code does not. So, when these scripts change, the cache entry cannot be reused. When this happens, Gradle falls back to fine-grained caching.

To do this, Gradle caches the result of creating the tooling models for each project and reuses these models when nothing that affects them has changed. When a project's cached models can be reused, Gradle short-circuits all work on that project, including the configuration phase and other work such as dependency resolution.

This means that Gradle will only configure and create tooling models for projects whose configuration has changed. This work is done in parallel for each project.

Current limitations

Isolated Projects is a pre-alpha feature, and as such, the current implementation has a number of limitations. These will be addressed in future Gradle releases:

- Gradle, IntelliJ IDEA, Android Studio, and Kotlin plugins are not yet 100% compatible with Isolated Projects, so you should expect to see some violations reported. The teams are actively working on fixing these incompatibilities.
- Parallel configuration does not support [Configuration on Demand](#). All projects will be configured, even if they aren't running tasks.
- Changes to included builds invalidate all cached results, even when the change would not affect the cached results.
- The implementation does not exploit the isolation to limit peak memory consumption. Currently, peak memory consumption is a function of how many projects must be configured.
- All caching, including the configuration cache, is done on the local machine. Remote caching is not supported yet.

How do I use it?

You will need Gradle 8.5 or later to use Isolated Projects, preferably a recent nightly. You should also use the most recent version of IDEA or Android Studio.

The feature is off by default. You can enable it by setting the `org.gradle.unsafe.isolated-projects` system property to `true`. For example:

```
$ gradle build -Dorg.gradle.unsafe.isolated-projects=true
```

When enabled, Gradle will fail the build whenever build logic attempts to cross project boundaries and access the model of another project. Gradle collects all of these access problems in the configuration cache report, as it does other problems.

The configuration cache command-line options can be used to control how Gradle handles these problems. For example:

- `--configuration-cache-problems=warn` can be used to treat access problems as warnings instead of errors.
- `-Dorg.gradle.configuration-cache.max-problems=x` can be used to increase the maximum number of problems included in the report.

You can also use `-Dorg.gradle.internal.invalidate-coupled-projects=false` to force parallel configuration when there are access problems.

Note that these options disable the validation that makes execution parallel- and caching-safe when Isolated Projects is enabled, so you may see some unexpected behavior when using them.

Build logic constraints

Isolated Projects prevents build logic from accessing the state of another project. This includes:

- Using most methods on the `Project` type. A small number of methods that return immutable information about the project are allowed:
 - `getName()`
 - `getPath()`
 - `getBuildTreePath()`
 - `getProjectDir()`
 - `getRootDir()`
 - `getChildProjects()`
 - `getSubprojects()`
 - `getAllProjects()`
 - `project()` overloads
 - `subprojects()` overloads
 - `allprojects()` overloads

Note that Isolated Projects is a pre-alpha feature. These constraints are not final and can change at any time.

ChangeLog

Gradle 8.11

Projects configured in parallel

When building from the command line (i.e. for task execution) with Isolated Projects enabled, projects are now configured in parallel.

Parallel configuration does not yet support Configuration on Demand. All projects will be configured, even those which have no tasks to run. If this tradeoff is undesirable and you want projects to configure serially with Configure on Demand, set the `org.gradle.internal.isolated-projects.configure-on-demand.tasks` system property to `true`.

Gradle 8.9

Lift restriction on string-based task dependency notation

Depending on a task from another project in string-notated form is a common idiom:

```
foo.dependsOn(":a:bar")
```

Starting with this release, this is no longer considered a violation of Isolated Projects boundaries.

gradle init generates Isolated Projects compatible projects

The [Build Init Plugin](#) supports creating multi-module projects.

Starting with this release, `gradle init` generates projects compatible with Isolated Projects restrictions.

IsolatedProject provides a project identifier in composite builds

The [IsolatedProject](#) type was introduced in Gradle 8.8 to explicitly mark the project state that is safe to access across projects.

Gradle 8.9 adds a `buildTreePath` member, which serves as a unique project identifier in [composite build](#) setups.

Gradle 8.8

New Gradle lifecycle callbacks

This release introduces a new `GradleLifecycle` API, accessible via `gradle.lifecycle`, which plugin authors and build engineers can use to register actions to be executed at certain points in the build lifecycle.

Actions registered as `GradleLifecycle` callbacks (currently, `beforeProject` and `afterProject`) are **isolated**, running in an isolated context that is private to every project. This will allow Gradle to perform additional performance optimizations and will be required in the future to take advantage of parallelism during the build configuration phase.

While the existing callbacks continue to work, we encourage everyone to adopt the new API and provide us with early feedback.

The example below shows how this new API could be used in a settings script or [settings plugins](#) to apply configuration to all projects, while avoiding [cross-project configuration](#):

settings.gradle.kts

```
include("sub1")
include("sub2")

gradle.lifecycle.beforeProject {
    apply(plugin = "base")
}
```

```
repositories {  
    mavenCentral()  
}  
}
```

Isolated project views

There is now support for obtaining an isolated view of a project as an `IsolatedProject` via `Project.getIsolated()`.

The view exposes only those properties that are safe to access across project boundaries when running the build configuration phase in parallel (to be supported in a future release).

The example below shows how the API could be used from a `Project` configuration callback to query the root project directory in a parallel-safe way:

```
gradle.lifecycle.beforeProject {  
    val rootDir = project.isolated.rootProject.projectDirectory  
    println("The root project directory is $rootDir")  
}
```

File System Watching

Gradle maintains a Virtual File System (VFS) to calculate what needs to be rebuilt on repeat builds of a project. By watching the file system, Gradle keeps the VFS current between builds.

Enable

Gradle enables file system watching by default for supported operating systems since Gradle 7.

Run the build with the `--watch-fs` flag to force file system watching for a build.

To force file system watching for all builds (unless disabled with `--no-watch-fs`), add the following value to `gradle.properties`:

gradle.properties

```
org.gradle.vfs.watch=true
```

Disable

To disable file system watching:

- use the `--no-watch-fs` flag
- set `org.gradle.vfs.watch=false` in `gradle.properties`

Supported Operating Systems

Gradle uses native operating system features to watch the file system. Gradle supports file system watching on the following operating systems:

- Windows 10, version 1709 and later
- Linux, tested on the following distributions:
 - Ubuntu 16.04
 - CentOS Stream 9
 - Red Hat Enterprise Linux (RHEL) 8
 - Amazon Linux 2
 - Alpine Linux 3.20
- macOS 12 (Monterey) or later on Intel and ARM architectures

Supported File Systems

File system watching supports the following file system types:

- APFS
- btrfs
- ext3
- ext4
- XFS
- HFS+
- NTFS

Gradle also supports VirtualBox's shared folders.

Network file systems like Samba and NFS are not supported. Microsoft Dev Drives (ReFS) are also not supported.

Unsupported File Systems

When enabled by default, file system watching acts conservatively when it encounters content on unsupported file systems. This can happen if you mount a project directory or subdirectory from a network drive. Gradle doesn't retain information about unsupported file systems between builds when enabled by default. If you explicitly enable file system watching, Gradle retains information about unsupported file systems between builds.

Symlinks

Files and directories in your project that are accessed via symlinks do not benefit from file system-watching optimizations.

Logging

To view information about Virtual File System (VFS) changes at the beginning and end of a build, enable verbose VFS logging.

Set the `org.gradle.vfs.verbose` Daemon option to `true` to enable verbose logging.

You can do this on the command line with the following command:

```
$ gradle <task> -Dorg.gradle.vfs.verbose=true
```

Or configure the property in the `gradle.properties` file in the project root or your Gradle User Home:

gradle.properties

```
org.gradle.vfs.verbose=true
```

This produces the following output at the start and end of the build:

```
$ gradle assemble --watch-fs -Dorg.gradle.vfs.verbose=true
```

```
Received 3 file system events since last build while watching 1 locations
Virtual file system retained information about 2 files, 2 directories and 0 missing
files since last build
> Task :compileJava NO-SOURCE
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE
> Task :jar UP-TO-DATE
> Task :assemble UP-TO-DATE

BUILD SUCCESSFUL in 58ms
1 actionable task: 1 up-to-date
Received 5 file system events during the current build while watching 1 locations
Virtual file system retains information about 3 files, 2 directories and 2 missing
files until next build
```

On Windows and macOS, Gradle might report changes received since the last build, even if you haven't changed anything. These are harmless notifications about changes to Gradle's caches and can be safely ignored.

Troubleshooting

Gradle does not detect some changes

Please [let us know on the Gradle community Slack](#). If a build declares its inputs and outputs

correctly, this should not happen. So it's either a bug we must fix or your build lacks declaration for some inputs or outputs.

VFS state dropped due to lost state

Did you receive a message that reads **Dropped VFS state due to lost state** during a build? Please [let us know on the Gradle community Slack](#). This means that your build cannot benefit from file system watching for one of the following reasons:

- the Daemon received an unknown file system event
- too many changes happened, and the watching API couldn't handle it

Too many open files on macOS

If you receive the **java.io.IOException: Too many open files** error on macOS, raise your open files limit. See [this post](#) for more details.

Adjust inotify watches limit on Linux

File system watching uses [inotify](#) on Linux. Depending on the size of your build, it may be necessary to increase inotify limits. If you are using an IDE, then you probably already had to increase the limits in the past.

File system watching uses one inotify watch per watched directory. You can see the current limit of inotify watches per user by running:

```
cat /proc/sys/fs/inotify/max_user_watches
```

To increase the limit to e.g. 512K watches run the following:

```
echo fs.inotify.max_user_watches=524288 | sudo tee -a /etc/sysctl.conf
```

```
sudo sysctl -p --system
```

Each used inotify watch takes up to 1KB of memory. Assuming inotify uses all the 512K watches then file system watching could use up to 500MB. In a memory-constrained environment, you may want to disable file system watching.

Inspect inotify instances limit on Linux

File system watching initializes one inotify instance per daemon. You can see the current limit of inotify instances per user by running:

```
cat /proc/sys/fs/inotify/max_user_instances
```

The default per-user instances limit should be high enough, so we don't recommend increasing that value manually.

[1] Not compatible with the configuration cache.

THE BUILD CACHE

Build Cache

```
<div class="badge-wrapper">
  <a class="badge" href="https://dpeuniversity.gradle.com/app/courses/ec69d0b8-9171-4969-ac3e-82dea16f87b0/" target="_blank">
    <span class="badge-type button--blue">LEARN</span>
    <span class="badge-text">Incremental Builds and Build Caching with
Gradle&nbsp;&nbsp;&nbsp;&gt;</span>
  </a>
</div>
```

Overview

The Gradle *build cache* is a cache mechanism that aims to save time by reusing outputs produced by other builds. The build cache works by storing (locally or remotely) build outputs and allowing builds to fetch these outputs from the cache when it is determined that inputs have not changed, avoiding the expensive work of regenerating them.

A first feature using the build cache is *task output caching*. Essentially, task output caching leverages the same intelligence as [up-to-date checks](#) that Gradle uses to avoid work when a previous local build has already produced a set of task outputs. But instead of being limited to the previous build in the same workspace, task output caching allows Gradle to reuse task outputs from any earlier build in any location on the local machine. When using a shared build cache for task output caching this even works across developer machines and build agents.

Apart from tasks, [artifact transforms](#) can also leverage the build cache and re-use their outputs similarly to task output caching.

TIP

For a hands-on approach to learning how to use the build cache, start with reading through the [use cases for the build cache](#) and the follow up sections. It covers the different scenarios that caching can improve and has detailed discussions of the different caveats you need to be aware of when enabling caching for a build.

Enable the Build Cache

By default, the build cache is not enabled. You can enable the build cache in a couple of ways:

Run with `--build-cache` on the command-line

Gradle will use the build cache for this build only.

Put `org.gradle.caching=true` in your `gradle.properties`

Gradle will try to reuse outputs from previous builds for all builds, unless explicitly disabled with `--no-build-cache`.

When the build cache is enabled, it will store build outputs in the Gradle User Home. For configuring this directory or different kinds of build caches see [Configure the Build Cache](#).

Task Output Caching

Beyond incremental builds described in [up-to-date checks](#), Gradle can save time by reusing outputs from previous executions of a task by matching inputs to the task. Task outputs can be reused between builds on one computer or even between builds running on different computers via a build cache.

We have focused on the use case where users have an organization-wide remote build cache that is populated regularly by continuous integration builds. Developers and other continuous integration agents should load cache entries from the remote build cache. We expect that developers will not be allowed to populate the remote build cache, and all continuous integration builds populate the build cache after running the `clean` task.

For your build to play well with task output caching it must work well with the [incremental build](#) feature. For example, when running your build twice in a row all tasks with outputs should be **UP-TO-DATE**. You cannot expect faster builds or correct builds when enabling task output caching when this prerequisite is not met.

Task output caching is automatically enabled when you enable the build cache, see [Enable the Build Cache](#).

What does it look like

Let us start with a project using the Java plugin which has a few Java source files. We run the build the first time.

```
> gradle --build-cache compileJava
:compileJava
:processResources
:classes
:jar
:assemble

BUILD SUCCESSFUL
```

We see the directory used by the local build cache in the output. Apart from that the build was the same as without the build cache. Let's clean and run the build again.

```
> gradle clean
:clean

BUILD SUCCESSFUL
```

```
> gradle --build-cache assemble
:compileJava FROM-CACHE
:processResources
:classes
```

```
:jar
:assemble
```

```
BUILD SUCCESSFUL
```

Now we see that, instead of executing the `:compileJava` task, the outputs of the task have been loaded from the build cache. The other tasks have not been loaded from the build cache since they are not cacheable. This is due to `:classes` and `:assemble` being [lifecycle tasks](#) and `:processResources` and `:jar` being Copy-like tasks which are not cacheable since it is generally faster to execute them.

Cacheable tasks

Since a task describes all of its inputs and outputs, Gradle can compute a *build cache key* that uniquely defines the task's outputs based on its inputs. That build cache key is used to request previous outputs from a build cache or store new outputs in the build cache. If the previous build outputs have been already stored in the cache by someone else, e.g. your continuous integration server or other developers, you can avoid executing most tasks locally.

The following inputs contribute to the build cache key for a task in the same way that they do for [up-to-date checks](#):

- The task type and its classpath
- The names of the output properties
- The names and values of properties annotated as described in [the section called "Custom task types"](#)
- The names and values of properties added by the DSL via [TaskInputs](#)
- The classpath of the Gradle distribution, buildSrc and plugins
- The content of the build script when it affects execution of the task

Task types need to opt-in to task output caching using the [@CacheableTask](#) annotation. Note that [@CacheableTask](#) is not inherited by subclasses. Custom task types are *not* cacheable by default.

Built-in cacheable tasks

Currently, the following built-in Gradle tasks are cacheable:

- Java toolchain: [JavaCompile](#), [Javadoc](#)
- Groovy toolchain: [GroovyCompile](#), [Groovydoc](#)
- Scala toolchain: [ScalaCompile](#), `org.gradle.language.scala.tasks.PlatformScalaCompile` (removed), [ScalaDoc](#)
- Native toolchain: [CppCompile](#), [CCompile](#), [SwiftCompile](#)
- Testing: [Test](#)
- Code quality tasks: [Checkstyle](#), [CodeNarc](#), [Pmd](#)
- JaCoCo: [JacocoReport](#)

- Other tasks: [AntlrTask](#), [ValidatePlugins](#), [WriteProperties](#)

All other built-in tasks are currently not cacheable.

Some tasks, like [Copy](#) or [Jar](#), usually do not make sense to make cacheable because Gradle is only copying files from one location to another. It also doesn't make sense to make tasks cacheable that do not produce outputs or have no task actions.

Third party plugins

There are third party plugins that work well with the build cache. The most prominent examples are the [Android plugin 3.1+](#) and the [Kotlin plugin 1.2.21+](#). For other third party plugins, check their documentation to find out whether they support the build cache.

Declaring task inputs and outputs

It is very important that a cacheable task has a complete picture of its inputs and outputs, so that the results from one build can be safely re-used somewhere else.

Missing task inputs can cause incorrect cache hits, where different results are treated as identical because the same cache key is used by both executions. Missing task outputs can cause build failures if Gradle does not completely capture all outputs for a given task. Wrongly declared task inputs can lead to cache misses especially when containing volatile data or absolute paths. (See [the section called "Task inputs and outputs"](#) on what should be declared as inputs and outputs.)

NOTE

The task path is *not* an input to the build cache key. This means that tasks with different task paths can re-use each other's outputs as long as Gradle determines that executing them yields the same result.

In order to ensure that the inputs and outputs are properly declared use integration tests (for example using TestKit) to check that a task produces the same outputs for identical inputs and captures all output files for the task. We suggest adding tests to ensure that the task inputs are relocatable, i.e. that the task can be loaded from the cache into a different build directory (see [@PathSensitive](#)).

In order to handle volatile inputs for your tasks consider [configuring input normalization](#).

Marking tasks as non-cacheable by default

There are certain tasks that don't benefit from using the build cache. One example is a task that only moves data around the file system, like a [Copy](#) task. You can signify that a task is not to be cached by adding the [@DisableCachingByDefault](#) annotation to it. You can also give a human-readable reason for not caching the task by default. The annotation can be used on its own, or together with [@CacheableTask](#).

NOTE

This annotation is only for documenting the reason behind not caching the task by default. Build logic can override this decision via the runtime API (see below).

Enable caching of non-cacheable tasks

As we have seen, built-in tasks, or tasks provided by plugins, are cacheable if their class is annotated with the `Cacheable` annotation. But what if you want to make cacheable a task whose class is not cacheable? Let's take a concrete example: your build script uses a generic `NpmTask` task to create a JavaScript bundle by delegating to NPM (and running `npm run bundle`). This process is similar to a complex compilation task, but `NpmTask` is too generic to be cacheable by default: it just takes arguments and runs npm with those arguments.

The inputs and outputs of this task are simple to figure out. The inputs are the directory containing the JavaScript files, and the NPM configuration files. The output is the bundle file generated by this task.

Using annotations

We create a subclass of the `NpmTask` and use [annotations to declare the inputs and outputs](#).

When possible, it is better to use delegation instead of creating a subclass. That is the case for the built in `JavaExec`, `Exec`, `Copy` and `Sync` tasks, which have a method on `Project` to do the actual work.

If you're a modern JavaScript developer, you know that bundling can be quite long, and is worth caching. To achieve that, we need to tell Gradle that it's allowed to cache the output of that task, using the `@CacheableTask` annotation.

This is sufficient to make the task cacheable on your own machine. However, input files are identified by default by their absolute path. So if the cache needs to be shared between several developers or machines using different paths, that won't work as expected. So we also need to set the [path sensitivity](#). In this case, the relative path of the input files can be used to identify them.

Note that it is possible to override property annotations from the base class by overriding the getter of the base class and annotating that method.

Example 6. Custom cacheable BundleTask

build.gradle.kts

```
@CacheableTask ①
abstract class BundleTask : NpmTask() {

    @get:Internal ②
    override val args
        get() = super.args

    @get:InputDirectory
    @get:SkipWhenEmpty
    @get:PathSensitive(PathSensitivity.RELATIVE) ③
    abstract val scripts: DirectoryProperty

    @get:InputFiles
```

```

@get:PathSensitive(PathSensitivity.RELATIVE) ④
abstract val configFiles: ConfigurableFileCollection

@get:OutputFile
abstract val bundle: RegularFileProperty

init {
    args.addAll("run", "bundle")
    bundle = projectLayout.buildDirectory.file("bundle.js")
    scripts = projectLayout.projectDirectory.dir("scripts")
    configFiles.from(projectLayout.projectDirectory.file("package.json"))
    configFiles.from(projectLayout.projectDirectory.file("package-
lock.json"))
}

tasks.register<BundleTask>("bundle")

```

build.gradle

```

@CacheableTask ①
abstract class BundleTask extends NpmTask {

    @Override @Internal ②
    ListProperty<String> getArgs() {
        super.getArgs()
    }

    @InputDirectory
    @SkipWhenEmpty
    @PathSensitive(PathSensitivity.RELATIVE) ③
    abstract DirectoryProperty getScripts()

    @InputFiles
    @PathSensitive(PathSensitivity.RELATIVE) ④
    abstract ConfigurableFileCollection getConfigFiles()

    @OutputFile
    abstract RegularFileProperty getBundle()

    BundleTask() {
        args.addAll("run", "bundle")
        bundle = projectLayout.buildDirectory.file("bundle.js")
        scripts = projectLayout.projectDirectory.dir("scripts")
        configFiles.from(projectLayout.projectDirectory.file("package.json"))
        configFiles.from(projectLayout.projectDirectory.file("package-
lock.json"))
    }
}

```

```
tasks.register('bundle', BundleTask)
```

- (1) Add `@CacheableTask` to enable caching for the task.
- (2) Override the getter of a property of the base class to change the input annotation to `@Internal`.
- (3) (4) Declare the path sensitivity.

Using the runtime API

If for some reason you cannot create a new custom task class, it is also possible to make a task cacheable using the [runtime API](#) to declare the inputs and outputs.

For enabling caching for the task you need to use the [TaskOutputs.cacheIf\(\)](#) method.

The declarations via the runtime API have the same effect as the annotations described above. Note that you cannot override file inputs and outputs via the runtime API. Input properties can be overridden by specifying the same property name.

Example 7. [Make the bundle task cacheable](#)

build.gradle.kts

```
tasks.register<NpmTask>("bundle") {
    args = listOf("run", "bundle")

    outputs.cacheIf { true }

    inputs.dir(file("scripts"))
        .withPropertyName("scripts")
        .withPathSensitivity(PathSensitivity.RELATIVE)

    inputs.files("package.json", "package-lock.json")
        .withPropertyName("configFiles")
        .withPathSensitivity(PathSensitivity.RELATIVE)

    outputs.file(layout.buildDirectory.file("bundle.js"))
        .withPropertyName("bundle")
}
```

build.gradle

```
tasks.register('bundle', NpmTask) {
    args = ['run', 'bundle']
}
```

```

outputs.cacheIf { true }

inputs.dir(file("scripts"))
    .withPropertyName("scripts")
    .withPathSensitivity(PathSensitivity.RELATIVE)

inputs.files("package.json", "package-lock.json")
    .withPropertyName("configFiles")
    .withPathSensitivity(PathSensitivity.RELATIVE)

outputs.file(layout.buildDirectory.file("bundle.js"))
    .withPropertyName("bundle")
}

```

Configure the Build Cache

You can configure the build cache by using the [Settings.buildCache\(org.gradle.api.Action\)](#) block in `settings.gradle`.

Gradle supports a `local` and a `remote` build cache that can be configured separately. When both build caches are enabled, Gradle tries to load build outputs from the local build cache first, and then tries the remote build cache if no build outputs are found. If outputs are found in the remote cache, they are also stored in the local cache, so next time they will be found locally. Gradle stores ("pushes") build outputs in any build cache that is enabled and has [BuildCache.isPush\(\)](#) set to `true`.

By default, the local build cache has push enabled, and the remote build cache has push disabled.

The local build cache is pre-configured to be a [DirectoryBuildCache](#) and enabled by default. The remote build cache can be configured by specifying the type of build cache to connect to ([BuildCacheConfiguration.remote\(java.lang.Class\)](#)).

Built-in local build cache

The built-in local build cache, [DirectoryBuildCache](#), uses a directory to store build cache artifacts. By default, this directory resides in the Gradle User Home, but its location is configurable.

For more details on the configuration options refer to the DSL documentation of [DirectoryBuildCache](#). Here is an example of the configuration.

Example 8. Configure the local cache

settings.gradle.kts

```

buildCache {
    local {
        directory = File(rootDir, "build-cache")
    }
}

```

```
}
```

settings.gradle

```
buildCache {  
    local {  
        directory = new File(rootDir, 'build-cache')  
    }  
}
```

Gradle will periodically clean-up the local cache directory by removing entries that have not been used recently to conserve disk space. How often Gradle will perform this clean-up and how long entries will be retained is configurable via an init-script as demonstrated [in this section](#).

Remote HTTP build cache

[HttpBuildCache](#) provides the ability read to and write from a remote cache via HTTP.

With the following configuration, the local build cache will be used for storing build outputs while the local and the remote build cache will be used for retrieving build outputs.

Example 9. Load from HttpBuildCache

settings.gradle.kts

```
buildCache {  
    remote<HttpBuildCache> {  
        url = uri("https://example.com:8123/cache/")  
    }  
}
```

settings.gradle

```
buildCache {  
    remote(HttpBuildCache) {  
        url = 'https://example.com:8123/cache/'  
    }  
}
```

When attempting to load an entry, a **GET** request is made to <https://example.com:8123/cache/«cache-key»>. The response must have a **2xx** status and the cache entry as the body, or a **404 Not Found** status

if the entry does not exist.

When attempting to store an entry, a **PUT** request is made to `https://example.com:8123/cache/«cache-key»`. Any **2xx** response status is interpreted as success. A **413 Payload Too Large** response may be returned to indicate that the payload is larger than the server will accept, which will not be treated as an error.

Specifying access credentials

HTTP Basic Authentication is supported, with credentials being sent preemptively.

Example 10. Specifying access credentials

settings.gradle.kts

```
buildCache {
    remote<HttpBuildCache> {
        url = uri("https://example.com:8123/cache/")
        credentials {
            username = "build-cache-user"
            password = "some-complicated-password"
        }
    }
}
```

settings.gradle

```
buildCache {
    remote(HttpBuildCache) {
        url = 'https://example.com:8123/cache/'
        credentials {
            username = 'build-cache-user'
            password = 'some-complicated-password'
        }
    }
}
```

Redirects

3xx redirecting responses will be followed automatically.

Servers must take care when redirecting **PUT** requests as only **307** and **308** redirect responses will be followed with a **PUT** request. All other redirect responses will be followed with a **GET** request, as per [RFC 7231](#), without the entry payload as the body.

Network error handling

Requests that fail during request transmission, after having established a TCP connection, will be retried automatically.

This prevents temporary problems, such as connection drops, read or write timeouts, and low level network failures such as a connection resets, causing cache operations to fail and disabling the remote cache for the remainder of the build.

Requests will be retried up to 3 times. If the problem persists, the cache operation will fail and the remote cache will be disabled for the remainder of the build.

Using SSL

By default, use of HTTPS requires the server to present a certificate that is trusted by the build's Java runtime. If your server's certificate is not trusted, you can:

1. Update the trust store of your Java runtime to allow it to be trusted
2. Change the [build environment](#) to use an alternative trust store for the build runtime
3. Disable the requirement for a trusted certificate

The trust requirement can be disabled by setting `HttpBuildCache.isAllowUntrustedServer()` to `true`. Enabling this option is a security risk, as it allows any cache server to impersonate the intended server. It should only be used as a temporary measure or in very tightly controlled network environments.

Example 11. [Allow untrusted cache server](#)

settings.gradle.kts

```
buildCache {
    remote<HttpBuildCache> {
        url = uri("https://example.com:8123/cache/")
        isAllowUntrustedServer = true
    }
}
```

settings.gradle

```
buildCache {
    remote(HttpBuildCache) {
        url = 'https://example.com:8123/cache/'
        allowUntrustedServer = true
    }
}
```

HTTP expect-continue

Use of [HTTP Expect-Continue](#) can be enabled. This causes upload requests to happen in two parts: first a check whether a body would be accepted, then transmission of the body if the server indicates it will accept it.

This is useful when uploading to cache servers that routinely redirect or reject upload requests, as it avoids uploading the cache entry just to have it rejected (e.g. the cache entry is larger than the cache will allow) or redirected. This additional check incurs extra latency when the server accepts the request, but reduces latency when the request is rejected or redirected.

Not all HTTP servers and proxies reliably implement Expect-Continue. Be sure to check that your cache server does support it before enabling.

To enable, set [HttpBuildCache.isUseExpectContinue\(\)](#) to `true`.

Example 12. Use Expect-Continue

settings.gradle.kts

```
buildCache {
    remote<HttpBuildCache> {
        url = uri("https://example.com:8123/cache/")
        isUseExpectContinue = true
    }
}
```

settings.gradle

```
buildCache {
    remote(HttpBuildCache) {
        url = 'https://example.com:8123/cache/'
        useExpectContinue = true
    }
}
```

Configuration use cases

The recommended use case for the remote build cache is that your continuous integration server populates it from clean builds while developers only load from it. The configuration would then look as follows.

Example 13. *Recommended setup for CI push use case*

settings.gradle.kts

```
val isCiServer = System.getenv().containsKey("CI")

buildCache {
    remote<HttpBuildCache> {
        url = uri("https://example.com:8123/cache/")
        isPush = isCiServer
    }
}
```

settings.gradle

```
boolean isCiServer = System.getenv().containsKey("CI")

buildCache {
    remote(HttpBuildCache) {
        url = 'https://example.com:8123/cache/'
        push = isCiServer
    }
}
```

It is also possible to configure the build cache from an [init script](#), which can be used from the command line, added to your Gradle User Home or be a part of your custom Gradle distribution.

Example 14. *Init script to configure the build cache*

init.gradle.kts

```
gradle.settingsEvaluated {
    buildCache {
        // vvv Your custom configuration goes here
        remote<HttpBuildCache> {
            url = uri("https://example.com:8123/cache/")
        }
        // ^^^ Your custom configuration goes here
    }
}
```

init.gradle

```
gradle.settingsEvaluated { settings ->
    settings.buildCache {
        // vvv Your custom configuration goes here
        remote(HttpBuildCache) {
            url = 'https://example.com:8123/cache/'
        }
        // ^^^ Your custom configuration goes here
    }
}
```

Build cache, composite builds and **buildSrc**

Gradle's [composite build feature](#) allows including other complete Gradle builds into another. Such included builds will inherit the build cache configuration from the top level build, regardless of whether the included builds define build cache configuration themselves or not.

The build cache configuration present for any included build is effectively ignored, in favour of the top level build's configuration. This also applies to any **buildSrc** projects of any included builds.

The **buildSrc** [directory](#) is treated as an [included build](#), and as such it inherits the build cache configuration from the top-level build.

NOTE

This configuration precedence does not apply to [plugin builds](#) included through [pluginManagement](#) as these are loaded *before* the cache configuration itself.

How to set up an HTTP build cache backend

Gradle provides a Docker image for a [build cache node](#), which can connect with Develocity for centralized management. The cache node can also be used without a Develocity installation with restricted functionality.

Implement your own Build Cache

Using a different build cache backend to store build outputs (which is not covered by the built-in support for connecting to an HTTP backend) requires implementing your own logic for connecting to your custom build cache backend. To this end, custom build cache types can be registered via [BuildCacheConfiguration.registerBuildCacheService\(java.lang.Class, java.lang.Class\)](#).

[Develocity](#) includes a high-performance, easy to install and operate, shared build cache backend.

Use cases for the build cache

This section covers the different use cases for Gradle's build cache, from local-only development to caching task outputs across large teams.

Speed up developer builds with the local cache

Even when used by a single developer only, the build cache can be very useful. Gradle's *incremental build* feature helps to avoid work that is already done, but once you re-execute a task, any previous results are forgotten. When you are switching branches back and forth, the local results get rebuilt over and over again, even if you are building something that has already been built before. The build cache remembers the earlier build results, and greatly reduces the need to rebuild things when they have already been built locally. This can also extend to rebuilding different commits, like when running `git bisect`.

The local cache can also be useful when working with a project that has multiple variants, as in the case of Android projects. Each variant has a number of tasks associated with it, and some of those task variant dimensions, despite having different names, can end up producing the same output. With the local cache enabled, reuse between task variants will happen automatically when applicable.

Share results between CI builds

The build cache can do more than go back-and-forth in time: it can also bridge physical distance between computers, allowing results generated on one machine to be re-used by another. A typical first step when introducing the build cache within a team is to enable it for builds running as part of *continuous integration* only. Using a shared HTTP build cache backend (such as [the one provided by Develocity](#)) can significantly reduce the work CI agents need to do. This translates into faster feedback for developers, and less money spent on the CI resources. Faster builds also mean fewer commits being part of each build, which makes debugging issues more efficient.

Beginning with the build cache on CI is a good first step as the environment on CI agents is usually more stable and predictable than developer machines. This helps to identify any possible issues with the build that may affect cacheability.

If you are subject to audit requirements regarding the artifacts you ship to your customers you may need to disable the build cache for certain builds. Develocity may help you with fulfilling these requirements while still using the build cache for all your builds. It allows you to easily find out which build produced an artifact coming from the build cache via build scans.

Build Scan

origin-local-cache build Dec 12, 2017 7:00:13 AM GMT

11 tasks executed in 1 project in 0.111s

Summary Console log Timeline Performance Projects Dependencies Plugins Custom values Switches Infrastructure

Path Started after Duration Class Order: Execution

:compileJava FROM-CACHE 0.000s 0.009s org.gradle.api.tasks.compile.JavaCompile

:processResources NO-SOURCE 0.009s 0.001s org.gradle.language.jvm.tasks.ProcessResources

:classes UP-TO-DATE 0.010s 0.000s org.gradle.api.DefaultTask

:jar 0.010s 0.004s org.gradle.api.tasks.bundling.Jar

:assemble 0.014s 0.000s org.gradle.api.tasks.bundling.Assembly

:compileTestJava FROM-CACHE 0.015s 0.080s org.gradle.api.tasks.compile.JavaCompile

:processTestResources NO-SOURCE 0.015s 0.000s org.gradle.language.jvm.tasks.ProcessResources

:testClasses UP-TO-DATE 0.080s 0.000s org.gradle.api.DefaultTask

:test FROM-CACHE 0.080s 0.000s org.gradle.api.tasks.bundling.Jar

:check UP-TO-DATE 0.000s 0.000s org.gradle.api.tasks.bundling.Assembly

:build 0.000s 0.000s org.gradle.api.tasks.bundling.Assembly

Navigate to scan that produced the output reused by this task

Build cache result Hit (local)

Cache key cc555ddb8eb74f1e66a255777916664

Cache artifact 939 B / 4 entries

Unpack 0.001s

The task was not up-to-date because there was no task history available.

Home Timeline Close timeline (esc)

Accelerate developer builds by reusing CI results

When multiple developers work on the same project, they don't just need to build their own changes: whenever they pull from version control, they end up having to build each other's changes as well. Whenever a developer is working on something independent of the pulled changes, they can safely reuse outputs already generated on CI. Say, you're working on module "A", and you pull in some changes to module "B" (which does not depend on your module). If those changes were already built in CI, you can download the task outputs for module "B" from the cache instead of generating them locally. A typical use case for this is when developers start their day, pull all changes from version control and then run their first build.

The changes don't need to be completely independent, either; we'll take a look at the strategies to reuse results when dependencies are involved in the section about the [different forms of normalization](#).

Combine remote results with local caching

You can utilize both a local and a remote cache for a compound effect. While loading results from a CI-filled remote cache helps to avoid work needed because of changes by other developers, the local cache can speed up switching branches and doing `git bisect`. On CI machines the local cache can act as a mirror of the remote cache, significantly reducing network usage.

Share results between developers

Allowing developers to upload their results to a shared cache is possible, but not recommended. Developers can make changes to task inputs or outputs while the task is executing. They can do this unintentionally and without noticing, for example by making changes in their IDEs while a build is running. Currently, Gradle has no good way to defend against these changes, and will simply cache whatever is in the output directory once the task is finished. This again can lead to corrupted

results being uploaded to the shared cache. This recommendation might change when Gradle has added the necessary safeguards against unintentional modification of task inputs and outputs.

WARNING

If you want to share task output from incremental builds, i.e. non-clean builds, you have to make sure that all cacheable tasks are properly configured and implemented to deal with stale output. There are for example annotation processors that do not clean up stale files in the corresponding classes/resources directories. The cache is a great forcing function to fix these problems, which will also make your incremental builds much more reliable. At the same time, until you have confidence that the incremental build behavior is flawless, only use clean builds to upload content to the cache.

Build cache performance

```
<div class="badge-wrapper">
  <a class="badge" href="https://dpeuniversity.gradle.com/app/courses/4fcbecbc-7cff-449a-a509-07cf70403f0c/" target="_blank">
    <span class="badge-type button--blue">LEARN</span>
    <span class="badge-text">Maintaining Optimal Gradle Build Cache
Performance&nbsp;&nbsp;&nbsp;&gt;</span>
  </a>
</div>
```

The sole reason to use any build cache is to make builds faster. But how much faster can you go when using the cache? Measuring the impact is both important and complicated, as cache performance is determined by many factors. Performing measurements of the cache's impact can validate the extra effort (work, infrastructure) that is required to start using the cache. These measurements can later serve as baselines for future improvements, and to watch for signs of regressions.

NOTE

Proper configuration and maintenance of a build can improve caching performance in a big way.

Fully cached builds

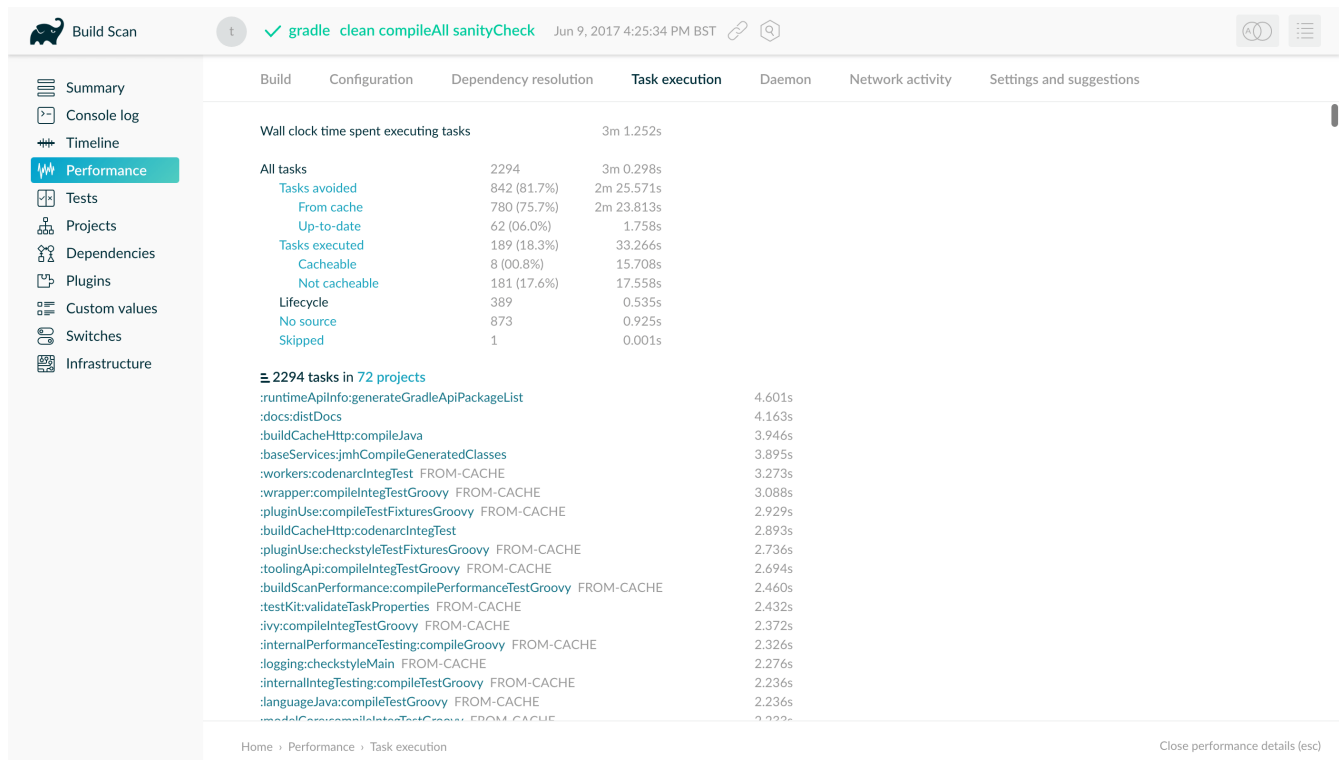
The most straightforward way to get a feel for what the cache can do for you is to measure the difference between a non-cached build and a *fully cached* build. This will give you the theoretical limit of how fast builds with the cache can get, if everything you're trying to build has already been built. The easiest way to measure this is using the local cache:

1. Clean the cache directory to avoid any hits from previous builds (`rm -rf $GRADLE_USER_HOME/caches/build-cache-*`)
2. Run the build (e.g. `./gradlew --build-cache clean assemble`), so that all the results from cacheable tasks get stored in the cache.
3. Run the build again (e.g. `./gradlew --build-cache clean assemble`); depending on your build, you should see many of the tasks being retrieved from the cache.
4. Compare the execution time for the two builds

NOTE

You may encounter a few cached tasks even in the first of the two builds, where no previously cached results should be available. This can happen if you have tasks in your build that are configured to produce the same results from the same inputs; in such a case once one of these tasks has finished, Gradle will simply reuse its output for the rest of the tasks.

Normally, your *fully cached* build should be significantly faster than the **clean** build: this is the theoretical limit of how much time using the build cache can save on your particular build. You usually don't get the achievable performance gains on the first try, see [finding problems with task output caching](#). As your build logic is evolving and changing it is also important to make sure that the cache effectiveness is not regressing. Build scans provide a detailed performance breakdown which show you how effectively your build is using the build cache:



Fully cached builds occur in situations when developers check out the latest from version control and then build, for example to generate the latest sources they need in their IDE. The purpose of running most builds though is to process some new changes. The structure of the software being built (how many modules are there, how independent are its parts etc.), and the nature of the changes themselves ("big refactor in the core of the system" vs. "small change to a unit test" etc.) strongly influence the performance gains delivered by the build cache. As developers tend to submit different kinds of changes over time, caching performance is expected to vary with each change. As with any cache, the impact should therefore be measured over time.

In a setup where a team uses a shared cache backend, there are two locations worth measuring cache impact at: on CI and on developer machines.

Cache impact on CI builds

The best way to learn about the impact of caching on CI is to set up the same builds with the cache enabled and disabled, and compare the results over time. If you have a single Gradle build step that

you want to enable caching for, it's easy to compare the results using your CI system's built-in statistical tools.

Measuring complex pipelines may require more work or external tools to collect and process measurements. It's important to distinguish those parts of the pipeline that caching has no effect on, for example, the time builds spend waiting in the CI system's queue, or time taken by checking out source code from version control.

When using Develocity, you can use the [Export API](#) to access the necessary data and run your analytics. Develocity provides much richer data compared to what can be obtained from CI servers. For example, you can get insights into the execution of single tasks, how many tasks were retrieved from the cache, how long it took to download from the cache, the properties that were used to calculate the cache key and more. When using your CI servers built in functions, you can use [statistic charts](#) if you use Teamcity for your CI builds. Most of time you will end up extracting data from your CI server via the corresponding REST API (see [Jenkins remote access API](#) and [Teamcity REST API](#)).

Typically, CI builds above a certain size include parallel sections to utilize multiple agents. With parallel pipelines you can measure the wall-clock time it takes for a set of changes to go from having been pushed to version control to being built, verified and deployed. The build cache's effect in this case can be measured in the reduction of the time developers have to wait for feedback from CI.

You can also measure the cumulative time your build agents spent building a changeset, which will give you a sense of the amount of work the CI infrastructure has to exert. The cache's effect here is less money spent on CI resources, as you don't need as many CI agents to maintain the same number of changes built.

If you want to look at the measurement for the Gradle build itself you can have a look at the blog post "[Introducing the build cache](#)".

Measuring developer builds

Gradle's build cache can be very useful in reducing CI infrastructure cost and feedback time, but it usually has the biggest impact when developers can reuse cached results in their local builds. This is also the hardest to quantify for a number of reasons:

- developers run different builds
- developers can have different hardware, or have different settings
- developers run all kinds of other things on their machines that can slow them down

When using Develocity you can use the [Export API](#) to extract data about developer builds, too. You can then create statistics on how many tasks were cached per developer or build. You can even compare the times it took to execute the task vs loading it from the cache and then estimate the time saved per developer.

When using the [Develocity build cache backend](#) you should pay close attention to the hit rate in the admin UI. A rise in the hit rate there probably indicates better usage by developers:

Gradle Enterprise Build Cache

Usage summary

Last 24 hours

Node	Stored	Evicted	Hits	Misses	Hit Rate	Copy Ins	Copy Outs	Data Received	Data Sent
Built-in	24,839	18,799	933,884	28,649	97 %	0	1,519	4.66 GB	305.57 GB
California	103	0	39,502	113	99 %	1,387	0	359.84 MB	12.43 GB
Sydney	0	0	212	562	27 %	132	0	10.34 MB	14.15 MB
Totals	24,942	18,799	973,598	29,324	97 %	1,519	1,519	5.03 GB	318.02 GB

Last 7 days

Node	Stored	Evicted	Hits	Misses	Hit Rate	Copy Ins	Copy Outs	Data Received	Data Sent
Built-in	176,777	174,368	7,703,018	200,971	97 %	0	13,757	35.91 GB	2.55 TB
California	3,625	0	817,014	4,295	99 %	13,107	0	3.55 GB	257.25 GB
Sydney	0	0	927	945	49 %	650	0	69.47 MB	80.61 MB
Totals	180,402	174,368	8,520,959	206,211	97 %	13,757	13,757	39.53 GB	2.80 TB

Last 30 days

Node	Stored	Evicted	Hits	Misses	Hit Rate	Copy Ins	Copy Outs	Data Received	Data Sent
------	--------	---------	------	--------	----------	----------	-----------	---------------	-----------

Analyzing performance in build scans

Build scans provide a summary of all cache operations for a build via the "Build cache" section of the "Performance" page.

Build Scan

t

✓ gradle clean compileAll sanityCheck

Dec 11, 2017 10:05:14 PM AEST

Summary

Console log

Timeline

Performance

Tests

Projects

Dependencies

Plugins

Custom values

Switches

Infrastructure

Build

Configuration

Dependency resolution

Task execution

Build cache

Daemon

Network activity

Settings and suggestions

Tasks whose outputs were requested from cache

974

Hit

972 (99%)

Local

0

Remote

972 (99%)

Miss

2

Tasks whose outputs were stored to cache

2

Local cache

(disabled)

Remote cache (HTTP)

Push

enabled

Configuration

Authenticated

true

AllowUntrustedServer

false

URL

<https://gradle.company.com/cache/>

Operations

Hit >

972 2.234s 33.01 MB 14.8 MB/s

Miss >

2 0.002s

Store >

2 0.025s 2.29 MB 91.7 MB/s

Packing and unpacking ⓘ

Pack >

2 0.376s 2.29 MB

Unpack >

972 4.247s 33.01 MB

Home > Performance > Build cache

Close performance details (esc)

This page details which tasks were able to be avoided by cache hits, and which missed. It also indicates the hits and misses for the local and remote caches individually. For remote cache operations, the time taken to transfer artifacts to and from the cache is given, along with the transfer rate. This is particularly important for assessing the impact of network link quality on performance, as transfer times contribute to build time.

Remote cache performance

Improving the network link between the build and the remote cache can significantly improve build cache performance. How to do this depends on the remote cache in use and your network environment.

The multi-node remote build cache provided by Develocity is a fast and efficient, purpose built, remote build cache. In particular, if your development team is geographically distributed, its replication features can significantly improve performance by allowing developers to use a cache that they have a good network link to. See the ["Build Cache Replication" section of the Develocity Admin Manual](#) for more information.

Important concepts

How much of your build gets loaded from the cache depends on many factors. In this section you will see some of the tools that are essential for well-cached builds. [Build scans](#) are part of that toolchain and will be used throughout this guide.

Build cache key

Artifacts in the build cache are uniquely identified by a [build cache key](#). A build cache key is assigned to each cacheable task when running with the build cache enabled and is used for both loading and storing task outputs to the build cache. The following inputs contribute to the build cache key for a task:

- The task implementation
- The task action implementations
- The names of the output properties
- The names and values of task inputs

Two tasks can reuse their outputs by using the build cache if their associated build cache keys are the same.

Repeatable task outputs

Assume that you have a code generator task as part of your build. When you have a fully up to date build and you clean and re-run the code generator task on the same code base it should generate *exactly the same output*, so anything that depends on that output will stay up-to-date.

It might also be that your code generator adds some extra information to its output that doesn't depend on its declared inputs, like a timestamp. In such a case re-executing the task *will* result in different code being generated (because the timestamp will be updated). Tasks that depend on the code generator's output will need to be re-executed.

When a task is cacheable, then the very nature of task output caching makes sure that the task will have the same outputs for a given set of inputs. Therefore, cacheable tasks should have repeatable task outputs. If they don't, then the result of executing the task and loading the task from the cache may be different, which can lead to hard-to-diagnose cache misses.

In some cases even well-trusted tools can produce non-repeatable outputs, and lead to cascading effects. One example is Oracle's Java compiler, which, [due to a bug](#), was producing different bytecode depending on the order source files to be compiled were presented to it. If you were using Oracle JDK 8u31 or earlier to compile code in the `buildSrc` subproject, this could lead to all of your custom tasks producing occasional cache misses, because of the difference in their classpaths (which include `buildSrc`).

The key here is that cacheable tasks should not use non-repeatable task outputs as an input.

Stable task inputs

Having a task repeatably produce the same output is not enough if its inputs keep changing all the time. Such unstable inputs can be supplied directly to the task. Consider a version number that includes a timestamp being added to the jar file's manifest:

build.gradle.kts

```
version = "3.2-${System.currentTimeMillis()}"

tasks.jar {
    manifest {
        attributes(mapOf("Implementation-Version" to project.version))
    }
}
```

build.gradle

```
version = "3.2-${System.currentTimeMillis()}"

tasks.named('jar') {
    manifest {
        attributes('Implementation-Version': project.version)
    }
}
```

In the above example the inputs for the `jar` task will be different for each build execution since this timestamp will continually change.

Another example for unstable inputs is the commit ID from version control. Maybe your version number is generated via `git describe` (and you include it in the jar manifest as shown above). Or maybe you include the commit hash directly in `version.properties` or a jar manifest attribute. Either way, the outputs produced by any tasks depending on such data will only be re-usable by builds running against the exact same commit.

Another common, but less obvious source of unstable inputs is when a task consumes the output of another task which produces non-repeatable results, such as the example before of a code generator that embeds timestamps in its output.

A task can only be loaded from the cache if it has stable task inputs. Unstable task inputs result in the task having a unique set of inputs for every build, which will always result in a cache miss.

Better reuse via input normalization

Having stable inputs is crucial for cacheable tasks. However, achieving byte for byte identical inputs for each task can be challenging. In some cases sanitizing the output of a task to remove unnecessary information can be a good approach, but this also means that a task's output can only be normalized for a single purpose.

This is where [input normalization](#) comes into play. Input normalization is used by Gradle to determine if two task inputs are *essentially* the same. Gradle uses normalized inputs when doing up-to-date checks and when determining if a cached result can be re-used instead of executing the task. As input normalization is declared by the task *consuming* the data as input, different tasks can define different ways to normalize the same data.

When it comes to file inputs, Gradle can normalize the path of the files as well as their contents.

Path sensitivity and relocatability

When sharing cached results between computers, it's rare that everyone runs the build from the exact same location on their computers. To allow cached results to be shared even when builds are executed from different root directories, Gradle needs to understand which inputs can be relocated and which cannot.

Tasks having files as inputs can declare the parts of a file's path what are essential to them: this is called the [path sensitivity](#) of the input. Task properties declared with **ABSOLUTE** path sensitivity are considered non-relocatable. This is the default for properties not declaring path sensitivity, too.

For example, the class files produced by the Java compiler are dependent on the file names of the Java source files: renaming the source files with public classes in them would fail the build. Though moving the files around wouldn't have an effect on the result of the compilation, for incremental compilation the **JavaCompile** task relies on the relative path to find other classes in the same package. Therefore, the path sensitivity for the sources of the **JavaCompile** task is **RELATIVE**. Because of this only the normalized (relative) paths of the Java source files are considered as inputs to the **JavaCompile** task.

NOTE

The Java compiler only respects the package declaration in the Java source files, not the relative path of the sources. As a consequence, path sensitivity for Java sources is **NAME_ONLY** and not **RELATIVE**.

Content normalization

Compile avoidance for Java

When it comes to the dependencies of a `JavaCompile` task (i.e. its *compile classpath*), only changes to the Application Binary Interface (ABI) of these dependencies require compilation to be executed. Gradle has a deep understanding of what a compile classpath is and uses a sophisticated normalization strategy for it. Task outputs can be re-used as long as the ABI of the classes on the compile classpath stays the same. This enables Gradle to avoid Java compilation by using incremental builds, or load results from the cache that were produced by different (but ABI-compatible) versions of dependencies. For more information on compile avoidance see the [corresponding section](#).

Runtime classpath normalization

Similar to compile avoidance, Gradle also understands the concept of a runtime classpath, and uses tailored input normalization to avoid running e.g. tests. For runtime classpaths Gradle inspects the contents of jar files and ignores the timestamps and order of the entries in the jar file. This means that a rebuilt jar file would be considered the same runtime classpath input. For details on what level of understanding Gradle has for detecting changes to classpaths and what is considered as a classpath see [this section](#).

Filtering runtime classpaths

For a runtime classpath it is possible to provide better insights to Gradle which files are essential to the input by [configuring input normalization](#).

Given that you want to add a file `build-info.properties` to all your produced jar files which contains volatile information about the build, e.g. the timestamp when the build started or some ID to identify the CI job that published the artifact. This file is only used for auditing purposes, and has no effect on the outcome of running tests. Nonetheless, this file is part of the runtime classpath for the `test` task. Since the file changes on every build invocation, tests cannot be cached effectively. To fix this you can ignore `build-info.properties` on any runtime classpath by adding the following configuration to the build script in the *consuming* project:

build.gradle.kts

```
normalization {
    runtimeClasspath {
        ignore("build-info.properties")
    }
}
```

build.gradle

```
normalization {
    runtimeClasspath {
        ignore 'build-info.properties'
    }
}
```

```
}
```

If adding such a file to your jar files is something you do for all of the projects in your build, and you want to filter this file for all consumers, you may wrap the configurations described above in an `allprojects {}` or `subprojects {}` block in the root build script.

The effect of this configuration would be that changes to `build-info.properties` would be ignored for both up-to-date checks and task output caching. All runtime classpath inputs for all tasks in the project where this configuration has been made will be affected. This will not change the runtime behavior of the `test` task—i.e. any test is still able to load `build-info.properties`, and the runtime classpath stays the same as before.

The case against overlapping outputs

When two tasks write to the same output directory or output file, it is difficult for Gradle to determine which output belongs to which task. There are many edge cases, and executing the tasks in parallel cannot be done safely. For the same reason, Gradle cannot remove [stale output files](#) for these tasks. Tasks that have discrete, non-overlapping outputs can always be handled in a safe fashion by Gradle. For the aforementioned reasons, task output caching is automatically disabled for tasks whose output directories overlap with another task.

Build scans show tasks where caching was disabled due to overlapping outputs in the timeline:

The screenshot shows the Build Scan interface for a task named `cacheability-not-cacheabl...` (full name: `notCacheableTask noOutputsT...`). The task type is `Not cacheable: Overlapping outputs`. The timeline shows a single task executed in 1 project totaling 0.006s.

Path	Started after	Duration	Cacheable	Class
<code>:cacheableTask</code>				<code>heableCustomTask</code>
Cacheable details:				
Started after	0.018s			
Duration	0.006s			
Class	<code>CacheableCustomTask</code>			
The task was not up-to-date because there was no task history available.				
Cache key	0313581203baeef484cee3a8c0a7a9ba			
Not cacheable	Overlapping outputs			

A tooltip explains the reason: "Gradle does not know how file 'build/tmp/cacheableTask/output.txt' was created (output property 'outputFile'). Task output caching requires exclusive access to output paths to guarantee correctness."

Navigation: Home > Timeline | Close timeline (esc)

Reuse of outputs between different tasks

Some builds exhibit a surprising characteristic: even when executed against an empty cache, they produce tasks loaded from cache. How is this possible? Rest assured that this is completely normal.

When considering task outputs, Gradle only cares about the inputs to the task: the task type itself, input files and parameters etc., but it doesn't care about the task's name or which project it can be found in. Running `javac` will produce the same output regardless of the name of the `JavaCompile` task that invoked it. If your build includes two tasks that share every input, the one executing later will be able to reuse the output produced by the first.

Having two tasks in the same build that do the same might sound like a problem to fix, but it is not necessarily something bad. For example, the Android plugin creates several tasks for each variant of the project; some of those tasks will potentially do the same thing. These tasks can safely reuse each other's outputs.

As [discussed previously](#), you can use Develocity to diagnose the source build of these unexpected cache-hits.

Non-cacheable tasks

You've seen quite a bit about cacheable tasks, which implies there are non-cacheable ones, too. If caching task outputs is as awesome as it sounds, why not cache every task?

There are tasks that are definitely worth caching: tasks that do complex, repeatable processing and produce moderate amounts of output. Compilation tasks are usually ideal candidates for caching. At the other end of the spectrum lie I/O-heavy tasks, like `Copy` and `Sync`. Moving files around locally typically cannot be sped up by copying them from a cache. Caching those tasks would even waste good resources by storing all those redundant results in the cache.

Most tasks are either obviously worth caching, or obviously not. For those in-between a good rule of thumb is to see if downloading results would be significantly faster than producing them locally.

Caching Java projects

As of Gradle 4.0, the build tool fully supports caching plain Java projects. Built-in tasks for compiling, testing, documenting and checking the quality of Java code support the build cache out of the box.

Java compilation

Caching Java compilation makes use of Gradle's deep understanding of compile classpaths. The mechanism [avoids recompilation](#) when dependencies change in a way that doesn't affect their application binary interfaces (ABI). Since the cache key is only influenced by the ABI of dependencies (and not by their implementation details like private types and method bodies), task output caching can also reuse compiled classes if they were produced by the same sources and ABI-equivalent dependencies.

For example, take a project with two modules: an application depending on a library. Suppose the latest version is already built by CI and uploaded to the shared cache. If a developer now modifies a method's body in the library, the library will need to be rebuilt on their computer. But they will be able to load the compiled classes for the application from the shared cache. Gradle can do this because the library used to compile the application on CI, and the modified library available locally share the same ABI.

Annotation processors

Compile avoidance works out of the box. There is one caveat though: when using annotation processors, Gradle uses the annotation processor classpath as an input. Unlike most compile dependencies, in which only the ABI influences compilation, the *implementation* of annotation processors must be considered as an input to the compiler. For this reason Gradle will treat annotation processors as a *runtime* classpath, meaning less [input normalization](#) is taking place there. If Gradle detects an annotation processor on the compile classpath, the annotation processor classpath defaults to the compile classpath when not explicitly set, which in turn means the entire compile classpath is treated as a runtime classpath input.

For the example above this would mean the ABI extracted from the compile classpath would be unchanged, but the annotation processor classpath (because it's not treated with compile avoidance) would be different. Ultimately, the developer would end up having to recompile the application.

The easiest way to avoid this performance penalty is to not use annotation processors. However, if you need to use them, make sure you set the annotation processor classpath explicitly to include only the libraries needed for annotation processing. The [section on Java compile avoidance](#) describes how to do this.

NOTE

Some common Java dependencies (such as Log4j 2.x) come bundled with annotation processors. If you use these dependencies, but do not leverage the features of the bundled annotation processors, it's best to disable annotation processing entirely. This can be done by setting the annotation processor classpath to an empty set.

Unit test execution

The **Test** task used for test execution for JVM languages employs [runtime classpath normalization](#) for its classpath. This means that changes to order and timestamps in jars on the test classpath will not cause the task to be out-of-date or change the build cache key. For achieving [stable task inputs](#) you can also wield the power of [filtering the runtime classpath](#).

Integration test execution

Unit tests are easy to cache as they normally have no external dependencies. For integration tests the situation can be quite different, as they can depend on a variety of inputs outside of the test and production code. These external factors can be for example:

- operating system type and version,
- external tools being installed for the tests,
- environment variables and Java system properties,
- other services being up and running,
- a distribution of the software under test.

You need to be careful to declare these additional inputs for your integration test in order to avoid incorrect cache hits. For example, declaring the operating system in use by Gradle as an input to a

Test task called `integTest` would work as follows:

build.gradle.kts

```
tasks.integTest {
    inputs.property("operatingSystem") {
        System.getProperty("os.name")
    }
}
```

build.gradle

```
tasks.named('integTest') {
    inputs.property("operatingSystem") {
        System.getProperty("os.name")
    }
}
```

Archives as inputs

It is common for the integration tests to depend on your packaged application. If this happens to be a zip or tar archive, then adding it as an input to the integration test task may lead to cache misses. This is because, as described in [repeatable task outputs](#), rebuilding an archive often changes the metadata in the archive. You can depend on the exploded contents of the archive instead. See also the section on dealing with [non-repeatable outputs](#).

Dealing with file paths

You will probably pass some information from the build environment to your integration test tasks by using system properties. Passing absolute paths will break [relocatability](#) of the integration test task.

build.gradle.kts

```
// Don't do this! Breaks relocatability!
tasks.integTest {
    systemProperty("distribution.location",
        layout.buildDirectory.dir("dist").get().asFile.absolutePath)
}
```

build.gradle

```
// Don't do this! Breaks relocatability!
tasks.named('integTest') {
    systemProperty "distribution.location", layout.buildDirectory.dir('dist'
).get().asFile.absolutePath
}
```

Instead of adding the absolute path directly as a system property, it is possible to add an annotated [CommandLineArgumentProvider](#) to the `integTest` task:

build.gradle.kts

```
abstract class DistributionLocationProvider : CommandLineArgumentProvider {
    ①
    @get:InputDirectory
    @get:PathSensitive(PathSensitivity.RELATIVE) ②
    abstract val distribution: DirectoryProperty

    override fun asArguments(): Iterable<String> =
        listOf("-
Ddistribution.location=${distribution.get().asFile.absolutePath}") ③
}

tasks.integTest {
    jvmArgumentProviders.add(
        objects.newInstance<DistributionLocationProvider>().apply { ④
            distribution = layout.buildDirectory.dir("dist")
        }
    )
}
```

build.gradle

```
abstract class DistributionLocationProvider implements
CommandLineArgumentProvider { ①
    @InputDirectory
    @PathSensitive(PathSensitivity.RELATIVE) ②
    abstract DirectoryProperty getDistribution()

    @Override
    Iterable<String> asArguments() {
        ["-Ddistribution.location=${distribution.get().asFile.absolutePath}"]
    }
    ③
}
```

```

    }
}

tasks.named('integTest') {
    jvmArgumentProviders.add(
        objects.newInstance(DistributionLocationProvider).tap { ④
            distribution = layout.buildDirectory.dir('dist')
        }
    )
}

```

- ① Create a class implementing `CommandLineArgumentProvider`.
- ② Declare the inputs and outputs with the corresponding path sensitivity.
- ③ `asArguments` needs to return the JVM arguments passing the desired system properties to the test JVM.
- ④ Add an instance of the newly created class as JVM argument provider to the integration test task.^[1]

Ignoring system properties

It may be necessary to ignore some system properties as inputs as they do not influence the outcome of the integration tests. In order to do so, add a `CommandLineArgumentProvider` to the `integTest` task:

build.gradle.kts

```

abstract class CiEnvironmentProvider : CommandLineArgumentProvider {
    @get:Internal ①
    abstract val agentNumber: Property<String>

    override fun asArguments(): Iterable<String> =
        listOf("-DagentNumber=${agentNumber.get()}") ②
}

tasks.integTest {
    jvmArgumentProviders.add(
        objects.newInstance<CiEnvironmentProvider>().apply { ③
            agentNumber =
                providers.environmentVariable("AGENT_NUMBER").orElse("1")
        }
    )
}

```

build.gradle

```
abstract class CiEnvironmentProvider implements CommandLineArgumentProvider {  
    @Internal ①  
    abstract Property<String> getAgentNumber()  
  
    @Override  
    Iterable<String> asArguments() {  
        ["-DagentNumber=${agentNumber.get()}"] ②  
    }  
}  
  
tasks.named('integTest') {  
    jvmArgumentProviders.add(  
        objects.newInstance(CiEnvironmentProvider).tap { ③  
            agentNumber = providers.environmentVariable("AGENT_NUMBER")  
        }.orElse("1")  
    )  
}
```

① **@Internal** means that this property does not influence the output of the integration tests.

② The system properties for the actual test execution.

③ Add an instance of the newly created class as JVM argument provider to the integration test task.^[1]

Caching Android projects

While it is true that Android uses the Java toolchain as its foundation, there are nevertheless some significant differences from pure Java projects; these differences impact task cacheability. This is even more true for Android projects that include Kotlin source code (and therefore use the **kotlin-android** plugin).

Disambiguation

This guide is about Gradle's build cache, but you may have also heard about the Android build cache. These are different things. The Android cache is internal to certain tasks in the Android plugin, and will eventually be removed in favor of native Gradle support.

Why use the build cache?

The build cache can *significantly* improve build performance for Android projects, in many cases by 30-40%. Many of the compilation and assembly tasks provided by the Android Gradle Plugin are cacheable, and more are made so with each new iteration.

Faster CI builds

CI builds benefit particularly from the build cache. A typical CI build starts with a **clean**, which means that pre-existing build outputs are deleted and none of the tasks that make up the build will be **UP-TO-DATE**. However, it is likely that many of those tasks will have been run with exactly the same inputs in a prior CI build, populating the build cache; the outputs from those prior runs can safely be reused, resulting in dramatic build performance improvements.

Reusing CI builds for local development

When you sign into work at the start of your day, it's not unusual for your first task to be pulling the main branch and then running a build (Android Studio will probably do the latter, whether you ask it to or not). Assuming all merges to main are built on CI (a best practice!), you can expect this first local build of the day to enjoy a larger-than-typical benefit with Gradle's *remote cache*. CI already built this commit — why should you re-do that work?

Switching branches

During local development, it is not uncommon to switch branches several times per day. This defeats **incremental build** (i.e., **UP-TO-DATE** checks), but this issue is mitigated via the use of the local build cache. You might run a build on Branch A, which will populate the local cache. You then switch to Branch B to conduct a code review, help a colleague, or address feedback on an open PR. You then switch back to Branch A to continue your original work. When you next build, all of the outputs previously built while working on Branch A can be reused from the cache, saving potentially a lot of time.

The Android Gradle Plugin

Android Studio users should use the latest Android Gradle Plugin to ensure compatibility and benefit from performance improvements in new releases.

The first thing you should always do when working to optimize your build is ensure you're on the latest stable, supported versions of the Android Gradle Plugin and the Gradle Build Tool. At the time of writing, they are 3.3.0 and 5.0, respectively. Each new version of these tools includes many performance improvements, not least of which is to the build cache.

Java and Kotlin compilation

The **discussion** above in "Caching Java projects" is equally relevant here, with the caveat that, for projects that include Kotlin source code, the Kotlin compiler does not currently support **compile avoidance** in the way that the Java compiler does.

Annotation processors and Kotlin

The **advice above** for pure Java projects also applies to Android projects. However, if you are using annotation processors (such as Dagger2 or Butterknife) in conjunction with Kotlin and the `kotlin-kapt` plugin, you should know that before Kotlin 1.3.30 `kapt` **was not cached by default**.

You can opt into it (which is recommended) by adding the following to build scripts:

build.gradle.kts

```
pluginManager.withPlugin("kotlin-kapt") {  
    configure<KaptExtension> { useBuildCache = true }  
}
```

build.gradle

```
plugins.withId("kotlin-kapt") {  
    kapt.useBuildCache = true  
}
```

Unit test execution

Like unit tests in a pure Java project, the equivalent test task in an Android project ([AndroidUnitTest](#)) is also cacheable since Android Gradle Plugin 3.6.0.

Instrumented test execution (i.e., Espresso tests)

Android instrumented tests ([DeviceProviderInstrumentTestTask](#)), often referred to as "Espresso" tests, are also not cacheable. The Google Android team is also working to make such tests cacheable. Please see [this issue](#).

Lint

Users of Android's [Lint](#) task are well aware of the heavy performance penalty they pay for using it, but also know that it is indispensable for finding common issues in Android projects. Currently, this task is not cacheable. This task is planned to be cacheable with the release of Android Gradle Plugin 3.5. This is another reason to always use the latest version of the Android plugin!

The Fabric Plugin and Crashlytics

The [Fabric](#) plugin, which is used to integrate the Crashlytics crash-reporting tool (among others), is very popular, yet imposes some hefty performance penalties during the build process. This is due to the need for each version of your app to have a unique identifier so that it can be identified in the Crashlytics dashboard. In practice, the default behavior of Crashlytics is to treat "each version" as synonymous with "each build". This defeats [incremental build](#), because each build will be unique. It also breaks the cacheability of certain tasks in the build, and for the same reason. This can be fixed by simply disabling Crashlytics in "debug" builds. You may find instructions for that in the [Crashlytics documentation](#).

NOTE

The fix described in the referenced documentation does not work directly if you are using the Kotlin DSL; see below for the workaround.

Kotlin DSL

The fix described in the referenced documentation does not work directly if you are using the Kotlin DSL; this is due to incompatibilities between that Kotlin DSL and the Fabric plugin. There is a simple workaround for this, based on [this advice](#) from the Kotlin DSL primer.

Create a file, `fabric.gradle`, in the module where you apply the `io.fabric` plugin. This file (known as a script plugin), should have the following contents:

fabric.gradle

```
plugins.withId("com.android.application") { // or "com.android.library"
    android.buildTypes.debug.ext.enableCrashlytics = false
}
```

And then, in the module's `build.gradle.kts` file, apply this script plugin:

build.gradle.kts

```
apply(from = "fabric.gradle")
```

Debugging and diagnosing cache misses

To make the most of task output caching, it is important that any necessary inputs to your tasks are specified correctly, while at the same time avoiding unneeded inputs. Failing to specify an input that affects the task's outputs can result in incorrect builds, while needlessly specifying inputs that do not affect the task's output can cause cache misses.

This chapter is about finding out why a cache miss happened. If you have a cache hit which you didn't expect we suggest to declare whatever change you expected to trigger the cache miss as an input to the task.

Finding problems with task output caching

Below we describe a step-by-step process that should help shake out any problems with caching in your build.

Ensure incremental build works

First, make sure your build does the right thing without the cache. Run a build twice without enabling the Gradle build cache. The expected outcome is that all actionable tasks that produce file outputs are up-to-date. You should see something like this on the command-line:

```
$ ./gradlew clean --quiet ①
$ ./gradlew assemble ②

BUILD SUCCESSFUL
4 actionable tasks: 4 executed
```

```
$ ./gradlew assemble ③
```

```
BUILD SUCCESSFUL
```

```
4 actionable tasks: 4 up-to-date
```

- ① Make sure we start without any leftover results by running **clean** first.
- ② We are assuming your build is represented by running the **assemble** task in these examples, but you can substitute whatever tasks make sense for your build.
- ③ Run the build again without running **clean**.

NOTE

Tasks that have no outputs or no inputs will always be executed, but that shouldn't be a problem.

Use the methods as described below to **diagnose** and **fix** tasks that should be up-to-date but aren't. If you find a task which is out of date, but no cacheable tasks depends on its outcome, then you don't have to do anything about it. The goal is to achieve **stable task inputs** for cacheable tasks.

In-place caching with the local cache

When you are happy with the up-to-date performance then you can repeat the experiment above, but this time with a clean build, and the build cache turned on. The goal with clean builds and the build cache turned on is to retrieve all cacheable tasks from the cache.

WARNING

When running this test make sure that you have no **remote** cache configured, and storing in the **local** cache is enabled. These are the default settings.

This would look something like this on the command-line:

```
$ rm -rf ~/.gradle/caches/build-cache-1 ①
```

```
$ ./gradlew clean --quiet ②
```

```
$ ./gradlew assemble --build-cache ③
```

```
BUILD SUCCESSFUL
```

```
4 actionable tasks: 4 executed
```

```
$ ./gradlew clean --quiet ④
```

```
$ ./gradlew assemble --build-cache ⑤
```

```
BUILD SUCCESSFUL
```

```
4 actionable tasks: 1 executed, 3 from cache
```

- ① We want to start with an empty local cache.
- ② Clean the project to remove any unwanted leftovers from previous builds.
- ③ Build it once to let it populate the cache.
- ④ Clean the project again.

⑤ Build it again: this time everything cacheable should load from the just populated cache.

You should see all cacheable tasks loaded from cache, while non-cacheable tasks should be executed.

The screenshot shows the Build Scan Performance tab for a build named 'origin-local-cache build' on Dec 12, 2017. The left sidebar contains navigation links: Summary, Console log, Timeline, Performance (selected), Projects, Dependencies, Plugins, Custom values, Switches, and Infrastructure. The main content area displays a table of task execution metrics.

Task	Count	Percentage	Time
Wall clock time spent executing tasks			0.111s
All tasks	11		0.110s
Tasks avoided	3 (75.0%)		0.103s
From cache	3 (75.0%)		0.103s
Up-to-date	0 (00.0%)		0.000s
Tasks executed	1 (25.0%)		0.004s
Cacheable	0 (00.0%)		0.000s
Not cacheable	1 (25.0%)		0.004s
Lifecycle	5		0.002s
No source	2		0.001s
Skipped	0		0.000s
≡ 11 tasks in 1 project			
:compileTestJava FROM-CACHE			0.080s
:test FROM-CACHE			0.014s
:compileJava FROM-CACHE			0.009s
:jar			0.004s
:check UP-TO-DATE			0.001s
:processResources NO-SOURCE			0.001s
:testClasses UP-TO-DATE			0.001s
:assemble			0.000s
:build			0.000s
:classes UP-TO-DATE			0.000s
:processTestResources NO-SOURCE			0.000s

At the bottom, there is a breadcrumb trail: Home > Performance > Task execution, and a link: Close performance details (esc).

Again, use the below methods to [diagnose](#) and [fix](#) cacheability issues.

Testing cache relocatability

Once everything loads properly while building the same checkout with the local cache enabled, it's time to see if there are any *relocation problems*. A task is considered *relocatable* if its output can be reused when the task is executed in a different location. (More on this in [path sensitivity and relocatability](#).)

NOTE

Tasks that should be relocatable but aren't are usually a result of absolute paths being present among the task's inputs.

To discover these problems, first check out the same commit of your project in two different directories on your machine. For the following example let's assume we have a checkout in `~/checkout-1` and `~/checkout-2`.

WARNING

Like with the previous test, you should have no **remote** cache configured, and storing in the **local** cache should be enabled.

```
$ rm -rf ~/.gradle/caches/build-cache-1 ①
$ cd ~/checkout-1 ②
$ ./gradlew clean --quiet ③
$ ./gradlew assemble --build-cache ④
```

BUILD SUCCESSFUL

4 actionable tasks: 4 executed

```
$ cd ~/checkout-2 ⑤
```

```
$ ./gradlew clean --quiet ⑥
```

```
$ ./gradlew clean assemble --build-cache ⑦
```

BUILD SUCCESSFUL

4 actionable tasks: 1 executed, 3 from cache

- ① Remove all entries in the local cache first.
- ② Go to the first checkout directory.
- ③ Clean the project to remove any unwanted leftovers from previous builds.
- ④ Run a build to populate the cache.
- ⑤ Go to the other checkout directory.
- ⑥ Clean the project again.
- ⑦ Run a build again.

You should see the exact same results as you saw with the previous [in place caching test](#) step.

Cross-platform tests

If your build passes the [relocation test](#), it is in good shape already. If your build requires support for multiple platforms, it is best to see if the required tasks get reused between platforms, too. A typical example of cross-platform builds is when CI runs on Linux VMs, while developers use macOS or Windows, or a different variety or version of Linux.

To test cross-platform cache reuse, set up a [remote](#) cache (see [share results between CI builds](#)) and populate it from one platform and consume it from the other.

Incremental cache usage

After these experiments with fully cached builds, you can go on and try to make typical changes to your project and see if enough tasks are still cached. If the results are not satisfactory, you can think about restructuring your project to reduce dependencies between different tasks.

Evaluating cache performance over time

Consider recording execution times of your builds, generating graphs, and analyzing the results. Keep an eye out for certain patterns, like a build recompiling everything even though you expected compilation to be cached.

You can also make changes to your code base manually or automatically and check that the expected set of tasks is cached.

If you have tasks that are re-executing instead of loading their outputs from the cache, then it may point to a problem in your build. Techniques for debugging a cache miss are explained in the following section.

Helpful data for diagnosing a cache miss

A cache miss happens when Gradle calculates a build cache key for a task which is different from any existing build cache key in the cache. Only comparing the build cache key on its own does not give much information, so we need to look at some finer grained data to be able to diagnose the cache miss. A list of all inputs to the computed build cache key can be found in the [section on cacheable tasks](#).

From most coarse grained to most fine grained, the items we will use to compare two tasks are:

- Build cache keys
- Task and Task action implementations
 - classloader hash
 - class name
- Task output property names
- Individual task property input hashes
- Hashes of files which are part of task input properties

If you want information about the build cache key and individual input property hashes, use `-Dorg.gradle.caching.debug=true`:

```
$ ./gradlew :compileJava --build-cache -Dorg.gradle.caching.debug=true

.
.
.
Appending implementation to build cache key:
org.gradle.api.tasks.compile.JavaCompile_Decorated@470c67ec713775576db4e818e7a4c75d
Appending additional implementation to build cache key:
org.gradle.api.tasks.compile.JavaCompile_Decorated@470c67ec713775576db4e818e7a4c75d
Appending input value fingerprint for 'options' to build cache key:
e4eae32137a6a587e57eea660d7f85d
Appending input value fingerprint for 'options.compilerArgs' to build cache key:
8222d82255460164427051d7537fa305
Appending input value fingerprint for 'options.debug' to build cache key:
f6d7ed39fe24031e22d54f3fe65b901c
Appending input value fingerprint for 'options.debugOptions' to build cache key:
a91a8430ae47b11a17f6318b53f5ce9c
Appending input value fingerprint for 'options.debugOptions.debugLevel' to build cache
key: f6bd6b3389b872033d462029172c8612
Appending input value fingerprint for 'options.encoding' to build cache key:
f6bd6b3389b872033d462029172c8612
.
.
.
Appending input file fingerprints for 'options.sourcepath' to build cache key:
5fd1e7396e8de4cb5c23dc6aadd7787a - RELATIVE_PATH{EMPTY}
Appending input file fingerprints for 'stableSources' to build cache key:
```

```
f305ada95aeae858c233f46fc1ec4d01 - RELATIVE_PATH{.../src/main/java=IGNORED / DIR,
.../src/main/java/Hello.java='Hello.java' / 9c306ba203d618dfbe1be83354ec211d}
Appending output property name to build cache key: destinationDir
Appending output property name to build cache key:
options.annotationProcessorGeneratedSourcesDirectory
Build cache key for task ':compileJava' is 8ebf682168823f662b9be34d27afdf77
```

The log shows e.g. which source files constitute the **stableSources** for the **compileJava** task. To find the actual differences between two builds you need to resort to matching up and comparing those hashes yourself.

TIP **Develocity** already takes care of this for you; it lets you quickly diagnose a cache miss with the Build Scan™ Comparison tool.

Diagnosing the reasons for a cache miss

Having the data from the last section at hand, you should be able to diagnose why the outputs of a certain task were not found in the build cache. Since you were expecting more tasks to be cached, you should be able to pinpoint a build which would have produced the artifact under question.

Before diving into how to find out why one task has not been loaded from the cache we should first look into which task caused the cache misses. There is a cascade effect which causes dependent tasks to be executed if one of the tasks earlier in the build is not loaded from the cache and has different outputs. Therefore, you should locate the first cacheable task which was executed and continue investigating from there. This can be done from the timeline view in a Build Scan™:

Build Scan

✓ gradle clean quickTest1 Jun 9, 2017 12:22:33 PM BST

Task path: [Task path] SUCCESS

Task type: [Cacheable]

Found 11 tasks executed in 6 projects totaling 15m 18.183s

Path	Started after	Duration	Cacheable	Class
:dependencyManagement:compileTestFixturesGroovy	24.984s	3.930s	Cacheable	org.gradle.api.tasks.compile.GroovyCompile
:launcher:compileTestFixturesGroovy	29.197s	0.492s	Cacheable	org.gradle.api.tasks.compile.GroovyCompile
:languageJvm:compileTestFixturesGroovy	29.694s	1.808s	Cacheable	org.gradle.api.tasks.compile.GroovyCompile
:platformBase:compileTestFixturesGroovy	31.612s	0.977s	Cacheable	org.gradle.api.tasks.compile.GroovyCompile
:platformPlay:compileTestFixturesGroovy	32.595s	0.945s	Cacheable	org.gradle.api.tasks.compile.GroovyCompile
:platformPlay:compileTestGroovy	33.678s	2.050s	Cacheable	org.gradle.api.tasks.compile.GroovyCompile
:platformPlay:test	35.735s	10.515s	Cacheable	org.gradle.api.tasks.testing.Test
:languageScala:compileTestFixturesGroovy	59.951s	0.712s	Cacheable	org.gradle.api.tasks.compile.GroovyCompile
:platformPlay:compileIntegTestGroovy	1m 0.675s	1.774s	Cacheable	org.gradle.api.tasks.compile.GroovyCompile
:platformPlay:integTestPrepare	1m 8.201s	2m 38.781s	Cacheable	org.gradle.testing.IntegrationTest
:platformPlay:integTest	3m 46.983s	12m 16.199s	Cacheable	org.gradle.testing.IntegrationTest

Order: Execution

Home > Timeline

Close timeline (esc)

At first, you should check if the implementation of the task changed. This would mean checking the class names and classloader hashes for the task class itself and for each of its actions. If there is a change, this means that the build script, **buildSrc** or the Gradle version has changed.

NOTE

A change in the output of `buildSrc` also marks all the logic added by your build as changed. Especially, custom actions added to cacheable tasks will be marked as changed. This can be problematic, see [section about `doFirst` and `doLast`](#).

If the implementation is the same, then you need to start comparing inputs between the two builds. There should be at least one different input hash. If it is a simple value property, then the configuration of the task changed. This can happen for example by

- changing the build script,
- conditionally configuring the task differently for CI or the developer builds,
- depending on a system property or an environment variable for the task configuration,
- or having an absolute path which is part of the input.

If the changed property is a file property, then the reasons can be the same as for the change of a value property. Most probably though a file on the filesystem changed in a way that Gradle detects a difference for this input. The most common case will be that the source code was changed by a check in. It is also possible that a file generated by a task changed, e.g. since it includes a timestamp. As described in [Java version tracking](#), the Java version can also influence the output of the Java compiler. If you did not expect the file to be an input to the task, then it is possible that you should alter the configuration of the task to not include it. For example, having your integration test configuration including all the unit test classes as a dependency has the effect that all integration tests are re-executed when a unit test changes. Another option is that the task tracks absolute paths instead of relative paths and the location of the project directory changed on disk.

Example

We will walk you through the process of diagnosing a cache miss. Let's say we have build **A** and build **B** and we expected all the test tasks for a sub-project `sub1` to be cached in build **B** since only a unit test for another sub-project `sub2` changed. Instead, all the tests for the sub-project have been executed. Since we have the cascading effect when we have cache misses, we need to find the task which caused the caching chain to fail. This can easily be done by filtering for all cacheable tasks which have been executed and then select the first one. In our case, it turns out that the tests for the sub-project `internal-testing` were executed even though there was no code change to this project. This means that the property `classpath` changed and some file on the runtime classpath actually did change. Looking deeper into this, we actually see that the inputs for the task `processResources` changed in that project, too. Finally, we find this in our build file:

build.gradle.kts

```
val currentVersionInfo =
tasks.register<CurrentVersionInfo>("currentVersionInfo") {
    version = project.version as String
    versionInfoFile = layout.buildDirectory.file("generated-
resources/currentVersion.properties")
}
```

```

sourceSets.main.get().output.dir(currentVersionInfo.map {
it.versionInfoFile.get().asFile.parentFile })

abstract class CurrentVersionInfo : DefaultTask() {
    @get:Input
    abstract val version: Property<String>

    @get:OutputFile
    abstract val versionInfoFile: RegularFileProperty

    @TaskAction
    fun writeVersionInfo() {
        val properties = Properties()
        properties.setProperty("latestMilestone", version.get())
        versionInfoFile.get().asFile.outputStream().use { out ->
            properties.store(out, null)
        }
    }
}

```

build.gradle

```

def currentVersionInfo = tasks.register('currentVersionInfo',
CurrentVersionInfo) {
    version = project.version
    versionInfoFile = layout.buildDirectory.file('generated-
resources/currentVersion.properties')
}

sourceSets.main.output.dir(currentVersionInfo.map { it.versionInfoFile.get()
.asFile.parentFile })

abstract class CurrentVersionInfo extends DefaultTask {
    @Input
    abstract Property<String> getVersion()

    @OutputFile
    abstract RegularFileProperty getVersionInfoFile()

    @TaskAction
    void writeVersionInfo() {
        def properties = new Properties()
        properties.setProperty('latestMilestone', version.get())
        versionInfoFile.get().asFile.withOutputStream { out ->
            properties.store(out, null)
        }
    }
}

```

Since properties files stored by Java's `Properties.store` method contain a timestamp, this will cause a change to the runtime classpath every time the build runs. In order to solve this problem see [non-repeatable task outputs](#) or use [input normalization](#).

NOTE

The compile classpath is not affected since compile avoidance ignores non-class files on the classpath.

Solving common problems

Small problems in a build, like forgetting to declare a configuration file as an input to your task, can be easily overlooked. The configuration file might change infrequently, or only change when some other (correctly tracked) input changes as well. The worst that could happen is that your task doesn't execute when it should. Developers can always re-run the build with `clean`, and "fix" their builds for the price of a slow rebuild. In the end nobody gets blocked in their work, and the incident is chalked up to "Gradle acting up again."

With cacheable tasks incorrect results are stored permanently, and can come back to haunt you later; re-running with `clean` won't help in this situation either. When using a shared cache, these problems even cross machine boundaries. In the example above, Gradle might end up loading a result for your task that was produced with a different configuration. Resolving these problems with the build therefore becomes even more important when task output caching is enabled.

Other issues with the build won't cause it to produce incorrect results, but will lead to unnecessary cache misses. In this chapter you will learn about some typical problems and ways to avoid them. Fixing these issues will have the added benefit that your build will stop "acting up," and developers can forget about running builds with `clean` altogether.

System file encoding

Most Java tools use the system file encoding when no specific encoding is specified. This means that running the same build on machines with different file encoding can yield different outputs. Currently Gradle only tracks on a per-task basis that no file encoding has been specified, but it does not track the system encoding of the JVM in use. This can cause incorrect builds. You should always set the file system encoding to avoid these kind of problems.

NOTE

Build scripts are compiled with the file encoding of the Gradle daemon. By default, the daemon uses the system file encoding, too.

Setting the file encoding for the Gradle daemon mitigates both above problems by making sure that the encoding is the same across builds. You can do so in your `gradle.properties`:

gradle.properties

```
org.gradle.jvmargs=-Dfile.encoding=UTF-8
```

Environment variable tracking

Gradle does not track changes in environment variables for tasks. For example for **Test** tasks it is completely possible that the outcome depends on a few environment variables. To ensure that only the right artifacts are re-used between builds, you need to add environment variables as inputs to tasks depending on them.

Absolute paths are often passed as environment variables, too. You need to pay attention what you add as an input to the task in this case. You would need to ensure that the absolute path is the same between machines. Most times it makes sense to track the file or the contents of the directory the absolute path points to. If the absolute path represents a tool being used it probably makes sense to track the tool version as an input instead.

For example, if you are using tools in your **Test** task called **integTest** which depend on the contents of the **LANG** variable you should do this:

build.gradle.kts

```
tasks.integTest {
    inputs.property("langEnvironment") {
        System.getenv("LANG")
    }
}
```

build.gradle

```
tasks.named('integTest') {
    inputs.property("langEnvironment") {
        System.getenv("LANG")
    }
}
```

If you add conditional logic to distinguish CI builds from local development builds, you have to ensure that this does not break the loading of task outputs from CI onto developer machines. For example, the following setup would break caching of **Test** tasks, since Gradle always detects the differences in custom task actions.

build.gradle.kts

```
if ("CI" in System.getenv()) {
    tasks.withType<Test>().configureEach {
        doFirst {
            println("Running test on CI")
        }
    }
}
```

```
}  
}  
}
```

build.gradle

```
if (System.getenv().containsKey("CI")) {  
    tasks.withType(Test).configureEach {  
        doFirst {  
            println "Running test on CI"  
        }  
    }  
}
```

You should always add the action unconditionally:

build.gradle.kts

```
tasks.withType<Test>().configureEach {  
    doFirst {  
        if ("CI" in System.getenv()) {  
            println("Running test on CI")  
        }  
    }  
}
```

build.gradle

```
tasks.withType(Test).configureEach {  
    doFirst {  
        if (System.getenv().containsKey("CI")) {  
            println "Running test on CI"  
        }  
    }  
}
```

This way, the task has the same custom action on CI and on developer builds and its outputs can be re-used if the remaining inputs are the same.

Line endings

If you are building on different operating systems be aware that some version control systems convert line endings on check-out. For example, Git on Windows uses `autocrlf=true` by default which converts all line endings to `\r\n`. As a consequence, compilation outputs can't be re-used on Windows since the input sources are different. If sharing the build cache across multiple operating systems is important in your environment, then setting `autocrlf=false` across your build machines is crucial for optimal build cache usage.

Symbolic links

When using symbolic links, Gradle does not store the link in the build cache but the actual file contents of the destination of the link. As a consequence you might have a hard time when trying to reuse outputs which heavily use symbolic links. There currently is no workaround for this behavior.

For operating systems supporting symbolic links, the content of the destination of the symbolic link will be added as an input. If the operating system does not support symbolic links, the actual symbolic link file is added as an input. Therefore, tasks which have symbolic links as input files, e.g. `Test` tasks having symbolic link as part of its runtime classpath, will not be cached between Windows and Linux. If caching between operating systems is desired, symbolic links should not be checked into version control.

Java version tracking

Gradle tracks only the major version of Java as an input for compilation and test execution. Currently, it does *not* track the vendor nor the minor version. Still, the vendor and the minor version may influence the bytecode produced by compilation.

NOTE

If you're using [Java Toolchains](#), the Java major version, the vendor (if specified) and implementation (if specified) will be tracked automatically as an input for compilation and test execution.

If you use different JVM vendors for compiling or running Java we strongly suggest that you add the vendor as an input to the corresponding tasks. This can be achieved by using the [runtime API](#) as shown in the following snippet.

build.gradle.kts

```
tasks.withType<AbstractCompile>().configureEach {
    inputs.property("java.vendor") {
        System.getProperty("java.vendor")
    }
}

tasks.withType<Test>().configureEach {
    inputs.property("java.vendor") {
        System.getProperty("java.vendor")
    }
}
```

```
}  
}
```

build.gradle

```
tasks.withType(AbstractCompile).configureEach {  
    inputs.property("java.vendor") {  
        System.getProperty("java.vendor")  
    }  
}  
  
tasks.withType(Test).configureEach {  
    inputs.property("java.vendor") {  
        System.getProperty("java.vendor")  
    }  
}
```

With respect to tracking the Java minor version there are different competing aspects: developers having cache hits and "perfect" results on CI. There are basically two situations when you may want to track the minor version of Java: for compilation and for runtime. In the case of compilation, there can sometimes be differences in the produced bytecode for different minor versions. However, the bytecode should still result in the same runtime behavior.

NOTE | [Java compile avoidance](#) will treat this bytecode the same since it extracts the ABI.

Treating the minor number as an input can decrease the likelihood of a cache hit for developer builds. Depending on how standard development environments are across your team, it's common for many different Java minor version to be in use.

Even without tracking the Java minor version you may have cache misses for developers due to some locally compiled class files which constitute an input to test execution. If these outputs made it into the local build cache on this developers machine even a clean will not solve the situation. Therefore, the choice for tracking the Java minor version is between sometimes or never re-using outputs between different Java minor versions for test execution.

NOTE | The compiler infrastructure provided by the JVM used to run Gradle is also used by the Groovy compiler. Therefore, you can expect differences in the bytecode of compiled Groovy classes for the same reasons as above and the same suggestions apply.

Avoid changing inputs external to your build

If your build is dependent on external dependencies like binary artifacts or dynamic data from a web page you need to make sure that these inputs are consistent throughout your infrastructure. Any variations across machines will result in cache misses.

Never re-release a non-changing binary dependency with the same version number but different contents: if this happens with a plugin dependency, you will never be able to explain why you don't see cache reuse between machines (it's because they have different versions of that artifact).

Using **SNAPSHOTs** or other changing dependencies in your build by design violates the [stable task inputs](#) principle. To use the build cache effectively, you should depend on fixed dependencies. You may want to look into [dependency locking](#) or switch to using [composite builds](#) instead.

The same is true for depending on volatile external resources, for example a list of released versions. One way of locking the changes would be to check the volatile resource into source control whenever it changes so that the builds only depend on the state in source control and not on the volatile resource itself.

Suggestions for authoring your build

Review usages of **doFirst** and **doLast**

Using **doFirst** and **doLast** from a build script on a cacheable task ties you to build script changes since the implementation of the closure comes from the build script. If possible, you should use separate tasks instead.

Modifying input or output properties via the runtime API in **doFirst** is discouraged since these changes will not be detected for up-to-date checks and the build cache. Even worse, when the task does not execute, then the configuration of the task is actually different from when it executes. Instead of using **doFirst** for modifying the inputs consider using a separate task to configure the task under question - a so called configure task. E.g., instead of doing

build.gradle.kts

```
tasks.jar {
    val runtimeClasspath: FileCollection =
        configurations.runtimeClasspath.get()
    doFirst {
        manifest {
            val classPath = runtimeClasspath.map { it.name }.joinToString("
")
            attributes("Class-Path" to classPath)
        }
    }
}
```

build.gradle

```
tasks.named('jar') {
    FileCollection runtimeClasspath = configurations.runtimeClasspath
    doFirst {
```

```

        manifest {
            def classPath = runtimeClasspath.collect { it.name }.join(" ")
            attributes('Class-Path': classPath)
        }
    }
}

```

do

build.gradle.kts

```

val configureJar = tasks.register("configureJar") {
    doLast {
        tasks.jar.get().manifest {
            val classPath = configurations.runtimeClasspath.get().map {
it.name }.joinToString(" ")
            attributes("Class-Path" to classPath)
        }
    }
}
tasks.jar { dependsOn(configureJar) }

```

build.gradle

```

def configureJar = tasks.register('configureJar') {
    doLast {
        tasks.jar.manifest {
            def classPath = configurations.runtimeClasspath.collect { it.name
}.join(" ")
            attributes('Class-Path': classPath)
        }
    }
}

tasks.named('jar') { dependsOn(configureJar) }

```

WARNING

Note that configuring a task from other task is not supported when using the [configuration cache](#).

Build logic based on the outcome of a task

Do not base build logic on whether a task has been *executed*. In particular you should not assume

that the output of a task can only change if it actually executed. Actually, loading the outputs from the build cache would also change them. Instead of relying on custom logic to deal with changes to input or output files you should leverage Gradle's built-in support by declaring the correct inputs and outputs for your tasks and leave it to Gradle to decide if the task actions should be executed. For the very same reason using `outputs.upToDateWhen` is discouraged and should be replaced by properly declaring the task's inputs.

Overlapping outputs

You already saw that [overlapping outputs are a problem for task output caching](#). When you add new tasks to your build or re-configure built-in tasks make sure you do not create overlapping outputs for cacheable tasks. If you must you can add a `Sync` task which then would sync the merged outputs into the target directory while the original tasks remain cacheable.

Develocity will show tasks where caching was disabled for overlapping outputs in the timeline and in the task input comparison:

Comparing with differences		
:customTask		
Task class	CustomTask	CustomTask
Task doLast actions	Build script closure	
Input changes	inputFile	
Resulting cache key	02ffb69ec9bb694579d59aa62653e178	b7d10e3f7ed1e01cd3f452c45861ef5c
Resulting outcome	SUCCESS	FROM-CACHE

Achieving stable task inputs

It is crucial to have [stable task inputs](#) for every cacheable task. In the following section you will learn about different situations which violate stable task inputs and look at possible solutions.

Volatile task inputs

If you use a volatile input like a timestamp as an input property for a task, then there is nothing Gradle can do to make the task cacheable. You should really think hard if the volatile data is really essential to the output or if it is only there for e.g. auditing purposes.

If the volatile input is essential to the output then you can try to make the task using the volatile input cheaper to execute. You can do this by splitting the task into two tasks - the first task doing the

expensive work which is cacheable and the second task adding the volatile data to the output. In this way the output stays the same and the build cache can be used to avoid doing the expensive work. For example, for building a jar file the expensive part - Java compilation - is already a different task while the jar task itself, which is not cacheable, is cheap.

If it is not an essential part of the output, then you should not declare it as an input. As long as the volatile input does not influence the output then there is nothing else to do. Most times though, the input will be part of the output.

Non-repeatable task outputs

Having tasks which generate different outputs for the same inputs can pose a challenge for the effective use of task output caching as seen in [repeatable task outputs](#). If the non-repeatable task output is not used by any other task then the effect is very limited. It basically means that loading the task from the cache might produce a different result than executing the same task locally. If the only difference between the outputs is a timestamp, then you can either accept the effect of the build cache or decide that the task is not cacheable after all.

Non-repeatable task outputs lead to non-stable task inputs as soon as another task depends on the non-repeatable output. For example, re-creating a jar file from the files with the same contents but different modification times yields a different jar file. Any other task depending on this jar file as an input file cannot be loaded from the cache when the jar file is rebuilt locally. This can lead to hard-to-diagnose cache misses when the consuming build is not a clean build or when a cacheable task depends on the output of a non-cacheable task. For example, when doing incremental builds it is possible that the artifact on disk which is considered up-to-date and the artifact in the build cache are different even though they are essentially the same. A task depending on this task output would then not be able to load outputs from the build cache since the inputs are not exactly the same.

As described in the [stable task inputs section](#), you can either make the task outputs repeatable or use input normalization. You already learned about the possibilities with [configurable input normalization](#).

Gradle includes some support for creating repeatable output for archive tasks. For tar and zip files Gradle can be configured to create [reproducible archives](#). This is done by configuring e.g. the `Zip` task via the following snippet.

build.gradle.kts

```
tasks.register<Zip>("createZip") {
    isPreserveFileTimestamps = false
    isReproducibleFileOrder = true
    // ...
}
```

build.gradle

```
tasks.register('createZip', Zip) {  
    preserveFileTimestamps = false  
    reproducibleFileOrder = true  
    // ...  
}
```

Another way to make the outputs repeatable is to activate caching for a task with non-repeatable outputs. If you can make sure that the same build cache is used for all builds then the task will always have the same outputs for the same inputs by design of the build cache. Going down this road can lead to different problems with cache misses for incremental builds as described above. Moreover, race conditions between different builds trying to store the same outputs in the build cache in parallel can lead to hard-to-diagnose cache misses. If possible, you should avoid going down that route.

Limit the effect of volatile data

If none of the described solutions for dealing with volatile data work for you, you should still be able to limit the effect of volatile data on effective use of the build cache. This can be done by adding the volatile data later to the outputs as described in the [volatile task inputs section](#). Another option would be to move the volatile data so it affects fewer tasks. For example moving the dependency from the `compile` to the `runtime` configuration may already have quite an impact.

Sometimes it is also possible to build two artifacts, one containing the volatile data and another one containing a constant representation of the volatile data. The non-volatile output would be used e.g. for testing while the volatile one would be published to an external repository. While this conflicts with the Continuous Delivery "build artifacts once" principle it can sometimes be the only option.

Custom and third party tasks

If your build contains custom or third party tasks, you should take special care that these don't influence the effectiveness of the build cache. Special care should also be taken for code generation tasks which may not have [repeatable task outputs](#). This can happen if the code generator includes e.g. a timestamp in the generated files or depends on the order of the input files. Other pitfalls can be the use of `HashMaps` or other data structures without order guarantees in the task's code.

WARNING

Some third party plugins can even influence cacheability of Gradle's built-in tasks. This can happen if they add inputs like absolute paths or volatile data to tasks via the runtime API. In the worst case this can lead to incorrect builds when the plugins try to depend on the [outcome of a task](#) and do not take `FROM-CACHE` into account.

[1] The `CommandLineArgumentProvider` in this example is implemented as a [managed type](#).

DEPENDENCY MANAGEMENT

CORE CONCEPTS

1. Declaring dependencies

Declaring dependencies in Gradle involves specifying libraries or files that your project depends on.

Understanding producers and consumers

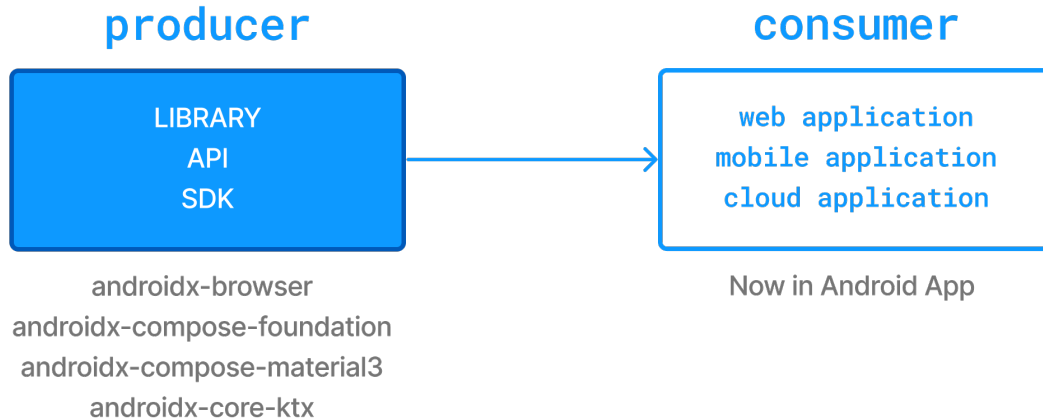
In dependency management, it is essential to understand the distinction between *producers* and *consumers*.

When you build a library, you are acting as a *producer*, creating artifacts that will be consumed by others, the *consumers*.

When you depend on that library, you are acting as a *consumer*. *Consumers* can be broadly defined as:

- Projects that depend on other projects.
- Configurations that declare dependencies on specific artifacts.

The decisions we make in dependency management often depend on the type of project we are building, specifically, what kind of *consumer* we are.



Adding a dependency

To add a dependency in Gradle, you use the `dependencies{}` block in your build script.

The `dependencies` block allows you to specify various types of dependencies such as external libraries, local JAR files, or other projects within a multi-project build.

External dependencies in Gradle are declared using a configuration name (e.g., `implementation`, `compileOnly`, `testImplementation`) followed by the dependency notation, which includes the group ID (group), artifact ID (name), and version.

build.gradle

```
dependencies {  
    // Configuration Name + Dependency Notation - GroupID : ArtifactID (Name) :  
    Version  
    configuration('<group>:<name>:<version>')  
}
```

Note:

1. Gradle automatically includes *transitive dependencies*, which are dependencies of your dependencies.
2. Gradle offers several *configuration* options for dependencies, which define the scope in which dependencies are used, such as compile-time, runtime, or test-specific scenarios.
3. You can specify the *repositories* where Gradle should look for dependencies in your build file.

Understanding types of dependencies

There are three kinds of dependencies, module dependencies, project dependencies, and file dependencies.

1. Module dependencies

Module dependencies are the most common dependencies. They refer to a module in a repository:

build.gradle.kts

```
dependencies {  
    implementation("org.codehaus.groovy:groovy:3.0.5")  
    implementation("org.codehaus.groovy:groovy-json:3.0.5")  
    implementation("org.codehaus.groovy:groovy-nio:3.0.5")  
}
```

build.gradle

```
dependencies {  
    implementation 'org.codehaus.groovy:groovy:3.0.5'  
    implementation 'org.codehaus.groovy:groovy-json:3.0.5'  
    implementation 'org.codehaus.groovy:groovy-nio:3.0.5'  
}
```

2. Project dependencies

Project dependencies allow you to declare dependencies on other projects within the same build. This is useful in multi-project builds where multiple projects are part of the same Gradle build.

Project dependencies are declared by referencing the project path:

build.gradle.kts

```
dependencies {  
    implementation(project(":utils"))  
    implementation(project(":api"))  
}
```

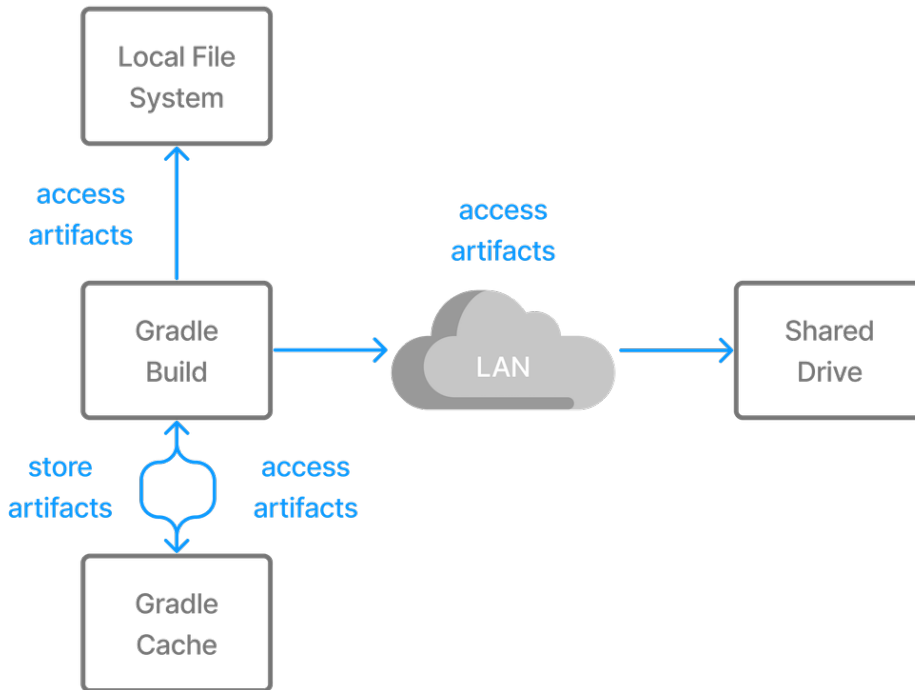
build.gradle

```
dependencies {  
    implementation project(':utils')  
    implementation project(':api')  
}
```

3. File dependencies

In some projects, you might not rely on binary repository products like [JFrog Artifactory](#) or [Sonatype Nexus](#) for hosting and resolving external dependencies. Instead, you might host these dependencies on a shared drive or to check them into version control alongside the project source code.

These are known as file dependencies because they represent files without any [metadata](#) (such as information about transitive dependencies, origin, or author) attached to them.



To add files as dependencies for a configuration, you simply pass a [file collection](#) as a dependency:

build.gradle.kts

```
dependencies {
    runtimeOnly(files("libs/a.jar", "libs/b.jar"))
    runtimeOnly(fileTree("libs") { include("*.jar") })
}
```

build.gradle

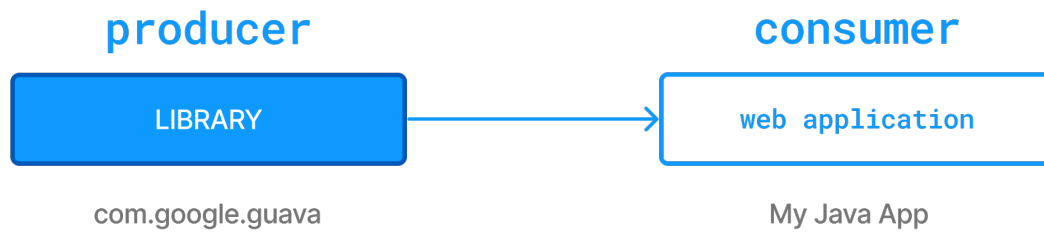
```
dependencies {
    runtimeOnly files('libs/a.jar', 'libs/b.jar')
    runtimeOnly fileTree('libs') { include '*.jar' }
}
```

WARNING

It is recommended to use project dependencies or external dependencies over file dependencies.

Looking at an example

Let's imagine an example for a Java application which uses [Guava](#), a set of core Java libraries from Google:



The Java app contains the following Java class:

InitializeCollection.java

```
package org.example;

import com.google.common.collect.ImmutableMap; // Comes from the Guava
library

public class InitializeCollection {
    public static void main(String[] args) {
        ImmutableMap<String, Integer> immutableMap
            = ImmutableMap.of("coin", 3, "glass", 4, "pencil", 1);
    }
}
```

To add the [Guava](#) library to your Gradle project as a dependency, you must add the following line to your build file:

build.gradle.kts

```
dependencies {
    implementation("com.google.guava:guava:23.0")
}
```

build.gradle

```
dependencies {
    implementation 'com.google.guava:guava:23.0'
}
```

Where:

- **implementation** is the configuration.

- `com.google.guava:guava:23.0` specifies the group, name, and version of the library:
 - `com.google.guava` is the group ID.
 - `guava` is the artifact ID (i.e., name).
 - `23.0` is the version.

Take a quick look at the [Guava page in Maven Central](#) as a reference.

Listing project dependencies

The `dependencies` task provides an overview of the dependencies of your project. It helps you understand what dependencies are being used, how they are resolved, and their relationships, including any transitive dependencies by rendering a dependency tree from the command line.

This task can be particularly useful for debugging dependency issues, such as version conflicts or missing dependencies.

For example, let's say our `app` project contains the follow lines in its build script:

build.gradle.kts

```
dependencies {  
    implementation("com.google.guava:guava:30.0-jre")  
    runtimeOnly("org.apache.commons:commons-lang3:3.14.0")  
}
```

build.gradle

```
dependencies {  
    implementation("com.google.guava:guava:30.0-jre")  
    runtimeOnly("org.apache.commons:commons-lang3:3.14.0")  
}
```

Running the `dependencies` task on the `app` project yields the following:

```
$ ./gradlew app:dependencies
```

```
> Task :app:dependencies
```

```
-----  
Project ':app'
```

```
implementation - Implementation dependencies for the 'main' feature. (n)
\--- com.google.guava:guava:30.0-jre (n)

runtimeClasspath - Runtime classpath of source set 'main'.
+--- com.google.guava:guava:30.0-jre
|    +--- com.google.guava:failureaccess:1.0.1
|    +--- com.google.guava:listenablefuture:9999.0-empty-to-avoid-conflict-with-guava
|    +--- com.google.code.findbugs:jsr305:3.0.2
|    +--- org.checkerframework:checker-qual:3.5.0
|    +--- com.google.errorprone:error_prone_annotations:2.3.4
|    \--- com.google.j2objc:j2objc-annotations:1.3
\--- org.apache.commons:commons-lang3:3.14.0

runtimeOnly - Runtime-only dependencies for the 'main' feature. (n)
\--- org.apache.commons:commons-lang3:3.14.0 (n)
```

We can clearly see that for the `implementation` configuration, the `com.google.guava:guava:30.0-jre` dependency has been added. As for the `runtimeOnly` configuration, the `org.apache.commons:commons-lang3:3.14.0` dependency has been added.

We also see a list of transitive dependencies for `com.google.guava:guava:30.0-jre` (which are the dependencies for the `guava` library), such as `com.google.guava:failureaccess:1.0.1` in the `runtimeClasspath` configuration.

Next Step: [Learn about Dependency Configurations](#) >>

2. Dependency Configurations

Every dependency declared for a Gradle project applies to a specific scope.

For example, some dependencies should be used for compiling source code whereas others only need to be available at runtime:

build.gradle.kts

```
dependencies {
    implementation("com.google.guava:guava:30.0-jre") // Needed to compile
    and run the app
    runtimeOnly("org.slf4j:slf4j-simple:2.0.13")      // Only needed at
    runtime
}
```

build.gradle

```
dependencies {  
    implementation("com.google.guava:guava:30.0-jre") // Needed to compile  
    and run the app  
    runtimeOnly("org.slf4j:slf4j-simple:2.0.13") // Only needed at  
    runtime  
}
```

Dependency configurations are a way to define different sets of dependencies for different purposes within a project. They determine how and when dependencies are used in various stages of the build process.

Configurations are a fundamental part of dependency resolution in Gradle.

Understanding dependency configurations

Gradle represents the scope of a dependency with the help of a [Configuration](#). Every configuration can be identified by a unique name.

Many Gradle plugins add pre-defined configurations to your project.

The Java Library plugin is used to define a project that produces a Java library. The plugin adds many dependency configurations. These configurations represent the various classpaths needed for source code compilation, executing tests, and more:

Configuration Name	Description	Used to:
<code>api</code>	Dependencies required for both compilation and runtime, and included in the published API.	Declare Dependencies
<code>implementation</code>	Dependencies required for both compilation and runtime.	Declare Dependencies
<code>compileOnly</code>	Dependencies needed only for compilation, not included in runtime or publication.	Declare Dependencies
<code>compileOnlyApi</code>	Dependencies needed only for compilation, but included in the published API.	Declare Dependencies
<code>runtimeOnly</code>	Dependencies needed only at runtime, not included in the compile classpath.	Declare Dependencies
<code>testImplementation</code>	Dependencies required for compiling and running tests.	Declare Dependencies

Configuration Name	Description	Used to:
<code>testCompileOnly</code>	Dependencies needed only for test compilation.	Declare Dependencies
<code>testRuntimeOnly</code>	Dependencies needed only for running tests.	Declare Dependencies

Dependency declaration Configurations

The **dependency declaration configurations** (`compileOnly`, `implementation`, `runtimeOnly`) focus on declaring and managing dependencies based on their usage (compile time, runtime, API exposure):

```
dependencies {
    implementation("com.google.guava:guava:30.1.1-jre")    // Implementation
    dependency
        compileOnly("org.projectlombok:lombok:1.18.20") // Compile-only
    dependency
        runtimeOnly("mysql:mysql-connector-java:8.0.23") // Runtime-only
    dependency
}
```

```
dependencies {
    implementation("com.google.guava:guava:30.1.1-jre") // Implementation
    dependency
        compileOnly("org.projectlombok:lombok:1.18.20") // Compile-only
    dependency
        runtimeOnly("mysql:mysql-connector-java:8.0.23") // Runtime-only
    dependency
}
```

Other Configurations

There are other types of configurations (such as `runtimeClasspath`, `compileClasspath`, `apiElements`, `runtimeElements`), but they are not used to declare dependencies.

It is also possible to create custom configurations. A custom configuration allows you to define a distinct group of dependencies that can be used for specific purposes, such as toolchains or code generation, separate from the standard configurations (e.g., `implementation`, `testImplementation`):

build.gradle.kts

```
val customConfig by configurations.creating

dependencies {
    customConfig("org.example:example-lib:1.0")
}
```

build.gradle

```
configurations {
    customConfig
}

dependencies {
    customConfig("org.example:example-lib:1.0")
}
```

Creating a custom configuration helps manage and isolate dependencies, ensuring they are only included in the relevant classpaths and build processes.

Viewing configurations

The **dependencies** task provides an overview of the dependencies of your project. To focus on the information about one dependency configuration, provide the optional parameter **--configuration**.

The following example show dependencies in the **implementation** dependency configuration of a Java project:

```
$ ./gradlew -q app:dependencies --configuration implementation

-----
Project ':app'
-----

implementation - Implementation only dependencies for source set 'main'.
\--- com.google.guava:guava:30.0-jre
```

Next Step: [Learn about Declaring Repositories >>](#)

3. Declaring repositories

Gradle needs to know where it can download the dependencies used in the project.

For example, the `com.google.guava:guava:30.0-jre` dependency can be downloaded from the public Maven Central repository `mavenCentral()`. Gradle will find and download the `guava` source code (as a `jar`) from Maven Central and use it build the project.

You can add any number of repositories for your dependencies by configuring the `repositories` block in your `build.gradle(.kts)` file:

build.gradle.kts

```
repositories {  
    mavenCentral() ①  
    maven {         ②  
        url = uri("https://company/com/maven2")  
    }  
    mavenLocal()    ③  
    flatDir {       ④  
        dirs("libs")  
    }  
}
```

- ① Public repository
- ② Private/Custom repository
- ③ Local repository
- ④ File location

build.gradle

```
repositories {  
    mavenCentral() ①  
    maven {         ②  
        url = uri("https://company/com/maven2")  
    }  
    mavenLocal()    ③  
    flatDir {       ④  
        dirs "libs"  
    }  
}
```

- ① Public repository
- ② Private/Custom repository
- ③ Local repository
- ④ File location

Gradle can resolve dependencies from one or many repositories based on Maven, Ivy or flat directory formats.

If a library is available from more than one of the listed repositories, Gradle will simply pick the first one.

Declaring a public repository

Organizations building software may want to leverage public binary repositories to download and consume open source dependencies. Popular public repositories include [Maven Central](#) and the [Google Android](#) repository.

Gradle provides built-in shorthand notations for these widely-used repositories:

build.gradle.kts

```
repositories {  
    mavenCentral()  
    google()  
    gradlePluginPortal()  
}
```

build.gradle

```
repositories {  
    mavenCentral()  
    google()  
    gradlePluginPortal()  
}
```

Under the covers Gradle resolves dependencies from the respective URL of the public repository defined by the shorthand notation. All shorthand notations are available via the [RepositoryHandler](#) API.

Declaring a private or custom repository

Most enterprise projects establish a binary repository accessible only within their intranet. In-house repositories allow teams to publish internal binaries, manage users and security, and ensure uptime and availability.

Specifying a custom URL is useful for declaring less popular but publicly-available repositories. Repositories with custom URLs can be specified as Maven or Ivy repositories by calling the corresponding methods available on the [RepositoryHandler](#) API:

build.gradle.kts

```
repositories {  
    maven {  
        url = uri("https://maven-central.storage.apis.com")  
    }  
    ivy {  
        url = uri("https://github.com/ivy-rep/")  
    }  
}
```

build.gradle

```
repositories {  
    maven {  
        url = uri("https://maven-central.storage.apis.com")  
    }  
    ivy {  
        url = uri("https://github.com/ivy-rep/")  
    }  
}
```

Declaring a local repository

Gradle can consume dependencies available in a [local Maven repository](#).

To declare the local Maven cache as a repository, add this to your build script:

build.gradle.kts

```
repositories {  
    mavenLocal()  
}
```

build.gradle

```
repositories {  
    mavenLocal()  
}
```

Understanding supported repository types

Gradle supports a wide range of sources for dependencies, both in terms of format and in terms of connectivity. You may resolve dependencies from:

- Different formats
 - a [Maven compatible](#) artifact repository (e.g: Maven Central)
 - an [Ivy compatible](#) artifact repository (including custom layouts)
 - [local \(flat\) directories](#)
- with different connectivity
 - [authenticated repositories](#)
 - a wide variety of [remote protocols](#) such as HTTPS, SFTP, AWS S3 and Google Cloud Storage based on the presence of artifacts.

Here is a quick snapshot:

build.gradle

```
repositories {  
  
    // Ivy Repository with Custom Layout  
    ivy {  
        url = 'https://your.ivy.repo/url'  
        layout 'pattern', {  
            ivy '[organisation]/[module]/[revision]/[type]s/[artifact]-  
[revision].[ext]'  
            artifact '[organisation]/[module]/[revision]/[type]s/[artifact]-  
[revision].[ext]'  
        }  
    }  
  
    // Authenticated HTTPS Maven Repository  
    maven {  
        url = 'https://your.secure.repo/url'  
        credentials {  
            username = 'your-username'  
            password = 'your-password'  
        }  
    }  
  
    // SFTP Repository  
    maven {  
        url = 'sftp://your.sftp.repo/url'  
        credentials {  
            username = 'your-username'  
            password = 'your-password'  
        }  
    }  
}
```

```
// AWS S3 Repository
maven {
    url = "s3://your-bucket/repository-path"
    credentials(AwsCredentials) {
        accessKey = 'your-access-key'
        secretKey = 'your-secret-key'
    }
}

// Google Cloud Storage Repository
maven {
    url = "gs://your-bucket/repository-path"
}
}
```

Next Step: [Learn about Centralizing Dependencies](#) >>

4. Centralizing dependencies

Central dependencies can be managed in Gradle using various techniques such as platforms and version catalogs. Each approach offers its own advantages and helps in centralizing and managing dependencies efficiently.

Using platforms

A platform is a set of dependency constraints designed to manage the transitive dependencies of a library or application.

When you define a platform in Gradle, you're essentially specifying a set of dependencies that are meant to be used together, ensuring compatibility and simplifying dependency management:

platform/build.gradle.kts

```
plugins {
    id("java-platform")
}

dependencies {
    constraints {
        api("org.apache.commons:commons-lang3:3.12.0")
        api("com.google.guava:guava:30.1.1-jre")
        api("org.slf4j:slf4j-api:1.7.30")
    }
}
```

platform/build.gradle

```
plugins {  
    id("java-platform")  
}  
  
dependencies {  
    constraints {  
        api("org.apache.commons:commons-lang3:3.12.0")  
        api("com.google.guava:guava:30.1.1-jre")  
        api("org.slf4j:slf4j-api:1.7.30")  
    }  
}
```

Then, you can use that platform in your project:

app/build.gradle.kts

```
plugins {  
    id("java-library")  
}  
  
dependencies {  
    implementation(platform(project(":platform")))  
}
```

app/build.gradle

```
plugins {  
    id("java-library")  
}  
  
dependencies {  
    implementation(platform(":platform"))  
}
```

Here, `platform` defines versions for `commons-lang3`, `guava`, and `slf4j-api`, ensuring they are compatible.

Maven's BOM (Bill of Materials) is a popular type of platform that Gradle supports. A BOM file lists dependencies with specific versions, allowing you to manage these versions in a centralized way.

A popular platform is the [Spring Boot Bill of Materials](#). To use the BOM, you add it to the dependencies of your project:

build.gradle.kts

```
dependencies {
    // import a BOM
    implementation(platform("org.springframework.boot:spring-boot-
dependencies:1.5.8.RELEASE"))
    // define dependencies without versions
    implementation("com.google.code.gson:gson")
    implementation("dom4j:dom4j")
}
```

build.gradle

```
dependencies {
    // import a BOM
    implementation platform('org.springframework.boot:spring-boot-
dependencies:1.5.8.RELEASE')
    // define dependencies without versions
    implementation 'com.google.code.gson:gson'
    implementation 'dom4j:dom4j'
}
```

By including the `spring-boot-dependencies` platform dependency, you ensure that all Spring components use the versions defined in the BOM file.

Using a Version catalog

A version catalog is a centralized list of dependency coordinates that can be referenced in multiple projects. You can reference this catalog in your build scripts to ensure each project depends on a common set of well-known dependencies.

First, create a `libs.versions.toml` file in the `gradle` directory of your project. This file will define the versions of your dependencies and plugins:

gradle/libs.versions.toml

```
[versions]
groovy = "3.0.5"
checkstyle = "8.37"
```

```
[libraries]
groovy-core = { module = "org.codehaus.groovy:groovy", version.ref = "groovy"
}
groovy-json = { module = "org.codehaus.groovy:groovy-json", version.ref =
"groovy" }
groovy-nio = { module = "org.codehaus.groovy:groovy-nio", version.ref =
"groovy" }
commons-lang3 = { group = "org.apache.commons", name = "commons-lang3",
version = { strictly = "[3.8, 4.0[" , prefer="3.9" } }

[bundles]
groovy = ["groovy-core", "groovy-json", "groovy-nio"]

[plugins]
versions = { id = "com.github.ben-manes.versions", version = "0.45.0" }
```

Then, you can use the version catalog in you build file:

build.gradle.kts

```
plugins {
    `java-library`
    alias(libs.plugins.versions)
}

dependencies {
    api(libs.bundles.groovy)
}
```

build.gradle

```
plugins {
    id 'java-library'
    alias(libs.plugins.versions)
}

dependencies {
    api libs.bundles.groovy
}
```

Next Step: [Learn about Dependency Constraints and Conflict Resolution](#) >>

5. Dependency Constraints and Conflict Resolution

When the same library is declared multiple times or when two different libraries provide the same functionality, a conflict can occur during dependency resolution.

Understanding types of conflicts

During dependency resolution, Gradle handles two types of conflicts:

1. **Version conflicts:** That is when two or more dependencies require a given module but with different versions.
2. **Capability conflicts:** That is when the dependency graph contains multiple artifacts that provide the same functionality.

Resolving version conflicts

A version conflict occurs when a component declares two dependencies that:

- Depend on the same module, let's say `com.google.guava:guava`
- But on different versions, let's say `20.0` and `25.1-android`
 - Our project itself depends on `com.google.guava:guava:20.0`
 - Our project also depends on `com.google.inject:guice:4.2.2` which itself depends on `com.google.guava:guava:25.1-android`

Gradle will consider *all* requested versions, wherever they appear in the dependency graph. By default, it will select the *highest* one out of these versions.

Resolving capability conflicts

Gradle uses attributes and capabilities to identify which artifacts a component *provides*. A capability conflict occurs whenever two or more variants of a component in dependency graph declare the same capability.

Gradle will generally fail the build and report the conflict.

You can resolve conflicts manually by specifying which capability to use in the `resolutionStrategy` block:

```
configurations.configureEach {
    resolutionStrategy.capabilitiesResolution.withCapability("com.example:logging") {
        selectHighestVersion()
    }
}
```

Understanding dependency constraints

In order to help Gradle resolve issue with dependencies, a number of solutions are provided.

For example, the `dependencies` block provides a `constraints` block which can be used to help Gradle pick a specific version of a dependency:

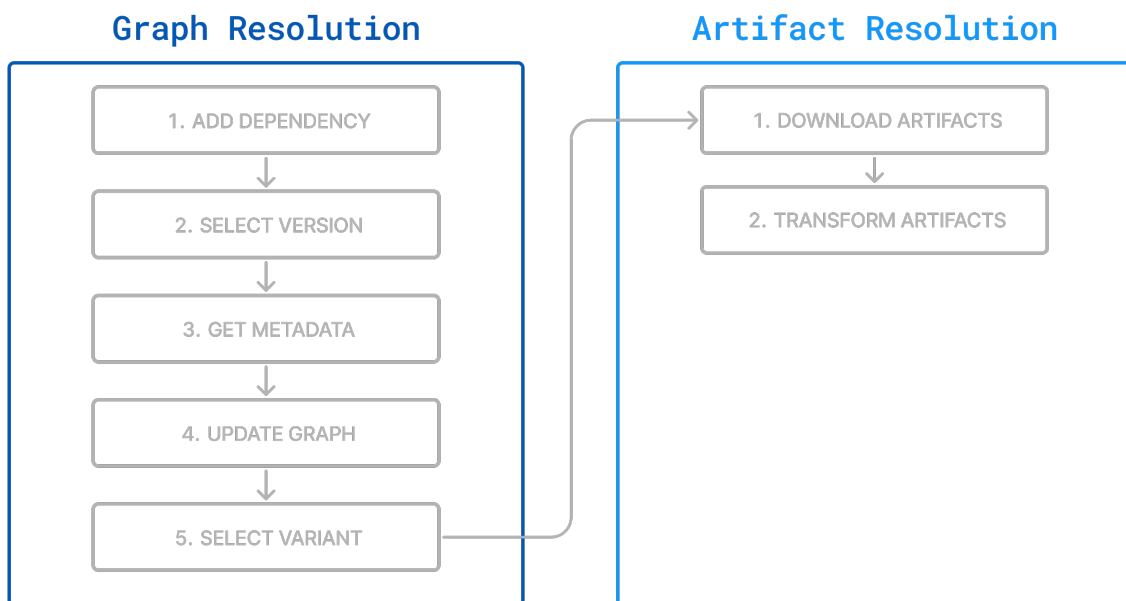
```
dependencies {  
    constraints {  
        implementation("org.apache.commons:commons-lang3:3.12.0")  
    }  
}
```

Next Step: [Learn about Dependency Resolution >>](#)

6. Dependency Resolution

Dependency resolution in Gradle involves two main steps:

1. **Graph Resolution**
2. **Artifact Resolution**



1. Graph Resolution

Graph resolution is the process of determining the full set of transitive dependencies, and their versions, that are required for a given set of declared dependencies.

Graph resolution operates solely on dependency metadata (GMM, POMs). In this phase, artifacts (JARs) are *not* resolved. Only the structure of the graph, based on the relationship between dependencies, are calculated at this time.

1. Discovering dependencies

1. ADD DEPENDENCY

`com.fasterxml.jackson.core:jackson-databind`

Graph resolution begins with the **project** and external (**module**) dependencies declared in the build script.

- A **module** is a discrete unit of software that can be built and published, such as `com.fasterxml.jackson.core:jackson-databind`.
- Each version of a **module** is referred to as a **component**, such as `com.fasterxml.jackson.core:jackson-databind:2.17.2`.

A **project** contributes a single component to the dependency graph, which itself belongs to a module.

In the example below, the **component** `com.fasterxml.jackson.core:jackson-databind:2.17.2` is added as a dependency to the **implementation configuration** in a Java application:

build.gradle.kts

```
dependencies {  
    implementation("com.fasterxml.jackson.core:jackson-databind:2.17.2")  
}
```

2. Perform conflict resolution

2. SELECT VERSION

`com.fasterxml.jackson.core:jackson-databind:2.17.2`

Gradle identifies and resolves any version conflicts when multiple declared or transitive dependencies request different **versions** of the same **module**.

Even though a user might declare version `2.17.2` of a **module**, this may not be the version ultimately resolved in the graph. Gradle's conflict resolution strategy, which defaults to selecting the highest version, selects a single version of a module when multiple are requested.

However, Gradle APIs can be used to change the outcome:

- **Resolution Rules:** Gradle allows configuring rules to enforce specific versions, reject certain versions, or substitute dependencies as needed.
- **Dependency Substitution:** Rules defined in build logic can replace one dependency with another, alter versions, or redirect requests for one module with another.
- **Dynamic Versions:** If dependencies are defined with dynamic versions (e.g., `1.0.+`) or version ranges (e.g., `[1.0, 2.0)`), Gradle resolves these to specific versions by querying the repositories.
- **Dependency Locking:** If enabled, Gradle checks lock files to ensure consistent versions across build invocations, preventing unexpected changes in dependency versions.

In the example, Gradle selects the **component** `com.fasterxml.jackson.core:jackson-databind:2.17.2` (the `2.17.2` version of the `com.fasterxml.jackson.core:jackson-databind` **module**).

3. Retrieve the metadata

3. GET METADATA

`jackson-databind-2.17.2.module`

Once Gradle has determined which version of an external **module** to resolve, it fetches the **metadata** for the **component** from an `ivy`, `pom`, or `GMM` metadata file in the repository.

Here's a sample of the **metadata** for `com.fasterxml.jackson.core:jackson-databind:2.17.2`:

```
{
  "formatVersion": "1.1",
  "component": {
    "group": "com.fasterxml.jackson.core",
    "module": "jackson-databind",
    "version": "2.17.2",
  },
  "variants": [
    {
      "name": "apiElements"
    },
    {
      "name": "runtimeElements",
      "attributes": {
        "org.gradle.category": "library",
        "org.gradle.dependency.bundling": "external",
        "org.gradle.libraryelements": "jar",
        "org.gradle.usage": "java-runtime"
      }
    },
  ],
  "dependencies": [
```

```

    {
      "group": "com.fasterxml.jackson.core",
      "module": "jackson-annotations",
      "version": {
        "requires": "2.17.2"
      }
    },
    {
      "group": "com.fasterxml.jackson.core",
      "module": "jackson-core",
      "version": {
        "requires": "2.17.2"
      }
    },
    {
      "group": "com.fasterxml.jackson",
      "module": "jackson-bom",
      "version": {
        "requires": "2.17.2"
      }
    }
  ],
  "files": [
    {
      "name": "jackson-databind-2.17.2.jar"
    }
  ]
}
]
}

```

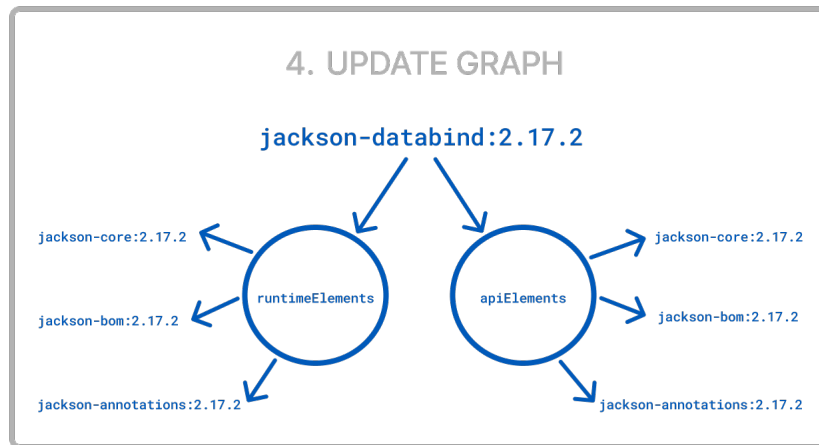
As you can see, the `com.fasterxml.jackson.core:jackson-databind:2.17.2` **component** offers two **variants**:

- The `apiElements` **variant** includes dependencies required for compiling projects against Jackson Databind.
- The `runtimeElements` **variant** includes dependencies required for executing Jackson Databind during runtime.

A **variant** is a specific variation of a **component** tailored for a particular use case or environment. **Variants** allow you to provide different definitions of your **component** depending on the context in which it's used.

Each **variant** consists of a set of **artifacts** and defines a set of **dependencies**. The `runtimeElements` variant provides the `jackson-databind-2.17.2.jar` artifact, which will be downloaded later in the **Artifact Resolution** phase.

4. Update the graph



Gradle builds a dependency graph that represents a configuration's dependencies and their relationships. This graph includes both direct dependencies (explicitly declared in the build script) and transitive dependencies (dependencies of the direct dependencies and other transitive dependencies).

The dependency graph is made up of *nodes* where:

- Each *node* represents a **variant**.
- Each **dependency** selects a **variant** from a **component**.

These *nodes* are connected by edges, representing the dependencies between **variants**. The edges indicate how one variant relies on another.

For instance, if your project depends on Jackson Databind, and Jackson Databind depends on **jackson-annotations**, the edge in the graph represents that **jackson-annotations** is a dependency of one of Jackson Databind's variants.

The **dependencies** task can be used to visualize the structure of a dependency graph:

```
$ ./gradlew app:dependencies
```

```
[...]
```

```
runtimeClasspath - Runtime classpath of source set 'main'.
\--- com.fasterxml.jackson.core:jackson-databind:2.17.2
    +--- com.fasterxml.jackson.core:jackson-annotations:2.17.2
        |    \--- com.fasterxml.jackson:jackson-bom:2.17.2
        |          +--- com.fasterxml.jackson.core:jackson-annotations:2.17.2 (c)
        |          +--- com.fasterxml.jackson.core:jackson-core:2.17.2 (c)
        |          \--- com.fasterxml.jackson.core:jackson-databind:2.17.2 (c)
    +--- com.fasterxml.jackson.core:jackson-core:2.17.2
        |    \--- com.fasterxml.jackson:jackson-bom:2.17.2 (*)
    \--- com.fasterxml.jackson:jackson-bom:2.17.2 (*)
```

In this output, `runtimeClasspath` represent specific resolvable configurations in the project. Each resolvable configuration calculates a separate dependency graph.

Different configurations can resolve to a different set of transitive dependencies for the same set of declared dependencies. Each **variant** is owned by a specific version of a **component**.

5. Select a variant



Based on the requirements of the build, Gradle selects one of the **variants** of the module.

To describe and differentiate between **variants**, you use **attributes**. **Attributes** are used to define specific characteristics or properties of **variants** and the context in which those **variants** should be used.

In the metadata for Jackson Databind, we see that the `runtimeElements` variant is described by the `org.gradle.category`, `org.gradle.dependency.bundling`, `org.gradle.libraryelement`, and `org.gradle.usage` attributes:

```
{
  "variants": [
    {
      "name": "runtimeElements",
      "attributes": {
        "org.gradle.category": "library",
        "org.gradle.dependency.bundling": "external",
        "org.gradle.libraryelements": "jar",
        "org.gradle.usage": "java-runtime" ①
      }
    }
  ]
}
```

① For the `apiElements` variant, this attribute differs: `"org.gradle.usage": "java-api"`

Attributes are used to select the appropriate **variant** during dependency resolution.

In the case of our Java application example, which has Jackson Databind as a dependency, Gradle will select the runtime variant to build the app.

To see a more detailed view of which **variant** Gradle resolved for a given configuration, you can run the `dependencyInsight` task:

```
$ ./gradlew :app:dependencyInsight --configuration runtimeClasspath --dependency com.fasterxml.jackson.core:jackson-databind:2.17.2
```

```
> Task :app:dependencyInsight
```

```
com.fasterxml.jackson.core:jackson-databind:2.17.2 (by constraint)
```

```
Variant runtimeElements:
```

Attribute Name	Provided	Requested
org.gradle.status	release	
org.gradle.category	library	library
org.gradle.dependency.bundling	external	external
org.gradle.libraryelements	jar	jar
org.gradle.usage	java-runtime	java-runtime
org.gradle.jvm.environment		standard-jvm
org.gradle.jvm.version		11

```
com.fasterxml.jackson.core:jackson-databind:2.17.2
```

```
+--- runtimeClasspath
```

```
\--- com.fasterxml.jackson:jackson-bom:2.17.2
```

```
+--- com.fasterxml.jackson.core:jackson-annotations:2.17.2
```

```
| +--- com.fasterxml.jackson:jackson-bom:2.17.2 (*)
```

```
| \--- com.fasterxml.jackson.core:jackson-databind:2.17.2 (*)
```

```
+--- com.fasterxml.jackson.core:jackson-core:2.17.2
```

```
| +--- com.fasterxml.jackson:jackson-bom:2.17.2 (*)
```

```
| \--- com.fasterxml.jackson.core:jackson-databind:2.17.2 (*)
```

```
\--- com.fasterxml.jackson.core:jackson-databind:2.17.2 (*)
```

In this example, Gradle uses the `runtimeElements` **variant** of `jackson-databind` for the `runtimeClasspath` configuration.

2. Artifact Resolution

Artifact resolution occurs after the dependency graph is constructed. For each node in the dependency graph, Gradle fetches the necessary physical files (**artifacts**).

This process uses the resolved graph and repository definitions to produce the required files as output.

1. Fetching artifacts

1. DOWNLOAD ARTIFACTS

[jackson-databind-2.17.2.jar](#)

Gradle locates and downloads the actual **artifacts** (such as JAR files, ZIP files, etc.) referenced in the graph. These artifacts correspond to the nodes discovered during graph resolution.

In our example, Gradle resolved the **runtimeElements** **variant** of **com.fasterxml.jackson.core:jackson-databind** during the dependency graph resolution. That **variant** corresponds to the JAR file **jackson-databind-2.17.2.jar** as the artifact:

```
{
  "component": {
    "group": "com.fasterxml.jackson.core",
    "module": "jackson-databind",
    "version": "2.17.2"
  },
  "variants": [
    {
      "name": "apiElements",
      "dependencies": [],
      "files": [
        {
          "name": "jackson-databind-2.17.2.jar"
        }
      ]
    }
  ]
}
```

Gradle also fetches the resolved transitive dependencies of Jackson Databind including **jackson-annotations** and **jackson-core** which correspond to **jackson-annotations-2.17.2.jar** and **jackson-core-2.17.2.jar** respectively.

2. Transform artifacts

2. TRANSFORM ARTIFACTS

```
jackson-databind-2.17.2.zip -> jackson-databind-2.17.2.jar
```

Gradle can transform artifacts using **artifact transforms** if needed or requested. Transforms are typically applied automatically during dependency resolution when Gradle needs to convert one artifact format into another that your build requires.

For example, **jackson-databind** might only produce a ZIP file as an artifact called **jackson-databind-2.17.2.zip**, but the build needs **jackson-databind-2.17.2.jar**. Gradle can use Gradle provided transforms or user programmed transforms to convert the **zip** file into a **jar** file.

7. Variant Aware Dependency Resolution

In Gradle, dependency resolution is often thought of from the standpoint of a **consumer** and a **producer**. The **consumer** declares dependencies and performs dependency resolution, while **producers** satisfy those dependencies by exposing **variants**.

Gradle's resolution engine follows a dynamic approach to dependency resolution called **variant-aware resolution**, where the consumer defines requirements using **attributes**, which are matched with the attributes declared by the producer.

Variant-aware resolution allows Gradle to automatically select the correct **variant** from a producer without the consumer explicitly specifying which one to use.

For instance, if you're working with different architectures (like **arm64** and **i386**), Gradle can choose the appropriate version of a library (**myLib**) for each architecture:

1. The **producer**, **myLib**, exposes variants (**arm64Elements**, **i386Elements**) with specific **attributes** (e.g., **ArchType.ARM64**, **ArchType.I386**).
2. The **consumer**, **myApp**, specifies the required **attributes** (e.g., **ArchType.ARM64**) in its resolvable configuration (**runtimeClasspath**).
3. If the **consumer**, **myApp**, requires dependencies for the **arm64** architecture, Gradle will automatically pick the **arm64Elements** **variant** from the **myLib** **producer** and use its corresponding **artifact**.

A coded example

Consider a Java library where you create a new **variant** called **instrumentedJars** and want to ensure it's selected for testing:

1. **Producer Project:** Creates a specialized **instrumentedJars** **variant** marked with specific attributes.
2. **Consumer Project:** Configured to request the **instrumented-jar** **attribute** for testing.

Let's look at the build files of the producer and consumer.

The producer side

1. Create an instrumented JAR:

Our Java library has a task called **instrumentedJar** which produces a JAR file. We expect other projects to consume this JAR file.

```
producer/build.gradle.kts
```

```
val instrumentedJar = tasks.register("instrumentedJar", Jar::class) {
```

```
    archiveClassifier = "instrumented"
}
```

producer/build.gradle

```
def instrumentedJar = tasks.register("instrumentedJar", Jar) {
    archiveClassifier = "instrumented"
}
```

2. Create a custom outgoing configuration:

We want our instrumented classes to be used when executing tests, so we need to define proper **attributes** on our **variant**. We create a new configuration named `instrumentedJars`. This configuration:

- Can be consumed by other projects.
- Cannot be resolved (i.e., it's meant to be used as an output, not an input).
- Has specific **attributes**, including `LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE` set to "instrumented-jar", which explains what the **variant** contains.

producer/build.gradle.kts

```
val instrumentedJars by configurations.creating {
    isCanBeConsumed = true
    isCanBeResolved = false
    attributes {
        attribute(Category.CATEGORY_ATTRIBUTE,
objects.named(Category.LIBRARY))
        attribute(Usage.USAGE_ATTRIBUTE, objects.named(Usage.JAVA_RUNTIME))
        attribute(Bundling.BUNDLING_ATTRIBUTE,
objects.named(Bundling.EXTERNAL))
        attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
JavaVersion.current().majorVersion.toInt())
        attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE,
objects.named("instrumented-jar"))
    }
}
```

producer/build.gradle

```
configurations {
    instrumentedJars {
```

```

        canBeConsumed = true
        canBeResolved = false
        attributes {
            attribute(Category.CATEGORY_ATTRIBUTE, objects.named(Category,
Category.LIBRARY))
            attribute(Usage.USAGE_ATTRIBUTE, objects.named(Usage, Usage
.JAVA_RUNTIME))
            attribute(Bundling.BUNDLING_ATTRIBUTE, objects.named(Bundling,
Bundling.EXTERNAL))
            attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
JavaVersion.current().majorVersion.toInteger())
            attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE, objects
.named(LibraryElements, 'instrumented-jar'))
        }
    }
}

```

3. Attach the Artifact:

The `instrumentedJar` task's output is added to the `instrumentedJars` configuration as an **artifact**. When this variant is included in a dependency graph, this artifact will be resolved during artifact resolution.

producer/build.gradle.kts

```

artifacts {
    add("instrumentedJars", instrumentedJar)
}

```

producer/build.gradle

```

artifacts {
    instrumentedJars(instrumentedJar)
}

```

What we have done here is that we have added a *new variant*, which can be used *at runtime*, but contains instrumented classes instead of the normal classes. However, it now means that for runtime, the consumer has to choose between two variants:

1. `runtimeElements`, the regular variant offered by the `java-library` plugin
2. `instrumentedJars`, the variant we have created

The consumer side

1. Add dependencies:

First, on the consumer side, like any other project, we define the Java library as a dependency:

consumer/build.gradle.kts

```
dependencies {  
    testImplementation("junit:junit:4.13")  
    testImplementation(project(":producer"))  
}
```

consumer/build.gradle

```
dependencies {  
    testImplementation 'junit:junit:4.13'  
    testImplementation project(':producer')  
}
```

At this point, Gradle will still select the default `runtimeElements` **variant** for your dependencies. This is because the `testRuntimeClasspath` configuration is requesting **artifacts** with the `jar` library elements **attribute**, while the producer defines the `instrumentedJars` **variant** with a different **attribute**.

2. Adjust the requested attributes:

The `testRuntimeClasspath` configuration is modified to ask for "instrumented-jar" versions of the dependencies. This means that when Gradle resolves dependencies for this configuration, it will prefer JAR files that are marked as "instrumented":

consumer/build.gradle.kts

```
configurations {  
    testRuntimeClasspath {  
        attributes {  
            attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE,  
objects.named(LibraryElements::class.java, "instrumented-jar"))  
        }  
    }  
}
```

consumer/build.gradle

```
configurations {
    testRuntimeClasspath {
        attributes {
            attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE, objects
.named(LibraryElements, 'instrumented-jar'))
        }
    }
}
```

By following these steps, Gradle will intelligently select the correct **variants** based on the configuration and **attributes**, while also handling cases where specialized **variants** are not available.

DECLARING DEPENDENCIES

Declaring Dependencies Basics

Types of dependencies

There are three main types of dependencies in Gradle:

1. **Module Dependencies:** Refer to libraries from external repositories.
2. **Project Dependencies:** Refer to other projects in the same multi-project build.
3. **File Dependencies:** Refer to local files or directories, such as `.jar` or `.aar` files.

1. Module dependencies

Module dependencies are the most common dependencies. They refer to a dependency that is identified by module coordinates (group, name, and version):

build.gradle.kts

```
dependencies {
    runtimeOnly(group = "org.springframework", name = "spring-core", version = "2.5")
    runtimeOnly("org.springframework:spring-aop:2.5")
    runtimeOnly("org.hibernate:hibernate:3.0.5") {
        isTransitive = true
    }
    runtimeOnly(group = "org.hibernate", name = "hibernate", version = "3.0.5") {
        isTransitive = true
    }
}
```

build.gradle

```
dependencies {
    runtimeOnly group: 'org.springframework', name: 'spring-core', version: '2.5'
    runtimeOnly 'org.springframework:spring-core:2.5',
               'org.springframework:spring-aop:2.5'
    runtimeOnly(
        [group: 'org.springframework', name: 'spring-core', version: '2.5'],
        [group: 'org.springframework', name: 'spring-aop', version: '2.5']
    )
    runtimeOnly('org.hibernate:hibernate:3.0.5') {
        transitive = true
    }
}
```

```

    }
    runtimeOnly group: 'org.hibernate', name: 'hibernate', version: '3.0.5',
    transitive: true
    runtimeOnly(group: 'org.hibernate', name: 'hibernate', version: '3.0.5')
    {
        transitive = true
    }
}

```

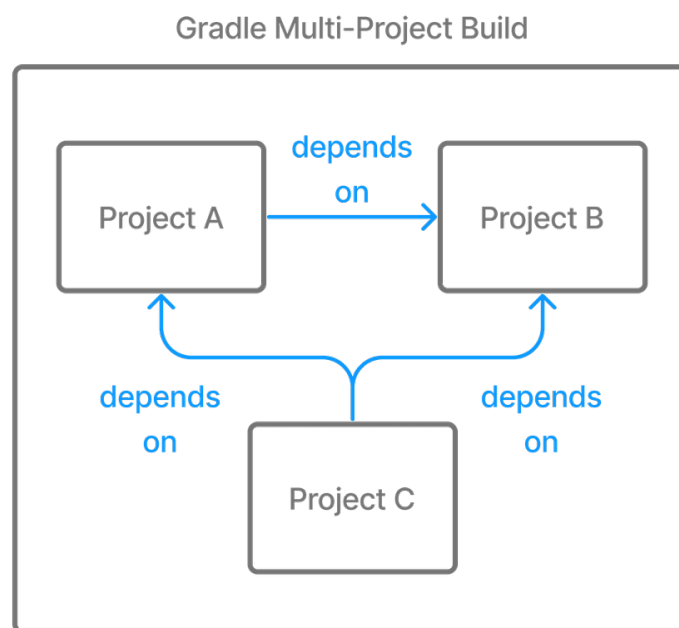
Gradle offers multiple notations for declaring module dependencies, including string notation and map notation.

- **String Notation:** Simplifies dependency declaration by combining the group, name, and version into a single string.
- **Map Notation:** Allows for specifying each part of the coordinates separately.

For advanced configurations, such as enforcing strict versions, you can also provide a closure when alongside these notations.

2. Project dependencies

Project dependencies allow you to reference other projects within a multi-project Gradle build.



This is useful for organizing large projects into smaller, modular components:

web-service/build.gradle.kts

```

dependencies {
    implementation(project(":utils"))
    implementation(project(":api"))
}

```

```
}
```

web-service/build.gradle

```
dependencies {  
    implementation project(':utils')  
    implementation project(':api')  
}
```

Gradle uses the `project()` function to define a project dependency. This function takes the relative path to the target project within the build. The path is typically defined using a colon (:) to separate different levels of the project structure.

Project dependencies are automatically resolved such that the dependent project is always built before the project that depends on it.

Type-safe project dependencies

Type-safe project accessors are an incubating feature which must be enabled explicitly. Implementation may change at any time.

To add support for type-safe project accessors, add `enableFeaturePreview("TYPESAFE_PROJECT_ACCESSORS")` this to your `settings.gradle(.kts)` file:

settings.gradle.kts

```
enableFeaturePreview("TYPESAFE_PROJECT_ACCESSORS")
```

settings.gradle

```
enableFeaturePreview 'TYPESAFE_PROJECT_ACCESSORS'
```

One downside of using the `project(":some:path")` notation is the need to remember project paths for dependencies. Moreover, changing a project path requires manually updating every occurrence, increasing the risk of missing one.

Instead, the experimental type-safe project accessors API provides IDE completion, making it easier to declare dependencies:

web-service/build.gradle.kts

```
dependencies {  
    implementation/projects.utils)  
    implementation/projects.api)  
}
```

web-service/build.gradle

```
dependencies {  
    implementation projects.utils  
    implementation projects.api  
}
```

With this API, incorrectly specified projects in Kotlin DSL scripts trigger compilation errors, helping you avoid missed updates.

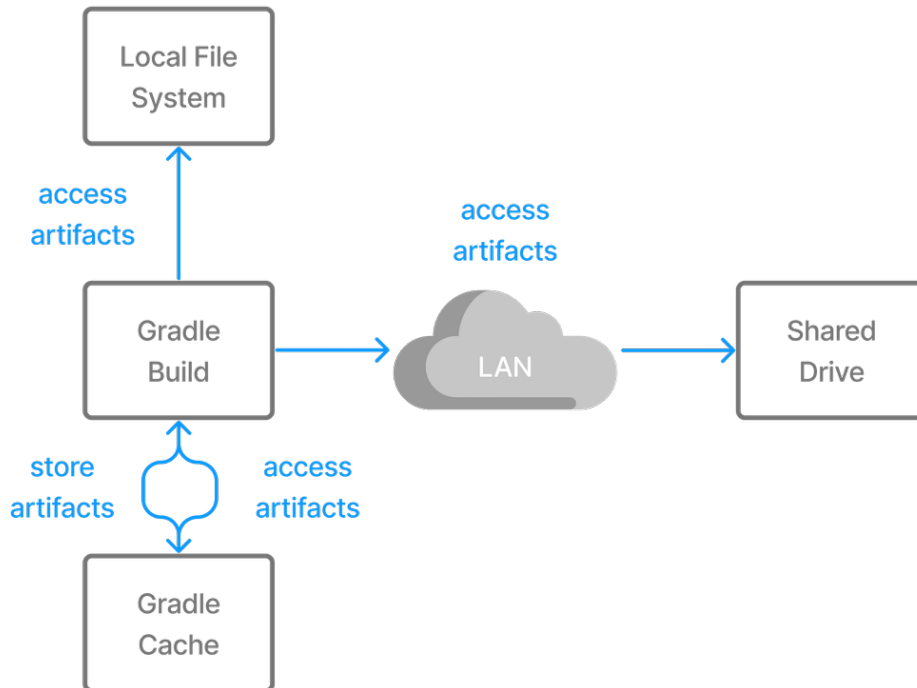
Project accessors are based on project paths. For instance, the path `:commons:utils:some:lib` becomes `projects.commonUtils.someLib`, while kebab-case (`some-lib`) and snake-case (`some_lib`) are converted to camel case: `projects.someLib`.

3. File dependencies

File dependencies allow you to include external JARs or other files directly into your project by referencing their file paths. File dependencies also allow you to add a set of files directly to a configuration without using a repository.

NOTE

File dependencies are generally discouraged. Instead, prefer declaring dependencies on an external repository, or if necessary, declaring a maven or ivy repository using a `file://` URL.



File dependencies are unique because they represent a direct reference to files on the filesystem without any associated [metadata](#), such as transitive dependencies, origin, or author information.

build.gradle.kts

```
configurations {
    create("antContrib")
    create("externalLibs")
    create("deploymentTools")
}

dependencies {
    "antContrib"(files("ant/antcontrib.jar"))
    "externalLibs"(files("libs/commons-lang.jar", "libs/log4j.jar"))
    "deploymentTools"(fileTree("tools") { include("*.exe") })
}
```

build.gradle

```
configurations {
    antContrib
    externalLibs
    deploymentTools
}

dependencies {
    antContrib files('ant/antcontrib.jar')
    externalLibs files('libs/commons-lang.jar', 'libs/log4j.jar')
```

```
deploymentTools(fileTree('tools') { include '*.exe' })
}
```

In this example, each dependency explicitly specifies its location within the file system. Common methods for referencing these files include:

- `link:Project.files()`: Accepts one or more file paths directly.
- `ProjectLayout.files()`: Accepts one or more file paths directly.
- `Project.fileTree()`: Defines a directory and includes or excludes specific file patterns.

NOTE

The order of files in a `FileTree` is not guaranteed to be stable, even on the same computer. As a result, dependency configurations using `FileTree` may produce resolution results with varying order, which can impact the cacheability of tasks that use these results as inputs. To ensure more predictable and stable behavior, it is recommended to use the simpler `files` method where possible.

Alternatively, you can use a [flat directory](#) repository to specify the source directory for multiple file dependencies.

Ideally, you should use Maven or Ivy repository with a local URL:

```
repositories {
    maven {
        url = 'file:///path/to/local/files' // Replace with your actual path
    }
}
```

To add files as dependencies, pass a [file collection](#) to the configuration:

build.gradle.kts

```
dependencies {
    runtimeOnly(files("libs/a.jar", "libs/b.jar"))
    runtimeOnly(fileTree("libs") { include("*.jar") })
}
```

build.gradle

```
dependencies {
    runtimeOnly files('libs/a.jar', 'libs/b.jar')
    runtimeOnly fileTree('libs') { include '*.jar' }
}
```

Note that file dependencies are not included in the published dependency descriptor for your project. However, they are available in transitive dependencies within the same build, meaning they can be used within the current build but not outside it.

You should specify which tasks produce the files for a file dependency. Otherwise, the necessary tasks might not run when you depend on them transitively from another project:

build.gradle.kts

```
dependencies {
    implementation(files(layout.buildDirectory.dir("classes")) {
        builtBy("compile")
    })
}

tasks.register("compile") {
    doLast {
        println("compiling classes")
    }
}

tasks.register("list") {
    val compileClasspath: FileCollection = configurations["compileClasspath"]
    dependsOn(compileClasspath)
    doLast {
        println("classpath = ${compileClasspath.map { file: File -> file.name
    }}")
    }
}
```

build.gradle

```
dependencies {
    implementation files(layout.buildDirectory.dir('classes')) {
        builtBy 'compile'
    }
}

tasks.register('compile') {
    doLast {
        println 'compiling classes'
    }
}

tasks.register('list') {
    FileCollection compileClasspath = configurations.compileClasspath
    dependsOn compileClasspath
}
```

```
doLast {  
    println "classpath = ${compileClasspath.collect { File file -> file  
    .name } }"  
}  
}
```

```
$ gradle -q list  
compiling classes  
classpath = [classes]
```

Gradle distribution-specific dependencies

Gradle API dependency

You can declare a dependency on the API of the current version of Gradle by using the [DependencyHandler.gradleApi\(\)](#) method. This is useful when you are developing custom Gradle tasks or plugins:

build.gradle.kts

```
dependencies {  
    implementation(gradleApi())  
}
```

build.gradle

```
dependencies {  
    implementation gradleApi()  
}
```

Gradle TestKit dependency

You can declare a dependency on the TestKit API of the current version of Gradle by using the [DependencyHandler.gradleTestKit\(\)](#) method. This is useful for writing and executing functional tests for Gradle plugins and build scripts:

build.gradle.kts

```
dependencies {
```

```
testImplementation(gradleTestKit())
}
```

build.gradle

```
dependencies {
    testImplementation gradleTestKit()
}
```

Local Groovy dependency

You can declare a dependency on the Groovy that is distributed with Gradle by using the [DependencyHandler.localGroovy\(\)](#) method. This is useful when you are developing custom Gradle tasks or plugins in Groovy:

build.gradle.kts

```
dependencies {
    implementation(localGroovy())
}
```

build.gradle

```
dependencies {
    implementation localGroovy()
}
```

Documenting dependencies

When declaring a dependency or a [dependency constraint](#), you can provide a reason to clarify why the dependency is included. This helps make your build script and the [dependency insight report](#) easier to interpret:

build.gradle.kts

```
plugins {
    `java-library`
}
```

```

repositories {
    mavenCentral()
}

dependencies {
    implementation("org.ow2.asm:asm:7.1") {
        because("we require a JDK 9 compatible bytecode generator")
    }
}

```

build.gradle

```

plugins {
    id 'java-library'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation('org.ow2.asm:asm:7.1') {
        because 'we require a JDK 9 compatible bytecode generator'
    }
}

```

In this example, the `because()` method provides a reason for including the `asm` library, which helps explain its purpose in the context of the build:

```

$ gradle -q dependencyInsight --dependency asm
org.ow2.asm:asm:7.1
Variant compile:
| Attribute Name          | Provided | Requested |
|-----|-----|-----|
| org.gradle.status       | release  |           |
| org.gradle.category     | library  | library   |
| org.gradle.libraryelements | jar      | classes   |
| org.gradle.usage        | java-api | java-api   |
| org.gradle.dependency.bundling |         | external  |
| org.gradle.jvm.environment |         | standard-jvm |
| org.gradle.jvm.version   |         | 11         |
Selection reasons:
- Was requested: we require a JDK 9 compatible bytecode generator

org.ow2.asm:asm:7.1
\--- compileClasspath

```

A web-based, searchable dependency report is available by adding the `--scan` option.

Viewing Dependencies

Gradle offers tools to navigate the results of dependency management, allowing you to more precisely understand how and why Gradle resolves dependencies. You can render a full dependency graph, identify the origin of a given dependency, and see why specific versions were selected. Dependencies can come from build script declarations or transitive relationships.

To visualize dependencies, you can use:

- The `dependencies` task
- The `dependencyInsight` task
- A [Build Scan](#)

List project dependencies using the `dependencies` task

Gradle provides the built-in `dependencies` task to render a dependency tree from the command line. By default, the task shows dependencies for all `configurations` within a `single project`. The dependency tree shows the selected version of each dependency and provides information on conflict resolution.

The `dependencies` task is particularly useful for analyzing transitive dependencies. While your build file lists direct dependencies, the task helps you understand which transitive dependencies are resolved during the build.

```
$ ./gradlew dependencies
```

TIP

To render the graph of dependencies `declared in the buildscript classpath configuration`, use the `buildEnvironment` task.

Understanding output annotations

```
$ ./gradlew :app:dependencies

> Task :app:dependencies

-----
Project ':app'
-----

annotationProcessor - Annotation processors and their dependencies for source set 'main'.
No dependencies
```

```
compileClasspath - Compile classpath for source set 'main'.
\--- com.fasterxml.jackson.core:jackson-databind:2.17.2
    +--- com.fasterxml.jackson.core:jackson-annotations:2.17.2
         |    \--- com.fasterxml.jackson:jackson-bom:2.17.2
         |         +--- com.fasterxml.jackson.core:jackson-annotations:2.17.2 (c)
         |         +--- com.fasterxml.jackson.core:jackson-core:2.17.2 (c)
         |         \--- com.fasterxml.jackson.core:jackson-databind:2.17.2 (c)
    +--- com.fasterxml.jackson.core:jackson-core:2.17.2
         |    \--- com.fasterxml.jackson:jackson-bom:2.17.2 (*)
    \--- com.fasterxml.jackson:jackson-bom:2.17.2 (*)

...

```

The **dependencies** task marks dependency trees with the following annotations:

- **(*)**: Indicates repeated occurrences of a transitive dependency subtree. Gradle expands transitive dependency subtrees only once per project; repeat occurrences only display the root of the subtree, followed by this annotation.
- **(c)**: This element is a **dependency constraint**, not a dependency. Look for the matching dependency elsewhere in the tree.
- **(n)**: A dependency or dependency configuration that **cannot be resolved**.

Specifying a dependency configuration

To focus on a specific dependency configuration, use the optional **--configuration** parameter.

Like **project and task names**, Gradle allows abbreviated names for dependency configurations. For example, you can use **tRC** instead of **testRuntimeClasspath**, as long as it matches a unique configuration.

The following examples display dependencies for the **testRuntimeClasspath** configuration in a Java project:

```
$ gradle -q dependencies --configuration testRuntimeClasspath
```

```
$ gradle -q dependencies --configuration tRC
```

To view a list of all configurations in a project, including those provided by plugins, run the **resolvableConfigurations** report. For more details, refer to the plugin's documentation, such as the Java Plugin [here](#).

Looking at an example

Consider a project that uses the **JGit library** to execute Source Control Management (SCM) operations for a release process. You can declare dependencies for external tooling with the help of a **custom dependency configuration**. This avoids polluting other contexts, such as the compilation classpath for your production source code.

The following example declares a custom dependency configuration named `scm` that contains the JGit dependency:

build.gradle.kts

```
configurations {
    create("scm")
}

dependencies {
    "scm"("org.eclipse.jgit:org.eclipse.jgit:4.9.2.201712150930-r")
}
```

build.gradle

```
configurations {
    scm
}

dependencies {
    scm 'org.eclipse.jgit:org.eclipse.jgit:4.9.2.201712150930-r'
}
```

Use the following command to view a dependency tree for the `scm` dependency configuration:

```
$ gradle -q dependencies --configuration scm
```

```
-----
Root project 'dependencies-report'
-----
```

```
scm
```

```
\--- org.eclipse.jgit:org.eclipse.jgit:4.9.2.201712150930-r
    +--- com.jcraft:jsch:0.1.54
    +--- com.googlecode.javaewah:JavaEWAH:1.1.6
    +--- org.apache.httpcomponents:httpclient:4.3.6
        |   +--- org.apache.httpcomponents:httpcore:4.3.3
        |   +--- commons-logging:commons-logging:1.1.3
        |   \--- commons-codec:commons-codec:1.6
    \--- org.slf4j:slf4j-api:1.7.2
```

A web-based, searchable dependency report is available by adding the `--scan` option.

Identify the selected version using the `dependencyInsight` task

A project may request two different versions of the same dependency either directly or transitively that may result in a [version conflict](#).

The following example introduces a conflict with `commons-codec:commons-codec`, added both as a direct dependency and a transitive dependency of JGit:

build.gradle.kts

```
repositories {
    mavenCentral()
}

configurations {
    create("scm")
}

dependencies {
    "scm"("org.eclipse.jgit:org.eclipse.jgit:4.9.2.201712150930-r")
    "scm"("commons-codec:commons-codec:1.7")
}
```

build.gradle

```
repositories {
    mavenCentral()
}

configurations {
    scm
}

dependencies {
    scm 'org.eclipse.jgit:org.eclipse.jgit:4.9.2.201712150930-r'
    scm 'commons-codec:commons-codec:1.7'
}
```

Gradle provides the built-in `dependencyInsight` task to render a *dependency insight report* from the command line.

Dependency insights provide information about a single dependency within a single [configuration](#). Given a dependency, you can identify the reason and origin for its version selection.

`dependencyInsight` accepts the following parameters:

--dependency <dependency> (mandatory)

The dependency to investigate. You can supply a complete `group:name`, or part of it. If multiple dependencies match, Gradle generates a report covering all matching dependencies.

--configuration <name> (mandatory)

The dependency configuration which resolves the given dependency. This parameter is optional for projects that use the `Java plugin`, since the plugin provides a default value of `compileClasspath`.

--single-path (optional)

Render only a single path to the dependency.

--all-variants (optional)

Render information about all variants, not only the selected variant.

The following code snippet demonstrates how to run a dependency insight report for all paths to a dependency named `commons-codec` within the `scm` configuration:

```
$ gradle -q dependencyInsight --dependency commons-codec --configuration scm
commons-codec:commons-codec:1.7
Variant default:
  | Attribute Name      | Provided | Requested |
  |-----|-----|-----|
  | org.gradle.status   | release  |           |
Selection reasons:
  - By conflict resolution: between versions 1.7 and 1.6

commons-codec:commons-codec:1.7
\--- scm

commons-codec:commons-codec:1.6 -> 1.7
\--- org.apache.httpcomponents:httpclient:4.3.6
      \--- org.eclipse.jgit:org.eclipse.jgit:4.9.2.201712150930-r
            \--- scm
```

A web-based, searchable dependency report is available by adding the `--scan` option.

Understanding the selection reasons

The "Selection reasons" section of the dependency insight report lists the reasons why a dependency was selected.

Reason	Meaning
(Absent)	No reason other than a reference, direct or transitive, was present.
Was requested : <text>	The dependency appears in the graph, and the inclusion came with a <code>because text</code> .

Reason	Meaning
Was requested : didn't match versions <versions>	The dependency appears with a dynamic version which did not include the listed versions. May be followed by a because text.
Was requested : reject version <versions>	The dependency appears with a rich version containing one or more reject . May be followed by a because text.
By conflict resolution : between versions <version>	The dependency appeared multiple times, with different version requests. This resulted in conflict resolution to select the most appropriate version.
By constraint	A dependency constraint participated in the version selection. May be followed by a because text.
By ancestor	There is a rich version with a strictly which enforces the version of this dependency.
Selected by rule	A dependency resolution rule overruled the default selection process. May be followed by a because text.
Rejection : <version> by rule because <text>	A ComponentSelection.reject rejected the given version of the dependency.
Rejection: version <version>: <attributes information>	The dependency has a dynamic version and some versions did not match the requested attributes .
Forced	The build enforces the version of the dependency through an enforced platform or resolution strategy.

If multiple selection reasons exist, the insight report lists all of them.

Get a holistic view using Build Scans

The dependency tree in a **Build Scan** shows information about conflicts.

A **Build Scan was created** for the **commons-codec** example above and a URL was provided with the results.

Head over to the **Dependencies** tab and navigate to your desired dependency. Select the **Required By** tab to see the selection reason and origin of the dependency:

8 dependencies resolved in 1 project across 1 configuration

scm - 0.010s

- commons-codec:commons-codec:1.7
- org.eclipse.jgit:org.eclipse.jgit:4.9.2.201712150930-r
- com.googlecode.javaewah:JavaEWAH:1.1.6
- com.jcraft:jsch:0.1.54
- org.apache.httpcomponents:httpclient:4.3.6
- commons-codec:commons-codec:1.6 → 1.7** conflict resolution
- commons-logging:commons-logging:1.1.3
- org.apache.httpcomponents:httpcore:4.3.3
- org.slf4j:slf4j-api:1.7.2

Dependencies

Required By

- commons-codec:commons-codec:1.6 → 1.7 conflict resolution
- org.apache.httpcomponents:httpclient:4.3.6
- org.eclipse.jgit:org.eclipse.jgit:4.9.2.201712150930-r
- : scm

Resolving unsafe configuration resolution errors

Resolving a configuration can have side effects on Gradle's project model, so Gradle needs manage access to each project's configurations. There are a number of ways a configuration might be resolved unsafely. Gradle will produce a deprecation warning for each unsafe access. Each of these are bad practices and can cause strange and indeterminate errors.

If your build has an unsafe access deprecation warning, it needs to be fixed.

For example:

- A task from one project directly resolves a configuration in another project in the task's action.
- A task specifies a configuration from another project as an input file collection.
- A build script for one project resolves a configuration in another project during evaluation.
- Project configurations are resolved in the settings file.

In most cases, this issue can be resolved by creating a cross-project dependency on the other project. See the documentation for [sharing outputs between projects](#) for more information.

If you find a use case that can't be resolved using these techniques, please let us know by filing a [GitHub Issue](#) adhering to our issue guidelines.

Declaring Versions and Ranges

You can specify dependencies with exact versions or version ranges to define which versions your project can use:

```
dependencies {  
    implementation("org.springframework:spring-core:5.3.8")  
    implementation("org.springframework:spring-core:5.3.+")  
    implementation("org.springframework:spring-core:latest.release")  
    implementation("org.springframework:spring-core:[5.2.0, 5.3.8]")  
    implementation("org.springframework:spring-core:[5.2.0,)")  
}
```

Understanding version declaration

Gradle supports various ways to declare **versions and ranges**:

Version	Example	Note
Exact version	1.3, 1.3.0-beta3, 1.0-20150201.131010-1	A specific version.
Maven-style range	[1.0,), [1.1, 2.0), (1.2, 1.5]	[] indicates inclusive bounds; () indicates exclusive bounds. See below to learn more . When the upper or lower bound is missing, the range has no upper or lower bound. An upper bound exclude acts as a prefix exclude.
Prefix version range	1.+, 1.3.+	Only versions exactly matching the portion before the + are included. Declaring a version as +, without any prefix, will include any version.
latest-status version	latest.integration, latest.release	Matches the highest version with the specified status. See ComponentMetadata.getStatus() .
Maven SNAPSHOT version	1.0-SNAPSHOT, 1.4.9-beta1-SNAPSHOT	Indicates a snapshot version.

Maven-style range

There are a number of options to indicate bounds in the Maven-style:

- [and] indicate an inclusive bound → [1.1, 2.0]
- (and) indicate an exclusive bound → (1.1, 2.0) or (1.2, 1.5] or [1.1, 2.0)
-] can be used instead of (for an exclusive lower bound →]1.2, 1.5] instead of (1.2, 1.5]
- [can be used instead of) for exclusive upper bound → [1.1, 2.0[instead of [1.1, 2.0)

Understanding version ordering

```
dependencies {  
    implementation("org.springframework:spring-core:1.1") // This is a newer version  
    than 1.a  
    implementation("org.springframework:spring-core:1.a") // This is a older version  
    than 1.1  
}
```

Version ordering is used to:

- Determine if a particular **version is included in a range**.
- Determine which **version is newest** when performing conflict resolution (using "base versions").

Versions are ordered based on the following rules:

- **Splitting Versions into Parts:**
 - Versions are divided into parts using the characters `[. - _ +]`.
 - Parts containing both digits and letters are split further, e.g., `1a1` becomes `1.a.1`.
 - Only the parts are compared, not the separators, so `1.a.1`, `1-a+1`, `1.a-1`, and `1a1` are equivalent. (Note: There are [exceptions](#) during conflict resolution).
- **Comparing Equivalent Parts:**
 - **Numeric vs. Numeric:** Higher numeric value is considered higher: `1.1 < 1.2`.
 - **Numeric vs. Non-numeric:** Numeric parts are higher than non-numeric parts: `1.a < 1.1`.
 - **Non-numeric vs. Non-numeric:** Parts are compared alphabetically and case-sensitively: `1.A < 1.B < 1.a < 1.b`.
 - **Extra Numeric Part:** A version with an additional numeric part is higher, even if it's zero: `1.1 < 1.1.0`.
 - **Extra Non-numeric Part:** A version with an extra non-numeric part is lower: `1.1.a < 1.1`.
- **Special Non-numeric Parts:**
 - `dev` is **lower** than any other non-numeric part: `1.0-dev < 1.0-ALPHA < 1.0-alpha < 1.0-rc`.
 - `rc`, `snapshot`, `final`, `ga`, `release`, and `sp` are **higher** than any other string part, in this order: `1.0-zeta < 1.0-rc < 1.0-snapshot < 1.0-final < 1.0-ga < 1.0-release < 1.0-sp`.
 - These special values are **not case-sensitive** and their ordering does not depend on the separator used: `1.0-RC-1 == 1.0.rc.1`.

Declaring rich versions

When you declare a version using the shorthand notation, then the version is considered a [required version](#):

build.gradle.kts

```
dependencies {
    implementation("org.slf4j:slf4j-api:1.7.15")
}
```

build.gradle

```
dependencies {
```

```
        implementation('org.slf4j:slf4j-api:1.7.15')
    }
```

This means the minimum version will be **1.7.15** and it can be optimistically upgraded by the engine.

To enforce a **strict version** and ensure that only the specified version of a dependency is used, rejecting any other versions even if they would normally be compatible:

build.gradle.kts

```
dependencies {
    implementation("org.slf4j:slf4j-api") {
        version {
            strictly("[1.7, 1.8[")
            prefer("1.7.25")
        }
    }
}
```

build.gradle

```
dependencies {
    implementation('org.slf4j:slf4j-api') {
        version {
            strictly '[1.7, 1.8['
            prefer '1.7.25'
        }
    }
}
```

Gradle supports a model for **rich version** declarations, allowing you to combine different levels of version specificity.

The key terms, listed from strongest to weakest, are:

strictly or !!

This is the strongest version declaration. Any version not matching this notation will be excluded. If used on a declared dependency, **strictly** can downgrade a version. For transitive dependencies, if no acceptable version is found, dependency resolution will fail.

Dynamic versions are supported.

When defined, it overrides any previous `require` declaration and clears any previous `reject` already declared on that dependency.

`require`

This ensures that the selected version cannot be lower than what `require` accepts, but it can be higher through conflict resolution, even if the higher version has an exclusive upper bound. This is the default behavior for a direct dependency.

Dynamic versions are supported.

When defined, it overrides any previous `strictly` declaration and clears any previous `reject` already declared on that dependency.

`prefer`

This is the softest version declaration. It applies only if there is no stronger non-dynamic version specified.

This term does not support dynamic versions and can complement `strictly` or `require`.

When defined, it overrides any previous `prefer` declaration and clears any previous `reject` already declared on that dependency.

Additionally, there is a term outside the hierarchy:

`reject`

This term specifies versions that are not accepted for the module, causing dependency resolution to fail if a rejected version is selected.

Dynamic versions are supported.

Rich version declaration is accessed through the `version` DSL method on a dependency or constraint declaration, which gives you access to [MutableVersionConstraint](#):

build.gradle.kts

```
dependencies {
    implementation("org.slf4j:slf4j-api") {
        version {
            strictly("[1.7, 1.8[")
            prefer("1.7.25")
        }
    }
}

constraints {
    add("implementation", "org.springframework:spring-core") {
        version {
            require("4.2.9.RELEASE")
            reject("4.3.16.RELEASE")
        }
    }
}
```

```

    }
  }
}

```

build.gradle

```

dependencies {
    implementation('org.slf4j:slf4j-api') {
        version {
            strictly '[1.7, 1.8['
            prefer '1.7.25'
        }
    }

    constraints {
        implementation('org.springframework:spring-core') {
            version {
                require '4.2.9.RELEASE'
                reject '4.3.16.RELEASE'
            }
        }
    }
}

```

To enforce [strict versions](#), you can also use the **!!** notation:

build.gradle.kts

```

dependencies {
    // short-hand notation with !!
    implementation("org.slf4j:slf4j-api:1.7.15!!")
    // is equivalent to
    implementation("org.slf4j:slf4j-api") {
        version {
            strictly("1.7.15")
        }
    }

    // or...
    implementation("org.slf4j:slf4j-api:[1.7, 1.8[!!1.7.25")
    // is equivalent to
    implementation("org.slf4j:slf4j-api") {
        version {
            strictly("[1.7, 1.8[")
            prefer("1.7.25")
        }
    }
}

```

```

    }
  }
}

```

build.gradle

```

dependencies {
    // short-hand notation with !!
    implementation('org.slf4j:slf4j-api:1.7.15!!')
    // is equivalent to
    implementation("org.slf4j:slf4j-api") {
        version {
            strictly '1.7.15'
        }
    }

    // or...
    implementation('org.slf4j:slf4j-api:[1.7, 1.8[!!1.7.25')
    // is equivalent to
    implementation('org.slf4j:slf4j-api') {
        version {
            strictly '[1.7, 1.8['
            prefer '1.7.25'
        }
    }
}

```

The notation `[1.7, 1.8[!!1.7.25` above is equivalent to:

- strictly `[1.7, 1.8[`
- prefer `1.7.25`

This means that the engine **must** select a version between `1.7` (included) and `1.8` (excluded). If no other component in the graph needs a different version, it should *prefer* `1.7.25`.

TIP

A strict version cannot be upgraded and overrides any transitive dependency versions, therefore using ranges with strict versions is recommended.

The following table illustrates several use cases:

Which version(s) of this dependency are acceptable?	strictly	require	prefer	rejects	Selection result
Tested with version <code>1.5</code> ; believe all future versions should work.		<code>1.5</code>			Any version starting from <code>1.5</code> , equivalent to <code>org:foo:1.5</code> . An upgrade to <code>2.4</code> is accepted.

Which version(s) of this dependency are acceptable?	strictly	require	prefer	rejects	Selection result
Tested with 1.5, soft constraint upgrades according to semantic versioning.		[1.0, 2.0[1.5		Any version between 1.0 and 2.0, 1.5 if nobody else cares. An upgrade to 2.4 is accepted. □
Tested with 1.5, but follows semantic versioning.	[1.0, 2.0[1.5		Any version between 1.0 and 2.0 (exclusive), 1.5 if nobody else cares. Overwrites versions from transitive dependencies. □
Same as above, with 1.4 known broken.	[1.0, 2.0[1.5	1.4	Any version between 1.0 and 2.0 (exclusive) except for 1.4, 1.5 if nobody else cares. Overwrites versions from transitive dependencies. □
No opinion, works with 1.5.			1.5		1.5 if no other opinion, any otherwise.
No opinion, prefer the latest release.			latest release		The latest release at build time. □
On the edge, latest release, no downgrade.		latest release			The latest release at build time. □
No other version than 1.5.	1.5				1.5, or failure if another strict or higher require constraint disagrees. Overwrites versions from transitive dependencies.
1.5 or a patch version of it exclusively.	[1.5, 1.6[Latest 1.5.x patch release, or failure if another strict or higher require constraint disagrees. Overwrites versions from transitive dependencies. □

Lines annotated with a lock (□) indicate situations where leveraging [dependency locking](#) is recommended. NOTE: When using dependency locking, publishing [resolved versions](#) is always recommended.

Using **strictly** in a library requires careful consideration, as it affects downstream consumers. However, when used correctly, it helps consumers understand which combinations of libraries may be incompatible in their context. For more details, refer to the section on [overriding dependency versions](#).

NOTE

Rich version information is preserved in the Gradle Module Metadata format. However, converting this information to Ivy or Maven metadata formats is lossy.

The highest level of version declaration—**strictly** or **require** over **prefer**—will be published, and any **reject** will be ignored.

Endorsing strict versions

Gradle resolves any dependency version conflicts by selecting the greatest version found in the dependency graph. Some projects might need to divert from the default behavior and enforce an earlier version of a dependency e.g. if the source code of the project depends on an older API of a dependency than some of the external libraries.

In general, forcing dependencies is done to downgrade a dependency. There are common use cases for downgrading:

- A bug was discovered in the latest release.
- Your code depends on an older version that is not binary compatible with the newer one.
- Your code does not use the parts of the library that require a newer version.

WARNING

Forcing a version of a dependency requires careful consideration, as changing the version of a transitive dependency might lead to runtime errors if external libraries expect a different version. It is often better to upgrade your source code to be compatible with newer versions if possible.

Let's say a project uses the **HttpClient library** for performing HTTP calls. **HttpClient** pulls in **Commons Codec** as transitive dependency with version **1.10**. However, the production source code of the project requires an API from **Commons Codec 1.9** which is no longer available in **1.10**. The dependency version can be enforced by declaring it as **strict** it in the build script:

build.gradle.kts

```
dependencies {
    implementation("org.apache.httpcomponents:httpclient:4.5.4")
    implementation("commons-codec:commons-codec") {
        version {
            strictly("1.9")
        }
    }
}
```

build.gradle

```
dependencies {
    implementation 'org.apache.httpcomponents:httpclient:4.5.4'
    implementation('commons-codec:commons-codec') {
        version {
            strictly '1.9'
        }
    }
}
```

```
}  
}  
}
```

Consequences of using strict versions

Using a strict version must be carefully considered:

- **For Library Authors:** Strict versions effectively act like forced versions. They take precedence over transitive dependencies and override any other strict versions found transitively. This could lead to build failures if the consumer project requires a different version.
- **For Consumers:** Strict versions are considered globally during resolution. If a strict version conflicts with a consumer's version requirement, it will trigger a resolution error.

For example, if project **B** *strictly* depends on **C:1.0**, but consumer project A requires **C:1.1**, a resolution error will occur.

To avoid this, it is recommended to use version ranges and a preferred version within those ranges.

For example, **B** might say, instead of *strictly* **1.0**, that it *strictly depends* on the **[1.0, 2.0[** range, but *prefers* **1.0**. Then if a consumer chooses **1.1** (or any other version in the range), the build will *no longer fail*.

Declaring without version

For larger projects, it's advisable to declare **dependencies without versions** and manage versions using [platforms](#):

build.gradle.kts

```
dependencies {  
    implementation("org.springframework:spring-web")  
}  
  
dependencies {  
    constraints {  
        implementation("org.springframework:spring-web:5.0.2.RELEASE")  
    }  
}
```

build.gradle

```
dependencies {  
    implementation 'org.springframework:spring-web'
```

```

}

dependencies {
    constraints {
        implementation 'org.springframework:spring-web:5.0.2.RELEASE'
    }
}

```

This approach centralizes version management, including transitive dependencies.

Declaring dynamic versions

There are many situations where you might need to use the latest version of a specific module dependency or the latest within a range of versions. This is often necessary during development or when creating a library that needs to be compatible with various dependency versions. Projects might adopt a more aggressive approach to consuming dependencies by always integrating the latest version to access cutting-edge features.

You can easily manage these ever-changing dependencies by using a **dynamic version**. A dynamic version can be either a version range (e.g., `2.+`) or a placeholder for the latest available version (e.g., `latest.integration`):

build.gradle.kts

```

plugins {
    `java-library`
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("org.springframework:spring-web:5.+")
}

```

build.gradle

```

plugins {
    id 'java-library'
}

repositories {
    mavenCentral()
}

```

```
dependencies {  
    implementation 'org.springframework:spring-web:5.+'  
}
```

Using dynamic versions and changing modules can lead to unreproducible builds. As new versions of a module are published, its API may become incompatible with your source code. Therefore, use this feature with caution.

CAUTION

For reproducible builds, it's crucial to use [dependency locking](#) when declaring dependencies with dynamic versions. Without this, the module you request may change even for the same version, which is known as a [changing version](#). For example, a Maven **SNAPSHOT** module always points to the latest artifact published, making it a "changing module."

Declaring changing versions

A team may implement a series of features before releasing a new version of the application or library. A common strategy to allow consumers to integrate an unfinished version of their artifacts early is to release a module with a **changing version**. A **changing version** indicates that the feature set is still under active development and hasn't released a stable version for general availability yet.

In Maven repositories, changing versions are commonly referred to as [snapshot versions](#). Snapshot versions contain the suffix **-SNAPSHOT**.

The following example demonstrates how to declare a snapshot version on the Spring dependency:

build.gradle.kts

```
plugins {  
    `java-library`  
}  
  
repositories {  
    mavenCentral()  
    maven {  
        url = uri("https://repo.spring.io/snapshot/")  
    }  
}  
  
dependencies {  
    implementation("org.springframework:spring-web:5.0.3.BUILD-SNAPSHOT")  
}
```

build.gradle

```
plugins {  
    id 'java-library'  
}  
  
repositories {  
    mavenCentral()  
    maven {  
        url = 'https://repo.spring.io/snapshot/'  
    }  
}  
  
dependencies {  
    implementation 'org.springframework:spring-web:5.0.3.BUILD-SNAPSHOT'  
}
```

Gradle is flexible enough to treat any version as a changing version. All you need to do is to set the property `ExternalModuleDependency.setChanging(boolean)` to `true`.

Declaring Dependency Constraints

Dependency constraints function similarly to [dependencies](#), with the key distinction that they do not introduce a dependency themselves. Instead, *constraints define version requirements* that influence the resolution process when a dependency is brought into the project by other means.

Although constraints are not [strict versions](#) by default, you can specify a strict version constraint if needed. Once the dependency is included, the version specified by the constraint participates in conflict resolution just as if it were declared as a direct dependency.

When developing a single-project library, **constraints** can be directly declared alongside direct dependencies. However, when developing multi-project libraries and applications, dependencies are best declared centrally in a [platform](#):

build.gradle.kts

```
plugins {  
    `java-platform`  
}  
  
dependencies {  
    constraints {  
        // Platform declares some versions of libraries used in subprojects  
        api("commons-httpclient:commons-httpclient:3.1")  
        api("org.apache.commons:commons-lang3:3.8.1")  
    }  
}
```

```
}
```

build.gradle

```
plugins {  
    id 'java-platform'  
}  
  
dependencies {  
    constraints {  
        // Platform declares some versions of libraries used in subprojects  
        api 'commons-httpclient:commons-httpclient:3.1'  
        api 'org.apache.commons:commons-lang3:3.8.1'  
    }  
}
```

In general, dependencies are categorized as either **direct** or **transitive**:

- **Direct dependencies** are those explicitly specified within a component's build or metadata.
- **Transitive dependencies** are not directly specified; they are pulled in automatically as dependencies of the direct dependencies.

A component may require both direct and transitive dependencies to compile or run.

Declaring constraints alongside direct dependencies

Dependency constraints allow you to define the version or version range for a specific dependency, *whenever that dependency is encountered during resolution*.

This is the preferred method for managing the version of a component across multiple configurations or projects.

When Gradle resolves a module version, it considers all relevant factors, including [rich versions](#), transitive dependencies, and dependency constraints for that module. The highest version that meets all the conditions is selected. If no such version exists, Gradle will fail with an error, detailing the conflicting declarations.

In such cases, you can adjust your dependency declarations, dependency constraints, or make necessary changes to transitive dependencies.

Like dependency declarations, dependency constraints are [scoped by configurations](#), allowing you to selectively apply them to specific parts of a build.

The `constraints{}` block is used within the `dependencies{}` block to declare these constraints:

build.gradle.kts

```
plugins {
    `java-platform`
}

dependencies {
    constraints {
        api("commons-httpclient:commons-httpclient:3.1")
        runtime("org.postgresql:postgresql:42.2.5")
    }
}
```

build.gradle

```
plugins {
    id 'java-platform'
}

dependencies {
    constraints {
        api 'commons-httpclient:commons-httpclient:3.1'
        runtime 'org.postgresql:postgresql:42.2.5'
    }
}
```

1. `api("commons-httpclient:commons-httpclient:3.1")`:

- This line creates a constraint on the `api` configuration, asserting that if `commons-httpclient` is ever resolved by a resolvable configuration that extends the `api` configuration, its version must be `3.1` or higher.
- If a transitive dependency (a dependency of a dependency) or another module in the project pulls in a different version of `commons-httpclient`, Gradle enforces the dependency to resolve to at least version `3.1`.
- This constraint ensures that the library `commons-httpclient` will be at least version `3.1` across all configuration that extend the `api` configuration.

2. `runtime("org.postgresql:postgresql:42.2.5")`:

- Similarly, this line applies a constraint on the `runtime` configuration, enforcing that `org.postgresql:postgresql` must resolve to at least version `42.2.5`.
- Even if other dependencies or modules within the project try to bring in a different version of `postgresql`, Gradle will choose the higher of `42.2.5` and the other declared versions.
- This ensures that any runtime dependencies on `postgresql` will resolve to at least version

42.2.5 across all resolvable configurations that extend the `runtime` configuration.

Adding constraints on transitive dependencies

Issues with dependency management often arise from transitive dependencies. Developers sometimes mistakenly address these issues by adding direct dependencies instead of handling them properly with constraints.

Dependency constraints allow you to control the selection of transitive dependencies.

In the following example, the version constraint for `commons-codec:1.11` applies only when `commons-codec` is brought in as a transitive dependency since it is not directly declared as a dependency in the project. If `commons-codec` is not pulled in transitively, the constraint has no effect:

build.gradle.kts

```
dependencies {
    implementation("org.apache.httpcomponents:httpclient")
    constraints {
        implementation("org.apache.httpcomponents:httpclient:4.5.3") {
            because("previous versions have a bug impacting this
application")
        }
        implementation("commons-codec:commons-codec:1.11") {
            because("version 1.9 pulled from httpclient has bugs affecting
this application")
        }
    }
}
```

build.gradle

```
dependencies {
    implementation 'org.apache.httpcomponents:httpclient'
    constraints {
        implementation('org.apache.httpcomponents:httpclient:4.5.3') {
            because 'previous versions have a bug impacting this application'
        }
        implementation('commons-codec:commons-codec:1.11') {
            because 'version 1.9 pulled from httpclient has bugs affecting
this application'
        }
    }
}
```

Dependency constraints can also define [rich version](#) constraints and support [strict versions](#), allowing you to enforce a specific version even if it conflicts with a transitive dependency's version (e.g., if a downgrade is necessary).

NOTE

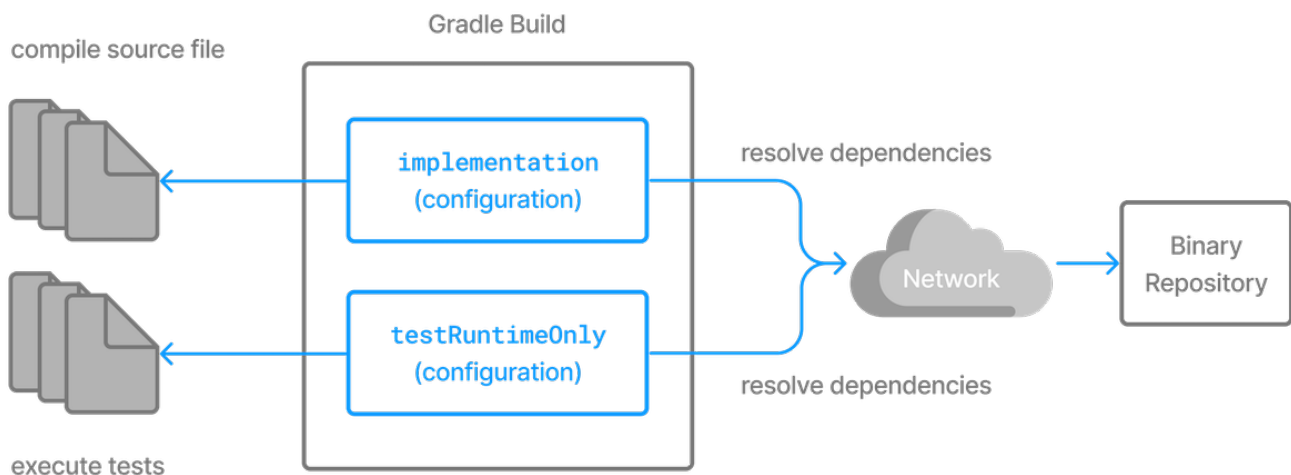
Dependency constraints are only published when using [Gradle Module Metadata](#). This means they are fully supported only when both publishing and consuming modules with Gradle. If modules are consumed with Maven or Ivy, the constraints may not be preserved.

Dependency constraints are transitive. If [library A](#) depends on [library B](#), and [library B](#) declares a constraint on [module C](#), that constraint will affect the version of [module C](#) that [library A](#) depends on.

For example, if [library A](#) depends on [module C version 2](#), but [library B](#) declares a constraint on [module C version 3](#), [library A](#) will resolve [version 3 of module C](#).

Declaring Dependency Configurations

In Gradle, dependencies are associated with specific scopes, such as compile-time or runtime. These scopes are represented by configurations, each identified by a unique name.



Gradle plugins often add pre-defined [configurations](#) to your project.

For example, when applied, the Java plugin adds configurations to your project for source code compilation ([implementation](#)), test execution ([testImplementation](#)), and more ([api](#), [compileOnly](#), [runtimeOnly](#), etc.):

build.gradle.kts

```
plugins {
    `java-library`
}
dependencies {
    implementation("org.hibernate:hibernate-core:3.6.7.Final")
    testImplementation("junit:junit:4.+")
}
```

```
api("com.google.guava:guava:23.0")
}
```

build.gradle

```
plugins {
    id 'java-library'
}
dependencies {
    implementation 'org.hibernate:hibernate-core:3.6.7.Final'
    testImplementation 'junit:junit:4.+'
    api 'com.google.guava:guava:23.0'
}
```

This example highlights dependencies declared on the **implementation**, **testImplementation**, and **api** configuration for a Java project. See the [Java plugin documentation](#) for details.

Resolvable and consumable configurations

Configurations aren't used just for declaring dependencies, they serve various roles in dependency management:

1. **Declaring Dependencies Role:** Configurations that define a set of dependencies.
2. **Consumer Role:** Configurations that are used to resolve dependencies into artifacts.
3. **Producer Role:** Configurations that expose artifacts for consumption by other projects.

1. Configurations for declaring dependencies (i.e, declarable configuration)

To declare dependencies in your project, you can use or create **declarable configurations**. These configurations help organize and categorize dependencies for different parts of the project.

For example, to express a dependency on another project, you would use a **declarable configurations** like **implementation**:

build.gradle.kts

```
dependencies {
    // add a project dependency to the implementation configuration
    implementation(project(":lib"))
}
```

build.gradle

```
dependencies {  
    // add a project dependency to the implementation configuration  
    implementation project(":lib")  
}
```

Configurations used for declaring dependencies define and manage the specific libraries or projects your code requires for tasks such as compilation or testing.

2. Configurations for consumers (i.e, resolvable configuration)

To control how dependencies are resolved and used within your project, you can use or create **resolvable configurations**. These configurations define classpaths and other sets of artifacts that your project needs during different stages, like compilation or runtime.

For example, the **implementation** configuration declares the dependencies, while **compileClasspath** and **runtimeClasspath** are **resolvable configurations** designed for specific purposes. When resolved, they represent the classpaths needed for compilation and runtime, respectively:

build.gradle.kts

```
configurations {  
    // declare a resolvable configuration that is going to resolve the  
    compile classpath of the application  
    resolvable("compileClasspath") {  
        //isCanBeConsumed = false  
        //isCanBeDeclared = false  
        extendsFrom(implementation)  
    }  
}
```

build.gradle

```
configurations {  
    // declare a resolvable configuration that is going to resolve the  
    compile classpath of the application  
    resolvable("compileClasspath") {  
        //canBeConsumed = false  
        //canBeDeclared = false  
        extendsFrom(implementation)  
    }  
}
```

Resolvable configurations are those that can be resolved to produce a set of files or artifacts. These configurations are used to define the classpath for different stages of a build process, such as compilation or runtime.

3. Configurations for producers (i.e., consumable configuration)

Consumable configurations are used to expose artifacts to other projects. These configurations define what parts of your project can be consumed by others, like APIs or runtime dependencies, but are not meant to be resolved directly within your project.

For example, the `exposedApi` configuration is a **consumable configuration** that exposes the API of a component to consumers:

build.gradle.kts

```
configurations {
    // a consumable configuration meant for consumers that need the API of
    this component
    consumable("exposedApi") {
        //isCanBeResolved = false
        //isCanBeDeclared = false
        extendsFrom(implementation)
    }
}
```

build.gradle

```
configurations {
    // a consumable configuration meant for consumers that need the API of
    this component
    consumable("exposedApi") {
        //canBeResolved = false
        //canBeDeclared = false
        extendsFrom(implementation)
    }
}
```

A library typically provides consumable configurations like `apiElements` (for compilation) and `runtimeElements` (for runtime dependencies). These configurations expose the necessary artifacts for other projects to consume, without being resolvable within the current project. The `canBeDeclared`, `isCanBeConsumed` and `isCanBeResolved` flags help distinguish the roles of these configurations.

Configuration flags and roles

Configurations have three key flags:

- **canBeResolved**: Indicates that this configuration is intended for resolving a set of dependencies into a dependency graph. A resolvable configuration should not be declarable or consumable.
- **canBeConsumed**: Indicates that this configuration is intended for exposing artifacts outside this project. A consumable configuration should not be declarable or resolvable.
- **canBeDeclared**: Indicates that this configuration is intended for declaring dependencies. A declarable configuration should not be resolvable or consumable.

TIP Configurations should only have one of these flags enabled.

In short, a configuration's role is determined by the **canBeResolved**, **canBeConsumed**, or **canBeDeclared** flag:

Configuration role	Can be resolved	Can be consumed	Can be declared
Dependency Scope	false	false	true
Resolve for certain usage	true	false	false
Exposed to consumers	false	true	false
Legacy, don't use	true	true	true

For backwards compatibility, the flags have a default value of **true**, but as a plugin author, you should always determine the right values for those flags, or you might accidentally introduce resolution errors.

This example demonstrates how to manually declare the core Java configurations (normally provided by the [Java plugin](#)) in Gradle:

build.gradle.kts

```
// declare a "configuration" named "implementation"
val implementation by configurations.create {
    isCanBeConsumed = false
    isCanBeResolved = false
}

dependencies {
    // add a project dependency to the implementation configuration
    implementation(project(":lib"))
}

configurations {
    // declare a resolvable configuration that is going to resolve the
    compile classpath of the application
    resolvable("compileClasspath") {
```

```

        //isCanBeConsumed = false
        //isCanBeDeclared = false
        extendsFrom(implementation)
    }
    // declare a resolvable configuration that is going to resolve the
runtime classpath of the application
    resolvable("runtimeClasspath") {
        //isCanBeConsumed = false
        //isCanBeDeclared = false
        extendsFrom(implementation)
    }
}

configurations {
    // a consumable configuration meant for consumers that need the API of
this component
    consumable("exposedApi") {
        //isCanBeResolved = false
        //isCanBeDeclared = false
        extendsFrom(implementation)
    }
    // a consumable configuration meant for consumers that need the
implementation of this component
    consumable("exposedRuntime") {
        //isCanBeResolved = false
        //isCanBeDeclared = false
        extendsFrom(implementation)
    }
}
}

```

build.gradle

```

// declare a "configuration" named "implementation"
configurations {
    // declare a "configuration" named "implementation"
    implementation {
        canBeConsumed = false
        canBeResolved = false
    }
}

dependencies {
    // add a project dependency to the implementation configuration
    implementation project(":lib")
}

configurations {
    // declare a resolvable configuration that is going to resolve the
compile classpath of the application
}

```

```

    resolvable("compileClasspath") {
        //canBeConsumed = false
        //canBeDeclared = false
        extendsFrom(implementation)
    }
    // declare a resolvable configuration that is going to resolve the
    runtime classpath of the application
    resolvable("runtimeClasspath") {
        //canBeConsumed = false
        //canBeDeclared = false
        extendsFrom(implementation)
    }
}

configurations {
    // a consumable configuration meant for consumers that need the API of
    this component
    consumable("exposedApi") {
        //canBeResolved = false
        //canBeDeclared = false
        extendsFrom(implementation)
    }
    // a consumable configuration meant for consumers that need the
    implementation of this component
    consumable("exposedRuntime") {
        //canBeResolved = false
        //canBeDeclared = false
        extendsFrom(implementation)
    }
}

```

The following configurations are created:

- **implementation**: Used for declaring project dependencies but neither consumed nor resolved.
- **compileClasspath** + **runtimeClasspath**: Resolvable configurations that collect compile-time and runtime dependencies from **implementation**.
- **exposedApi** + **exposedRuntime**: Consumable configurations that expose artifacts (API and runtime) to other projects, but aren't meant for internal resolution.

This setup mimics the behavior of the **implementation**, **compileClasspath**, **runtimeClasspath**, **apiElements**, and **runtimeElements** configurations in the [Java plugin](#).

Deprecated configurations

In the past, some configurations did not define which role they were intended to be used for.

A deprecation warning is emitted when a configuration is used in a way that was not intended. To fix the deprecation, you will need to stop using the configuration in the deprecated role. The exact

changes required depend on how the configuration is used and if there are alternative configurations that should be used instead.

Creating custom configurations

You can define custom configurations to declare separate scopes of dependencies for specific purposes.

Suppose you want to generate Javadocs with AsciiDoc formatting embedded within your Java source code comments. By setting up the `asciidoclet` configuration, you enable Gradle to use AsciiDoclet, allowing your Javadoc task to produce HTML documentation with enhanced formatting options:

build.gradle.kts

```
val asciidoclet by configurations.creating

dependencies {
    asciidoclet("org.asciidoctor:asciidoclet:1.+")
}

tasks.register("configureJavadoc") {
    doLast {
        tasks.javadoc {
            options.doclet = "org.asciidoctor.Asciidoclet"
            options.docletpath = asciidoclet.files.toList()
        }
    }
}
```

build.gradle

```
configurations {
    asciidoclet
}

dependencies {
    asciidoclet 'org.asciidoctor:asciidoclet:1.+'
}
```

You can manage custom configurations using the `configurations` block. Configurations must have names and can extend each other. For more details, refer to the `ConfigurationContainer` API.

NOTE

Configurations are intended to be used for a single role: declaring dependencies,

performing resolution, or defining consumable variants.

There are three main use cases for creating custom configurations:

1. **API/Implementation Separation:** Create custom configurations to separate API dependencies (exposed to consumers) from implementation dependencies (used internally during compilation or runtime).
 - You might create an `api` configuration for libraries that consumers will depend on, and an `implementation` configuration for libraries that are only needed internally. The `api` configuration is typically consumed by downstream projects, while `implementation` dependencies are hidden from consumers but used internally.
 - This separation ensures that your project maintains clean boundaries between its public API and strictly internal mechanisms.
2. **Resolvable Configuration Creation:** Create a custom resolvable configuration to resolve specific sets of dependencies, like classpaths, at various build stages.
 - You might create a `compileClasspath` configuration that resolves only the dependencies needed to compile your project. Similarly, you could create a `runtimeClasspath` configuration to resolve the dependencies needed to run the project at runtime.
 - This allows fine-grained control over which dependencies are available during different build phases, such as compilation or testing.
3. **Consumable Configuration from Dependency Configuration:** Create a custom consumable configuration to expose artifacts or dependencies for other projects to consume, typically when your project produces artifacts like JARs.
 - You might create an `exposedApi` configuration to expose the API dependencies of your project for consumption by other projects. Similarly, a `runtimeElements` configuration could be created to expose the runtime dependencies or artifacts that other projects need.
 - Consumable configurations ensure that only the necessary artifacts or dependencies are exposed to consumers.

Configuration API incubating methods

Several incubating factory methods—`resolvable()`, `consumable()`, and `dependencyScope()`—within the `ConfigurationContainer` API can be used to simplify the creation of configurations with specific roles.

These methods help build authors document the purpose of a configuration and avoid manually setting various configuration flags, streamlining the process and ensuring consistency:

- `resolvable()`: Creates a configuration intended for resolving dependencies. This means the configuration can be used to resolve dependencies but not consumed by other projects.
- `consumable()`: Creates a configuration meant to be consumed by other projects but not used to resolve dependencies itself.
- `dependencyScope()`: Creates a configuration that establishes a dependency scope, setting up the necessary properties to act both as a consumer and provider, depending on the use case.

Configuration inheritance

Configurations can inherit from other configurations, creating an inheritance hierarchy.

Configurations form an inheritance hierarchy using the `Configuration.extendsFrom(Configuration...)` method. A configuration can extend any other configuration other than a `detached configuration`, regardless of how it is defined in the build script or plugin.

TIP

Avoid extending consumable or resolvable configurations with configurations that are not consumable or resolvable, respectively.

Configurations can only extend configurations within the same project.

When extending a configuration, the new configuration inherits:

- dependencies
- dependency constraints
- exclude rules
- artifacts
- capabilities

The extension does **not** include attributes. It also does **not** extend consumable/resolvable/declarable status.

Dependency resolution

The entrypoint to all dependency resolution APIs is a *resolvable* `Configuration`. The Java plugins primarily use the `compileClasspath`, and `runtimeClasspath` configurations to resolve jars for compilation and runtime respectively.

A resolvable configuration is intended for initiating dependency resolution. The dependencies to be resolved are declared on *dependency scope* configurations. The Java plugins use the `api`, `implementation`, and `runtimeOnly` dependency scope configurations, among others, as a source of dependencies to be resolved by the resolvable configurations.

Consider the following example that demonstrates how to declare a set of configurations intended for resolution:

NOTE

This example uses incubating APIs.

build.gradle.kts

```
val implementation = configurations.dependencyScope("implementation")
val runtimeClasspath = configurations.resolvable("runtimeClasspath") {
    extendsFrom(implementation.get())
}
```

build.gradle

```
configurations {  
    dependencyScope("implementation")  
    resolvable("runtimeClasspath") {  
        extendsFrom(implementation)  
    }  
}
```

Dependencies can be declared on the `implementation` configuration using the `dependencies` block. See the [Declaring Dependencies](#) chapter for more information on the types of dependencies that can be declared, and the various options for customizing dependency declarations.

build.gradle.kts

```
dependencies {  
    implementation("com.google.guava:guava:33.2.1-jre")  
}
```

build.gradle

```
dependencies {  
    implementation("com.google.guava:guava:33.2.1-jre")  
}
```

Now that we've created a *dependency scope* configuration for declaring dependencies, and a *resolvable* configuration for resolving those dependencies, we can use Gradle's [dependency resolution APIs](#) to access the results of resolution.

Unsafe configuration resolution errors

Resolving a configuration can have side effects on Gradle's project model. As a result, Gradle must manage access to each project's configurations.

There are a number of ways a configuration might be resolved unsafely. For example:

- A task from one project directly resolves a configuration in another project in the task's action.
- A task specifies a configuration from another project as an input file collection.
- A build script for one project resolves a configuration in another project during evaluation.
- Project configurations are resolved in the settings file.

Gradle produces a deprecation warning for each unsafe access.

Unsafe access can cause indeterminate errors. You should [fix unsafe access warnings](#) in your build.

In most cases, you can resolve unsafe accesses by creating a [proper cross-project dependency](#).

DECLARING REPOSITORIES

Declaring Repositories Basics

Gradle can resolve local or external dependencies from one or many repositories based on Maven, Ivy or flat directory formats.

Repositories intended for use in a single project are declared in your `build.gradle(.kts)` file:

build.gradle.kts

```
repositories {  
    mavenCentral()  
    maven {  
        url = uri("https://repo.spring.io/snapshot/")  
    }  
}
```

build.gradle

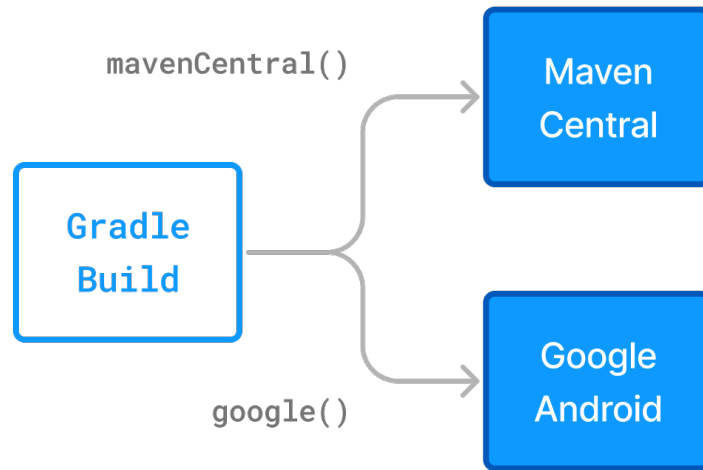
```
repositories {  
    mavenCentral()  
    maven {  
        url = 'https://repo.spring.io/snapshot/'  
    }  
}
```

To centralize repository declarations in your `settings.gradle(.kts)` file, head over to [Centralizing Repository Declarations](#).

Declaring a publicly-available repository

Organizations building software may want to leverage public binary repositories to download and consume publicly available dependencies. Popular public repositories include [Maven Central](#) and the [Google Android](#) repository.

Gradle provides built-in shorthand notations for these widely-used repositories.



Under the covers, Gradle resolves dependencies from the respective URL of the public repository defined by the shorthand notation. All shorthand notations are available via the [RepositoryHandler](#) API.

Alternatively, you can [explicitly specify the URL of the repository](#) for more fine-grained control.

Maven Central repository

Maven Central is a popular repository hosting open source libraries for consumption by Java projects.

To declare the [Maven Central repository](#) for your build add this to your script:

build.gradle.kts

```
repositories {  
    mavenCentral()  
}
```

build.gradle

```
repositories {  
    mavenCentral()  
}
```

Google Maven repository

The Google repository hosts Android-specific artifacts including the Android SDK. For usage examples, see the [relevant Android documentation](#).

To declare the [Google Maven repository](#) add this to your build script:

build.gradle.kts

```
repositories {  
    google()  
}
```

build.gradle

```
repositories {  
    google()  
}
```

Declaring a custom repository by URL

Most enterprise projects set up a binary repository available only within an intranet. In-house repositories enable teams to publish internal binaries, setup user management and security measures, and ensure uptime and availability.

Specifying a custom URL is also helpful if you want to declare publicly-available repository that Gradle does not provide a shorthand for.

Repositories with custom URLs can be specified as Maven or Ivy repositories by calling the corresponding methods available on the [RepositoryHandler](#) API:

build.gradle.kts

```
repositories {  
    maven {  
        url = uri("http://repo.mycompany.com/maven2")  
    }  
}
```

build.gradle

```
repositories {  
    maven {  
        url = "http://repo.mycompany.com/maven2"  
    }  
}
```

Gradle supports additional [protocols](#) beyond [http](#) and [https](#), such as [file](#), [sftp](#), and [s3](#) for custom URLs.

For full coverage, see the section on [supported repository types](#).

You can also [define your own repository layout](#) by using `ivy { }` repositories, as they are very flexible in terms of how modules are organised in a repository:

build.gradle.kts

```
repositories {
    ivy {
        url = uri("http://repo.mycompany.com/repo")
    }
}
```

build.gradle

```
repositories {
    ivy {
        url = "http://repo.mycompany.com/repo"
    }
}
```

Declaring multiple repositories

You can define more than one repository for resolving dependencies. Declaring multiple repositories is helpful if some dependencies are only available in one repository but not the other.

You can mix any type of repository described in the [reference section](#).

build.gradle.kts

```
repositories {
    mavenCentral()
    maven {
        url = uri("https://repo.spring.io/release")
    }
    maven {
        url = uri("https://repository.jboss.org/maven2")
    }
}
```

build.gradle

```
repositories {  
    mavenCentral()  
    maven {  
        url = "https://repo.spring.io/release"  
    }  
    maven {  
        url = "https://repository.jboss.org/maven2"  
    }  
}
```

The order of repository declaration determines the order that Gradle will search for dependencies during resolution. If Gradle finds a dependency's metadata in a particular repository, it will attempt to download all the artifacts for that module from *the same repository*.

You can learn more about the inner workings of [dependency downloads](#).

Plugin repositories

Gradle uses a different set of repositories for resolving Gradle plugins and resolving project dependencies:

1. **Plugin dependencies:** When resolving plugins for build scripts, Gradle uses a distinct set of repositories to locate and load the required plugins.
2. **Project dependencies:** When resolving project dependencies, Gradle only uses the repositories declared in the build script and ignores the plugin repositories.

By default, Gradle uses the [Gradle Plugin Portal](#) to search for plugins:

settings.gradle.kts

```
pluginManagement {  
    repositories {  
        mavenCentral()  
        gradlePluginPortal()  
    }  
}
```

settings.gradle

```
pluginManagement {  
    repositories {  
        mavenCentral()  
    }  
}
```

```
        gradlePluginPortal()
    }
}
```

However, some plugins may be hosted in other repositories (public or private). To include these plugins, you need to specify additional repositories in your build script so Gradle knows where to search.

Since declaring repositories depends on how the plugin is applied, refer to the [Custom Plugin Repositories](#) for more details on configuring repositories for plugins from different sources.

Centralizing Repository Declarations

Instead of declaring repositories in every subproject of your build or via an `allprojects` block, Gradle provides a way to declare them centrally for all projects.

NOTE | Central declaration of repositories is an incubating feature.

You can declare repositories that will be used by convention in every subproject in the `settings.gradle(.kts)` file:

settings.gradle.kts

```
dependencyResolutionManagement {
    repositories {
        mavenCentral()
    }
}
```

settings.gradle

```
dependencyResolutionManagement {
    repositories {
        mavenCentral()
    }
}
```

The `dependencyResolutionManagement` repositories block accepts the same notations as in a project, including Maven or Ivy repositories, with or without credentials.

Repositories mode

By default, repositories declared in a project's `build.gradle(.kts)` file will override those declared in `settings.gradle(.kts)`. However, you can control this behavior using the `repositoriesMode` setting:

settings.gradle.kts

```
dependencyResolutionManagement {  
    repositoriesMode = RepositoriesMode.PREFER_PROJECT  
}
```

settings.gradle

```
dependencyResolutionManagement {  
    repositoriesMode = RepositoriesMode.PREFER_PROJECT  
}
```

[[sec:available-modes] == Available modes

There are three modes for dependency resolution management:

Mode	Description	Default?	Use-Case
PREFER_PROJECT	Repositories declared in a project override those declared in <code>settings.gradle(.kts)</code> .	Yes	Useful when teams need to use different repositories specific to their subprojects.
PREFER_SETTINGS	Repositories declared in <code>settings.gradle(.kts)</code> override those declared in a project.	No	Useful for enforcing the use of approved repositories across large teams.
FAIL_ON_PROJECT_REPOS	Declaring a repository in a project triggers a build error.	No	Strictly enforces the use of repositories declared in <code>settings.gradle(.kts)</code> .

You can change the behavior to prefer the repositories in `settings.gradle(.kts)`:

settings.gradle.kts

```
dependencyResolutionManagement {  
    repositoriesMode = RepositoriesMode.PREFER_SETTINGS  
}
```

settings.gradle

```
dependencyResolutionManagement {  
    repositoriesMode = RepositoriesMode.PREFER_SETTINGS  
}
```

Gradle will warn you if a project or plugin declares a repository when using this mode.

To enforce that **only** repositories declared in `settings.gradle(.kts)` are used, you can configure Gradle to fail the build when a project plugin is declared:

settings.gradle.kts

```
dependencyResolutionManagement {  
    repositoriesMode = RepositoriesMode.FAIL_ON_PROJECT_REPOS  
}
```

settings.gradle

```
dependencyResolutionManagement {  
    repositoriesMode = RepositoriesMode.FAIL_ON_PROJECT_REPOS  
}
```

Repository Types

Gradle supports various sources for resolving dependencies, accommodating different metadata formats and connectivity methods. You can resolve dependencies from:

- [Maven-compatible artifact repositories](#) (e.g., Maven Central)
- [Ivy-compatible artifact repositories](#) (including custom layouts)
- [Local \(flat\) directories](#)

Maven repositories

Many organizations host dependencies in Maven repositories. Gradle can declare Maven repositories by specifying their URL:

build.gradle.kts

```
repositories {  
    maven {  
        url = uri("http://repo.mycompany.com/maven2")  
    }  
}
```

build.gradle

```
repositories {  
    maven {  
        url = "http://repo.mycompany.com/maven2"  
    }  
}
```

Composite Maven repository

Sometimes, POMs are published in one location, and JARs in another. You can define such a repository as follows:

build.gradle.kts

```
repositories {  
    maven {  
        // Look for POMs and artifacts, such as JARs, here  
        url = uri("http://repo2.mycompany.com/maven2")  
        // Look for artifacts here if not found at the above location  
        artifactUrls("http://repo.mycompany.com/jars")  
        artifactUrls("http://repo.mycompany.com/jars2")  
    }  
}
```

build.gradle

```
repositories {
```

```

maven {
    // Look for POMs and artifacts, such as JARs, here
    url = "http://repo2.mycompany.com/maven2"
    // Look for artifacts here if not found at the above location
    artifactUrls "http://repo.mycompany.com/jars"
    artifactUrls "http://repo.mycompany.com/jars2"
}
}

```

Gradle will first look for POMs and artifacts at the base URL, and if the artifact is not found, it will check the additional `artifactUrls`.

Authenticated Maven repository

You can specify credentials for Maven repositories that require authentication. See [Supported Repository Protocols](#) for authentication options.

Local Maven repository

Gradle can consume dependencies from a local Maven repository, that is repositories on the local file system:

build.gradle.kts

```

repositories {
    maven {
        url = uri(layout.buildDirectory.dir("repo"))
    }
}

```

build.gradle

```

repositories {
    maven {
        url = uri(layout.buildDirectory.dir("repo"))
    }
}

```

Gradle can consume dependencies from the [local Maven repository](#). This is useful for teams that want to test their setup locally before publishing their plugin.

You should ensure that using the local Maven repository is necessary before adding `mavenLocal()` to your build script:

build.gradle.kts

```
repositories {  
    mavenLocal()  
}
```

build.gradle

```
repositories {  
    mavenLocal()  
}
```

NOTE

Gradle manages its own [cache](#) and doesn't need to declare the local Maven repository even if you resolve dependencies from a remote Maven repository.

Gradle uses the same logic as Maven to identify the location of your local Maven cache. If a [settings.xml](#) file is defined in the user's home directory ([~/.m2/settings.xml](#)), this location takes precedence over [M2_HOME/conf](#). Otherwise, Gradle defaults to [~/.m2/repository](#).

TIP

As a general recommendation, avoid using [mavenLocal\(\)](#). Unlike Maven builds, Gradle can share artifacts between projects using project dependencies. Publishing to the local maven repo is not necessary for sharing artifacts between projects.

Ivy repositories

Many organizations host dependencies in Ivy repositories.

Standard layout Ivy repository

To declare an Ivy repository with the standard layout, simply specify the URL:

build.gradle.kts

```
repositories {  
    ivy {  
        url = uri("http://repo.mycompany.com/repo")  
    }  
}
```

build.gradle

```
repositories {  
    ivy {  
        url = "http://repo.mycompany.com/repo"  
    }  
}
```

Named layout Ivy repository

You can specify that your repository follows the Ivy default layout:

build.gradle.kts

```
repositories {  
    ivy {  
        url = uri("http://repo.mycompany.com/repo")  
        layout("maven")  
    }  
}
```

build.gradle

```
repositories {  
    ivy {  
        url = "http://repo.mycompany.com/repo"  
        layout "maven"  
    }  
}
```

Valid named layout values are **gradle** (default), **maven**, and **ivy**. Refer to [IvyArtifactRepository.layout\(java.lang.String\)](#) in the API documentation for more details.

Custom pattern layout Ivy repository

To define an Ivy repository with a non-standard layout, you can set up a pattern layout:

build.gradle.kts

```
repositories {
```

```

    ivy {
        url = uri("http://repo.mycompany.com/repo")
        patternLayout {
            artifact("[module]/[revision]/[type]/[artifact].[ext]")
        }
    }
}

```

build.gradle

```

repositories {
    ivy {
        url = "http://repo.mycompany.com/repo"
        patternLayout {
            artifact "[module]/[revision]/[type]/[artifact].[ext]"
        }
    }
}

```

For an Ivy repository that fetches Ivy files and artifacts from different locations, define separate patterns:

build.gradle.kts

```

repositories {
    ivy {
        url = uri("http://repo.mycompany.com/repo")
        patternLayout {
            artifact("3rd-party-
artifacts/[organisation]/[module]/[revision]/[artifact]-[revision].[ext]")
            artifact("company-
artifacts/[organisation]/[module]/[revision]/[artifact]-[revision].[ext]")
            ivy("ivy-files/[organisation]/[module]/[revision]/ivy.xml")
        }
    }
}

```

build.gradle

```

repositories {
    ivy {
        url = "http://repo.mycompany.com/repo"
        patternLayout {

```

```

        artifact "3rd-party-
artifacts/[organisation]/[module]/[revision]/[artifact]-[revision].[ext]"
        artifact "company-
artifacts/[organisation]/[module]/[revision]/[artifact]-[revision].[ext]"
        ivy "ivy-files/[organisation]/[module]/[revision]/ivy.xml"
    }
}
}

```

Optionally, you can enable Maven-style layout for the 'organisation' part, with forward slashes replacing dots:

build.gradle.kts

```

repositories {
    ivy {
        url = uri("http://repo.mycompany.com/repo")
        patternLayout {
            artifact("[organisation]/[module]/[revision]/[artifact]-
[revision].[ext]")
            setM2compatible(true)
        }
    }
}

```

build.gradle

```

repositories {
    ivy {
        url = "http://repo.mycompany.com/repo"
        patternLayout {
            artifact "[organisation]/[module]/[revision]/[artifact]-
[revision].[ext]"
            m2compatible = true
        }
    }
}

```

Authenticated Ivy repository

You can specify credentials for Ivy repositories that require authentication. See [Supported Repository Protocols](#) for authentication options.

Local Ivy repository

Gradle can consume dependencies from a local Ivy repository, that is repositories on the local file system:

build.gradle.kts

```
repositories {
    ivy {
        // URL can refer to a local directory
        url = uri("../local-repo")
    }
}
```

build.gradle

```
repositories {
    ivy {
        // URL can refer to a local directory
        url = file("../local-repo")
    }
}
```

Flat directory repository

Some projects store dependencies on a shared drive or within the project's source code rather than using a binary repository. To use a flat filesystem directory as a repository, you can configure it like this:

build.gradle.kts

```
repositories {
    flatDir {
        dirs("lib")
    }
    flatDir {
        dirs("lib1", "lib2")
    }
}
```

build.gradle

```
repositories {  
    flatDir {  
        dirs 'lib'  
    }  
    flatDir {  
        dirs 'lib1', 'lib2'  
    }  
}
```

This configuration adds repositories that search specified directories for dependencies.

NOTE

Flat directory repositories are discouraged, as they do not support metadata formats like Ivy XML or Maven POM files.

In general, binary dependencies should be sourced from an external repository, but if storing dependencies externally is not an option, prefer declaring a Maven or Ivy repository using a local file URL instead.

When resolving dependencies from a flat dir repo, Gradle dynamically generates adhoc dependency metadata based on the presence of artifacts. Gradle prefers modules with real metadata over those generated by flat directory repositories. For this reason, flat directories cannot override artifacts with real metadata from other declared repositories.

For instance, if Gradle finds `jmxri-1.2.1.jar` in a flat directory and `jmxri-1.2.1.pom` in another repository, it will use the metadata from the latter.

Metadata Formats

Dependency metadata refers to the information associated with a dependency that describes its characteristics, relationships, and requirements.

This metadata includes details such as:

1. **Identity:** Module dependencies are uniquely identified by their group, name, and version (GAV) coordinates.
2. **Dependencies:** A list of other binaries that this dependency requires, including their versions.
3. **Variants:** Different forms of the component (e.g., compile, runtime, apiElements, runtimeElements) that can be consumed in different contexts.
4. **Artifacts:** The actual files (like JARs, ZIPs, etc.) produced by the component, which may include compiled code, resources, or documentation.
5. **Capabilities:** Describes the functionality or features that a module provides or consumes, helping to avoid conflicts when different modules provide the same capability.

6. **Attributes:** Key-value pairs used to differentiate between variants (e.g. `org.gradle.jvm.version:8`).

Depending on the repository type, dependency metadata are stored in different formats:

- **Gradle:** Gradle Module Metadata (`.module`) files
- **Maven:** Maven POM (`pom.xml`) files
- **Ivy:** Ivy Descriptor (`ivy.xml`) files

Some repositories may contain multiple types of metadata for a single component. When Gradle publishes to a Maven repository, it publishes both a Gradle Module Metadata (GMM) files and a Maven POM file.

This metadata plays a crucial role in dependency resolution, by allowing the dependencies of binary artifacts to be tracked alongside the artifact itself. By reading dependency metadata, Gradle is able to determine which versions of other artifacts a given dependency requires.

Supported metadata formats

External module dependencies require module metadata so that Gradle can determine the transitive dependencies of a module. Gradle supports various metadata formats to achieve this.

Gradle Module Metadata (GMM) files

Gradle Module Metadata is specifically designed to support all features of Gradle's dependency management model, making it the preferred format.

You can find the [specification here](#).

```
{
  "formatVersion": "1.1",
  "component": {
    "group": "com.example",
    "module": "my-library",
    "version": "1.0"
  }
}
```

POM files

Gradle natively supports [Maven POM files](#). By default, Gradle will first look for a POM file. However, if the file contains a special marker, Gradle will use [Gradle Module Metadata](#) instead.

```
<project xmlns="http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>my-library</artifactId>
  <version>1.0</version>
```

```
</project>
```

Ivy files

Gradle also supports [Ivy descriptor files](#). Gradle will first look for an `ivy.xml` file, but if this file contains a special marker, it will use [Gradle Module Metadata](#) instead.

```
<ivy-module version="2.0">
  <info organisation="com.example" module="my-library" revision="1.0"/>
  <dependencies>
    <dependency org="org.example" name="dependency" rev="1.2"/>
  </dependencies>
</ivy-module>
```

Supported metadata sources

When searching for a component in a repository, Gradle checks for supported metadata file formats by default.

Gradle first looks for `.module` (Gradle module metadata) files. In a Maven repository, Gradle then looks for `.pom` files. In an Ivy repository, it checks for `ivy.xml` files. And in a flat directory repository, it looks directly for `.jar` files without expecting any metadata.

If you define a custom repository, you can configure how Gradle searches for metadata. For instance, you can set up a Maven repository will optionally resolve JARs that don't have associated POM files. This is done by configuring metadata sources for the repository:

build.gradle.kts

```
repositories {
    maven {
        url = uri("http://repo.mycompany.com/repo")
        metadataSources {
            mavenPom()
            artifact()
        }
    }
}
```

build.gradle

```
repositories {
    maven {
        url = "http://repo.mycompany.com/repo"
        metadataSources {
```

```

        mavenPom()
        artifact()
    }
}
}

```

You can specify multiple metadata sources, and Gradle will search through them in a predefined order. The following metadata sources are supported:

Metadata source	Description	Default Order	Maven	Ivy / flat dir
<code>gradleMetadata()</code>	Look for Gradle <code>.module</code> files	1st	yes	yes
<code>mavenPom()</code>	Look for Maven <code>.pom</code> files	2nd	yes	yes
<code>ivyDescriptor()</code>	Look for <code>ivy.xml</code> files	2nd	no	yes
<code>artifact()</code>	Look directly for artifact without associated metadata	3rd	yes	yes

By default, Gradle will require that a dependency has associated metadata.

To relax this requirement and allow Gradle to resolve artifacts without associated metadata, specify the `artifact` metadata source:

```

mavenCentral {
    metadataSources {
        mavenPom()
        artifact()
    }
}

```

The above example instructs Gradle to first look for component metadata from a POM file, and if not present, to derive metadata from the artifact itself.

When parsing metadata files (Ivy or Maven), Gradle checks for a marker that indicates the presence of a matching Gradle Module Metadata file. If found, Gradle will prefer the Gradle metadata.

To disable this behavior, use the `ignoreGradleMetadataRedirection()` option:

build.gradle.kts

```

repositories {
    maven {
        url = uri("http://repo.mycompany.com/repo")
        metadataSources {

```

```

        mavenPom()
        artifact()
        ignoreGradleMetadataRedirection()
    }
}

```

build.gradle

```

repositories {
    maven {
        url = "http://repo.mycompany.com/repo"
        metadataSources {
            mavenPom()
            artifact()
            ignoreGradleMetadataRedirection()
        }
    }
}

```

Supported Protocols

Gradle supports a variety of transport protocols for Maven and Ivy repositories.

Supported transport protocols

These protocols determine how Gradle communicates with the repositories to resolve dependencies.

Type	Credential types	Link
file	none	
http	username/password	Documentation
https	username/password	Documentation
sftp	username/password	Documentation
s3	access key/secret key/session token or Environment variables	Documentation
gcs	default application credentials sourced from well known files, Environment variables etc.	Documentation

NOTE

Usernames and passwords should never be stored in plain text in your build files. Instead, store credentials in a local `gradle.properties` file or use an open-source Gradle plugin for encrypting and consuming credentials, such as the [credentials plugin](#).

The transport protocol is specified as part of the repository URL.

Below are examples of how to declare repositories using various protocols:

Example 15. [Declaring HTTP-based Maven and Ivy Repositories](#)

build.gradle.kts

```
repositories {  
    maven {  
        url = uri("http://repo.mycompany.com/maven2")  
    }  
  
    ivy {  
        url = uri("http://repo.mycompany.com/repo")  
    }  
}
```

build.gradle

```
repositories {  
    maven {  
        url = "http://repo.mycompany.com/maven2"  
    }  
  
    ivy {  
        url = "http://repo.mycompany.com/repo"  
    }  
}
```

Example 16. [Declaring an SFTP Protocol for a Repository](#)

build.gradle.kts

```
repositories {  
    maven {  
        url = uri("sftp://repo.mycompany.com:22/maven2")  
        credentials {  
            username = "user"  
            password = "password"  
        }  
    }  
  
    ivy {  
        url = uri("sftp://repo.mycompany.com:22/repo")  
    }  
}
```

```

        credentials {
            username = "user"
            password = "password"
        }
    }
}

```

build.gradle

```

repositories {
    maven {
        url = "sftp://repo.mycompany.com:22/maven2"
        credentials {
            username = "user"
            password = "password"
        }
    }

    ivy {
        url = "sftp://repo.mycompany.com:22/repo"
        credentials {
            username = "user"
            password = "password"
        }
    }
}

```

Example 17. Declaring an S3-Backed Maven and Ivy Repository

build.gradle.kts

```

repositories {
    maven {
        url = uri("s3://myCompanyBucket/maven2")
        credentials(AwsCredentials::class) {
            accessKey = "someKey"
            secretKey = "someSecret"
            // optional
            sessionToken = "someSTSToken"
        }
    }

    ivy {
        url = uri("s3://myCompanyBucket/ivyrepo")
        credentials(AwsCredentials::class) {
            accessKey = "someKey"

```

```

        secretKey = "someSecret"
        // optional
        sessionToken = "someSTSToken"
    }
}

```

build.gradle

```

repositories {
    maven {
        url = "s3://myCompanyBucket/maven2"
        credentials(AwsCredentials) {
            accessKey = "someKey"
            secretKey = "someSecret"
            // optional
            sessionToken = "someSTSToken"
        }
    }

    ivy {
        url = "s3://myCompanyBucket/ivyrepo"
        credentials(AwsCredentials) {
            accessKey = "someKey"
            secretKey = "someSecret"
            // optional
            sessionToken = "someSTSToken"
        }
    }
}

```

Example 18. Declaring an S3-Backed Maven and Ivy Repository Using IAM

build.gradle.kts

```

repositories {
    maven {
        url = uri("s3://myCompanyBucket/maven2")
        authentication {
            create<AwsImAuthentication>("awsIm") // load from EC2 role or env
        }
    }

    ivy {
        url = uri("s3://myCompanyBucket/ivyrepo")
    }
}

```

```

        authentication {
            create<AwsImAuthentication>("awsIm")
        }
    }
}

```

build.gradle

```

repositories {
    maven {
        url = "s3://myCompanyBucket/maven2"
        authentication {
            awsIm(AwsImAuthentication) // load from EC2 role or env var
        }
    }

    ivy {
        url = "s3://myCompanyBucket/ivyrepo"
        authentication {
            awsIm(AwsImAuthentication)
        }
    }
}

```

Example 19. Declaring a GCS-Backed Maven and Ivy Repository

build.gradle.kts

```

repositories {
    maven {
        url = uri("gcs://myCompanyBucket/maven2")
    }

    ivy {
        url = uri("gcs://myCompanyBucket/ivyrepo")
    }
}

```

build.gradle

```

repositories {
    maven {
        url = "gcs://myCompanyBucket/maven2"
    }
}

```

```

    }

    ivy {
        url = "gcs://myCompanyBucket/ivyrepo"
    }
}

```

Configuring authentication schemes

HTTP(S) authentication schemes configuration

When configuring a repository that uses HTTP or HTTPS transport protocols, several authentication schemes are available. By default, Gradle attempts to use all schemes supported by the [Apache HttpClient library](#). However, you may want to explicitly specify which authentication schemes should be used when interacting with a remote server. When explicitly declared, only those specified schemes will be used.

Basic authentication

You can specify credentials for Maven repositories secured by basic authentication using [PasswordCredentials](#):

build.gradle.kts

```

repositories {
    maven {
        url = uri("http://repo.mycompany.com/maven2")
        credentials {
            username = "user"
            password = "password"
        }
    }
}

```

build.gradle

```

repositories {
    maven {
        url = "http://repo.mycompany.com/maven2"
        credentials {
            username = "user"
            password = "password"
        }
    }
}

```

```
}
```

Digest Authentication

To configure a repository to use only [DigestAuthentication](#):

build.gradle.kts

```
repositories {  
    maven {  
        url = uri("https://repo.mycompany.com/maven2")  
        credentials {  
            username = "user"  
            password = "password"  
        }  
        authentication {  
            create<DigestAuthentication>("digest")  
        }  
    }  
}
```

build.gradle

```
repositories {  
    maven {  
        url = 'https://repo.mycompany.com/maven2'  
        credentials {  
            username = "user"  
            password = "password"  
        }  
        authentication {  
            digest(DigestAuthentication)  
        }  
    }  
}
```

Supported Authentication Schemes

BasicAuthentication

Basic access authentication over HTTP. Credentials are sent preemptively.

DigestAuthentication

Digest access authentication over HTTP.

HttpHeaderAuthentication

Authentication based on a custom HTTP header, such as private tokens or OAuth tokens.

Using preemptive authentication

By default, Gradle submits credentials only when a server responds with an authentication challenge (HTTP 401). However, some servers might respond with a different code (e.g., GitHub returns a 404) that could cause dependency resolution to fail. In such cases, you can configure Gradle to send credentials preemptively by explicitly using the [BasicAuthentication](#) scheme:

build.gradle.kts

```
repositories {
    maven {
        url = uri("https://repo.mycompany.com/maven2")
        credentials {
            username = "user"
            password = "password"
        }
        authentication {
            create<BasicAuthentication>("basic")
        }
    }
}
```

build.gradle

```
repositories {
    maven {
        url = 'https://repo.mycompany.com/maven2'
        credentials {
            username = "user"
            password = "password"
        }
        authentication {
            basic(BasicAuthentication)
        }
    }
}
```

Using HTTP header authentication

For Maven repositories that require token-based, OAuth2, or other HTTP header-based authentication, you can use [HttpHeaderCredentials](#) and [HttpHeaderAuthentication](#):

build.gradle.kts

```
repositories {
    maven {
        url = uri("http://repo.mycompany.com/maven2")
        credentials(HttpHeaderCredentials::class) {
            name = "Private-Token"
            value = "TOKEN"
        }
        authentication {
            create<HttpHeaderAuthentication>("header")
        }
    }
}
```

build.gradle

```
repositories {
    maven {
        url = "http://repo.mycompany.com/maven2"
        credentials(HttpHeaderCredentials) {
            name = "Private-Token"
            value = "TOKEN"
        }
        authentication {
            header(HttpHeaderAuthentication)
        }
    }
}
```

AWS S3 repositories configuration

When configuring a repository that uses AWS S3, several options and settings are available.

S3 configuration properties

The following system properties can be used to configure interactions with S3 repositories:

org.gradle.s3.endpoint

Overrides the AWS S3 endpoint when using a non-AWS, S3 API-compatible storage service.

`org.gradle.s3.maxErrorRetry`

Specifies the maximum number of retry attempts when the S3 server responds with an HTTP 5xx status code. The default value is 3 if not specified.

S3 URL formats

S3 URLs must use the 'virtual-hosted-style' format:

```
s3://<bucketName>[.<regionSpecificEndpoint>]/<s3Key>
```

Example: `s3://myBucket.s3.eu-central-1.amazonaws.com/maven/release`

- `myBucket`: The AWS S3 bucket name.
- `s3.eu-central-1.amazonaws.com`: The optional region-specific endpoint.
- `/maven/release`: The AWS S3 key (a unique identifier for an object within a bucket).

S3 proxy settings

A proxy for S3 can be configured using the following system properties:

- For HTTPS:
 - `https.proxyHost`
 - `https.proxyPort`
 - `https.proxyUser`
 - `https.proxyPassword`
 - `http.nonProxyHosts` (NOTE: this is not a typo.) *For HTTP (if `org.gradle.s3.endpoint` is set with an HTTP URI):
 - `http.proxyHost`
 - `http.proxyPort`
 - `http.proxyUser`
 - `http.proxyPassword`
 - `http.nonProxyHosts`

S3 V4 Signatures (AWS4-HMAC-SHA256)

Some S3 regions (e.g., `eu-central-1` in Frankfurt) require that all HTTP requests are signed using AWS's signature version 4. It is recommended to specify S3 URLs containing the region-specific endpoint when using buckets that require V4 signatures:

```
s3://somebucket.s3.eu-central-1.amazonaws.com/maven/release
```

If the region-specific endpoint is not specified for buckets requiring V4 Signatures, Gradle defaults to the `us-east-1` region and will issue a warning:

Attempting to re-send the request to with AWS V4 authentication. To avoid this warning in the future, use region-specific endpoint to access buckets located in regions that require V4 signing.

Failing to specify the region-specific endpoint for such buckets results in:

- **Increased network traffic:** Three round-trips to AWS per file upload/download instead of one.
- **Slower builds:** Due to increased network latency.
- **Higher transmission failure rates:** Due to additional network overhead.

S3 Cross Account Access

In organizations with multiple AWS accounts (e.g., one per team), the bucket owner may differ from the artifact publisher or consumers. To ensure consumers can access the artifacts, the bucket owner must grant the appropriate access. Gradle automatically applies the `bucket-owner-full-control` Canned ACL to uploaded objects. Ensure the publisher has the required IAM permissions (`PutObjectAcl` and `PutObjectVersionAcl` if bucket versioning is enabled), either directly or through an assumed IAM Role. For more details, see [AWS S3 Access Permissions](#).

Google Cloud Storage repositories configuration

When configuring a repository that uses Google Cloud Storage (GCS), several configuration options and settings are available.

GCS configuration properties

You can use the following system properties to configure interactions with [GCS](#) repositories:

`org.gradle.gcs.endpoint`

Overrides the Google Cloud Storage endpoint, useful when working with a storage service compatible with the GCS API but not hosted on Google Cloud Platform.

`org.gradle.gcs.servicePath`

Specifies the root service path from which the GCS client builds requests, with a default value of `/`.

GCS URL formats

GCS URLs use a 'virtual-hosted-style' format and must adhere to the following structure:

```
gcs://<bucketName>/<objectKey>
```

- `<bucketName>`: The name of the Google Cloud Storage bucket.
- `<objectKey>`: The unique identifier for an object within a bucket.

Example: `gcs://myBucket/maven/release`

- **myBucket**: The bucket name.
- **/maven/release**: The GCS object key.

Handling credentials

Repository credentials should never be hardcoded in your build script but kept external. Gradle provides an [API in artifact repositories](#) that allows you to declare the type of credentials required, with their values being looked up from [Gradle properties](#) during the build.

For example, consider the following repository configuration:

build.gradle.kts

```
repositories {  
    maven {  
        name = "mySecureRepository"  
        credentials(PasswordCredentials::class)  
        // url = uri(<<some repository url>>)  
    }  
}
```

build.gradle

```
repositories {  
    maven {  
        name = 'mySecureRepository'  
        credentials(PasswordCredentials)  
        // url = uri(<<some repository url>>)  
    }  
}
```

In this example, the username and password are automatically looked up from properties named **mySecureRepositoryUsername** and **mySecureRepositoryPassword**.

Configuration property prefix

The configuration property prefix, known as the **identity**, is derived from the repository name. Credentials can be provided through any of the supported Gradle property mechanisms: **gradle.properties** file, command-line arguments, environment variables, or a combination of these.

Conditional credential requirement

Credentials are only required when the build process needs them. For example, if a project is configured to publish artifacts to a secured repository, but the publishing task isn't invoked, Gradle

won't require the credentials. However, if a task requiring credentials is part of the build process, Gradle will check for their presence before running any tasks to prevent build failures due to missing credentials.

Supported credential types

Lookup is only supported for the credential types listed in the table below:

Type	Argument	Base property name	Required?
PasswordCredentials	username	Username	required
	password	Password	required
AwsCredentials	accessKey	AccessKey	required
	secretKey	SecretKey	required
	sessionToken	SessionToken	optional
HttpHeaderCredentials	name	AuthHeaderName	required
	value	AuthHeaderValue	required

Filtering Repository Content

Gradle exposes an API to declare what a repository may or may not contain. There are different use cases for it:

- **Performance** when you know a dependency will never be found in a specific repository
- **Security** by avoiding leaking what dependencies are used in a private project
- **Reliability** when some repositories contain invalid or incorrect metadata or artifacts

It's even more important when considering that the declared order of repositories matter.

Declaring a repository filter

build.gradle.kts

```
repositories {  
    maven {  
        url = uri("https://repo.mycompany.com/maven2")  
        content {  
            // this repository *only* contains artifacts with group  
            "my.company"  
            includeGroup("my.company")  
        }  
    }  
    mavenCentral {  
        content {  
            // this repository contains everything BUT artifacts with group
```

```

        starting with "my.company"
        excludeGroupByRegex("my\\.company.*")
    }
}
}

```

build.gradle

```

repositories {
    maven {
        url = "https://repo.mycompany.com/maven2"
        content {
            // this repository only contains artifacts with group
            "my.company"
            includeGroup "my.company"
        }
    }
    mavenCentral {
        content {
            // this repository contains everything BUT artifacts with group
            starting with "my.company"
            excludeGroupByRegex "my\\.company.*"
        }
    }
}
}

```

By default, repositories include everything and exclude nothing:

- If you declare an include, then it excludes everything *but* what is included.
- If you declare an exclude, then it includes everything *but* what is excluded.
- If you declare both includes and excludes, then it includes only what is explicitly included and not excluded.

It is possible to filter either by explicit *group*, *module* or *version*, either strictly or using regular expressions. When using a strict version, it is possible to use a version range, using [the format supported](#) by Gradle. In addition, there are filtering options by resolution context: configuration name or even configuration attributes. See [RepositoryContentDescriptor](#) for details.

Declaring content exclusively found in one repository

Filters declared using the [repository-level content filter](#) are not exclusive. This means that declaring that a repository *includes* an artifact doesn't mean that the other repositories can't have it either: you must declare what every repository contains in extension.

Alternatively, Gradle provides an API which lets you declare that a repository *exclusively includes*

an artifact. If you do so:

- an artifact declared in a repository *can't* be found in any other
- exclusive repository content must be declared in extension (just like for [repository-level content](#))

build.gradle.kts

```
repositories {
    // This repository will _not_ be searched for artifacts in my.company
    // despite being declared first
    mavenCentral()
    exclusiveContent {
        forRepository {
            maven {
                url = uri("https://repo.mycompany.com/maven2")
            }
        }
        filter {
            // this repository *only* contains artifacts with group
            "my.company"
            includeGroup("my.company")
        }
    }
}
```

build.gradle

```
repositories {
    // This repository will _not_ be searched for artifacts in my.company
    // despite being declared first
    mavenCentral()
    exclusiveContent {
        forRepository {
            maven {
                url = "https://repo.mycompany.com/maven2"
            }
        }
        filter {
            // this repository *only* contains artifacts with group
            "my.company"
            includeGroup "my.company"
        }
    }
}
```

It is possible to filter either by explicit *group*, *module* or *version*, either strictly or using regular expressions. See [InclusiveRepositoryContentDescriptor](#) for details.

NOTE

If you leverage exclusive content filtering in the [pluginManagement](#) [section of the settings.gradle\(.kts\)](#), it becomes illegal to add more repositories through the project `buildscript.repositories`. In that case, the build configuration will fail.

Your options are either to declare all repositories in settings or to use non-exclusive content filtering.

Maven repository filtering

For [Maven repositories](#), it's often the case that a repository would either contain releases or snapshots. Gradle lets you declare what kind of artifacts are found in a repository using this DSL:

build.gradle.kts

```
repositories {
    maven {
        url = uri("https://repo.mycompany.com/releases")
        mavenContent {
            releasesOnly()
        }
    }
    maven {
        url = uri("https://repo.mycompany.com/snapshots")
        mavenContent {
            snapshotsOnly()
        }
    }
}
```

build.gradle

```
repositories {
    maven {
        url = "https://repo.mycompany.com/releases"
        mavenContent {
            releasesOnly()
        }
    }
    maven {
        url = "https://repo.mycompany.com/snapshots"
        mavenContent {
            snapshotsOnly()
        }
    }
}
```

}

CENTRALIZING DEPENDENCIES

Platforms

Platforms are used to ensure that all dependencies in a project align with a consistent set of versions.

Platforms help you manage and enforce version consistency across different modules or libraries, especially when you are working with a set of related dependencies that need to be kept in sync.

Using a platform

A platform is a specialized software component used to control transitive dependency versions. Typically, it consists of [dependency constraints](#) that either recommend or enforce specific versions. Platforms are particularly useful when you need to share consistent dependency versions across multiple projects.

In a typical setup you have:

- **A Platform Project:** Which defines constraints for dependencies used across different subprojects.
- **A Number of Subprojects:** Which depend on the platform and declare dependencies without specifying versions.

The `java-platform` plugin supports creating platforms in the Java ecosystem. Platforms are also commonly published as Maven BOMs (Bill of Materials), which Gradle natively supports.

To use a platform, declare a dependency with the `platform` keyword:

build.gradle.kts

```
dependencies {  
    // get recommended versions from the platform project  
    api(platform(project(":platform")))  
    // no version required  
    api("commons-httpclient:commons-httpclient")  
}
```

build.gradle

```
dependencies {  
    // get recommended versions from the platform project  
    api platform(project(':platform'))  
    // no version required  
    api 'commons-httpclient:commons-httpclient'
```

```
}
```

This notation automatically performs several actions:

- Sets the `org.gradle.category` attribute to `platform`, ensuring Gradle selects the platform component.
- Enables the `endorseStrictVersions` behavior by default, enforcing strict versions defined in the platform.

If `strict version` enforcement isn't needed, you can disable it using the `doNotEndorseStrictVersions` method.

Creating a platform

In Java projects, the `java-platform` plugin combined with `dependency constraints` can be used to create a platform:

```
plugins {  
    id("java-platform")  
}  
  
dependencies {  
    constraints {  
        api("com.google.guava:guava:30.1-jre")  
        api("org.apache.commons:commons-lang3:3.12.0")  
    }  
}
```

This defines a custom platform with specific versions of `guava` and `commons-lang3` that can be applied in other projects.

Importing a platform

Gradle supports importing `BOMs`, which are POM files containing `<dependencyManagement>` sections that manage dependency versions.

In order to qualify as a BOM, a `.pom` file needs to have `pom` set. This means that the POM file should explicitly specify `<packaging>pom</packaging>` in its metadata.

Gradle treats all entries in the block of a BOM similar to [Adding Constraints On Dependencies](#).

Regular Platform

To import a BOM, declare a dependency on it using the `platform` dependency modifier method:

build.gradle.kts

```
dependencies {  
    // import a BOM  
    implementation(platform("org.springframework.boot:spring-boot-  
dependencies:1.5.8.RELEASE"))  
    // define dependencies without versions  
    implementation("com.google.code.gson:gson")  
    implementation("dom4j:dom4j")  
}
```

build.gradle

```
dependencies {  
    // import a BOM  
    implementation platform('org.springframework.boot:spring-boot-  
dependencies:1.5.8.RELEASE')  
    // define dependencies without versions  
    implementation 'com.google.code.gson:gson'  
    implementation 'dom4j:dom4j'  
}
```

In this example, the Spring Boot BOM provides the versions for `gson` and `dom4j`, so no explicit versions are needed.

Enforced Platform

The `enforcedPlatform` keyword can be used to override any versions found in the dependency graph, but should be used with caution as it is effectively transitive and exports forced versions to all consumers of your project:

build.gradle.kts

```
dependencies {  
    // import a BOM. The versions used in this file will override any other  
    version found in the graph  
    implementation(enforcedPlatform("org.springframework.boot:spring-boot-  
dependencies:1.5.8.RELEASE"))  
  
    // define dependencies without versions  
    implementation("com.google.code.gson:gson")  
    implementation("dom4j:dom4j")  
}
```

```
// this version will be overridden by the one found in the BOM
implementation("org.codehaus.groovy:groovy:1.8.6")
}
```

build.gradle

```
dependencies {
    // import a BOM. The versions used in this file will override any other
    // version found in the graph
    implementation enforcedPlatform('org.springframework.boot:spring-boot-
dependencies:1.5.8.RELEASE')

    // define dependencies without versions
    implementation 'com.google.code.gson:gson'
    implementation 'dom4j:dom4j'

    // this version will be overridden by the one found in the BOM
    implementation 'org.codehaus.groovy:groovy:1.8.6'
}
```

When using `enforcedPlatform`, exercise caution if your software component is intended for consumption by others. This declaration is transitive and affects the dependency graph of your consumers. If they disagree with any enforced versions, they'll need to use `exclude`. Instead, if your reusable component strongly favors specific third-party dependency versions, consider using a `rich version` declaration with `strictly`.

Version Catalogs

A version catalog is a selected list of dependencies that can be referenced in build scripts, simplifying dependency management.

Instead of specifying dependencies directly using string notation, you can pick them from a version catalog:

build.gradle.kts

```
dependencies {
    implementation(libs.groovy.core)
}
```

build.gradle

```
dependencies {  
    implementation(libs.groovy.core)  
}
```

In this example, **libs** represents the catalog, and **groovy** is a dependency available in it.

Where the version catalog defining **libs.groovy.core** is a **libs.versions.toml** file in the **gradle** directory:

gradle/libs.versions.toml

```
[libraries]  
groovy-core = { group = "org.codehaus.groovy", name = "groovy", version = "3.0.5" }
```

Version catalogs offer several advantages:

- **Type-Safe Accessors:** Gradle generates type-safe accessors for each catalog, enabling autocompletion in IDEs.
- **Centralized Version Management:** Each catalog is visible to all projects in a build.
- **Dependency Bundles:** Catalogs can group commonly used dependencies into [bundles](#).
- **Version Separation:** Catalogs can separate dependency coordinates from version information, allowing shared version declarations.
- **Conflict Resolution:** Like regular dependency notation, version catalogs declare requested [versions](#) but do not enforce them during [conflict resolution](#).

While version catalogs define versions, they don't influence the dependency resolution process. Gradle may still select different versions due to dependency graph conflicts or constraints applied through platforms or dependency management APIs.

WARNING

Versions declared in a catalog are typically not enforced, meaning the actual version used in the build may differ based on dependency resolution.

Accessing a catalog

To access items in a version catalog defined in the standard **libs.versions.toml** file located in the **gradle** directory, you use the **libs** object in your build scripts. For example, to reference a library, you can use **libs.<alias>**, and for a plugin, you can use **libs.plugins.<alias>**.

Declaring dependencies using a version catalog:

build.gradle.kts

```
dependencies {  
    implementation(libs.groovy.core)  
    implementation(libs.groovy.json)  
    implementation(libs.groovy.nio)  
}
```

build.gradle

```
dependencies {  
    implementation libs.groovy.core  
    implementation libs.groovy.json  
    implementation libs.groovy.nio  
}
```

Is the same as:

build.gradle.kts

```
dependencies {  
    implementation("org.codehaus.groovy:groovy:3.0.5")  
    implementation("org.codehaus.groovy:groovy-json:3.0.5")  
    implementation("org.codehaus.groovy:groovy-nio:3.0.5")  
}
```

build.gradle

```
dependencies {  
    implementation 'org.codehaus.groovy:groovy:3.0.5'  
    implementation 'org.codehaus.groovy:groovy-json:3.0.5'  
    implementation 'org.codehaus.groovy:groovy-nio:3.0.5'  
}
```

Accessors map directly to the aliases and versions defined in the TOML file, offering type-safe access to dependencies and plugins. This enables IDEs to provide autocompletion, highlight typos, and identify missing dependencies as errors.

Aliases and type-safe accessors

Aliases in a version catalog consist of identifiers separated by a dash (-), underscore (_), or dot (.). Type-safe accessors are generated for each alias, normalized to dot notation:

Example aliases	Generated accessors
guava	libs.guava
groovy-core	libs.groovy.core
androidx.awesome.lib	libs.androidx.awesome.lib

Creating a catalog

Version catalogs are conventionally declared using a `libs.versions.toml` file located in the `gradle` subdirectory of the root build:

gradle/libs.versions.toml

```
[versions]
groovy = "3.0.5"
checkstyle = "8.37"

[libraries]
groovy-core = { module = "org.codehaus.groovy:groovy", version.ref = "groovy" }
groovy-json = { module = "org.codehaus.groovy:groovy-json", version.ref = "groovy" }
groovy-nio = { module = "org.codehaus.groovy:groovy-nio", version.ref = "groovy" }
commons-lang3 = { group = "org.apache.commons", name = "commons-lang3",
version = { strictly = "[3.8, 4.0[" , prefer="3.9" } }

[bundles]
groovy = ["groovy-core", "groovy-json", "groovy-nio"]

[plugins]
versions = { id = "com.github.ben-manes.versions", version = "0.45.0" }
```

The TOML catalog format

The [TOML](#) file has four sections:

- `[versions]` – Declares version identifiers.
- `[libraries]` – Maps aliases to GAV coordinates.
- `[bundles]` – Defines dependency bundles.
- `[plugins]` – Declares plugin versions.

The TOML file format is very lenient and lets you write "dotted" properties as shortcuts to full object declarations.

Versions

Versions can be declared either as a single string, in which case they are interpreted as a *required* version, or as a [rich version](#):

```
[versions]
other-lib = "5.5.0" # Required version
my-lib = { strictly = "[1.0, 2.0[" , prefer = "1.2" } # Rich version
```

Supported members of a version declaration are:

- **require**: the [required version](#)
- **strictly**: the [strict version](#)
- **prefer**: the [preferred version](#)
- **reject**: the list of [rejected versions](#)
- **rejectAll**: a boolean to reject all [versions](#)

Libraries

Each library is mapped to a GAV coordinate: group, artifact, version. They can be declared as a simple string, in which case they are interpreted coordinates, or separate group and name:

```
[versions]
common = "1.4"

[libraries]
my-lib = "com.mycompany:mylib:1.4"
my-lib-no-version.module = "com.mycompany:mylib"
my-other-lib = { module = "com.mycompany:other", version = "1.4" }
my-other-lib2 = { group = "com.mycompany", name = "alternate", version = "1.4" }
mylib-full-format = { group = "com.mycompany", name = "alternate", version = { require = "1.4" } }

[plugins]
short-notation = "some.plugin.id:1.4"
long-notation = { id = "some.plugin.id", version = "1.4" }
reference-notation = { id = "some.plugin.id", version.ref = "common" }
```

You can also define strict or preferred versions using **strictly** or **prefer**:

```
[libraries]
commons-lang3 = { group = "org.apache.commons", name = "commons-lang3", version = { strictly = "[3.8, 4.0[" , prefer = "3.9" } }
```

In case you want to reference a version declared in the `[versions]` section, use the `version.ref` property:

```
[versions]
some = "1.4"

[libraries]
my-lib = { group = "com.mycompany", name="mylib", version.ref="some" }
```

Bundles

Bundles group multiple library aliases, so they can be referenced together in the build script.

```
[versions]
groovy = "3.0.9"

[libraries]
groovy-core = { group = "org.codehaus.groovy", name = "groovy", version.ref = "groovy"
}
groovy-json = { group = "org.codehaus.groovy", name = "groovy-json", version.ref =
"groovy" }
groovy-nio = { group = "org.codehaus.groovy", name = "groovy-nio", version.ref =
"groovy" }

[bundles]
groovy = ["groovy-core", "groovy-json", "groovy-nio"]
```

This is useful for pulling in several related dependencies with a single alias:

build.gradle.kts

```
dependencies {
    implementation(libs.bundles.groovy)
}
```

build.gradle

```
dependencies {
    implementation libs.bundles.groovy
}
```

Plugins

This section defines the plugins and their versions by mapping plugin IDs to version numbers. Just like libraries, you can define plugin versions using aliases from the [\[versions\]](#) section or directly specify the version.

```
[plugins]
versions = { id = "com.github.ben-manes.versions", version = "0.45.0" }
```

Which can be accessed in any project of the build using the `plugins {}` block. To refer to a plugin from the catalog, use the `alias()` function:

build.gradle.kts

```
plugins {
    `java-library`
    checkstyle
    alias(libs.plugins.versions)
}
```

build.gradle

```
plugins {
    id 'java-library'
    id 'checkstyle'
    // Use the plugin 'versions' as declared in the 'libs' version catalog
    alias(libs.plugins.versions)
}
```

WARNING

You cannot use a plugin declared in a version catalog in your settings file or settings plugin.

Avoiding subgroup accessors

To avoid generating subgroup accessors, use camelCase notation:

Aliases	Accessors
<code>groovyCore</code>	<code>libs.groovyCore</code>
<code>groovyJson-core</code>	<code>libs.groovyJson.core</code>

Reserved keywords

Certain keywords, like `extensions`, `class`, and `convention`, are reserved and cannot be used as aliases. Additionally, `bundles`, `versions`, and `plugins` cannot be the first subgroup in a dependency alias.

For example, the alias `versions-dependency` is not valid, but `versionsDependency` or `dependency-versions` are valid.

Publishing a catalog

In most cases, the `gradle/libs.versions.toml` will be checked into a repository and available for consumption.

However, this doesn't always solve the problem of sharing a catalog in an organization or for external consumers. Another option to share a catalog is to write a settings plugin, publish it on the Gradle plugin portal or an internal repository, and let the consumers apply the plugin on their settings file.

Alternatively, Gradle offers a *version catalog* plugin, which has the ability to declare and publish a catalog.

To do this, you need to apply the `version-catalog` plugin:

build.gradle.kts

```
plugins {  
    `version-catalog`  
    `maven-publish`  
}
```

build.gradle

```
plugins {  
    id 'version-catalog'  
    id 'maven-publish'  
}
```

This plugin will then expose the `catalog extension` that you can use to declare a catalog:

build.gradle.kts

```
catalog {  
    // declare the aliases, bundles and versions in this block  
    versionCatalog {
```

```

        library("my-lib", "com.mycompany:mylib:1.2")
    }
}

```

build.gradle

```

catalog {
    // declare the aliases, bundles and versions in this block
    versionCatalog {
        library('my-lib', 'com.mycompany:mylib:1.2')
    }
}

```

The plugin must be created programmatically, see [Programming catalogs](#) for details.

Such a catalog can then be published by applying either the `maven-publish` or `ivy-publish` plugin and configuring the publication to use the `versionCatalog` component:

build.gradle.kts

```

publishing {
    publications {
        create<MavenPublication>("maven") {
            from(components["versionCatalog"])
        }
    }
}

```

build.gradle

```

publishing {
    publications {
        maven(MavenPublication) {
            from components.versionCatalog
        }
    }
}

```

When publishing such a project, a `libs.versions.toml` file will automatically be generated (and uploaded), which can then be [consumed from other Gradle builds](#).

Importing a published catalog

A catalog produced by the [Version Catalog Plugin](#) can be imported via the [Settings API](#):

settings.gradle.kts

```
dependencyResolutionManagement {
    versionCatalogs {
        create("libs") {
            from("com.mycompany:catalog:1.0")
        }
    }
}
```

settings.gradle

```
dependencyResolutionManagement {
    versionCatalogs {
        libs {
            from("com.mycompany:catalog:1.0")
        }
    }
}
```

Importing a catalog from a file

IMPORTANT

Gradle automatically imports a catalog in the **gradle** directory named **libs.versions.toml**.

The [version catalog builder API](#) allows importing a catalog from an external file, enabling reuse across different parts of a build, such as sharing the main build's catalog with **buildSrc**.

For example, you can include a catalog in the **buildSrc/settings.gradle(.kts)** file as follows:

settings.gradle.kts

```
dependencyResolutionManagement {
    versionCatalogs {
        create("libs") {
            from(files("../gradle/libs.versions.toml"))
        }
    }
}
```

```
}
```

settings.gradle

```
dependencyResolutionManagement {
    versionCatalogs {
        libs {
            from(files("../gradle/libs.versions.toml"))
        }
    }
}
```

The [VersionCatalogBuilder.from\(Object dependencyNotation\)](#) method accepts only a single file, meaning that notations like [Project.files\(java.lang.Object...\)](#) must refer to one file. Otherwise, the build will fail.

TIP

Remember that you don't need to import the version catalog named `libs.versions.toml` if it resides in your `gradle` folder. It will be imported automatically.

However, if you need to import version catalogs from multiple files, it's recommended to use a code-based approach instead of relying on TOML files. This approach allows for the declaration of multiple catalogs from different files:

settings.gradle.kts

```
dependencyResolutionManagement {
    versionCatalogs {
        // declares an additional catalog, named 'testLibs', from the 'test-
        libs.versions.toml' file
        create("testLibs") {
            from(files("gradle/test-lib.versions.toml"))
        }
    }
}
```

settings.gradle

```
dependencyResolutionManagement {
    versionCatalogs {
        // declares an additional catalog, named 'testLibs', from the 'test-
        libs.versions.toml' file
        testLibs {
```

```
        from(files('gradle/test-libs.versions.toml'))
    }
}
}
```

Importing multiple catalogs

You can declare multiple catalogs to organize dependencies better by using the [Settings API](#):

settings.gradle.kts

```
dependencyResolutionManagement {
    versionCatalogs {
        create("libs") {
            from(files("gradle/libs.versions.toml"))
        }
        create("tools") {
            from(files("gradle/tools.versions.toml"))
        }
    }
}
```

build.gradle.kts

```
dependencies {
    implementation(libs.someDependency)
    implementation(tools.someTool)
}
```

NOTE

To minimize the risk of naming conflicts, each catalog generates an extension applied to all projects, so it's advisable to choose a unique name. One effective approach is to select a name that ends with **Libs**.

Changing the catalog name

By default, the **libs.versions.toml** file is used as input for the **libs** catalog. However, you can rename the default catalog if an extension with the same name already exists:

settings.gradle.kts

```
dependencyResolutionManagement {
    defaultLibrariesExtensionName = "projectLibs"
}
```

settings.gradle

```
dependencyResolutionManagement {  
    defaultLibrariesExtensionName = 'projectLibs'  
}
```

Overwriting catalog versions

You can overwrite versions when importing a catalog:

settings.gradle.kts

```
dependencyResolutionManagement {  
    versionCatalogs {  
        create("amendedLibs") {  
            from("com.mycompany:catalog:1.0")  
            // overwrite the "groovy" version declared in the imported  
catalog  
            version("groovy", "3.0.6")  
        }  
    }  
}
```

settings.gradle

```
dependencyResolutionManagement {  
    versionCatalogs {  
        amendedLibs {  
            from("com.mycompany:catalog:1.0")  
            // overwrite the "groovy" version declared in the imported  
catalog  
            version("groovy", "3.0.6")  
        }  
    }  
}
```

In the examples above, any dependency referencing the **groovy** version will automatically be updated to use **3.0.6**.

NOTE

Overwriting a version only affects what is imported and used when declaring dependencies. The actual resolved dependency version may differ due to conflict

Programming catalogs

Version catalogs can be declared programmatically in the `settings.gradle(.kts)` file.

settings.gradle.kts

```
dependencyResolutionManagement {
    versionCatalogs {
        create("libs") {
            version("groovy", "3.0.5")
            version("checkstyle", "8.37")
            library("groovy-core", "org.codehaus.groovy",
"groovy").versionRef("groovy")
            library("groovy-json", "org.codehaus.groovy", "groovy-
json").versionRef("groovy")
            library("groovy-nio", "org.codehaus.groovy", "groovy-
nio").versionRef("groovy")
            library("commons-lang3", "org.apache.commons", "commons-
lang3").version {
                strictly("[3.8, 4.0[")
                prefer("3.9")
            }
        }
    }
}
```

settings.gradle

```
dependencyResolutionManagement {
    versionCatalogs {
        libs {
            version('groovy', '3.0.5')
            version('checkstyle', '8.37')
            library('groovy-core', 'org.codehaus.groovy', 'groovy')
.versionRef('groovy')
            library('groovy-json', 'org.codehaus.groovy', 'groovy-json')
.versionRef('groovy')
            library('groovy-nio', 'org.codehaus.groovy', 'groovy-nio')
.versionRef('groovy')
            library('commons-lang3', 'org.apache.commons', 'commons-lang3')
.version {
                strictly '[3.8, 4.0['
                prefer '3.9'
            }
        }
    }
}
```

```
}  
}
```

TIP

Don't use `libs` for your programmatic version catalog name if you have the default `libs.versions.toml` in your project.

Creating a version catalog programmatically uses the [Settings API](#):

settings.gradle.kts

```
dependencyResolutionManagement {  
    versionCatalogs {  
        create("libs") {  
            version("groovy", "3.0.5")  
            version("checkstyle", "8.37")  
            library("groovy-core", "org.codehaus.groovy",  
"groovy").versionRef("groovy")  
            library("groovy-json", "org.codehaus.groovy", "groovy-  
json").versionRef("groovy")  
            library("groovy-nio", "org.codehaus.groovy", "groovy-  
nio").versionRef("groovy")  
            library("commons-lang3", "org.apache.commons", "commons-  
lang3").version {  
                strictly("[3.8, 4.0[")  
                prefer("3.9")  
            }  
            bundle("groovy", listOf("groovy-core", "groovy-json", "groovy-  
nio"))  
        }  
    }  
}
```

settings.gradle

```
dependencyResolutionManagement {  
    versionCatalogs {  
        libs {  
            version('groovy', '3.0.5')  
            version('checkstyle', '8.37')  
            library('groovy-core', 'org.codehaus.groovy', 'groovy')  
            .versionRef('groovy')  
            library('groovy-json', 'org.codehaus.groovy', 'groovy-json')  
            .versionRef('groovy')  
            library('groovy-nio', 'org.codehaus.groovy', 'groovy-nio')  
            .versionRef('groovy')
```

```

        library('commons-lang3', 'org.apache.commons', 'commons-lang3')
    .version {
        strictly '[3.8, 4.0['
        prefer '3.9'
    }
    bundle('groovy', ['groovy-core', 'groovy-json', 'groovy-nio'])
}
}
}

```

Using Catalogs with Platforms

Both platforms and version catalogs help manage dependency versions in a project, but they serve different purposes and have different effects on dependency resolution:

Version Catalogs

- **Purpose:** A version catalog centralizes and standardizes dependency coordinates (group, name, version) and provides type-safe accessors in the build script, making dependencies easier to manage.
- **Effect on Dependency Graph:** Version catalogs **do not** directly affect dependency resolution. The versions defined in the catalog must be explicitly referenced in a **dependencies** block, and once referenced, they behave the same as any locally declared dependency. Additionally, the catalog's contents are **transparent to downstream consumers**, meaning that consumers cannot identify whether a dependency was declared locally or sourced from a catalog.

libs.versions.toml

```

[libraries]
mylib = { group = "com.example", name = "mylib", version = "1.0.0" }

```

Platforms

- **Purpose:** A platform is a module in the dependency graph that enforces or aligns versions of dependencies (including transitive dependencies). It influences dependency resolution and ensures version consistency across different modules.
- **Effect on Dependency Graph:** Platforms **apply** or **enforce** versions to dependencies that are declared locally without versions. These versions in a platform are propagated through the dependency graph, affecting transitive dependencies and downstream consumers. They are a formal part of the dependency graph and can dictate the version chosen during resolution.

build.gradle.kts

```

plugins {
    `java-platform`
}

```

```
}

dependencies {
    constraints {
        api("com.example:mylib:2.0.0")
    }
}
```

Using a catalog with a platform

Even if a version catalog defines a version for a dependency, Gradle might pick a different version during resolution if another component (e.g., a platform or a transitive dependency) suggests a different version (unless `enforcedPlatform` is used).

For example, a version catalog may define `mylib` as version `1.0.0`, but if a platform enforces `2.0.0`, Gradle will select version `2.0.0`.

To ensure consistent version alignment, a good approach is to use a version catalog to define dependency versions alongside a platform to enforce them.

Version Catalog:

gradle/libs.versions.toml

```
[versions]
junit-jupiter = "5.10.3"

[libraries]
guava = { module = "com.google.guava:guava" }
junit-jupiter = { module = "org.junit.jupiter:junit-jupiter", version.ref = "junit-jupiter" }
junit-jupiter-launcher = { module = "org.junit.platform:junit-platform-launcher" }
```

Platform:

platform/build.gradle.kts

```
plugins {
    `java-platform`
}

javaPlatform {
    allowDependencies()
}
```

```
dependencies {
    constraints {
        api("org.junit.jupiter:junit-jupiter:5.11.1") // Enforcing specific
version
        api("com.google.guava:guava:[33.1.0-jre,)") // Enforcing version
range
    }
}
```

platform/build.gradle

```
plugins {
    id 'java-platform'
}

javaPlatform {
    allowDependencies()
}

dependencies {
    constraints {
        api 'org.junit.jupiter:junit-jupiter:5.11.1' // Enforcing specific
version
        api 'com.google.guava:guava:[33.1.0-jre,)' // Enforcing version range
    }
}
```

Consumer:

consumer/build.gradle.kts

```
dependencies {
    // Platform
    implementation(platform(project(":platform")))
    // Catalog
    testImplementation(libs.junit.jupiter)
    testRuntimeOnly(libs.junit.jupiter.launcher)
    implementation(libs.guava)
}
```

consumer/build.gradle

```
dependencies {  
    // Platform  
    implementation platform(project(":platform"))  
    // Catalog  
    testImplementation libs.junit.jupiter  
    testRuntimeOnly libs.junit.jupiter.launcher  
    implementation libs.guava  
}
```

Best Practices for using both a catalog and a platform:

1. **Use version catalogs** for defining and sharing dependency coordinates across projects. They make dependency declarations consistent and easier to manage but do not guarantee version alignment.
2. **Use platforms** when you need to influence or enforce version alignment across modules. Platforms ensure that dependencies resolve to the desired version, particularly in large or multi-module projects.

MANAGING DEPENDENCIES

Locking Versions

Using [dynamic dependency versions](#) (e.g., `1.+` or `[1.0,2.0)`) can cause builds to break unexpectedly because the exact version of a dependency that gets resolved can change over time:

build.gradle.kts

```
dependencies {  
    // Depend on the latest 5.x release of Spring available in the searched  
    repositories  
        implementation("org.springframework:spring-web:5.+")  
}
```

build.gradle

```
dependencies {  
    // Depend on the latest 5.x release of Spring available in the searched  
    repositories  
        implementation 'org.springframework:spring-web:5.+'  
}
```

To ensure [reproducible builds](#), it's necessary to **lock versions** of dependencies and their transitive dependencies. This guarantees that a build with the same inputs will always resolve to the same module versions, a process known as **dependency locking**.

Dependency locking is a process where Gradle saves the resolved versions of dependencies to a lock file, ensuring that subsequent builds use the same dependency versions. This lock state is stored in a file and helps to prevent unexpected changes in the dependency graph.

Dependency locking offers several key advantages:

- **Avoiding Cascading Failures:** Teams managing multiple repositories no longer need to rely on `-SNAPSHOT` or changing dependencies, which can lead to unexpected failures if a dependency introduces a bug or incompatibility.
- **Dynamic Version Flexibility with Stability:** Teams that use the latest versions of dependencies can rely on dynamic versions during development and testing phases, locking them only for releases.
- **Publishing Resolved Versions:** By combining dependency locking with the practice of [publishing resolved versions](#), dynamic versions are replaced with the actual resolved versions at the time of publication.

- **Optimizing Build Cache Usage:** Since dynamic or changing dependencies violate the principle of [stable task inputs](#), locking dependencies ensures that tasks have consistent inputs.
- **Enhanced Development Workflow:** Developers can lock dependencies locally for stability while working on a feature or debugging an issue, while CI environments can test the latest **SNAPSHOT** or nightly versions to provide early feedback on integration issues. This allows teams to balance stability and early feedback during development.

Activate locking for specific configurations

Locking is enabled per [dependency configuration](#).

Once enabled, you must create an initial lock state, causing Gradle to verify that resolution results do not change. This ensures that if the selected dependencies differ from the locked ones (due to newer versions being available), the build will fail, preventing unexpected version changes.

Dependency locking is effective with [dynamic versions](#), but it **should not be used** with [changing versions](#) (e.g., **-SNAPSHOT**), where the coordinates remain the same, but the content may change.

WARNING

Using dependency locking with changing versions indicates a misunderstanding of these features and can lead to unpredictable results.

Gradle will emit a warning when persisting the lock state if changing dependencies are present in the resolution result.

Locking of a configuration happens through the [ResolutionStrategy](#) API:

build.gradle.kts

```
configurations {
    compileClasspath {
        resolutionStrategy.activateDependencyLocking()
    }
}
```

build.gradle

```
configurations {
    compileClasspath {
        resolutionStrategy.activateDependencyLocking()
    }
}
```

Only configurations that can be resolved will have lock state attached to them. Applying locking on

non resolvable-configurations is a no-op.

Activate locking for all configurations

The following locks all configurations:

build.gradle.kts

```
dependencyLocking {  
    lockAllConfigurations()  
}
```

build.gradle

```
dependencyLocking {  
    lockAllConfigurations()  
}
```

The above will lock all *project* configurations, but not the *buildscript* ones.

Disable locking for specific configurations

You can also disable locking on a specific configuration.

This can be useful if a plugin configured locking on all configurations, but you happen to add one that should not be locked:

build.gradle.kts

```
configurations.compileClasspath {  
    resolutionStrategy.deactivateDependencyLocking()  
}
```

build.gradle

```
configurations {  
    compileClasspath {  
        resolutionStrategy.deactivateDependencyLocking()  
    }  
}
```

Activate locking for a buildscript classpath configuration

If you apply plugins to your build, you may want to leverage dependency locking there as well.

To lock the `classpath` configuration used for script plugins:

build.gradle.kts

```
buildscript {
    configurations.classpath {
        resolutionStrategy.activateDependencyLocking()
    }
}
```

build.gradle

```
buildscript {
    configurations.classpath {
        resolutionStrategy.activateDependencyLocking()
    }
}
```

Generating and updating dependency locks

To generate or update the lock state, add the `--write-locks` argument while invoking whatever tasks that would trigger the locked configurations to be resolved:

```
$ ./gradlew dependencies --write-locks
```

This will create or update the lock state for each resolved configuration during that build execution. If a lock state already exists, it will be overwritten.

gradle.lockfile

```
# This is a Gradle generated file for dependency locking.
# Manual edits can break the build and are not advised.
# This file is expected to be part of source control.
com.google.code.findbugs:jsr305:3.0.2=classpath
com.google.errorprone:error_prone_annotations:2.3.2=classpath
com.google.gradle:osdetector-gradle-plugin:1.7.1=classpath
com.google.guava:failureaccess:1.0.1=classpath
com.google.guava:guava:28.1-jre=classpath
com.google.guava:listenablefuture:9999.0-empty-to-avoid-conflict-with-guava=classpath
com.google.j2objc:j2objc-annotations:1.3=classpath
```

empty=

NOTE

Gradle won't write the lock state to disk if the build fails, preventing the persistence of potentially invalid states.

Lock all configurations in a single build execution

When working with multiple configurations, you may want to lock them all at once in a single build execution. You have two options for this:

1. Run `gradle dependencies --write-locks`:

- This command will lock all resolvable configurations that have locking enabled.
- In a multi-project setup, note that `dependencies` is executed only on one project, typically the root project.

2. Declare a Custom Task to Resolve All Configurations:

- This approach is particularly useful if you need more control over which configurations are locked.

This custom task resolves all configurations, locking them in the process:

build.gradle.kts

```
tasks.register("resolveAndLockAll") {
    notCompatibleWithConfigurationCache("Filters configurations at execution time")
    doFirst {
        require(gradle.startParameter.isWriteDependencyLocks) { "$path must be run from the command line with the '--write-locks' flag" }
    }
    doLast {
        configurations.filter {
            // Add any custom filtering on the configurations to be resolved
            it.isCanBeResolved
        }.forEach { it.resolve() }
    }
}
```

build.gradle

```
tasks.register('resolveAndLockAll') {
    notCompatibleWithConfigurationCache("Filters configurations at execution time")
    doFirst {
        assert gradle.startParameter.writeDependencyLocks : "$path must be
```

```
run from the command line with the '--write-locks' flag"
}
doLast {
    configurations.findAll {
        // Add any custom filtering on the configurations to be resolved
        it.canBeResolved
    }.each { it.resolve() }
}
}
```

By filtering and resolving specific configurations, you ensure that only the relevant ones are locked, tailoring the locking process to your project's needs. This is especially useful in environments like native builds, where not all configurations can be resolved on a single platform.

Understanding lock state location and format

A lockfile is a critical component that records the exact versions of dependencies used in a project, allowing for verification during builds to ensure consistent results across different environments and over time. It helps identify discrepancies in dependencies when a project is built on different machines or at different times.

TIP Lockfiles should be checked in to source control.

Location of lock files

- The lock state is preserved in a file named `gradle.lockfile`, located at the root of each project or subproject directory.
- The exception is the lockfile for the `buildscript` itself, which is named `buildscript-gradle.lockfile`.

Structure of lock files

Consider the following dependency declaration:

build.gradle.kts

```
configurations {
    compileClasspath {
        resolutionStrategy.activateDependencyLocking()
    }
    runtimeClasspath {
        resolutionStrategy.activateDependencyLocking()
    }
    annotationProcessor {
        resolutionStrategy.activateDependencyLocking()
    }
}
```

```

}

dependencies {
    implementation("org.springframework:spring-beans:[5.0,6.0)")
}

```

build.gradle

```

configurations {
    compileClasspath {
        resolutionStrategy.activateDependencyLocking()
    }
    runtimeClasspath {
        resolutionStrategy.activateDependencyLocking()
    }
    annotationProcessor {
        resolutionStrategy.activateDependencyLocking()
    }
}

dependencies {
    implementation 'org.springframework:spring-beans:[5.0,6.0)'
}

```

With the above configuration, the generated **gradle.lockfile** will look like this:

gradle.lockfile

```

# This is a Gradle generated file for dependency locking.
# Manual edits can break the build and are not advised.
# This file is expected to be part of source control.
org.springframework:spring-beans:5.0.5.RELEASE=compileClasspath, runtimeClasspath
org.springframework:spring-core:5.0.5.RELEASE=compileClasspath, runtimeClasspath
org.springframework:spring-jcl:5.0.5.RELEASE=compileClasspath, runtimeClasspath
empty=annotationProcessor

```

Where:

- **Each line** represents a single dependency in the **group:artifact:version** format.
- **Configurations:** After the version, the configurations that include the dependency are listed.
- **Ordering:** Dependencies and configurations are listed alphabetically to make version control diffs easier to manage.
- **Empty Configurations:** The last line lists configurations that are empty, meaning they contain no dependencies.

This **lockfile** should be included in source control to ensure that all team members and environments use the exact same dependency versions.

Migrating your legacy lockfile

If your project uses the legacy lock file format of a file per locked configuration, follow these instructions to migrate to the new format:

1. Follow the documentation for [writing](#) or [updating](#) dependency lock state.
2. Upon writing the single lock file per project, Gradle will also delete all lock files per configuration for which the state was transferred.

NOTE

Migration can be done one configuration at a time. Gradle will keep sourcing the lock state from the per configuration files as long as there is no information for that configuration in the single lock file.

Configuring the lock file name and location

When using a single lock file per project, you can configure its name and location.

This capability allows you to specify a file name based on project properties, enabling a single project to store different lock states for different execution contexts.

For example, in the JVM ecosystem, the Scala version is often included in artifact coordinates:

build.gradle.kts

```
val scalaVersion = "2.12"
dependencyLocking {
    lockFile = file("${projectDir}/locking/gradle-${scalaVersion}.lockfile")
}
```

build.gradle

```
def scalaVersion = "2.12"
dependencyLocking {
    lockFile = file("${projectDir}/locking/gradle-${scalaVersion}.lockfile")
}
```

Running a build with lock state present

The moment a build needs to resolve a configuration that has locking enabled, and it finds a matching lock state, it will use it to verify that the given configuration still resolves the same

versions.

A successful build indicates that the same dependencies are used by your build as stored in the lock state, regardless if new versions matching the dynamic selector are available in any of the repositories your build uses.

The complete validation is as follows:

- Existing entries in the lock state must be matched in the build
 - A version mismatch or missing resolved module causes a build failure
- Resolution result must not contain extra dependencies compared to the lock state

Fine-tuning dependency locking behaviour with lock mode

While the default lock mode behaves as described above, two other modes are available:

Strict mode

In this mode, in addition to the validations above, dependency locking will fail if a configuration marked as *locked* does not have lock state associated with it.

Lenient mode

In this mode, dependency locking will still pin dynamic versions but otherwise changes to the dependency resolution are no longer errors. Other changes include:

- Adding or removing dependencies, even if they are strictly versioned, without causing a build failure.
- Allowing transitive dependencies to shift, as long as dynamic versions are still pinned.

This mode offers flexibility for situations where you might want to explore or test new dependencies or changes in versions without breaking the build, making it useful for testing nightly or snapshot builds.

The lock mode can be controlled from the `dependencyLocking` block as shown below:

build.gradle.kts

```
dependencyLocking {  
    lockMode = LockMode.STRICT  
}
```

build.gradle

```
dependencyLocking {  
    lockMode = LockMode.STRICT  
}
```

Updating lock state entries selectively

In order to update only specific modules of a configuration, you can use the `--update-locks` command line flag. It takes a comma (,) separated list of module notations. In this mode, the existing lock state is still used as input to resolution, filtering out the modules targeted by the update:

```
$ ./gradlew dependencies --update-locks org.apache.commons:commons-lang3,org.slf4j:slf4j-api
```

Wildcards, indicated with `*`, can be used in the group or module name. They can be the only character or appear at the end of the group or module respectively. The following wildcard notation examples are valid:

- `org.apache.commons:*`: will let all modules belonging to group `org.apache.commons` update
- `*:guava`: will let all modules named `guava`, whatever their group, update
- `org.springframework.spring*:spring*`: will let all modules having their group starting with `org.springframework.spring` and name starting with `spring` update

NOTE

The resolution may cause other module versions to update, as dictated by the Gradle resolution rules.

Disabling dependency locking

To disable dependency locking for a configuration:

1. **Remove Locking Configuration:** Ensure that the configuration you no longer want to lock is not configured with dependency locking. This means removing or commenting out any `activateDependencyLocking()` calls for that configuration.
2. **Update Lock State:** The next time you update and save the lock state (using the `--write-locks` option), Gradle will automatically clean up any stale lock state associated with the configurations that are no longer locked.

NOTE

Gradle must resolve a configuration that is no longer marked as locked to detect and drop the associated lock state. Without resolving the configuration, Gradle cannot identify which lock state should be cleaned up.

Ignoring specific dependencies from the lock state

In some scenarios, you may want to use dependency locking for other reasons than build reproducibility.

As a build author, you might want certain dependencies to update more frequently than others. For example, internal dependencies within an organization might always use the latest version, while third-party dependencies follow a different update cycle.

CAUTION

This approach can compromise reproducibility. Consider using [different lock modes](#) or [separate lock files](#) for specific cases.

You can configure dependencies to be ignored in the `dependencyLocking` project extension:

build.gradle.kts

```
dependencyLocking {  
    ignoredDependencies.add("com.example:*")  
}
```

build.gradle

```
dependencyLocking {  
    ignoredDependencies.add('com.example:*')  
}
```

The notation `<group>:<name>` is used to specify dependencies, where `*` acts as a trailing wildcard. Note that `*:*` is not accepted, as it effectively disables locking. See [the description](#) on updating lock files for more details.

Ignoring dependencies will have the following effects:

- Ignored dependencies apply across all locked configurations, and the setting is project scoped.
- Ignoring a dependency does not exclude its transitive dependencies from the lock state.
- No validation ensures that an ignored dependency is present in any configuration resolution.
- If the dependency is present in lock state, loading it will filter out the dependency.
- If the dependency is present in the resolution result, it will be ignored when validating the resolution against the lock state.
- When the lock state is updated and persisted, any ignored dependency will be omitted from the written lock state.

Understanding locking limitations

- Dependency locking does not currently apply to source dependencies.

Using Resolution Rules

Gradle provides several mechanisms to directly influence the behavior of the dependency resolution engine.

Unlike [dependency constraints](#) or [component metadata rules](#), which serve as inputs to the resolution process, these mechanisms allow you to inject rules directly into the resolution engine. Because of their direct impact, they can be considered brute-force solutions that may mask underlying issues, such as the introduction of new dependencies.

TIP

It's generally advisable to resort to **resolution rules** only when other approaches are insufficient.

If you're developing a [library](#), it's best to use [dependency constraints](#), as they are shared with your consumers.

Here are the key resolution strategies in Gradle:

#	Strategy	Info
1	Forcing Dependency Versions	Force a specific version of a dependency.
2	Module Replacement	Substitute one module for another with an explanation.
3	Dependency Substitution	Substitute dependencies dynamically.
4	Component Selection Rules	Control which versions of a module are allowed. Reject specific versions that are known to be broken or undesirable.
5	Default Dependencies	Automatically add dependencies to a configuration when no dependencies are explicitly declared.
6	Excluding Transitive Dependencies	Exclude transitive dependencies that you don't want to be included in the dependency graph.
7	Force Failed Resolution Strategies	Force builds to fail when certain conditions occur during resolution.
8	Disabling Transitive Dependencies	Dependencies are transitive by default, but you can disable this behavior for individual dependencies.
9	Dependency Resolve Rules and Other Conditionals	Transform or filter dependencies directly as they are resolved and other corner case scenarios.

1. Forcing Dependency Versions

You can enforce a specific version of a dependency, regardless of what versions might be requested or resolved by other parts of the build script.

This is useful for ensuring consistency and avoiding conflicts due to different versions of the same dependency being used.

build.gradle.kts

```
configurations {  
    "compileClasspath" {
```

```
        resolutionStrategy.force("commons-codec:commons-codec:1.9")
    }
}

dependencies {
    implementation("org.apache.httpcomponents:httpclient:4.5.4")
}
```

build.gradle

```
configurations {
    compileClasspath {
        resolutionStrategy.force 'commons-codec:commons-codec:1.9'
    }
}

dependencies {
    implementation 'org.apache.httpcomponents:httpclient:4.5.4'
}
```

2. Module Replacement

While it's generally better to manage module conflicts using [capabilities](#), there are scenarios—especially when working with older versions of Gradle—that require a different approach. In these cases, **module replacement rules** offer a solution by allowing you to specify that a legacy library has been replaced by a newer one.

Module replacement rules allow you to declare that a legacy library has been replaced by a newer one. For instance, the migration from `google-collections` to `guava` involved renaming the module from `com.google.collections:google-collections` to `com.google.guava:guava`. Such changes impact conflict resolution because Gradle doesn't treat them as version conflicts due to different module coordinates.

Consider a scenario where both libraries appear in the dependency graph. Your project depends on `guava`, but a transitive dependency pulls in `google-collections`. This can cause runtime errors since Gradle won't automatically resolve this as a conflict. Common solutions include:

- Declaring an exclusion rule to avoid `google-collections`.
- Avoiding dependencies that pull in legacy libraries.
- Upgrading dependencies that no longer use `google-collections`.
- Downgrading to `google-collections` (not recommended).
- Assigning capabilities so `google-collections` and `guava` are mutually exclusive.

These methods can be insufficient for large-scale projects. By declaring module replacements, you

can address this issue globally across projects, allowing organizations to handle such conflicts holistically.

build.gradle.kts

```
dependencies {
    modules {
        module("com.google.collections:google-collections") {
            replacedBy("com.google.guava:guava", "google-collections is now
part of Guava")
        }
    }
}
```

build.gradle

```
dependencies {
    modules {
        module("com.google.collections:google-collections") {
            replacedBy("com.google.guava:guava", "google-collections is now
part of Guava")
        }
    }
}
```

Once declared, Gradle treats any version of **guava** as superior to **google-collections** during conflict resolution, ensuring only **guava** appears in the classpath. However, if **google-collections** is the only module present, it won't be automatically replaced unless there's a conflict.

For more examples, refer to the DSL reference for [ComponentMetadataHandler](#).

NOTE

Gradle does not currently support replacing a module with multiple modules, but multiple modules can be replaced by a single module.

3. Dependency Substitution

Dependency substitution rules allow for replacing project and module dependencies with specified alternatives, making them interchangeable. While similar to dependency resolve rules, they offer more flexibility by enabling substitution between project and module dependencies.

However, adding a **dependency substitution rule** affects the timing of configuration resolution. Instead of resolving on first use, the configuration is resolved during task graph construction, which can cause issues if the configuration is modified later or depends on modules published during task execution.

Explanation:

- A configuration can serve as input to a task and include project dependencies when resolved.
- If a project dependency is an input to a task (via a configuration), then tasks to build those artifacts are added as dependencies.
- To determine project dependencies that are inputs to a task, Gradle must resolve the configuration inputs.
- Because the Gradle task graph is fixed once task execution has commenced, Gradle needs to perform this resolution prior to executing any tasks.

Without substitution rules, Gradle assumes that external module dependencies don't reference project dependencies, simplifying dependency traversal. With substitution rules, this assumption no longer holds, so Gradle must fully resolve the configuration to determine project dependencies.

Substituting an external module dependency with a project dependency

Dependency substitution can be used to replace an external module with a locally developed project, which is helpful when testing a patched or unreleased version of a module.

The external module can be replaced whether a version is specified:

build.gradle.kts

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute(module("org.utils:api"))
            .using(project(":api")).because("we work with the unreleased
development version")
        substitute(module("org.utils:util:2.5")).using(project(":util"))
    }
}
```

build.gradle

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute module("org.utils:api") using project(":api") because "we
work with the unreleased development version"
        substitute module("org.utils:util:2.5") using project(":util")
    }
}
```

- Substituted projects must be part of the multi-project build (included via [settings.gradle](#)).

- The substitution replaces the module dependency with the project dependency and sets up task dependencies, but doesn't automatically include the project in the build.

Substituting a project dependency with a module replacement

You can also use substitution rules to replace a project dependency with an external module in a multi-project build.

This technique can accelerate development by allowing certain dependencies to be downloaded from a repository instead of being built locally:

build.gradle.kts

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute(project(":api"))
            .using(module("org.utils:api:1.3")).because("we use a stable
version of org.utils:api")
    }
}
```

build.gradle

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute project(":api") using module("org.utils:api:1.3") because
        "we use a stable version of org.utils:api"
    }
}
```

- The substituted module must include a version.
- Even after substitution, the project remains part of the multi-project build, but tasks to build it won't be executed when resolving the configuration.

Conditionally substituting a dependency

You can conditionally substitute a module dependency with a local project in a multi-project build using dependency substitution rules.

This is particularly useful when you want to use a locally developed version of a dependency if it exists, otherwise fall back to the external module:

build.gradle.kts

```
configurations.all {
    resolutionStrategy.dependencySubstitution.all {
        requested.let {
            if (it is ModuleComponentSelector && it.group == "org.example") {
                val targetProject = findProject(":${it.module}")
                if (targetProject != null) {
                    useTarget(targetProject)
                }
            }
        }
    }
}
```

build.gradle

```
configurations.all {
    resolutionStrategy.dependencySubstitution.all { DependencySubstitution
dependency ->
        if (dependency.requested instanceof ModuleComponentSelector &&
dependency.requested.group == "org.example") {
            def targetProject = findProject(":${dependency.requested.module}
")
            if (targetProject != null) {
                dependency.useTarget targetProject
            }
        }
    }
}
```

- The substitution only occurs if a local project matching the dependency name is found.
- The local project must already be included in the multi-project build (via `settings.gradle`).

Substituting a dependency with another variant

You can substitute a dependency with another [variant](#), such as switching between a platform dependency and a regular library dependency.

This is useful when your build process needs to change the type of dependency based on specific conditions:

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
```

```

    all {
        if (requested.is ModuleComponentSelector && requested.group ==
"org.example" && requested.version == "1.0") {
            useTarget(module("org.example:library:1.0")).because("Switching from
platform to library variant")
        }
    }
}

```

- The substitution is based on the requested dependency's attributes (like group and version).
- This approach allows you to switch from a platform component to a library or vice versa.
- It uses Gradle's variant-aware engine to ensure the correct variant is selected based on the configuration and consumer attributes.

This flexibility is often required when working with complex dependency graphs where different component types (platforms, libraries) need to be swapped dynamically.

Substituting a dependency with attributes

Substituting a dependency based on attributes allows you to override the default selection of a component by targeting specific attributes (like platform vs. regular library).

This technique is useful for managing platform and library dependencies in complex builds, particularly when you want to consume a regular library but the platform dependency was incorrectly declared:

lib/build.gradle.kts

```

dependencies {
    // This is a platform dependency but you want the library
    implementation(platform("com.google.guava:guava:28.2-jre"))
}

```

lib/build.gradle

```

dependencies {
    // This is a platform dependency but you want the library
    implementation platform('com.google.guava:guava:28.2-jre')
}

```

In this example, the substitution rule targets the platform version of `com.google.guava:guava` and replaces it with the regular library version:

consumer/build.gradle.kts

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute(platform(module("com.google.guava:guava:28.2-jre")))
            .using(module("com.google.guava:guava:28.2-jre"))
    }
}
```

consumer/build.gradle

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute(platform(module('com.google.guava:guava:28.2-jre'))).
            using module('com.google.guava:guava:28.2-jre')
    }
}
```

Without the **platform** keyword, the substitution would not specifically target the platform dependency.

The following rule performs the same substitution but uses the more granular **variant** notation, allowing for customization of the dependency's attributes:

consumer/build.gradle.kts

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute(variant(module("com.google.guava:guava:28.2-jre")) {
            attributes {
                attribute(Category.CATEGORY_ATTRIBUTE,
objects.named(Category.REGULAR_PLATFORM))
            }
        }).using(module("com.google.guava:guava:28.2-jre"))
    }
}
```

consumer/build.gradle

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
```

```

        substitute variant(module('com.google.guava:guava:28.2-jre')) {
            attributes {
                attribute(Category.CATEGORY_ATTRIBUTE, objects.named(
                    Category, Category.REGULAR_PLATFORM))
            }
        } using module('com.google.guava:guava:28.2-jre')
    }
}

```

By using attribute-based substitution, you can precisely control which dependencies are replaced, ensuring Gradle resolves the correct versions and variants in your build.

Refer to the [DependencySubstitutions API](#) for a complete reference.

WARNING

In [composite builds](#), the rule that you have to match the exact requested dependency attributes is not applied. When using composites, Gradle will automatically match the requested attributes. In other words, it is implicit that if you include another build, you are substituting *all variants* of the substituted module with an equivalent variant in the included build.

Substituting a dependency with a dependency with capabilities

You can substitute a dependency with a different variant that includes specific **capabilities**. Capabilities allow you to specify that a particular variant of a dependency offers a set of related features or functionality, such as [test fixtures](#).

This example substitutes a regular dependency with its [test fixtures](#) using a capability:

build.gradle.kts

```

configurations.testCompileClasspath {
    resolutionStrategy.dependencySubstitution {

        substitute(module("com.acme:lib:1.0")).using(variant(module("com.acme:lib:1.0
        "))) {
            capabilities {
                requireCapability("com.acme:lib-test-fixtures")
            }
        }
    }
}

```

build.gradle

```

configurations.testCompileClasspath {

```

```

resolutionStrategy.dependencySubstitution {
    substitute(module('com.acme:lib:1.0'))
        .using variant(module('com.acme:lib:1.0')) {
            capabilities {
                requireCapability('com.acme:lib-test-fixtures')
            }
        }
}

```

Here, we substitute the regular `com.acme:lib:1.0` dependency with its `lib-test-fixtures` variant. The `requireCapability` function specifies that the new variant must have the `com.acme:lib-test-fixtures` capability, ensuring the right version of the dependency is selected for testing purposes.

Capabilities within the substitution rule are used to precisely match dependencies, and Gradle only substitutes dependencies that match the required capabilities.

Refer to the [DependencySubstitutions API](#) for a complete reference of the variant substitution API.

Substituting a dependency with a classifier or artifact

You can substitute dependencies that have a classifier with ones that don't or vice versa. Classifiers are often used to represent different versions of the same artifact, such as platform-specific builds or dependencies with different APIs. Although Gradle discourages the use of classifiers, it provides a way to handle substitutions for cases where classifiers are still in use.

Consider the following setup:

consumer/build.gradle.kts

```

dependencies {
    implementation("com.google.guava:guava:28.2-jre")
    implementation("co.paralleluniverse:quasar-core:0.8.0")
    implementation(project(":lib"))
}

```

consumer/build.gradle

```

dependencies {
    implementation 'com.google.guava:guava:28.2-jre'
    implementation 'co.paralleluniverse:quasar-core:0.8.0'
    implementation project(':lib')
}

```

In the example above, the first level dependency on `quasar` makes us think that Gradle would resolve `quasar-core-0.8.0.jar` but it's not the case.

The build fails with this message:

```
Execution failed for task ':consumer:resolve'.
> Could not resolve all files for configuration ':consumer:runtimeClasspath'.
   > Could not find quasar-core-0.8.0-jdk8.jar (co.paralleluniverse:quasar-core:0.8.0).
      Searched in the following locations:
        https://repo.maven.apache.org/maven2/co/paralleluniverse/quasar-core/0.8.0/quasar-core-0.8.0-jdk8.jar
```

That's because there's a dependency on another project, `lib`, which itself depends on a different version of `quasar-core`:

lib/build.gradle.kts

```
dependencies {
    implementation("co.paralleluniverse:quasar-core:0.7.10:jdk8")
}
```

lib/build.gradle

```
dependencies {
    implementation "co.paralleluniverse:quasar-core:0.7.10:jdk8"
}
```

- The consumer depends on `quasar-core:0.8.0` without a classifier.
- The lib project depends on `quasar-core:0.7.10` with the `jdk8` classifier.
- Gradle's conflict resolution selects the higher version (`0.8.0`), but `quasar-core:0.8.0` doesn't have the `jdk8` classifier, leading to a resolution error.

To resolve this conflict, you can instruct Gradle to ignore classifiers when resolving `quasar-core` dependencies:

consumer/build.gradle.kts

```
configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute(module("co.paralleluniverse:quasar-core"))
    }
}
```

```

        .using(module("co.paralleluniverse:quasar-core:0.8.0"))
        .withoutClassifier()
    }
}

```

consumer/build.gradle

```

configurations.all {
    resolutionStrategy.dependencySubstitution {
        substitute module('co.paralleluniverse:quasar-core') using module(
            'co.paralleluniverse:quasar-core:0.8.0') withoutClassifier()
        }
    }
}

```

This rule effectively replaces any dependency on `quasar-core` found in the graph with a dependency without classifier.

If you need to substitute with a specific classifier or artifact, you can specify the classifier or artifact details in the substitution rule.

For more detailed information, refer to:

- Artifact selection via the [Substitution DSL](#)
- Artifact selection via the [DependencySubstitution API](#)
- Artifact selection via the [ResolutionStrategy API](#)

4. Component Selection Rules

Component selection rules may influence which component instance should be selected when multiple versions are available that match a version selector. Rules are applied against every available version and allow the version to be explicitly rejected.

This allows Gradle to ignore any component instance that does not satisfy conditions set by the rule. Examples include:

- For a dynamic version like `1.+` certain versions may be explicitly rejected from selection.
- For a static version like `1.4` an instance may be rejected based on extra component metadata such as the Ivy branch attribute, allowing an instance from a subsequent repository to be used.

Rules are configured via the [ComponentSelectionRules](#) object. Each rule configured will be called with a [ComponentSelection](#) object as an argument that contains information about the candidate version being considered. Calling [ComponentSelection.reject\(java.lang.String\)](#) causes the given candidate version to be explicitly rejected, in which case the candidate will not be considered for the selector.

The following example shows a rule that disallows a particular version of a module but allows the dynamic version to choose the next best candidate:

build.gradle.kts

```
configurations {
    implementation {
        resolutionStrategy {
            componentSelection {
                // Accept the highest version matching the requested version
                // that isn't '1.5'
                all {
                    if (candidate.group == "org.sample" && candidate.module
                        == "api" && candidate.version == "1.5") {
                        reject("version 1.5 is broken for 'org.sample:api'")
                    }
                }
            }
        }
    }
}

dependencies {
    implementation("org.sample:api:1.+")
}
```

build.gradle

```
configurations {
    implementation {
        resolutionStrategy {
            componentSelection {
                // Accept the highest version matching the requested version
                // that isn't '1.5'
                all { ComponentSelection selection ->
                    if (selection.candidate.group == 'org.sample' &&
                        selection.candidate.module == 'api' && selection.candidate.version == '1.5')
                    {
                        selection.reject("version 1.5 is broken for
                        'org.sample:api'")
                    }
                }
            }
        }
    }
}
```

```
dependencies {
    implementation 'org.sample:api:1.+'
}
```

Note that version selection is applied starting with the highest version first. The version selected will be the first version found that all component selection rules accept.

IMPORTANT

A version is considered accepted if no rule explicitly rejects it.

Similarly, rules can be targeted at specific modules. Modules must be specified in the form of `group:module:`

build.gradle.kts

```
configurations {
    create("targetConfig") {
        resolutionStrategy {
            componentSelection {
                withModule("org.sample:api") {
                    if (candidate.version == "1.5") {
                        reject("version 1.5 is broken for 'org.sample:api'")
                    }
                }
            }
        }
    }
}
```

build.gradle

```
configurations {
    targetConfig {
        resolutionStrategy {
            componentSelection {
                withModule("org.sample:api") { ComponentSelection selection
->
                    if (selection.candidate.version == "1.5") {
                        selection.reject("version 1.5 is broken for
'org.sample:api'")
                    }
                }
            }
        }
    }
}
```

```
}
```

Component selection rules can also consider component metadata when selecting a version. Possible additional metadata that can be considered are [ComponentMetadata](#) and [IvyModuleDescriptor](#).

Note that this extra information may not always be available and thus should be checked for `null` values:

build.gradle.kts

```
configurations {
    create("metadataRulesConfig") {
        resolutionStrategy {
            componentSelection {
                // Reject any versions with a status of 'experimental'
                all {
                    if (candidate.group == "org.sample" && metadata?.status
== "experimental") {
                        reject("don't use experimental candidates from
'org.sample'")
                    }
                }
                // Accept the highest version with either a "release" branch
or a status of 'milestone'
                withModule("org.sample:api") {
                    if (getDescriptor(IvyModuleDescriptor::class)?.branch !=
"release" && metadata?.status != "milestone") {
                        reject("'org.sample:api' must have testing branch or
milestone status")
                    }
                }
            }
        }
    }
}
```

build.gradle

```
configurations {
    metadataRulesConfig {
        resolutionStrategy {
            componentSelection {
                // Reject any versions with a status of 'experimental'
                all { ComponentSelection selection ->
```

```

        if (selection.candidate.group == 'org.sample' &&
selection.metadata?.status == 'experimental') {
            selection.reject("don't use experimental candidates
from 'org.sample'")
        }
    }
    // Accept the highest version with either a "release" branch
or a status of 'milestone'
    withModule('org.sample:api') { ComponentSelection selection
->
        if (selection.getDescriptor(IvyModuleDescriptor)?.branch
!= "release" && selection.metadata?.status != 'milestone') {
            selection.reject("'org.sample:api' must be a release
branch or have milestone status")
        }
    }
}
}
}
}
}
}
}
}
}
}

```

A [ComponentSelection](#) argument is *always* required as a parameter when declaring a component selection rule.

5. Default Dependencies

You can set default dependencies for a configuration to ensure that a default version is used when no explicit dependencies are specified.

This is useful for plugins that rely on versioned tools and want to provide a default if the user doesn't specify a version:

build.gradle.kts

```

configurations {
    create("pluginTool") {
        defaultDependencies {
            add(project.dependencies.create("org.gradle:my-util:1.0"))
        }
    }
}

```

build.gradle

```

configurations {

```

```

    pluginTool {
        defaultDependencies { dependencies ->
            dependencies.add(project.dependencies.create("org.gradle:my-
util:1.0"))
        }
    }
}

```

In this example, the `pluginTool` configuration will use `org.gradle:my-util:1.0` as a default dependency unless another version is specified.

6. Excluding Transitive Dependencies

To completely exclude a transitive dependency for a particular configuration, use the `Configuration.exclude(Map)` method.

This approach will automatically exclude the specified transitive dependency from all dependencies declared within the configuration:

build.gradle.kts

```

configurations {
    "implementation" {
        exclude(group = "commons-collections", module = "commons-
collections")
    }
}

dependencies {
    implementation("commons-beanutils:commons-beanutils:1.9.4")
    implementation("com.opencsv:opencsv:4.6")
}

```

build.gradle

```

configurations {
    implementation {
        exclude group: 'commons-collections', module: 'commons-collections'
    }
}

dependencies {
    implementation 'commons-beanutils:commons-beanutils:1.9.4'
    implementation 'com.opencsv:opencsv:4.6'
}

```

```
}
```

In this example, the `commons-collections` dependency will be excluded from the `implementation` configuration, regardless of whether it is a direct or transitive dependency.

7. Force Failed Resolution Strategies

Version conflicts can be forced to fail using:

- `failOnNonReproducibleResolution()`
- `failOnDynamicVersions()`
- `failOnChangingVersions()`
- `failOnVersionConflict()`

This will fail the build when conflicting versions of the same dependency are found:

build.gradle.kts

```
configurations.all {  
    resolutionStrategy {  
        failOnVersionConflict()  
    }  
}
```

build.gradle

```
configurations.all {  
    resolutionStrategy {  
        failOnVersionConflict()  
    }  
}
```

8. Disabling Transitive Dependencies

By default, Gradle resolves all transitive dependencies for a given module.

However, there are situations where you may want to disable this behavior, such as when you need more control over dependencies or when the dependency metadata is incorrect.

You can tell Gradle to disable transitive dependency management for a dependency by setting `ModuleDependency.setTransitive(boolean)` to `false`.

In the following example, transitive dependency resolution is disabled for the **guava** dependency:

build.gradle.kts

```
dependencies {
    implementation("com.google.guava:guava:23.0") {
        isTransitive = false
    }
}
```

build.gradle

```
dependencies {
    implementation('com.google.guava:guava:23.0') {
        transitive = false
    }
}
```

This ensures only the main artifact for **guava** is resolved, and none of its transitive dependencies will be included.

NOTE

Disabling transitive dependency resolution will likely require you to declare the necessary runtime dependencies in your build script which otherwise would have been resolved automatically. Not doing so might lead to runtime classpath issues.

If you want to disable transitive resolution globally across all dependencies, you can set this behavior at the configuration level:

build.gradle.kts

```
configurations.all {
    isTransitive = false
}

dependencies {
    implementation("com.google.guava:guava:23.0")
}
```

build.gradle

```
configurations.all {
```

```

    transitive = false
  }
  dependencies {
    implementation 'com.google.guava:guava:23.0'
  }

```

This disables transitive resolution for all dependencies in the project. Be aware that this may require you to manually declare any transitive dependencies that are required at runtime.

For more information, see [Configuration.setTransitive\(boolean\)](#).

9. Dependency Resolve Rules and Other Conditionals

Dependency resolve rules are executed for each dependency as it's being resolved, providing a powerful API to modify a dependency's attributes—such as group, name, or version—before the resolution is finalized.

This allows for advanced control over dependency resolution, enabling you to substitute one module for another during the resolution process.

This feature is particularly useful for implementing advanced dependency management patterns. With **dependency resolve rules**, you can redirect dependencies to specific versions or even different modules entirely, allowing you to enforce consistent versions across a project or override problematic dependencies:

build.gradle.kts

```

configurations.all {
    resolutionStrategy {
        eachDependency {
            if (requested.group == "com.example" && requested.name == "old-
library") {
                useTarget("com.example:new-library:1.0.0")
                because("Our license only allows use of version 1")
            }
        }
    }
}

```

build.gradle

```

configurations.all {
    resolutionStrategy {
        eachDependency {
            if (requested.group == "com.example" && requested.name == "old-

```

```
library") {
    useTarget("com.example:new-library:1.0.0")
    because("Our license only allows use of version 1")
}
}
```

In this example, if a dependency on `com.example:old-library` is requested, it will be substituted with `com.example:new-library:1.0.0` during resolution.

For more advanced usage and additional examples, refer to the [ResolutionStrategy](#) class in the API documentation.

Implementing a custom versioning scheme

In some corporate environments, module versions in Gradle builds are maintained and audited externally. **Dependency resolve rules** offer an effective way to implement this:

- Developers declare dependencies in the build script using the module's group and name, but specify a placeholder version like `default`.
- A dependency resolve rule then resolves the `default` version to an approved version, which is retrieved from a corporate catalog of sanctioned modules.

This approach ensures that only approved versions are used, while allowing developers to work with a simplified and consistent versioning scheme.

The rule implementation can be encapsulated in a corporate plugin, making it easy to apply across all projects within the organization:

build.gradle.kts

```
configurations.all {
    resolutionStrategy.eachDependency {
        if (requested.version == "default") {
            val version = findDefaultVersionInCatalog(requested.group,
requested.name)
            useVersion(version.version)
            because(version.because)
        }
    }
}

data class DefaultVersion(val version: String, val because: String)

fun findDefaultVersionInCatalog(group: String, name: String): DefaultVersion
{
```

```
//some custom logic that resolves the default version into a specific
version
return DefaultVersion(version = "1.0", because = "tested by QA")
}
```

build.gradle

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.version == 'default') {
            def version = findDefaultVersionInCatalog(details.requested.
group, details.requested.name)
            details.useVersion version.version
            details.because version.because
        }
    }
}

def findDefaultVersionInCatalog(String group, String name) {
    //some custom logic that resolves the default version into a specific
version
    [version: "1.0", because: 'tested by QA']
}
```

In this setup, whenever a developer specifies `default` as the version, the resolve rule replaces it with the approved version from the corporate catalog.

This strategy ensures compliance with corporate policies while providing flexibility and ease of use for developers. Encapsulating this logic in a plugin also ensures consistency across multiple projects.

Replacing unwanted dependency versions

Dependency resolve rules offer a powerful mechanism for blocking specific versions of a dependency and substituting them with an alternative.

This is particularly useful when a specific version is known to be problematic—such as a version that introduces bugs or relies on a library that isn't available in public repositories. By defining a resolve rule, you can automatically replace a problematic version with a stable one.

Consider a scenario where version `1.2` of a library is broken, but version `1.2.1` contains important fixes and should always be used instead. With a resolve rule, you can enforce this substitution: "anytime version `1.2` is requested, it will be replaced with `1.2.1`. Unlike forcing a version, this rule only affects the specific version `1.2`, leaving other versions unaffected:

build.gradle.kts

```
configurations.all {
    resolutionStrategy.eachDependency {
        if (requested.group == "org.software" && requested.name == "some-
library" && requested.version == "1.2") {
            useVersion("1.2.1")
            because("fixes critical bug in 1.2")
        }
    }
}
```

build.gradle

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.group == 'org.software' && details.requested
.name == 'some-library' && details.requested.version == '1.2') {
            details.useVersion '1.2.1'
            details.because 'fixes critical bug in 1.2'
        }
    }
}
```

If version 1.3 is also present in the dependency graph, then even with this rule, Gradle's default conflict resolution strategy would select 1.3 as the latest version.

Difference from Rich Version Constraints: Using [rich version](#) constraints, you can reject certain versions outright, causing the build to fail or select a non-rejected version if a dynamic dependency is used. In contrast, a dependency resolve rule like the one shown here manipulates the version being requested, replacing it with a known good version when a rejected one is found. This approach is a **solution** for handling rejected versions, while rich version constraints are about expressing the intent to avoid certain versions.

Lazily influencing resolved dependencies

Plugins can lazily influence dependencies by adding them conditionally or setting preferred versions when no version is specified by the user.

Below are two examples illustrating these use cases.

This example demonstrates how to add a dependency to a configuration based on some condition, evaluated lazily:

Example 20. *Example 1: Conditionally Adding a Dependency*

build.gradle.kts

```
configurations {
    implementation {
        dependencies.addLater(project.provider {
            val dependencyNotation = conditionalLogic()
            if (dependencyNotation != null) {
                project.dependencies.create(dependencyNotation)
            } else {
                null
            }
        })
    }
}
```

build.gradle

```
configurations {
    implementation {
        dependencies.addLater(project.provider {
            def dependencyNotation = conditionalLogic()
            if (dependencyNotation != null) {
                return project.dependencies.create(dependencyNotation)
            } else {
                return null
            }
        })
    }
}
```

In this case, `addLater` is used to defer the evaluation of the dependency, allowing it to be added only when certain conditions are met.

In this example, the build script sets a preferred version of a dependency, which will be used if no version is explicitly specified:

Example 2: Preferring a Default Version of a Dependency

build.gradle.kts

```
dependencies {
    implementation("org:foo")
}
```

```
// Can indiscriminately be added by build logic
constraints {
    implementation("org:foo:1.0") {
        version {
            // Applied to org:foo if no other version is specified
            prefer("1.0")
        }
    }
}
}
```

build.gradle

```
dependencies {
    implementation("org:foo")

    // Can indiscriminately be added by build logic
    constraints {
        implementation("org:foo:1.0") {
            version {
                // Applied to org:foo if no other version is specified
                prefer("1.0")
            }
        }
    }
}
```

This ensures that `org:foo` uses version `1.0` unless the user specifies another version.

Modifying Dependency Metadata

Each component pulled from a repository includes metadata, such as its group, name, version, and the various variants it provides along with their artifacts and dependencies.

Occasionally, this metadata might be incomplete or incorrect.

Gradle offers an API to address this issue, allowing you to write **component metadata rules** directly within the build script. These rules are applied after a module's metadata is downloaded, but before it's used in dependency resolution.

Writing a component metadata rule

Component metadata rules are applied within the `components` section of the `dependencies` block in a build script or in the settings script.

These rules can be defined in two ways:

1. **Inline as an Action:** Directly within the `components` section.
2. **As a Separate Class:** Implementing the `ComponentMetadataRule` interface.

While inline actions are convenient for quick experimentation, it's generally recommended to define rules as separate classes.

Rules written as isolated classes can be annotated with `@CacheableRule`, allowing their results to be cached and avoiding re-execution each time dependencies are resolved.

TIP

A rule should always be cacheable to avoid major impacts on build performance and ensure faster build times.

build.gradle.kts

```
@CacheableRule
abstract class TargetJvmVersionRule @Inject constructor(val jvmVersion: Int)
: ComponentMetadataRule {
    @get:Inject abstract val objects: ObjectFactory

    override fun execute(context: ComponentMetadataContext) {
        context.details.withVariant("compile") {
            attributes {
                attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
jvmVersion)
                attribute(Usage.USAGE_ATTRIBUTE,
objects.named(Usage.JAVA_API))
            }
        }
    }
}
dependencies {
    components {
        withModule<TargetJvmVersionRule>("commons-io:commons-io") {
            params(7)
        }
        withModule<TargetJvmVersionRule>("commons-collections:commons-
collections") {
            params(8)
        }
    }
    implementation("commons-io:commons-io:2.6")
    implementation("commons-collections:commons-collections:3.2.2")
}
```

build.gradle

```
@CacheableRule
abstract class TargetJvmVersionRule implements ComponentMetadataRule {
    final Integer jvmVersion
    @Inject TargetJvmVersionRule(Integer jvmVersion) {
        this.jvmVersion = jvmVersion
    }

    @Inject abstract ObjectFactory getObjects()

    void execute(ComponentMetadataContext context) {
        context.details.withVariant("compile") {
            attributes {
                attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
jvmVersion)
                attribute(Usage.USAGE_ATTRIBUTE, objects.named(Usage, Usage
.JAVA_API))
            }
        }
    }
}

dependencies {
    components {
        withModule("commons-io:commons-io", TargetJvmVersionRule) {
            params(7)
        }
        withModule("commons-collections:commons-collections",
TargetJvmVersionRule) {
            params(8)
        }
    }
    implementation("commons-io:commons-io:2.6")
    implementation("commons-collections:commons-collections:3.2.2")
}
```

In this example, the `TargetJvmVersionRule` class implements `ComponentMetadataRule` and is further configured using `ActionConfiguration`.

Gradle enforces isolation of instances of `ComponentMetadataRule`, requiring that all parameters must be `Serializable` or recognized Gradle types.

Additionally, services like `ObjectFactory` can be injected into your rule's constructor using `@Inject`.

A component metadata rule can be applied to all modules using `all(rule)` or to a specific module using `withModule(groupAndName, rule)`. Typically, a rule is tailored to enrich the metadata of a specific module, so the `withModule` API is preferred.

Declaring rules in a central place

NOTE | Declaring component metadata rules in settings is an incubating feature

Component metadata rules can be declared in the `settings.gradle(.kts)` file for the entire build, rather than in each subproject individually. Rules declared in settings are applied to all projects by default unless overridden by project-specific rules.

settings.gradle.kts

```
dependencyResolutionManagement {
    components {
        withModule<GuavaRule>("com.google.guava:guava")
    }
}
```

settings.gradle

```
dependencyResolutionManagement {
    components {
        withModule("com.google.guava:guava", GuavaRule)
    }
}
```

By default, project-specific rules take precedence over settings rules. However, this behavior can be adjusted:

settings.gradle.kts

```
dependencyResolutionManagement {
    rulesMode = RulesMode.PREFER_SETTINGS
}
```

settings.gradle

```
dependencyResolutionManagement {
    rulesMode = RulesMode.PREFER_SETTINGS
}
```

If this method is called and that a project or plugin declares rules, a warning will be issued. You can make this a failure instead by using this alternative:

settings.gradle.kts

```
dependencyResolutionManagement {  
    rulesMode = RulesMode.FAIL_ON_PROJECT_RULES  
}
```

settings.gradle

```
dependencyResolutionManagement {  
    rulesMode = RulesMode.FAIL_ON_PROJECT_RULES  
}
```

The default behavior is equivalent to calling this method:

settings.gradle.kts

```
dependencyResolutionManagement {  
    rulesMode = RulesMode.PREFER_PROJECT  
}
```

settings.gradle

```
dependencyResolutionManagement {  
    rulesMode = RulesMode.PREFER_PROJECT  
}
```

Which parts of metadata can be modified?

The Component Metadata Rules API focuses on the features supported by Gradle Module Metadata and the dependencies API. The key difference between using metadata rules and defining dependencies/artifacts in a build script is that component metadata rules operate directly on [variants](#), whereas build scripts often affect multiple variants at once (e.g., an **api** dependency is applied to both **api** and **runtime** variants of a Java library).

Variants can be modified through the following methods:

- **allVariants:** Modify all variants of a component.
- **withVariant(name):** Modify a specific variant identified by its name.
- **addVariant(name) or addVariant(name, base):** Add a new variant from scratch or copy details from an existing variant (**base**).

The following variant details can be modified:

- **Attributes:** Use the `attributes {}` block to adjust `attributes` that identify the variant.
- **Capabilities:** Use the `withCapabilities {}` block to define the `capabilities` the variant provides.
- **Dependencies:** Use the `withDependencies {}` block to manage the variant's dependencies, including `rich version` constraints.
- **Dependency Constraints:** Use the `withDependencyConstraints {}` block to define the variant's `dependency constraints`, including rich versions.
- **Published Files:** Use the `withFiles {}` block to specify the location of the files that make up the variant's content.

Additionally, several component-level properties can be changed:

- **Component Attributes:** The only `meaningful attribute` here is `org.gradle.status`.
- **Status Scheme:** Influence how the `org.gradle.status` attribute is interpreted during version selection.
- **BelongsTo Property:** Used for `<component_capabilities.adoc#sec:declaring-capabilities-external-modules,version alignment>>` via virtual platforms.

The format of a module's metadata affects how it maps to the variant-centric representation:

- **Gradle Module Metadata:** The data structure is similar to the module's `.module` file.
- **POM Metadata:** For modules published with `.pom` metadata, fixed variants are derived as explained in the "[Mapping POM Files to Variants](#)", section.
- **Ivy Metadata:** If a module was published with an `ivy.xml` file, Ivy configurations can be accessed in place of variants. Their dependencies, constraints, and files can be modified. You can also use `addVariant(name, baseVariantOrConfiguration)` to derive variants from Ivy configurations, such as defining `compile and runtime variants for the Java library plugin`.

Before using component metadata rules to adjust a module's metadata, determine whether the module was published with **Gradle Module Metadata** (`.module` file) or **traditional metadata** (`.pom` or `ivy.xml`):

- **Modules with Gradle Module Metadata:** These typically have complete metadata, but issues can still occur. Only apply component metadata rules if you've clearly identified a problem with the metadata. For dependency resolution issues, first consider using `dependency constraints with rich versions`. If you're developing a library, note that dependency constraints are published as part of your own library's metadata, making it easier to share the solution with consumers. In contrast, component metadata rules apply only within your own build.
- **Modules with Traditional Metadata** (`.pom` or `ivy.xml`): These are more likely to have

incomplete metadata since features like variants and dependency constraints aren't supported in these formats. Such modules might have variants or constraints that were omitted or incorrectly defined as dependencies. In the following sections, we explore examples of OSS modules with incomplete metadata and the rules to add missing information.

As a rule of thumb, you should contemplate if the rule you are writing also works out of context of your build. That is, does the rule still produce a correct and useful result if applied in any other build that uses the module(s) it affects?

Fixing incorrect dependency details

Consider the Jaxen XPath Engine (version 1.1.3) published on Maven Central. Its `pom` file declares several unnecessary dependencies in the `compile` scope, which were later removed in version 1.1.4. If you need to work with version 1.1.3, you can fix the metadata using the following rule:

build.gradle.kts

```
@CacheableRule
abstract class JaxenDependenciesRule: ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        context.details.allVariants {
            withDependencies {
                removeAll { it.group in listOf("dom4j", "jdom", "xerces",
"maven-plugins", "xml-apis", "xom") }
            }
        }
    }
}
```

build.gradle

```
@CacheableRule
abstract class JaxenDependenciesRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext context) {
        context.details.allVariants {
            withDependencies {
                removeAll { it.group in ["dom4j", "jdom", "xerces", "maven-
plugins", "xml-apis", "xom"] }
            }
        }
    }
}
```

In the `withDependencies` block, you have access to the full list of dependencies and can use Java

collection methods to inspect and modify that list. You can also add dependencies using the `add(notation, configureAction)` method. Similarly, you can inspect and modify dependency constraints within the `withDependencyConstraints` block.

In Jaxen version 1.1.4, the `dom4j`, `jdom`, and `xerces` dependencies are still present but marked as **optional**. Optional dependencies are not processed automatically by Gradle or Maven, as they indicate **feature variants** that require additional dependencies. However, the `pom` file lacks information about these features and their corresponding dependencies. This can be represented in Gradle Module Metadata through variants and **capabilities**, which we can add via a component metadata rule.

build.gradle.kts

```
@CacheableRule
abstract class JaxenCapabilitiesRule: ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        context.details.addVariant("runtime-dom4j", "runtime") {
            withCapabilities {
                removeCapability("jaxen", "jaxen")
                addCapability("jaxen", "jaxen-dom4j",
context.details.id.version)
            }
            withDependencies {
                add("dom4j:dom4j:1.6.1")
            }
        }
    }
}
```

build.gradle

```
@CacheableRule
abstract class JaxenCapabilitiesRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext context) {
        context.details.addVariant("runtime-dom4j", "runtime") {
            withCapabilities {
                removeCapability("jaxen", "jaxen")
                addCapability("jaxen", "jaxen-dom4j", context.details.id
.version)
            }
            withDependencies {
                add("dom4j:dom4j:1.6.1")
            }
        }
    }
}
```

In this example, we create a new variant called `runtime-dom4j` using the `addVariant(name, baseVariant)` method. This variant represents an optional feature, defined by the capability `jaxen-dom4j`. We then add the required dependency `dom4j:dom4j:1.6.1` to this feature.

build.gradle.kts

```
dependencies {
    components {
        withModule<JaxenDependenciesRule>("jaxen:jaxen")
        withModule<JaxenCapabilitiesRule>("jaxen:jaxen")
    }
    implementation("jaxen:jaxen:1.1.3")
    runtimeOnly("jaxen:jaxen:1.1.3") {
        capabilities { requireCapability("jaxen:jaxen-dom4j") }
    }
}
```

build.gradle

```
dependencies {
    components {
        withModule("jaxen:jaxen", JaxenDependenciesRule)
        withModule("jaxen:jaxen", JaxenCapabilitiesRule)
    }
    implementation("jaxen:jaxen:1.1.3")
    runtimeOnly("jaxen:jaxen:1.1.3") {
        capabilities { requireCapability("jaxen:jaxen-dom4j") }
    }
}
```

By applying these rules, Gradle uses the enriched metadata to correctly resolve the optional dependencies when the `jaxen-dom4j` feature is required.

Making variants published as classified jars explicit

In modern builds, variants are often published as separate artifacts, each represented by its own jar file. For example, libraries may provide distinct jars for different Java versions, ensuring that the correct version is used at runtime or compile time based on the environment.

For instance, version `0.7.9` of the asynchronous programming library Quasar, published on [Maven Central](#), includes both `quasar-core-0.7.9.jar` and `quasar-core-0.7.9-jdk8.jar`. Publishing jars with a classifier, such as `jdk8`, is common practice in Maven repositories. However, neither Maven nor Gradle metadata provides information about these classified jars. As a result, there is no clear way to determine their existence or any differences, such as dependencies, between the variants.

In Gradle Module Metadata, variant information would be present. For the already published Quasar library, we can add this information using the following rule:

build.gradle.kts

```
@CacheableRule
abstract class QuasarRule: ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        listOf("compile", "runtime").forEach { base ->
            context.details.addVariant("jdk8${base.capitalize()}", base) {
                attributes {
                    attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
8)
                }
                withFiles {
                    removeAllFiles()
                    addFile("${context.details.id.name}-
${context.details.id.version}-jdk8.jar")
                }
            }
            context.details.withVariant(base) {
                attributes {
                    attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
7)
                }
            }
        }
    }
}
```

build.gradle

```
@CacheableRule
abstract class QuasarRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext context) {
        ["compile", "runtime"].each { base ->
            context.details.addVariant("jdk8${base.capitalize()}", base) {
                attributes {
                    attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
8)
                }
                withFiles {
                    removeAllFiles()
                    addFile("${context.details.id.name}-${context.details.id
.version}-jdk8.jar")
                }
            }
        }
    }
}
```

```

        context.details.withVariant(base) {
            attributes {
                attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
7)
            }
        }
    }
}

```

In this case, the `jdk8` classifier clearly indicates the target Java version, which corresponds to a known attribute in the Java ecosystem. Since we need both compile and runtime variants for Java 8, we create two new variants using the existing compile and runtime variants as a base. This ensures that all other Java ecosystem attributes are set correctly, and dependencies are carried over.

We assign the `TARGET_JVM_VERSION_ATTRIBUTE` to `8` for both new variants, remove any existing files with `removeAllFiles()`, and then add the `jdk8` jar using `addFile()`. Removing the files is necessary because the reference to the main jar `quasar-core-0.7.9.jar` is copied from the base variant.

Finally, we enrich the existing compile and runtime variants with the information that they target Java 7 using `attribute(TARGET_JVM_VERSION_ATTRIBUTE, 7)`.

With these changes, you can now request Java 8 versions for all dependencies on the compile classpath, and Gradle will automatically select the best-fitting variant. In the case of Quasar, this will be the `jdk8Compile` variant, which exposes the `quasar-core-0.7.9-jdk8.jar`.

build.gradle.kts

```

configurations["compileClasspath"].attributes {
    attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE, 8)
}
dependencies {
    components {
        withModule<QuasarRule>("co.paralleluniverse:quasar-core")
    }
    implementation("co.paralleluniverse:quasar-core:0.7.9")
}

```

build.gradle

```

configurations.compileClasspath.attributes {
    attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE, 8)
}
dependencies {

```

```
components {
    withModule("co.paralleluniverse:quasar-core", QuasarRule)
}
implementation("co.paralleluniverse:quasar-core:0.7.9")
}
```

With this configuration, Gradle will select the Java 8 variant of Quasar for the compile classpath.

Making variants encoded in versions explicit

Another solution to publish multiple alternatives for the same library is the usage of a versioning pattern as done by the popular Guava library. Here, each new version is published twice by appending the classifier to the version instead of the jar artifact. In the case of Guava 28 for example, we can find a *28.0-jre* (Java 8) and *28.0-android* (Java 6) version on [Maven central](#). The advantage of using this pattern when working only with pom metadata is that both variants are discoverable through the version. The disadvantage is that there is no information as to what the different version suffixes mean semantically. So in the case of conflict, Gradle would just pick the highest version when comparing the version strings.

Turning this into proper variants is a bit more tricky, as Gradle first selects a version of a module and then selects the best fitting variant. So the concept that variants are encoded as versions is not supported directly. However, since both variants are always published together we can assume that the files are physically located in the same repository. And since they are published with Maven repository conventions, we know the location of each file if we know module name and version. We can write the following rule:

```
build.gradle.kts
```

```
@CacheableRule
abstract class GuavaRule: ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        val variantVersion = context.details.id.version
        val version = variantVersion.substring(0, variantVersion.indexOf("-"))

        listOf("compile", "runtime").forEach { base ->
            mapOf(6 to "android", 8 to "jre").forEach { (targetJvmVersion,
jarName) ->

context.details.addVariant("jdk$targetJvmVersion${base.capitalize()}", base)
{
            attributes {

attributes.attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
targetJvmVersion)

        }
        withFiles {
```


build.gradle.kts

```
configurations["compileClasspath"].attributes {
    attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE, 6)
}
dependencies {
    components {
        withModule<GuavaRule>("com.google.guava:guava")
    }
    // '23.3-android' and '23.3-jre' are now the same as both offer both
    variants
        implementation("com.google.guava:guava:23.3+")
}
```

build.gradle

```
configurations.compileClasspath.attributes {
    attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE, 6)
}
dependencies {
    components {
        withModule("com.google.guava:guava", GuavaRule)
    }
    // '23.3-android' and '23.3-jre' are now the same as both offer both
    variants
        implementation("com.google.guava:guava:23.3+")
}
```

Adding variants for native jars

Jars with classifiers are also used to separate parts of a library for which multiple alternatives exists, for example native code, from the main artifact. This is for example done by the Lightweight Java Game Library (LWJGL), which publishes several platform specific jars to [Maven central](#) from which always one is needed, in addition to the main jar, at runtime. It is not possible to convey this information in pom metadata as there is no concept of putting multiple artifacts in relation through the metadata. In Gradle Module Metadata, each variant can have arbitrary many files and we can leverage that by writing the following rule:

build.gradle.kts

```
@CacheableRule
abstract class LwjglRule: ComponentMetadataRule {
    data class NativeVariant(val os: String, val arch: String, val
```

```

classifier: String)

    private val nativeVariants = listOf(
        NativeVariant(OperatingSystemFamily.LINUX, "arm32", "natives-
linux-arm32"),
        NativeVariant(OperatingSystemFamily.LINUX, "arm64", "natives-
linux-arm64"),
        NativeVariant(OperatingSystemFamily.WINDOWS, "x86", "natives-
windows-x86"),
        NativeVariant(OperatingSystemFamily.WINDOWS, "x86-64", "natives-
windows"),
        NativeVariant(OperatingSystemFamily.MACOS, "x86-64", "natives-
macos")
    )

    @get:Inject abstract val objects: ObjectFactory

    override fun execute(context: ComponentMetadataContext) {
        context.details.withVariant("runtime") {
            attributes {

attributes.attribute(OperatingSystemFamily.OPERATING_SYSTEM_ATTRIBUTE,
objects.named("none"))

attributes.attribute(MachineArchitecture.ARCHITECTURE_ATTRIBUTE,
objects.named("none"))
            }
        }
        nativeVariants.forEach { variantDefinition ->
            context.details.addVariant("${variantDefinition.classifier}-
runtime", "runtime") {
                attributes {

attributes.attribute(OperatingSystemFamily.OPERATING_SYSTEM_ATTRIBUTE,
objects.named(variantDefinition.os))

attributes.attribute(MachineArchitecture.ARCHITECTURE_ATTRIBUTE,
objects.named(variantDefinition.arch))
                }
                withFiles {
                    addFile("${context.details.id.name}-
${context.details.id.version}-${variantDefinition.classifier}.jar")
                }
            }
        }
    }
}

```

```
@CacheableRule
abstract class LwjglRule implements ComponentMetadataRule { //val os: String,
val arch: String, val classifier: String)
    private def nativeVariants = [
        [os: OperatingSystemFamily.LINUX,    arch: "arm32", classifier:
"natives-linux-arm32"],
        [os: OperatingSystemFamily.LINUX,    arch: "arm64", classifier:
"natives-linux-arm64"],
        [os: OperatingSystemFamily.WINDOWS,  arch: "x86",    classifier:
"natives-windows-x86"],
        [os: OperatingSystemFamily.WINDOWS,  arch: "x86-64", classifier:
"natives-windows"],
        [os: OperatingSystemFamily.MACOS,    arch: "x86-64", classifier:
"natives-macos"]
    ]

    @Inject abstract ObjectFactory getObjects()

    void execute(ComponentMetadataContext context) {
        context.details.withVariant("runtime") {
            attributes {
                attributes.attribute(OperatingSystemFamily
.OPERATING_SYSTEM_ATTRIBUTE, objects.named(OperatingSystemFamily, "none"))
                attributes.attribute(MachineArchitecture
.ARCHITECTURE_ATTRIBUTE, objects.named(MachineArchitecture, "none"))
            }
        }
        nativeVariants.each { variantDefinition ->
            context.details.addVariant("${variantDefinition.classifier}
-runtime", "runtime") {
                attributes {
                    attributes.attribute(OperatingSystemFamily
.OPERATING_SYSTEM_ATTRIBUTE, objects.named(OperatingSystemFamily,
variantDefinition.os))
                    attributes.attribute(MachineArchitecture
.ARCHITECTURE_ATTRIBUTE, objects.named(MachineArchitecture,
variantDefinition.arch))
                }
                withFiles {
                    addFile("${context.details.id.name}-${context.details.id
.version}-${variantDefinition.classifier}.jar")
                }
            }
        }
    }
}
```

This rule is quite similar to the Quasar library example above. Only this time we have five different runtime variants we add and nothing we need to change for the compile variant. The runtime variants are all based on the existing *runtime* variant and we do not change any existing information. All Java ecosystem attributes, the dependencies and the main jar file stay part of each of the runtime variants. We only set the additional attributes `OPERATING_SYSTEM_ATTRIBUTE` and `ARCHITECTURE_ATTRIBUTE` which are defined as part of Gradle's [native support](#). And we add the corresponding native jar file so that each runtime variant now carries two files: the main jar and the native jar.

In the build script, we can now request a specific variant and Gradle will fail with a selection error if more information is needed to make a decision.

Gradle is able to understand the common case where a single attribute is missing that would have removed the ambiguity. In this case, rather than listing information about all attributes on all available variants, Gradle helpfully lists only possible values for that attribute along with the variants each value would select.

build.gradle.kts

```
configurations["runtimeClasspath"].attributes {
    attribute(OperatingSystemFamily.OPERATING_SYSTEM_ATTRIBUTE,
objects.named("windows"))
}
dependencies {
    components {
        withModule<LwjglRule>("org.lwjgl:lwjgl")
    }
    implementation("org.lwjgl:lwjgl:3.2.3")
}
```

build.gradle

```
configurations["runtimeClasspath"].attributes {
    attribute(OperatingSystemFamily.OPERATING_SYSTEM_ATTRIBUTE, objects.
named(OperatingSystemFamily, "windows"))
}
dependencies {
    components {
        withModule("org.lwjgl:lwjgl", LwjglRule)
    }
    implementation("org.lwjgl:lwjgl:3.2.3")
}
```

Gradle fails to select a variant because a machine architecture needs to be chosen:

```
> Could not resolve all files for configuration ':runtimeClasspath'.
> Could not resolve org.lwjgl:lwjgl:3.2.3.
   Required by:
       project :
       > The consumer was configured to find a library for use during runtime,
compatible with Java 11, packaged as a jar, preferably optimized for standard JVMs,
and its dependencies declared externally, as well as attribute
'org.gradle.native.operatingSystem' with value 'windows'. There are several available
matching variants of org.lwjgl:lwjgl:3.2.3
       The only attribute distinguishing these variants is
'org.gradle.native.architecture'. Add this attribute to the consumer's configuration
to resolve the ambiguity:
       - Value: 'x86-64' selects variant: 'natives-windows-runtime'
       - Value: 'x86' selects variant: 'natives-windows-x86-runtime'
```

Making different flavors of a library available through capabilities

Because it is difficult to model [optional feature variants](#) as separate jars with pom metadata, libraries sometimes comprise different jars with different feature sets. That is, instead of composing your flavor of the library from different feature variants, you select one of the pre-composed variants (offering everything in one jar). One such library is the well-known dependency injection framework Guice, published on [Maven central](#), which offers a complete flavor (the main jar) and a reduced variant without aspect-oriented programming support ([guice-4.2.2-no_aop.jar](#)). That second variant with a classifier is not mentioned in the pom metadata. With the following rule, we create compile and runtime variants based on that file and make it selectable through a capability named `com.google.inject:guice-no_aop`.

build.gradle.kts

```
@CacheableRule
abstract class GuiceRule: ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        listOf("compile", "runtime").forEach { base ->
            context.details.addVariant("noAop${base.capitalize()}", base) {
                withCapabilities {
                    addCapability("com.google.inject", "guice-no_aop",
context.details.id.version)
                }
                withFiles {
                    removeAllFiles()
                    addFile("guice-${context.details.id.version}-no_aop.jar")
                }
                withDependencies {
                    removeAll { it.group == "aopalliance" }
                }
            }
        }
    }
}
```

```
}
}
```

build.gradle

```
@CacheableRule
abstract class GuiceRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext context) {
        ["compile", "runtime"].each { base ->
            context.details.addVariant("noAop${base.capitalize()}", base) {
                withCapabilities {
                    addCapability("com.google.inject", "guice-no_aop",
context.details.id.version)
                }
                withFiles {
                    removeAllFiles()
                    addFile("guice-${context.details.id.version}-no_aop.jar")
                }
                withDependencies {
                    removeAll { it.group == "aopalliance" }
                }
            }
        }
    }
}
```

The new variants also have the dependency on the standardized aop interfaces library `aopalliance:aopalliance` removed, as this is clearly not needed by these variants. Again, this is information that cannot be expressed in pom metadata. We can now select a `guice-no_aop` variant and will get the correct jar file **and** the correct dependencies.

build.gradle.kts

```
dependencies {
    components {
        withModule<GuiceRule>("com.google.inject:guice")
    }
    implementation("com.google.inject:guice:4.2.2") {
        capabilities { requireCapability("com.google.inject:guice-no_aop") }
    }
}
```

build.gradle

```
dependencies {
    components {
        withModule("com.google.inject:guice", GuiceRule)
    }
    implementation("com.google.inject:guice:4.2.2") {
        capabilities { requireCapability("com.google.inject:guice-no_aop") }
    }
}
```

Adding missing capabilities to detect conflicts

Another usage of capabilities is to express that two different modules, for example `log4j` and `log4j-over-slf4j`, provide alternative implementations of the same thing. By declaring that both provide the same capability, Gradle only accepts one of them in a dependency graph. This example, and how it can be tackled with a component metadata rule, is described in detail in the [feature modelling](#) section.

Making Ivy modules variant-aware

Modules published using Ivy do not have variants available by default.

However, *Ivy configurations* can be mapped to variants as the `addVariant(name, baseVariantOrConfiguration)` accepts any Ivy configuration that was published as base. This can be used, for example, to define runtime and compile variants. An example of a corresponding rule can be found [here](#). Ivy details of Ivy configurations (e.g. dependencies and files) can also be modified using the `withVariant(configurationName)` API. However, modifying attributes or capabilities on Ivy configurations has no effect.

For very Ivy specific use cases, the component metadata rules API also offers access to other details only found in Ivy metadata. These are available through the [IvyModuleDescriptor](#) interface and can be accessed using `getDescriptor(IvyModuleDescriptor)` on the [ComponentMetadataContext](#).

build.gradle.kts

```
@CacheableRule
abstract class IvyComponentRule : ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        val descriptor = context.getDescriptor(IvyModuleDescriptor::class)
        if (descriptor != null && descriptor.branch == "testing") {
            context.details.status = "rc"
        }
    }
}
```

build.gradle

```
@CacheableRule
abstract class IvyComponentRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext context) {
        def descriptor = context.getDescriptor(IvyModuleDescriptor)
        if (descriptor != null && descriptor.branch == "testing") {
            context.details.status = "rc"
        }
    }
}
```

Filter using Maven metadata

For Maven specific use cases, the component metadata rules API also offers access to other details only found in POM metadata. These are available through the [PomModuleDescriptor](#) interface and can be accessed using `getDescriptor(PomModuleDescriptor)` on the [ComponentMetadataContext](#).

build.gradle.kts

```
@CacheableRule
abstract class MavenComponentRule : ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        val descriptor = context.getDescriptor(PomModuleDescriptor::class)
        if (descriptor != null && descriptor.packaging == "war") {
            // ...
        }
    }
}
```

build.gradle

```
@CacheableRule
abstract class MavenComponentRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext context) {
        def descriptor = context.getDescriptor(PomModuleDescriptor)
        if (descriptor != null && descriptor.packaging == "war") {
            // ...
        }
    }
}
```

Modifying metadata on the component level for alignment

While all the examples above made modifications to variants of a component, there is also a limited set of modifications that can be done to the metadata of the component itself. This information can influence the [version selection](#) process for a module during dependency resolution, which is performed *before* one or multiple variants of a component are selected.

The first API available on the component is `belongsTo()` to create virtual platforms for aligning versions of multiple modules without Gradle Module Metadata. It is explained in detail in the section on [aligning versions of modules not published with Gradle](#).

Modifying metadata on the component level for version selection based on status

Gradle and Gradle Module Metadata also allow attributes to be set on the whole component instead of a single variant. Each of these attributes carries special semantics as they influence version selection which is done *before* variant selection. While variant selection can handle [any custom attribute](#), version selection only considers attributes for which specific semantics are implemented. At the moment, the only attribute with meaning here is `org.gradle.status`.

The `org.gradle.status` module attribute indicates the lifecycle status or maturity level of a module or library:

1. **integration**: This indicates that the module is under active development and may not be stable.
2. **milestone**: A module with this status is more mature than one marked as **integration**.
3. **release**: This status signifies that the module is stable and officially released.

It is therefore recommended to only modify this attribute, if any, on the component level. A dedicated API `setStatus(value)` is available for this. To modify another attribute for all variants of a component `withAllVariants { attributes {} }` should be utilised instead.

A module's status is taken into consideration when a [latest version selector](#) is resolved. Specifically, `latest.someStatus` will resolve to the highest module version that has status `someStatus` or a more mature status. For example, `latest.integration` will select the highest module version regardless of its status (because **integration** is the least mature status as explained below), whereas `latest.release` will select the highest module version with status **release**.

The interpretation of the status can be influenced by changing a module's *status scheme* through the `setStatusScheme(valueList)` API. This concept models the different levels of maturity that a module transitions through over time with different publications. The default status scheme, ordered from least to most mature status, is **integration**, **milestone**, **release**. The `org.gradle.status` attribute must be set, to one of the values in the component's status scheme. Thus each component always has a status which is determined from the metadata as follows:

- Gradle Module Metadata: the value that was published for the `org.gradle.status` attribute on the component
- Ivy metadata: `status` defined in the `ivy.xml`, defaults to **integration** if missing
- Pom metadata: **integration** for modules with a SNAPSHOT version, **release** for all others

The following example demonstrates **latest** selectors based on a custom status scheme declared in a component metadata rule that applies to all modules:

build.gradle.kts

```
@CacheableRule
abstract class CustomStatusRule : ComponentMetadataRule {
    override fun execute(context: ComponentMetadataContext) {
        context.details.statusScheme = listOf("nightly", "milestone", "rc",
"release")
        if (context.details.status == "integration") {
            context.details.status = "nightly"
        }
    }
}

dependencies {
    components {
        all<CustomStatusRule>()
    }
    implementation("org.apache.commons:commons-lang3:latest.rc")
}
```

build.gradle

```
@CacheableRule
abstract class CustomStatusRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext context) {
        context.details.statusScheme = ["nightly", "milestone", "rc",
"release"]
        if (context.details.status == "integration") {
            context.details.status = "nightly"
        }
    }
}

dependencies {
    components {
        all(CustomStatusRule)
    }
    implementation("org.apache.commons:commons-lang3:latest.rc")
}
```

Compared to the default scheme, the rule inserts a new status **rc** and replaces **integration** with **nightly**. Existing modules with the status **integration** are mapped to **nightly**.

Dependency Caching

Gradle contains a highly sophisticated dependency caching mechanism, which seeks to minimise the number of remote requests made in dependency resolution, while striving to guarantee that the results of dependency resolution are correct and reproducible.

1. **Local Cache:** Gradle caches dependencies locally to avoid repeated downloads. The cache is located in the `.gradle` directory under the user's home folder (e.g., `~/.gradle/caches/modules-2`). When a dependency is requested, Gradle first checks this local cache before attempting to fetch it from remote repositories.
2. **Changing Dependencies:** By default, Gradle treats dependencies marked as "changing" (e.g., SNAPSHOT or dynamic dependencies) differently and refreshes them more frequently. The caching times for these dependencies can be altered programmatically.
3. **Offline Mode:** Gradle can run in **offline mode**, using only the cached dependencies without trying to download anything from remote repositories. You can enable offline mode with the `--offline` flag, ensuring that your build only uses cached artifacts.
4. **Refreshing Dependencies:** To force Gradle to update its dependencies, use the `--refresh-dependencies` flag. This option instructs Gradle to bypass the cache and check for updated artifacts in remote repositories. Gradle downloads them, but only if it detects a change, using hashes to avoid unnecessary downloads.

1. The dependency cache

The Gradle dependency cache consists of two storage types located under `$GRADLE_USER_HOME/caches`:

1. A file-based store of downloaded artifacts, including binaries like jars as well as raw downloaded meta-data like POM files and Ivy files. Artifacts are stored under a checksum, so name clashes will not cause issues.
2. A binary store of resolved module metadata, including the results of resolving dynamic versions, module descriptors, and artifacts.

Separate metadata cache

Gradle keeps a record of various aspects of dependency resolution in binary format in the metadata cache.

The information stored in the metadata cache includes:

- The result of resolving a dynamic version (e.g. `1.+`) to a concrete version (e.g. `1.2`).
- The resolved module metadata for a particular module, including module artifacts and module dependencies.
- The resolved artifact metadata for a particular artifact, including a pointer to the downloaded artifact file.
- The *absence* of a particular module or artifact in a particular repository, eliminating repeated attempts to access a resource that does not exist.

Every entry in the metadata cache includes a record of the repository that provided the information as well as a timestamp that can be used for cache expiry.

Repository caches are independent

As described above, for each repository there is a separate metadata cache. A repository is identified by its URL, type and layout.

If a module or artifact has not been previously resolved from *this repository*, Gradle will attempt to resolve the module against the repository. This will always involve a remote lookup on the repository, however in many cases [no download will be required](#).

Dependency resolution will fail if required artifacts aren't available in the repository from which they were originally resolved. Once resolved from a specific repository, artifacts become "sticky," meaning Gradle will avoid resolving them from other repositories to prevent unexpected or potentially unsafe changes in artifact sources. This ensures consistency across environments, but it may also lead to failures if repositories differ between machines.

Repository independence allows builds to be isolated from each other. This is a key feature to create builds that are reliable and reproducible in any environment.

Artifact reuse

Before downloading an artifact, Gradle attempts to retrieve the artifact's checksum by downloading an associated `.sha512`, `.sha256`, `.sha1`, or `.md5` file (attempting each in order).

If the checksum is available, Gradle skips the download if an artifact with the same ID and checksum already exists. However, if the checksum cannot be retrieved from the remote server, Gradle proceeds to download the artifact but will ignore it if it matches an existing one.

Gradle also tries to reuse artifacts from the local Maven repository. If an artifact previously downloaded by Maven is a match, Gradle will use it, provided it can be verified against the checksum from the remote server.

Checksum based storage

It is possible for different repositories to provide a different binary artifact in response to the same artifact identifier.

This is often the case with Maven SNAPSHOT artifacts, but can also be true for any artifact which is republished without changing its identifier. By caching artifacts based on their checksum, Gradle is able to maintain multiple versions of the same artifact. This means that when resolving against one repository Gradle will never overwrite the cached artifact file from a different repository. This is done without requiring a separate artifact file store per repository.

Cache locking

The Gradle dependency cache uses file-based locking to ensure that it can safely be used by multiple Gradle processes concurrently. The lock is held whenever the binary metadata store is being read or written, but is released for slow operations such as downloading remote artifacts.

This concurrent access is only supported if the different Gradle processes can communicate together. This is usually *not the case* for containerized builds.

Cache cleanup

Gradle tracks which artifacts in the dependency cache are accessed. Based on this information, the cache is periodically scanned (no more than once every 24 hours) to identify artifacts that haven't been used in over 30 days. These obsolete artifacts are then deleted to prevent the cache from growing indefinitely.

You can learn more about cache cleanup in [Gradle-managed Directories](#).

2. Changing dependencies

Gradle treats dependencies marked as "changing" (such as SNAPSHOT dependencies) differently from regular dependencies, refreshing them more frequently to ensure that you are always using the latest version.

To declare a dependency as changing, you can set the `changing = true` attribute in your dependency declaration. This is useful for dependencies expected to change frequently without a new version number:

```
dependencies {  
    implementation("com.example:some-library:1.0-SNAPSHOT") // Automatically gets  
    treated as changing  
    implementation("com.example:my-library:1.0") { // Must be explicitly set as  
        changing  
        changing = true  
    }  
}
```

Caching changing dependencies

By default, Gradle caches these dependencies (including dynamic versions and changing modules) for 24 hours, meaning it does not contact remote repositories for new versions during this time.

To have Gradle check for newer versions more frequently or with every build, you can adjust the caching threshold or time-to-live (TTL) settings accordingly.

NOTE

Using a short TTL threshold for dynamic or changing versions may result in longer build times due to increased remote repository accesses.

You can fine-tune certain aspects of caching programmatically using the [ResolutionStrategy](#) for a configuration. The programmatic approach is useful if you want to change the settings permanently.

To change how long Gradle will cache the resolved version for a dynamic version, use:

build.gradle.kts

```
configurations.all {  
    resolutionStrategy.cacheDynamicVersionsFor(10, "minutes")  
}
```

build.gradle

```
configurations.all {  
    resolutionStrategy.cacheDynamicVersionsFor 10, 'minutes'  
}
```

To change how long Gradle will cache the metadata and artifacts for a changing module, use:

build.gradle.kts

```
configurations.all {  
    resolutionStrategy.cacheChangingModulesFor(4, "hours")  
}
```

build.gradle

```
configurations.all {  
    resolutionStrategy.cacheChangingModulesFor 4, 'hours'  
}
```

3. Using offline mode

The `--offline` command-line switch instructs Gradle to use dependency modules from the cache, regardless of whether they are due to be checked again. When running with `offline`, Gradle will not attempt to access the network for dependency resolution. If the required modules are not in the dependency cache, the build will fail.

4. Force-refreshing dependencies

You can control the behavior of dependency caching for a distinct build invocation from the command line. Command line options help make a selective, ad-hoc choice for a single build execution.

At times, the Gradle Dependency Cache can become out of sync with the actual state of the configured repositories. Perhaps a repository was initially misconfigured, or maybe a "non-changing" module was published incorrectly. To refresh all dependencies in the dependency cache, use the `--refresh-dependencies` option on the command line.

The `--refresh-dependencies` option tells Gradle to ignore all cached entries for resolved modules and artifacts. A fresh resolve will be performed against all configured repositories, with dynamic versions recalculated, modules refreshed, and artifacts downloaded. However, where possible Gradle will check if the previously downloaded artifacts are valid before downloading again. This is done by comparing published checksum values in the repository with the checksum values for existing downloaded artifacts.

Refreshing dependencies will cause Gradle to invalidate its listing caches. However:

- it will perform HTTP HEAD requests on metadata files but *will not re-download them* if they are identical
- it will perform HTTP HEAD requests on artifact files but *will not re-download them* if they are identical

In other words, refreshing dependencies *only* has an impact if you actually use dynamic dependencies *or* that you have changing dependencies that you were not aware of (in which case it is your responsibility to declare them correctly to Gradle as changing dependencies).

It's a common misconception to think that using `--refresh-dependencies` will force the download of dependencies. This is **not** the case: Gradle will only perform what is strictly required to refresh the dynamic dependencies. This *may* involve downloading new listings, metadata files, or even artifacts, but the impact is minimal if nothing changed.

Dealing with ephemeral builds

It's a common practice to run builds in ephemeral containers. A container is typically spawned to only execute a single build before it is destroyed. This can become a practical problem when a build depends on a lot of dependencies which each container has to re-download. To help with this scenario, Gradle provides a couple of options:

- [copying the dependency cache](#) into each container
- [sharing a read-only dependency cache](#) between multiple containers

Copying and reusing the cache

The dependency cache, both the file and metadata parts, are fully encoded using relative paths. This means that it is perfectly possible to copy a cache around and see Gradle benefit from it.

The path that can be copied is `$GRADLE_USER_HOME/caches/modules-<version>`. The only constraint is placing it using the same structure at the destination, where the value of `GRADLE_USER_HOME` can be different.

Do not copy the `*.lock` or `gc.properties` files if they exist.

Note that creating the cache and consuming it should be done using compatible Gradle version, as shown in the table below. Otherwise, the build might still require some interactions with remote repositories to complete missing information, which might be available in a different version. If multiple incompatible Gradle versions are in play, all should be used when seeding the cache.

Module cache version	File cache version	Metadata cache version	Gradle version(s)
modules-2	files-2.1	metadata-2.95	Gradle 6.1 to Gradle 6.3
modules-2	files-2.1	metadata-2.96	Gradle 6.4 to Gradle 6.7
modules-2	files-2.1	metadata-2.97	Gradle 6.8 to Gradle 7.4
modules-2	files-2.1	metadata-2.99	Gradle 7.5 to Gradle 7.6.1
modules-2	files-2.1	metadata-2.101	Gradle 7.6.2
modules-2	files-2.1	metadata-2.100	Gradle 8.0
modules-2	files-2.1	metadata-2.105	Gradle 8.1
modules-2	files-2.1	metadata-2.106	Gradle 8.2 to Gradle 8.10.2
modules-2	files-2.1	metadata-2.107	Gradle 8.11 and above

Sharing the dependency cache with other Gradle instances

Instead of [copying the dependency cache into each container](#), it's possible to mount a shared, read-only directory that will act as a dependency cache for all containers. This cache, unlike the classical dependency cache, is accessed without locking, making it possible for multiple builds to read from the cache concurrently. It's important that the read-only cache is not written to when other builds may be reading from it.

When using the shared read-only cache, Gradle looks for dependencies (artifacts or metadata) in both the writable cache in the local Gradle User Home directory and the shared read-only cache. If a dependency is present in the read-only cache, it will not be downloaded. If a dependency is missing from the read-only cache, it will be downloaded and added to the writable cache. In practice, this means that the writable cache will only contain dependencies that are unavailable in the read-only cache.

The read-only cache should be sourced from a Gradle dependency cache that already contains some of the required dependencies. The cache can be incomplete; however, an empty shared cache will only add overhead.

NOTE | The shared read-only dependency cache is an incubating feature.

The first step in using a shared dependency cache is to create one by copying of an existing *local* cache. For this you need to follow the [instructions above](#).

Then set the `GRADLE_RO_DEP_CACHE` environment variable to point to the directory containing the cache:

```
$GRADLE_RO_DEP_CACHE
|-- modules-2 : the read-only dependency cache, should be mounted with read-only
```

```
privileges
```

```
$GRADLE_HOME
```

```
|-- caches
```

```
    |-- modules-2 : the container specific dependency cache, should be writable
```

```
    |-- ...
```

```
|-- ...
```

In a CI environment, it's a good idea to have one build which "seeds" a Gradle dependency cache, which is then *copied* to a different directory or distributed, for example, as a Docker volume. This directory can then be used as the read-only cache for other builds. You shouldn't use an existing Gradle installation cache as the read-only cache, because this directory may contain locks and may be modified by the seeding build.

UNDERSTANDING DEPENDENCY RESOLUTION

Understanding the Dependency Resolution Model

This chapter explains how dependency resolution works within Gradle. After learning how to declare [dependencies](#) and [specify the versions](#) your project should use, the next step is understanding how these declarations are combined during the dependency resolution process.

Dependency resolution happens in two key phases, repeated until the entire dependency graph is constructed:

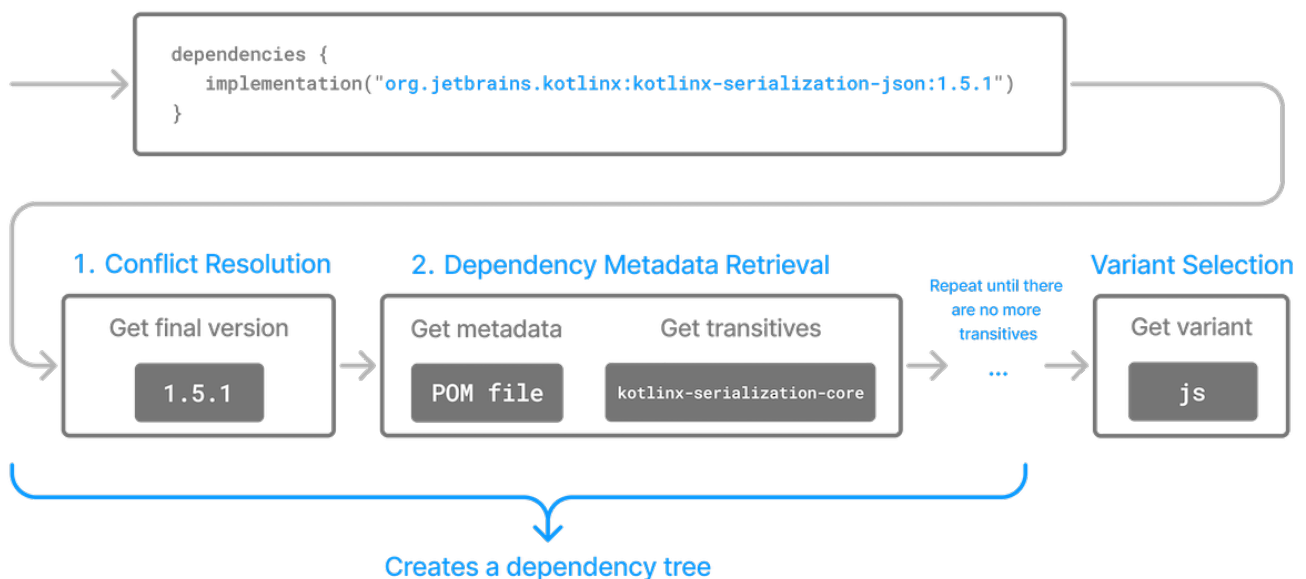
1. Conflict Resolution:

- When a new dependency is introduced, Gradle resolves any conflicts to determine the version that should be added to the graph.
- Gradle will apply its conflict resolution rules (e.g., "latest version wins" or custom version resolution strategies) to determine the version to be used.

2. Dependency Metadata Retrieval:

- Once a specific dependency (a module with a version) is included in the graph, Gradle retrieves its metadata (such as POM, Ivy, or module metadata) to:
 - add its own dependencies (transitives) to the graph.
 - understand the available variants of that dependency.

This process continues until the entire dependency tree is resolved.



Phase 1. Conflict resolution

When performing dependency resolution, Gradle handles two types of conflicts:

1. **Version conflicts:** Which occur when multiple dependencies request the same dependency but with different versions. Gradle must choose which version to include in the graph.
2. **Implementation / Capability conflicts:** Which occur when the dependency graph contains different modules that provide the same functionality or *capability*. Gradle resolves these by selecting one module to avoid duplicate implementations.

The dependency resolution process is highly customizable and many APIs can influence the process.

A. Version conflicts

A version conflict can occur when two components:

- Depend on the same module, such as `com.google.guava:guava`
- But on different versions, for example, `20.0` and `25.1-android`:
 - Our project directly depends on `com.google.guava:guava:20.0`
 - Our project also depends on `com.google.inject:guice:4.2.2`, which in turn depends on `com.google.guava:guava:25.1-android`

Gradle must resolve this conflict by selecting one version to include in the dependency graph.

Gradle considers all requested versions across the dependency graph and, by default, selects the *highest* version. Detailed version ordering is explained in [version ordering](#).

Gradle also supports the concept of [rich version declarations](#), which means that what constitutes the "highest" version depends on how the versions were declared:

- **Without ranges:** The highest non-rejected version will be selected.
 - If a `strictly` version is declared that is lower than the highest, resolution will fail.
- **With ranges:**
 - If a non-range version fits within the range or is higher than the upper bound, it will be selected.
 - If only ranges exist, the selection depends on the intersection of those ranges:
 - If ranges overlap, the highest *existing* version in the intersection is selected.
 - If no clear intersection exists, the highest version from the largest range will be selected. If no version exists in the highest range, the resolution fails.
 - If a `strictly` version is declared that is lower than the highest, resolution will fail.

For version ranges, Gradle needs to perform intermediate metadata lookups to determine what variations are available, as explained in [Phase 2. Dependency metadata retrieval](#).

Versions with qualifiers

The term "qualifier" refers to the portion of a version string that comes after a non-dot separator, like a hyphen or underscore.

For example:

Original version	Base version	Qualifier
1.2.3	1.2.3	<none>
1.2-3	1.2	3
1_alpha	1	alpha
abc	abc	<none>
1.2b3	1.2	b3
abc.1+3	abc.1	3
b1-2-3.3	b	1-2-3.3

As you can see separators are any of the `.`, `-`, `_`, `+` characters, plus the empty string when a numeric and a non-numeric part of the version are next to each-other.

By default, Gradle gives preference to versions without qualifiers when resolving conflicts.

For example, in version `1.0-beta`, the base form is `1.0`, and `beta` is the qualifier. Versions without qualifiers are considered more stable, so Gradle will prioritize them.

Here are a few examples to clarify:

- `1.0.0` (no qualifier)
- `1.0.0-beta` (qualifier: `beta`)
- `2.1-rc1` (qualifier: `rc1`)

Even if the qualifier is lexicographically higher, Gradle will typically consider a version like `1.0.0` higher than `1.0.0-beta`.

When resolving conflicts between versions, Gradle applies the following logic:

1. **Base version comparison:** Gradle first selects versions with the highest base version, ignoring any qualifiers. All others are discarded.
2. **Qualifier handling:** If there are still multiple versions with the same base version, Gradle picks one with a preference for versions without qualifiers (i.e., release versions). If all versions have qualifiers, Gradle will consider the qualifier's order, preferring more stable ones like "release" over others such as "beta" or "alpha."

B. Implementation / Capability conflicts

Gradle uses **variants** and **capabilities** to define what a module **provides**.

Variants are essentially different forms of a dependency, often based on factors such as platform (e.g., JVM or Android), or configuration (e.g., compile, runtime).

Capabilities are a way to express mutually exclusive variants of a dependency.

Conflicts arise in the following scenarios:

- **Incompatible variants:** When two modules attempt to select different, incompatible variants of a dependency.
- **Same capability:** When multiple modules declare the same capability, creating an overlap in functionality.

For more details on how variant selection works and how it enables flexible dependency management, refer to the [Understanding Variant Selection](#) below.

Phase 2. Dependency metadata retrieval

Gradle requires module metadata in the dependency graph for two reasons:

1. **Determining existing versions for dynamic dependencies:** When a dynamic version (like `1.+` or `latest.release`) is specified, Gradle must identify the concrete versions available.
2. **Resolving module dependencies for a specific version:** Gradle retrieves the dependencies associated with a module based on the specified version, ensuring the correct transitive dependencies are included in the build.

A. Determining existing versions for dynamic dependencies

When faced with a dynamic version, Gradle must identify the available concrete versions through the following steps:

1. **Inspecting repositories:** Gradle checks each defined repository in the order they were added. It doesn't stop at the first one that returns metadata but continues through all available repositories.
2. **Maven repositories:** Gradle retrieves version information from the `maven-metadata.xml` file, which lists available versions.
3. **Ivy repositories:** Gradle resorts to a directory listing to gather available versions.

The result is a list of candidate versions that Gradle evaluates and matches to the dynamic version. Gradle [caches](#) this information to optimize future resolution. At this point, [version conflict resolution](#) is resumed.

B. Resolving module dependencies for a specific version

When Gradle tries to resolve a required dependency with a specific version, it follows this process:

1. **Repository inspection:** Gradle checks each repository in the order they are defined.
 - It looks for metadata files describing the module (`.module`, `.pom`, or `ivy.xml`), or directly for artifact files.
 - Modules with metadata files (`.module`, `.pom`, or `ivy.xml`) are prioritized over those with just an artifact file.
 - Once metadata is found in a repository, subsequent repositories are ignored.
2. **Retrieving and parsing metadata:** If metadata is found, it is parsed.
 - If the POM file has a parent POM, Gradle recursively resolves each parent module.

3. **Requesting artifacts:** All artifacts for the module are fetched from the same repository that provided the metadata.
4. **Caching:** All data, including the repository source and any potential misses, are stored in the [dependency cache](#) for future use.

NOTE

The point above highlights a potential issue with integrating [Maven Local](#). Since Maven Local acts as a Maven cache, it may occasionally miss artifacts for a module. When Gradle sources a module from Maven Local and artifacts are missing, it assumes those artifacts are entirely unavailable.

Repository disabling

When Gradle fails to retrieve information from a repository, it disables the repository for the remainder of the build and fails all dependency resolution.

This behavior ensures reproducibility.

If the build were to continue while ignoring the faulty repository, subsequent builds could produce different results once the repository is back online.

HTTP Retries

Gradle will attempt to connect to a repository multiple times before disabling it. If the connection fails, Gradle retries on specific errors that might be temporary, with increasing wait times between retries.

A repository is marked as unavailable when it cannot be reached, either due to a permanent error or after the maximum number of retries has been exhausted.

The dependency tree

Once the process is complete, a dependency tree is created.

The dependency tree is a hierarchical representation of all the dependencies required by a project, including direct dependencies (declared explicitly) and transitive dependencies (pulled in automatically by those direct dependencies). The graph shows how dependencies relate to each other and how Gradle resolves them.

For the dependency `org.jetbrains.kotlin:kotlinx-serialization-json:1.5.1`, the graph includes the primary dependency and all its transitive dependencies. Here's what the graph looks like:

```
org.jetbrains.kotlin:kotlinx-serialization-json:1.5.1
├── JVM variant
│   ├── org.jetbrains.kotlin:kotlinx-serialization-core:1.5.1
│   │   ├── org.jetbrains.kotlin:kotlin-stdlib:1.8.0
│   │   ├── org.jetbrains.kotlin:kotlin-stdlib-common:1.8.0
│   │   └── org.jetbrains:annotations:13.0
│   ├── org.jetbrains.kotlin:kotlin-stdlib:1.8.0
│   └── org.jetbrains.kotlin:kotlin-stdlib-common:1.8.0
```

```

└── org.jetbrains:annotations:13.0
└── Android variant
    ├── org.jetbrains.kotlinx:kotlinx-serialization-core:1.5.1
    │   ├── org.jetbrains.kotlin:kotlin-stdlib:1.8.0
    │   │   ├── org.jetbrains.kotlin:kotlin-stdlib-common:1.8.0
    │   │   └── org.jetbrains:annotations:13.0
    │   ├── org.jetbrains.kotlin:kotlin-stdlib:1.8.0
    │   ├── org.jetbrains.kotlin:kotlin-stdlib-common:1.8.0
    │   ├── org.jetbrains:annotations:13.0
    │   └── com.android.tools:common-library:1.0.0
    ├── Native variant
    │   ├── org.jetbrains.kotlinx:kotlinx-serialization-core:1.5.1
    │   │   ├── org.jetbrains.kotlin:kotlin-stdlib:1.8.0
    │   │   │   ├── org.jetbrains.kotlin:kotlin-stdlib-common:1.8.0
    │   │   │   └── org.jetbrains:annotations:13.0
    │   │   ├── org.jetbrains.kotlin:kotlin-stdlib:1.8.0
    │   │   ├── org.jetbrains.kotlin:kotlin-stdlib-common:1.8.0
    │   │   ├── org.jetbrains:annotations:13.0
    │   │   └── kotlinx.coroutines:kotlinx-coroutines-core-native:1.6.4
    └── JavaScript variant
        ├── org.jetbrains.kotlinx:kotlinx-serialization-core:1.5.1
        │   ├── org.jetbrains.kotlin:kotlin-stdlib-js:1.8.0
        │   │   └── org.jetbrains.kotlin:kotlin-stdlib-common:1.8.0
        ├── org.jetbrains.kotlin:kotlin-stdlib-js:1.8.0
        ├── org.jetbrains.kotlin:kotlin-stdlib-common:1.8.0
        └── kotlinx.coroutines:kotlinx-coroutines-core-js:1.6.4

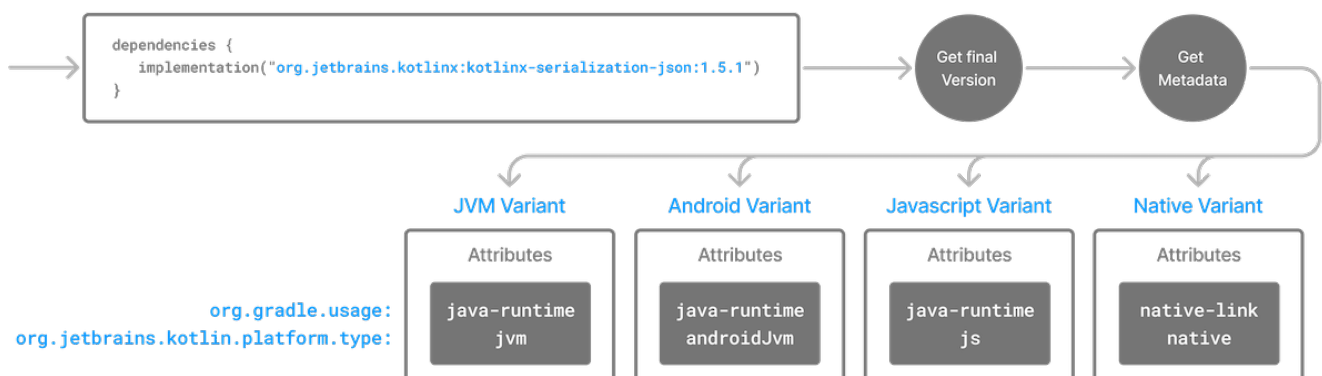
```

Next Chapter: [Learn how Gradle selects Variants >>](#)

Understanding Variant Selection

Gradle's dependency management engine is *variant aware*.

In the [previous section](#), Gradle built a graph of resolved dependencies. During graph resolution, Gradle must select the proper variants of each dependency based on the requirements of your build.



Variants represent different ways a *component* can be used, such as for Java compilation, native linking, or documentation. Each variant may have its own artifacts and dependencies.

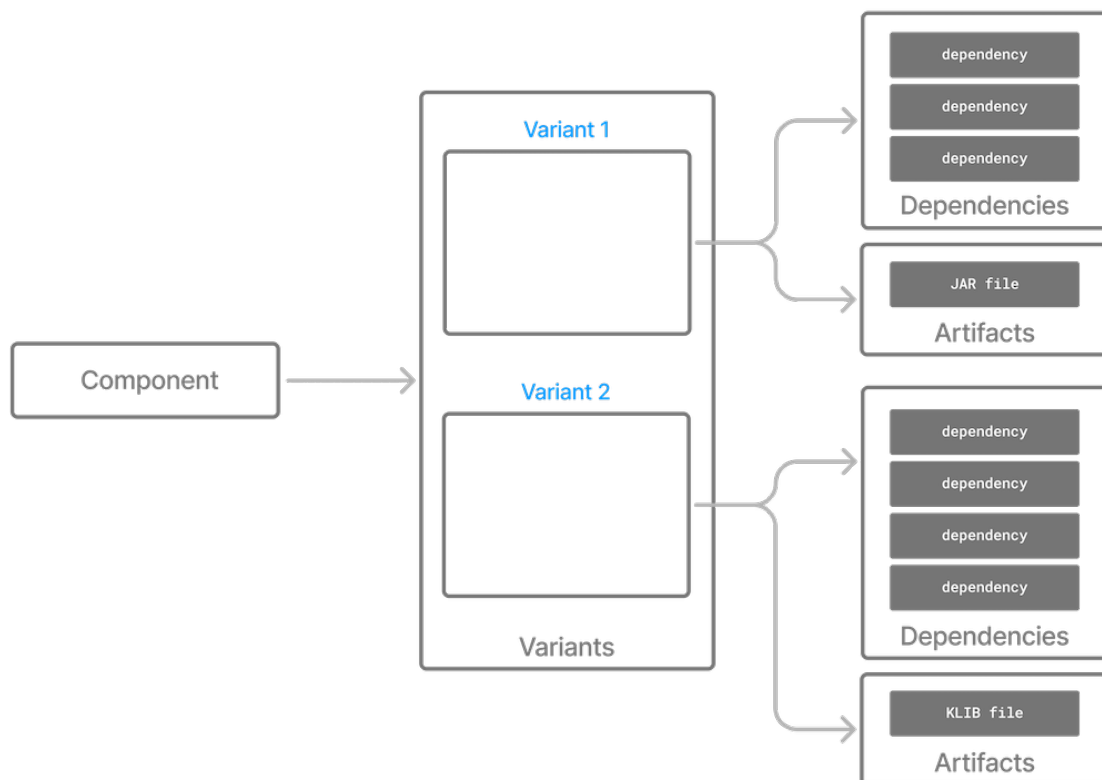
Gradle uses *attributes* to determine which variant to choose. These attributes add context to each variant, describing when they should be used.

Components

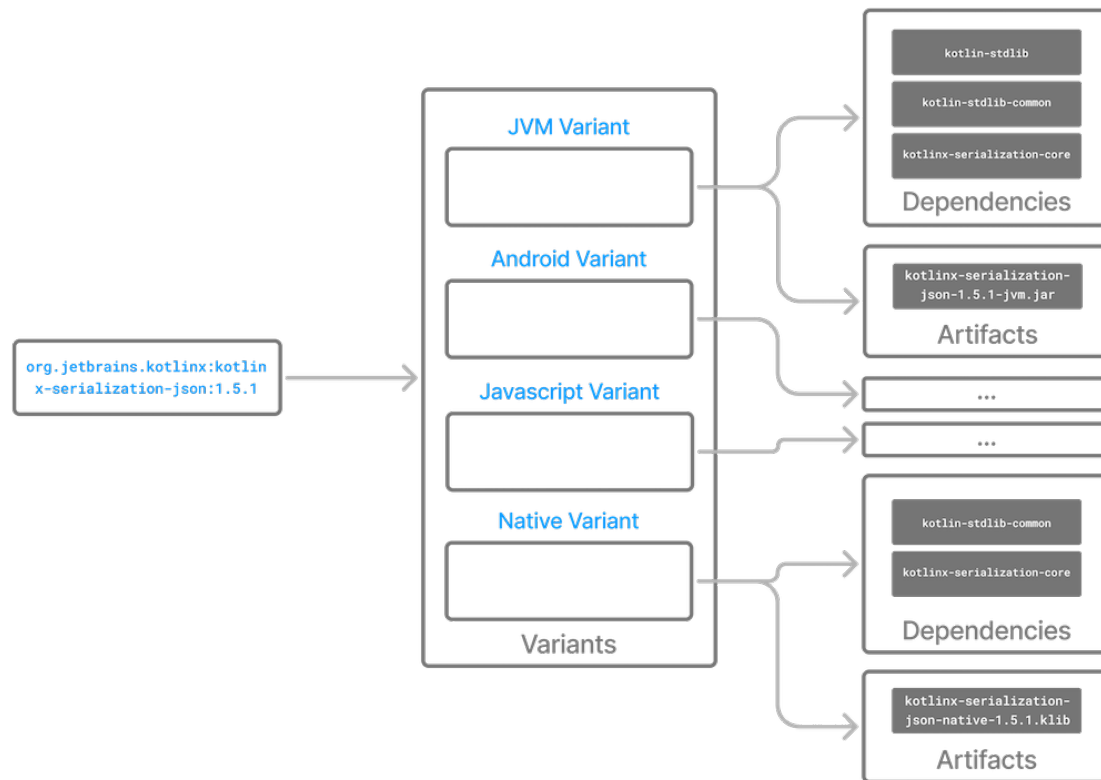
A *component*, in most cases, represents a **version** of a **module**.

A component:

- contains variants
 - which contain one or more artifacts
 - which contain zero or more dependencies
 - which is described by metadata



In the example above, `org.jetbrains.kotlinx:kotlinx-serialization-json:1.5.1` is the component. It has a module `org.jetbrains.kotlinx:kotlinx-serialization-json` and a version `1.5.1`:



Variant attributes

Variants represent different versions or aspects of a *component*, like `api` vs `implementation` or `jar` vs `classes`.

In the example above, `org.jetbrains.kotlinx.serialization-json:1.5.1` has four *variants*: `jvm`, `android`, `js`, and `native`.

Attributes are type-safe key-value pairs used by both the *consumer* and the *producer* during *variant* selection: `attribute : value`.

In the example above, there are two *attributes* of importance to note for the *variants* of `org.jetbrains.kotlinx.serialization-json:1.5.1`:

However, it is also the case that in the metadata of `org.jetbrains.kotlinx.serialization-json:1.5.1`, there are many more attributes that describe each *variant* such as `org.gradle.libraryelements: jar` or `org.gradle.category: library`:

```
"variants": [
  ...
  {
    "name": "jsLegacyRuntimeElements-published",
    "attributes": {
      "org.gradle.category": "library",
      "org.gradle.usage": "kotlin-runtime",
      "org.jetbrains.kotlin.js.compiler": "legacy",
      "org.jetbrains.kotlin.platform.type": "js"
    }
  },
]
```

```

{
  "name": "jvmRuntimeElements-published",
  "attributes": {
    "org.gradle.category": "library",
    "org.gradle.libraryelements": "jar",
    "org.gradle.usage": "java-runtime",
    "org.jetbrains.kotlin.platform.type": "jvm"
  }
},
...
]

```

You can learn more about *variants* and *attributes* in [Variants and Attributes](#).

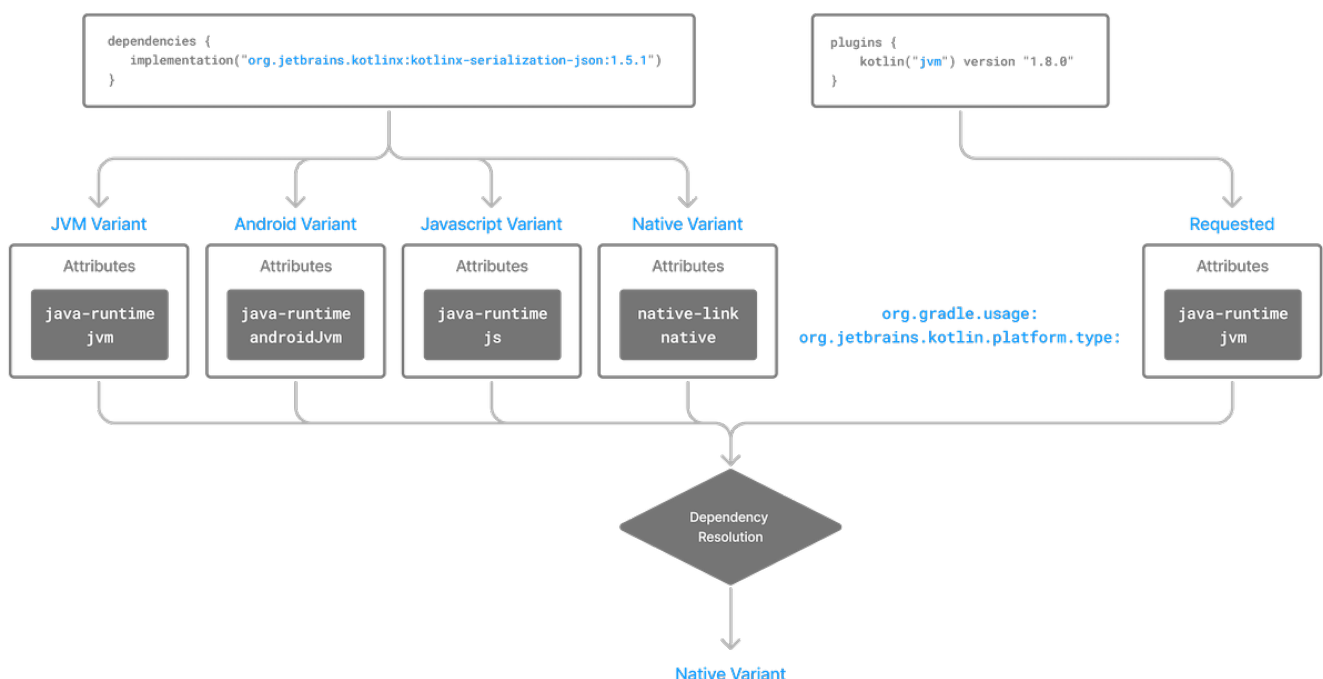
There are two types of *attributes* Gradle uses to match the available *variants* to the one required for the build:

- **Consumer attributes:** Define the desired characteristics of a variant requested by a resolvable configuration.
- **Producer attributes:** Each variant has a set of attributes that describe its purpose.

Variant attribute matching

There are no restrictions on how many *variants* a *component* can define. A typical *component* will include at least an implementation *variant* but may also provide additional *variants*, such as test fixtures, documentation, or source code. Furthermore, a *component* can offer *different variants* for the same usage, depending on the consumer. For instance, during compilation, a *component* may provide different headers for Linux, Windows, and macOS.

Gradle performs *variant-aware selection* by **matching the attributes specified by the consumer with those defined by the producer**. The details of this process are covered in the section below.



IMPORTANT

The variant name is primarily used for debugging and error messages. It does not play a role in variant matching; only the variant's attributes are used in the matching process.

Variant attribute matching algorithm

Gradle's dependency resolution engine performs the following *variant* matching algorithm to find the best result (or fail):

1. Each candidate's attribute value is compared to the consumer's requested attribute value. A candidate is considered compatible if its value matches the consumer's value exactly, passes the attribute's compatibility rule, or is not provided.
2. If only one candidate is considered compatible, that candidate wins.
3. If several candidates are compatible, but one of the candidates matches all of the same attributes as the other candidates, Gradle chooses that candidate. This is the candidate with the "longest" match.
4. If several candidates are compatible and are compatible with an equal number of attributes, Gradle needs to disambiguate the candidates.
 1. For each requested attribute, if a candidate does not have a value matching the disambiguation rule, it's eliminated from consideration.
 2. If the attribute has a known precedence, Gradle will stop as soon as there is a single candidate remaining.
 3. If the attribute does not have a known precedence, Gradle must consider all attributes.
5. If several candidates still remain, Gradle will start to consider "extra" attributes to disambiguate between multiple candidates. Extra attributes are attributes that were not requested by the consumer but are present on at least one candidate. These extra attributes are considered in precedence order.
 1. If the attribute has a known precedence, Gradle will stop as soon as there is a single candidate remaining.
 2. After all extra attributes with precedence are considered, the remaining candidates can be chosen if they are compatible with all of the non-ordered disambiguation rules.
6. If several candidates still remain, Gradle will consider extra attributes again. A candidate can be chosen if it has the fewest number of extra attributes.

If at any step no compatible candidates remain, resolution fails. Additionally, Gradle outputs a list of all compatible candidates from step 1 to help with debugging variant matching failures.

Plugins and ecosystems can influence the selection algorithm by implementing compatibility rules, disambiguation rules, and defining the precedence of attributes. Attributes with a higher precedence are used to eliminate candidates in order.

For example, in the Java ecosystem, the `org.gradle.usage` attribute has a higher precedence than `org.gradle.libraryelements`. This means that if two candidates were available with compatible values for both `org.gradle.usage` and `org.gradle.libraryelements`, Gradle will choose the candidate

that passes the disambiguation rule for `org.gradle.usage`.

NOTE

There are two exceptions to the variant-aware resolution process:

- When a producer has no variants, a default artifact is selected.
- When a consumer *explicitly selects a configuration by name*, the artifacts associated with that configuration are used.

A simple example

Let's walk through an example where a consumer is trying to use a library for compilation.

First, the consumer details how it's going to use the result of dependency resolution. This is achieved by setting *attributes* on the consumer's resolvable configuration.

In this case, the consumer wants to resolve a variant that matches `org.gradle.usage=java-api`.

Next, the producer exposes different variants of its *component*:

- **API variant** (named `apiElements`) with the attribute `org.gradle.usage=java-api`
- **Runtime variant** (named `runtimeElements`) with the attribute `org.gradle.usage=java-runtime`

Finally, Gradle evaluates the variants and selects the correct one:

- The consumer requests a variant with attributes `org.gradle.usage=java-api`
- The producer's `apiElements` variant matches this request.
- The producer's `runtimeElements` variant does not match.

As a result, Gradle selects the `apiElements` variant and provides its artifacts and dependencies to the consumer.

A complicated example

In real-world scenarios, both consumers and producers often work with multiple attributes.

For instance, a Java Library project in Gradle will involve several attributes:

- `org.gradle.usage` describes how the variant is used.
- `org.gradle.dependency.bundling` describes how the variant handles dependencies (e.g., shadow jar, fat jar, regular jar).
- `org.gradle.libraryelements` describes the packaging of the variant (e.g., classes or jar).
- `org.gradle.jvm.version` describes the *minimal version* of Java the variant targets.
- `org.gradle.jvm.environment` describes the type of JVM the variant targets.

Let's consider a scenario where the consumer wants to run tests using a library on Java 8, and the producer supports two versions: Java 8 and Java 11.

Step 1: Consumer specifies the requirements.

The consumer wants to resolve a variant that:

- Can be used at runtime (`org.gradle.usage=java-runtime`).
- Can run on *at least* Java 8 (`org.gradle.jvm.version=8`).

Step 2: Producer exposes multiple variants.

The producer offers variants for both Java 8 and Java 11 for both API and runtime usage:

- **API variant for Java 8** (named `apiJava8Elements`) with attributes `org.gradle.usage=java-api` and `org.gradle.jvm.version=8`.
- **Runtime variant for Java 8** (named `runtime8Elements`) with attributes `org.gradle.usage=java-runtime` and `org.gradle.jvm.version=8`.
- **API variant for Java 11** (named `apiJava11Elements`) with attributes `org.gradle.usage=java-api` and `org.gradle.jvm.version=11`.
- **Runtime variant for Java 11** (named `runtime11Elements`) with attributes `org.gradle.usage=java-runtime` and `org.gradle.jvm.version=11`.

Step 3: Gradle matches the attributes.

Gradle compares the consumer's requested attributes with the producer's variants:

- The consumer requests a variant with `org.gradle.usage=java-runtime` and `org.gradle.jvm.version=8`.
- Both `runtime8Elements` and `runtime11Elements` match the `org.gradle.usage=java-runtime` attribute.
- The API variants (`apiJava8Elements` and `apiJava11Elements`) are discarded as they don't match `org.gradle.usage=java-runtime`.
- The variant `runtime8Elements` is selected because it is compatible with Java 8.
- The variant `runtime11Elements` is incompatible because it requires Java 11.

Gradle selects `runtime8Elements` and provides its artifacts and dependencies to the consumer.

What happens if the consumer sets `org.gradle.jvm.version=7`?

In this case, dependency resolution would *fail*, with an error explaining there is no suitable variant. Gradle knows the consumer requires a Java 7-compatible library, but the producer's minimum version is 8.

If the consumer requested `org.gradle.jvm.version=15`, Gradle could choose either the Java 8 or Java 11 variant. Gradle would then select the highest compatible version—Java 11.

Variant selection errors

When Gradle attempts to select the most compatible variant of a *component*, resolution may fail due to:

- **Ambiguity error:** When more than one variant from the producer matches the consumer's

attributes, leading to confusion over which to select.

- **Incompatibility error:** When none of the producer's variants match the consumer's attributes, causing the resolution to fail.

Dealing with ambiguity errors

An ambiguous variant selection looks like this:

```
> Could not resolve all files for configuration ':compileClasspath'.
  > Could not resolve project :lib.
    Required by:
      project :ui
    > Cannot choose between the following variants of project :lib:
      - feature1ApiElements
      - feature2ApiElements
    All of them match the consumer attributes:
      - Variant 'feature1ApiElements' capability org.test:test-capability:1.0:
        - Unmatched attribute:
          - Found org.gradle.category 'library' but wasn't required.
        - Compatible attributes:
          - Provides org.gradle.dependency.bundling 'external'
          - Provides org.gradle.jvm.version '11'
          - Required org.gradle.libraryelements 'classes' and found value
'jar'.
          - Provides org.gradle.usage 'java-api'
      - Variant 'feature2ApiElements' capability org.test:test-capability:1.0:
        - Unmatched attribute:
          - Found org.gradle.category 'library' but wasn't required.
        - Compatible attributes:
          - Provides org.gradle.dependency.bundling 'external'
          - Provides org.gradle.jvm.version '11'
          - Required org.gradle.libraryelements 'classes' and found value
'jar'.
          - Provides org.gradle.usage 'java-api'
```

In this scenario, all *compatible* candidate variants are listed along with their attributes:

- **Unmatched attributes:** Shown first, these indicate what attributes may be missing or misaligned for selecting the proper variant.
- **Compatible attributes:** Shown next, these highlight how the candidate variants align with the consumer's requirements.
- **Incompatible attributes:** Will not be shown, as incompatible variants are excluded.

In the example above, the issue isn't with attribute matching but with **capability matching**. Both `feature1ApiElements` and `feature2ApiElements` offer the same attributes and capabilities, making them indistinguishable to Gradle.

To resolve this, you can modify the producer (`project :lib`) to provide different capabilities or

express a capability choice on the consumer side (`project :ui`) to disambiguate between the variants.

Dealing with no matching variant errors

A no matching variant error might look like this:

```
> No variants of project :lib match the consumer attributes:
- Configuration ':lib:compile':
  - Incompatible attribute:
    - Required artifactType 'dll' and found incompatible value 'jar'.
  - Other compatible attribute:
    - Provides usage 'api'
- Configuration ':lib:compile' variant debug:
  - Incompatible attribute:
    - Required artifactType 'dll' and found incompatible value 'jar'.
  - Other compatible attributes:
    - Found buildType 'debug' but wasn't required.
    - Provides usage 'api'
- Configuration ':lib:compile' variant release:
  - Incompatible attribute:
    - Required artifactType 'dll' and found incompatible value 'jar'.
  - Other compatible attributes:
    - Found buildType 'release' but wasn't required.
    - Provides usage 'api'
```

Or:

```
> No variants of project : match the consumer attributes:
- Configuration ':myElements' declares attribute 'color' with value 'blue':
  - Incompatible because this component declares attribute 'artifactType' with
value 'jar' and the consumer needed attribute 'artifactType' with value 'dll'
- Configuration ':myElements' variant secondary declares attribute 'color' with
value 'blue':
  - Incompatible because this component declares attribute 'artifactType' with
value 'jar' and the consumer needed attribute 'artifactType' with value 'dll'
```

In these cases, *potentially compatible* candidate variants are displayed, showing:

- **Incompatible attributes:** Listed first to help identify why a variant could not be selected.
- **Other attributes:** Including requested and compatible attributes, and any extra producer attributes that the consumer did not request.

The goal here is to understand which variant could be selected, if any. In some cases, there may simply be no compatible variants from the producer (for example, if the consumer requires a `dll` but the producer only offers a `jar` or if a library is built for Java 11, but the consumer requires Java 8).

Dealing with incompatible variant errors

An incompatible variant error looks like the following example, where a consumer wants to select a variant with `color=green`, but the only variant available has `color=blue`:

```
> Could not resolve all dependencies for configuration ':resolveMe'.
> Could not resolve project :.
   Required by:
       project :
       > Configuration 'mismatch' in project : does not match the consumer attributes
           Configuration 'mismatch':
               - Incompatible because this component declares attribute 'color' with value 'blue' and the consumer needed attribute 'color' with value 'green'
```

It occurs when Gradle cannot select a single variant of a dependency because an explicitly requested attribute value does not match (and is not compatible with) the value of that attribute on any of the variants of the dependency.

A sub-type of this failure occurs when Gradle **successfully** selects multiple variants of the same component, but the selected variants are incompatible with **each other**.

This looks like the following, where a consumer wants to select two different variants of a component, each supplying different capabilities, which is acceptable. Unfortunately one variant has `color=blue` and the other has `color=green`:

```
> Could not resolve all dependencies for configuration ':resolveMe'.
> Could not resolve project :.
   Required by:
       project :
       > Multiple incompatible variants of org.example:nyvu:1.0 were selected:
           - Variant org.example:nyvu:1.0 variant blueElementsCapability1 has
attributes {color=blue}
           - Variant org.example:nyvu:1.0 variant greenElementsCapability2 has
attributes {color=green}

> Could not resolve project :.
   Required by:
       project :
       > Multiple incompatible variants of org.example:pi2e5:1.0 were selected:
           - Variant org.example:pi2e5:1.0 variant blueElementsCapability1 has
attributes {color=blue}
           - Variant org.example:pi2e5:1.0 variant greenElementsCapability2 has
attributes {color=green}
```

Dealing with ambiguous transformation errors

ArtifactTransforms can be used to transform artifacts from one type to another, changing their attributes. Variant selection can use the attributes available as the result of an artifact transform as

a candidate variant.

If a project registers multiple artifact transforms, needs to use an artifact transform to produce a matching variant for a consumer's request, and multiple artifact transforms could each be used to accomplish this, then Gradle will fail with an ambiguous transformation error like the following:

```
> Could not resolve all dependencies for configuration ':resolveMe'.
  > Found multiple transforms that can produce a variant of project : with requested
  attributes:
    - color 'red'
    - shape 'round'
  Found the following transforms:
    - From 'configuration ':roundBlueLiquidElements'':
      - With source attributes:
        - color 'blue'
        - shape 'round'
        - state 'liquid'
      - Candidate transform(s):
        - Transform 'BrokenTransform' producing attributes:
          - color 'red'
          - shape 'round'
          - state 'gas'
        - Transform 'BrokenTransform' producing attributes:
          - color 'red'
          - shape 'round'
          - state 'solid'
```

Visualizing variant information

Gradle offers built-in tasks to visualize the variant selection process and display the producer and consumer attributes involved.

Outgoing variants report

The report task `outgoingVariants` shows the list of variants available for selection by consumers of the project. It displays the capabilities, attributes and artifacts for each variant.

This task is similar to the [dependencyInsight reporting task](#).

By default, `outgoingVariants` prints information about all variants. It offers the optional parameter `--variant <variantName>` to select a single variant to display. It also accepts the `--all` flag to include information about legacy and deprecated configurations, or `--no-all` to exclude this information.

Here is the output of the `outgoingVariants` task on a freshly generated `java-library` project:

```
> Task :outgoingVariants
-----
Variant apiElements
-----
```

API elements for the 'main' feature.

Capabilities

- new-java-library:lib:unspecified (default capability)

Attributes

- org.gradle.category = library
- org.gradle.dependency.bundling = external
- org.gradle.jvm.version = 11
- org.gradle.libraryelements = jar
- org.gradle.usage = java-api

Artifacts

- build/libs/lib.jar (artifactType = jar)

Secondary Variants (*)

Secondary Variant classes

Description = Directories containing compiled class files for main.

Attributes

- org.gradle.category = library
- org.gradle.dependency.bundling = external
- org.gradle.jvm.version = 11
- org.gradle.libraryelements = classes
- org.gradle.usage = java-api

Artifacts

- build/classes/java/main (artifactType = java-classes-directory)

Variant mainSourceElements (i)

Description = List of source directories contained in the Main SourceSet.

Capabilities

- new-java-library:lib:unspecified (default capability)

Attributes

- org.gradle.category = verification
- org.gradle.dependency.bundling = external
- org.gradle.verificationtype = main-sources

Artifacts

- src/main/java (artifactType = directory)
- src/main/resources (artifactType = directory)

Variant runtimeElements

Runtime elements for the 'main' feature.

Capabilities

- new-java-library:lib:unspecified (default capability)

Attributes

- org.gradle.category = library
- org.gradle.dependency.bundling = external
- org.gradle.jvm.version = 11
- org.gradle.libraryelements = jar
- org.gradle.usage = java-runtime

Artifacts

- build/libs/lib.jar (artifactType = jar)

Secondary Variants (*)

----- Secondary Variant classes -----

Description = Directories containing compiled class files for main.

Attributes

- org.gradle.category = library
- org.gradle.dependency.bundling = external
- org.gradle.jvm.version = 11
- org.gradle.libraryelements = classes
- org.gradle.usage = java-runtime

Artifacts

- build/classes/java/main (artifactType = java-classes-directory)

----- Secondary Variant resources -----

Description = Directories containing the project's assembled resource files for use at runtime.

Attributes

- org.gradle.category = library
- org.gradle.dependency.bundling = external
- org.gradle.jvm.version = 11
- org.gradle.libraryelements = resources
- org.gradle.usage = java-runtime

Artifacts

- build/resources/main (artifactType = java-resources-directory)

----- Variant testResultsElementsForTest (i) -----

Description = Directory containing binary results of running tests for the test Test Suite's test target.

Capabilities

- new-java-library:lib:unspecified (default capability)

Attributes

- org.gradle.category = verification
- org.gradle.testsuite.name = test

```
- org.gradle.testsuite.target.name = test
- org.gradle.testsuite.type       = unit-test
- org.gradle.verificationtype     = test-results
Artifacts
- build/test-results/test/binary (artifactType = directory)

(i) Configuration uses incubating attributes such as Category.VERIFICATION.
(*) Secondary variants are variants created via the Configuration#getOutgoing():
ConfigurationPublications API which also participate in selection, in addition to the
configuration itself.
```

From this you can see the two main variants that are exposed by a java library, `apiElements` and `runtimeElements`. Notice that the main difference is on the `org.gradle.usage` attribute, with values `java-api` and `java-runtime`. As they indicate, this is where the difference is made between what needs to be on the *compile* classpath of consumers, versus what's needed on the *runtime* classpath.

It also shows *secondary* variants, which are exclusive to Gradle projects and not published. For example, the secondary variant `classes` from `apiElements` is what allows Gradle to skip the JAR creation when compiling against a `java-library` project.

Information about invalid consumable configurations

A project cannot have multiple configurations with the same attributes and capabilities. In that case, the project will fail to build.

In order to be able to visualize such issues, the outgoing variant reports handle those errors in a lenient fashion. This allows the report to display information about the issue.

Resolvable configurations report

Gradle also offers a complimentary report task called `resolvableConfigurations` that displays the *resolvable* configurations of a project, which are those which can have dependencies added and be resolved. The report will list their attributes and any configurations that they extend. It will also list a summary of any attributes which will be affected by [Compatibility Rules](#) or [Disambiguation Rules](#) during resolution.

By default, `resolvableConfigurations` prints information about all purely resolvable configurations. These are configurations that are marked resolvable but **not** marked consumable. Though some resolvable configurations are also marked consumable, these are legacy configurations that should **not** have dependencies added in build scripts. This report offers the optional parameter `--configuration <configurationName>` to select a single configuration to display. It also accepts the `--all` flag to include information about legacy and deprecated configurations, or `--no-all` to exclude this information. Finally, it accepts the `--recursive` flag to list in the extended configurations section those configurations which are extended *transitively* rather than directly. Alternatively, `--no-recursive` can be used to exclude this information.

Here is the output of the `resolvableConfigurations` task on a freshly generated `java-library` project:

```
> Task :resolvableConfigurations
```

Configuration annotationProcessor

Description = Annotation processors and their dependencies for source set 'main'.

Attributes

- org.gradle.category = library
- org.gradle.dependency.bundling = external
- org.gradle.jvm.environment = standard-jvm
- org.gradle.libraryelements = jar
- org.gradle.usage = java-runtime

Configuration compileClasspath

Description = Compile classpath for source set 'main'.

Attributes

- org.gradle.category = library
- org.gradle.dependency.bundling = external
- org.gradle.jvm.environment = standard-jvm
- org.gradle.jvm.version = 11
- org.gradle.libraryelements = classes
- org.gradle.usage = java-api

Extended Configurations

- compileOnly
- implementation

Configuration runtimeClasspath

Description = Runtime classpath of source set 'main'.

Attributes

- org.gradle.category = library
- org.gradle.dependency.bundling = external
- org.gradle.jvm.environment = standard-jvm
- org.gradle.jvm.version = 11
- org.gradle.libraryelements = jar
- org.gradle.usage = java-runtime

Extended Configurations

- implementation
- runtimeOnly

Configuration testAnnotationProcessor

Description = Annotation processors and their dependencies for source set 'test'.

Attributes

- org.gradle.category = library

- org.gradle.dependency.bundling = external
- org.gradle.jvm.environment = standard-jvm
- org.gradle.libraryelements = jar
- org.gradle.usage = java-runtime

Configuration testCompileClasspath

Description = Compile classpath for source set 'test'.

Attributes

- org.gradle.category = library
- org.gradle.dependency.bundling = external
- org.gradle.jvm.environment = standard-jvm
- org.gradle.jvm.version = 11
- org.gradle.libraryelements = classes
- org.gradle.usage = java-api

Extended Configurations

- testCompileOnly
- testImplementation

Configuration testRuntimeClasspath

Description = Runtime classpath of source set 'test'.

Attributes

- org.gradle.category = library
- org.gradle.dependency.bundling = external
- org.gradle.jvm.environment = standard-jvm
- org.gradle.jvm.version = 11
- org.gradle.libraryelements = jar
- org.gradle.usage = java-runtime

Extended Configurations

- testImplementation
- testRuntimeOnly

Compatibility Rules

Description = The following Attributes have compatibility rules defined.

- org.gradle.dependency.bundling
- org.gradle.jvm.environment
- org.gradle.jvm.version
- org.gradle.libraryelements
- org.gradle.plugin.api-version
- org.gradle.usage

Disambiguation Rules

Description = The following Attributes have disambiguation rules defined.

- org.gradle.category
- org.gradle.dependency.bundling
- org.gradle.jvm.environment
- org.gradle.jvm.version
- org.gradle.libraryelements
- org.gradle.plugin.api-version
- org.gradle.usage

From this you can see the two main configurations used to resolve dependencies, `compileClasspath` and `runtimeClasspath`, as well as their corresponding test configurations.

Capabilities

In a dependency graph, it's common for multiple implementations of the same API to be accidentally included, especially with libraries like logging frameworks where different bindings are selected by various transitive dependencies.

Since these implementations typically reside at different group, artifact, and version (GAV) coordinates, build tools often can't detect the conflict.

To address this, Gradle introduces the concept of **capability**.

Understanding capabilities

A capability is essentially a way to declare that different components (dependencies) offer the same functionality.

It's **illegal** for Gradle to include more than one component providing the same capability in a single dependency graph. If Gradle detects two components providing the same capability (e.g., different bindings for a logging framework), it will fail the build with an error, indicating the conflicting modules. This ensures that conflicting implementations are resolved, avoiding issues on the classpath.

For instance, suppose you have dependencies on two different libraries for database connection pooling:

```
dependencies {
    implementation("com.zaxxer:HikariCP:4.0.3") // A popular connection pool
    implementation("org.apache.commons:commons-dbcp2:2.8.0") // Another connection
    pool
}

configurations.all {
    resolutionStrategy.capabilitiesResolution.withCapability("database:connection-
pool") {
        select("com.zaxxer:HikariCP")
    }
}
```

```
}  
}
```

In this case, both `HikariCP` and `commons-dbcp2` provide the same functionality (connection pooling). Gradle will fail if both are on the classpath.

Since only one should be used, Gradle's resolution strategy allows you to **select** `HikariCP`, resolving the conflict.

Understanding capability coordinates

A **capability** is identified by a `(group, module, version)` triplet.

Every component defines an implicit capability based on its GAV coordinates: group, artifact, and version.

For instance, the `org.apache.commons:commons-lang3:3.8` module has an implicit capability with the group `org.apache.commons`, name `commons-lang3`, and version `3.8`:

```
dependencies {  
    implementation("org.apache.commons:commons-lang3:3.8")  
}
```

It's important to note that capabilities are **versioned**.

Declaring component capabilities

To detect conflicts early, it's useful to declare **component capabilities** through rules, allowing conflicts to be caught during the build instead of at runtime.

One common scenario is when a component is relocated to different coordinates in a newer release.

For example, the ASM library was published under `asm:asm` until version `3.3.1`, and then relocated to `org.ow2.asm:asm` starting with version `4.0`. Including both versions on the classpath is illegal because they provide the same feature, under different coordinates.

Since each component has an implicit capability based on its GAV coordinates, we can address this conflict by using a rule that declares the `asm:asm` module as providing the `org.ow2.asm:asm` capability:

build.gradle.kts

```
class AsmCapability : ComponentMetadataRule {  
    override  
    fun execute(context: ComponentMetadataContext) = context.details.run {  
        if (id.group == "asm" && id.name == "asm") {
```

```

        allVariants {
            withCapabilities {
                // Declare that ASM provides the org.ow2.asm:asm
                // capability, but with an older version
                addCapability("org.ow2.asm", "asm", id.version)
            }
        }
    }
}

```

build.gradle

```

@CompileStatic
class AsmCapability implements ComponentMetadataRule {
    void execute(ComponentMetadataContext context) {
        context.details.with {
            if (id.group == "asm" && id.name == "asm") {
                allVariants {
                    it.withCapabilities {
                        // Declare that ASM provides the org.ow2.asm:asm
                        // capability, but with an older version
                        it.addCapability("org.ow2.asm", "asm", id.version)
                    }
                }
            }
        }
    }
}

```

With this rule in place, the build will fail if both `asm:asm` (`<= 3.3.1`) and `org.ow2.asm:asm` (`4.0+`) are present in the dependency graph.

NOTE

Gradle won't resolve the conflict automatically, but this helps you realize that the problem exists. It's recommended to package such rules into plugins for use in builds, allowing users to decide which version to use or to fix the classpath conflict.

Selecting between candidates

At some point, a dependency graph is going to include either *incompatible modules*, or modules which are *mutually exclusive*.

For example, you may have different logger implementations, and you need to choose one binding. [Capabilities](#) help *understand* the conflict, then Gradle provides you with tools to solve the conflicts.

Selecting between different capability candidates

In the relocation example above, Gradle was able to tell you that you have two versions of the same API on classpath: an "old" module and a "relocated" one. We can solve the conflict by automatically choosing the component which has the highest capability version:

build.gradle.kts

```
configurations.all {  
    resolutionStrategy.capabilitiesResolution.withCapability("org.ow2.asm:asm") {  
        selectHighestVersion()  
    }  
}
```

build.gradle

```
configurations.all {  
    resolutionStrategy.capabilitiesResolution.withCapability('org.ow2.asm:asm') {  
        selectHighestVersion()  
    }  
}
```

However, choosing the highest capability version conflict resolution is not always suitable.

For a logging framework, for example, it doesn't matter what version of the logging frameworks we use. In this case, we explicitly select **slf4j** as the preferred option:

build.gradle.kts

```
configurations.all {  
    resolutionStrategy.capabilitiesResolution.withCapability("log4j:log4j") {  
        val toBeSelected = candidates.firstOrNull { it.id.let { id -> id is  
ModuleComponentIdentifier && id.module == "log4j-over-slf4j" } }  
        if (toBeSelected != null) {  
            select(toBeSelected)  
        }  
        because("use slf4j in place of log4j")  
    }  
}
```

build.gradle

```
configurations.all {
    resolutionStrategy.capabilitiesResolution.withCapability("log4j:log4j") {
        def toBeSelected = candidates.find { it.id instanceof
ModuleComponentIdentifier && it.id.module == 'log4j-over-slf4j' }
        if (toBeSelected != null) {
            select(toBeSelected)
        }
        because 'use slf4j in place of log4j'
    }
}
```

This approach works also well if you have multiple `slf4j` bindings on the classpath; bindings are basically different logger implementations, and you need only one. However, the selected implementation may depend on the configuration being resolved.

For instance, in testing environments, the lightweight `slf4j-simple` logging implementation might be sufficient, while in production, a more robust solution like `logback` may be preferable.

Resolution can only be made in favor of a module that is found in the dependency graph. The `select` method accepts only a module from the current set of candidates. If the desired module is not part of the conflict, you can choose not to resolve that particular conflict, effectively leaving it unresolved. Another conflict in the graph may have the module you want to select.

If no resolution is provided for all conflicts on a given capability, the build will fail because the module chosen for resolution was not found in the graph. Additionally, calling `select(null)` will result in an error and should be avoided.

For more information, refer to the [capabilities resolution API](#).

Variants and Attributes

Variants represent different versions or aspects of a component, like `api` vs `implementation`. **Attributes** define which variant is selected based on the consumer's requirements.

For example, a library may have an `api` and an `implementation` variant. Here, the consumer wants an external `implementation` variant:

```
configurations {
    implementation {
        attributes {
            attribute(Bundling.BUNDLING_ATTRIBUTE, objects.named(Bundling.EXTERNAL))
        }
    }
}
```

For example, a build might have **debug** and **release** variants. This selects the **debug** variant based on the attribute.

```
configurations {
    compileClasspath {
        attributes {
            attribute(TargetConfiguration.TARGET_ATTRIBUTE, objects.named("debug"))
        }
    }
}
```

Attributes help Gradle match the right **variant** by comparing the requested attributes with what's available:

```
attribute(TargetConfiguration.TARGET_ATTRIBUTE, objects.named("debug"))
```

This sets the `TargetConfiguration.TARGET_ATTRIBUTE` to `"debug"`, meaning Gradle will attempt to resolve dependencies that have a "debug" variant, instead of other available variants (like "release").

Standard attributes defined by Gradle

As a user of Gradle, attributes are often hidden as implementation details. But it might be useful to understand the *standard attributes* defined by Gradle and its core plugins.

As a plugin author, these attributes, and the way they are defined, can serve as a basis for [building your own set of attributes](#) in your ecosystem plugin.

Ecosystem-independent standard attributes

Attribute name	Description	Values	compatibility and disambiguation rules
<code>org.gradle.usage</code>	Indicates main purpose of variant	<code>Usage</code> values built from constants defined in <code>Usage</code>	Following ecosystem semantics (e.g. <code>java-runtime</code> can be used in place of <code>java-api</code> but not the opposite)
<code>org.gradle.category</code>	Indicates the category of this software component	<code>Category</code> values built from constants defined in <code>Category</code>	Following ecosystem semantics (e.g. <code>library</code> is default on the JVM, no compatibility otherwise)
<code>org.gradle.libraryelements</code>	Indicates the contents of a <code>org.gradle.category=library</code> variant	<code>LibraryElements</code> values built from constants defined in <code>LibraryElements</code>	Following ecosystem semantics (e.g. in the JVM world, <code>jar</code> is the default and is compatible with <code>classes</code>)

Attribute name	Description	Values	compatibility and disambiguation rules
<code>org.gradle.docstyp</code>	Indicates the contents of a <code>org.gradle.category=documentation</code> variant	<code>Docstyp</code> values built from constants defined in <code>Docstyp</code>	No default, no compatibility
<code>org.gradle.dependency.bundling</code>	Indicates how dependencies of a variant are accessed.	<code>Bundling</code> values built from constants defined in <code>Bundling</code>	Following ecosystem semantics (e.g. in the JVM world, <code>embedded</code> is compatible with <code>external</code>)
<code>org.gradle.verificatio</code> <code>ntype</code>	Indicates what kind of verification task produced this output.	<code>VerificationType</code> values built from constants defined in <code>VerificationType</code>	No default, no compatibility

When the `Category` attribute is present with the incubating value `org.gradle.category=verification` on a variant, that variant is considered to be a verification-time only variant.

These variants are meant to contain only the results of running verification tasks, such as test results or code coverage reports. They are **not publishable**, and will produce an error if added to a component which is published.

Attribute name	Description	Values	compatibility and disambiguation rules
<code>org.gradl</code> <code>e.status</code>	Component level attribute, derived	Based on a <code>status scheme</code> , with a default one existing based on the source repository.	Based on the scheme in use

JVM ecosystem specific attributes

In addition to the ecosystem independent attributes defined above, the JVM ecosystem adds the following attribute:

Attribut e name	Description	Values	compatibility and disambiguation rules
<code>org.gradl</code> <code>e.jvm.ver</code> <code>sion</code>	Indicates the JVM version compatibility.	Integer using the version after the <code>1.</code> for Java 1.4 and before, the major version for Java 5 and beyond.	Defaults to the JVM version used by Gradle, lower is compatible with higher, prefers highest compatible.
<code>org.gradl</code> <code>e.jvm.env</code> <code>ironment</code>	Indicates that a variant is optimized for a certain JVM environment.	Common values are <code>standard-jvm</code> and <code>android</code> . Other values are allowed.	The attribute is used to prefer one variant over another if multiple are available, but in general all values are compatible. The default is <code>standard-jvm</code> .

Attribute name	Description	Values	compatibility and disambiguation rules
<code>org.gradle.testsuite.name</code>	Indicates the name of the <code>TestSuite</code> that produced this output.	Value is the name of the Suite.	No default, no compatibility

The JVM ecosystem also contains a number of compatibility and disambiguation rules over the different attributes. The reader willing to know more can take a look at the code for `org.gradle.api.internal.artifacts.JavaEcosystemSupport`.

Native ecosystem specific attributes

In addition to the ecosystem independent attributes defined above, the native ecosystem adds the following attributes:

Attribute name	Description	Values	compatibility and disambiguation rules
<code>org.gradle.native.debuggable</code>	Indicates if the binary was built with debugging symbols	Boolean	N/A
<code>org.gradle.native.optimized</code>	Indicates if the binary was built with optimization flags	Boolean	N/A
<code>org.gradle.native.architecture</code>	Indicates the target architecture of the binary	<code>MachineArchitecture</code> values built from constants defined in <code>MachineArchitecture</code>	None
<code>org.gradle.native.operatingSystem</code>	Indicates the target operating system of the binary	<code>OperatingSystemFamily</code> values built from constants defined in <code>OperatingSystemFamily</code>	None

Gradle plugin ecosystem specific attributes

For Gradle plugin development, the following attribute is supported since Gradle 7.0. A Gradle plugin variant can specify compatibility with a Gradle API version through this attribute.

Attribute name	Description	Values	compatibility and disambiguation rules
<code>org.gradle.plugin.api.version</code>	Indicates the Gradle API version compatibility.	Valid Gradle version strings.	Defaults to the currently running Gradle, lower is compatible with higher, prefers highest compatible.

Using a standard attribute

For this example, let's assume you are creating a library with different variants for different JVM versions.

lib/build.gradle.kts

```
plugins {
    id("java-library")
}

configurations {
    named("apiElements") {
        attributes {
            attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE, 17)
        }
    }
}
```

lib/build.gradle

```
plugins {
    id 'java-library'
}

configurations {
    apiElements {
        attributes {
            attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE, 17)
        }
    }
}
```

In the consumer project (that uses the library), you can specify the JVM version attribute when declaring dependencies.

consumer/build.gradle.kts

```
plugins {
    id("application")
}

dependencies {
```

```

implementation(project(":lib")) {
    attributes {
        attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE, 17)
    }
}
}

```

consumer/build.gradle

```

plugins {
    id 'application'
}

dependencies {
    implementation(project(':lib')) {
        attributes {
            attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE, 17)
        }
    }
}
}

```

By defining and using the JVM version attribute, you ensure that your library and its consumers are compatible with the specified JVM version. Essentially, this ensures that Gradle resolves to the variant that matches the desired JVM version.

Viewing and debugging attributes

The `dependencyInsight` task is useful for inspecting specific dependencies and their attributes, including how they are resolved:

```

$ ./gradlew dependencyInsight --configuration compileClasspath --dependency
com.example:your-library

> Task :dependencyInsight

com.example:your-library:1.0 (compileClasspath)
  variant "apiElements" [
    org.gradle.api.attributes.Attribute: org.gradle.api.attributes.Usage = [java-
api]
    org.gradle.api.attributes.Attribute: org.gradle.api.attributes.Usage = [java-
runtime]
    org.gradle.api.attributes.Attribute:
org.gradle.api.attributes.JavaLanguageVersion = [1.8]
  ]
  variant "runtimeElements" [
    org.gradle.api.attributes.Attribute: org.gradle.api.attributes.Usage = [java-

```

```
runtime]
    org.gradle.api.attributes.Attribute:
org.gradle.api.attributes.JavaLanguageVersion = [1.8]
]

Selection reasons:
- By constraint from configuration ':compileClasspath'
- Declared in build.gradle.kts

Resolved to:
com.example:your-library:1.0 (runtime)

Additional Information:
- Dependency declared in the 'implementation' configuration
- No matching variants found for the requested attributes in the
'compileClasspath' configuration
```

Declaring custom attributes

When extending Gradle with **custom attributes**, it's important to consider their long-term impact, especially if you plan to publish libraries. Custom attributes allow you to integrate **variant-aware dependency management** in your plugin, but libraries using these attributes must also ensure consumers can interpret them correctly. This is typically done by applying the corresponding plugin, which defines compatibility and disambiguation rules.

If your plugin is publicly available and libraries are published to public repositories, introducing new attributes becomes a significant responsibility. Published attributes must remain supported or have a [compatibility layer](#) in future versions of the plugin to ensure backward compatibility.

Here's an example of declaring and using custom attributes in a Gradle plugin:

lib/build.gradle.kts

```
// Define a custom attribute
val myAttribute = Attribute.of("com.example.my-attribute",
String::class.java)

configurations {
    create("myConfig") {
        // Set custom attribute
        attributes {
            attribute(myAttribute, "special-value")
        }
    }
}

dependencies {
    // Apply the custom attribute to a dependency
```

```
add("myConfig","com.google.guava:guava:31.1-jre") {  
    attributes {  
        attribute(myAttribute, "special-value")  
    }  
}
```

lib/build.gradle

```
// Define a custom attribute  
def myAttribute = Attribute.of("com.example.my-attribute", String)  
  
// Create a custom configuration  
configurations {  
    create("myConfig") {  
        // Set custom attribute  
        attributes {  
            attribute(myAttribute, "special-value")  
        }  
    }  
}  
  
dependencies {  
    // Apply the custom attribute to a dependency  
    add("myConfig", "com.google.guava:guava:31.1-jre") {  
        attributes {  
            attribute(myAttribute, "special-value")  
        }  
    }  
}
```

In this example: - A custom attribute `my-attribute` is defined. - The attribute is set on a custom configuration (`myConfig`). - When adding a dependency, the custom attribute is applied to match the configuration.

If publishing a library with this attribute, ensure that consumers apply the plugin that understands and handles `my-attribute`.

Creating attributes in a build script or plugin

Attributes are *typed*. An attribute can be created via the `Attribute<T>.of` method:

build.gradle.kts

```
// An attribute of type `String`
```

```
val myAttribute = Attribute.of("my.attribute.name", String::class.java)
// An attribute of type `Usage`
val myUsage = Attribute.of("my.usage.attribute", Usage::class.java)
```

build.gradle

```
// An attribute of type `String`
def myAttribute = Attribute.of("my.attribute.name", String)
// An attribute of type `Usage`
def myUsage = Attribute.of("my.usage.attribute", Usage)
```

Attribute types support most Java primitive classes; such as `String` and `Integer`. Or anything extending `org.gradle.api.Named`.

Attributes should always be declared in the *attribute schema* found on the `dependencies` handler:

build.gradle.kts

```
dependencies.attributesSchema {
    // registers this attribute to the attributes schema
    attribute(myAttribute)
    attribute(myUsage)
}
```

build.gradle

```
dependencies.attributesSchema {
    // registers this attribute to the attributes schema
    attribute(myAttribute)
    attribute(myUsage)
}
```

Registering an attribute with the schema is required in order to use Compatibility and Disambiguation rules that can resolve ambiguity between multiple selectable variants during [Attribute Matching](#).

Each configuration has a container of attributes. Attributes can be configured to set values:

build.gradle.kts

```
configurations {  
    create("myConfiguration") {  
        attributes {  
            attribute(myAttribute, "my-value")  
        }  
    }  
}
```

build.gradle

```
configurations {  
    myConfiguration {  
        attributes {  
            attribute(myAttribute, 'my-value')  
        }  
    }  
}
```

For attributes which type extends **Named**, the value of the attribute **must** be created via the *object factory*:

build.gradle.kts

```
configurations {  
    "myConfiguration" {  
        attributes {  
            attribute(myUsage, project.objects.named(Usage::class.java, "my-value"))  
        }  
    }  
}
```

build.gradle

```
configurations {  
    myConfiguration {  
        attributes {  
            attribute(myUsage, project.objects.named(Usage, 'my-value'))  
        }  
    }  
}
```

```
}  
}
```

Dealing with attribute matching

In Gradle, **attribute matching** and **attribute disambiguation** are key mechanisms for resolving dependencies with varying attributes.

Attribute matching allows Gradle to select compatible dependency variants based on predefined rules, even if an exact match isn't available. **Attribute disambiguation**, on the other hand, helps Gradle choose the most suitable variant when multiple compatible options exist.

Attribute compatibility rules

Attributes let the engine select *compatible variants*. There are cases where a producer may not have *exactly* what the consumer requests but has a variant that can be used.

This example defines and registers a custom compatibility rule to ensure that dependencies are selected based on their compatibility with specific Java versions:

lib/build.gradle.kts

```
// Define the compatibility rule class  
class TargetJvmVersionCompatibilityRule : AttributeCompatibilityRule<Int> {  
    // Implement the execute method which will check compatibility  
    override fun execute(details: CompatibilityCheckDetails<Int>) {  
        // Switch case to check the consumer value for supported Java  
        versions  
        when (details.consumerValue) {  
            8, 11 -> details.compatible() // Compatible with Java 8 and 11  
            else -> details.incompatible()  
        }  
    }  
}  
  
// Register the compatibility rule within the dependencies block  
dependencies {  
    attributesSchema {  
        // Add the compatibility rule for the TargetJvmVersion attribute  
        attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE) {  
            // Add the defined compatibility rule to this attribute  
  
            compatibilityRules.add(TargetJvmVersionCompatibilityRule::class.java)  
        }  
    }  
}
```

lib/build.gradle

```
// Define the compatibility rule
class TargetJvmVersionCompatibilityRule implements
AttributeCompatibilityRule<Integer> {
    @Override
    void execute(CompatibilityCheckDetails<Integer> details) {
        switch (details.consumerValue) {
            case 8:
            case 11:
                details.compatible() // Compatible with Java 8 and 11
                break
            default:
                details.incompatible("Unsupported Java version: ${details
.consumerValue}")
        }
    }
}

// Register a compatibility rule
dependencies {
    attributesSchema {
        attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE) {
            compatibilityRules.add(TargetJvmVersionCompatibilityRule)
        }
    }
}
```

Gradle provides [attribute compatibility rules](#) that can be defined for each attribute. The role of a compatibility rule is to explain which attribute values are *compatible* based on what the consumer asked for.

Attribute compatibility rules have to be registered via the [attributes schema](#).

Attribute disambiguation rules

When multiple variants of a dependency are compatible with the consumer's requested attributes, Gradle needs to decide which variant to select. This process of determining the "best" candidate among compatible options is called **attribute disambiguation**.

In Gradle, different variants might satisfy the consumer's request, but not all are equal. For example, you might have several versions of a library that are compatible with a Java version requested by the consumer. Disambiguation helps Gradle choose the most appropriate one based on additional criteria.

You can define disambiguation rules to guide Gradle in selecting the most suitable variant when multiple candidates are found. This is done by implementing an [attribute disambiguation rule](#):

```

import org.gradle.api.attributes.Attribute
import org.gradle.api.attributes.AttributeMatchingStrategy

// Define custom attribute
val javaLanguageVersion = Attribute.of("org.gradle.jvm.version", String::class.java)

// Register disambiguation rules
configurations.all {
    attributes {
        // Define the attribute matching strategy
        attribute(javaLanguageVersion, "1.8") {
            // Set up disambiguation logic
            disambiguationStrategy {
                // Example disambiguation: Prefer newer versions
                preferNewer()
            }
        }
    }
}

```

1. **Attribute Definition:** Create or reference the attribute you want to apply disambiguation rules to. Here, `javaLanguageVersion` is used.
2. **Register Disambiguation Rules:** Apply the disambiguation strategy using `disambiguationStrategy` within the `attributes` block. This example sets up a simple rule to prefer newer versions.
3. **Disambiguation Logic:** The `preferNewer()` method is a placeholder for your custom logic. You can implement more complex rules based on your requirements.

Attribute disambiguation rules have to be registered via the [attribute matching strategy](#) that you can obtain from the [attributes schema](#), which is a member of [DependencyHandler](#).

Mapping from Maven/Ivy to Gradle variants

Neither Maven nor Ivy have the concept of *variants*, which are only natively supported by Gradle Module Metadata. Gradle can still work with Maven and Ivy by using different variant derivation strategies.

Relationship with Gradle Module Metadata

Gradle Module Metadata is a metadata format for modules published on Maven, Ivy and other kinds of repositories. It is similar to the `pom.xml` or `ivy.xml` metadata file, but this format contains details about variants.

See the [Gradle Module Metadata specification](#) for more information.

Mapping of Maven POM metadata to variants

Modules published to a Maven repository are automatically converted into variant-aware modules when resolved by Gradle.

There is no way for Gradle to know which kind of *component* was published:

- a BOM that represents a Gradle platform
- a BOM used as a super-POM
- a POM that is both a platform *and* a library

The default strategy used by Java projects in Gradle is to derive 8 different variants:

- two "library" variants (attribute `org.gradle.category = library`)
 - the `compile` variant maps the `<scope>compile</scope>` dependencies. This variant is equivalent to the `apiElements` variant of the [Java Library plugin](#). All dependencies of this scope are considered *API dependencies*.
 - the `runtime` variant maps both the `<scope>compile</scope>` and `<scope>runtime</scope>` dependencies. This variant is equivalent to the `runtimeElements` variant of the [Java Library plugin](#). All dependencies of those scopes are considered *runtime dependencies*.
 - in both cases, the `<dependencyManagement>` dependencies are *not converted to constraints*
- a "sources" variant that represents the sources jar for the component
- a "javadoc" variant that represents the javadoc jar for the component
- four "platform" variants derived from the `<dependencyManagement>` block (attribute `org.gradle.category = platform`):
 - the `platform-compile` variant maps the `<scope>compile</scope>` dependency management dependencies as *dependency constraints*.
 - the `platform-runtime` variant maps both the `<scope>compile</scope>` and `<scope>runtime</scope>` dependency management dependencies as *dependency constraints*.
 - the `enforced-platform-compile` is similar to `platform-compile` but all the constraints are *forced*
 - the `enforced-platform-runtime` is similar to `platform-runtime` but all the constraints are *forced*

You can understand more about the use of platform and enforced platforms variants by looking at the [importing BOMs](#) section of the manual. By default, whenever you declare a dependency on a Maven module, Gradle is going to look for the `library` variants. However, using the `platform` or `enforcedPlatform` keyword, Gradle is now looking for one of the "platform" variants, which allows you to import the constraints from the POM files, instead of the dependencies.

Mapping of Ivy files to variants

Gradle has no built-in derivation strategy implemented for Ivy files. Ivy is a flexible format that allows you to publish arbitrary files and can be heavily customized.

If you want to implement a derivation strategy for *compile* and *runtime* variants for Ivy, you can do so with [component metadata rule](#). The *component* metadata rules API allows you to [access Ivy configurations](#) and create variants based on them. If you know that all the Ivy modules your are consuming have been published with Gradle without further customizations of the `ivy.xml` file, you can add the following rule to your build:

build.gradle.kts

```
abstract class IvyVariantDerivationRule @Inject internal
constructor(objectFactory: ObjectFactory) : ComponentMetadataRule {
    private val jarLibraryElements: LibraryElements
    private val libraryCategory: Category
    private val javaRuntimeUsage: Usage
    private val javaApiUsage: Usage

    init {
        jarLibraryElements = objectFactory.named(LibraryElements.JAR)
        libraryCategory = objectFactory.named(Category.LIBRARY)
        javaRuntimeUsage = objectFactory.named(Usage.JAVA_RUNTIME)
        javaApiUsage = objectFactory.named(Usage.JAVA_API)
    }

    override fun execute(context: ComponentMetadataContext) {
        // This filters out any non Ivy module
        if(context.getDescriptor(IvyModuleDescriptor::class) == null) {
            return
        }

        context.details.addVariant("runtimeElements", "default") {
            attributes {
                attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE,
jarLibraryElements)
                attribute(Category.CATEGORY_ATTRIBUTE, libraryCategory)
                attribute(Usage.USAGE_ATTRIBUTE, javaRuntimeUsage)
            }
        }
        context.details.addVariant("apiElements", "compile") {
            attributes {
                attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE,
jarLibraryElements)
                attribute(Category.CATEGORY_ATTRIBUTE, libraryCategory)
                attribute(Usage.USAGE_ATTRIBUTE, javaApiUsage)
            }
        }
    }

    dependencies {
        components { all<IvyVariantDerivationRule>() }
    }
}
```

build.gradle

```
abstract class IvyVariantDerivationRule implements ComponentMetadataRule {
    final LibraryElements jarLibraryElements
    final Category libraryCategory
    final Usage javaRuntimeUsage
    final Usage javaApiUsage

    @Inject
    IvyVariantDerivationRule(ObjectFactory objectFactory) {
        jarLibraryElements = objectFactory.named(LibraryElements,
LibraryElements.JAR)
        libraryCategory = objectFactory.named(Category, Category.LIBRARY)
        javaRuntimeUsage = objectFactory.named(Usage, Usage.JAVA_RUNTIME)
        javaApiUsage = objectFactory.named(Usage, Usage.JAVA_API)
    }

    void execute(ComponentMetadataContext context) {
        // This filters out any non Ivy module
        if(context.getDescriptor(IvyModuleDescriptor) == null) {
            return
        }

        context.details.addVariant("runtimeElements", "default") {
            attributes {
                attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE,
jarLibraryElements)
                attribute(Category.CATEGORY_ATTRIBUTE, libraryCategory)
                attribute(Usage.USAGE_ATTRIBUTE, javaRuntimeUsage)
            }
        }
        context.details.addVariant("apiElements", "compile") {
            attributes {
                attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE,
jarLibraryElements)
                attribute(Category.CATEGORY_ATTRIBUTE, libraryCategory)
                attribute(Usage.USAGE_ATTRIBUTE, javaApiUsage)
            }
        }
    }

    dependencies {
        components { all(IvyVariantDerivationRule) }
    }
}
```

The rule creates an `apiElements` variant based on the `compile` configuration and a `runtimeElements` variant based on the `default` configuration of each ivy module. For each variant, it sets the

corresponding [Java ecosystem attributes](#). Dependencies and artifacts of the variants are taken from the underlying configurations. If not all consumed Ivy modules follow this pattern, the rule can be adjusted or only applied to a selected set of modules.

For all Ivy modules without variants, Gradle has a fallback selection method. Gradle does *not* perform variant aware resolution and instead selects either the **default** configuration or an explicitly named configuration.

CONTROLLING DEPENDENCY RESOLUTION

Dependency Resolution

Dependency resolution in Gradle can largely be thought of as a two-step process.

First, the *graph resolution* phase constructs the dependency graph based on declared dependencies. Second, the *artifact resolution* phase fetches the actual files (artifacts) for the resolved components:

1. Graph resolution phase:

- Driven by declared dependencies and their metadata
- Uses the request attributes defined by the configuration being resolved

2. Artifact resolution phase:

- Based on nodes in the resolved dependency graph
- Matches each node to a variant and an artifact

The outcome of these processes can be accessed via different APIs, each designed for specific use cases.

1. Graph Resolution

During the *graph resolution* phase, Gradle downloads and analyzes component metadata (GMM, POM, or Ivy XML) for declared and transitive dependencies. This information is used to construct a **dependency graph**, which models the relationships between different components and their variants. You can learn about how the graph is constructed in [Understanding the Dependency Resolution Model](#).

The `ResolutionResult` API represents the output of the graph resolution phase, providing access to the resolved dependency graph without triggering artifact downloads. The graph itself focuses on component variants, not the artifacts (files) associated with those variants:

- `ResolvedComponentResult` - Represents a resolved component in the raw dependency graph.
- `ResolvedVariantResult` - Represents a resolved variant of a component in the raw dependency graph.

See [Graph Resolution](#) to learn more.

2. Artifact Resolution

Once the **dependency graph** is resolved, the *artifact resolution* phase determines which actual files (artifacts) need to be downloaded or retrieved.

An `ArtifactView` operates on top of the resolved graph, defined by the `ResolutionResult`. It allows you to query for specific artifacts based on attributes. The same attributes used during *graph resolution* typically guide *artifact selection*.

The [ArtifactView API](#) provides flexible ways to access these resolved artifacts:

- [FileCollection](#) - A flat list of files, which is the most commonly way to work with resolved artifacts.
- [ArtifactCollection](#) - Offers access to both the metadata and the files of resolved artifacts, allowing for more advanced artifact handling.

See [Artifact Resolution](#) to learn more.

Controlling Resolution

An [ArtifactTransform](#) modifies the content of an artifact during dependency resolution, such as converting JAR files into custom formats.

An [ArtifactView](#) permits customizing the artifact selection process, for example filtering components or requesting attributes that trigger artifact transforms.

When used together:

1. Register a [ArtifactTransform](#) to process artifacts.
2. Configure an [ArtifactView](#) to select artifacts based on attributes that differ from the original graph.
3. Query the transformed artifacts via the [ArtifactView](#) to access or process the transformed files.

See [Artifact Views](#) and [Artifact Transforms](#) to learn more.

Next Step: [Learn about Graph Resolution >>](#)

Dependency Graph Resolution

The output of the *graph resolution* phase is a fully resolved **dependency graph**, which is used as the input to the *artifact resolution* phase.

You can learn about how the graph is constructed in [Understanding the Dependency Resolution Model](#).

The [ResolutionResult](#) API provides access to the resolved dependency graph without triggering artifact resolution. This API presents the resolved dependency graph, where each node in the graph is a variant of a component.

Raw access to the dependency graph can be useful for a number of use cases:

- Visualizing the dependency graph, for example generating a [.dot](#) file for Graphviz.
- Exposing [diagnostics](#) about a given resolution, similar to the [dependencies](#) or [dependencyInsight](#) tasks.
- Resolving a subset of the artifacts for a dependency graph when used in conjunction with the [ArtifactView](#) API.

Consider the following function that traverses a dependency graph, starting from the root node. Callbacks are notified for each node and edge in the graph. This function can be used as a base for any use case that requires traversing a dependency graph:

build.gradle.kts

```
fun traverseGraph(
    rootComponent: ResolvedComponentResult,
    rootVariant: ResolvedVariantResult,
    nodeCallback: (ResolvedVariantResult) -> Unit,
    edgeCallback: (ResolvedVariantResult, ResolvedVariantResult) -> Unit
) {
    val seen = mutableSetOf<ResolvedVariantResult>(rootVariant)
    nodeCallback(rootVariant)

    val queue = ArrayDeque(listOf(rootVariant to rootComponent))
    while (queue.isNotEmpty()) {
        val (variant, component) = queue.removeFirst()

        // Traverse this variant's dependencies
        component.getDependenciesForVariant(variant).forEach { dependency ->
            val resolved = when (dependency) {
                is ResolvedDependencyResult -> dependency
                is UnresolvedDependencyResult -> throw dependency.failure
                else -> throw AssertionError("Unknown dependency type:
$dependency")
            }
            if (!resolved.isConstraint) {
                val toVariant = resolved.resolvedVariant

                if (seen.add(toVariant)) {
                    nodeCallback(toVariant)
                    queue.addLast(toVariant to resolved.selected)
                }

                edgeCallback(variant, toVariant)
            }
        }
    }
}
```

build.gradle

```
void traverseGraph(
    ResolvedComponentResult rootComponent,
    ResolvedVariantResult rootVariant,
    Consumer<ResolvedVariantResult> nodeCallback,
```

```

    BiConsumer<ResolvedVariantResult, ResolvedVariantResult> edgeCallback
) {
    Set<ResolvedVariantResult> seen = new HashSet<>()
    seen.add(rootVariant)
    nodeCallback(rootVariant)

    def queue = new ArrayDeque<Tuple2<ResolvedVariantResult,
ResolvedComponentResult>>()
    queue.add(new Tuple2(rootVariant, rootComponent))
    while (!queue.isEmpty()) {
        def entry = queue.removeFirst()
        def variant = entry.v1
        def component = entry.v2

        // Traverse this variant's dependencies
        component.getDependenciesForVariant(variant).each { dependency ->
            if (dependency instanceof UnresolvedDependencyResult) {
                throw dependency.failure
            }
            if ((!dependency instanceof ResolvedDependencyResult)) {
                throw new RuntimeException("Unknown dependency type:
$dependency")
            }

            def resolved = dependency as ResolvedDependencyResult
            if (!dependency.constraint) {
                def toVariant = resolved.resolvedVariant

                if (seen.add(toVariant)) {
                    nodeCallback(toVariant)
                    queue.add(new Tuple2(toVariant, resolved.selected))
                }

                edgeCallback(variant, toVariant)
            }
        }
    }
}

```

This function starts at the root variant, and performs a breadth-first traversal of the graph. The **ResolutionResult** API is lenient, so it is important to check whether a visited edge is unresolved (failed) or resolved. With this function, the node callback is always called before the edge callback for any given node.

Below, we leverage the above traversal function to transform a dependency graph into a **.dot** file for visualization:

build.gradle.kts

```
abstract class GenerateDot : DefaultTask() {

    @get:Input
    abstract val rootComponent: Property<ResolvedComponentResult>

    @get:Input
    abstract val rootVariant: Property<ResolvedVariantResult>

    @TaskAction
    fun traverse() {
        println("digraph {")
        traverseGraph(
            rootComponent.get(),
            rootVariant.get(),
            { node -> println("    ${toNodeId(node)} [shape=box]") },
            { from, to -> println("    ${toNodeId(from)} -> ${toNodeId(to)}") }
        )
        println("}")
    }

    fun toNodeId(variant: ResolvedVariantResult): String {
        return "\"${variant.owner.displayName}:${variant.displayName}\""
    }
}
```

build.gradle

```
abstract class GenerateDot extends DefaultTask {

    @Input
    abstract Property<ResolvedComponentResult> getRootComponent()

    @Input
    abstract Property<ResolvedVariantResult> getRootVariant()

    @TaskAction
    void traverse() {
        println("digraph {")
        traverseGraph(
            rootComponent.get(),
            rootVariant.get(),
            node -> { println("    ${toNodeId(node)} [shape=box]") },
            (from, to) -> { println("    ${toNodeId(from)} -> ${toNodeId(to)}") }
        )
    }
```

```

    )
    println("{}")
}

String toNodeId(ResolvedVariantResult variant) {
    return "\"${variant.owner.displayName}:${variant.displayName}\""
}
}

```

NOTE

A proper implementation would not use `println` but would write to an output file. For more details on declaring task inputs and outputs, see the [Writing Tasks](#) section.

When we register the task, we use the `ResolutionResult` API to access the root component and root variant of the `runtimeClasspath` configuration:

build.gradle.kts

```

tasks.register<GenerateDot>("generateDot") {
    rootComponent = runtimeClasspath.flatMap {
        it.incoming.resolutionResult.rootComponent
    }
    rootVariant = runtimeClasspath.flatMap {
        it.incoming.resolutionResult.rootVariant
    }
}

```

build.gradle

```

tasks.register("generateDot", GenerateDot) {
    rootComponent = configurations.runtimeClasspath.incoming.
resolutionResult.rootComponent
    rootVariant = configurations.runtimeClasspath.incoming.resolutionResult
.rootVariant
}

```

NOTE

This example uses incubating APIs.

Running this task, we get the following output:

```

digraph {
    "root project ::runtimeClasspath" [shape=box]
}

```

```

"com.google.guava:guava:33.2.1-jre:jreRuntimeElements" [shape=box]
"root project ::runtimeClasspath" -> "com.google.guava:guava:33.2.1-
jre:jreRuntimeElements"
"com.google.guava:failureaccess:1.0.2:runtime" [shape=box]
"com.google.guava:guava:33.2.1-jre:jreRuntimeElements" ->
"com.google.guava:failureaccess:1.0.2:runtime"
"com.google.guava:listenablefuture:9999.0-empty-to-avoid-conflict-with-
guava:runtime" [shape=box]
"com.google.guava:guava:33.2.1-jre:jreRuntimeElements" ->
"com.google.guava:listenablefuture:9999.0-empty-to-avoid-conflict-with-guava:runtime"
"com.google.code.findbugs:jsr305:3.0.2:runtime" [shape=box]
"com.google.guava:guava:33.2.1-jre:jreRuntimeElements" ->
"com.google.code.findbugs:jsr305:3.0.2:runtime"
"org.checkerframework:checker-qual:3.42.0:runtimeElements" [shape=box]
"com.google.guava:guava:33.2.1-jre:jreRuntimeElements" ->
"org.checkerframework:checker-qual:3.42.0:runtimeElements"
"com.google.errorprone:error_prone_annotations:2.26.1:runtime" [shape=box]
"com.google.guava:guava:33.2.1-jre:jreRuntimeElements" ->
"com.google.errorprone:error_prone_annotations:2.26.1:runtime"
}

```

Compare this to the output of the **dependencies** task:

```

runtimeClasspath
\--- com.google.guava:guava:33.2.1-jre
     +--- com.google.guava:failureaccess:1.0.2
     +--- com.google.guava:listenablefuture:9999.0-empty-to-avoid-conflict-with-guava
     +--- com.google.code.findbugs:jsr305:3.0.2
     +--- org.checkerframework:checker-qual:3.42.0
     \--- com.google.errorprone:error_prone_annotations:2.26.1

```

Notice how the graph is the same for both representations.

Next Step: [Learn about Artifact Resolution](#) >>

Artifact Resolution

After constructing a dependency graph, Gradle can perform **artifact resolution** on the resolved graph.

Gradle APIs can be used to influence the process of **artifact selection** — the mapping of a graph to a set of artifacts.

Gradle can then expose the results of artifact selection as an **ArtifactCollection**. More commonly, the results are exposed as a **FileCollection**, which is a flat list of files.

Artifact selection

Artifact selection operates on the dependency graph on a node-by-node basis. Each node in the graph may expose multiple sets of artifacts, but only one of those sets may be selected. For example, the `runtimeElements` variant of the Java plugins exposes a `jar`, `classes`, and `resources` artifact set. These three artifact sets represent the same distributable, but in different forms.

For each node (variant) in a graph, Gradle performs attribute matching over each set of artifacts exposed by that node to determine the best artifact set. If no artifact sets match the requested attributes, Gradle will attempt to construct an [artifact transform](#) chain to satisfy the request.

For more details on the attribute matching process, see the [attribute matching](#) section.

Implicit artifact selection

By default, the attributes used for artifact selection are the same as those used for variant selection during graph resolution. These attributes are specified by the `Configuration#getAttributes()` property.

To perform artifact selection (and implicitly, graph resolution) using these default attributes, use the `FileCollection` and `ArtifactCollection` APIs.

NOTE

Files can also be accessed from the configuration's `ResolvedConfiguration`, `LenientConfiguration`, `ResolvedArtifact` and `ResolvedDependency` APIs. However, these APIs are in maintenance mode and are discouraged for use in new development. These APIs perform artifact selection using the default attributes.

Resolving files

To resolve files, we first define a task that accepts a `ConfigurableFileCollection` as input:

build.gradle.kts

```
abstract class ResolveFiles : DefaultTask() {

    @get:InputFiles
    abstract val files: ConfigurableFileCollection

    @TaskAction
    fun print() {
        files.forEach {
            println(it.name)
        }
    }
}
```

build.gradle

```
abstract class ResolveFiles extends DefaultTask {  
  
    @InputFiles  
    abstract ConfigurableFileCollection getFiles()  
  
    @TaskAction  
    void print() {  
        files.each {  
            println(it.name)  
        }  
    }  
}
```

Then, we can wire up a resolvable configuration's files to the task's input. The **Configuration** directly implements **FileCollection** and can be wired directly. Alternatively, wiring through **Configuration#getIncoming()** is a more explicit approach:

build.gradle.kts

```
tasks.register<ResolveFiles>("resolveConfiguration") {  
    files.from(configurations.runtimeClasspath)  
}  
tasks.register<ResolveFiles>("resolveIncomingFiles") {  
    files.from(configurations.runtimeClasspath.map { it.incoming.files })  
}
```

build.gradle

```
tasks.register("resolveConfiguration", ResolveFiles) {  
    files.from(configurations.runtimeClasspath)  
}  
tasks.register("resolveIncomingFiles", ResolveFiles) {  
    files.from(configurations.runtimeClasspath.incoming.files)  
}
```

Running both of these tasks, we can see the output is identical:

```
> Task :resolveConfiguration  
junit-platform-commons-1.11.0.jar
```

```
junit-jupiter-api-5.11.0.jar
opentest4j-1.3.0.jar
```

```
> Task :resolveIncomingFiles
junit-platform-commons-1.11.0.jar
junit-jupiter-api-5.11.0.jar
opentest4j-1.3.0.jar
```

Resolving artifacts

Instead of consuming the files directly from the implicit artifact selection process, we can consume the artifacts, which contain both the files and the metadata.

This process is slightly more complicated, as in order to maintain Configuration Cache compatibility, we need to split the fields of `ResolvedArtifactResult` into two task inputs:

build.gradle.kts

```
data class ArtifactDetails(
    val id: ComponentArtifactIdentifier,
    val variant: ResolvedVariantResult
)

abstract class ResolveArtifacts : DefaultTask() {

    @get:Input
    abstract val details: ListProperty<ArtifactDetails>

    @get:InputFiles
    abstract val files: ListProperty<File>

    fun from(artifacts: Provider<Set<ResolvedArtifactResult>>) {
        details.set(artifacts.map {
            it.map { artifact -> ArtifactDetails(artifact.id,
artifact.variant) }
        })
        files.set(artifacts.map {
            it.map { artifact -> artifact.file }
        })
    }

    @TaskAction
    fun print() {
        assert(details.get().size == files.get().size)
        details.get().zip(files.get()).forEach { (details, file) ->
            println("${details.variant.displayName}:${file.name}")
        }
    }
}
```

```
}
```

build.gradle

```
class ArtifactDetails {
    ComponentArtifactIdentifier id
    ResolvedVariantResult variant

    ArtifactDetails(ComponentArtifactIdentifier id, ResolvedVariantResult
variant) {
        this.id = id
        this.variant = variant
    }
}

abstract class ResolveArtifacts extends DefaultTask {

    @Input
    abstract ListProperty<ArtifactDetails> getDetails()

    @InputFiles
    abstract ListProperty<File> getFiles()

    void from(Provider<Set<ResolvedArtifactResult>> artifacts) {
        details.set(artifacts.map {
            it.collect { artifact -> new ArtifactDetails(artifact.id,
artifact.variant) }
        })
        files.set(artifacts.map {
            it.collect { artifact -> artifact.file }
        })
    }

    @TaskAction
    void print() {
        List<ArtifactDetails> allDetails = details.get()
        List<File> allFiles = files.get()

        assert allDetails.size() == allFiles.size()
        for (int i = 0; i < allDetails.size(); i++) {
            def details = allDetails.get(i)
            def file = allFiles.get(i)
            println("${details.variant.displayName}:${file.name}")
        }
    }
}
```

This task is initialized similarly to the file resolution task:

build.gradle.kts

```
tasks.register<ResolveArtifacts>("resolveIncomingArtifacts") {  
    from(configurations.runtimeClasspath.flatMap {  
        it.incoming.artifacts.resolvedArtifacts })  
}
```

build.gradle

```
tasks.register("resolveIncomingArtifacts", ResolveArtifacts) {  
    from(configurations.runtimeClasspath.incoming.artifacts.  
        resolvedArtifacts)  
}
```

Running this task, we can see that file metadata is included in the output:

```
org.junit.platform:junit-platform-commons:1.11.0 variant runtimeElements:junit-  
platform-commons-1.11.0.jar  
org.junit.jupiter:junit-jupiter-api:5.11.0 variant runtimeElements:junit-jupiter-api-  
5.11.0.jar  
org.opentest4j:opentest4j:1.3.0 variant runtimeElements:opentest4j-1.3.0.jar
```

Customizing artifact selection

An **ArtifactView** operates on top of the resolved dependency graph (i.e., **ResolutionResult**) but allows you to apply *different attributes*.

When you call a configuration's **getFiles()**, Gradle selects artifacts based on the attributes used during graph resolution. However, an **ArtifactView** is more flexible. It allows you to resolve artifacts from the graph with *custom attributes*.

An **ArtifactView** allows you to:

1. Query artifacts with different attributes:

- Suppose the graph resolved a dependency's **runtime** variant. You can use an **ArtifactView** to extract artifacts from its **api** variant instead, even if they weren't originally part of the resolved graph.

2. Extract specific types of artifacts:

- You can request only the **.jar** files or a specific artifact type (e.g., sources, Javadoc) by specifying an attribute like **artifactType**.

3. Avoid side effects:

- Using an `ArtifactView` allows you to extract artifacts without changing the underlying dependency resolution logic or configuration state.

In the following example, a producer project creates a library with the following variants and their attributes:

Next Step: [Learn about Artifact Views >>](#)

Artifact Views

In some cases, it is desirable to customize the artifact resolution process. The `ArtifactView` API is the primary mechanism for influencing artifact selection in Gradle.

An `ArtifactView` operates on top of the resolved graph but allows you to apply *different attributes*. It lets you retrieve artifacts that match a new set of criteria, even if they were not part of the original graph resolution.

An `ArtifactView` can:

1. **Select Alternative Variants** such as sources or javadoc, for an entire resolution: Normally, an artifact must match both the graph's attributes and the `ArtifactView` attributes. With `withVariantReselection`, you can select artifacts from any available variants within a component.
2. **Perform Lenient Artifact Selection** and resolution: Using `ArtifactView` with `lenient=true` allows you to ignore missing dependencies and other errors.
3. **Filter Selected Artifacts:** Using `ArtifactView` with `componentFilter` lets you exclude specific components from the selected artifacts.
4. **Trigger Transforms:** Trigger an `ArtifactTransform` to change an artifact from one type to another.

NOTE

The `ArtifactView` can produce results as both a `FileCollection` and an `ArtifactCollection`. The examples below only demonstrate using a `FileCollection` as the output.

1. Performing variant reselection

Standard artifact selection can only select between variants of the component selected by the result of graph selection. However, in some cases, it may be desirable to select artifacts from a variant *parallel* to the graph node being selected.

Consider the example component structure below, describing a typical local Java library with sources and javadoc:

```
variant 'apiElements'  
    artifact set 'jar'  
    artifact set 'classes'
```

```

    artifact set 'resources'
variant 'runtimeElements'
    artifact set 'jar'
    artifact set 'classes'
    artifact set 'resources'
variant 'javadocElements'
    artifact set 'jar'
variant 'sourcesElements'
    artifact set 'jar'

```

Resolving a Java runtime classpath will select the `runtimeElements` variant from the above example component. During standard artifact selection, Gradle will select solely from the artifact sets under `runtimeElements`.

However, it is common to want to select all sources or all javadoc for every node in the graph. Consider the following example which selects all sources for a given runtime classpath:

NOTE This example uses incubating APIs.

build.gradle.kts

```

tasks.register<ResolveFiles>("resolveSources") {
    files.from(configurations.runtimeClasspath.map {
        it.incoming.artifactView {
            withVariantReselection()
            attributes {
                attribute(Usage.USAGE_ATTRIBUTE,
objects.named(Usage.JAVA_RUNTIME));
                attribute(Category.CATEGORY_ATTRIBUTE,
objects.named(Category.DOCUMENTATION));
                attribute(Bundling.BUNDLING_ATTRIBUTE,
objects.named(Bundling.EXTERNAL));
                attribute(DocsType.DOCS_TYPE_ATTRIBUTE,
objects.named(DocsType.SOURCES));
            }
        }.files
    })
}

```

build.gradle

```

tasks.register("resolveSources", ResolveFiles) {
    files.from(configurations.runtimeClasspath.incoming.artifactView {
        withVariantReselection()
        attributes {
            attribute(Usage.USAGE_ATTRIBUTE, objects.named(Usage, Usage

```

```

        .JAVA_RUNTIME));
        attribute(Category.CATEGORY_ATTRIBUTE, objects.named(Category,
Category.DOCUMENTATION));
        attribute(Bundling.BUNDLING_ATTRIBUTE, objects.named(Bundling,
Bundling.EXTERNAL));
        attribute(DocsType.DOCS_TYPE_ATTRIBUTE, objects.named(DocsType,
DocsType.SOURCES));
    }
}.files)
}

```

Using the `ArtifactView.withVariantReselection()` API, Gradle will optionally perform graph variant selection again *before* performing artifact selection on the new selected variant. When Gradle selects artifacts for the `runtimeElements` node, it will use the attributes specified on the `ArtifactView` to reselect the graph variant, thus selecting the `sourcesElements` variant instead. Then, traditional artifact selection will be performed on the `sourcesElements` variant to select the `jar` artifact set.

As a result, the sources jar is resolved for each node:

```

junit-platform-commons-1.11.0-sources.jar
junit-jupiter-api-5.11.0-sources.jar
opentest4j-1.3.0-sources.jar

```

When this API is used, the attributes used for variant reselection are specified solely by the `ArtifactView.getAttributes()` method. The graph resolution attributes specified on the configuration are completely ignored during variant reselection.

2. Performing lenient artifact selection and resolution

The `ArtifactView` API can also be used to perform lenient artifact resolution. This allows artifact resolution to be performed on a graph that contains failures—for example when a requested module was not found, the requested module version did not exist, or a conflict was not resolved. Furthermore, lenient artifact resolution can be used to resolve artifacts when the graph was successfully resolved, but the corresponding artifacts could not be downloaded.

Consider the following example, where some dependencies may not exist:

build.gradle.kts

```

dependencies {
    implementation("does:not:exist")
    implementation("org.junit.jupiter:junit-jupiter-api:5.11.0")
}

```

build.gradle

```
dependencies {  
    implementation("does:not:exist")  
    implementation("org.junit.jupiter:junit-jupiter-api:5.11.0")  
}
```

Lenient resolution is performed by using the `ArtifactView.lenient()` method:

build.gradle.kts

```
tasks.register<ResolveFiles>("resolveLenient") {  
    files.from(configurations.runtimeClasspath.map {  
        it.incoming.artifactView {  
            isLenient = true  
        }.files  
    })  
}
```

build.gradle

```
tasks.register("resolveLenient", ResolveFiles) {  
    files.from(configurations.runtimeClasspath.incoming.artifactView {  
        lenient = true  
    }.files)  
}
```

We can see that the task succeeds with the failing artifact omitted:

```
> Task :resolveLenient  
junit-platform-commons-1.11.0.jar  
junit-jupiter-api-5.11.0.jar  
opentest4j-1.3.0.jar  
  
BUILD SUCCESSFUL in 0s
```

3. Filtering selected artifacts

The `ArtifactView` API can be used to filter specific artifacts from the resulting `FileCollection` or `ArtifactCollection`.

`ArtifactViews` allow results to be filtered on a per-component basis. Using the `ArtifactView.componentFilter(Action)` method, artifacts from the selected variant can be filtered from the resolved result. The action is passed the `ComponentIdentifier` of the component that owns the variant that artifacts are being selected for.

Consider the following example, where we have one project dependency and one external dependency:

build.gradle.kts

```
dependencies {
    implementation(project(":other"))
    implementation("org.junit.jupiter:junit-jupiter-api:5.11.0")
}
```

build.gradle

```
dependencies {
    implementation(project(":other"))
    implementation("org.junit.jupiter:junit-jupiter-api:5.11.0")
}
```

Using the `componentFilter` method, we can specify filters that select only artifacts of a certain type:

build.gradle.kts

```
tasks.register<ResolveFiles>("resolveProjects") {
    files.from(configurations.runtimeClasspath.map {
        it.incoming.artifactView {
            componentFilter {
                it is ProjectComponentIdentifier
            }
        }.files
    })
}
tasks.register<ResolveFiles>("resolveModules") {
    files.from(configurations.runtimeClasspath.map {
        it.incoming.artifactView {
            componentFilter {
                it is ModuleComponentIdentifier
            }
        }.files
    })
}
```

```
}
```

build.gradle

```
tasks.register("resolveProjects", ResolveFiles) {
    files.from(configurations.runtimeClasspath.incoming.artifactView {
        componentFilter {
            it instanceof ProjectComponentIdentifier
        }
    }.files)
}
tasks.register("resolveModules", ResolveFiles) {
    files.from(configurations.runtimeClasspath.incoming.artifactView {
        componentFilter {
            it instanceof ModuleComponentIdentifier
        }
    }.files)
}
```

Notice how we resolve project dependencies and module dependencies separately:

```
> Task :resolveProjects
other.jar

> Task :resolveModules
junit-platform-commons-1.11.0.jar
junit-jupiter-api-5.11.0.jar
opentest4j-1.3.0.jar
```

4. Triggering artifact transforms

An **ArtifactView** can be used to trigger artifact selection using *attributes* different from those used to resolve the graph.

For each node in the graph, artifact selection is performed for that node. Most commonly, this API is used to request attributes that are not present on the selected artifact. When Gradle cannot find a matching set of artifacts from the node in question, it will attempt to satisfy the request by transforming the available artifact using the artifact transforms registered on the project.

Below, we use the unzip example from the [artifact transforms](#) chapter to demonstrate how to use the **ArtifactView** API to request attributes that trigger a transform:

build.gradle.kts

```
tasks.register<ResolveFiles>("resolveTransformedFiles") {  
    files.from(configurations.runtimeClasspath.map {  
        it.incoming.artifactView {  
            attributes {  
                attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE,  
objects.named(LibraryElements.CLASSES_AND_RESOURCES))  
                attribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE,  
ArtifactTypeDefinition.DIRECTORY_TYPE)  
            }  
        }.files  
    })  
}
```

build.gradle

```
tasks.register("resolveTransformedFiles", ResolveFiles) {  
    files.from(configurations.runtimeClasspath.incoming.artifactView {  
        attributes {  
            attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE, objects  
.named(LibraryElements, LibraryElements.CLASSES_AND_RESOURCES))  
            attribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE,  
ArtifactTypeDefinition.DIRECTORY_TYPE)  
        }  
    }.files)  
}
```

Gradle performs artifact selection using the graph resolution attributes specified on the configuration, concatenated with the attributes specified in the `attributes` block of the `ArtifactView`.

The task output shows that the artifacts have been transformed:

```
junit-platform-commons-1.11.0.jar-unzipped  
junit-jupiter-api-5.11.0.jar-unzipped  
opentest4j-1.3.0.jar-unzipped
```

The transform code (`ArtifactTransform`) used to uncompress the JAR file (from `ZIP` to `UNZIP`) can be seen in the next chapter.

Next Step: [Learn about Artifact Transforms >>](#)

Artifact Transforms

What if you want to make changes to the files contained in one of your dependencies before you use it?

For example, you might want to unzip a compressed file, adjust the contents of a JAR, or delete unnecessary files from a dependency that contains multiple files prior to using the result in a task.

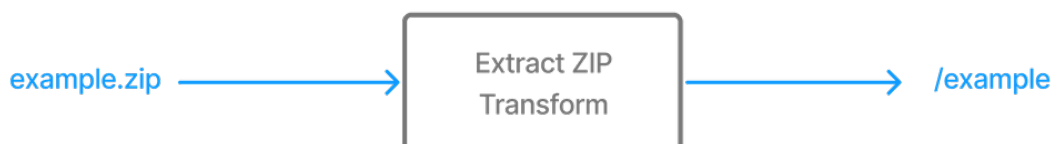
Gradle has a built-in feature for this called **Artifact Transforms**. With Artifact Transforms, you can modify, add to, remove from the set files (or artifacts) - like JAR files - contained in a dependency. This is done as the last step when resolving artifacts, before tasks or tools like the IDE can consume the artifacts.

Artifact Transforms Overview

Each component exposes a set of **variants**, where each variant is identified by a set of **attributes** (i.e., key-value pairs such as `debug=true`).

When Gradle resolves a configuration, it looks at each dependency, resolves it to a component, and selects the corresponding variant from that component that matches the requested attributes. If the component does not have a matching variant, resolution fails **unless** Gradle can construct a sequence of transformations that will modify an existing artifact to create a valid match (without changing its transitive dependencies).

Artifact Transforms are a mechanism for converting one type of artifact into another during the build process. They provide the consumer an efficient and flexible mechanism for transforming the artifacts of a given producer to the required format without needing the producer to expose variants in that format.

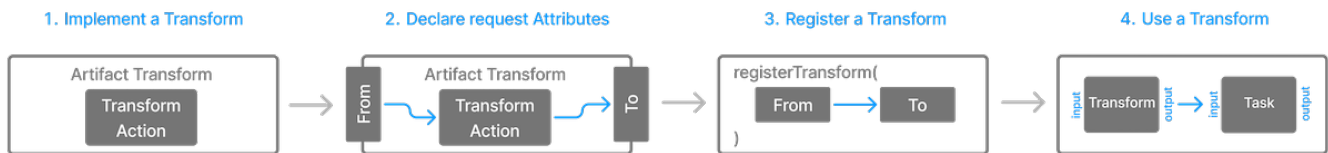


Artifact Transforms are a lot like tasks. They are units of work with some inputs and outputs. Mechanisms like **UP-TO-DATE** and caching work for transforms as well.



The primary difference between tasks and transforms is how they are scheduled and put into the chain of actions Gradle executes when a build configures and runs. At a high level, transforms always run before tasks because they are executed during dependency resolution. Transforms modify artifacts **BEFORE** they become an input to a task.

Here's a brief overview of how to create and use Artifact Transforms:



- 1. Implement a Transform:** You define an artifact transform by creating a class that implements the `TransformAction` interface. This class specifies how the input artifact should be transformed into the output artifact.
- 2. Declare request Attributes:** Attributes (key-value pairs used to describe different variants of a component) like `org.gradle.usage=java-api` and `org.gradle.usage=java-runtime` are used to specify the desired artifact format or type.
- 3. Register a Transform:** You register the transform by using the `registerTransform()` method of the `dependencies` block. This method tells Gradle that a transform can be used to modify the artifacts of any variant that possesses the given "from" attributes. It also tells Gradle what new set of "to" attributes will describe the format or type of the resulting artifacts.
- 4. Use the Transform:** When a resolution requires an artifact that isn't already present in the selected component (because none of the actual artifact possess compatible attributes to the requested attributes), Gradle doesn't just give up! Instead, Gradle first **automatically** searches all registered transforms to see if it can construct a chain of transformations that will ultimately produce a match. If Gradle finds such a chain, it then runs each transform in sequence, and delivers the transformed artifacts as a result.

1. Implement a Transform

A transform is typically written as an abstract class that implements the `TransformAction` interface. It can optionally have parameters defined in a separate interface.

Each transform has exactly one input artifact. It must be annotated with the `@InputArtifact` annotation.

Then, you implement the `transform(TransformOutputs)` method from the `TransformAction` interface. This method's implementation defines what the transform should do when triggered. The method has a `TransformOutputs` parameter that you use to tell Gradle what artifacts the transform produces.

Here, `MyTransform` is the custom transform action that converts a `jar` artifact to a `transformed-jar` artifact:

build.gradle.kts

```
abstract class MyTransform : TransformAction<TransformParameters.None> {
    @get:InputArtifact
    abstract val inputArtifact: Provider<FileSystemLocation>

    override fun transform(outputs: TransformOutputs) {
        val inputFile = inputArtifact.get().asFile
    }
}
```

```

        val outputFile = outputs.file(inputFile.name.replace(".jar", "-transformed.jar"))
        // Perform transformation logic here
        inputFile.copyTo(outputFile, overwrite = true)
    }
}

```

build.gradle

```

abstract class MyTransform implements TransformAction<TransformParameters
    .None> {
    @InputArtifact
    abstract Provider<FileSystemLocation> getInputArtifact()

    @Override
    void transform(TransformOutputs outputs) {
        def inputFile = inputArtifact.get().asFile
        def outputFile = outputs.file(inputFile.name.replace(".jar", "-transformed.jar"))
        // Perform transformation logic here
        inputFile.withInputStream { input ->
            outputFile.withOutputStream { output ->
                output << input
            }
        }
    }
}

```

2. Declare request Attributes

Attributes specify the required properties of a dependency.

Here we specify that we need the **transformed-jar** format for the **runtimeClasspath** configuration:

build.gradle.kts

```

configurations.named("runtimeClasspath") {
    attributes {
        attribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE,
            "transformed-jar")
    }
}

```

build.gradle

```
configurations.named("runtimeClasspath") {  
    attributes {  
        attribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE,  
"transformed-jar")  
    }  
}
```

3. Register a Transform

A transform must be registered using the `dependencies.registerTransform()` method.

Here, our transform is registered with the `dependencies` block:

build.gradle.kts

```
dependencies {  
    registerTransform(MyTransform::class) {  
        from.attribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE, "jar")  
        to.attribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE,  
"transformed-jar")  
    }  
}
```

build.gradle

```
dependencies {  
    registerTransform(MyTransform) {  
        from.attribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE, "jar")  
        to.attribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE,  
"transformed-jar")  
    }  
}
```

"To" attributes are used to describe the format or type of the artifacts that this transform can use as an input, and "from" attributes to describe the format or type of the artifacts that it produces as an output.

4. Use the Transform

During a build, Gradle **automatically** runs registered transforms to satisfy a resolution request if a

match is not directly available.

Since no variants exist supplying artifacts of requested format (as none contain the `artifactType` attribute with a value of `"transformed-jar"`), Gradle attempts to construct a chain of transformations that will supply an artifact matching the requested attributes.

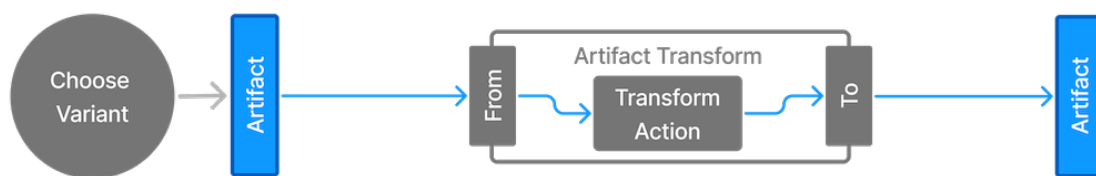
Gradle's search finds `MyTransform`, which is registered as producing the requested format, so it will automatically be run. Running this transform action modifies the artifacts of an existing source variant to produce new artifacts that are delivered to the consumer, in the requested format.

Gradle produces a "virtual artifact set" of the component as part of this process.

Understanding Artifact Transforms

Dependencies can have different `variants`, essentially different versions or forms of the same dependency. These variants can each provide a different artifact set, meant to satisfy different use cases, such as compiling code, browsing documentation or running applications.

Each variant is identified by a set of `attributes`. Attributes are key-value pairs that describe specific characteristics of the variant.



Let's use the following example where an external Maven dependency has two variants:

Table 6. Maven Dependencies

Variant	Description
<code>org.gradle.usage=java-api</code>	Used for compiling against the dependency.
<code>org.gradle.usage=java-runtime</code>	Used for running an application that uses the dependency.

And a project dependency has even more variants:

Table 7. Project Dependencies

Variant	Description
<code>org.gradle.usage=java-api</code> <code>org.gradle.libraryelements=classes</code>	Represents classes directories.
<code>org.gradle.usage=java-api</code> <code>org.gradle.libraryelements=jar</code>	Represents a packaged JAR file, containing classes and resources.

The variants of a dependency may differ in their transitive dependencies or in the set of artifacts they contain, or both.

For example, the `java-api` and `java-runtime` variants of the Maven dependency only differ in their transitive dependencies, and both use the same artifact — the JAR file. For the project dependency, the `java-api,classes` and the `java-api,jars` variants have the same transitive dependencies but different artifacts — the `classes` directories and the `JAR` files respectively.

When Gradle resolves a configuration, it uses the attributes defined to select the appropriate variant of each dependency. The attributes that Gradle uses to determine which variant to select are called the **requested attributes**.

For example, if a configuration requests `org.gradle.usage=java-api` and `org.gradle.libraryelements=classes`, Gradle will select the variant of each dependency that matches these attributes (in this case, classes directories intended for use as an API during compilation). Matches do not have to be exact, as some attribute values can be identified to Gradle as **compatible** with other values and used interchangeably during **matching**.

Sometimes, a dependency might not have a variant with attributes that match the requested attributes. In such cases, Gradle can transform one variant's artifacts into another "virtual artifact set" by modifying its artifacts without changing its transitive dependencies.

IMPORTANT

Gradle will not attempt to select or run Artifact Transforms when a variant of the dependency matching the requested attributes already exists.

For example, if the requested variant is `java-api,classes`, but the dependency only has `java-api,jar`, Gradle can potentially transform the `JAR` file into a `classes` directory by unzipping it using an Artifact Transform that is registered with these attributes.

TIP

Gradle applies transformations to artifacts, not variants.

Executing Artifact Transforms

Gradle automatically selects Artifact Transforms as needed to satisfy resolution requests.

To run an Artifact Transform, you can configure a custom [Artifact View](#) to request an artifact set that is not exposed by any variant of the target component.

When resolving the `ArtifactView` Gradle will search for appropriate Artifact Transforms based on the requested attributes in the view. Gradle will run these transformations on the original artifacts found in a variant of the target component to produce a result compatible with the attributes in the view.

In the example below, the `TestTransform` class defines a transformation that is registered to process artifacts of type "jar" into artifacts of type "stub":

build.gradle.kts

```
// The TestTransform class implements TransformAction,  
// transforming input JAR files into text files with specific content  
abstract class TestTransform : TransformAction<TransformParameters.None> {
```

```

@get:InputArtifact
abstract val inputArtifact: Provider<FileSystemLocation>

    override fun transform(outputs: TransformOutputs) {
        val outputFile = outputs.file("transformed-stub.txt")
        outputFile.writeText("Transformed from
${inputArtifact.get().asFile.name}")
    }
}

// The transform is registered to convert artifacts from the type "jar" to
"stub"
dependencies {
    registerTransform(TestTransform::class.java) {
        from.attribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE, "jar")
        to.attribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE, "stub")
    }
}

dependencies {
    runtimeOnly("com.github.javafaker:javafaker:1.0.2")
}

// The testArtifact task queries and prints the attributes of resolved
artifacts,
// showing the type conversion in action.
tasks.register("testArtifact") {
    val resolvedArtifacts =
configurations.runtimeClasspath.get().incoming.artifactView {
    attributes {
        attribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE, "stub")
    }
}.artifacts.resolvedArtifacts

    resolvedArtifacts.get().forEach {
        println("Resolved artifact variant:")
        println("- ${it.variant}")
        println("Resolved artifact attributes:")
        println("- ${it.variant.attributes}")
        println("Resolved artifact type:")
        println("-
${it.variant.attributes.getAttribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATT
RIBUTE)}")
    }
}

```

build.gradle

```

// The TestTransform class implements TransformAction,

```

```

// transforming input JAR files into text files with specific content
abstract class TestTransform implements TransformAction<TransformParameters
.None> {
    @InputArtifact
    abstract Provider<FileSystemLocation> getInputArtifact()

    @Override
    void transform(TransformOutputs outputs) {
        def outputFile = outputs.file("transformed-stub.txt")
        outputFile.text = "Transformed from ${getInputArtifact().get().
asFile.name}"
    }
}

// The transform is registered to convert artifacts from the type "jar" to
"stub"
dependencies {
    registerTransform(TestTransform) {
        from.attribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE, "jar")
        to.attribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE, "stub")
    }
}

dependencies {
    runtimeOnly("com.github.javafaker:javafaker:1.0.2")
}

// The testArtifact task queries and prints the attributes of resolved
artifacts,
// showing the type conversion in action.
tasks.register("testArtifact") {
    def resolvedArtifacts = configurations.runtimeClasspath.incoming
.artifactView {
        attributes {
            attribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE, "stub")
        }
    }.artifacts.resolvedArtifacts

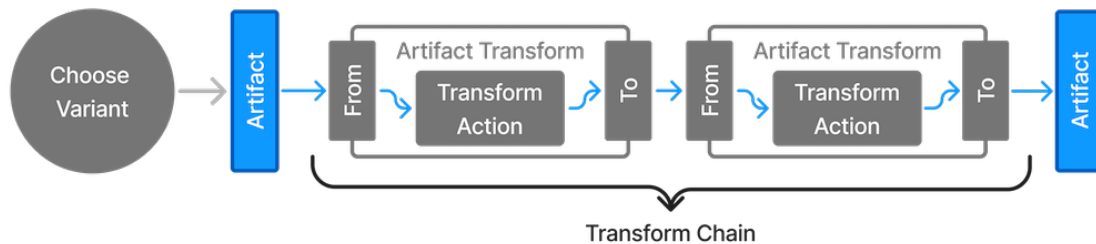
    resolvedArtifacts.get().each {
        println "Resolved artifact variant:"
        println "- ${it.variant}"
        println "Resolved artifact attributes:"
        println "- ${it.variant.attributes}"
        println "Resolved artifact type:"
        println "- ${it.variant.attributes.getAttribute
(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE)}"
    }
}

```

The `testArtifact` task resolves artifacts of type "stub" using the `runtimeClasspath` configuration. This is achieved by creating an `ArtifactView` that filters for artifacts with `ARTIFACT_TYPE_ATTRIBUTE = "stub"`.

Understanding Artifact Transforms Chains

When Gradle resolves a configuration and a variant in the graph does not have an artifact set with the requested attributes, it attempts to find a chain of one or more Artifact Transforms that can be run sequentially to create the desired artifact set. This process is called **Artifact Transform selection**:



The Artifact Transform Selection Process:

1. Start with requested Attributes:

- Gradle starts with the attributes specified on the configuration being resolved, appends any attributes specified on an `ArtifactView`, and finally appends any attributes declared directly on the dependency.
- It considers all registered transforms that modify these attributes.

2. Find a path to existing Variants:

- Gradle works backwards, trying to find a path from the requested attributes to an existing variant.

For example, if the `minified` attribute has values `true` and `false`, and a transform can change `minified=false` to `minified=true`, Gradle will use this transform if only `minified=false` variants are available but `minified=true` is requested.

Gradle selects a chain of transforms using the following process:

- If there is only one possible chain that produces the requested attributes, it is selected.
- If there are multiple such chains, then only the shortest chains are considered.
- If there are still multiple chains remaining that are equally suitable but produce different results, the selection fails, and an error is reported.
- If all the remaining chains produce the same set of resulting attributes, Gradle arbitrarily selects one.

How can multiple chains produce different suitable results? Transforms can alter multiple attributes at a time. A suitable result of a transformation chain is one possessing attributes compatible with the requested attributes. But a result may contain other attributes as well, that were not requested, and are irrelevant to the result.

For example: if attributes **A=a** and **B=b** are requested, and variant **V1** contains attributes **A=a**, **B=b**, and **C=c**, and variant **V2** contains attributes **A=a**, **B=b**, and **D=d**, then since all the values of **A** and **B** are identical (or compatible) either **V1** or **V2** would satisfy the request.

A Full Example

Let's continue exploring the **minified** example begun above: a configuration requests **org.gradle.usage=java-runtime**, **org.gradle.libraryelements=jar**, **minified=true**. The dependencies are:

- External **guava** dependency with variants:
 - **org.gradle.usage=java-runtime**, **org.gradle.libraryelements=jar**, **minified=false**
 - **org.gradle.usage=java-api**, **org.gradle.libraryelements=jar**, **minified=false**
- Project **producer** dependency with variants:
 - **org.gradle.usage=java-runtime**, **org.gradle.libraryelements=jar**, **minified=false**
 - **org.gradle.usage=java-runtime**, **org.gradle.libraryelements=classes**, **minified=false**
 - **org.gradle.usage=java-api**, **org.gradle.libraryelements=jar**, **minified=false**
 - **org.gradle.usage=java-api**, **org.gradle.libraryelements=classes**, **minified=false**

Gradle uses the **minify** transform to convert **minified=false** variants to **minified=true**.

- For **guava**, Gradle converts
 - **org.gradle.usage=java-runtime**, **org.gradle.libraryelements=jar**, **minified=false** to
 - **org.gradle.usage=java-runtime**, **org.gradle.libraryelements=jar**, **minified=true**.
- For **producer**, Gradle converts
 - **org.gradle.usage=java-runtime**, **org.gradle.libraryelements=jar**, **minified=false** to
 - **org.gradle.usage=java-runtime**, **org.gradle.libraryelements=jar**, **minified=true**.

Then, during execution:

- Gradle downloads the **guava** JAR and runs the transform to minify it.
- Gradle executes the **producer:jar** task to produce the JAR and then runs the transform to minify it.
- These tasks and transforms are executed in parallel where possible.

To set up the **minified** attribute so that the above works you must add the attribute to all JAR variants being produced, and also add it to all resolvable configurations being requested. You should also register the attribute in the attributes schema.

build.gradle.kts

```
val artifactType = Attribute.of("artifactType", String::class.java)
val minified = Attribute.of("minified", Boolean::class.javaObjectType)
```

```

dependencies {
    attributesSchema {
        attribute(minified) ①
    }
    artifactTypes.getBy_name("jar") {
        attributes.attribute(minified, false) ②
    }
}

configurations.runtimeClasspath.configure {
    attributes {
        attribute(minified, true) ③
    }
}

dependencies {
    registerTransform(Minify::class) {
        from.attribute(minified, false).attribute(artifactType, "jar")
        to.attribute(minified, true).attribute(artifactType, "jar")
    }
}

dependencies { ④
    implementation("com.google.guava:guava:27.1-jre")
    implementation(project(":producer"))
}

tasks.register<Copy>("resolveRuntimeClasspath") { ⑤
    from(configurations.runtimeClasspath)
    into(layout.buildDirectory.dir("runtimeClasspath"))
}

```

build.gradle

```

def artifactType = Attribute.of('artifactType', String)
def minified = Attribute.of('minified', Boolean)
dependencies {
    attributesSchema {
        attribute(minified) ①
    }
    artifactTypes.getBy_name("jar") {
        attributes.attribute(minified, false) ②
    }
}

configurations.runtimeClasspath {
    attributes {
        attribute(minified, true) ③
    }
}

```

```

}

dependencies {
    registerTransform(Minify) {
        from.attribute(minified, false).attribute(artifactType, "jar")
        to.attribute(minified, true).attribute(artifactType, "jar")
    }
}

dependencies {
    implementation('com.google.guava:guava:27.1-jre')
    implementation(project(':producer'))
}

tasks.register("resolveRuntimeClasspath", Copy) {
    from(configurations.runtimeClasspath)
    into(layout.buildDirectory.dir("runtimeClasspath"))
}

```

- ① Add the attribute to the schema
- ② All JAR files are not minified
- ③ Request that the runtime classpath is minified
- ④ Add the dependencies which will be transformed
- ⑤ Add task that requires the transformed artifacts

You can now see what happens when we run the `resolveRuntimeClasspath` task, which resolves the `runtimeClasspath` configuration. Gradle transforms the project dependency before the `resolveRuntimeClasspath` task starts. Gradle transforms the binary dependencies when it executes the `resolveRuntimeClasspath` task:

```

$ gradle resolveRuntimeClasspath
> Task :producer:compileJava
> Task :producer:processResources NO-SOURCE
> Task :producer:classes
> Task :producer:jar

> Transform producer.jar (project :producer) with Minify
Nothing to minify - using producer.jar unchanged

> Task :resolveRuntimeClasspath
Minifying guava-27.1-jre.jar
Nothing to minify - using listenablefuture-9999.0-empty-to-avoid-conflict-with-guava.jar unchanged
Nothing to minify - using jsr305-3.0.2.jar unchanged
Nothing to minify - using checker-qual-2.5.2.jar unchanged
Nothing to minify - using error_prone_annotations-2.2.0.jar unchanged
Nothing to minify - using j2objc-annotations-1.1.jar unchanged

```

```
Nothing to minify - using animal-sniffer-annotations-1.17.jar unchanged
Nothing to minify - using failureaccess-1.0.1.jar unchanged
```

```
BUILD SUCCESSFUL in 0s
3 actionable tasks: 3 executed
```

Implementing Artifact Transforms

Similar to task types, an artifact transform consists of an action and some optional parameters. The major difference from custom task types is that the action and the parameters are implemented as two separate classes.

Artifact Transforms without Parameters

An artifact transform action is provided by a class implementing [TransformAction](#). Such a class implements the `transform()` method, which converts the input artifacts into zero, one, or multiple output artifacts.

Most Artifact Transforms are one-to-one, so the `transform` method will be used to transform each input artifact contained in the from variant into exactly one output artifact.

The implementation of the artifact transform action needs to register each output artifact by calling [TransformOutputs.dir\(\)](#) or [TransformOutputs.file\(\)](#).

You can supply two types of paths to the `dir` or `file` methods:

- An absolute path to the input artifact or within the input artifact (for an input directory).
- A relative path.

Gradle uses the absolute path as the location of the output artifact. For example, if the input artifact is an exploded WAR, the transform action can call `TransformOutputs.file()` for all JAR files in the `WEB-INF/lib` directory. The output of the transform would then be the library JARs of the web application.

For a relative path, the `dir()` or `file()` method returns a workspace to the transform action. The transform action needs to create the transformed artifact(s) at the location of the provided workspace.

The output artifact(s) replace the input artifact(s) in the transformed variant in the order they were registered. For example, if the selected input variant contains the artifacts `lib1.jar`, `lib2.jar`, `lib3.jar`, and the transform action registers a minified output artifact `<artifact-name>-min.jar` for each input artifact, then the transformed configuration will consist of the artifacts `lib1-min.jar`, `lib2-min.jar`, and `lib3-min.jar`.

Here is the implementation of an `Unzip` transform, which unzips a JAR file into a `classes` directory. The `Unzip` transform does not require any parameters:

build.gradle.kts

```
abstract class Unzip : TransformAction<TransformParameters.None> {  
    ① @get:InputArtifact  
    ② abstract val inputArtifact: Provider<FileSystemLocation>  
  
    override  
    fun transform(outputs: TransformOutputs) {  
        val input = inputArtifact.get().asFile  
        val unzipDir = outputs.dir(input.name + "-unzipped")  
    ③ unzipTo(input, unzipDir)  
    ④ }  
  
    private fun unzipTo(zipFile: File, unzipDir: File) {  
        // implementation...  
    }  
}
```

build.gradle

```
abstract class Unzip implements TransformAction<TransformParameters.None> {  
    ① @InputArtifact  
    ② abstract Provider<FileSystemLocation> getInputArtifact()  
  
    @Override  
    void transform(TransformOutputs outputs) {  
        def input = inputArtifact.get().asFile  
        def unzipDir = outputs.dir(input.name + "-unzipped")  
    ③ unzipTo(input, unzipDir)  
    ④ }  
  
    private static void unzipTo(File zipFile, File unzipDir) {  
        // implementation...  
    }  
}
```

- ① Use `TransformParameters.None` if the transform does not use parameters
- ② Inject the input artifact
- ③ Request an output location for the unzipped files
- ④ Do the actual work of the transform

Note how the implementation uses `@InputArtifact` to inject an artifact to transform into the action class, so that it can be accessed within the `transform` method. This method requests a directory for the unzipped classes by using `TransformOutputs.dir()` and then unzips the JAR file into this directory.

Artifact Transforms with Parameters

An artifact transform may require parameters, such as a `String` for filtering or a file collection used to support the transformation of the input artifact. To pass these parameters to the transform action, you must define a new type with the desired parameters. This type must implement the marker interface `TransformParameters`.

The parameters must be represented using `managed properties` and the parameter type must be a `managed type`. You can use an interface or abstract class to declare the getters, and Gradle will generate the implementation. All getters need to have proper input annotations, as described in the `incremental build annotations` table.

Here is the implementation of a `Minify` transform that makes JARs smaller by only keeping certain classes in them. The `Minify` transform requires knowledge of the classes to keep within each JAR, which is provided as an `Map` property within its parameters:

build.gradle.kts

```
abstract class Minify : TransformAction<Minify.Parameters> { ①
    interface Parameters : TransformParameters { ②
        @get:Input
        var keepClassesByArtifact: Map<String, Set<String>>

    }

    @get:PathSensitive(PathSensitivity.NAME_ONLY)
    @get:InputArtifact
    abstract val inputArtifact: Provider<FileSystemLocation>

    override
    fun transform(outputs: TransformOutputs) {
        val fileName = inputArtifact.get().asFile.name
        for (entry in parameters.keepClassesByArtifact) { ③
            if (fileName.startsWith(entry.key)) {
                val nameWithoutExtension = fileName.substring(0,
                    fileName.length - 4)
                minify(inputArtifact.get().asFile, entry.value,
                    outputs.file("${nameWithoutExtension}-min.jar"))
            }
        }
    }
}
```

```

        return
    }
}
println("Nothing to minify - using ${fileName} unchanged")
outputs.file(inputArtifact) ④
}

private fun minify(artifact: File, keepClasses: Set<String>, jarFile:
File) {
    println("Minifying ${artifact.name}")
    // Implementation ...
}
}

```

build.gradle

```

abstract class Minify implements TransformAction<Parameters> { ①
    interface Parameters extends TransformParameters { ②
        @Input
        Map<String, Set<String>> getKeepClassesByArtifact()
        void setKeepClassesByArtifact(Map<String, Set<String>> keepClasses)
    }

    @PathSensitive(PathSensitivity.NAME_ONLY)
    @InputArtifact
    abstract Provider<FileSystemLocation> getInputArtifact()

    @Override
    void transform(TransformOutputs outputs) {
        def fileName = inputArtifact.get().asFile.name
        for (entry in parameters.keepClassesByArtifact) { ③
            if (fileName.startsWith(entry.key)) {
                def nameWithoutExtension = fileName.substring(0, fileName
.length() - 4)
                minify(inputArtifact.get().asFile, entry.value, outputs.file
("${nameWithoutExtension}-min.jar"))
                return
            }
        }
        println "Nothing to minify - using ${fileName} unchanged"
        outputs.file(inputArtifact) ④
    }

    private void minify(File artifact, Set<String> keepClasses, File jarFile)
{
        println "Minifying ${artifact.name}"
        // Implementation ...
    }
}

```

```
}
```

- ① Declare the parameter type
- ② Interface for the transform parameters
- ③ Use the parameters
- ④ Use the unchanged input artifact when no minification is required

Observe how you can obtain the parameters by `TransformAction.getParameters()` in the `transform()` method. The implementation of the `transform()` method requests a location for the minified JAR by using `TransformOutputs.file()` and then creates the minified JAR at this location.

Remember that the input artifact is a dependency, which may have its own dependencies. Suppose your artifact transform needs access to those transitive dependencies. In that case, it can declare an abstract getter returning a `FileCollection` and annotate it with `@InputArtifactDependencies`. When your transform runs, Gradle will inject the transitive dependencies into the `FileCollection` property by implementing the getter.

Note that using input artifact dependencies in a transform has performance implications; only inject them when needed.

Artifact Transforms with Caching

Artifact Transforms can make use of the `build cache` to store their outputs and avoid rerunning their transform actions when the result is known.

To enable the build cache to store the results of an artifact transform, add the `@CacheableTransform` annotation on the action class.

For cacheable transforms, you must annotate its `@InputArtifact` property — and any property marked with `@InputArtifactDependencies` — with normalization annotations such as `@PathSensitive`.

The following example demonstrates a more complex transform that relocates specific classes within a JAR to a different package. This process involves rewriting the bytecode of both the relocated classes and any classes that reference them (class relocation):

build.gradle.kts

```
@CacheableTransform
① abstract class ClassRelocator : TransformAction<ClassRelocator.Parameters> {
    interface Parameters : TransformParameters {
        ② @get:CompileClasspath
        ③ val externalClasspath: ConfigurableFileCollection
    }
}
```

```

    @get:Input
    val excludedPackage: Property<String>
}

@get:Classpath
④ @get:InputArtifact
    abstract val primaryInput: Provider<FileSystemLocation>

    @get:CompileClasspath
    @get:InputArtifactDependencies
⑤    abstract val dependencies: FileCollection

    override
    fun transform(outputs: TransformOutputs) {
        val primaryInputFile = primaryInput.get().asFile
        if (parameters.externalClasspath.contains(primaryInputFile)) {
⑥            outputs.file(primaryInput)
        } else {
            val baseName = primaryInputFile.name.substring(0,
primaryInputFile.name.length - 4)
            relocateJar(outputs.file("$baseName-relocated.jar"))
        }
    }

    private fun relocateJar(output: File) {
        // implementation...
        val relocatedPackages = (dependencies.flatMap { it.readPackages() } +
primaryInput.get().asFile.readPackages()).toSet()
        val nonRelocatedPackages = parameters.externalClasspath.flatMap {
it.readPackages() }
        val relocations = (relocatedPackages - nonRelocatedPackages).map {
packageName ->
            val toPackage = "relocated.$packageName"
            println("$packageName -> $toPackage")
            Relocation(packageName, toPackage)
        }
        JarRelocator(primaryInput.get().asFile, output, relocations).run()
    }
}

```

build.gradle

```

@CacheableTransform
① abstract class ClassRelocator implements TransformAction<Parameters> {
    interface Parameters extends TransformParameters {

```

```

②    @CompileClasspath

③    ConfigurableFileCollection getExternalClasspath()
    @Input
    Property<String> getExcludedPackage()
}

    @Classpath

④    @InputArtifact
    abstract Provider<FileSystemLocation> getPrimaryInput()

    @CompileClasspath
    @InputArtifactDependencies

⑤    abstract FileCollection getDependencies()

    @Override
    void transform(TransformOutputs outputs) {
        def primaryInputFile = primaryInput.get().asFile
        if (parameters.externalClasspath.contains(primaryInput)) {

⑥        outputs.file(primaryInput)
        } else {
            def baseName = primaryInputFile.name.substring(0,
primaryInputFile.name.length - 4)
            relocateJar(outputs.file("$baseName-relocated.jar"))
        }
    }

    private relocateJar(File output) {
        // implementation...
        def relocatedPackages = (dependencies.collectMany { readPackages(it)
} + readPackages(primaryInput.get().asFile)) as Set
        def nonRelocatedPackages = parameters.externalClasspath.collectMany {
readPackages(it) }
        def relocations = (relocatedPackages - nonRelocatedPackages).collect
{ packageName ->
            def toPackage = "relocated.$packageName"
            println("$packageName -> $toPackage")
            new Relocation(packageName, toPackage)
        }
        new JarRelocator(primaryInput.get().asFile, output, relocations).run
()
    }
}

```

- ① Declare the transform cacheable
- ② Interface for the transform parameters
- ③ Declare input type for each parameter
- ④ Declare a normalization for the input artifact
- ⑤ Inject the input artifact dependencies
- ⑥ Use the parameters

Note the classes to be relocated are determined by examining the packages of the input artifact and its dependencies. Additionally, the transform ensures that packages contained in JAR files on an external classpath are not relocated.

Incremental Artifact Transforms

Similar to [incremental tasks](#), Artifact Transforms can avoid some work by only processing files that have changed since the last execution. This is done by using the [InputChanges](#) interface.

For Artifact Transforms, only the input artifact is an incremental input; therefore, the transform can only query for changes there. To use [InputChanges](#) in the transform action, inject it into the action.

For more information on how to use [InputChanges](#), see the corresponding documentation for [incremental tasks](#).

Here is an example of an incremental transform that counts the lines of code in Java source files:

build.gradle.kts

```
abstract class CountLoc : TransformAction<TransformParameters.None> {  
  
    @get:Inject  
    abstract val inputChanges: InputChanges  
  
    @get:PathSensitive(PathSensitivity.RELATIVE)  
    @get:InputArtifact  
    abstract val input: Provider<FileSystemLocation>  
  
    override  
    fun transform(outputs: TransformOutputs) {  
        val outputDir = outputs.dir("${input.get().asFile.name}.loc")  
        println("Running transform on ${input.get().asFile.name},  
incremental: ${inputChanges.isIncremental}")  
        inputChanges.getFileChanges(input).forEach { change ->  
            val changedFile = change.file  
            if (change.fileType != FileType.FILE) {  
                return@forEach  
            }  
        }  
        val outputLocation =
```

```

outputDir.resolve("${change.normalizedPath}.loc")
    when (change.changeType) {
        ChangeType.ADDED, ChangeType.MODIFIED -> {

            println("Processing file ${changedFile.name}")
            outputLocation.parentFile.mkdirs()

        }
    }

outputLocation.writeText(changedFile.readLines().size.toString())
    }
    ChangeType.REMOVED -> {
        println("Removing leftover output file
${outputLocation.name}")
        outputLocation.delete()
    }
}
}
}
}
}
}
}
}

```

build.gradle

```

abstract class CountLoc implements TransformAction<TransformParameters.None>
{

    @Inject ①
    abstract InputChanges getInputChanges()

    @PathSensitive(PathSensitivity.RELATIVE)
    @InputArtifact
    abstract Provider<FileSystemLocation> getInput()

    @Override
    void transform(TransformOutputs outputs) {
        def outputDir = outputs.dir("${input.get().asFile.name}.loc")
        println("Running transform on ${input.get().asFile.name},
incremental: ${inputChanges.incremental}")
        inputChanges.getFileChanges(input).forEach { change -> ②
            def changedFile = change.file
            if (change.fileType != FileType.FILE) {
                return
            }
            def outputLocation = new File(outputDir, "${change.
normalizedPath}.loc")
            switch (change.changeType) {
                case ADDED:
                case MODIFIED:
                    println("Processing file ${changedFile.name}")
                    outputLocation.parentFile.mkdirs()
            }
        }
    }
}

```

```

        outputLocation.text = changedFile.readLines().size()

    case REMOVED:
        println("Removing leftover output file ${outputLocation
.name}")
        outputLocation.delete()
    }
}
}
}
}

```

① Inject `InputChanges`

② Query for changes in the input artifact

This transform will only run on source files that have changed since the last run, as otherwise the line count would not need to be recalculated.

Registering Artifact Transforms

You need to register the artifact transform actions, providing parameters if necessary so that they can be selected when resolving dependencies.

To register an artifact transform, you must use `registerTransform()` within the `dependencies {}` block.

There are a few points to consider when using `registerTransform()`:

- At least one `from` and `to` attributes are required.
- Each `to` attribute must have a corresponding `from` attribute.
- Additional `from` attributes can be included which do *not* have corresponding `to` attributes.
- The transform action itself can have configuration options. You can configure them with the `parameters {}` block.
- You must register the transform on the project that has the configuration that will be resolved.
- You can supply any type implementing `TransformAction` to the `registerTransform()` method.

For example, imagine you want to unpack some dependencies and put the unpacked directories and files on the classpath. You can do so by registering an artifact transform action of type `Unzip`, as shown here:

build.gradle.kts

```
dependencies {
```

```

registerTransform(Unzip::class.java) {
    from.attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE,
objects.named<LibraryElements>(LibraryElements.JAR))
    from.attribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE,
ArtifactTypeDefinition.JAR_TYPE)
    to.attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE,
objects.named<LibraryElements>(LibraryElements.CLASSES_AND_RESOURCES))
    to.attribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE,
ArtifactTypeDefinition.DIRECTORY_TYPE)
}
}

```

build.gradle

```

dependencies {
    registerTransform(Unzip) {
        from.attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE, objects
.named(LibraryElements, LibraryElements.JAR))
        from.attribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE,
ArtifactTypeDefinition.JAR_TYPE)
        to.attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE, objects
.named(LibraryElements, LibraryElements.CLASSES_AND_RESOURCES))
        to.attribute(ArtifactTypeDefinition.ARTIFACT_TYPE_ATTRIBUTE,
ArtifactTypeDefinition.DIRECTORY_TYPE)
    }
}

```

Another example is that you want to minify JARs by only keeping some **class** files from them. Note the use of the **parameters {}** block to provide the classes to keep in the minified JARs to the **Minify** transform:

build.gradle.kts

```

val artifactType = Attribute.of("artifactType", String::class.java)
val minified = Attribute.of("minified", Boolean::class.javaObjectType)
val keepPatterns = mapOf(
    "guava" to setOf(
        "com.google.common.base.Optional",
        "com.google.common.base.AbstractIterator"
    )
)

dependencies {
    registerTransform(Minify::class) {

```

```

        from.attribute(minified, false).attribute(artifactType, "jar")
        to.attribute(minified, true).attribute(artifactType, "jar")

        parameters {
            keepClassesByArtifact = keepPatterns
        }
    }
}

```

build.gradle

```

def artifactType = Attribute.of('artifactType', String)
def minified = Attribute.of('minified', Boolean)
def keepPatterns = [
    "guava": [
        "com.google.common.base.Optional",
        "com.google.common.base.AbstractIterator"
    ] as Set
]

dependencies {
    registerTransform(Minify) {
        from.attribute(minified, false).attribute(artifactType, "jar")
        to.attribute(minified, true).attribute(artifactType, "jar")

        parameters {
            keepClassesByArtifact = keepPatterns
        }
    }
}

```

Executing Artifact Transforms

On the command line, Gradle runs tasks; not Artifact Transforms: `./gradlew build`. So how and when does it run transforms?

There are two ways Gradle executes a transform:

1. Artifact Transforms execution for *project dependencies* can be discovered ahead of task execution and therefore can be scheduled before the task execution.
2. Artifact Transforms execution for *external module dependencies* cannot be discovered ahead of task execution and, therefore are scheduled inside the task execution.

In well-declared builds, project dependencies can be fully discovered during task configuration ahead of task execution scheduling. If the project dependency is badly declared (e.g., missing a task

input), the transform execution will happen inside the task.

It's important to remember that Artifact Transforms:

- will only ever be run if no matching variants exist to satisfy a request
- can be run in parallel
- will not be rerun if possible (if multiple resolution requests require the same transform to be executed on the same artifacts, and the transform is cacheable, the transform will only be run once and the results fetched from the cache on each subsequent request)

IMPORTANT

`TransformAction`s are only instantiated and run if input artifacts exist. If there are no artifacts present in an input variant to a transform, that transform will be skipped. This can happen in the middle of a chain of actions, resulting in all subsequent transforms being skipped.

PUBLISHING LIBRARIES

Publishing a project as module

The vast majority of software projects build something that aims to be consumed in some way. It could be a library that other software projects use or it could be an application for end users. *Publishing* is the process by which the thing being built is made available to consumers.

In Gradle, that process looks like this:

1. Define **what** to publish
2. Define **where** to publish it to
3. **Do** the publishing

Each of these steps is dependent on the type of repository to which you want to publish artifacts. The two most common types are Maven-compatible and Ivy-compatible repositories, or Maven and Ivy repositories for short.

As of Gradle 6.0, the **Gradle Module Metadata** will always be published alongside the Ivy XML or Maven POM metadata file.

Gradle makes it easy to publish to these types of repository by providing some prepackaged infrastructure in the form of the **Maven Publish Plugin** and the **Ivy Publish Plugin**. These plugins allow you to configure what to publish and perform the publishing with a minimum of effort.



Figure 17. The publishing process

Let's take a look at those steps in more detail:

What to publish

Gradle needs to know what files and information to publish so that consumers can use your project. This is typically a combination of **artifacts** and metadata that Gradle calls a **publication**. Exactly what a publication contains depends on the type of repository it's being published to.

For example, a publication destined for a Maven repository includes:

- One or more artifacts — typically built by the project,
- The Gradle Module Metadata file which will describe the variants of the published

component,

- The Maven POM file will identify the primary artifact and its dependencies. The primary artifact is typically the project's production JAR and secondary artifacts might consist of "-sources" and "-javadoc" JARs.

In addition, Gradle will publish checksums for all of the above, and [signatures](#) when configured to do so. From Gradle 6.0 onwards, this includes [SHA256](#) and [SHA512](#) checksums.

Where to publish

Gradle needs to know where to publish artifacts so that consumers can get hold of them. This is done via [repositories](#), which store and make available all sorts of artifact. Gradle also needs to interact with the repository, which is why you must provide the type of the repository and its location.

How to publish

Gradle automatically generates publishing tasks for all possible combinations of publication and repository, allowing you to publish any artifact to any repository. If you're publishing to a Maven repository, the tasks are of type [PublishToMavenRepository](#), while for Ivy repositories the tasks are of type [PublishToIvyRepository](#).

What follows is a practical example that demonstrates the entire publishing process.

Setting up basic publishing

The first step in publishing, irrespective of your project type, is to apply the appropriate publishing plugin. As mentioned in the introduction, Gradle supports both Maven and Ivy repositories via the following plugins:

- [Maven Publish Plugin](#)
- [Ivy Publish Plugin](#)

These provide the specific publication and repository classes needed to configure publishing for the corresponding repository type. Since Maven repositories are the most commonly used ones, they will be the basis for this example and for the other samples in the chapter. Don't worry, we will explain how to adjust individual samples for Ivy repositories.

Let's assume we're working with a simple Java library project, so only the following plugins are applied:

Example 22. [Applying the necessary plugins](#)

build.gradle.kts

```
plugins {  
    `java-library`  
    `maven-publish`  
}
```

build.gradle

```
plugins {  
    id 'java-library'  
    id 'maven-publish'  
}
```

Once the appropriate plugin has been applied, you can configure the publications and repositories. For this example, we want to publish the project's production JAR file — the one produced by the `jar` task — to a custom Maven repository. We do that with the following `publishing {}` block, which is backed by [PublishingExtension](#):

Example 23. Configuring a Java library for publishing

build.gradle.kts

```
group = "org.example"  
version = "1.0"  
  
publishing {  
    publications {  
        create<MavenPublication>("myLibrary") {  
            from(components["java"])  
        }  
    }  
  
    repositories {  
        maven {  
            name = "myRepo"  
            url = uri(layout.buildDirectory.dir("repo"))  
        }  
    }  
}
```

build.gradle

```
group = 'org.example'  
version = '1.0'  
  
publishing {  
    publications {  
        myLibrary(MavenPublication) {  
            from components.java  
        }  
    }  
}
```

```

repositories {
    maven {
        name = 'myRepo'
        url = layout.buildDirectory.dir("repo")
    }
}

```

This defines a publication called "myLibrary" that can be published to a Maven repository by virtue of its type: [MavenPublication](#). This publication consists of just the production JAR artifact and its metadata, which combined are represented by the [java component](#) of the project.

NOTE

Components are the standard way of defining a publication. They are provided by plugins, usually of the language or platform variety. For example, the Java Plugin defines the `components.java` [SoftwareComponent](#), while the War Plugin defines `components.web`.

The example also defines a file-based Maven repository with the name "myRepo". Such a file-based repository is convenient for a sample, but real-world builds typically work with HTTPS-based repository servers, such as Maven Central or an internal company server.

NOTE

You may define one, and only one, repository without a name. This translates to an implicit name of "Maven" for Maven repositories and "Ivy" for Ivy repositories. All other repository definitions must be given an explicit name.

In combination with the project's `group` and `version`, the publication and repository definitions provide everything that Gradle needs to publish the project's production JAR. Gradle will then create a dedicated `publishMyLibraryPublicationToMyRepoRepository` task that does just that. Its name is based on the template `publishPubNamePublicationToRepoNameRepository`. See the appropriate publishing plugin's documentation for more details on the nature of this task and any other tasks that may be available to you.

You can either execute the individual publishing tasks directly, or you can execute `publish`, which will run all the available publishing tasks. In this example, `publish` will just run `publishMyLibraryPublicationToMavenRepository`.

NOTE

Basic publishing to an Ivy repository is very similar: you simply use the Ivy Publish Plugin, replace `MavenPublication` with `IvyPublication`, and use `ivy` instead of `maven` in the repository definition.

There are differences between the two types of repository, particularly around the extra metadata that each support — for example, Maven repositories require a POM file while Ivy ones have their own metadata format — so see the plugin chapters for comprehensive information on how to configure both publications and repositories for whichever repository type you're working with.

That's everything for the basic use case. However, many projects need more control over what gets published, so we look at several common scenarios in the following sections.

Suppressing validation errors

Gradle performs validation of generated module metadata. In some cases, validation can fail, indicating that you most likely have an error to fix, but you may have done something intentionally. If this is the case, Gradle will indicate the name of the validation error you can disable on the `GenerateModuleMetadata` tasks:

Example 24. [Disabling some validation errors](#)

build.gradle.kts

```
tasks.withType<GenerateModuleMetadata> {  
    // The value 'enforced-platform' is provided in the validation  
    // error message you got  
    suppressedValidationErrors.add("enforced-platform")  
}
```

build.gradle

```
tasks.withType(GenerateModuleMetadata).configureEach {  
    // The value 'enforced-platform' is provided in the validation  
    // error message you got  
    suppressedValidationErrors.add('enforced-platform')  
}
```

Understanding Gradle Module Metadata

Gradle Module Metadata is a format used to serialize the Gradle component model. It is similar to [Apache Maven™'s POM file](#) or [Apache Ivy™ ivy.xml](#) files. The goal of metadata files is to provide *to consumers* a reasonable model of what is published on a repository.

Gradle Module Metadata is a unique format aimed at improving dependency resolution by making it multi-platform and variant-aware.

In particular, Gradle Module Metadata supports:

- [rich version constraints](#)
- [dependency constraints](#)
- [component capabilities](#)
- [variant-aware resolution](#)

Publication of Gradle Module Metadata will enable better dependency management for your consumers:

- early discovery of problems by detecting [incompatible modules](#)
- consistent selection of [platform-specific dependencies](#)
- native [dependency version alignment](#)
- automatically getting dependencies for specific [features of your library](#)

Gradle Module Metadata is automatically published when using the [Maven Publish plugin](#) or the [Ivy Publish plugin](#).

The specification for Gradle Module Metadata can be found [here](#).

Mapping with other formats

Gradle Module Metadata is automatically published on Maven or Ivy repositories. However, it doesn't replace the *pom.xml* or *ivy.xml* files: it is published alongside those files. This is done to maximize compatibility with third-party build tools.

Gradle does its best to map Gradle-specific concepts to Maven or Ivy. When a build file uses features that can only be represented in Gradle Module Metadata, Gradle will warn you at publication time. The table below summarizes how some Gradle specific features are mapped to Maven and Ivy:

Table 8. Mapping of Gradle specific concepts to Maven and Ivy

Gradle	Maven	Ivy	Description
dependency constraints	<code><dependencyManagement></code> dependencies	Not published	Gradle dependency constraints are <i>transitive</i> , while Maven's dependency management block <i>isn't</i>
rich version constraints	Publishes the <i>requires</i> version	Published the <i>requires</i> version	
component capabilities	Not published	Not published	Component capabilities are unique to Gradle
Feature variants	Variant artifacts are uploaded, dependencies are published as <i>optional</i> dependencies	Variant artifacts are uploaded, dependencies are not published	Feature variants are a good replacement for optional dependencies

Gradle	Maven	Ivy	Description
Custom component types	Artifacts are uploaded, dependencies are those described by the mapping	Artifacts are uploaded, dependencies are ignored	Custom component types are probably not consumable from Maven or Ivy in any case. They usually exist in the context of a custom ecosystem.

Disabling metadata compatibility publication warnings

If you want to suppress warnings, you can use the following APIs to do so:

- For Maven, see the `suppress*` methods in [MavenPublication](#)
- For Ivy, see the `suppress*` methods in [IvyPublication](#)

Example 25. [Disabling publication warnings](#)

build.gradle.kts

```
publications {
    register<MavenPublication>("maven") {
        from(components["java"])
        suppressPomMetadataWarningsFor("runtimeElements")
    }
}
```

build.gradle

```
publications {
    maven(MavenPublication) {
        from components.java
        suppressPomMetadataWarningsFor('runtimeElements')
    }
}
```

Interactions with other build tools

Because Gradle Module Metadata is not widely spread and because it aims at [maximizing compatibility with other tools](#), Gradle does a couple of things:

- Gradle Module Metadata is systematically published alongside the normal descriptor for a given repository (Maven or Ivy)

- the `pom.xml` or `ivy.xml` file will contain a *marker comment* which tells Gradle that Gradle Module Metadata exists for this module

The goal of the marker is *not* for other tools to parse module metadata: it's for Gradle users only. It explains to Gradle that a *better* module metadata file exists and that it should use it instead. It doesn't mean that consumption from Maven or Ivy would be broken either, only that it works in [degraded mode](#).

NOTE

This must be seen as a *performance optimization*: instead of having to do 2 network requests, one to get Gradle Module Metadata, then one to get the POM/Ivy file in case of a miss, Gradle will first look at the file which is most likely to be present, then only perform a 2nd request if the module was actually published with Gradle Module Metadata.

If you know that the modules you depend on are always published with Gradle Module Metadata, you can optimize the network calls by configuring the metadata sources for a repository:

Example 26. [Resolving Gradle Module Metadata only](#)

build.gradle.kts

```
repositories {
    maven {
        url = uri("http://repo.mycompany.com/repo")
        metadataSources {
            gradleMetadata()
        }
    }
}
```

build.gradle

```
repositories {
    maven {
        url = "http://repo.mycompany.com/repo"
        metadataSources {
            gradleMetadata()
        }
    }
}
```

Gradle Module Metadata validation

Gradle Module Metadata is validated before being published.

The following rules are enforced:

- Variant names must be unique,
- Each variant must have at least [one attribute](#),
- Two variants cannot have the [exact same attributes and capabilities](#),
- If there are dependencies, at least one, across all variants, must carry [version information](#).

These rules ensure the quality of the metadata produced, and help confirm that consumption will not be problematic.

Gradle Module Metadata reproducibility

The task generating the module metadata files is currently never marked **UP-TO-DATE** by Gradle due to the way it is implemented. However, if neither build inputs nor build scripts changed, the task result is effectively up-to-date: it always produces the same output.

If users desire to have a unique **module** file per build invocation, it is possible to link an identifier in the produced metadata to the build that created it. Users can choose to enable this unique identifier in their **publication**:

Example 27. [Configuring the build identifier of a publication](#)

build.gradle.kts

```
publishing {
    publications {
        create<MavenPublication>("myLibrary") {
            from(components["java"])
            withBuildIdentifier()
        }
    }
}
```

build.gradle

```
publishing {
    publications {
        myLibrary(MavenPublication) {
            from components.java
            withBuildIdentifier()
        }
    }
}
```

With the changes above, the generated Gradle Module Metadata file will always be different, forcing downstream tasks to consider it out-of-date.

Disabling Gradle Module Metadata publication

There are situations where you might want to disable publication of Gradle Module Metadata:

- the repository you are uploading to rejects the metadata file (unknown format)
- you are using Maven or Ivy specific concepts which are not properly mapped to Gradle Module Metadata

In this case, disabling the publication of Gradle Module Metadata is done simply by disabling the task which generates the metadata file:

Example 28. [Disabling publication of Gradle Module Metadata](#)

build.gradle.kts

```
tasks.withType<GenerateModuleMetadata> {  
    enabled = false  
}
```

build.gradle

```
tasks.withType(GenerateModuleMetadata) {  
    enabled = false  
}
```

Signing artifacts

The [Signing Plugin](#) can be used to sign all artifacts and metadata files that make up a publication, including Maven POM files and Ivy module descriptors. In order to use it:

1. Apply the Signing Plugin
2. Configure the [signatory credentials](#) — follow the link to see how
3. Specify the publications you want signed

Here's an example that configures the plugin to sign the `mavenJava` publication:

Example 29. *Signing a publication*

build.gradle.kts

```
signing {  
    sign(publishing.publications["mavenJava"])  
}
```

build.gradle

```
signing {  
    sign publishing.publications.mavenJava  
}
```

This will create a **Sign** task for each publication you specify and wire all **publish** **PubNamePublicationToRepoNameRepository** tasks to depend on it. Thus, publishing any publication will automatically create and publish the signatures for its artifacts and metadata, as you can see from this output:

Example: Sign and publish a project

Output of **gradle publish**

```
> gradle publish  
> Task :compileJava  
> Task :processResources  
> Task :classes  
> Task :jar  
> Task :javadoc  
> Task :javadocJar  
> Task :sourcesJar  
> Task :generateMetadataFileForMavenJavaPublication  
> Task :generatePomFileForMavenJavaPublication  
> Task :signMavenJavaPublication  
> Task :publishMavenJavaPublicationToMavenRepository  
> Task :publish
```

```
BUILD SUCCESSFUL in 0s  
10 actionable tasks: 10 executed
```

Customizing publishing

Modifying and adding variants to existing components for publishing

Gradle's publication model is based on the notion of *components*, which are defined by plugins. For example, the Java Library plugin defines a `java` component which corresponds to a library, but the Java Platform plugin defines another kind of component, named `javaPlatform`, which is effectively a different kind of software component (a *platform*).

Sometimes we want to add *more variants* to or modify *existing variants* of an existing component. For example, if you [added a variant of a Java library for a different platform](#), you may just want to declare this additional variant on the `java` component itself. In general, declaring additional variants is often the best solution to publish *additional artifacts*.

To perform such additions or modifications, the `AdhocComponentWithVariants` interface declares two methods called `addVariantsFromConfiguration` and `withVariantsFromConfiguration` which accept two parameters:

- the [outgoing configuration](#) that is used as a variant source
- a customization action which allows you to *filter* which variants are going to be published

To utilise these methods, you must make sure that the `SoftwareComponent` you work with is itself an `AdhocComponentWithVariants`, which is the case for the components created by the Java plugins (Java, Java Library, Java Platform). Adding a variant is then very simple:

Example 30. [Adding a variant to an existing software component](#)

InstrumentedJarsPlugin.kt

```
val javaComponent = components.findByName("java") as
AdhocComponentWithVariants
javaComponent.addVariantsFromConfiguration(outgoing) {
    // dependencies for this variant are considered runtime dependencies
    mapToMavenScope("runtime")
    // and also optional dependencies, because we don't want them to leak
    mapToOptional()
}
```

InstrumentedJarsPlugin.groovy

```
AdhocComponentWithVariants javaComponent = (AdhocComponentWithVariants)
project.components.findByName("java")
javaComponent.addVariantsFromConfiguration(outgoing) {
    // dependencies for this variant are considered runtime dependencies
    it.mapToMavenScope("runtime")
    // and also optional dependencies, because we don't want them to leak
    it.mapToOptional()
}
```

In other cases, you might want to modify a variant that was added by one of the Java plugins already. For example, if you activate publishing of Javadoc and sources, these become additional variants of the `java` component. If you only want to publish one of them, e.g. only Javadoc but no sources, you can modify the `sources` variant to not being published:

Example 31. Publish a java library with Javadoc but without sources

build.gradle.kts

```
java {
    withJavadocJar()
    withSourcesJar()
}

val javaComponent = components["java"] as AdhocComponentWithVariants
javaComponent.withVariantsFromConfiguration(configurations["sourcesElements"]
) {
    skip()
}

publishing {
    publications {
        create<MavenPublication>("mavenJava") {
            from(components["java"])
        }
    }
}
```

build.gradle

```
java {
    withJavadocJar()
    withSourcesJar()
}

components.java.withVariantsFromConfiguration(configurations.sourcesElements)
{
    skip()
}

publishing {
    publications {
        mavenJava(MavenPublication) {
            from components.java
        }
    }
}
```

Creating and publishing custom components

In the [previous example](#), we have demonstrated how to extend or modify an existing component, like the components provided by the Java plugins. But Gradle also allows you to build a custom component (not a Java Library, not a Java Platform, not something supported natively by Gradle).

To create a custom component, you first need to create an empty *adhoc* component. At the moment, this is only possible via a plugin because you need to get a handle on the [SoftwareComponentFactory](#):

Example 32. Injecting the software component factory

InstrumentedJarsPlugin.kt

```
class InstrumentedJarsPlugin @Inject constructor(  
    private val softwareComponentFactory: SoftwareComponentFactory) :  
    Plugin<Project> {
```

InstrumentedJarsPlugin.groovy

```
private final SoftwareComponentFactory softwareComponentFactory  
  
@Inject  
InstrumentedJarsPlugin(SoftwareComponentFactory softwareComponentFactory) {  
    this.softwareComponentFactory = softwareComponentFactory  
}
```

Declaring *what* a custom component publishes is still done via the [AdhocComponentWithVariants](#) API. For a custom component, the first step is to create custom outgoing variants, following the instructions in [this chapter](#). At this stage, what you should have is variants which can be used in cross-project dependencies, but that we are now going to publish to external repositories.

Example 33. Creating a custom, adhoc component

InstrumentedJarsPlugin.kt

```
// create an adhoc component  
val adhocComponent = softwareComponentFactory.adhoc("myAdhocComponent")  
// add it to the list of components that this project declares  
components.add(adhocComponent)  
// and register a variant for publication  
adhocComponent.addVariantsFromConfiguration(outgoing) {  
    mapToMavenScope("runtime")  
}
```

InstrumentedJarsPlugin.groovy

```
// create an adhoc component
def adhocComponent = softwareComponentFactory.adhoc("myAdhocComponent")
// add it to the list of components that this project declares
project.components.add(adhocComponent)
// and register a variant for publication
adhocComponent.addVariantsFromConfiguration(outgoing) {
    it.mapToMavenScope("runtime")
}
```

First we use the factory to create a new adhoc component. Then we add a variant through the `addVariantsFromConfiguration` method, which is described in more detail in the [previous section](#).

In simple cases, there's a one-to-one mapping between a `Configuration` and a variant, in which case you can publish all variants issued from a single `Configuration` because they are effectively the same thing. However, there are cases where a `Configuration` is associated with additional [configuration publications](#) that we also call *secondary variants*. Such configurations make sense in a multi-project build, but not when publishing externally. This is for example the case when between projects you share a *directory of files*, but there's no way you can publish a *directory* directly on a Maven repository (only packaged things like jars or zips). Look at the [ConfigurationVariantDetails](#) class for details about how to skip publication of a particular variant. If `addVariantsFromConfiguration` has already been called for a configuration, further modification of the resulting variants can be performed using `withVariantsFromConfiguration`.

When publishing an adhoc component like this:

- Gradle Module Metadata will *exactly* represent the published variants. In particular, all outgoing variants will inherit dependencies, artifacts and attributes of the published configuration.
- Maven and Ivy metadata files will be generated, but you need to declare how the dependencies are mapped to Maven scopes via the [ConfigurationVariantDetails](#) class.

In practice, it means that components created this way can be consumed by Gradle the same way as if they were "local components".

Adding custom artifacts to a publication

Instead of thinking in terms of artifacts, you should embrace the variant aware model of Gradle. It is expected that a single module may need multiple artifacts. However this rarely stops there, if the additional artifacts represent an [optional feature](#), they might also have different dependencies and more.

Gradle, via *Gradle Module Metadata*, supports the publication of *additional variants* which make those artifacts known to the dependency resolution engine. Please refer to the [variant-aware sharing](#) section of the documentation to see how to declare such variants and [check out how to](#)

[publish custom components](#).

If you attach extra artifacts to a publication directly, they are published "out of context". That means, they are not referenced in the metadata at all and can then only be addressed directly through a classifier on a dependency. In contrast to Gradle Module Metadata, Maven pom metadata will not contain information on additional artifacts regardless of whether they are added through a variant or directly, as variants cannot be represented in the pom format.

The following section describes how you publish artifacts directly if you are sure that metadata, for example Gradle or POM metadata, is irrelevant for your use case. For example, if your project doesn't need to be consumed by other projects and the only thing required as result of the publishing are the artifacts themselves.

In general, there are two options:

- Create a publication only with artifacts
- Add artifacts to a publication based on a component with metadata (not recommended, instead [adjust a component](#) or use a [adhoc component publication](#) which will both also produce metadata fitting your artifacts)

To create a publication based on artifacts, start by defining a custom artifact and attaching it to a Gradle [configuration](#) of your choice. The following sample defines an RPM artifact that is produced by an `rpm` task (not shown) and attaches that artifact to the `conf` configuration:

Example 34. [Defining a custom artifact for a configuration](#)

build.gradle.kts

```
configurations {
    create("conf")
}
val rpmFile = layout.buildDirectory.file("rpms/my-package.rpm")
val rpmArtifact = artifacts.add("conf", rpmFile.get().asFile) {
    type = "rpm"
    builtBy("rpm")
}
```

build.gradle

```
configurations {
    conf
}
def rpmFile = layout.buildDirectory.file('rpms/my-package.rpm')
def rpmArtifact = artifacts.add('conf', rpmFile.get().asFile) {
    type = 'rpm'
    builtBy 'rpm'
}
```

The `artifacts.add()` method —from [ArtifactHandler](#) — returns an artifact object of type [PublishArtifact](#) that can then be used in defining a publication, as shown in the following sample:

Example 35. Attaching a custom [PublishArtifact](#) to a publication

build.gradle.kts

```
publishing {
    publications {
        create<MavenPublication>("maven") {
            artifact(rpmArtifact)
        }
    }
}
```

build.gradle

```
publishing {
    publications {
        maven(MavenPublication) {
            artifact rpmArtifact
        }
    }
}
```

- The `artifact()` method accepts *publish artifacts* as argument — like `rpmArtifact` in the sample — as well as any type of argument accepted by [Project.file\(java.lang.Object\)](#), such as a `File` instance, a string file path or a archive task.
- Publishing plugins support different artifact configuration properties, so always check the plugin documentation for more details. The `classifier` and `extension` properties are supported by both the [Maven Publish Plugin](#) and the [Ivy Publish Plugin](#).
- Custom artifacts need to be distinct within a publication, typically via a unique combination of `classifier` and `extension`. See the documentation for the plugin you're using for the precise requirements.
- If you use `artifact()` with an archive task, Gradle automatically populates the artifact's metadata with the `classifier` and `extension` properties from that task.

Now you can publish the RPM.

If you really want to add an artifact to a publication based on a component, instead of [adjusting the component](#) itself, you can combine the `from components.someComponent` and `artifact someArtifact` notations.

Restricting publications to specific repositories

When you have defined multiple publications or repositories, you often want to control which publications are published to which repositories. For instance, consider the following sample that defines two publications — one that consists of just a binary and another that contains the binary and associated sources — and two repositories — one for internal use and one for external consumers:

Example 36. [Adding multiple publications and repositories](#)

build.gradle.kts

```
publishing {
    publications {
        create<MavenPublication>("binary") {
            from(components["java"])
        }
        create<MavenPublication>("binaryAndSources") {
            from(components["java"])
            artifact(tasks["sourcesJar"])
        }
    }
    repositories {
        // change URLs to point to your repos, e.g. http://my.org/repo
        maven {
            name = "external"
            url = uri(layout.buildDirectory.dir("repos/external"))
        }
        maven {
            name = "internal"
            url = uri(layout.buildDirectory.dir("repos/internal"))
        }
    }
}
```

build.gradle

```
publishing {
    publications {
        binary(MavenPublication) {
            from components.java
        }
        binaryAndSources(MavenPublication) {
            from components.java
            artifact sourcesJar
        }
    }
    repositories {
```

```
// change URLs to point to your repos, e.g. http://my.org/repo
maven {
    name = 'external'
    url = layout.buildDirectory.dir('repos/external')
}
maven {
    name = 'internal'
    url = layout.buildDirectory.dir('repos/internal')
}
}
```

The publishing plugins will create tasks that allow you to publish either of the publications to either repository. They also attach those tasks to the `publish` aggregate task. But let's say you want to restrict the binary-only publication to the external repository and the binary-with-sources publication to the internal one. To do that, you need to make the publishing *conditional*.

Gradle allows you to skip any task you want based on a condition via the [Task.onlyIf\(String, org.gradle.api.specs.Spec\)](#) method. The following sample demonstrates how to implement the constraints we just mentioned:

Example 37. Configuring which artifacts should be published to which repositories

build.gradle.kts

```
tasks.withType<PublishToMavenRepository>().configureEach {
    val predicate = provider {
        (repository == publishing.repositories["external"] &&
            publication == publishing.publications["binary"]) ||
        (repository == publishing.repositories["internal"] &&
            publication == publishing.publications["binaryAndSources"])
    }
    onlyIf("publishing binary to the external repository, or binary and
sources to the internal one") {
        predicate.get()
    }
}
tasks.withType<PublishToMavenLocal>().configureEach {
    val predicate = provider {
        publication == publishing.publications["binaryAndSources"]
    }
    onlyIf("publishing binary and sources") {
        predicate.get()
    }
}
```

build.gradle

```
tasks.withType(PublishToMavenRepository) {
    def predicate = provider {
        (repository == publishing.repositories.external &&
            publication == publishing.publications.binary) ||
        (repository == publishing.repositories.internal &&
            publication == publishing.publications.binaryAndSources)
    }
    onlyIf("publishing binary to the external repository, or binary and
sources to the internal one") {
        predicate.get()
    }
}
tasks.withType(PublishToMavenLocal) {
    def predicate = provider {
        publication == publishing.publications.binaryAndSources
    }
    onlyIf("publishing binary and sources") {
        predicate.get()
    }
}
```

Output of gradle publish

```
> gradle publish
> Task :compileJava
> Task :processResources
> Task :classes
> Task :jar
> Task :generateMetadataFileForBinaryAndSourcesPublication
> Task :generatePomFileForBinaryAndSourcesPublication
> Task :sourcesJar
> Task :publishBinaryAndSourcesPublicationToExternalRepository SKIPPED
> Task :publishBinaryAndSourcesPublicationToInternalRepository
> Task :generateMetadataFileForBinaryPublication
> Task :generatePomFileForBinaryPublication
> Task :publishBinaryPublicationToExternalRepository
> Task :publishBinaryPublicationToInternalRepository SKIPPED
> Task :publish
```

```
BUILD SUCCESSFUL in 0s
10 actionable tasks: 10 executed
```

You may also want to define your own aggregate tasks to help with your workflow. For example, imagine that you have several publications that should be published to the external repository. It could be very useful to publish all of them in one go without publishing the internal ones.

The following sample demonstrates how you can do this by defining an aggregate task — `publishToExternalRepository` — that depends on all the relevant publish tasks:

Example 38. Defining your own shorthand tasks for publishing

build.gradle.kts

```
tasks.register("publishToExternalRepository") {
    group = "publishing"
    description = "Publishes all Maven publications to the external Maven
repository."
    dependsOn(tasks.withType<PublishToMavenRepository>().matching {
        it.repository == publishing.repositories["external"]
    })
}
```

build.gradle

```
tasks.register('publishToExternalRepository') {
    group = 'publishing'
    description = 'Publishes all Maven publications to the external Maven
repository.'
    dependsOn tasks.withType(PublishToMavenRepository).matching {
        it.repository == publishing.repositories.external
    }
}
```

This particular sample automatically handles the introduction or removal of the relevant publishing tasks by using `TaskCollection.withType(java.lang.Class)` with the `PublishToMavenRepository` task type. You can do the same with `PublishToIvyRepository` if you're publishing to Ivy-compatible repositories.

Configuring publishing tasks

The publishing plugins create their non-aggregate tasks after the project has been evaluated, which means you cannot directly reference them from your build script. If you would like to configure any of these tasks, you should use deferred task configuration. This can be done in a number of ways via the project's `tasks` collection.

For example, imagine you want to change where the `generatePomFileForPubNamePublication` tasks write their POM files. You can do this by using the `TaskCollection.withType(java.lang.Class)` method, as demonstrated by this sample:

Example 39. *Configuring a dynamically named task created by the publishing plugins*

build.gradle.kts

```
tasks.withType<GenerateMavenPom>().configureEach {  
    val matcher =  
    Regex("""generatePomFileFor(\w+)Publication""").matchEntire(name)  
    val publicationName = matcher?.let { it.groupValues[1] }  
    destination = layout.buildDirectory.file("poms/${publicationName}-  
pom.xml").get().asFile  
}
```

build.gradle

```
tasks.withType(GenerateMavenPom).all {  
    def matcher = name =~ /generatePomFileFor(\w+)Publication/  
    def publicationName = matcher[0][1]  
    destination = layout.buildDirectory.file("poms/${publicationName}-  
pom.xml").get().asFile  
}
```

The above sample uses a regular expression to extract the name of the publication from the name of the task. This is so that there is no conflict between the file paths of all the POM files that might be generated. If you only have one publication, then you don't have to worry about such conflicts since there will only be one POM file.

The Maven Publish Plugin

The Maven Publish Plugin provides the ability to publish build artifacts to an [Apache Maven](#) repository. A module published to a Maven repository can be consumed by Maven, Gradle (see [Declaring Dependencies](#)) and other tools that understand the Maven repository format. You can learn about the fundamentals of publishing in [Publishing Overview](#).

Usage

To use the Maven Publish Plugin, include the following in your build script:

Example 40. *Applying the Maven Publish Plugin*

build.gradle.kts

```
plugins {  
    'maven-publish'
```

```
}
```

build.gradle

```
plugins {  
    id 'maven-publish'  
}
```

The Maven Publish Plugin uses an extension on the project named `publishing` of type `PublishingExtension`. This extension provides a container of named publications and a container of named repositories. The Maven Publish Plugin works with `MavenPublication` publications and `MavenArtifactRepository` repositories.

Tasks

`generatePomFileForPubNamePublication` — `GenerateMavenPom`

Creates a POM file for the publication named *PubName*, populating the known metadata such as project name, project version, and the dependencies. The default location for the POM file is `build/publications/$pubName/pom-default.xml`.

`publishPubNamePublicationToRepoNameRepository` — `PublishToMavenRepository`

Publishes the *PubName* publication to the repository named *RepoName*. If you have a repository definition without an explicit name, *RepoName* will be "Maven".

`publishPubNamePublicationToMavenLocal` — `PublishToMavenLocal`

Copies the *PubName* publication to the local Maven cache — typically `<home directory of the current user>/.m2/repository` — along with the publication's POM file and other metadata.

`publish`

Depends on: All `publishPubNamePublicationToRepoNameRepository` tasks

An aggregate task that publishes all defined publications to all defined repositories. It does *not* include copying publications to the local Maven cache.

`publishToMavenLocal`

Depends on: All `publishPubNamePublicationToMavenLocal` tasks

Copies all defined publications to the local Maven cache, including their metadata (POM files, etc.).

Publications

This plugin provides `publications` of type `MavenPublication`. To learn how to define and use publications, see the section on `basic publishing`.

There are four main things you can configure in a Maven publication:

- A **component** — via [MavenPublication.from\(org.gradle.api.component.SoftwareComponent\)](#).
- **Custom artifacts** — via the [MavenPublication.artifact\(java.lang.Object\)](#) method. See [MavenArtifact](#) for the available configuration options for custom Maven artifacts.
- Standard metadata like **artifactId**, **groupId** and **version**.
- Other contents of the POM file — via [MavenPublication.pom\(org.gradle.api.Action\)](#).

You can see all of these in action in the [complete publishing example](#). The API documentation for [MavenPublication](#) has additional code samples.

Identity values in the generated POM

The attributes of the generated POM file will contain identity values derived from the following project properties:

- **groupId** - [Project.getGroup\(\)](#)
- **artifactId** - [Project.getName\(\)](#)
- **version** - [Project.getVersion\(\)](#)

Overriding the default identity values is easy: simply specify the **groupId**, **artifactId** or **version** attributes when configuring the [MavenPublication](#).

Example 41. Customizing the publication identity

build.gradle.kts

```
publishing {
    publications {
        create<MavenPublication>("maven") {
            groupId = "org.gradle.sample"
            artifactId = "library"
            version = "1.1"

            from(components["java"])
        }
    }
}
```

build.gradle

```
publishing {
    publications {
        maven(MavenPublication) {
            groupId = 'org.gradle.sample'
            artifactId = 'library'
        }
    }
}
```

```

        version = '1.1'

        from components.java
    }
}
}

```

TIP

Certain repositories will not be able to handle all supported characters. For example, the `:` character cannot be used as an identifier when publishing to a filesystem-backed repository on Windows.

Maven restricts `groupId` and `artifactId` to a limited character set (`[A-Za-z0-9_\\-\\.]+`) and Gradle enforces this restriction. For `version` (as well as the artifact `extension` and `classifier` properties), Gradle will handle any valid Unicode character.

The only Unicode values that are explicitly prohibited are `\`, `/` and any ISO control character. Supplied values are validated early in publication.

Customizing the generated POM

The generated POM file can be customized before publishing. For example, when publishing a library to Maven Central you will need to set certain metadata. The Maven Publish Plugin provides a DSL for that purpose. Please see [MavenPom](#) in the DSL Reference for the complete documentation of available properties and methods. The following sample shows how to use the most common ones:

Example 42. Customizing the POM file

build.gradle.kts

```

publishing {
    publications {
        create<MavenPublication>("mavenJava") {
            pom {
                name = "My Library"
                description = "A concise description of my library"
                url = "http://www.example.com/library"
                properties = mapOf(
                    "myProp" to "value",
                    "prop.with.dots" to "anotherValue"
                )
                licenses {
                    license {
                        name = "The Apache License, Version 2.0"
                        url = "http://www.apache.org/licenses/LICENSE-
2.0.txt"
                    }
                }
            }
        }
    }
}

```

```
    }  
    developers {  
        developer {  
            id = "johnd"  
            name = "John Doe"  
            email = "john.doe@example.com"  
        }  
    }  
    scm {  
        connection = "scm:git:git://example.com/my-library.git"  
        developerConnection = "scm:git:ssh://example.com/my-  
library.git"  
        url = "http://example.com/my-library/"  
    }  
}  
}  
}
```

build.gradle

```
publishing {
    publications {
        mavenJava(MavenPublication) {
            pom {
                name = 'My Library'
                description = 'A concise description of my library'
                url = 'http://www.example.com/library'
                properties = [
                    myProp: "value",
                    "prop.with.dots": "anotherValue"
                ]
                licenses {
                    license {
                        name = 'The Apache License, Version 2.0'
                        url = 'http://www.apache.org/licenses/LICENSE-
2.0.txt'
                    }
                }
                developers {
                    developer {
                        id = 'johnd'
                        name = 'John Doe'
                        email = 'john.doe@example.com'
                    }
                }
                scm {
                    connection = 'scm:git:git://example.com/my-library.git'
                    developerConnection = 'scm:git:ssh://example.com/my-
```

```
library.git'
        url = 'http://example.com/my-library/'
    }
}
}
```

Customizing dependencies versions

Two strategies are supported for publishing dependencies:

Declared versions (default)

This strategy publishes the versions that are defined by the build script author with the dependency declarations in the `dependencies` block. Any other kind of processing, for example through [a rule changing the resolved version](#), will not be taken into account for the publication.

Resolved versions

This strategy publishes the versions that were resolved during the build, possibly by applying resolution rules and automatic conflict resolution. This has the advantage that the published versions correspond to the ones the published artifact was tested against.

Example use cases for resolved versions:

- A project uses dynamic versions for dependencies but prefers exposing the resolved version for a given release to its consumers.
- In combination with [dependency locking](#), you want to publish the locked versions.
- A project leverages the rich versions constraints of Gradle, which have a lossy conversion to Maven. Instead of relying on the conversion, it publishes the resolved versions.

This is done by using the `versionMapping` DSL method which allows to configure the `VersionMappingStrategy`:

Example 43. Using resolved versions

build.gradle.kts

```
publishing {
    publications {
        create<MavenPublication>("mavenJava") {
            versionMapping {
                usage("java-api") {
                    fromResolutionOf("runtimeClasspath")
                }
                usage("java-runtime") {
                    fromResolutionResult()
                }
            }
        }
    }
}
```

```

    }
  }
}

```

build.gradle

```

publishing {
    publications {
        mavenJava(MavenPublication) {
            versionMapping {
                usage('java-api') {
                    fromResolutionOf('runtimeClasspath')
                }
                usage('java-runtime') {
                    fromResolutionResult()
                }
            }
        }
    }
}

```

In the example above, Gradle will use the versions resolved on the `runtimeClasspath` for dependencies declared in `api`, which are mapped to the `compile` scope of Maven. Gradle will also use the versions resolved on the `runtimeClasspath` for dependencies declared in `implementation`, which are mapped to the `runtime` scope of Maven. `fromResolutionResult()` indicates that Gradle should use the default classpath of a variant and `runtimeClasspath` is the default classpath of `java-runtime`.

Repositories

This plugin provides [repositories](#) of type `MavenArtifactRepository`. To learn how to define and use repositories for publishing, see the section on [basic publishing](#).

Here's a simple example of defining a publishing repository:

Example 44. [Declaring repositories to publish to](#)

build.gradle.kts

```

publishing {
    repositories {
        maven {
            // change to point to your repo, e.g. http://my.org/repo
            url = uri(layout.buildDirectory.dir("repo"))
        }
    }
}

```

```
}  
}
```

build.gradle

```
publishing {  
    repositories {  
        maven {  
            // change to point to your repo, e.g. http://my.org/repo  
            url = layout.buildDirectory.dir('repo')  
        }  
    }  
}
```

The two main things you will want to configure are the repository's:

- URL (required)
- Name (optional)

You can define multiple repositories as long as they have unique names within the build script. You may also declare one (and only one) repository without a name. That repository will take on an implicit name of "Maven".

You can also configure any authentication details that are required to connect to the repository. See [MavenArtifactRepository](#) for more details.

Snapshot and release repositories

It is a common practice to publish snapshots and releases to different Maven repositories. A simple way to accomplish this is to configure the repository URL based on the project version. The following sample uses one URL for versions that end with "SNAPSHOT" and a different URL for the rest:

Example 45. [Configuring repository URL based on project version](#)

build.gradle.kts

```
publishing {  
    repositories {  
        maven {  
            val releasesRepoUrl = layout.buildDirectory.dir("repos/releases")  
            val snapshotsRepoUrl =  
                layout.buildDirectory.dir("repos/snapshots")  
            url = uri(if (version.toString().endsWith("SNAPSHOT"))  
                snapshotsRepoUrl else releasesRepoUrl)  
        }  
    }  
}
```

```

    }
  }
}

```

build.gradle

```

publishing {
    repositories {
        maven {
            def releasesRepoUrl = layout.buildDirectory.dir('repos/releases')
            def snapshotsRepoUrl = layout.buildDirectory.dir('
repos/snapshots')
            url = version.endsWith('SNAPSHOT') ? snapshotsRepoUrl :
releasesRepoUrl
        }
    }
}

```

Similarly, you can use a [project or system property](#) to decide which repository to publish to. The following example uses the release repository if the project property `release` is set, such as when a user runs `gradle -Prelease publish`:

Example 46. Configuring repository URL based on project property

build.gradle.kts

```

publishing {
    repositories {
        maven {
            val releasesRepoUrl = layout.buildDirectory.dir("repos/releases")
            val snapshotsRepoUrl =
layout.buildDirectory.dir("repos/snapshots")
            url = uri(if (project.hasProperty("release")) releasesRepoUrl
else snapshotsRepoUrl)
        }
    }
}

```

build.gradle

```

publishing {
    repositories {
        maven {

```

```

        def releasesRepoUrl = layout.buildDirectory.dir('repos/releases')
        def snapshotsRepoUrl = layout.buildDirectory.dir('
repos/snapshots')
        url = project.hasProperty('release') ? releasesRepoUrl :
snapshotsRepoUrl
    }
}
}

```

Publishing to Maven Local

For integration with a local Maven installation, it is sometimes useful to publish the module into the Maven local repository (typically at *<home directory of the current user>/m2/repository*), along with its POM file and other metadata. In Maven parlance, this is referred to as 'installing' the module.

The Maven Publish Plugin makes this easy to do by automatically creating a [PublishToMavenLocal](#) task for each [MavenPublication](#) in the `publishing.publications` container. The task name follows the pattern of `publishPubNamePublicationToMavenLocal`. Each of these tasks is wired into the `publishToMavenLocal` aggregate task. You do not need to have `mavenLocal()` in your `publishing.repositories` section.

Publishing Maven relocation information

When a project changes the `groupId` or `artifactId` (the *coordinates*) of an artifact it publishes, it is important to let users know where the new artifact can be found. Maven can help with that through the *relocation* feature. The way this works is that a project publishes an additional artifact under the old coordinates consisting only of a minimal *relocation POM*; that POM file specifies where the new artifact can be found. Maven repository browsers and build tools can then inform the user that the coordinates of an artifact have changed.

For this, a project adds an additional [MavenPublication](#) specifying a [MavenPomRelocation](#):

Example 47. Specifying a relocation POM

build.gradle.kts

```

publishing {
    publications {
        // ... artifact publications

        // Specify relocation POM
        create<MavenPublication>("relocation") {
            pom {
                // Old artifact coordinates
                groupId = "com.example"
                artifactId = "lib"
                version = "2.0.0"
            }
        }
    }
}

```

```

distributionManagement {
    relocation {
        // New artifact coordinates
        groupId = "com.new-example"
        artifactId = "lib"
        version = "2.0.0"
        message = "groupId has been changed"
    }
}
}
}
}
}
}
}

```

build.gradle

```

publishing {
    publications {
        // ... artifact publications

        // Specify relocation POM
        relocation(MavenPublication) {
            pom {
                // Old artifact coordinates
                groupId = "com.example"
                artifactId = "lib"
                version = "2.0.0"

                distributionManagement {
                    relocation {
                        // New artifact coordinates
                        groupId = "com.new-example"
                        artifactId = "lib"
                        version = "2.0.0"
                        message = "groupId has been changed"
                    }
                }
            }
        }
    }
}
}
}
}
}
}
}

```

Only the property which has changed needs to be specified under *relocation*, that is *artifactId* and / or *groupId*. All other properties are optional.

TIP

Specifying the `version` can be useful when the new artifact has a different version, for example because version numbering has started at 1.0.0 again.

A custom `message` allows explaining why the artifact coordinates have changed.

The relocation POM should be created for what would be the next version of the old artifact. For example when the artifact coordinates of `com.example:lib:1.0.0` are changed and the artifact with the new coordinates continues version numbering and is published as `com.new-example:lib:2.0.0`, then the relocation POM should specify a relocation from `com.example:lib:2.0.0` to `com.new-example:lib:2.0.0`.

A relocation POM only has to be published once, the build file configuration for it should be removed again once it has been published.

Note that a relocation POM is not suitable for all situations; when an artifact has been split into two or more separate artifacts then a relocation POM might not be helpful.

Retroactively publishing relocation information

It is possible to publish relocation information retroactively after the coordinates of an artifact have changed in the past, and no relocation information was published back then.

The same recommendations as described above apply. To ease migration for users, it is important to pay attention to the `version` specified in the relocation POM. The relocation POM should allow the user to move to the new artifact in one step, and then allow them to update to the latest version in a separate step. For example when for the coordinates of `com.new-example:lib:5.0.0` were changed in version 2.0.0, then ideally the relocation POM should be published for the old coordinates `com.example:lib:2.0.0` relocating to `com.new-example:lib:2.0.0`. The user can then switch from `com.example:lib` to `com.new-example` and then separately update from version 2.0.0 to 5.0.0, handling breaking changes (if any) step by step.

When relocation information is published retroactively, it is not necessary to wait for next regular release of the project, it can be published in the meantime. As mentioned above, the relocation information should then be removed again from the build file once the relocation POM has been published.

Avoiding duplicate dependencies

When only the coordinates of the artifact have changed, but package names of the classes inside the artifact have remained the same, dependency conflicts can occur. A project might (transitively) depend on the old artifact but at the same time also have a dependency on the new artifact which both contain the same classes, potentially with incompatible changes.

To detect such conflicting duplicate dependencies, [capabilities](#) can be published as part of the [Gradle Module Metadata](#). For an example using a [Java Library](#) project, see [declaring additional capabilities for a local component](#).

Performing a dry run

To verify that relocation information works as expected before publishing it to a remote repository,

it can first be [published to the local Maven repository](#). Then a local test Gradle or Maven project can be created which has the relocation artifact as dependency.

Complete example

The following example demonstrates how to sign and publish a Java library including sources, Javadoc, and a customized POM:

Example 48. [Publishing a Java library](#)

build.gradle.kts

```
plugins {
    `java-library`
    `maven-publish`
    signing
}

group = "com.example"
version = "1.0"

java {
    withJavadocJar()
    withSourcesJar()
}

publishing {
    publications {
        create<MavenPublication>("mavenJava") {
            artifactId = "my-library"
            from(components["java"])
            versionMapping {
                usage("java-api") {
                    fromResolutionOf("runtimeClasspath")
                }
                usage("java-runtime") {
                    fromResolutionResult()
                }
            }
        }
    }
    pom {
        name = "My Library"
        description = "A concise description of my library"
        url = "http://www.example.com/library"
        properties = mapOf(
            "myProp" to "value",
            "prop.with.dots" to "anotherValue"
        )
        licenses {
            license {
                name = "The Apache License, Version 2.0"
```

```

        url = "http://www.apache.org/licenses/LICENSE-
2.0.txt"
    }
}
developers {
    developer {
        id = "johnd"
        name = "John Doe"
        email = "john.doe@example.com"
    }
}
scm {
    connection = "scm:git:git://example.com/my-library.git"
    developerConnection = "scm:git:ssh://example.com/my-
library.git"
    url = "http://example.com/my-library/"
}
}
}
repositories {
    maven {
        // change URLs to point to your repos, e.g. http://my.org/repos
        val releasesRepoUrl =
uri(layout.buildDirectory.dir("repos/releases"))
        val snapshotsRepoUrl =
uri(layout.buildDirectory.dir("repos/snapshots"))
        url = if (version.toString().endsWith("SNAPSHOT"))
snapshotsRepoUrl else releasesRepoUrl
    }
}
}
signing {
    sign(publishing.publications["mavenJava"])
}
tasks.javadoc {
    if (JavaVersion.current().isJava9Compatible) {
        (options as StandardJavadocDocletOptions).addBooleanOption("html5",
true)
    }
}
}

```

build.gradle

```

plugins {
    id 'java-library'
    id 'maven-publish'
}

```

```

    id 'signing'
}

group = 'com.example'
version = '1.0'

java {
    withJavadocJar()
    withSourcesJar()
}

publishing {
    publications {
        mavenJava(MavenPublication) {
            artifactId = 'my-library'
            from components.java
            versionMapping {
                usage('java-api') {
                    fromResolutionOf('runtimeClasspath')
                }
                usage('java-runtime') {
                    fromResolutionResult()
                }
            }
        }
        pom {
            name = 'My Library'
            description = 'A concise description of my library'
            url = 'http://www.example.com/library'
            properties = [
                myProp: "value",
                "prop.with.dots": "anotherValue"
            ]
            licenses {
                license {
                    name = 'The Apache License, Version 2.0'
                    url = 'http://www.apache.org/licenses/LICENSE-
2.0.txt'
                }
            }
            developers {
                developer {
                    id = 'johnd'
                    name = 'John Doe'
                    email = 'john.doe@example.com'
                }
            }
            scm {
                connection = 'scm:git:git://example.com/my-library.git'
                developerConnection = 'scm:git:ssh://example.com/my-
library.git'
                url = 'http://example.com/my-library/'
            }
        }
    }
}

```

```

    }
  }
}
repositories {
    maven {
        // change URLs to point to your repos, e.g. http://my.org/repo
        def releasesRepoUrl = layout.buildDirectory.dir('repos/releases')
        def snapshotsRepoUrl = layout.buildDirectory.dir('
repos/snapshots')
        url = version.endsWith('SNAPSHOT') ? snapshotsRepoUrl :
releasesRepoUrl
    }
}

signing {
    sign publishing.publications.mavenJava
}

javadoc {
    if(JavaVersion.current().isJava9Compatible()) {
        options.addBooleanOption('html5', true)
    }
}

```

The result is that the following artifacts will be published:

- The POM: `my-library-1.0.pom`
- The primary JAR artifact for the Java component: `my-library-1.0.jar`
- The sources JAR artifact that has been explicitly configured: `my-library-1.0-sources.jar`
- The Javadoc JAR artifact that has been explicitly configured: `my-library-1.0-javadoc.jar`

The [Signing Plugin](#) is used to generate a signature file for each artifact. In addition, checksum files will be generated for all artifacts and signature files.

TIP

`publishToMavenLocal` does not create checksum files in $USER_HOME/.m2/repository. If you want to verify that the checksum files are created correctly, or use them for later publishing, consider configuring a custom Maven repository with a file:// URL and using that as the publishing target instead.`

Removal of deferred configuration behavior

Prior to Gradle 5.0, the `publishing {}` block was (by default) implicitly treated as if all the logic inside it was executed after the project is evaluated. This behavior caused quite a bit of confusion

and was deprecated in Gradle 4.8, because it was the only block that behaved that way.

You may have some logic inside your publishing block or in a plugin that is depending on the deferred configuration behavior. For instance, the following logic assumes that the subprojects will be evaluated when the artifactId is set:

build.gradle.kts

```
subprojects {
    publishing {
        publications {
            create<MavenPublication>("mavenJava") {
                from(components["java"])
                artifactId = tasks.jar.get().archiveBaseName.get()
            }
        }
    }
}
```

build.gradle

```
subprojects {
    publishing {
        publications {
            mavenJava(MavenPublication) {
                from components.java
                artifactId = jar.archiveBaseName
            }
        }
    }
}
```

This kind of logic must now be wrapped in an `afterEvaluate {}` block.

build.gradle.kts

```
subprojects {
    publishing {
        publications {
            create<MavenPublication>("mavenJava") {
                from(components["java"])
                afterEvaluate {
                    artifactId = tasks.jar.get().archiveBaseName.get()
                }
            }
        }
    }
}
```

```

    }
  }
}

```

build.gradle

```

subprojects {
    publishing {
        publications {
            mavenJava(MavenPublication) {
                from components.java
                afterEvaluate {
                    artifactId = jar.archiveBaseName
                }
            }
        }
    }
}

```

The Ivy Publish Plugin

The Ivy Publish Plugin provides the ability to publish build artifacts in the [Apache Ivy](#) format, usually to a repository for consumption by other builds or projects. What is published is one or more artifacts created by the build, and an Ivy *module descriptor* (normally `ivy.xml`) that describes the artifacts and the dependencies of the artifacts, if any.

A published Ivy module can be consumed by Gradle (see [Declaring Dependencies](#)) and other tools that understand the Ivy format. You can learn about the fundamentals of publishing in [Publishing Overview](#).

Usage

To use the Ivy Publish Plugin, include the following in your build script:

Example 49. [Applying the Ivy Publish Plugin](#)

build.gradle.kts

```

plugins {
    `ivy-publish`
}

```

build.gradle

```
plugins {  
    id 'ivy-publish'  
}
```

The Ivy Publish Plugin uses an extension on the project named `publishing` of type `PublishingExtension`. This extension provides a container of named publications and a container of named repositories. The Ivy Publish Plugin works with `IvyPublication` publications and `IvyArtifactRepository` repositories.

Tasks

`generateDescriptorFileForPubNamePublication` — `GenerateIvyDescriptor`

Creates an Ivy descriptor file for the publication named *PubName*, populating the known metadata such as project name, project version, and the dependencies. The default location for the descriptor file is *build/publications/\$pubName/ivy.xml*.

`publishPubNamePublicationToRepoNameRepository` — `PublishToIvyRepository`

Publishes the *PubName* publication to the repository named *RepoName*. If you have a repository definition without an explicit name, *RepoName* will be "Ivy".

`publish`

Depends on: All `publishPubNamePublicationToRepoNameRepository` tasks

An aggregate task that publishes all defined publications to all defined repositories.

Publications

This plugin provides `publications` of type `IvyPublication`. To learn how to define and use publications, see the section on [basic publishing](#).

There are four main things you can configure in an Ivy publication:

- A `component` — via `IvyPublication.from(org.gradle.api.component.SoftwareComponent)`.
- `Custom artifacts` — via the `IvyPublication.artifact(java.lang.Object)` method. See `IvyArtifact` for the available configuration options for custom Ivy artifacts.
- Standard metadata like `module`, `organisation` and `revision`.
- Other contents of the module descriptor — via `IvyPublication.descriptor(org.gradle.api.Action)`.

You can see all of these in action in the [complete publishing example](#). The API documentation for `IvyPublication` has additional code samples.

Identity values for the published project

The generated Ivy module descriptor file contains an `<info>` element that identifies the module. The default identity values are derived from the following:

- `organisation` - `Project.getGroup()`
- `module` - `Project.getName()`
- `revision` - `Project.getVersion()`
- `status` - `Project.getStatus()`
- `branch` - (not set)

Overriding the default identity values is easy: simply specify the `organisation`, `module` or `revision` properties when configuring the `IvyPublication`. `status` and `branch` can be set via the `descriptor` property — see `IvyModuleDescriptorSpec`.

The `descriptor` property can also be used to add additional custom elements as children of the `<info>` element, like so:

Example 50. customizing the publication identity

build.gradle.kts

```
publishing {
    publications {
        create<IvyPublication>("ivy") {
            organisation = "org.gradle.sample"
            module = "project1-sample"
            revision = "1.1"
            descriptor.status = "milestone"
            descriptor.branch = "testing"
            descriptor.extraInfo("http://my.namespace", "myElement", "Some
value")

            from(components["java"])
        }
    }
}
```

build.gradle

```
publishing {
    publications {
        ivy(IvyPublication) {
            organisation = 'org.gradle.sample'
            module = 'project1-sample'
            revision = '1.1'
```

```

        descriptor.status = 'milestone'
        descriptor.branch = 'testing'
        descriptor.extraInfo 'http://my.namespace', 'myElement', 'Some
value'

        from components.java
    }
}
}

```

TIP

Certain repositories are not able to handle all supported characters. For example, the `:` character cannot be used as an identifier when publishing to a filesystem-backed repository on Windows.

Gradle will handle any valid Unicode character for `organisation`, `module` and `revision` (as well as the artifact's `name`, `extension` and `classifier`). The only values that are explicitly prohibited are `\`, `/` and any ISO control character. The supplied values are validated early during publication.

Customizing the generated module descriptor

At times, the module descriptor file generated from the project information will need to be tweaked before publishing. The Ivy Publish Plugin provides a DSL for that purpose. Please see [IvyModuleDescriptorSpec](#) in the DSL Reference for the complete documentation of available properties and methods.

The following sample shows how to use the most common aspects of the DSL:

Example 51. Customizing the module descriptor file

build.gradle.kts

```

publications {
    create<IvyPublication>("ivyCustom") {
        descriptor {
            license {
                name = "The Apache License, Version 2.0"
                url = "http://www.apache.org/licenses/LICENSE-2.0.txt"
            }
            author {
                name = "Jane Doe"
                url = "http://example.com/users/jane"
            }
            description {
                text = "A concise description of my library"
                homepage = "http://www.example.com/library"
            }
        }
    }
}

```

```

        versionMapping {
            usage("java-api") {
                fromResolutionOf("runtimeClasspath")
            }
            usage("java-runtime") {
                fromResolutionResult()
            }
        }
    }
}

```

build.gradle

```

publications {
    ivyCustom(IvyPublication) {
        descriptor {
            license {
                name = 'The Apache License, Version 2.0'
                url = 'http://www.apache.org/licenses/LICENSE-2.0.txt'
            }
            author {
                name = 'Jane Doe'
                url = 'http://example.com/users/jane'
            }
            description {
                text = 'A concise description of my library'
                homepage = 'http://www.example.com/library'
            }
        }
        versionMapping {
            usage('java-api') {
                fromResolutionOf('runtimeClasspath')
            }
            usage('java-runtime') {
                fromResolutionResult()
            }
        }
    }
}

```

In this example we are simply adding a 'description' element to the generated Ivy dependency descriptor, but this hook allows you to modify any aspect of the generated descriptor. For example, you could replace the version range for a dependency with the actual version used to produce the build.

You can also add arbitrary XML to the descriptor file via

`IvyModuleDescriptorSpec.withXml(org.gradle.api.Action)`, but you cannot use it to modify any part of the module identifier (organisation, module, revision).

CAUTION

It is possible to modify the descriptor in such a way that it is no longer a valid Ivy module descriptor, so care must be taken when using this feature.

Customizing dependencies versions

Two strategies are supported for publishing dependencies:

Declared versions (default)

This strategy publishes the versions that are defined by the build script author with the dependency declarations in the `dependencies` block. Any other kind of processing, for example through [a rule changing the resolved version](#), will not be taken into account for the publication.

Resolved versions

This strategy publishes the versions that were resolved during the build, possibly by applying resolution rules and automatic conflict resolution. This has the advantage that the published versions correspond to the ones the published artifact was tested against.

Example use cases for resolved versions:

- A project uses dynamic versions for dependencies but prefers exposing the resolved version for a given release to its consumers.
- In combination with [dependency locking](#), you want to publish the locked versions.
- A project leverages the rich versions constraints of Gradle, which have a lossy conversion to Ivy. Instead of relying on the conversion, it publishes the resolved versions.

This is done by using the `versionMapping` DSL method which allows to configure the `VersionMappingStrategy`:

Example 52. Using resolved versions

build.gradle.kts

```
publications {
    create<IvyPublication>("ivyCustom") {
        versionMapping {
            usage("java-api") {
                fromResolutionOf("runtimeClasspath")
            }
            usage("java-runtime") {
                fromResolutionResult()
            }
        }
    }
}
```

build.gradle

```
publications {
    ivyCustom(IvyPublication) {
        versionMapping {
            usage('java-api') {
                fromResolutionOf('runtimeClasspath')
            }
            usage('java-runtime') {
                fromResolutionResult()
            }
        }
    }
}
```

In the example above, Gradle will use the versions resolved on the `runtimeClasspath` for dependencies declared in `api`, which are mapped to the `compile` configuration of Ivy. Gradle will also use the versions resolved on the `runtimeClasspath` for dependencies declared in `implementation`, which are mapped to the `runtime` configuration of Ivy. `fromResolutionResult()` indicates that Gradle should use the default classpath of a variant and `runtimeClasspath` is the default classpath of `java-runtime`.

Repositories

This plugin provides `repositories` of type `IvyArtifactRepository`. To learn how to define and use repositories for publishing, see the section on [basic publishing](#).

Here's a simple example of defining a publishing repository:

Example 53. [Declaring repositories to publish to](#)

build.gradle.kts

```
publishing {
    repositories {
        ivy {
            // change to point to your repo, e.g. http://my.org/repo
            url = uri(layout.buildDirectory.dir("repo"))
        }
    }
}
```

build.gradle

```
publishing {
    repositories {
        ivy {
            // change to point to your repo, e.g. http://my.org/repo
            url = layout.buildDirectory.dir("repo")
        }
    }
}
```

The two main things you will want to configure are the repository's:

- URL (required)
- Name (optional)

You can define multiple repositories as long as they have unique names within the build script. You may also declare one (and only one) repository without a name. That repository will take on an implicit name of "Ivy".

You can also configure any authentication details that are required to connect to the repository. See [IvyArtifactRepository](#) for more details.

Complete example

The following example demonstrates publishing with a multi-project build. Each project publishes a Java component configured to also build and publish Javadoc and source code artifacts. The descriptor file is customized to include the project description for each project.

Example 54. [Publishing a Java module](#)

settings.gradle.kts

```
rootProject.name = "ivy-publish-java"
include("project1", "project2")
```

buildSrc/build.gradle.kts

```
plugins {
    `kotlin-dsl`
}

repositories {
    gradlePluginPortal()
}
```

buildSrc/src/main/kotlin/myproject.publishing-conventions.gradle.kts

```
plugins {
    id("java-library")
    id("ivy-publish")
}

version = "1.0"
group = "org.gradle.sample"

repositories {
    mavenCentral()
}

java {
    withJavadocJar()
    withSourcesJar()
}

publishing {
    repositories {
        ivy {
            // change to point to your repo, e.g. http://my.org/repo
            url = uri(rootProject.layout.buildDirectory.dir("repo"))
        }
    }
    publications {
        create<IvyPublication>("ivy") {
            from(components["java"])
            descriptor.description {
                text = providers.provider({ description })
            }
        }
    }
}
```

project1/build.gradle.kts

```
plugins {
    id("myproject.publishing-conventions")
}

description = "The first project"

dependencies {
    implementation("junit:junit:4.13")
    implementation(project(":project2"))
}
```

project2/build.gradle.kts

```
plugins {  
    id("myproject.publishing-conventions")  
}  
  
description = "The second project"  
  
dependencies {  
    implementation("commons-collections:commons-collections:3.2.2")  
}
```

settings.gradle

```
rootProject.name = 'ivy-publish-java'  
include 'project1', 'project2'
```

buildSrc/build.gradle

```
plugins {  
    id 'groovy-gradle-plugin'  
}
```

buildSrc/src/main/groovy/myproject.publishing-conventions.gradle

```
plugins {  
    id 'java-library'  
    id 'ivy-publish'  
}  
  
version = '1.0'  
group = 'org.gradle.sample'  
  
repositories {  
    mavenCentral()  
}  
  
java {  
    withJavadocJar()  
    withSourcesJar()  
}  
  
publishing {  
    repositories {  
        ivy {  
            // change to point to your repo, e.g. http://my.org/repo  
            url = rootProject.layout.buildDirectory.dir('repo')  
        }  
    }  
}
```

```

    }
    publications {
        ivy(IvyPublication) {
            from components.java
            descriptor.description {
                text = providers.provider({ description })
            }
        }
    }
}

```

project1/build.gradle

```

plugins {
    id 'myproject.publishing-conventions'
}

description = 'The first project'

dependencies {
    implementation 'junit:junit:4.13'
    implementation project(':project2')
}

```

project2/build.gradle

```

plugins {
    id 'myproject.publishing-conventions'
}

description = 'The second project'

dependencies {
    implementation 'commons-collections:commons-collections:3.2.2'
}

```

The result is that the following artifacts will be published for each project:

- The Gradle Module Metadata file: `project1-1.0.module`.
- The Ivy module metadata file: `ivy-1.0.xml`.
- The primary JAR artifact for the Java component: `project1-1.0.jar`.
- The Javadoc and sources JAR artifacts of the Java component (because we configured `withJavadocJar()` and `withSourcesJar()`): `project1-1.0-javadoc.jar`, `project1-1.0-source.jar`.

OTHER TOPICS

Verifying dependencies

Working with external dependencies and plugins published on third-party repositories puts your build at risk. In particular, you need to be aware of what binaries are brought in transitively and if they are legit. To mitigate the security risks and avoid integrating compromised dependencies in your project, Gradle supports *dependency verification*.

Dependency verification is, by nature, an inconvenient feature to use. It means that whenever you're going to update a dependency, builds are likely to fail. It means that merging branches are going to be harder because each branch can have different dependencies. It means that you will be tempted to switch it off.

So why should you bother?

Dependency verification is about **trust** in what you get and what you ship.

Without dependency verification it's easy for an attacker to compromise your supply chain. There are many real world examples of tools compromised by adding a malicious dependency. Dependency verification is meant to protect yourself from those attacks, by forcing you to ensure that the artifacts you include in your build are the ones that you expect. It is not meant, however, to prevent you from including *vulnerable* dependencies.

Finding the right balance between security and convenience is hard but Gradle will try to let you choose the "right level" for you.

Dependency verification consists of two different and complementary operations:

- *checksum verification*, which allows asserting the integrity of a dependency
- *signature verification*, which allows asserting the provenance of a dependency

Gradle supports both checksum and signature verification out of the box but performs no dependency verification by default. This section will guide you into configuring dependency verification properly for your needs.

This feature can be used for:

- detecting compromised dependencies
- detecting compromised plugins
- detecting tampered dependencies in the local dependency caches

Enabling dependency verification

The verification metadata file

NOTE

Currently the only source of dependency verification metadata is this XML configuration file. Future versions of Gradle may include other sources (for

example via external services).

Dependency verification is automatically enabled once the configuration file for dependency verification is discovered. This configuration file is located at `$PROJECT_ROOT/gradle/verification-metadata.xml`. This file minimally consists of the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata xmlns="https://schema.gradle.org/dependency-verification"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://schema.gradle.org/dependency-verification
https://schema.gradle.org/dependency-verification/dependency-verification-1.3.xsd">
  <configuration>
    <verify-metadata>true</verify-metadata>
    <verify-signatures>false</verify-signatures>
  </configuration>
</verification-metadata>
```

Doing so, Gradle will verify all artifacts using `checksums`, but will not verify `signatures`. Gradle will verify any artifact downloaded using its dependency management engine, which includes, but is not limited to:

- artifact files (e.g jar files, zips, ...) used during a build
- metadata artifacts (POM files, Ivy descriptors, Gradle Module Metadata)
- plugins (both project and settings plugins)
- artifacts resolved using the advanced dependency resolution APIs

Gradle will *not* verify changing dependencies (in particular `SNAPSHOT` dependencies) nor locally produced artifacts (typically jars produced during the build itself) as by nature their checksums and signatures would always change.

With such a minimal configuration file, a project using *any* external dependency or plugin would immediately start failing because it doesn't contain any checksum to verify.

Scope of the dependency verification

A dependency verification configuration is *global*: a single file is used to configure verification of the whole build. In particular, the same file is used for both the (sub)projects and `buildSrc`.

If an included build is used:

- the configuration file of the *current* build is used for verification
- so if the included build itself uses verification, its configuration is ignored in favor of the current one
- which means that including a build works similarly to upgrading a dependency: it may require you to update your current verification metadata

An easy way to get started is therefore to generate the minimal configuration for an existing build.

Configuring the console output

By default, if dependency verification fails, Gradle will generate a small summary about the verification failure as well as an HTML report containing the full information about the failures. If your environment prevents you from reading this HTML report file (for example if you run a build on CI and that it's not easy to fetch the remote artifacts), Gradle provides a way to opt-in a verbose console report. For this, you need to add this Gradle property to your `gradle.properties` file:

```
org.gradle.dependency.verification.console=verbose
```

Bootstrapping dependency verification

It's worth mentioning that while Gradle can generate a dependency verification file for you, you should always check whatever Gradle generated for you because your build may *already* contain compromised dependencies without you knowing about it. Please refer to the appropriate [checksum verification](#) or [signature verification](#) section for more information.

If you plan on using [signature verification](#), please also read the [corresponding section](#) of the docs.

Bootstrapping can either be used to create a file from the beginning, or also to *update* an existing file with new information. Therefore, it's recommended to always use the same parameters once you started bootstrapping.

The dependency verification file can be generated with the following CLI instructions:

```
gradle --write-verification-metadata sha256 help
```

The `write-verification-metadata` flag requires the list of [checksums](#) that you want to generate or `pgp` for [signatures](#).

Executing this command line will cause Gradle to:

- resolve all [resolvable configurations](#), which includes:
 - configurations from the root project
 - configurations from all subprojects
 - configurations from `buildSrc`
 - included builds configurations
 - configurations used by plugins
- download all artifacts discovered during resolution
- compute the requested checksums and possibly verify signatures depending on what you asked
- At the end of the build, generate the configuration file which will contain the inferred *verification metadata*

As a consequence, the `verification-metadata.xml` file will be used in subsequent builds to verify dependencies.

There are dependencies that Gradle *cannot* discover this way. In particular, you will notice that the CLI above uses the `help` task. If you don't specify any task, Gradle will automatically run the default task and generate a configuration file at the end of the build too.

The difference is that Gradle *may* discover more dependencies and artifacts depending on the tasks you execute. As a matter of fact, Gradle cannot automatically discover *detached configurations*, which are basically dependency graphs resolved as an internal implementation detail of the execution of a task: they are not, in particular, declared as an input of the task because they effectively depend on the configuration of the task at execution time.

A good way to start is just to use the simplest task, `help`, which will discover as much as possible, and if subsequent builds fail with a verification error, you can re-execute generation with the appropriate tasks to "discover" more dependencies.

Gradle won't verify either checksums or signatures of plugins which use their own HTTP clients. Only plugins which use the infrastructure provided by Gradle for performing requests will see their requests verified.

Using generation for incremental updates

The verification file generated by Gradle has a strict ordering for all its content. It also uses the information from the existing state to limit changes to the strict minimum.

This means that generation is actually a convenient tool for *updating* a verification file:

- Checksum entries generated by Gradle will have a clear `origin` that starts with "Generated by Gradle", which is a good indicator that an entry needs to be reviewed,
- Entries added by hand will immediately be accounted for, and appear at the right location after writing the file,
- The header comments of the file will be preserved, i.e. comments before the root XML node. This allows you to have a license header or instructions on which tasks and which parameters to use for generating that file.

With the above benefits, it is really easy to account for new dependencies or dependency versions by simply generating the file again and reviewing the changes.

Using dry mode

By default, bootstrapping is incremental, which means that if you run it multiple times, information is *added* to the file and in particular you can rely on your VCS to check the diffs. There are situations where you would just want to see what the generated verification metadata file would look like without actually changing the existing one or overwriting it.

For this purpose, you can just add `--dry-run`:

```
gradle --write-verification-metadata sha256 help --dry-run
```

Then instead of generating the `verification-metadata.xml` file, a *new file* will be generated, called

verification-metadata.dryrun.xml.

NOTE

Because `--dry-run` doesn't execute tasks, this would be much faster, but it will miss any resolution happening at task execution time.

Disabling metadata verification

By default, Gradle will not only verify artifacts (jars, ...) but also the metadata associated with those artifacts (typically POM files). Verifying this ensures the maximum level of security: metadata files typically tell what transitive dependencies will be included, so a compromised metadata file may cause the introduction of undesired dependencies in the graph. However, because all artifacts are verified, such artifacts would in general easily be discovered by you, because they would cause a checksum verification failure (checksums would be *missing* from verification metadata). Because metadata verification can significantly increase the size of your configuration file, you may therefore want to disable verification of metadata. If you understand the risks of doing so, set the `<verify-metadata>` flag to `false` in the configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata xmlns="https://schema.gradle.org/dependency-verification"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://schema.gradle.org/dependency-verification
https://schema.gradle.org/dependency-verification/dependency-verification-1.3.xsd">
  <configuration>
    <verify-metadata>false</verify-metadata>
    <verify-signatures>false</verify-signatures>
  </configuration>
  <!-- the rest of this file doesn't need to declare anything about metadata files
-->
</verification-metadata>
```

Verifying dependency checksums

Checksum verification allows you to ensure the integrity of an artifact. This is the simplest thing that Gradle can do for you to make sure that the artifacts you use are un-tampered.

Gradle supports MD5, SHA1, SHA-256 and SHA-512 checksums. However, only SHA-256 and SHA-512 checksums are considered secure nowadays.

Adding the checksum for an artifact

External components are identified by GAV coordinates, then each of the artifacts by their file names. To declare the checksums of an artifact, you need to add the corresponding section in the verification metadata file. For example, to declare the checksum for [Apache PDFBox](#). The GAV coordinates are:

- group `org.apache.pdfbox`
- name `pdfbox`

- version **2.0.17**

Using this dependency will trigger the download of 2 different files:

- **pdfbox-2.0.17.jar** which is the main artifact
- **pdfbox-2.0.17.pom** which is the metadata file associated with this artifact

As a consequence, you need to declare the checksums for both of them (unless you [disabled metadata verification](#)):

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata xmlns="https://schema.gradle.org/dependency-verification"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://schema.gradle.org/dependency-verification
https://schema.gradle.org/dependency-verification/dependency-verification-1.3.xsd">
  <configuration>
    <verify-metadata>true</verify-metadata>
    <verify-signatures>false</verify-signatures>
  </configuration>
  <components>
    <component group="org.apache.pdfbox" name="pdfbox" version="2.0.17">
      <artifact name="pdfbox-2.0.17.jar">
        <sha512 value=
"7e11e54a21c395d461e59552e88b0de0ebaf1bf9d9bcacadf17b240d9bbc29bf6beb8e36896c186fe405d
287f5d517b02c89381aa0fcc5e0aa5814e44f0ab331" origin="PDFBox Official site
(https://pdfbox.apache.org/download.cgi)"/>
      </artifact>
      <artifact name="pdfbox-2.0.17.pom">
        <sha512 value=
"82de436b38faf6121d8d2e71dda06e79296fc0f7bc7aba0766728c8d306fd1b0684b5379c18808ca724bf
91707277eba81eb4fe19518e99e8f2a56459b79742f" origin="Generated by Gradle"/>
      </artifact>
    </component>
  </components>
</verification-metadata>
```

Where to get checksums from?

In general, checksums are published alongside artifacts on public repositories. However, if a dependency is compromised in a repository, it's likely its checksum will be too, so it's a good practice to get the checksum from a different place, usually the website of the library itself.

In fact, it's a good security practice to publish the checksums of artifacts on a *different server* than the server where the artifacts themselves are hosted: it's harder to compromise a library both on the repository *and* the official website.

In the example above, the checksum was published on the website for the JAR, but not the POM file. This is why it's usually easier to [let Gradle generate the checksums](#) and verify by reviewing the generated file carefully.

In this example, not only could we check that the checksum was correct, but we could also find it on the official website, which is why we changed the value of the `origin` attribute on the `sha512` element from `Generated by Gradle` to `PDFBox Official site`. Changing the `origin` gives users a sense of how trustworthy your build is.

Interestingly, using `pdfbox` will require *much more* than those 2 artifacts, because it will also bring in transitive dependencies. If the dependency verification file only included the checksums for the main artifacts you used, the build would fail with an error like this one:

```
Execution failed for task ':compileJava'.
> Dependency verification failed for configuration ':compileClasspath':
   - On artifact commons-logging-1.2.jar (commons-logging:commons-logging:1.2) in
     repository 'MavenRepo': checksum is missing from verification metadata.
   - On artifact commons-logging-1.2.pom (commons-logging:commons-logging:1.2) in
     repository 'MavenRepo': checksum is missing from verification metadata.
```

What this indicates is that your build requires `commons-logging` when executing `compileJava`, however the verification file doesn't contain enough information for Gradle to verify the integrity of the dependencies, meaning you need to add the required information to the verification metadata file.

See [troubleshooting dependency verification](#) for more insights on what to do in this situation.

What checksums are verified?

If a dependency verification metadata file declares more than one checksum for a dependency, Gradle will verify *all of them* and fail if *any of them fails*. For example, the following configuration would check both the `md5` and `sha1` checksums:

```
<component group="org.apache.pdfbox" name="pdfbox" version="2.0.17">
  <artifact name="pdfbox-2.0.17.jar">
    <md5 value="c713a8e252d0add65e9282b151adf6b4" origin="official site"/>
    <sha1 value="b5c8dff799bd967c70ccae75e6972327ae640d35" origin="official site"
reason="Additional check for this artifact"/>
  </artifact>
</component>
```

There are multiple reasons why you'd like to do so:

1. an official site doesn't publish *secure* checksums (SHA-256, SHA-512) but publishes multiple insecure ones (MD5, SHA1). While it's easy to fake a MD5 checksum and hard but possible to fake a SHA1 checksum, it's harder to fake both of them for the same artifact.
2. you might want to add generated checksums to the list above
3. when *updating* dependency verification file with more secure checksums, you don't want to accidentally erase checksums

Verifying dependency signatures

In addition to [checksums](#), Gradle supports verification of signatures. Signatures are used to assess the *provenance* of a dependency (it tells who signed the artifacts, which usually corresponds to who produced it).

As enabling signature verification usually means a higher level of security, you might want to replace checksum verification with signature verification.

WARNING

Signatures *can* also be used to assess the integrity of a dependency similarly to checksums. Signatures are signatures of the *hash* of artifacts, not artifacts themselves. This means that if the signature is done on an *unsafe hash* (even SHA1), then you're not correctly assessing the *integrity* of a file. For this reason, if you care about both, you need to add both signatures *and* checksums to your verification metadata.

However:

- Gradle only supports verification of signatures published on remote repositories as ASCII-armored PGP files
- Not all artifacts are published with signatures
- A good signature doesn't mean that the signatory was legit

As a consequence, signature verification will often be used alongside checksum verification.

About expired keys

It's very common to find artifacts which are signed with an expired key. This is not a problem for *verification*: key expiry is mostly used to avoid signing with a stolen key. If an artifact was signed before expiry, it's still valid.

Enabling signature verification

Because verifying signatures is more expensive (both I/O and CPU wise) and harder to check manually, it's not enabled by default.

Enabling it requires you to change the configuration option in the `verification-metadata.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata xmlns="https://schema.gradle.org/dependency-verification"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://schema.gradle.org/dependency-verification
https://schema.gradle.org/dependency-verification/dependency-verification-1.3.xsd">
  <configuration>
    <verify-signatures>true</verify-signatures>
  </configuration>
</verification-metadata>
```

Understanding signature verification

Once signature verification is enabled, for each artifact, Gradle will:

- try to download the corresponding `.asc` file
- if it's present
 - automatically download the keys required to perform verification of the signature
 - verify the artifact using the downloaded public keys
 - if signature verification passes, perform additional requested checksum verification
- if it's absent, fallback to checksum verification

That is to say that Gradle's verification mechanism is much stronger if signature verification is enabled than just with checksum verification. In particular:

- if an artifact is signed with multiple keys, all of them must pass validation or the build will fail
- if an artifact passes verification, any additional checksum configured for the artifact *will also be checked*

However, it's not because an artifact passes signature verification that you can trust it: you need to *trust the keys*.

In practice, it means you need to list the keys that you trust for each artifact, which is done by adding a `pgp` entry instead of a `sha1` for example:

```
<component group="com.github.javaparser" name="javaparser-core" version="3.6.11">
  <artifact name="javaparser-core-3.6.11.jar">
    <pgp value="8756c4f765c9ac3cb6b85d62379ce192d401ab61"/>
  </artifact>
</component>
```

WARNING

For the `pgp` and `trusted-key` elements, Gradle *requires* full fingerprint IDs (e.g. `b801e2f8ef035068ec1139cc29579f18fa8fd93b` instead of a long ID `29579f18fa8fd93b`). This minimizes the chance of a [collision attack](#).

At the time, [V4 key fingerprints](#) are of 160-bit (40 characters) length. We accept longer keys to be future-proof in case a longer key fingerprint is introduced.

In `ignore-key` elements, either fingerprints or long (64-bit) IDs can be used. A shorter ID can only result in a bigger range of exclusion, therefore, it's safe to use.

This effectively means that you trust `com.github.javaparser:javaparser-core:3.6.11` if it's signed with the key `8756c4f765c9ac3cb6b85d62379ce192d401ab61`.

Without this, the build would fail with this error:

```
> Dependency verification failed for configuration ':compileClasspath':
  - On artifact javaparser-core-3.6.11.jar (com.github.javaparser:javaparser-core:3.6.11) in repository 'MavenRepo': Artifact was signed with key '8756c4f765c9ac3cb6b85d62379ce192d401ab61' (Bintray (by JFrog) <****>) and passed verification but the key isn't in your trusted keys list.
```

NOTE

The key IDs that Gradle shows in error messages are the key IDs found in the signature file it tries to verify. It doesn't mean that it's necessarily the keys that you should trust. In particular, if the signature is correct but done by a malicious entity, Gradle wouldn't tell you.

Trusting keys globally

Signature verification has the advantage that it can make the configuration of dependency verification easier by not having to explicitly list all artifacts like for checksum verification only. In fact, it's common that the same key can be used to sign several artifacts. If this is the case, you can move the trusted key from the artifact level to the global configuration block:

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata xmlns="https://schema.gradle.org/dependency-verification"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://schema.gradle.org/dependency-verification
https://schema.gradle.org/dependency-verification/dependency-verification-1.3.xsd">
  <configuration>
    <verify-metadata>true</verify-metadata>
    <verify-signatures>true</verify-signatures>
    <trusted-keys>
      <trusted-key id="8756c4f765c9ac3cb6b85d62379ce192d401ab61" group=
"com.github.javaparser"/>
    </trusted-keys>
  </configuration>
</verification-metadata>
```

The configuration above means that for any artifact belonging to the group `com.github.javaparser`, we trust it if it's signed with the `8756c4f765c9ac3cb6b85d62379ce192d401ab61` fingerprint.

The `trusted-key` element works similarly to the `trusted-artifact` element:

- `group`, the group of the artifact to trust
- `name`, the name of the artifact to trust
- `version`, the version of the artifact to trust
- `file`, the name of the artifact *file* to trust
- `regex`, a boolean saying if the `group`, `name`, `version` and `file` attributes need to be interpreted as regular expressions (defaults to `false`)

You should be careful when trusting a key globally.

Try to limit it to the appropriate groups or artifacts:

- a valid key may have been used to sign artifact **A** which you trust
- later on, the key is stolen and used to sign artifact **B**

It means you can trust the key **A** for the first artifact, probably only up to the released version before the key was stolen, but not for **B**.

Remember that anybody can put an arbitrary name when generating a PGP key, so never trust the key solely based on the key name. Verify if the key is listed at the official site. For example, Apache projects typically provide a KEYS.txt file that you can trust.

Specifying key servers and ignoring keys

Gradle will automatically download the public keys required to verify a signature. For this it uses a list of well known and trusted key servers (the list may change between Gradle versions, please refer to the implementation to figure out what servers are used by default).

You can explicitly set the list of key servers that you want to use by adding them to the configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata xmlns="https://schema.gradle.org/dependency-verification"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://schema.gradle.org/dependency-verification
https://schema.gradle.org/dependency-verification/dependency-verification-1.3.xsd">
  <configuration>
    <verify-metadata>true</verify-metadata>
    <verify-signatures>true</verify-signatures>
    <key-servers>
      <key-server uri="hkp://my-key-server.org"/>
      <key-server uri="https://my-other-key-server.org"/>
    </key-servers>
  </configuration>
</verification-metadata>
```

Despite this, it's possible that a key is not available:

- because it wasn't published to a public key server
- because it was lost

In this case, you can ignore a key in the configuration block:

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata xmlns="https://schema.gradle.org/dependency-verification"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://schema.gradle.org/dependency-verification
```

```
https://schema.gradle.org/dependency-verification/dependency-verification-1.3.xsd">
  <configuration>
    <verify-metadata>true</verify-metadata>
    <verify-signatures>true</verify-signatures>
    <ignored-keys>
      <ignored-key id="abcdef1234567890" reason="Key is not available in any key
server"/>
    </ignored-keys>
  </configuration>
</verification-metadata>
```

As soon as a key is ignored, it will not be used for verification, even if the signature file mentions it. However, if the signature cannot be verified with at least one other key, Gradle will mandate that you provide a checksum.

NOTE

If Gradle cannot download a key while bootstrapping, it will mark it as ignored. If you can find the key but Gradle does not, you can [manually add it](#) to the keyring file.

Exporting keys for faster verification

Gradle automatically downloads the required keys but this operation can be quite slow and requires everyone to download the keys. To avoid this, Gradle offers the ability to use a local keyring file containing the required public keys. Note that only public key packets and a single `userId` per key are stored and used. All other information (user attributes, signatures, etc.) is stripped from downloaded or exported keys.

Gradle supports 2 different file formats for keyrings: a binary format (`.gpg` file) and a plain text format (`.keys`), also known as ASCII-armored format.

There are pros and cons for each of the formats: the binary format is more compact and can be updated directly via GPG commands, but is completely opaque (binary). On the opposite, the ASCII-armored format is human-readable, can be easily updated by hand and makes it easier to do code reviews thanks to readable diffs.

You can configure which file type would be used by adding the `keyring-format` configuration option:

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata xmlns="https://schema.gradle.org/dependency-verification"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://schema.gradle.org/dependency-verification
https://schema.gradle.org/dependency-verification/dependency-verification-1.3.xsd">
  <configuration>
    <verify-metadata>true</verify-metadata>
    <verify-signatures>true</verify-signatures>
    <keyring-format>armored</keyring-format>
  </configuration>
</verification-metadata>
```

Available options for keyring format are **armored** and **binary**.

Without **keyring-format**, if the **gradle/verification-keyring.gpg** or **gradle/verification-keyring.keys** file is present, Gradle will search for keys there in priority. The plain text file will be ignored if there's already a **.gpg** file (the binary version takes precedence).

You can ask Gradle to export all keys it used for verification of this build to the keyring during bootstrapping:

```
./gradlew --write-verification-metadata pgp,sha256 --export-keys
```

Unless **keyring-format** is specified, this command will generate **both** the binary version and the ASCII-armored file. Use this option to choose the preferred format. You should only pick one for your project.

It's a good idea to commit this file to VCS (as long as you trust your VCS). If you use git and use the binary version, make sure to make it treat this file as binary, by adding this to your **.gitattributes** file:

```
*.gpg          binary
```

You can also ask Gradle to export all trusted keys without updating the verification metadata file:

```
./gradlew --export-keys
```

NOTE | This command will not report verification errors, only export keys.

Bootstrapping and signature verification

WARNING

Signature verification bootstrapping takes an *optimistic point of view* that signature verification is *enough*. Therefore, if you also care about *integrity*, you **must** first bootstrap using checksum verification, *then* with signature verification.

Similarly to bootstrapping for checksums, Gradle provides a convenience for bootstrapping a configuration file with signature verification enabled. For this, just add the **pgp** option to the list of verifications to generate. However, because there might be verification failures, missing keys or missing signature files, you **must** provide a fallback checksum verification algorithm:

```
./gradlew --write-verification-metadata pgp,sha256
```

this means that Gradle will verify the signatures and fallback to SHA-256 checksums when there's a problem.

When bootstrapping, Gradle performs *optimistic verification* and therefore assumes a sane build

environment. It will therefore:

- automatically add the trusted keys as soon as verification passes
- automatically add ignored keys for keys which couldn't be downloaded from public key servers. See [here](#) how to manually add keys if needed
- automatically generate checksums for artifacts without signatures or ignored keys

If, for some reason, verification fails during the generation, Gradle will automatically generate an ignored key entry but warn you that you must absolutely check what happens.

This situation is common as explained for [this section](#): a typical case is when the POM file for a dependency differs from one repository to the other (often in a non-meaningful way).

In addition, Gradle will try to group keys automatically and generate the `trusted-keys` block which reduced the configuration file size as much as possible.

Forcing use of local keyrings only

The local keyring files (`.gpg` or `.keys`) can be used to avoid reaching out to key servers whenever a key is required to verify an artifact. However, it may be that the local keyring doesn't contain a key, in which case Gradle would use the key servers to fetch the missing key. If the local keyring file isn't regularly updated, using [key export](#), then it may be that your CI builds, for example, would reach out to key servers too often (especially if you use disposable containers for builds).

To avoid this, Gradle offers the ability to disallow use of key servers altogether: only the local keyring file would be used, and if a key is missing from this file, the build will fail.

To enable this mode, you need to disable key servers in the configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata xmlns="https://schema.gradle.org/dependency-verification"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://schema.gradle.org/dependency-verification
https://schema.gradle.org/dependency-verification/dependency-verification-1.3.xsd">
  <configuration>
    <key-servers enabled="false"/>
    ...
  </configuration>
  ...
</verification-metadata>
```

NOTE

If you are asking Gradle to [generate a verification metadata file](#) and that an existing verification metadata file sets `enabled` to `false`, then this flag will be ignored, so that potentially missing keys are downloaded.

Disabling verification or making it lenient

Dependency verification can be expensive, or sometimes verification could get in the way of day to

day development (because of frequent dependency upgrades, for example).

Alternatively, you might want to enable verification on CI servers but not on local machines.

Gradle actually provides 3 different verification modes:

- **strict**, which is the default. Verification fails *as early as possible*, in order to avoid the use of compromised dependencies during the build.
- **lenient**, which will run the build even if there are verification failures. The verification errors will be displayed during the build without causing a build failure.
- **off** when verification is totally ignored.

All those modes can be activated on the CLI using the `--dependency-verification` flag, for example:

```
./gradlew --dependency-verification lenient build
```

Alternatively, you can set the `org.gradle.dependency.verification` system property, either on the CLI:

```
./gradlew -Dorg.gradle.dependency.verification=lenient build
```

or in a `gradle.properties` file:

```
org.gradle.dependency.verification=lenient
```

Disabling dependency verification for some configurations only

In order to provide the strongest security level possible, dependency verification is enabled globally. This will ensure, for example, that you trust all the plugins you use. However, the plugins themselves may need to resolve additional dependencies that it doesn't make sense to ask the user to accept. For this purpose, Gradle provides an API which allows *disabling dependency verification on some specific configurations*.

WARNING

Disabling dependency verification, if you care about security, is not a good idea. This API mostly exist for cases where it doesn't make sense to check dependencies. However, in order to be on the safe side, Gradle will systematically print a warning whenever verification has been disabled for a specific configuration.

As an example, a plugin may want to check if there are *newer* versions of a library available and list those versions. It doesn't make sense, in this context, to ask the user to put the checksums of the POM files of the newer releases because by definition, they don't know about them. So the plugin might need to run its code *independently of the dependency verification configuration*.

To do this, you need to call the `ResolutionStrategy#disableDependencyVerification` method:

Example 55. *Disabling dependency verification*

build.gradle.kts

```
configurations {
    "myPluginClasspath" {
        resolutionStrategy {
            disableDependencyVerification()
        }
    }
}
```

build.gradle

```
configurations {
    myPluginClasspath {
        resolutionStrategy {
            disableDependencyVerification()
        }
    }
}
```

It's also possible to disable verification on detached configurations like in the following example:

Example 56. *Disabling dependency verification*

build.gradle.kts

```
tasks.register("checkDetachedDependencies") {
    val detachedConf: FileCollection =
        configurations.detachedConfiguration(dependencies.create("org.apache.commons:commons-lang3:3.3.1")).apply {
            resolutionStrategy.disableDependencyVerification()
        }
    doLast {
        println(detachedConf.files)
    }
}
```

build.gradle

```
tasks.register("checkDetachedDependencies") {
    def detachedConf = configurations.detachedConfiguration(dependencies
```

```
.create("org.apache.commons:commons-lang3:3.3.1"))
    detachedConf.resolutionStrategy.disableDependencyVerification()
    doLast {
        println(detachedConf.files)
    }
}
```

Trusting some particular artifacts

You might want to trust some artifacts more than others. For example, it's legitimate to think that artifacts produced in your company and found in your internal repository only are safe, but you want to check every external component.

NOTE

This is a typical *company policy*. In practice, **nothing** prevents your internal repository from being compromised, so it's a good idea to check your internal artifacts too!

For this purpose, Gradle offers a way to automatically trust some artifacts. You can trust all artifacts in a group by adding this to your configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata xmlns="https://schema.gradle.org/dependency-verification"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://schema.gradle.org/dependency-verification
https://schema.gradle.org/dependency-verification/dependency-verification-1.3.xsd">
    <configuration>
        <trusted-artifacts>
            <trust group="com.mycompany" reason="We trust mycompany artifacts"/>
        </trusted-artifacts>
    </configuration>
</verification-metadata>
```

This means that all components which group is `com.mycompany` will automatically be trusted. Trusted means that Gradle will not perform any verification whatsoever.

The `trust` element accepts those attributes:

- `group`, the group of the artifact to trust
- `name`, the name of the artifact to trust
- `version`, the version of the artifact to trust
- `file`, the name of the artifact *file* to trust
- `regex`, a boolean saying if the `group`, `name`, `version` and `file` attributes need to be interpreted as regular expressions (defaults to `false`)
- `reason`, an optional reason, why matched artifacts are trusted

In the example above it means that the trusted artifacts would be artifacts in `com.mycompany` but not `com.mycompany.other`. To trust all artifacts in `com.mycompany` and all subgroups, you can use:

```
<?xml version="1.0" encoding="UTF-8"?>
<verification-metadata xmlns="https://schema.gradle.org/dependency-verification"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://schema.gradle.org/dependency-verification
https://schema.gradle.org/dependency-verification/dependency-verification-1.3.xsd">
  <configuration>
    <trusted-artifacts>
      <trust group="^com[.]mycompany($|([.].*))" regex="true" reason="We trust all
mycompany artifacts"/>
    </trusted-artifacts>
  </configuration>
</verification-metadata>
```

Trusting multiple checksums for an artifact

It's quite common to have *different checksums for the same artifact* in the wild. How is that possible? Despite progress, it's often the case that developers publish, for example, to Maven Central and another repository separately, using different builds. In general, this is not a problem but sometimes it means that the metadata files would be different (different timestamps, additional whitespaces, ...). Add to this that your build may use several repositories or repository mirrors and it makes it quite likely that a single build can "see" different metadata files for the same component! In general, it's not malicious (but you **must** verify that the artifact is actually correct), so Gradle lets you declare the additional artifact checksums. For example:

```
<component group="org.apache" name="apache" version="13">
  <artifact name="apache-13.pom">
    <sha256 value=
"2fafa38abefe1b40283016f506ba9e844bfcf18713497284264166a5dbf4b95e">
      <also-trust value=
"ff513db0361fd41237bef4784968bc15aae478d4ec0a9496f811072ccaf3841d"/>
    </sha256>
  </artifact>
</component>
```

You can have as many `also-trust` entries as needed, but in general you shouldn't have more than 2.

Skipping Javadocs and sources

By default Gradle will verify *all* downloaded artifacts, which includes Javadocs and sources. In general this is not a problem but you might face an issue with IDEs which automatically try to download them during import: if you didn't set the checksums for those too, importing would fail.

To avoid this, you can configure Gradle to trust automatically all javadocs/sources:

```
<trusted-artifacts>
  <trust file=".*-javadoc[.]jar" regex="true"/>
  <trust file=".*-sources[.]jar" regex="true"/>
</trusted-artifacts>
```

Adding keys manually to the keyring

Adding keys to the ASCII-armored keyring

The added key must be ASCII-armored formatted and can be simply added at the end of the file. If you already downloaded the key in the right format, you can simply append it to the file.

Or you can amend an existing KEYS file by issuing the following commands:

```
$ gpg --no-default-keyring --keyring /tmp/keyring.gpg --recv-keys
8756c4f765c9ac3cb6b85d62379ce192d401ab61

gpg: keybox '/tmp/keyring.gpg' created
gpg: key 379CE192D401AB61: public key "Bintray (by JFrog) <****>" imported
gpg: Total number processed: 1
gpg:                imported: 1

# Write its ASCII-armored version
$ gpg --keyring /tmp/keyring.gpg --export --armor
8756c4f765c9ac3cb6b85d62379ce192d401ab61 > gradle/verification-keyring.keys
```

Once done, make sure to run [the generation command again](#) so that the key is processed by Gradle. This will do the following:

- Add a standard header to the key
- Rewrite the key using Gradle's own format, which trims the key to the bare minimum
- Move the key to its sorted location, keeping the file reproducible

Adding keys to the binary keyring

You can add keys to the binary version using GPG, for example issuing the following commands (syntax may depend on the tool you use):

```
$ gpg --no-default-keyring --keyring gradle/verification-keyring.gpg --recv-keys
8756c4f765c9ac3cb6b85d62379ce192d401ab61

gpg: keybox 'gradle/verification-keyring.gpg' created
gpg: key 379CE192D401AB61: public key "Bintray (by JFrog) <****>" imported
gpg: Total number processed: 1
gpg:                imported: 1

$ gpg --no-default-keyring --keyring gradle/verification-keyring.gpg --recv-keys
```

```
6f538074ccebf35f28af9b066a0975f8b1127b83
```

```
gpg: key 0729A0AFF8999A87: public key "Kotlin Release <****>" imported
gpg: Total number processed: 1
gpg:                imported: 1
```

Dealing with a verification failure

Dependency verification can fail in different ways, this section explains how you should deal with the various cases.

Missing verification metadata

The simplest failure you can have is when verification metadata is missing from the dependency verification file. This is the case for example if you use [checksum verification](#), then you update a dependency and new versions of the dependency (and potentially its transitive dependencies) are brought in.

Gradle will tell you what metadata is missing:

```
Execution failed for task ':compileJava'.
> Dependency verification failed for configuration ':compileClasspath':
   - On artifact commons-logging-1.2.jar (commons-logging:commons-logging:1.2) in
     repository 'MavenRepo': checksum is missing from verification metadata.
```

- the missing module group is `commons-logging`, it's artifact name is `commons-logging` and its version is `1.2`. The corresponding artifact is `commons-logging-1.2.jar` so you need to add the following entry to the verification file:

```
<component group="commons-logging" name="commons-logging" version="1.2">
  <artifact name="commons-logging-1.2.jar">
    <sha256 value="daddea1ea0be0f56978ab3006b8ac92834afeefbd9b7e4e6316fca57df0fa636"
origin="official distribution"/>
  </artifact>
</component>
```

Alternatively, you can ask Gradle to generate the missing information by using the [bootstrapping mechanism](#): existing information in the metadata file will be preserved, Gradle will only add the missing verification metadata.

Incorrect checksums

A more problematic issue is when the actual checksum verification fails:

```
Execution failed for task ':compileJava'.
> Dependency verification failed for configuration ':compileClasspath':
   - On artifact commons-logging-1.2.jar (commons-logging:commons-logging:1.2) in
```

```
repository 'MavenRepo': expected a 'sha256' checksum of
'91f7a33096ea69bac2cbaf6d01feb934cac002c48d8c8cfa9c240b40f1ec21df' but was
'daddea1ea0be0f56978ab3006b8ac92834afeefbd9b7e4e6316fca57df0fa636'
```

This time, Gradle tells you what dependency is at fault, what was the expected checksum (the one you declared in the verification metadata file) and the one which was actually computed during verification.

Such a failure indicates that a **dependency may have been compromised**. At this stage, you **must** perform manual verification and check what happens. Several things can happen:

- a dependency was tampered in the local dependency cache of Gradle. This is usually harmless: erase the file from the cache and Gradle would redownload the dependency.
- a dependency is available in multiple sources with slightly different binaries (additional whitespace, ...)
 - please inform the maintainers of the library that they have such an issue
 - you can use `also-trust` to accept the additional checksums
- the dependency was compromised
 - immediately inform the maintainers of the library
 - notify the repository maintainers of the compromised library

Note that a variation of a compromised library is often *name squatting*, when a hacker would use GAV coordinates which *look legit* but are actually different by one character, or *repository shadowing*, when a dependency with the official GAV coordinates is published in a malicious repository which comes first in your build.

Untrusted signatures

If you have signature verification enabled, Gradle will perform verification of the signatures but will not trust them automatically:

```
> Dependency verification failed for configuration ':compileClasspath':
  - On artifact javaparser-core-3.6.11.jar (com.github.javaparser:javaparser-
core:3.6.11) in repository 'MavenRepo': Artifact was signed with key
'379ce192d401ab61' (Bintray (by JFrog) <***>) and passed verification but the key
isn't in your trusted keys list.
```

In this case it means you need to check yourself if the key that was used for verification (and therefore the signature) can be trusted, in which case refer to [this section of the documentation](#) to figure out how to declare trusted keys.

Failed signature verification

If Gradle fails to verify a signature, you will need to take action and verify artifacts manually because this **may indicate a compromised dependency**.

If such a thing happens, Gradle will fail with:

```
> Dependency verification failed for configuration ':compileClasspath':
  - On artifact javaparser-core-3.6.11.jar (com.github.javaparser:javaparser-
core:3.6.11) in repository 'MavenRepo': Artifact was signed with key
'379ce192d401ab61' (Bintray (by JFrog) <****>) but signature didn't match
```

There are several options:

1. signature was wrong in the first place, which happens frequently with [dependencies published on different repositories](#).
2. the signature is correct but the artifact has been compromised (either in the local dependency cache or remotely)

The right approach here is to go to the official site of the dependency and see if they publish signatures for their artifacts. If they do, verify that the signature that Gradle downloaded matches the one published.

If you have [checked that the dependency is *not* compromised](#) and that it's "only" the signature which is wrong, you should declare an *artifact level key exclusion*:

```
<components>
  <component group="com.github.javaparser" name="javaparser-core" version="
3.6.11">
    <artifact name="javaparser-core-3.6.11.pom">
      <ignored-keys>
        <ignored-key id="379ce192d401ab61" reason="internal repo has corrupted
POM"/>
      </ignored-keys>
    </artifact>
  </component>
</components>
```

However, if you only do so, Gradle will still fail because all keys for this artifact will be ignored and you didn't provide a checksum:

```
<components>
  <component group="com.github.javaparser" name="javaparser-core" version="
3.6.11">
    <artifact name="javaparser-core-3.6.11.pom">
      <ignored-keys>
        <ignored-key id="379ce192d401ab61" reason="internal repo has corrupted
POM"/>
      </ignored-keys>
      <sha256 value=
"a2023504cfd611332177f96358b6f6db26e43d96e8ef4cff59b0f5a2bee3c1e1"/>
    </artifact>
```

```
</component>  
</components>
```

Manual verification of a dependency

You will likely face a dependency verification failure (either checksum verification or signature verification) and will need to figure out if the dependency has been compromised or not.

In this section we give *an example* how you can manually check if a dependency was compromised.

For this we will take this example failure:

```
> Dependency verification failed for configuration ':compileClasspath':  
- On artifact j2objc-annotations-1.1.jar (com.google.j2objc:j2objc-annotations:1.1) in  
repository 'MyCompany Mirror': Artifact was signed with key '29579f18fa8fd93b' but  
signature didn't match
```

This error message gives us the GAV coordinates of the problematic dependency, as well as an indication of where the dependency was fetched from. Here, the dependency comes from **MyCompany Mirror**, which is a repository declared in our build.

The first thing to do is therefore to download the artifact and its signature manually from the mirror:

```
$ curl https://my-company-mirror.com/repo/com/google/j2objc/j2objc-  
annotations/1.1/j2objc-annotations-1.1.jar --output j2objc-annotations-1.1.jar  
$ curl https://my-company-mirror.com/repo/com/google/j2objc/j2objc-  
annotations/1.1/j2objc-annotations-1.1.jar.asc --output j2objc-annotations-1.1.jar.asc
```

Then we can use the key information provided in the error message to import the key locally:

```
$ gpg --recv-keys B801E2F8EF035068EC1139CC29579F18FA8FD93B
```

And perform verification:

```
$ gpg --verify j2objc-annotations-1.1.jar.asc  
gpg: assuming signed data in 'j2objc-annotations-1.1.jar'  
gpg: Signature made Thu 19 Jan 2017 12:06:51 AM CET  
gpg:                using RSA key 29579F18FA8FD93B  
gpg: BAD signature from "Tom Ball <****>" [unknown]
```

What this tells us is that the problem is *not* on the local machine: the repository *already contains a bad signature*.

The next step is to do the same by downloading what is actually on Maven Central:

```
$ curl https://my-company-mirror.com/repo/com/google/j2objc/j2objc-
annotations/1.1/j2objc-annotations-1.1.jar --output central-j2objc-annotations-
1.1.jar
$ curl https://my-company-mirror.com/repo/com/google/j2objc/j2objc-
annotations/1.1/j2objc-annotations-1.1.jar.asc --output central-j2objc-annotations-
1.1.jar.asc
```

And we can now check the signature again:

```
$ gpg --verify central-j2objc-annotations-1.1.jar.asc

gpg: assuming signed data in 'central-j2objc-annotations-1.1.jar'
gpg: Signature made Thu 19 Jan 2017 12:06:51 AM CET
gpg:                using RSA key 29579F18FA8FD93B
gpg: Good signature from "Tom Ball <****>" [unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg:                There is no indication that the signature belongs to the owner.
Primary key fingerprint: B801 E2F8 EF03 5068 EC11 39CC 2957 9F18 FA8F D93B
```

This indicates that the dependency is *valid* on Maven Central. At this stage, we already know that the problem lives in the mirror, it *may* have been compromised, but we need to verify.

A good idea is to compare the 2 artifacts, which you can do with a tool like [diffoscope](#).

We then figure out that the intent wasn't malicious but that somehow a build has been overwritten with a newer version (the version in Central is newer than the one in our repository).

In this case, you can decide to:

- ignore the signature for this artifact and trust the different possible checksums (both for the old artifact and the new version)
- or cleanup your mirror so that it contains the same version as in Maven Central

It's worth noting that if you choose to delete the version from your repository, you will *also* need to remove it from the local Gradle cache.

This is facilitated by the fact the error message tells you where the file is located:

```
> Dependency verification failed for configuration ':compileClasspath':
- On artifact j2objc-annotations-1.1.jar (com.google.j2objc:j2objc-
annotations:1.1) in repository 'MyCompany Mirror': Artifact was signed with key
'29579f18fa8fd93b' but signature didn't match
```

This can indicate that a dependency has been compromised. Please carefully verify the signatures and checksums.

For your information here are the path to the files which failed verification:

```
- $<<directory_layout.adoc#dir:gradle_user_home,GRADLE_USER_HOME>>/caches/modules-
```

```
2/files-2.1/com.google.j2objc/j2objc-
annotations/1.1/976d8d30bebc251db406f2bdb3eb01962b5685b3/j2objc-annotations-1.1.jar
(signature: GRADLE_USER_HOME/caches/modules-2/files-2.1/com.google.j2objc/j2objc-
annotations/1.1/82e922e14f57d522de465fd144ec26eb7da44501/j2objc-annotations-
1.1.jar.asc)
```

```
GRADLE_USER_HOME = /home/jiraya/.gradle
```

You can safely delete the artifact file as Gradle would automatically re-download it:

```
rm -rf ~/.gradle/caches/modules-2/files-2.1/com.google.j2objc/j2objc-annotations/1.1
```

Cleaning up the verification file

If you do nothing, the dependency verification metadata will grow over time as you add new dependencies or change versions: Gradle will not automatically remove *unused* entries from this file. The reason is that there's no way for Gradle to know upfront if a dependency will effectively be used during the build or not.

As a consequence, adding dependencies or changing dependency version can easily lead to more entries in the file, while leaving unnecessary entries out there.

One option to cleanup the file is to move the existing `verification-metadata.xml` file to a different location and call Gradle with the `--dry-run` mode: while not perfect (it will not notice dependencies only resolved at configuration time), it generates *a new file* that you can compare with the existing one.

We need to move the existing file because both the bootstrapping mode and the dry-run mode are incremental: they copy information from the existing metadata verification file (in particular, trusted keys).

Refreshing missing keys

Gradle caches missing keys for 24 hours, meaning it will not attempt to re-download the missing keys for 24 hours after failing.

If you want to retry immediately, you can run with the `--refresh-keys` CLI flag:

```
./gradlew build --refresh-keys
```

See [here](#) how to manually add keys if Gradle keeps failing to download them.

Aligning dependency versions

Dependency version alignment allows different modules belonging to the same logical group (a *platform*) to have identical versions in a dependency graph.

Handling inconsistent module versions

Gradle supports aligning versions of modules which belong to the same "platform". It is often preferable, for example, that the API and implementation modules of a component are using the same version. However, because of the game of transitive dependency resolution, it is possible that different modules belonging to the same platform end up using different versions. For example, your project may depend on the `jackson-databind` and `vert.x` libraries, as illustrated below:

Example 57. *Declaring dependencies*

build.gradle.kts

```
dependencies {  
    // a dependency on Jackson Databind  
    implementation("com.fasterxml.jackson.core:jackson-databind:2.8.9")  
  
    // and a dependency on vert.x  
    implementation("io.vertx:vertx-core:3.5.3")  
}
```

build.gradle

```
dependencies {  
    // a dependency on Jackson Databind  
    implementation 'com.fasterxml.jackson.core:jackson-databind:2.8.9'  
  
    // and a dependency on vert.x  
    implementation 'io.vertx:vertx-core:3.5.3'  
}
```

Because `vert.x` depends on `jackson-core`, we would actually resolve the following dependency versions:

- `jackson-core` version `2.9.5` (brought by `vertx-core`)
- `jackson-databind` version `2.9.5` (by conflict resolution)
- `jackson-annotation` version `2.9.0` (dependency of `jackson-databind:2.9.5`)

It's easy to end up with a set of versions which do not work well together. To fix this, Gradle supports dependency version alignment, which is supported by the concept of platforms. A platform represents a set of modules which "work well together". Either because they are actually published as a whole (when one of the members of the platform is published, all other modules are also published with the same version), or because someone tested the modules and indicates that they work well together (typically, the Spring Platform).

Aligning versions natively with Gradle

Gradle natively supports alignment of modules produced by Gradle. This is a direct consequence of the transitivity of [dependency constraints](#). So if you have a multi-project build, and you wish that consumers get the same version of all your modules, Gradle provides a simple way to do this using the [Java Platform Plugin](#).

For example, if you have a project that consists of 3 modules:

- `lib`
- `utils`
- `core`, depending on `lib` and `utils`

And a consumer that declares the following dependencies:

- `core` version 1.0
- `lib` version 1.1

Then by default resolution would select `core:1.0` and `lib:1.1`, because `lib` has no dependency on `core`. We can fix this by adding a new module in our project, a *platform*, that will add constraints on all the modules of your project:

Example 58. The platform module

build.gradle.kts

```
plugins {
    `java-platform`
}

dependencies {
    // The platform declares constraints on all components that
    // require alignment
    constraints {
        api(project(":core"))
        api(project(":lib"))
        api(project(":utils"))
    }
}
```

build.gradle

```
plugins {
    id 'java-platform'
}

dependencies {
```

```
// The platform declares constraints on all components that
// require alignment
constraints {
    api(project(":core"))
    api(project(":lib"))
    api(project(":utils"))
}
}
```

Once this is done, we need to make sure that all modules now *depend on the platform*, like this:

Example 59. Declaring a dependency on the platform

build.gradle.kts

```
dependencies {
    // Each project has a dependency on the platform
    api(platform(project(":platform")))

    // And any additional dependency required
    implementation(project(":lib"))
    implementation(project(":utils"))
}
```

build.gradle

```
dependencies {
    // Each project has a dependency on the platform
    api(platform(project(":platform")))

    // And any additional dependency required
    implementation(project(":lib"))
    implementation(project(":utils"))
}
```

It is important that the platform contains a constraint on all the components, but also that each component has a dependency on the platform. By doing this, whenever Gradle will add a dependency to a module of the platform on the graph, it will *also* include constraints on the other modules of the platform. This means that if we see another module belonging to the same platform, we will automatically upgrade to the same version.

In our example, it means that we first see `core:1.0`, which brings a platform `1.0` with constraints on `lib:1.0` and `lib:1.0`. Then we add `lib:1.1` which has a dependency on `platform:1.1`. By conflict

resolution, we select the **1.1** platform, which has a constraint on **core:1.1**. Then we conflict resolve between **core:1.0** and **core:1.1**, which means that **core** and **lib** are now aligned properly.

NOTE

This behavior is enforced for published components only if you use Gradle Module Metadata.

Aligning versions of modules not published with Gradle

Whenever the publisher doesn't use Gradle, like in our Jackson example, we can explain to Gradle that all Jackson modules "belong to" the same platform and benefit from the same behavior as with native alignment. There are two options to express that a set of modules belong to a platform:

1. A platform is **published** as a **BOM** and can be used: For example, **com.fasterxml.jackson:jackson-bom** can be used as platform. The information missing to Gradle in that case is that the platform should be added to the dependencies if one of its members is used.
2. No existing platform can be used. Instead, a **virtual platform** should be created by Gradle: In this case, Gradle builds up the platform itself based on all the members that are used.

To provide the missing information to Gradle, you can define **component metadata rules** as explained in the following.

Align versions of modules using a published BOM

Example 60. *A dependency version alignment rule*

build.gradle.kts

```
abstract class JacksonBomAlignmentRule: ComponentMetadataRule {
    override fun execute(ctx: ComponentMetadataContext) {
        ctx.details.run {
            if (id.group.startsWith("com.fasterxml.jackson")) {
                // declare that Jackson modules belong to the platform
                defined by the Jackson BOM
                belongsTo("com.fasterxml.jackson:jackson-bom:${id.version}",
                false)
            }
        }
    }
}
```

build.gradle

```
abstract class JacksonBomAlignmentRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext ctx) {
        ctx.details.with {
```

```

        if (id.group.startsWith("com.fasterxml.jackson")) {
            // declare that Jackson modules belong to the platform
            defined by the Jackson BOM
            belongsTo("com.fasterxml.jackson:jackson-bom:${id.version}",
false)
        }
    }
}
}

```

By using the `belongsTo` with `false` (**not** virtual), we declare that all modules belong to the same *published platform*. In this case, the platform is `com.fasterxml.jackson:jackson-bom` and Gradle will look for it, as for any other module, in the declared repositories.

Example 61. Making use of a dependency version alignment rule

build.gradle.kts

```

dependencies {
    components.all<JacksonBomAlignmentRule>()
}

```

build.gradle

```

dependencies {
    components.all(JacksonBomAlignmentRule)
}

```

Using the rule, the versions in the example above align to whatever the selected version of `com.fasterxml.jackson:jackson-bom` defines. In this case, `com.fasterxml.jackson:jackson-bom:2.9.5` will be selected as `2.9.5` is the highest version of a module selected. In that BOM, the following versions are defined and will be used: `jackson-core:2.9.5`, `jackson-databind:2.9.5` and `jackson-annotation:2.9.0`. The lower versions of `jackson-annotation` here might be the desired result as it is what the BOM recommends.

NOTE

This behavior is working reliable since Gradle 6.1. Effectively, it is similar to a [component metadata rule](#) that adds a platform dependency to all members of the platform using `withDependencies`.

Align versions of modules without a published platform

Example 62. *A dependency version alignment rule*

build.gradle.kts

```
abstract class JacksonAlignmentRule: ComponentMetadataRule {
    override fun execute(ctx: ComponentMetadataContext) {
        ctx.details.run {
            if (id.group.startsWith("com.fasterxml.jackson")) {
                // declare that Jackson modules all belong to the Jackson
                virtual platform
                belongsTo("com.fasterxml.jackson:jackson-virtual-
platform:${id.version}")
            }
        }
    }
}
```

build.gradle

```
abstract class JacksonAlignmentRule implements ComponentMetadataRule {
    void execute(ComponentMetadataContext ctx) {
        ctx.details.with {
            if (id.group.startsWith("com.fasterxml.jackson")) {
                // declare that Jackson modules all belong to the Jackson
                virtual platform
                belongsTo("com.fasterxml.jackson:jackson-virtual-platform:
${id.version}")
            }
        }
    }
}
```

By using the **belongsTo** keyword without further parameter (platform is virtual), we declare that all modules belong to the same *virtual platform*, which is treated specially by the engine. A virtual platform will not be retrieved from a repository. The identifier, in this case **com.fasterxml.jackson:jackson-virtual-platform**, is something you as the build author define yourself. The "content" of the platform is then created by Gradle on the fly by collecting all **belongsTo** statements pointing at the same virtual platform.

Example 63. *Making use of a dependency version alignment rule*

build.gradle.kts

```
dependencies {
```

```
components.all<JacksonAlignmentRule>()  
}
```

build.gradle

```
dependencies {  
    components.all(JacksonAlignmentRule)  
}
```

Using the rule, all versions in the example above would align to **2.9.5**. In this case, also **jackson-annotation:2.9.5** will be taken, as that is how we defined our local virtual platform.

For both published and virtual platforms, Gradle lets you override the version choice of the platform itself by specifying an *enforced* dependency on the platform:

Example 64. Forceful platform downgrade

build.gradle.kts

```
dependencies {  
    // Forcefully downgrade the virtual Jackson platform to 2.8.9  
    implementation(enforcedPlatform("com.fasterxml.jackson:jackson-virtual-  
platform:2.8.9"))  
}
```

build.gradle

```
dependencies {  
    // Forcefully downgrade the virtual Jackson platform to 2.8.9  
    implementation enforcedPlatform('com.fasterxml.jackson:jackson-virtual-  
platform:2.8.9')  
}
```

Modeling library features

Gradle supports the concept of *features*: it's often the case that a single library can be split up into multiple related yet distinct libraries, where each *feature* can be used alongside the *main* library.

Features allow a component to expose multiple related libraries, each of which can declare its own dependencies. These libraries are exposed as variants, similar to how the *main* library exposes

variants for its API and runtime.

This allows for a number of different scenarios (list is non-exhaustive):

- a (better) substitute for [Maven optional dependencies](#)
- a *main* library is built with support for different mutually-exclusive implementations of runtime features; the [user must choose one, and only one, implementation of each such feature](#)
- a *main* library is built with support for optional runtime features, each of which requires a different set of dependencies
- a *main* library comes with supplementary features like *test fixtures*
- a *main* library comes with a main artifact, and enabling an additional feature requires additional artifacts

Selection of features via capabilities

Declaring a dependency on a component is usually done by providing a set of coordinates (group, artifact, version also known as GAV coordinates). This allows the engine to determine the *component* we're looking for, but such a component may provide different *variants*. A *variant* is typically chosen based on the usage. For example, we might choose a different variant for compiling against a component (in which case we need the API of the component) or when executing code (in which case we need the runtime of the component). All variants of a component provide a number of [capabilities](#), which are denoted similarly using GAV coordinates.

A capability is denoted by GAV coordinates, but you must think of it as feature description:

- "I provide an SLF4J binding"
- "I provide runtime support for MySQL"
- "I provide a Groovy runtime"

And in general, having two components that provide the *same thing* in the graph is a problem (they conflict).

This is an important concept because:

- By default, a variant provides a capability corresponding to the GAV coordinates of its component
- No two variants in a dependency graph can provide the same capability
- Multiple variants of a single component may be selected as long as they provide *different capabilities*

A typical component will **only** provide variants with the default capability. A Java library, for example, exposes two variants (API and runtime) which provide the *same capability*. As a consequence, it is an error to have both the *API* and *runtime* of a single component in a dependency graph.

However, imagine that you need the *runtime* and the *test fixtures runtime* of a component. Then it is allowed as long as the *runtime* and *test fixtures runtime* variant of the library declare different

capabilities.

If we do so, a consumer would then have to declare two dependencies:

- one on the "main" feature, the library
- one on the "test fixtures" feature, by *requiring its capability*

NOTE

While the resolution engine supports multi-variant components independently of the ecosystem, *features* are currently only available using the Java plugins.

Registering features

Features can be declared by applying the `java-library` plugin. The following code illustrates how to declare a feature named `mongodbSupport`:

Example 65. *Registering a feature*

build.gradle.kts

```
sourceSets {
    create("mongodbSupport") {
        java {
            srcDir("src/mongodb/java")
        }
    }
}

java {
    registerFeature("mongodbSupport") {
        usingSourceSet(sourceSets["mongodbSupport"])
    }
}
```

build.gradle

```
sourceSets {
    mongodbSupport {
        java {
            srcDir 'src/mongodb/java'
        }
    }
}

java {
    registerFeature('mongodbSupport') {
        usingSourceSet(sourceSets.mongodbSupport)
    }
}
```

```
}
```

Gradle will automatically set up a number of things for you, in a very similar way to how the [Java Library Plugin](#) sets up configurations.

Dependency scope configurations are created in the same manner as for the main feature:

- the configuration `mongodbSupportApi`, used to *declare API dependencies* for this feature
- the configuration `mongodbSupportImplementation`, used to *declare implementation dependencies* for this feature
- the configuration `mongodbSupportRuntimeOnly`, used to *declare runtime-only dependencies* for this feature
- the configuration `mongodbSupportCompileOnly`, used to *declare compile-only dependencies* for this feature
- the configuration `mongodbSupportCompileOnlyApi`, used to *declare compile-only API dependencies* for this feature

Furthermore, consumable configurations are created in the same manner as for the main feature:

- the configuration `mongodbSupportApiElements`, used by consumers to fetch the artifacts and API dependencies of this feature
- the configuration `mongodbSupportRuntimeElements`, used by consumers to fetch the artifacts and runtime dependencies of this feature

A feature should have a *source set* with the same name. Gradle will create a `Jar` task to bundle the classes built from the feature source set, using a classifier corresponding to the kebab-case name of the feature.

WARNING

Do not use the *main* source set when registering a feature. This behavior will be deprecated in a future version of Gradle.

Most users will only need to care about the dependency scope configurations, to declare the specific dependencies of this feature:

Example 66. Declaring dependencies of a feature

build.gradle.kts

```
dependencies {  
    "mongodbSupportImplementation"("org.mongodb:mongodb-driver-sync:3.9.1")  
}
```

build.gradle

```
dependencies {  
    mongodbSupportImplementation 'org.mongodb:mongodb-driver-sync:3.9.1'  
}
```

By convention, Gradle maps the feature name to a capability whose group and version are the same as the group and version of the main component, respectively, but whose name is the main component name followed by a `-` followed by the kebab-cased feature name.

For example, if the component's group is `org.gradle.demo`, its name is `provider`, its version is `1.0`, and the feature is named `mongodbSupport`, the feature's variants will have the `org.gradle.demo:provider-mongodb-support:1.0` capability.

If you choose the capability name yourself or add more capabilities to a variant, it is recommended to follow the same convention.

Publishing features

Depending on the metadata file format, publishing features may be lossy:

- using [Gradle Module Metadata](#), everything is published and consumers will get the full benefit of features
- using POM metadata (Maven), features are published as **optional dependencies** and artifacts of features are published with different *classifiers*
- using Ivy metadata, features are published as extra configurations, which are *not* extended by the `default` configuration

Publishing features is supported using the `maven-publish` and `ivy-publish` plugins only. The Java Library Plugin will take care of registering the additional variants for you, so there's no additional configuration required, only the regular publications:

Example 67. [Publishing a component with features](#)

build.gradle.kts

```
plugins {  
    `java-library`  
    `maven-publish`  
}  
// ...  
publishing {  
    publications {  
        create("myLibrary", MavenPublication::class.java) {  
            from(components["java"])  
        }  
    }  
}
```

```
}  
}
```

build.gradle

```
plugins {  
    id 'java-library'  
    id 'maven-publish'  
}  
// ...  
publishing {  
    publications {  
        myLibrary(MavenPublication) {  
            from components.java  
        }  
    }  
}
```

Adding javadoc and sources JARs

Similar to the [main Javadoc and sources JARs](#), you can configure the added feature so that it produces JARs for the Javadoc and sources.

Example 68. Producing javadoc and sources JARs for features

build.gradle.kts

```
java {  
    registerFeature("mongodbSupport") {  
        usingSourceSet(sourceSets["mongodbSupport"])  
        withJavadocJar()  
        withSourcesJar()  
    }  
}
```

build.gradle

```
java {  
    registerFeature('mongodbSupport') {  
        usingSourceSet(sourceSets.mongodbSupport)  
        withJavadocJar()  
        withSourcesJar()  
    }  
}
```

```
}
```

Dependencies on features

As mentioned earlier, features can be lossy when published. As a consequence, a consumer can depend on a feature only in these cases:

- with a project dependency (in a multi-project build)
- with Gradle Module Metadata available, that is the publisher MUST have published it
- within the Ivy world, by declaring a dependency on the configuration matching the feature

A consumer can specify that it needs a specific feature of a producer by declaring required capabilities. For example, if a producer declares a "MySQL support" feature like this:

Example 69. A library declaring a feature to support MySQL

build.gradle.kts

```
group = "org.gradle.demo"

sourceSets {
    create("mysqlSupport") {
        java {
            srcDir("src/mysql/java")
        }
    }
}

java {
    registerFeature("mysqlSupport") {
        usingSourceSet(sourceSets["mysqlSupport"])
    }
}

dependencies {
    "mysqlSupportImplementation"("mysql:mysql-connector-java:8.0.14")
}
```

build.gradle

```
group = 'org.gradle.demo'

sourceSets {
    mysqlSupport {
        java {
```

```

        srcDir 'src/mysql/java'
    }
}

java {
    registerFeature('mysqlSupport') {
        usingSourceSet(sourceSets.mysqlSupport)
    }
}

dependencies {
    mysqlSupportImplementation 'mysql:mysql-connector-java:8.0.14'
}

```

Then the consumer can declare a dependency on the MySQL support feature by doing this:

Example 70. Consuming specific features in a multi-project build

build.gradle.kts

```

dependencies {
    // This project requires the main producer component
    implementation(project(":producer"))

    // But we also want to use its MySQL support
    runtimeOnly(project(":producer")) {
        capabilities {
            requireCapability("org.gradle.demo:producer-mysql-support")
        }
    }
}

```

build.gradle

```

dependencies {
    // This project requires the main producer component
    implementation(project(":producer"))

    // But we also want to use its MySQL support
    runtimeOnly(project(":producer")) {
        capabilities {
            requireCapability("org.gradle.demo:producer-mysql-support")
        }
    }
}

```

```
}
```

This will automatically bring the `mysql-connector-java` dependency on the runtime classpath. If there were more than one dependency, all of them would be brought, meaning that a feature can be used to group dependencies which contribute to a feature together.

Similarly, if an external library with features was published with [Gradle Module Metadata](#), it is possible to depend on a feature provided by that library:

Example 71. Consuming specific features from an external repository

build.gradle.kts

```
dependencies {
    // This project requires the main producer component
    implementation("org.gradle.demo:producer:1.0")

    // But we also want to use its MongoDB support
    runtimeOnly("org.gradle.demo:producer:1.0") {
        capabilities {
            requireCapability("org.gradle.demo:producer-mongodb-support")
        }
    }
}
```

build.gradle

```
dependencies {
    // This project requires the main producer component
    implementation('org.gradle.demo:producer:1.0')

    // But we also want to use its MongoDB support
    runtimeOnly('org.gradle.demo:producer:1.0') {
        capabilities {
            requireCapability("org.gradle.demo:producer-mongodb-support")
        }
    }
}
```

Handling mutually exclusive variants

The main advantage of using *capabilities* as a way to handle features is that you can precisely handle compatibility of variants. The rule is simple:

No two variants in a dependency graph can provide the same capability

We can leverage this to ensure that Gradle fails whenever the user mis-configures dependencies. Consider a situation where your library supports MySQL, Postgres and MongoDB, but that it's only allowed to choose *one* of those at the same time. We can model this restriction by ensuring each feature also provides the same capability, thus making it impossible for these features to be used together in the same graph.

Example 72. A producer of multiple features that are mutually exclusive

build.gradle.kts

```
java {
    registerFeature("mysqlSupport") {
        usingSourceSet(sourceSets["mysqlSupport"])
        capability("org.gradle.demo", "producer-db-support", "1.0")
        capability("org.gradle.demo", "producer-mysql-support", "1.0")
    }
    registerFeature("postgresSupport") {
        usingSourceSet(sourceSets["postgresSupport"])
        capability("org.gradle.demo", "producer-db-support", "1.0")
        capability("org.gradle.demo", "producer-postgres-support", "1.0")
    }
    registerFeature("mongoSupport") {
        usingSourceSet(sourceSets["mongoSupport"])
        capability("org.gradle.demo", "producer-db-support", "1.0")
        capability("org.gradle.demo", "producer-mongo-support", "1.0")
    }
}

dependencies {
    "mysqlSupportImplementation"("mysql:mysql-connector-java:8.0.14")
    "postgresSupportImplementation"("org.postgresql:postgresql:42.2.5")
    "mongoSupportImplementation"("org.mongodb:mongodb-driver-sync:3.9.1")
}
```

build.gradle

```
java {
    registerFeature('mysqlSupport') {
        usingSourceSet(sourceSets.mysqlSupport)
        capability('org.gradle.demo', 'producer-db-support', '1.0')
        capability('org.gradle.demo', 'producer-mysql-support', '1.0')
    }
    registerFeature('postgresSupport') {
        usingSourceSet(sourceSets.postgresSupport)
        capability('org.gradle.demo', 'producer-db-support', '1.0')
    }
}
```

```

        capability('org.gradle.demo', 'producer-postgres-support', '1.0')
    }
    registerFeature('mongoSupport') {
        usingSourceSet(sourceSets.mongoSupport)
        capability('org.gradle.demo', 'producer-db-support', '1.0')
        capability('org.gradle.demo', 'producer-mongo-support', '1.0')
    }
}

dependencies {
    mysqlSupportImplementation 'mysql:mysql-connector-java:8.0.14'
    postgresSupportImplementation 'org.postgresql:postgresql:42.2.5'
    mongoSupportImplementation 'org.mongodb:mongodb-driver-sync:3.9.1'
}

```

Here, the producer declares 3 features, one for each database runtime support:

- `mysql-support` provides both the `db-support` and `mysql-support` capabilities
- `postgres-support` provides both the `db-support` and `postgres-support` capabilities
- `mongo-support` provides both the `db-support` and `mongo-support` capabilities

Then if the consumer tries to get both the `postgres-support` and `mysql-support` features (this also works transitively):

Example 73. A consumer trying to use 2 incompatible variants at the same time

build.gradle.kts

```

dependencies {
    // This project requires the main producer component
    implementation(project(":producer"))

    // Let's try to ask for both MySQL and Postgres support
    runtimeOnly(project(":producer")) {
        capabilities {
            requireCapability("org.gradle.demo:producer-mysql-support")
        }
    }
    runtimeOnly(project(":producer")) {
        capabilities {
            requireCapability("org.gradle.demo:producer-postgres-support")
        }
    }
}

```

build.gradle

```
dependencies {
    implementation(project(":producer"))

    // Let's try to ask for both MySQL and Postgres support
    runtimeOnly(project(":producer")) {
        capabilities {
            requireCapability("org.gradle.demo:producer-mysql-support")
        }
    }
    runtimeOnly(project(":producer")) {
        capabilities {
            requireCapability("org.gradle.demo:producer-postgres-support")
        }
    }
}
```

Dependency resolution would fail with the following error:

```
Cannot choose between
  org.gradle.demo:producer:1.0 variant mysqlSupportRuntimeElements and
  org.gradle.demo:producer:1.0 variant postgresSupportRuntimeElements
because they provide the same capability: org.gradle.demo:producer-db-support:1.0
```

PLATFORMS

JVM BUILDS

Building Java & JVM projects

Gradle uses a convention-over-configuration approach to building JVM-based projects that borrows several conventions from Apache Maven. In particular, it uses the same default directory structure for source files and resources, and it works with Maven-compatible repositories.

We will look at Java projects in detail in this chapter, but most of the topics apply to other supported JVM languages as well, such as [Kotlin](#), [Groovy](#) and [Scala](#). If you don't have much experience with building JVM-based projects with Gradle, take a look at the [Java samples](#) for step-by-step instructions on how to build various types of basic Java projects.

NOTE

The example in this section use the Java Library Plugin. However the described features are shared by all JVM plugins. Specifics of the different plugins are available in their dedicated documentation.

TIP

There are a number of hands-on samples that you can explore for [Java](#), [Groovy](#), [Scala](#) and [Kotlin](#).

Introduction

The simplest build script for a Java project applies the [Java Library Plugin](#) and optionally sets the project version and selects the [Java toolchain](#) to use:

Example 74. [Applying the Java Library Plugin](#)

build.gradle.kts

```
plugins {
    `java-library`
}

java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(17)
    }
}

version = "1.2.1"
```

build.gradle

```
plugins {
    id 'java-library'
```

```
}

java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(17)
    }
}

version = '1.2.1'
```

By applying the Java Library Plugin, you get a whole host of features:

- A `compileJava` task that compiles all the Java source files under `src/main/java`
- A `compileTestJava` task for source files under `src/test/java`
- A `test` task that runs the tests from `src/test/java`
- A `jar` task that packages the `main` compiled classes and resources from `src/main/resources` into a single JAR named `<project>-<version>.jar`
- A `javadoc` task that generates Javadoc for the `main` classes

This isn't sufficient to build any non-trivial Java project — at the very least, you'll probably have some file dependencies. But it means that your build script only needs the information that is specific to *your* project.

NOTE

Although the properties in the example are optional, we recommend that you specify them in your projects. Configuring the toolchain protects against problems with the project being built with different Java versions. The version string is important for tracking the progression of the project. The project version is also used in archive names by default.

The Java Library Plugin also integrates the above tasks into the standard [Base Plugin lifecycle tasks](#):

- `jar` is attached to `assemble`
- `test` is attached to `check`

The rest of the chapter explains the different avenues for customizing the build to your requirements. You will also see later how to adjust the build for libraries, applications, web apps and enterprise apps.

Declaring your source files via source sets

Gradle's Java support was the first to introduce a new concept for building source-based projects: *source sets*. The main idea is that source files and resources are often logically grouped by type, such as application code, unit tests and integration tests. Each logical group typically has its own sets of file dependencies, classpaths, and more. Significantly, the files that form a source set *don't have to be located in the same directory*!

Source sets are a powerful concept that tie together several aspects of compilation:

- the source files and where they're located
- the compilation classpath, including any required dependencies (via Gradle [configurations](#))
- where the compiled class files are placed

You can see how these relate to one another in this diagram:

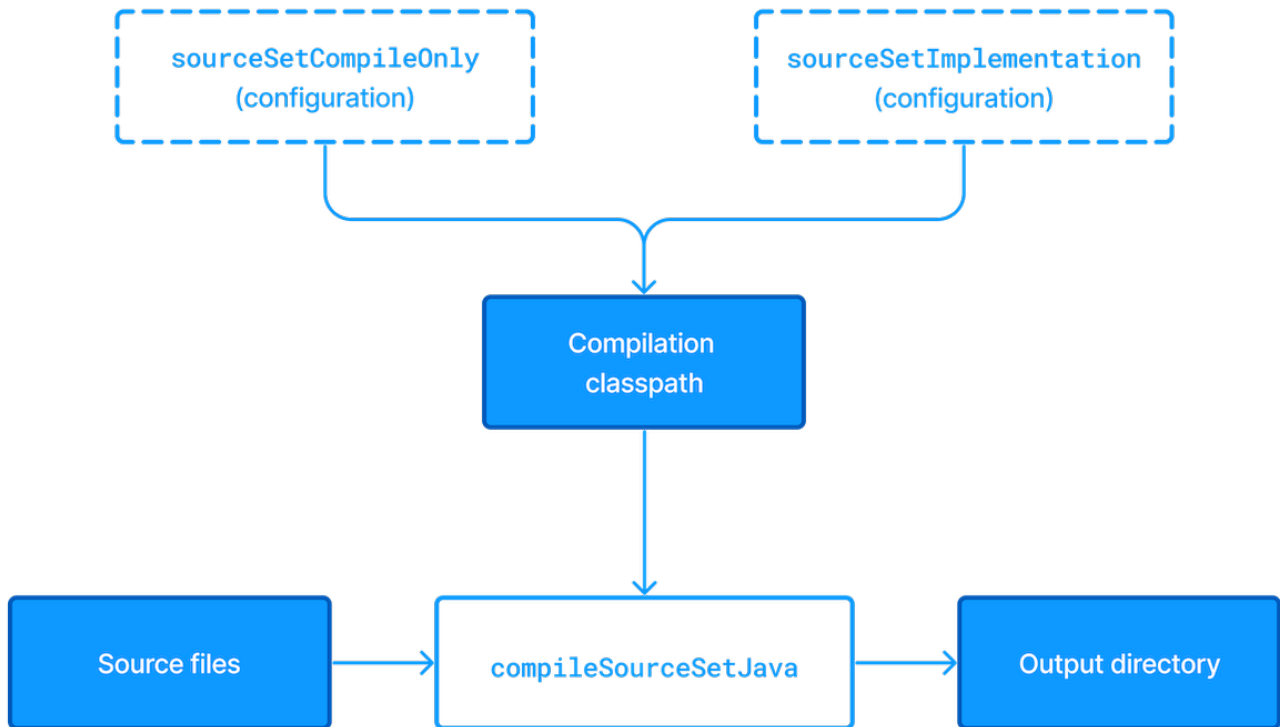


Figure 18. Source sets and Java compilation

The shaded boxes represent properties of the source set itself. On top of that, the Java Library Plugin automatically creates a compilation task for every source set you or a plugin defines — named `compileSourceSetJava` — and several [dependency configurations](#).

The `main` source set

Most language plugins, Java included, automatically create a source set called `main`, which is used for the project's production code. This source set is special in that its name is not included in the names of the configurations and tasks, hence why you have just a `compileJava` task and `compileOnly` and `implementation` configurations rather than `compileMainJava`, `mainCompileOnly` and `mainImplementation` respectively.

Java projects typically include resources other than source files, such as properties files, that may need processing — for example by replacing tokens within the files — and packaging within the final JAR. The Java Library Plugin handles this by automatically creating a dedicated task for each defined source set called `processSourceSetResources` (or `processResources` for the `main` source set). The following diagram shows how the source set fits in with this task:



Figure 19. Processing non-source files for a source set

As before, the shaded boxes represent properties of the source set, which in this case comprises the locations of the resource files and where they are copied to.

In addition to the `main` source set, the Java Library Plugin defines a `test` source set that represents the project's tests. This source set is used by the `test` task, which runs the tests. You can learn more about this task and related topics in the [Java testing](#) chapter.

Projects typically use this source set for unit tests, but you can also use it for integration, acceptance and other types of test if you wish. The alternative approach is to [define a new source set](#) for each of your other test types, which is typically done for one or both of the following reasons:

- You want to keep the tests separate from one another for aesthetics and manageability
- The different test types require different compilation or runtime classpaths or some other difference in setup

You can see an example of this approach in the Java testing chapter, which shows you [how to set up integration tests](#) in a project.

You'll learn more about source sets and the features they provide in:

- [Customizing file and directory locations](#)
- [Configuring Java integration tests](#)

Source set configurations

When a source set is created, it also creates a number of configurations as described above. Build logic should **not** attempt to create or access these configurations until they are first created by the source set.

When creating a source set, if one of these automatically created configurations already exists, Gradle will emit a deprecation warning. If the existing configuration's role is different than the role that the source set would have assigned, its role will be mutated to the correct value and another deprecation warning will be emitted.

The build below demonstrates this unwanted behavior.

Example 75. [Configurations created prior to their associated source sets](#)

build.gradle.kts

```
configurations {  
    val myCodeCompileClasspath: Configuration by creating  
}
```

```
sourceSets {  
    val myCode: SourceSet by creating  
}
```

build.gradle

```
configurations {  
    myCodeCompileClasspath  
}  
  
sourceSets {  
    myCode  
}
```

In this case, the following deprecation warning is emitted:

When creating configurations during sourceSet custom setup, Gradle found that configuration customCompileClasspath already exists with permitted usage(s):
Consumable - this configuration can be selected by another project as a dependency
Resolvable - this configuration can be resolved by this project to a set of files
Declarable - this configuration can have dependencies added to it
Yet Gradle expected to create it with the usage(s):
Resolvable - this configuration can be resolved by this project to a set of files

Following two simple best practices will avoid this problem:

1. Don't create configurations with names that will be used by source sets, such as names ending in `Api`, `Implementation`, `ApiElements`, `CompileOnly`, `CompileOnlyApi`, `RuntimeOnly`, `RuntimeClasspath` or `RuntimeElements`. (This list is not exhaustive.)
2. Create any custom source sets prior to any custom configurations.

Remember that any time you reference a configuration within the `configurations` container - with or without supplying an initialization action - Gradle will create the configuration. Sometimes when using the Groovy DSL this creation is not obvious, as in the example below, where `myCustomConfiguration` is created prior to the call to `extendsFrom`.

Example 76. Custom Configuration creation in Groovy

build.gradle

```
configurations {  
    myCustomConfiguration.extendsFrom(implementation)
```

```
}
```

Managing your dependencies

The vast majority of Java projects rely on libraries, so managing a project's dependencies is an important part of building a Java project. Dependency management is a big topic, so we will focus on the basics for Java projects here. If you'd like to dive into the detail, check out the [introduction to dependency management](#).

Specifying the dependencies for your Java project requires just three pieces of information:

- Which dependency you need, such as a name and version
- What it's needed for, e.g. compilation or running
- Where to look for it

The first two are specified in a `dependencies {}` block and the third in a `repositories {}` block. For example, to tell Gradle that your project requires version 3.6.7 of [Hibernate](#) Core to compile and run your production code, and that you want to download the library from the Maven Central repository, you can use the following fragment:

Example 77. [Declaring dependencies](#)

build.gradle.kts

```
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    implementation("org.hibernate:hibernate-core:3.6.7.Final")  
}
```

build.gradle

```
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    implementation 'org.hibernate:hibernate-core:3.6.7.Final'  
}
```

The Gradle terminology for the three elements is as follows:

- *Repository* (ex: `mavenCentral()`) — where to look for the modules you declare as dependencies
- *Configuration* (ex: `implementation`) — a named collection of dependencies, grouped together for a specific goal such as compiling or running a module — a more flexible form of Maven scopes
- *Module coordinate* (ex: `org.hibernate:hibernate-core-3.6.7.Final`) — the ID of the dependency, usually in the form '`<group>:<module>:<version>`' (or '`<groupId>:<artifactId>:<version>`' in Maven terminology)

You can find a more comprehensive glossary of dependency management terms [here](#).

As far as configurations go, the main ones of interest are:

- `compileOnly` — for dependencies that are necessary to compile your production code but shouldn't be part of the runtime classpath
- `implementation` (supersedes `compile`) — used for compilation and runtime
- `runtimeOnly` (supersedes `runtime`) — only used at runtime, not for compilation
- `testCompileOnly` — same as `compileOnly` except it's for the tests
- `testImplementation` — test equivalent of `implementation`
- `testRuntimeOnly` — test equivalent of `runtimeOnly`

You can learn more about these and how they relate to one another in the [plugin reference chapter](#).

Be aware that the [Java Library Plugin](#) offers two additional configurations — `api` and `compileOnlyApi` — for dependencies that are required for compiling both the module and any modules that depend on it.

Why no `compile` configuration?

The Java Library Plugin has historically used the `compile` configuration for dependencies that are required to both compile and run a project's production code. It is now deprecated, and will issue warnings when used, because it doesn't distinguish between dependencies that impact the public API of a Java library project and those that don't. You can learn more about the importance of this distinction in [Building Java libraries](#).

We have only scratched the surface here, so we recommend that you read the dedicated dependency management chapters once you're comfortable with the basics of building Java projects with Gradle. Some common scenarios that require further reading include:

- Defining a custom [Maven-](#) or [Ivy-compatible](#) repository
- Using dependencies from a [local filesystem directory](#)
- Declaring dependencies with [changing](#) (e.g. SNAPSHOT) and [dynamic](#) (range) versions
- Declaring a sibling [project as a dependency](#)
- [Controlling transitive dependencies and their versions](#)
- Testing your fixes to a 3rd-party dependency via [composite builds](#) (a better alternative to publishing to and consuming from [Maven Local](#))

You'll discover that Gradle has a rich API for working with dependencies — one that takes time to master, but is straightforward to use for common scenarios.

Compiling your code

Compiling both your production and test code can be trivially easy if you follow the conventions:

1. Put your production source code under the `src/main/java` directory
2. Put your test source code under `src/test/java`
3. Declare your production compile dependencies in the `compileOnly` or `implementation` configurations (see previous section)
4. Declare your test compile dependencies in the `testCompileOnly` or `testImplementation` configurations
5. Run the `compileJava` task for the production code and `compileTestJava` for the tests

Other JVM language plugins, such as the one for [Groovy](#), follow the same pattern of conventions. We recommend that you follow these conventions wherever possible, but you don't have to. There are several options for customization, as you'll see next.

Customizing file and directory locations

Imagine you have a legacy project that uses an `src` directory for the production code and `test` for the test code. The conventional directory structure won't work, so you need to tell Gradle where to find the source files. You do that via source set configuration.

Each source set defines where its source code resides, along with the resources and the output directory for the class files. You can override the convention values by using the following syntax:

Example 78. [Declaring custom source directories](#)

build.gradle.kts

```
sourceSets {
    main {
        java {
            setSrcDirs(listOf("src"))
        }
    }

    test {
        java {
            setSrcDirs(listOf("test"))
        }
    }
}
```

build.gradle

```
sourceSets {  
    main {  
        java {  
            srcDirs = ['src']  
        }  
    }  
  
    test {  
        java {  
            srcDirs = ['test']  
        }  
    }  
}
```

Now Gradle will only search directly in *src* and *test* for the respective source code. What if you don't want to override the convention, but simply want to *add* an extra source directory, perhaps one that contains some third-party source code you want to keep separate? The syntax is similar:

Example 79. Declaring custom source directories additively

build.gradle.kts

```
sourceSets {  
    main {  
        java {  
            srcDir("thirdParty/src/main/java")  
        }  
    }  
}
```

build.gradle

```
sourceSets {  
    main {  
        java {  
            srcDir 'thirdParty/src/main/java'  
        }  
    }  
}
```

Crucially, we're using the *method* `srcDir()` here to append a directory path, whereas setting the `srcDirs` property replaces any existing values. This is a common convention in Gradle: setting a property replaces values, while the corresponding method appends values.

You can see all the properties and methods available on source sets in the DSL reference for [SourceSet](#) and [SourceDirectorySet](#). Note that `srcDirs` and `srcDir()` are both on [SourceDirectorySet](#).

Changing compiler options

Most of the compiler options are accessible through the corresponding task, such as `compileJava` and `compileTestJava`. These tasks are of type [JavaCompile](#), so read the task reference for an up-to-date and comprehensive list of the options.

For example, if you want to use a separate JVM process for the compiler and prevent compilation failures from failing the build, you can use this configuration:

Example 80. [Setting Java compiler options](#)

build.gradle.kts

```
tasks.compileJava {
    options.isIncremental = true
    options.isFork = true
    options.failOnError = false
}
```

build.gradle

```
compileJava {
    options.incremental = true
    options.fork = true
    options.failOnError = false
}
```

That's also how you can change the verbosity of the compiler, disable debug output in the byte code and configure where the compiler can find annotation processors.

Targeting a specific Java version

By default, Gradle will compile Java code to the language level of the JVM running Gradle. If you need to target a specific version of Java when compiling, Gradle provides multiple options:

1. Using [Java toolchains](#) is a preferred way to target a language version.
A toolchain uniformly handles compilation, execution and Javadoc generation, and it can be configured on the project level.

2. Using `release` property is possible starting from Java 10.
Selecting a Java release makes sure that compilation is done with the configured language level and against the JDK APIs from that Java version.
3. Using `sourceCompatibility` and `targetCompatibility` properties.
Although not generally advised, these options were historically used to configure the Java version during compilation.

Using toolchains

When Java code is compiled using a specific toolchain, the actual compilation is carried out by a compiler of the specified Java version. The compiler provides access to the language features and JDK APIs for the requested Java language version.

In the simplest case, the toolchain can be configured for a project using the `java` extension. This way, not only compilation benefits from it, but also other tasks such as `test` and `javadoc` will also consistently use the same toolchain.

build.gradle.kts

```
java {  
    toolchain {  
        languageVersion = JavaLanguageVersion.of(17)  
    }  
}
```

build.gradle

```
java {  
    toolchain {  
        languageVersion = JavaLanguageVersion.of(17)  
    }  
}
```

You can learn more about this in the [Java toolchains](#) guide.

Using Java release version

Setting the `release` flag ensures the specified language level is used regardless of which compiler actually performs the compilation. To use this feature, the compiler must support the requested release version. It is possible to specify an earlier release version while compiling with a more recent [toolchain](#).

Gradle supports using the `release` flag from Java 10. It can be configured on the compilation task as follows.

Example 81. *Setting Java release flag*

build.gradle.kts

```
tasks.compileJava {  
    options.release = 7  
}
```

build.gradle

```
compileJava {  
    options.release = 7  
}
```

The release flag provides guarantees similar to toolchains. It validates that the Java sources are not using language features introduced in later Java versions, and also that the code does not access APIs from more recent JDKs. The bytecode produced by the compiler also corresponds to the requested Java version, meaning that the compiled code cannot be executed on older JVMs.

The `release` option of the Java compiler was introduced in Java 9. However, using this option with Gradle is only possible starting with Java 10, due to a [bug in Java 9](#).

Using Java compatibility options

WARNING

Using compatibility properties can lead to runtime failures when executing compiled code due to weaker guarantees they provide. Instead, consider using [toolchains](#) or the [release](#) flag.

The `sourceCompatibility` and `targetCompatibility` options correspond to the Java compiler options `-source` and `-target`. They are considered a legacy mechanism for targeting a specific Java version. However, these options do not protect against the use of APIs introduced in later Java versions.

`sourceCompatibility`

Defines the language version of Java used in your source files.

`targetCompatibility`

Defines the minimum JVM version your code should run on, i.e. it determines the version of the bytecode generated by the compiler.

These options can be set per `JavaCompile` task, or on the `java { }` extension for all compile tasks, using properties with the same names.

Targeting Java 6 and Java 7

Gradle itself can only run on a JVM with Java version 8 or higher. However, Gradle still supports

compiling, testing, generating Javadocs and executing applications for Java 6 and Java 7. Java 5 and below are not supported.

NOTE If using Java 10+, leveraging the `release` flag might be an easier solution, see above.

To use Java 6 or Java 7, the following tasks need to be configured:

- `JavaCompile` task to fork and use the correct Java home
- `Javadoc` task to use the correct `javadoc` executable
- `Test` and the `JavaExec` task to use the correct `java` executable.

With the usage of Java toolchains, this can be done as follows:

Example 82. [Configuring Java 7 build](#)

build.gradle.kts

```
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(7)
    }
}
```

build.gradle

```
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(7)
    }
}
```

The only requirement is that Java 7 is installed and has to be either in [a location Gradle can detect automatically](#) or [explicitly configured](#).

Compiling independent sources separately

Most projects have at least two independent sets of sources: the production code and the test code. Gradle already makes this scenario part of its Java convention, but what if you have other sets of sources? One of the most common scenarios is when you have separate integration tests of some form or other. In that case, a custom source set may be just what you need.

You can see a complete example for setting up integration tests in the [Java testing chapter](#). You can set up other source sets that fulfil different roles in the same way. The question then becomes: when should you define a custom source set?

To answer that question, consider whether the sources:

1. Need to be compiled with a unique classpath
2. Generate classes that are handled differently from the `main` and `test` ones
3. Form a natural part of the project

If your answer to both 3 and either one of the others is yes, then a custom source set is probably the right approach. For example, integration tests are typically part of the project because they test the code in `main`. In addition, they often have either their own dependencies independent of the `test` source set or they need to be run with a custom `Test` task.

Other common scenarios are less clear cut and may have better solutions. For example:

- Separate API and implementation JARs — it may make sense to have these as separate projects, particularly if you already have a multi-project build
- Generated sources — if the resulting sources should be compiled with the production code, add their path(s) to the `main` source set and make sure that the `compileJava` task depends on the task that generates the sources

If you're unsure whether to create a custom source set or not, then go ahead and do so. It should be straightforward and if it's not, then it's probably not the right tool for the job.

Debugging compiling errors

Gradle provides detailed reporting for compilation failures.

If a compilation task fails, the summary of errors is displayed in the following locations:

- The task's output, providing immediate context for the error.
- The "What went wrong" summary at the bottom of the build output, consolidated with all other failures for easy reference.

```
* What went wrong:
Execution failed for task ':project1:compileJava'.
> Compilation failed; see the compiler output below.
Java compilation warning
    sample-project/src/main/java/Problem1.java:6: warning: [cast] redundant cast to
String
        var warning = (String)"warning";
                        ^
Java compilation error
    sample-project/src/main/java/Problem2.java:6: error: incompatible types: int cannot
be converted to String
        String a = 1;
                   ^
```

This reporting feature works with the `-continue` flag.

Managing resources

Many Java projects make use of resources beyond source files, such as images, configuration files and localization data. Sometimes these files simply need to be packaged unchanged and sometimes they need to be processed as template files or in some other way. Either way, the Java Library Plugin adds a specific [Copy](#) task for each source set that handles the processing of its associated resources.

The task's name follows the convention of `processSourceSetResources` — or `processResources` for the `main` source set — and it will automatically copy any files in `src/[sourceSet]/resources` to a directory that will be included in the production JAR. This target directory will also be included in the runtime classpath of the tests.

Since `processResources` is an instance of the `ProcessResources` task, you can perform any of the processing described in the [Working With Files](#) chapter.

Java properties files and reproducible builds

You can easily create Java properties files via the [WriteProperties](#) task, which fixes a well-known problem with `Properties.store()` that can reduce the usefulness of [incremental builds](#).

The standard Java API for writing properties files produces a unique file every time, even when the same properties and values are used, because it includes a timestamp in the comments. Gradle's `WriteProperties` task generates exactly the same output byte-for-byte if none of the properties have changed. This is achieved by a few tweaks to how a properties file is generated:

- no timestamp comment is added to the output
- the line separator is system independent, but can be configured explicitly (it defaults to `'\n'`)
- the properties are sorted alphabetically

Sometimes it can be desirable to recreate archives in a byte for byte way on different machines. You want to be sure that building an artifact from source code produces the same result, byte for byte, no matter when and where it is built. This is necessary for projects like [reproducible-builds.org](#).

These tweaks not only lead to better incremental build integration, but they also help with [reproducible builds](#). In essence, reproducible builds guarantee that you will see the same results from a build execution — including test results and production binaries — no matter when or on what system you run it.

Running tests

Alongside providing automatic compilation of unit tests in `src/test/java`, the Java Library Plugin has native support for running tests that use JUnit 3, 4 & 5 (JUnit 5 support [came in Gradle 4.6](#)) and TestNG. You get:

- An automatic `test` task of type `Test`, using the `test` source set
- An HTML test report that includes the results from *all* `Test` tasks that run
- Easy filtering of which tests to run

- Fine-grained control over how the tests are run
- The opportunity to create your own test execution and test reporting tasks

You do *not* get a **Test** task for every source set you declare, since not every source set represents tests! That's why you typically need to [create your own Test tasks](#) for things like integration and acceptance tests if they can't be included with the **test** source set.

As there is a lot to cover when it comes to testing, the topic has its [own chapter](#) in which we look at:

- How tests are run
- How to run a subset of tests via filtering
- How Gradle discovers tests
- How to configure test reporting and add your own reporting tasks
- How to make use of specific JUnit and TestNG features

You can also learn more about configuring tests in the DSL reference for [Test](#).

Packaging and publishing

How you package and potentially publish your Java project depends on what type of project it is. Libraries, applications, web applications and enterprise applications all have differing requirements. In this section, we will focus on the bare bones provided by the Java Library Plugin.

By default, the Java Library Plugin provides the **jar** task that packages all the compiled production classes and resources into a single JAR. This JAR is also automatically built by the **assemble** task. Furthermore, the plugin can be configured to provide the **javadocJar** and **sourcesJar** tasks to package Javadoc and source code if so desired. If a publishing plugin is used, these tasks will automatically run during publishing or can be called directly.

build.gradle.kts

```
java {  
    withJavadocJar()  
    withSourcesJar()  
}
```

build.gradle

```
java {  
    withJavadocJar()  
    withSourcesJar()  
}
```

If you want to create an 'uber' (AKA 'fat') JAR, then you can use a task definition like this:

build.gradle.kts

```
plugins {
    java
}

version = "1.0.0"

repositories {
    mavenCentral()
}

dependencies {
    implementation("commons-io:commons-io:2.6")
}

tasks.register<Jar>("uberJar") {
    archiveClassifier = "uber"

    from(sourceSets.main.get().output)

    dependsOn(configurations.runtimeClasspath)
    from({
        configurations.runtimeClasspath.get().filter {
            it.name.endsWith("jar")
        }.map { zipTree(it) }
    })
}
```

build.gradle

```
plugins {
    id 'java'
}

version = '1.0.0'

repositories {
    mavenCentral()
}

dependencies {
    implementation 'commons-io:commons-io:2.6'
}

tasks.register('uberJar', Jar) {
```

```

archiveClassifier = 'uber'

from sourceSets.main.output

dependsOn configurations.runtimeClasspath
from {
    configurations.runtimeClasspath.findAll { it.name.endsWith('jar') }
}.collect { zipTree(it) }
}

```

See [Jar](#) for more details on the configuration options available to you. And note that you need to use `archiveClassifier` rather than `archiveAppendix` here for correct publication of the JAR.

You can use one of the publishing plugins to publish the JARs created by a Java project:

- [Maven Publish Plugin](#)
- [Ivy Publish Plugin](#)

Modifying the JAR manifest

Each instance of the `Jar`, `War` and `Ear` tasks has a `manifest` property that allows you to customize the `MANIFEST.MF` file that goes into the corresponding archive. The following example demonstrates how to set attributes in the JAR's manifest:

build.gradle.kts

```

tasks.jar {
    manifest {
        attributes(
            "Implementation-Title" to "Gradle",
            "Implementation-Version" to archiveVersion
        )
    }
}

```

build.gradle

```

jar {
    manifest {
        attributes("Implementation-Title": "Gradle",
            "Implementation-Version": archiveVersion)
    }
}

```

See [Manifest](#) for the configuration options it provides.

You can also create standalone instances of [Manifest](#). One reason for doing so is to share manifest information between JARs. The following example demonstrates how to share common attributes between JARs:

build.gradle.kts

```
val sharedManifest = java.manifest {
    attributes (
        "Implementation-Title" to "Gradle",
        "Implementation-Version" to version
    )
}

tasks.register<Jar>("fooJar") {
    manifest = java.manifest {
        from(sharedManifest)
    }
}
```

build.gradle

```
def sharedManifest = java.manifest {
    attributes("Implementation-Title": "Gradle",
              "Implementation-Version": version)
}

tasks.register('fooJar', Jar) {
    manifest = java.manifest {
        from sharedManifest
    }
}
```

Another option available to you is to merge manifests into a single [Manifest](#) object. Those source manifests can take the form of a text file or another [Manifest](#) object. In the following example, the source manifests are all text files except for `sharedManifest`, which is the [Manifest](#) object from the previous example:

build.gradle.kts

```
tasks.register<Jar>("barJar") {
    manifest {
        attributes("key1" to "value1")
    }
}
```

```

        from(sharedManifest, "src/config/basemanifest.txt")
        from(listOf("src/config/javabasemanifest.txt",
"src/config/libbasemanifest.txt")) {
            eachEntry(Action<ManifestMergeDetails> {
                if (baseValue != mergeValue) {
                    value = baseValue
                }
                if (key == "foo") {
                    exclude()
                }
            })
        }
    }
}

```

build.gradle

```

tasks.register('barJar', Jar) {
    manifest {
        attributes key1: 'value1'
        from sharedManifest, 'src/config/basemanifest.txt'
        from(['src/config/javabasemanifest.txt',
'src/config/libbasemanifest.txt']) {
            eachEntry { details ->
                if (details.baseValue != details.mergeValue) {
                    details.value = baseValue
                }
                if (details.key == 'foo') {
                    details.exclude()
                }
            }
        }
    }
}
}

```

Manifests are merged in the order they are declared in the `from` statement. If the base manifest and the merged manifest both define values for the same key, the merged manifest wins by default. You can fully customize the merge behavior by adding `eachEntry` actions in which you have access to a `ManifestMergeDetails` instance for each entry of the resulting manifest. Note that the merge is done lazily, either when generating the JAR or when `Manifest.writeTo()` or `Manifest.getEffectiveManifest()` are called.

Speaking of `writeTo()`, you can use that to easily write a manifest to disk at any time, like so:

build.gradle.kts

```
tasks.jar { manifest.writeTo(layout.buildDirectory.file("mymanifest.mf")) }
```

build.gradle

```
tasks.named('jar') { manifest.writeTo(layout.buildDirectory.file(
    'mymanifest.mf')) }
```

Generating API documentation

The Java Library Plugin provides a `javadoc` task of type `Javadoc`, that will generate standard Javadocs for all your production code, i.e. whatever source is in the `main` source set. The task supports the core Javadoc and standard doclet options described in the [Javadoc reference documentation](#). See [CoreJavadocOptions](#) and [StandardJavadocDocletOptions](#) for a complete list of those options.

As an example of what you can do, imagine you want to use Asciidoc syntax in your Javadoc comments. To do this, you need to add Asciidoclet to Javadoc's doclet path. Here's an example that does just that:

build.gradle.kts

```
val asciidoclet by configurations.creating

dependencies {
    asciidoclet("org.asciidoctor:asciidoclet:1.+")
}

tasks.register("configureJavadoc") {
    doLast {
        tasks.javadoc {
            options.doclet = "org.asciidoctor.Asciidoclet"
            options.docletpath = asciidoclet.files.toList()
        }
    }
}

tasks.javadoc {
    dependsOn("configureJavadoc")
}
```

build.gradle

```
configurations {
    asciidoclet
}

dependencies {
    asciidoclet 'org.asciidoctor:asciidoclet:1.+'
}

tasks.register('configureJavadoc') {
    doLast {
        javadoc {
            options.doclet = 'org.asciidoctor.Asciidoclet'
            options.docletpath = configurations.asciidoclet.files.toList()
        }
    }
}

javadoc {
    dependsOn configureJavadoc
}
```

You don't have to create a configuration for this, but it's an elegant way to handle dependencies that are required for a unique purpose.

You might also want to create your own Javadoc tasks, for example to generate API docs for the tests:

build.gradle.kts

```
tasks.register<Javadoc>("testJavadoc") {
    source = sourceSets.test.get().allJava
}
```

build.gradle

```
tasks.register('testJavadoc', Javadoc) {
    source = sourceSets.test.allJava
}
```

These are just two non-trivial but common customizations that you might come across.

Cleaning the build

The Java Library Plugin adds a `clean` task to your project by virtue of applying the [Base Plugin](#). This task simply deletes everything in the `layout.buildDirectory` directory, hence why you should always put files generated by the build in there. The task is an instance of `Delete` and you can change what directory it deletes by setting its `dir` property.

Building JVM components

All of the specific JVM plugins are built on top of the [Java Plugin](#). The examples above only illustrated concepts provided by this base plugin and shared with all JVM plugins.

Read on to understand which plugins fits which project type, as it is recommended to pick a specific plugin instead of applying the Java Plugin directly.

Building Java libraries

The unique aspect of library projects is that they are used (or "consumed") by other Java projects. That means the dependency metadata published with the JAR file — usually in the form of a Maven POM — is crucial. In particular, consumers of your library should be able to distinguish between two different types of dependencies: those that are only required to compile your library and those that are also required to compile the consumer.

Gradle manages this distinction via the [Java Library Plugin](#), which introduces an *api* configuration in addition to the *implementation* one covered in this chapter. If the types from a dependency appear in public fields or methods of your library's public classes, then that dependency is exposed via your library's public API and should therefore be added to the *api* configuration. Otherwise, the dependency is an internal implementation detail and should be added to *implementation*.

If you're unsure of the difference between an API and implementation dependency, the [Java Library Plugin chapter](#) has a detailed explanation. In addition, you can explore a basic, practical [sample of building a Java library](#).

Building Java applications

Java applications packaged as a JAR aren't set up for easy launching from the command line or a desktop environment. The [Application Plugin](#) solves the command line aspect by creating a distribution that includes the production JAR, its dependencies and launch scripts Unix-like and Windows systems.

See the plugin's chapter for more details, but here's a quick summary of what you get:

- `assemble` creates ZIP and TAR distributions of the application containing everything needed to run it
- A `run` task that starts the application from the build (for easy testing)
- Shell and Windows Batch scripts to start the application

You can see a basic example of building a Java application in the corresponding [sample](#).

Building Java web applications

Java web applications can be packaged and deployed in a number of ways depending on the technology you use. For example, you might use [Spring Boot](#) with a fat JAR or a [Reactive](#)-based system running on [Netty](#). Whatever technology you use, Gradle and its large community of plugins will satisfy your needs. Core Gradle, though, only directly supports traditional Servlet-based web applications deployed as WAR files.

That support comes via the [War Plugin](#), which automatically applies the Java Plugin and adds an extra packaging step that does the following:

- Copies static resources from *src/main/webapp* into the root of the WAR
- Copies the compiled production classes into a *WEB-INF/classes* subdirectory of the WAR
- Copies the library dependencies into a *WEB-INF/lib* subdirectory of the WAR

This is done by the `war` task, which effectively replaces the `jar` task — although that task remains — and is attached to the `assemble` lifecycle task. See the plugin's chapter for more details and configuration options.

There is no core support for running your web application directly from the build, but we do recommend that you try the [Gretty](#) community plugin, which provides an embedded Servlet container.

Building Java EE applications

Java enterprise systems have changed a lot over the years, but if you're still deploying to JEE application servers, you can make use of the [Ear Plugin](#). This adds conventions and a task for building EAR files. The plugin's chapter has more details.

Building Java Platforms

A Java platform represents a set of dependency declarations and constraints that form a cohesive unit to be applied on consuming projects. The platform has no source and no artifact of its own. It maps in the Maven world to a [BOM](#).

The support comes via the [Java Platform plugin](#), which sets up the different configurations and publication components.

NOTE This plugin is the exception as it does not apply the Java Plugin.

Enabling Java preview features

WARNING

Using a Java preview feature is very likely to make your code incompatible with that compiled without a feature preview. As a consequence, we strongly recommend you not to publish libraries compiled with preview features and restrict the use of feature previews to toy projects.

To enable Java [preview features](#) for compilation, test execution and runtime, you can use the

following DSL snippet:

build.gradle.kts

```
tasks.withType<JavaCompile>().configureEach {
    options.compilerArgs.add("--enable-preview")
}

tasks.withType<Test>().configureEach {
    jvmArgs("--enable-preview")
}

tasks.withType<JavaExec>().configureEach {
    jvmArgs("--enable-preview")
}
```

build.gradle

```
tasks.withType(JavaCompile).configureEach {
    options.compilerArgs += "--enable-preview"
}

tasks.withType(Test).configureEach {
    jvmArgs += "--enable-preview"
}

tasks.withType(JavaExec).configureEach {
    jvmArgs += "--enable-preview"
}
```

Building other JVM language projects

If you want to leverage the multi language aspect of the JVM, most of what was described here will still apply.

Gradle itself provides [Groovy](#) and [Scala](#) plugins. The plugins automatically apply support for compiling Java code and can be further enhanced by combining them with the [java-library](#) plugin.

Compilation dependency between languages

These plugins create a dependency between Groovy/Scala compilation and Java compilation (of source code in the [java](#) folder of a source set). You can change this default behavior by adjusting the classpath of the involved compile tasks as shown in the following example:

build.gradle.kts

```
tasks.named<AbstractCompile>("compileGroovy") {
    // Groovy only needs the declared dependencies
    // (and not longer the output of compileJava)
    classpath = sourceSets.main.get().compileClasspath
}
tasks.named<AbstractCompile>("compileJava") {
    // Java also depends on the result of Groovy compilation
    // (which automatically makes it depend of compileGroovy)
    classpath += files(sourceSets.main.get().groovy.classesDirectory)
}
```

build.gradle

```
tasks.named('compileGroovy') {
    // Groovy only needs the declared dependencies
    // (and not longer the output of compileJava)
    classpath = sourceSets.main.compileClasspath
}
tasks.named('compileJava') {
    // Java also depends on the result of Groovy compilation
    // (which automatically makes it depend of compileGroovy)
    classpath += files(sourceSets.main.groovy.classesDirectory)
}
```

1. By setting the `compileGroovy` classpath to be only `sourceSets.main.compileClasspath`, we effectively remove the previous dependency on `compileJava` that was declared by having the classpath also take into consideration `sourceSets.main.java.classesDirectory`
2. By adding `sourceSets.main.groovy.classesDirectory` to the `compileJava` classpath, we effectively declare a dependency on the `compileGroovy` task

All of this is possible through the use of [directory properties](#).

Extra language support

Beyond core Gradle, there are other [great plugins](#) for more JVM languages!

Testing in Java & JVM projects

Testing on the JVM is a rich subject matter. There are many different testing libraries and frameworks, as well as many different types of test. All need to be part of the build, whether they are executed frequently or infrequently. This chapter is dedicated to explaining how Gradle handles differing requirements between and within builds, with significant coverage of how it

integrates with the two most common testing frameworks: [JUnit](#) and [TestNG](#).

It explains:

- Ways to control how the tests are run ([Test execution](#))
- How to select specific tests to run ([Test filtering](#))
- What test reports are generated and how to influence the process ([Test reporting](#))
- How Gradle finds tests to run ([Test detection](#))
- How to make use of the major frameworks' mechanisms for grouping tests together ([Test grouping](#))

But first, let's look at the basics of JVM testing in Gradle.

NOTE

A new configuration DSL for modeling test execution phases is available via the incubating [JVM Test Suite](#) plugin.

The basics

All JVM testing revolves around a single task type: [Test](#). This runs a collection of test cases using any supported test library — JUnit, JUnit Platform or TestNG — and collates the results. You can then turn those results into a report via an instance of the [TestReport](#) task type.

In order to operate, the [Test](#) task type requires just two pieces of information:

- Where to find the compiled test classes (property: [Test.getTestClassesDirs\(\)](#))
- The execution classpath, which should include the classes under test as well as the test library that you're using (property: [Test.getClasspath\(\)](#))

When you're using a JVM language plugin — such as the [Java Plugin](#) — you will automatically get the following:

- A dedicated [test](#) source set for unit tests
- A [test](#) task of type [Test](#) that runs those unit tests

The JVM language plugins use the source set to configure the task with the appropriate execution classpath and the directory containing the compiled test classes. In addition, they attach the [test](#) task to the [check lifecycle task](#).

It's also worth bearing in mind that the [test](#) source set automatically creates [corresponding dependency configurations](#) — of which the most useful are [testImplementation](#) and [testRuntimeOnly](#) — that the plugins tie into the [test](#) task's classpath.

All you need to do in most cases is configure the appropriate compilation and runtime dependencies and add any necessary configuration to the [test](#) task. The following example shows a simple setup that uses JUnit Platform and changes the maximum heap size for the tests' JVM to 1 gigabyte:

Example 83. *A basic configuration for the 'test' task*

build.gradle.kts

```
dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter:5.7.1")
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")
}

tasks.named<Test>("test") {
    useJUnitPlatform()

    maxHeapSize = "16"

    testLogging {
        events("passed")
    }
}
```

build.gradle

```
dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter:5.7.1'
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'
}

tasks.named('test', Test) {
    useJUnitPlatform()

    maxHeapSize = '16'

    testLogging {
        events "passed"
    }
}
```

The [Test](#) task has many generic configuration options as well as several framework-specific ones that you can find described in [JUnitOptions](#), [JUnitPlatformOptions](#) and [TestNGOptions](#). We cover a significant number of them in the rest of the chapter.

If you want to set up your own [Test](#) task with its own set of test classes, then the easiest approach is to create your own source set and [Test](#) task instance, as shown in [Configuring integration tests](#).

Test execution

Gradle executes tests in a separate ('forked') JVM, isolated from the main build process. This prevents classpath pollution and excessive memory consumption for the build process. It also allows you to run the tests with different JVM arguments than the build is using.

You can control how the test process is launched via several properties on the `Test` task, including the following:

`maxParallelForks` — default: 1

You can run your tests in parallel by setting this property to a value greater than 1. This may make your test suites complete faster, particularly if you run them on a multi-core CPU. When using parallel test execution, make sure your tests are properly isolated from one another. Tests that interact with the filesystem are particularly prone to conflict, causing intermittent test failures.

Your tests can distinguish between parallel test processes by using the value of the `org.gradle.test.worker` property, which is unique for each process. You can use this for anything you want, but it's particularly useful for filenames and other resource identifiers to prevent the kind of conflict we just mentioned.

`forkEvery` — default: 0 (no maximum)

This property specifies the maximum number of test classes that Gradle should run on a test process before its disposed of and a fresh one created. This is mainly used as a way to manage leaky tests or frameworks that have static state that can't be cleared or reset between tests.

Warning: a low value (other than 0) can severely hurt the performance of the tests

`ignoreFailures` — default: false

If this property is `true`, Gradle will continue with the project's build once the tests have completed, even if some of them have failed. Note that, by default, the `Test` task always executes every test that it detects, irrespective of this setting.

`failFast` — (since Gradle 4.6) default: false

Set this to `true` if you want the build to fail and finish as soon as one of your tests fails. This can save a lot of time when you have a long-running test suite and is particularly useful when running the build on continuous integration servers. When a build fails before all tests have run, the test reports only include the results of the tests that have completed, successfully or not.

You can also enable this behavior by using the `--fail-fast` command line option, or disable it respectively with `--no-fail-fast`.

`testLogging` — default: *not set*

This property represents a set of options that control which test events are logged and at what level. You can also configure other logging behavior via this property. See [TestLoggingContainer](#) for more detail.

`dryRun` — default: false

If this property is `true`, Gradle will simulate the execution of the tests without actually running

them. This will still generate reports, allowing for inspection of what tests were selected. This can be used to verify that your test filtering configuration is correct without actually running the tests.

You can also enable this behavior by using the `--test-dry-run` command-line option, or disable it respectively with `--no-test-dry-run`.

See [Test](#) for details on all the available configuration options.

The test process can exit unexpectedly if configured incorrectly. For instance, if the Java executable does not exist or an invalid JVM argument is provided, the test process will fail to start. Similarly, if a test makes programmatic changes to the test process, this can also cause unexpected failures.

For example, issues may occur if a [SecurityManager](#) is modified in a test because Gradle's internal messaging depends on reflection and socket communication, which may be disrupted if the permissions on the security manager change. In this particular case, you should restore the original [SecurityManager](#) after the test so that the gradle test worker process can continue to function.

Test filtering

It's a common requirement to run subsets of a test suite, such as when you're fixing a bug or developing a new test case. Gradle provides two mechanisms to do this:

- Filtering (the preferred option)
- Test inclusion/exclusion

Filtering supersedes the inclusion/exclusion mechanism, but you may still come across the latter in the wild.

With Gradle's test filtering you can select tests to run based on:

- A fully-qualified class name or fully qualified method name, e.g. `org.gradle.SomeTest`, `org.gradle.SomeTest.someMethod`
- A simple class name or method name if the pattern starts with an upper-case letter, e.g. `SomeTest`, `SomeTest.someMethod` (since Gradle 4.7)
- `'*'` wildcard matching

You can enable filtering either in the build script or via the `--tests` command-line option. Here's an example of some filters that are applied every time the build runs:

Example 84. Filtering tests in the build script

build.gradle.kts

```
tasks.test {
    filter {
        //include specific method in any of the tests
        includeTestsMatching("*UiCheck")
    }
}
```

```
//include all tests from package
includeTestsMatching("org.gradle.internal.*")

//include all integration tests
includeTestsMatching("*IntegTest")
}
}
```

build.gradle

```
test {
    filter {
        //include specific method in any of the tests
        includeTestsMatching "*UiCheck"

        //include all tests from package
        includeTestsMatching "org.gradle.internal.*"

        //include all integration tests
        includeTestsMatching "*IntegTest"
    }
}
```

For more details and examples of declaring filters in the build script, please see the [TestFilter](#) reference.

The command-line option is especially useful to execute a single test method. When you use `--tests`, be aware that the inclusions declared in the build script are still honored. It is also possible to supply multiple `--tests` options, all of whose patterns will take effect. The following sections have several examples of using the command-line option.

NOTE

Not all test frameworks play well with filtering. Some advanced, synthetic tests may not be fully compatible. However, the vast majority of tests and use cases work perfectly well with Gradle's filtering mechanism.

The following two sections look at the specific cases of simple class/method names and fully-qualified names.

Simple name pattern

Since 4.7, Gradle has treated a pattern starting with an uppercase letter as a simple class name, or a class name + method name. For example, the following command lines run either all or exactly one of the tests in the `SomeTestClass` test case, regardless of what package it's in:

```
# Executes all tests in SomeTestClass
```

```
gradle test --tests SomeTestClass

# Executes a single specified test in SomeTestClass
gradle test --tests SomeTestClass.someSpecificMethod

gradle test --tests SomeTestClass.*someMethod*
```

Fully-qualified name pattern

Prior to 4.7 or if the pattern doesn't start with an uppercase letter, Gradle treats the pattern as fully-qualified. So if you want to use the test class name irrespective of its package, you would use `--tests *.SomeTestClass`. Here are some more examples:

```
# specific class
gradle test --tests org.gradle.SomeTestClass

# specific class and method
gradle test --tests org.gradle.SomeTestClass.someSpecificMethod

# method name containing spaces
gradle test --tests "org.gradle.SomeTestClass.some method containing spaces"

# all classes at specific package (recursively)
gradle test --tests 'all.in.specific.package*'

# specific method at specific package (recursively)
gradle test --tests 'all.in.specific.package*.someSpecificMethod'

gradle test --tests '*IntegTest'

gradle test --tests '*IntegTest*ui*'

gradle test --tests '*ParameterizedTest.foo*'

# the second iteration of a parameterized test
gradle test --tests '*ParameterizedTest.*[2]'
```

Note that the wildcard '*' has no special understanding of the '.' package separator. It's purely text based. So `--tests *.SomeTestClass` will match any package, regardless of its 'depth'.

You can also combine filters defined at the command line with [continuous build](#) to re-execute a subset of tests immediately after every change to a production or test source file. The following executes all tests in the 'com.mypackage.foo' package or subpackages whenever a change triggers the tests to run:

```
gradle test --continuous --tests "com.mypackage.foo.*"
```

Test reporting

The **Test** task generates the following results by default:

- An HTML test report
- XML test results in a format compatible with the Ant JUnit report task — one that is supported by many other tools, such as CI servers
- An efficient binary format of the results used by the **Test** task to generate the other formats

In most cases, you'll work with the standard HTML report, which automatically includes the results from *all* your **Test** tasks, even the ones you explicitly add to the build yourself. For example, if you add a **Test** task for integration tests, the report will include the results of both the unit tests and the integration tests if both tasks are run.

NOTE

To aggregate test results across multiple subprojects, see the [Test Report Aggregation Plugin](#).

Unlike with many of the testing configuration options, there are several project-level [convention properties that affect the test reports](#). For example, you can change the destination of the test results and reports like so:

Example 85. [Changing the default test report and results directories](#)

build.gradle.kts

```
reporting.baseDirectory = file("my-reports")
java.testResultsDir = layout.buildDirectory.dir("my-test-results")

tasks.register("showDirs") {
    val settingsDir = project.layout.settingsDirectory.asFile
    val reportsDir = project.reporting.baseDirectory
    val testResultsDir = project.java.testResultsDir

    doLast {

        logger.quiet(settingsDir.toPath().relativize(reportsDir.get().asFile.toPath())
            .toString())

        logger.quiet(settingsDir.toPath().relativize(testResultsDir.get().asFile.toPath())
            .toString())
    }
}
```

build.gradle

```
reporting.baseDirectory = file("my-reports")
```

```

java.testResultsDir = layout.buildDirectory.dir("my-test-results")

tasks.register('showDirs') {
    def settingsDir = project.layout.settingsDirectory.asFile
    def reportsDir = project.reporting.baseDirectory
    def testResultsDir = project.java.testResultsDir

    doLast {
        logger.quiet(settingsDir.toPath().relativize(reportsDir.get().asFile
.toPath()).toString())
        logger.quiet(settingsDir.toPath().relativize(testResultsDir.get()
.asFile.toPath()).toString())
    }
}

```

Output of `gradle -q showDirs`

```

> gradle -q showDirs
my-reports
build/my-test-results

```

Follow the link to the convention properties for more details.

There is also a standalone [TestReport](#) task type that you can use to generate a custom HTML test report. All it requires are a value for `destinationDir` and the test results you want included in the report. Here is a sample which generates a combined report for the unit tests from all subprojects:

Example 86. [Creating a unit test report for subprojects](#)

buildSrc/src/main/kotlin/myproject.java-conventions.gradle.kts

```

plugins {
    id("java")
}

// Disable the test report for the individual test task
tasks.named<Test>("test") {
    reports.html.required = false
}

// Share the test report data to be aggregated for the whole project
configurations.create("binaryTestResultsElements") {
    isCanBeResolved = false
    isCanBeConsumed = true
    attributes {
        attribute(Category.CATEGORY_ATTRIBUTE,
objects.named(Category.DOCUMENTATION))
    }
}

```

```

        attribute(DocsType.DOCS_TYPE_ATTRIBUTE, objects.named("test-report-
data"))
    }
    outgoing.artifact(tasks.test.map { task ->
task.getBinaryResultsDirectory().get() })
}

```

build.gradle.kts

```

val testReportData by configurations.creating {
    isCanBeConsumed = false
    attributes {
        attribute(Category.CATEGORY_ATTRIBUTE,
objects.named(Category.DOCUMENTATION))
        attribute(DocsType.DOCS_TYPE_ATTRIBUTE, objects.named("test-report-
data"))
    }
}

dependencies {
    testReportData(project(":core"))
    testReportData(project(":util"))
}

tasks.register<TestReport>("testReport") {
    destinationDirectory = reporting.baseDirectory.dir("allTests")
    // Use test results from testReportData configuration
    testResults.from(testReportData)
}

```

buildSrc/src/main/groovy/myproject.java-conventions.gradle

```

plugins {
    id 'java'
}

// Disable the test report for the individual test task
test {
    reports.html.required = false
}

// Share the test report data to be aggregated for the whole project
configurations {
    binaryTestResultsElements {
        canBeResolved = false
        canBeConsumed = true
        attributes {
            attribute(Category.CATEGORY_ATTRIBUTE, objects.named(Category,

```

```

Category.DOCUMENTATION))
        attribute(DocsType.DOCS_TYPE_ATTRIBUTE, objects.named(DocsType,
'test-report-data'))
    }
    outgoing.artifact(test.binaryResultsDirectory)
}
}

```

build.gradle

```

// A resolvable configuration to collect test reports data
configurations {
    testReportData {
        canBeConsumed = false
        attributes {
            attribute(Category.CATEGORY_ATTRIBUTE, objects.named(Category,
Category.DOCUMENTATION))
            attribute(DocsType.DOCS_TYPE_ATTRIBUTE, objects.named(DocsType,
'test-report-data'))
        }
    }
}

dependencies {
    testReportData project(':core')
    testReportData project(':util')
}

tasks.register('testReport', TestReport) {
    destinationDirectory = reporting.baseDirectory.dir('allTests')
    // Use test results from testReportData configuration
    testResults.from(configurations.testReportData)
}

```

In this example, we use a convention plugin `myproject.java-conventions` to expose the test results from a project to Gradle's [variant aware dependency management engine](#).

The plugin declares a consumable `binaryTestResultsElements` configuration that represents the binary test results of the `test` task. In the aggregation project's build file, we declare the `testReportData` configuration and depend on all of the projects that we want to aggregate the results from. Gradle will automatically select the binary test result variant from each of the subprojects instead of the project's jar file. Lastly, we add a `testReport` task that aggregates the test results from the `testResultsDirs` property, which contains all of the binary test results resolved from the `testReportData` configuration.

You should note that the `TestReport` type combines the results from multiple test tasks and needs to aggregate the results of individual test classes. This means that if a given test class is executed by multiple test tasks, then the test report will include executions of that class, but it can be hard to

distinguish individual executions of that class and their output.

Communicating test results to CI servers and other tools via XML files

The Test tasks creates XML files describing the test results, in the "JUnit XML" pseudo standard. This standard is used by the JUnit 4, JUnit Jupiter, and TestNG test frameworks, and is configured using the same DSL block for each of these. It is common for CI servers and other tooling to observe test results via these XML files.

By default, the files are written to `layout.buildDirectory.dir("test-results/$testTaskName")` with a file per test class. The location can be changed for all test tasks of a project, or individually per test task.

Example 87. Changing JUnit XML results location for all test tasks

build.gradle.kts

```
java.testResultsDir = layout.buildDirectory.dir("junit-xml")
```

build.gradle

```
java.testResultsDir = layout.buildDirectory.dir("junit-xml")
```

With the above configuration, the XML files will be written to `layout.buildDirectory.dir("junit-xml/$testTaskName")`.

Example 88. Changing JUnit XML results location for a particular test task

build.gradle.kts

```
tasks.test {
    reports {
        junitXml.outputLocation = layout.buildDirectory.dir("test-junit-xml")
    }
}
```

build.gradle

```
test {
    reports {
        junitXml.outputLocation = layout.buildDirectory.dir("test-junit-xml")
    }
}
```

```
}
```

With the above configuration, the XML files for the `test` task will be written to `layout.buildDirectory.dir("test-results/test-junit-xml")`. The location of the XML files for other test tasks will be unchanged.

Configuration options

The content of the XML files can also be configured to convey the results differently, by configuring the `JUnitXmlReport` options.

Example 89. Configuring how the results are conveyed

build.gradle.kts

```
tasks.test {
    reports {
        junitXml.apply {
            includeSystemOutLog = false // defaults to true
            includeSystemErrLog = false // defaults to true
            isOutputPerTestCase = true // defaults to false
            mergeReruns = true // defaults to false
        }
    }
}
```

build.gradle

```
test {
    reports {
        junitXml {
            includeSystemOutLog = false // defaults to true
            includeSystemErrLog = false // defaults to true
            outputPerTestCase = true // defaults to false
            mergeReruns = true // defaults to false
        }
    }
}
```

`includeSystemOutLog` & `includeSystemErrLog`

The `includeSystemOutLog` option allows configuring whether or not test output written to standard out is exported to the XML report file. The `includeSystemErrLog` option allows configuring whether

or not test error output written to standard error is exported to the XML report file.

These options affect both test-suite level output (such as `@BeforeClass/@BeforeAll` output) and test class and method-specific output (`@Before/@BeforeEach` and `@Test`). If either option is disabled, the element that normally contains that content will be excluded from the XML report file.

The default for each option is `true`.

outputPerTestCase

The `outputPerTestCase` option, when enabled, associates any output logging generated during a test case to that test case in the results. When disabled (the default) output is associated with the test class as whole and not the individual test cases (e.g. test methods) that produced the logging output. Most modern tools that observe JUnit XML files support the "output per test case" format.

If you are using the XML files to communicate test results, it is recommended to enable this option as it provides more useful reporting.

mergeReruns

When `mergeReruns` is enabled, if a test fails but is then retried and succeeds, its failures will be recorded as `<flakyFailure>` instead of `<failure>`, within one `<testcase>`. This is effectively the reporting produced by the [surefire plugin of Apache Maven™](#) when enabling reruns. If your CI server understands this format, it will indicate that the test was flaky. If it does not, it will indicate that the test succeeded as it will ignore the `<flakyFailure>` information. If the test does not succeed (i.e. it fails for every retry), it will be indicated as having failed whether your tool understands this format or not.

When `mergeReruns` is disabled (the default), each execution of a test will be listed as a separate test case.

If you are using [build scans](#) or [Develocity](#), flaky tests will be detected regardless of this setting.

Enabling this option is especially useful when using a CI tool that uses the XML test results to determine build failure instead of relying on Gradle's determination of whether the build failed or not, and you wish to not consider the build failed if all failed tests passed when retried. This is the case for the Jenkins CI server and its [JUnit plugin](#). With `mergeReruns` enabled, tests that pass-on-retry will no longer cause this Jenkins plugin to consider the build to have failed. However, failed test executions will be omitted from the Jenkins test result visualizations as it does not consider `<flakyFailure>` information. The separate [Flaky Test Handler Jenkins plugin](#) can be used in addition to the JUnit Jenkins plugin to have such "flaky failures" also be visualized.

Tests are grouped and merged based on their reported name. When using any kind of test parameterization that affects the reported test name, or any other kind of mechanism that produces a potentially dynamic test name, care should be taken to ensure that the test name is stable and does not unnecessarily change.

Enabling the `mergeReruns` option does not add any retry/rerun functionality to test execution. Rerunning can be enabled by the test execution framework (e.g. JUnit's `@RepeatedTest`), or via the separate [Test Retry Gradle plugin](#).

Test detection

By default, Gradle will run all tests that it detects, which it does by inspecting the compiled test classes. This detection uses different criteria depending on the test framework used.

For *JUnit*, Gradle scans for both JUnit 3 and 4 test classes. A class is considered to be a JUnit test if it:

- Ultimately inherits from `TestCase` or `GroovyTestCase`
- Is annotated with `@RunWith`
- Contains a method annotated with `@Test` or a super class does

For *TestNG*, Gradle scans for methods annotated with `@Test`.

Note that abstract classes are not executed. In addition, be aware that Gradle scans up the inheritance tree into jar files on the test classpath. So if those JARs contain test classes, they will also be run.

If you don't want to use test class detection, you can disable it by setting the `scanForTestClasses` property on `Test` to `false`. When you do that, the test task uses only the `includes` and `excludes` properties to find test classes.

If `scanForTestClasses` is false and no include or exclude patterns are specified, Gradle defaults to running any class that matches the patterns `**/*Tests.class` and `**/*Test.class`, excluding those that match `**/Abstract*.class`.

NOTE

With `JUnit Platform`, only `includes` and `excludes` are used to filter test classes — `scanForTestClasses` has no effect.

Test logging

Gradle allows fine-tuned control over events that are logged to the console. Logging is configurable on a per-log-level basis and by default, the following events are logged:

When the log level is	Events that are logged	Additional configuration
<code>ERROR</code> , <code>QUIET</code> or <code>WARNING</code>	None	None
<code>LIFECYCLE</code>	Test failures	Exception format is <code>SHORT</code>
<code>INFO</code>	Test failures, skipped tests, test standard output and test standard error	Stacktraces are truncated.
<code>DEBUG</code>	All events	Full stacktraces are logged.

Test logging can be modified on a per-log-level basis by adjusting the appropriate `TestLogging` instances in the `testLogging` property of the test task. For example, to adjust the `INFO` level test logging configuration, modify the `TestLoggingContainer.getInfo()` property.

Test grouping

JUnit, JUnit Platform and TestNG allow sophisticated groupings of test methods.

NOTE

This section applies to grouping individual test classes or methods within a collection of tests that serve the same testing purpose (unit tests, integration tests, acceptance tests, etc.). For dividing test classes based upon their purpose, see the incubating [JVM Test Suite](#) plugin.

JUnit 4.8 introduced the concept of categories for grouping JUnit 4 tests classes and methods.^[1] [Test.useJUnit\(org.gradle.api.Action\)](#) allows you to specify the JUnit categories you want to include and exclude. For example, the following configuration includes tests in `CategoryA` and excludes those in `CategoryB` for the `test` task:

Example 90. *JUnit Categories*

build.gradle.kts

```
tasks.test {
    useJUnit {
        includeCategories("org.gradle.junit.CategoryA")
        excludeCategories("org.gradle.junit.CategoryB")
    }
}
```

build.gradle

```
test {
    useJUnit {
        includeCategories 'org.gradle.junit.CategoryA'
        excludeCategories 'org.gradle.junit.CategoryB'
    }
}
```

[JUnit Platform](#) introduced [tagging](#) to replace categories. You can specify the included/excluded tags via [Test.useJUnitPlatform\(org.gradle.api.Action\)](#), as follows:

Example 91. *JUnit Platform Tags*

build.gradle.kts

```
tasks.withType<Test>().configureEach {
    useJUnitPlatform {
        includeTags("fast")
    }
}
```

```

        excludeTags("slow")
    }
}

```

build.gradle

```

tasks.withType(Test).configureEach {
    useJUnitPlatform {
        includeTags 'fast'
        excludeTags 'slow'
    }
}

```

The TestNG framework uses the concept of test groups for a similar effect.^[2] You can configure which test groups to include or exclude during the test execution via the [Test.useTestNG\(org.gradle.api.Action\)](#) setting, as seen here:

Example 92. Grouping TestNG tests

build.gradle.kts

```

tasks.named<Test>("test") {
    useTestNG {
        val options = this as TestNGOptions
        options.excludeGroups("integrationTests")
        options.includeGroups("unitTests")
    }
}

```

build.gradle

```

test {
    useTestNG {
        excludeGroups 'integrationTests'
        includeGroups 'unitTests'
    }
}

```

Using JUnit 5

[JUnit 5](#) is the latest version of the well-known JUnit test framework. Unlike its predecessor, JUnit 5 is

modularized and composed of several modules:

```
JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage
```

The JUnit Platform serves as a foundation for launching testing frameworks on the JVM. JUnit Jupiter is the combination of the new [programming model](#) and [extension model](#) for writing tests and extensions in JUnit 5. JUnit Vintage provides a [TestEngine](#) for running JUnit 3 and JUnit 4 based tests on the platform.

The following code enables JUnit Platform support in [build.gradle](#):

Example 93. [Enabling JUnit Platform to run your tests](#)

build.gradle.kts

```
tasks.named<Test>("test") {  
    useJUnitPlatform()  
}
```

build.gradle

```
tasks.named('test', Test) {  
    useJUnitPlatform()  
}
```

See [Test.useJUnitPlatform\(\)](#) for more details.

Compiling and executing JUnit Jupiter tests

To enable JUnit Jupiter support in Gradle, all you need to do is add the following dependency:

Example 94. [JUnit Jupiter dependencies](#)

build.gradle.kts

```
dependencies {  
    testImplementation("org.junit.jupiter:junit-jupiter:5.7.1")  
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")  
}
```

build.gradle

```
dependencies {  
    testImplementation 'org.junit.jupiter:junit-jupiter:5.7.1'  
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'  
}
```

You can then put your test cases into *src/test/java* as normal and execute them with **gradle test**.

Executing legacy tests with JUnit Vintage

If you want to run JUnit 3/4 tests on JUnit Platform, or even mix them with Jupiter tests, you should add extra JUnit Vintage Engine dependencies:

Example 95. JUnit Vintage dependencies

build.gradle.kts

```
dependencies {  
    testImplementation("org.junit.jupiter:junit-jupiter:5.7.1")  
    testCompileOnly("junit:junit:4.13")  
    testRuntimeOnly("org.junit.vintage:junit-vintage-engine")  
    testRuntimeOnly("org.junit.platform:junit-platform-launcher")  
}
```

build.gradle

```
dependencies {  
    testImplementation 'org.junit.jupiter:junit-jupiter:5.7.1'  
    testCompileOnly 'junit:junit:4.13'  
    testRuntimeOnly 'org.junit.vintage:junit-vintage-engine'  
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'  
}
```

In this way, you can use **gradle test** to test JUnit 3/4 tests on JUnit Platform, without the need to rewrite them.

Filtering test engine

JUnit Platform allows you to use different test engines. JUnit currently provides two **TestEngine** implementations out of the box: [junit-jupiter-engine](#) and [junit-vintage-engine](#). You can also write and plug in your own **TestEngine** implementation as documented [here](#).

By default, all test engines on the test runtime classpath will be used. To control specific test engine implementations explicitly, you can add the following setting to your build script:

Example 96. Filter specific engines

build.gradle.kts

```
tasks.withType<Test>().configureEach {
    useJUnitPlatform {
        includeEngines("junit-vintage")
        // excludeEngines("junit-jupiter")
    }
}
```

build.gradle

```
tasks.withType(Test).configureEach {
    useJUnitPlatform {
        includeEngines 'junit-vintage'
        // excludeEngines 'junit-jupiter'
    }
}
```

Test execution order in TestNG

TestNG allows explicit control of the execution order of tests when you use a *testng.xml* file. Without such a file — or an equivalent one configured by [TestNGOptions.getSuiteXmlBuilder\(\)](#) — you can't specify the test execution order. However, what you *can* do is control whether all aspects of a test — including its associated [@BeforeXXX](#) and [@AfterXXX](#) methods, such as those annotated with [@Before/AfterClass](#) and [@Before/AfterMethod](#) — are executed before the next test starts. You do this by setting the [TestNGOptions.getPreserveOrder\(\)](#) property to [true](#). If you set it to [false](#), you may encounter scenarios in which the execution order is something like: [TestA.doBeforeClass\(\)](#) → [TestB.doBeforeClass\(\)](#) → [TestA](#) tests.

While preserving the order of tests is the default behavior when directly working with *testng.xml* files, the [TestNG API](#) that is used by Gradle's TestNG integration executes tests in unpredictable order by default.^[3] The ability to preserve test execution order was introduced with TestNG version 5.14.5. Setting the [preserveOrder](#) property to [true](#) for an older TestNG version will cause the build to fail.

Example 97. *Preserving order of TestNG tests*

build.gradle.kts

```
tasks.test {
    useTestNG {
        preserveOrder = true
    }
}
```

build.gradle

```
test {
    useTestNG {
        preserveOrder = true
    }
}
```

The `groupByInstance` property controls whether tests should be grouped by instance rather than by class. The [TestNG documentation](#) explains the difference in more detail, but essentially, if you have a test method `A()` that depends on `B()`, grouping by instance ensures that each A-B pairing, e.g. `B(1)-A(1)`, is executed before the next pairing. With group by class, all `B()` methods are run and then all `A()` ones.

Note that you typically only have more than one instance of a test if you're using a data provider to parameterize it. Also, grouping tests by instances was introduced with TestNG version 6.1. Setting the `groupByInstances` property to `true` for an older TestNG version will cause the build to fail.

Example 98. *Grouping TestNG tests by instances*

build.gradle.kts

```
tasks.test {
    useTestNG {
        groupByInstances = true
    }
}
```

build.gradle

```
test {
    useTestNG {
```

```
        groupByInstances = true
    }
}
```

TestNG parameterized methods and reporting

TestNG supports [parameterizing test methods](#), allowing a particular test method to be executed multiple times with different inputs. Gradle includes the parameter values in its reporting of the test method execution.

Given a parameterized test method named `aTestMethod` that takes two parameters, it will be reported with the name `aTestMethod(toStringValueOfParam1, toStringValueOfParam2)`. This makes it easy to identify the parameter values for a particular iteration.

Configuring integration tests

A common requirement for projects is to incorporate integration tests in one form or another. Their aim is to verify that the various parts of the project are working together properly. This often means that they require special execution setup and dependencies compared to unit tests.

The simplest way to add integration tests to your build is by leveraging the incubating [JVM Test Suite](#) plugin. If an incubating solution is not something for you, here are the steps you need to take in your build:

1. Create a new [source set](#) for them
2. Add the dependencies you need to the appropriate configurations for that source set
3. Configure the compilation and runtime classpaths for that source set
4. Create a task to run the integration tests

You may also need to perform some additional configuration depending on what form the integration tests take. We will discuss those as we go.

Let's start with a practical example that implements the first three steps in a build script, centered around a new source set `intTest`:

Example 99. [Setting up working integration tests](#)

build.gradle.kts

```
sourceSets {
    create("intTest") {
        compileClasspath += sourceSets.main.get().output
        runtimeClasspath += sourceSets.main.get().output
    }
}
```

```

val intTestImplementation by configurations.getting {
    extendsFrom(configurations.implementation.get())
}
val intTestRuntimeOnly by configurations.getting

configurations["intTestRuntimeOnly"].extendsFrom(configurations.runtimeOnly.get())

dependencies {
    intTestImplementation("org.junit.jupiter:junit-jupiter:5.7.1")
    intTestRuntimeOnly("org.junit.platform:junit-platform-launcher")
}

```

build.gradle

```

sourceSets {
    intTest {
        compileClasspath += sourceSets.main.output
        runtimeClasspath += sourceSets.main.output
    }
}

configurations {
    intTestImplementation.extendsFrom implementation
    intTestRuntimeOnly.extendsFrom runtimeOnly
}

dependencies {
    intTestImplementation 'org.junit.jupiter:junit-jupiter:5.7.1'
    intTestRuntimeOnly 'org.junit.platform:junit-platform-launcher'
}

```

This will set up a new source set called `intTest` that automatically creates:

- `intTestImplementation`, `intTestCompileOnly`, `intTestRuntimeOnly` configurations (and [a few others](#) that are less commonly needed)
- A `compileIntTestJava` task that will compile all the source files under `src/intTest/java`

NOTE

If you are working with the IntelliJ IDE, you may wish to flag the directories in these additional source sets as containing test source rather than production source as explained in the [Idea Plugin](#) documentation.

The example also does the following, not all of which you may need for your specific integration tests:

- Adds the production classes from the `main` source set to the compilation and runtime classpaths

of the integration tests — `sourceSets.main.output` is a [file collection](#) of all the directories containing compiled production classes and resources

- Makes the `intTestImplementation` configuration extend from `implementation`, which means that all the declared dependencies of the production code also become dependencies of the integration tests
- Does the same for the `intTestRuntimeOnly` configuration

In most cases, you want your integration tests to have access to the classes under test, which is why we ensure that those are included on the compilation and runtime classpaths in this example. But some types of test interact with the production code in a different way. For example, you may have tests that run your application as an executable and verify the output. In the case of web applications, the tests may interact with your application via HTTP. Since the tests don't need direct access to the classes under test in such cases, you don't need to add the production classes to the test classpath.

Another common step is to attach all the unit test dependencies to the integration tests as well — via `intTestImplementation.extendsFrom testImplementation` — but that only makes sense if the integration tests require *all* or nearly all the same dependencies that the unit tests have.

There are a couple of other facets of the example you should take note of:

- `+=` allows you to append paths and collections of paths to `compileClasspath` and `runtimeClasspath` instead of overwriting them
- If you want to use the convention-based configurations, such as `intTestImplementation`, you *must* declare the dependencies *after* the new source set

Creating and configuring a source set automatically sets up the compilation stage, but it does nothing with respect to running the integration tests. So the last piece of the puzzle is a custom test task that uses the information from the new source set to configure its runtime classpath and the test classes:

Example 100. [Defining a working integration test task](#)

build.gradle.kts

```
val integrationTest = task<Test>("integrationTest") {
    description = "Runs integration tests."
    group = "verification"

    testClassesDirs = sourceSets["intTest"].output.classesDirs
    classpath = sourceSets["intTest"].runtimeClasspath
    shouldRunAfter("test")

    useJUnitPlatform()

    testLogging {
        events("passed")
    }
}
```

```
}

tasks.check { dependsOn(integrationTest) }
```

build.gradle

```
tasks.register('integrationTest', Test) {
    description = 'Runs integration tests.'
    group = 'verification'

    testClassesDirs = sourceSets.intTest.output.classesDirs
    classpath = sourceSets.intTest.runtimeClasspath
    shouldRunAfter test

    useJUnitPlatform()

    testLogging {
        events "passed"
    }
}

check.dependsOn integrationTest
```

Again, we're accessing a source set to get the relevant information, i.e. where the compiled test classes are — the `testClassesDirs` property — and what needs to be on the classpath when running them — `classpath`.

Users commonly want to run integration tests after the unit tests, because they are often slower to run and you want the build to fail early on the unit tests rather than later on the integration tests. That's why the above example adds a `shouldRunAfter()` declaration. This is preferred over `mustRunAfter()` so that Gradle has more flexibility in executing the build in parallel.

For information on how to determine code coverage for tests in additional source sets, see the [JaCoCo Plugin](#) and the [JaCoCo Report Aggregation Plugin](#) chapters.

Testing Java Modules

If you are [developing Java Modules](#), everything described in this chapter still applies and any of the supported test frameworks can be used. However, there are some things to consider depending on whether you need module information to be available, and module boundaries to be enforced, during test execution. In this context, the terms *whitebox testing* (module boundaries are deactivated or relaxed) and *blackbox testing* (module boundaries are in place) are often used. Whitebox testing is used/needed for unit testing and blackbox testing fits functional or integration test requirements.

Sample: [Java Modules multi-project with integration tests](#)

Whitebox unit test execution on the classpath

The simplest setup to write unit tests for functions or classes in modules is to *not* use module specifics during test execution. For this, you just need to write tests the same way you would write them for normal libraries. If you don't have a `module-info.java` file in your test source set (`src/test/java`) this source set will be considered as traditional Java library during compilation and test runtime. This means, all dependencies, including Jars with module information, are put on the classpath. The advantage is that all internal classes of your (or other) modules are then accessible directly in tests. This may be a totally valid setup for unit testing, where we do not care about the larger module structure, but only about testing single functions.

NOTE

If you are using Eclipse: By default, Eclipse also runs unit tests as modules using module patching (see [below](#)). In an imported Gradle project, unit testing a module with the Eclipse test runner might fail. You then need to manually adjust the classpath/module path in the test run configuration or delegate test execution to Gradle.

This only concerns the test execution. Unit test compilation and development works fine in Eclipse.

Blackbox integration testing

For integration tests, you have the option to define the test set itself as additional module. You do this similar to how you turn your main sources into a module: by adding a `module-info.java` file to the corresponding source set (e.g. `integrationTests/java/module-info.java`).

You can find a full example that includes blackbox integration tests [here](#).

NOTE

In Eclipse, compiling multiple modules in one project is [currently not support](#). Therefore the integration test (blackbox) setup described here only works in Eclipse if the tests are moved to a separate subproject.

Whitebox test execution with module patching

Another approach for whitebox testing is to stay in the module world by *patching* the tests into the module under test. This way, module boundaries stay in place, but the tests themselves become part of the module under test and can then access the module's internals.

For which uses cases this is relevant and how this is best done is a topic of discussion. There is no general best approach at the moment. Thus, there is no special support for this in Gradle right now.

You can however, setup module patching for tests like this:

- Add a `module-info.java` to your test source set that is a copy of the main `module-info.java` with additional dependencies needed for testing (e.g. `requires org.junit.jupiter.api`).
- Configure both the `testCompileJava` and `test` tasks with arguments to patch the main classes with the test classes as shown below.

build.gradle.kts

```
val moduleName = "org.gradle.sample"
val patchArgs = listOf("--patch-module",
"$moduleName=${tasks.compileJava.get().destinationDirectory.asFile.get().path}
}")
tasks.compileTestJava {
    options.compilerArgs.addAll(patchArgs)
}
tasks.test {
    jvmArgs(patchArgs)
}
```

build.gradle

```
def moduleName = "org.gradle.sample"
def patchArgs = ["--patch-module", "$moduleName=${tasks.compileJava
.destinationDirectory.asFile.get().path}"]
tasks.named('compileTestJava') {
    options.compilerArgs += patchArgs
}
tasks.named('test') {
    jvmArgs += patchArgs
}
```

NOTE

If custom arguments are used for patching, these are not picked up by Eclipse and IDEA. You will most likely see invalid compilation errors in the IDE.

Skipping the tests

If you want to skip the tests when running a build, you have a few options. You can either do it via [command line arguments](#) or [in the build script](#). To do it on the command line, you can use the `-x` or `--exclude-task` option like so:

```
gradle build -x test
```

This excludes the `test` task and any other task that it *exclusively* depends on, i.e. no other task depends on the same task. Those tasks will not be marked "SKIPPED" by Gradle, but will simply not appear in the list of tasks executed.

Skipping a test via the build script can be done a few ways. One common approach is to make test

execution conditional via the `Task.onlyIf(String, org.gradle.api.specs.Spec)` method. The following sample skips the `test` task if the project has a property called `mySkipTests`:

Example 102. Skipping the unit tests based on a project property

build.gradle.kts

```
tasks.test {
    val skipTestsProvider = providers.gradleProperty("mySkipTests")
    onlyIf("mySkipTests property is not set") {
        !skipTestsProvider.isPresent()
    }
}
```

build.gradle

```
def skipTestsProvider = providers.gradleProperty('mySkipTests')
test.onlyIf("mySkipTests property is not set") {
    !skipTestsProvider.present
}
```

In this case, Gradle will mark the skipped tests as "SKIPPED" rather than exclude them from the build.

Forcing tests to run

In well-defined builds, you can rely on Gradle to only run tests if the tests themselves or the production code change. However, you may encounter situations where the tests rely on a third-party service or something else that might change but can't be modeled in the build.

You can always use the `--rerun` built-in task option to force a task to rerun.

```
gradle test --rerun
```

Alternatively, if `build caching` is not enabled, you can also force tests to run by cleaning the output of the relevant `Test` task — say `test` — and running the tests again, like so:

```
gradle cleanTest test
```

`cleanTest` is based on a task rule provided by the `Base Plugin`. You can use it for *any* task.

Debugging when running tests

On the few occasions that you want to debug your code while the tests are running, it can be helpful if you can attach a debugger at that point. You can either set the `Test.getDebug()` property to `true` or use the `--debug-jvm` command line option, or use `--no-debug-jvm` to set it to false.

When debugging for tests is enabled, Gradle will start the test process suspended and listening on port 5005.

You can also enable debugging in the DSL, where you can also configure other properties:

```
test {
    debugOptions {
        enabled = true
        host = 'localhost'
        port = 4455
        server = true
        suspend = true
    }
}
```

With this configuration the test JVM will behave just like when passing the `--debug-jvm` argument but it will listen on port 4455.

To debug the test process remotely via network, the `host` needs to be set to the machine's IP address or `"*"` (listen on all interfaces).

Using test fixtures

Producing and using test fixtures within a single project

Test fixtures are commonly used to setup the code under test, or provide utilities aimed at facilitating the tests of a component. Java projects can enable test fixtures support by applying the `java-test-fixtures` plugin, in addition to the `java` or `java-library` plugins:

Example 103. Applying the Java test fixtures plugin

lib/build.gradle.kts

```
plugins {
    // A Java Library
    `java-library`
    // which produces test fixtures
    `java-test-fixtures`
    // and is published
    `maven-publish`
}
```

lib/build.gradle

```
plugins {  
    // A Java Library  
    id 'java-library'  
    // which produces test fixtures  
    id 'java-test-fixtures'  
    // and is published  
    id 'maven-publish'  
}
```

This will automatically create a `testFixtures` source set, in which you can write your test fixtures. Test fixtures are configured so that:

- they can see the *main* source set classes
- *test* sources can see the *test fixtures* classes

For example for this main class:

src/main/java/com/acme/Person.java

```
public class Person {  
    private final String firstName;  
    private final String lastName;  
  
    public Person(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    public String getFirstName() {  
        return firstName;  
    }  
  
    public String getLastName() {  
        return lastName;  
    }  
  
    // ...  
}
```

A test fixture can be written in `src/testFixtures/java`:

src/testFixtures/java/com/acme/Simpsons.java

```
public class Simpsons {  
    private static final Person HOMER = new Person("Homer", "Simpson");  
    private static final Person MARGE = new Person("Marjorie", "Simpson");  
}
```

```

private static final Person BART = new Person("Bartholomew", "Simpson");
private static final Person LISA = new Person("Elisabeth Marie", "Simpson");
private static final Person MAGGIE = new Person("Margaret Eve", "Simpson");
private static final List<Person> FAMILY = new ArrayList<Person>() {{
    add(HOMER);
    add(MARGE);
    add(BART);
    add(LISA);
    add(MAGGIE);
}};

public static Person homer() { return HOMER; }

public static Person marge() { return MARGE; }

public static Person bart() { return BART; }

public static Person lisa() { return LISA; }

public static Person maggie() { return MAGGIE; }

// ...

```

Declaring dependencies of test fixtures

Similarly to the [Java Library Plugin](#), test fixtures expose an API and an implementation configuration:

Example 104. Declaring test fixture dependencies

lib/build.gradle.kts

```

dependencies {
    testImplementation("junit:junit:4.13")

    // API dependencies are visible to consumers when building
    testFixturesApi("org.apache.commons:commons-lang3:3.9")

    // Implementation dependencies are not leaked to consumers when building
    testFixturesImplementation("org.apache.commons:commons-text:1.6")
}

```

lib/build.gradle

```

dependencies {
    testImplementation 'junit:junit:4.13'
}

```

```
// API dependencies are visible to consumers when building
testFixturesApi 'org.apache.commons:commons-lang3:3.9'

// Implementation dependencies are not leaked to consumers when building
testFixturesImplementation 'org.apache.commons:commons-text:1.6'
}
```

It's worth noticing that if a dependency is an *implementation* dependency of test fixtures, then *when compiling tests that depend on those test fixtures*, the implementation dependencies will *not leak* into the compile classpath. This results in improved separation of concerns and better compile avoidance.

Consuming test fixtures of another project

Test fixtures are not limited to a single project. It is often the case that a dependent project tests also needs the test fixtures of the dependency. This can be achieved very easily using the `testFixtures` keyword:

Example 105. Adding a dependency on test fixtures of another project

build.gradle.kts

```
dependencies {
    implementation(project(":lib"))

    testImplementation("junit:junit:4.13")
    testImplementation(testFixtures(project(":lib")))
}
```

build.gradle

```
dependencies {
    implementation(project(":lib"))

    testImplementation 'junit:junit:4.13'
    testImplementation(testFixtures(project(":lib")))
}
```

Publishing test fixtures

One of the advantages of using the `java-test-fixtures` plugin is that test fixtures are published. By convention, test fixtures will be published with an artifact having the `test-fixtures` classifier. For both Maven and Ivy, an artifact with that classifier is simply published alongside the regular

artifacts. However, if you use the `maven-publish` or `ivy-publish` plugin, test fixtures are published as additional variants in [Gradle Module Metadata](#) and you can directly depend on test fixtures of external libraries in another Gradle project:

Example 106. Adding a dependency on test fixtures of an external library

build.gradle.kts

```
dependencies {
    // Adds a dependency on the test fixtures of Gson, however this
    // project doesn't publish such a thing
    functionalTest(testFixtures("com.google.code.gson:gson:2.8.5"))
}
```

build.gradle

```
dependencies {
    // Adds a dependency on the test fixtures of Gson, however this
    // project doesn't publish such a thing
    functionalTest testFixtures("com.google.code.gson:gson:2.8.5")
}
```

It's worth noting that if the external project is *not* publishing Gradle Module Metadata, then resolution will fail with an error indicating that such a variant cannot be found:

Output of `gradle dependencyInsight --configuration functionalTestClasspath --dependency gson`

```
> gradle dependencyInsight --configuration functionalTestClasspath --dependency gson

> Task :dependencyInsight
com.google.code.gson:gson:2.8.5 FAILED
  Failures:
    - Could not resolve com.google.code.gson:gson:2.8.5.
      - Unable to find a variant with the requested capability: feature 'test-
        fixtures':
          - Variant 'compile' provides 'com.google.code.gson:gson:2.8.5'
          - Variant 'enforced-platform-compile' provides
            'com.google.code.gson:gson-derived-enforced-platform:2.8.5'
          - Variant 'enforced-platform-runtime' provides
            'com.google.code.gson:gson-derived-enforced-platform:2.8.5'
          - Variant 'javadoc' provides 'com.google.code.gson:gson:2.8.5'
          - Variant 'platform-compile' provides 'com.google.code.gson:gson-
            derived-platform:2.8.5'
          - Variant 'platform-runtime' provides 'com.google.code.gson:gson-
            derived-platform:2.8.5'
          - Variant 'runtime' provides 'com.google.code.gson:gson:2.8.5'
```

- Variant 'sources' provides 'com.google.code.gson:gson:2.8.5'

```
com.google.code.gson:gson:2.8.5 FAILED
\--- functionalTestClasspath
```

A web-based, searchable dependency report is available by adding the `--scan` option.

```
BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

The error message mentions the missing `com.google.code.gson:gson-test-fixtures` capability, which is indeed not defined for this library. That's because by convention, for projects that use the `java-test-fixtures` plugin, Gradle automatically creates test fixtures variants with a capability whose name is the name of the main component, with the appendix `-test-fixtures`.

NOTE

If you publish your library and use test fixtures, but do not want to publish the fixtures, you can deactivate publishing of the *test fixtures variants* as shown below.

Example 107. Disable publishing of test fixtures variants

build.gradle.kts

```
val javaComponent = components["java"] as AdhocComponentWithVariants
javaComponent.withVariantsFromConfiguration(configurations["testFixturesApiElements"]) { skip() }
javaComponent.withVariantsFromConfiguration(configurations["testFixturesRuntimeElements"]) { skip() }
```

build.gradle

```
components.java.withVariantsFromConfiguration(configurations.testFixturesApiElements) { skip() }
components.java.withVariantsFromConfiguration(configurations.testFixturesRuntimeElements) { skip() }
```

Managing Dependencies of JVM Projects

This chapter explains how to apply basic dependency management concepts to JVM-based projects. For a detailed introduction to dependency management, see [dependency management in Gradle](#).

Dissecting a typical build script

Let's have a look at a very simple build script for a JVM-based project. It applies the [Java Library](#)

[plugin](#) which automatically introduces a standard project layout, provides tasks for performing typical work and adequate support for dependency management.

Example 108. [Dependency declarations for a JVM-based project](#)

build.gradle.kts

```
plugins {
    `java-library`
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("org.hibernate:hibernate-core:3.6.7.Final")
    testImplementation("junit:junit:4.+")
    api("com.google.guava:guava:23.0")
}
```

build.gradle

```
plugins {
    id 'java-library'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.hibernate:hibernate-core:3.6.7.Final'
    testImplementation 'junit:junit:4.+'
    api 'com.google.guava:guava:23.0'
}
```

The `Project.dependencies{} code block declares that Hibernate core 3.6.7.Final is required to compile the project's production source code. It also states that junit >= 4.0 is required to compile the project's tests. All dependencies are supposed to be looked up in the Maven Central repository as defined by Project.repositories{} code block. The following sections explain each aspect in more detail.`

Declaring module dependencies

There are various [types of dependencies](#) that you can declare. One such type is a *module*

dependency. A [module dependency](#) represents a dependency on a module with a specific version built outside the current build. Modules are usually stored in a repository, such as Maven Central, a corporate Maven or Ivy repository, or a directory in the local file system.

To define an module dependency, you add it to a [dependency configuration](#):

Example 109. Definition of a module dependency

build.gradle.kts

```
dependencies {  
    implementation("org.hibernate:hibernate-core:3.6.7.Final")  
}
```

build.gradle

```
dependencies {  
    implementation 'org.hibernate:hibernate-core:3.6.7.Final'  
}
```

To find out more about defining dependencies, have a look at [Declaring Dependencies](#).

Using dependency configurations

A [Configuration](#) is a named set of dependencies and artifacts. There are three main purposes for a *configuration*:

Declaring dependencies

A plugin uses configurations to make it easy for build authors to declare what other subprojects or external artifacts are needed for various purposes during the execution of tasks defined by the plugin. For example a plugin may need the Spring web framework dependency to compile the source code.

Resolving dependencies

A plugin uses configurations to find (and possibly download) inputs to the tasks it defines. For example Gradle needs to download Spring web framework JAR files from Maven Central.

Exposing artifacts for consumption

A plugin uses configurations to define what *artifacts* it generates for other projects to consume. For example the project would like to publish its compiled source code packaged in the JAR file to an in-house Artifactory repository.

With those three purposes in mind, let's take a look at a few of the [standard configurations defined by the Java Library Plugin](#).

implementation

The dependencies required to compile the production source of the project which *are not* part of the API exposed by the project. For example the project uses Hibernate for its internal persistence layer implementation.

api

The dependencies required to compile the production source of the project which *are* part of the API exposed by the project. For example the project uses Guava and exposes public interfaces with Guava classes in their method signatures.

testImplementation

The dependencies required to compile and run the test source of the project. For example the project decided to write test code with the test framework JUnit.

Various plugins add further standard configurations. You can also define your own custom configurations in your build via [Project.configurations{}](#). See [What are dependency configurations](#) for the details of defining and customizing dependency configurations.

Declaring common Java repositories

How does Gradle know where to find the files for external dependencies? Gradle looks for them in a *repository*. A repository is a collection of modules, organized by **group**, **name** and **version**. Gradle understands different [repository types](#), such as Maven and Ivy, and supports various ways of accessing the repository via HTTP or other protocols.

By default, Gradle does not define any repositories. You need to define at least one with the help of [Project.repositories{}](#) before you can use module dependencies. One option is use the Maven Central repository:

Example 110. Usage of Maven central repository

build.gradle.kts

```
repositories {  
    mavenCentral()  
}
```

build.gradle

```
repositories {  
    mavenCentral()  
}
```

You can also have repositories on the local file system. This works for both Maven and Ivy

repositories.

Example 111. Usage of a local Ivy directory

build.gradle.kts

```
repositories {  
    ivy {  
        // URL can refer to a local directory  
        url = uri("../local-repo")  
    }  
}
```

build.gradle

```
repositories {  
    ivy {  
        // URL can refer to a local directory  
        url = file("../local-repo")  
    }  
}
```

A project can have multiple repositories. Gradle will look for a dependency in each repository in the order they are specified, stopping at the first repository that contains the requested module.

To find out more about defining repositories, have a look at [Declaring Repositories](#).

Publishing artifacts

To learn more about publishing artifacts, have a look at [publishing plugins](#).

[1] The JUnit wiki contains a detailed description on how to work with JUnit categories: <https://github.com/junit-team/junit/wiki/Categories>.

[2] The TestNG documentation contains more details about test groups: https://testng.org/#_test_groups.

[3] The TestNG documentation contains more details about test ordering when working with `testng.xml` files: <http://testng.org/doc/documentation-main.html#testng-xml>.

JAVA TOOLCHAINS

Toolchains for JVM projects

Working on multiple projects can require interacting with multiple versions of the Java language. Even within a single project different parts of the codebase may be fixed to a particular language level due to backward compatibility requirements. This means different versions of the same tools (a toolchain) must be installed and managed on each machine that builds the project.

A **Java toolchain** is a set of tools to build and run Java projects, which is usually provided by the environment via local JRE or JDK installations. Compile tasks may use `javac` as their compiler, test and exec tasks may use the `java` command while `javadoc` will be used to generate documentation.

By default, Gradle uses the same Java toolchain for running Gradle itself and building JVM projects. However, this may only sometimes be desirable. Building projects with different Java versions on different developer machines and CI servers may lead to unexpected issues. Additionally, you may want to build a project using a Java version that is not supported for running Gradle.

In order to improve reproducibility of the builds and make build requirements clearer, Gradle allows configuring toolchains on both project and task levels. You can also control the JVM used to run Gradle itself using the [Daemon JVM criteria](#).

Toolchains for projects

You can define what toolchain to use for a project by stating the Java language version in the `java` extension block:

build.gradle.kts

```
java {  
    toolchain {  
        languageVersion = JavaLanguageVersion.of(17)  
    }  
}
```

build.gradle

```
java {  
    toolchain {  
        languageVersion = JavaLanguageVersion.of(17)  
    }  
}
```

Executing the build (e.g. using `gradle check`) will now handle several things for you and others running your build:

1. Gradle configures all compile, test and javadoc tasks to use the defined toolchain.
2. Gradle detects [locally installed toolchains](#).
3. Gradle chooses a toolchain matching the requirements (any Java 17 toolchain for the example above).
4. If no matching toolchain is found, Gradle can automatically download a matching one based on the configured [toolchain download repositories](#).

NOTE

Toolchain support is available in the Java plugins and for the tasks they define.

For the Groovy plugin, compilation is supported but not yet Groovydoc generation.
For the Scala plugin, compilation and Scaladoc generation are supported.

Selecting toolchains by vendor

In case your build has specific requirements from the used JRE/JDK, you may want to define the vendor for the toolchain as well. `JvmVendorSpec` has a list of well-known JVM vendors recognized by Gradle. The advantage is that Gradle can handle any inconsistencies across JDK versions in how exactly the JVM encodes the vendor information.

build.gradle.kts

```
java {  
    toolchain {  
        languageVersion = JavaLanguageVersion.of(11)  
    }  
}
```

build.gradle

```
java {  
    toolchain {  
        languageVersion = JavaLanguageVersion.of(11)  
    }  
}
```

If the vendor you want to target is not a known vendor, you can still restrict the toolchain to those matching the `java.vendor` system property of the available toolchains.

The following snippet uses filtering to include a subset of available toolchains. This example only includes toolchains whose `java.vendor` property contains the given match string. The matching is

done in a case-insensitive manner.

build.gradle.kts

```
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(11)
        vendor = JvmVendorSpec.matching("customString")
    }
}
```

build.gradle

```
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(11)
        vendor = JvmVendorSpec.matching("customString")
    }
}
```

Selecting toolchains by virtual machine implementation

If your project requires a specific implementation, you can filter based on the implementation as well. Currently available implementations to choose from are:

VENDOR_SPECIFIC

Acts as a placeholder and matches any implementation from any vendor (e.g. hotspot, zulu, ...)

J9

Matches only virtual machine implementations using the OpenJ9/IBM J9 runtime engine.

For example, to use an [IBM](#) JVM, distributed via [AdoptOpenJDK](#), you can specify the filter as shown in the example below.

build.gradle.kts

```
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(11)
        vendor = JvmVendorSpec.IBM
        implementation = JvmImplementation.J9
    }
}
```

```
}
```

build.gradle

```
java {  
    toolchain {  
        languageVersion = JavaLanguageVersion.of(11)  
        vendor = JvmVendorSpec.IBM  
        implementation = JvmImplementation.J9  
    }  
}
```

NOTE

The Java major version, the vendor (if specified) and implementation (if specified) will be tracked as an input for compilation and test execution.

Configuring toolchain specifications

Gradle allows configuring multiple properties that affect the selection of a toolchain, such as language version or vendor. Even though these properties can be configured independently, the configuration must follow certain rules in order to form a *valid* specification.

A `JavaToolchainSpec` is considered *valid* in two cases:

1. when no properties have been set, i.e. the specification is *empty*;
2. when `languageVersion` has been set, optionally followed by setting any other property.

In other words, if a vendor or an implementation are specified, they must be accompanied by the language version. Gradle distinguishes between toolchain specifications that configure the language version and the ones that do not. A specification without a language version, in most cases, would be treated as a one that selects the toolchain of the current build.

Usage of *invalid* instances of `JavaToolchainSpec` results in a build error since Gradle 8.0.

Toolchains for tasks

In case you want to tweak which toolchain is used for a specific task, you can specify the exact tool a task is using. For example, the `Test` task exposes a `JavaLauncher` property that defines which java executable to use for launching the tests.

In the example below, we configure all java compilation tasks to use Java 8. Additionally, we introduce a new `Test` task that will run our unit tests using a JDK 17.

list/build.gradle.kts

```
tasks.withType<JavaCompile>().configureEach {
    javaCompiler = javaToolchains.compilerFor {
        languageVersion = JavaLanguageVersion.of(8)
    }
}

tasks.register<Test>("testsOn17") {
    javaLauncher = javaToolchains.launcherFor {
        languageVersion = JavaLanguageVersion.of(17)
    }
}
```

list/build.gradle

```
tasks.withType(JavaCompile).configureEach {
    javaCompiler = javaToolchains.compilerFor {
        languageVersion = JavaLanguageVersion.of(8)
    }
}

tasks.register('testsOn17', Test) {
    javaLauncher = javaToolchains.launcherFor {
        languageVersion = JavaLanguageVersion.of(17)
    }
}
```

In addition, in the `application` subproject, we add another Java execution task to run our application with JDK 17.

application/build.gradle.kts

```
tasks.register<JavaExec>("runOn17") {
    javaLauncher = javaToolchains.launcherFor {
        languageVersion = JavaLanguageVersion.of(17)
    }

    classpath = sourceSets["main"].runtimeClasspath
    mainClass = application.mainClass
}
```

application/build.gradle

```
tasks.register('runOn17', JavaExec) {
    javaLauncher = javaToolchains.launcherFor {
        languageVersion = JavaLanguageVersion.of(17)
    }

    classpath = sourceSets.main.runtimeClasspath
    mainClass = application.mainClass
}
```

Depending on the task, a JRE might be enough while for other tasks (e.g. compilation), a JDK is required. By default, Gradle prefers installed JDKs over JREs if they can satisfy the requirements.

Toolchains tool providers can be obtained from the `javaToolchains` extension.

Three tools are available:

- A `JavaCompiler` which is the tool used by the `JavaCompile` task
- A `JavaLauncher` which is the tool used by the `JavaExec` or `Test` tasks
- A `JavadocTool` which is the tool used by the `Javadoc` task

Integration with tasks relying on a Java executable or Java home

Any task that can be configured with a path to a Java executable, or a Java home location, can benefit from toolchains.

While you will not be able to wire a toolchain tool directly, they all have the metadata that gives access to their full path or to the path of the Java installation they belong to.

For example, you can configure the `java` executable for a task as follows:

build.gradle.kts

```
val launcher = javaToolchains.launcherFor {
    languageVersion = JavaLanguageVersion.of(11)
}

tasks.sampleTask {
    javaExecutable = launcher.map { it.executablePath }
}
```

build.gradle

```
def launcher = javaToolchains.launcherFor {  
    languageVersion = JavaLanguageVersion.of(11)  
}  
  
tasks.named('sampleTask') {  
    javaExecutable = launcher.map { it.executablePath }  
}
```

As another example, you can configure the *Java Home* for a task as follows:

build.gradle.kts

```
val launcher = javaToolchains.launcherFor {  
    languageVersion = JavaLanguageVersion.of(11)  
}  
  
tasks.anotherSampleTask {  
    javaHome = launcher.map { it.metadata.installationPath }  
}
```

build.gradle

```
def launcher = javaToolchains.launcherFor {  
    languageVersion = JavaLanguageVersion.of(11)  
}  
  
tasks.named('anotherSampleTask') {  
    javaHome = launcher.map { it.metadata.installationPath }  
}
```

If you require a path to a specific tool such as Java compiler, you can obtain it as follows:

build.gradle.kts

```
val compiler = javaToolchains.compilerFor {  
    languageVersion = JavaLanguageVersion.of(11)  
}
```

```
tasks.yetAnotherSampleTask {  
    javaCompilerExecutable = compiler.map { it.executablePath }  
}
```

build.gradle

```
def compiler = javaToolchains.compilerFor {  
    languageVersion = JavaLanguageVersion.of(11)  
}  
  
tasks.named('yetAnotherSampleTask') {  
    javaCompilerExecutable = compiler.map { it.executablePath }  
}
```

WARNING

The examples above use tasks with [RegularFileProperty](#) and [DirectoryProperty](#) properties which allow lazy configuration. Doing respectively `launcher.get().executablePath`, `launcher.get().metadata.installationPath` or `compiler.get().executablePath` instead will give you the full path for the given toolchain but note that this may realize (and provision) a toolchain eagerly.

Auto-detection of installed toolchains

By default, Gradle automatically detects local JRE/JDK installations so no further configuration is required by the user. The following is a list of common package managers, tools, and locations that are supported by the JVM auto-detection.

JVM auto-detection knows how to work with:

- Operation-system specific locations: Linux, macOS, Windows
- Package Managers: [Asdf-vm](#), [Jabba](#), [SDKMAN!](#)
- [Maven Toolchain](#) specifications
- [IntelliJ IDEA](#) installations

Among the set of all detected JRE/JDK installations, one will be picked according to the [Toolchain Precedence Rules](#).

NOTE

Whether you are using toolchain auto-detection or you are configuring [Custom toolchain locations](#), installations that are non-existing or without a `bin/java` executable will be ignored with a warning, but they won't generate an error.

How to disable auto-detection

In order to disable auto-detection, you can use the `org.gradle.java.installations.auto-detect` Gradle property:

- Either start gradle using `-Porg.gradle.java.installations.auto-detect=false`
- Or put `org.gradle.java.installations.auto-detect=false` into your `gradle.properties` file.

Auto-provisioning

If Gradle can't find a locally available toolchain that matches the requirements of the build, it can automatically download one (as long as a toolchain download repository has been configured; for detail, see [relevant section](#)). Gradle installs the downloaded JDKs in the [Gradle User Home](#).

NOTE

Gradle only downloads JDK versions for GA releases. There is no support for downloading early access versions.

Once installed in the [Gradle User Home](#), a provisioned JDK becomes one of the JDKs visible to [auto-detection](#) and can be used by any subsequent builds, just like any other JDK installed on the system.

Since auto-provisioning only kicks in when auto-detection fails to find a matching JDK, auto-provisioning can only download new JDKs and is in no way involved in updating any of the already installed ones. None of the auto-provisioned JDKs will ever be revisited and automatically updated by auto-provisioning, even if there is a newer minor version available for them.

Toolchain Download Repositories

Toolchain download repository definitions are added to a build by applying specific settings plugins. For details on writing such plugins, consult the [Toolchain Resolver Plugins](#) page.

One example of a toolchain resolver plugin is the [Foojay Toolchains Plugin](#), based on the [foojay Disco API](#). It even has a convention variant, which automatically takes care of all the needed configuration, just by being applied:

settings.gradle.kts

```
plugins {  
    id("org.gradle.toolchains.foojay-resolver-convention").version("0.9.0")  
}
```

settings.gradle

```
plugins {  
    id 'org.gradle.toolchains.foojay-resolver-convention' version '0.9.0'  
}
```

For advanced or highly specific configurations, a custom toolchain resolver plugin should be used.

In general, when applying toolchain resolver plugins, the toolchain download resolvers provided

by them also need to be configured. Let's illustrate with an example. Consider two toolchain resolver plugins applied by the build:

- One is the Foojay plugin mentioned above, which downloads toolchains via the `FoojayToolchainResolver` it provides.
- The other contains a **FICTITIOUS** resolver named `MadeUpResolver`.

The following example uses these toolchain resolvers in a build via the `toolchainManagement` block in the settings file:

settings.gradle.kts

```
toolchainManagement {  
    jvm { ❶  
        javaRepositories {  
            repository("foojay") { ❷  
                resolverClass =  
org.gradle.toolchains.foojay.FoojayToolchainResolver::class.java  
            }  
            repository("made_up") { ❸  
                resolverClass = MadeUpResolver::class.java  
                credentials {  
                    username = "user"  
                    password = "password"  
                }  
                authentication {  
                    create<DigestAuthentication>("digest")  
                } ❹  
            }  
        }  
    }  
}
```

settings.gradle

```
toolchainManagement {  
    jvm { ❶  
        javaRepositories {  
            repository('foojay') { ❷  
                resolverClass = org.gradle.toolchains.foojay  
.FoojayToolchainResolver  
            }  
            repository('made_up') { ❸  
                resolverClass = MadeUpResolver  
                credentials {  
                    username = "user"  
                    password = "password"  
                }  
            }  
        }  
    }  
}
```

```

        }
        authentication {
            digest(BasicAuthentication)
        } ④
    }
}
}
}
}

```

- ① In the `toolchainManagement` block, the `jvm` block contains configuration for Java toolchains.
- ② The `javaRepositories` block defines named Java toolchain repository configurations. Use the `resolverClass` property to link these configurations to plugins.
- ③ Toolchain declaration order matters. Gradle downloads from the first repository that provides a match, starting with the first repository in the list.
- ④ You can configure toolchain repositories with the same set of [authentication and authorization options](#) used for dependency management.

WARNING

The `jvm` block in `toolchainManagement` only resolves after applying a toolchain resolver plugin.

Viewing and debugging toolchains

Gradle can display the list of all detected toolchains including their metadata.

For example, to show all toolchains of a project, run:

```
gradle -q javaToolchains
```

Output of `gradle -q javaToolchains`

```

> gradle -q javaToolchains

+ Options
  | Auto-detection:      Enabled
  | Auto-download:      Enabled

+ AdoptOpenJDK 1.8.0_242
  | Location:            /Users/username/myJavaInstalls/8.0.242.hs-adpt/jre
  | Language Version:   8
  | Vendor:              AdoptOpenJDK
  | Architecture:       x86_64
  | Is JDK:              false
  | Detected by:         Gradle property 'org.gradle.java.installations.paths'

+ Microsoft JDK 16.0.2+7
  | Location:            /Users/username/.sdkman/candidates/java/16.0.2.7.1-ms

```

```
| Language Version: 16
| Vendor: Microsoft
| Architecture: aarch64
| Is JDK: true
| Detected by: SDKMAN!
```

+ OpenJDK 15-ea

```
| Location: /Users/user/customJdks/15.ea.21-open
| Language Version: 15
| Vendor: AdoptOpenJDK
| Architecture: x86_64
| Is JDK: true
| Detected by: environment variable 'JDK16'
```

+ Oracle JDK 1.7.0_80

```
| Location: /Library/Java/JavaVirtualMachines/jdk1.7.0_80.jdk/Contents/Home/jre
| Language Version: 7
| Vendor: Oracle
| Architecture: x86_64
| Is JDK: false
| Detected by: MacOS java_home
```

This can help to debug which toolchains are available to the build, how they are detected and what kind of metadata Gradle knows about those toolchains.

Disabling auto provisioning

In order to disable auto-provisioning, you can use the `org.gradle.java.installations.auto-download` Gradle property:

- Either start gradle using `-Porg.gradle.java.installations.auto-download=false`
- Or put `org.gradle.java.installations.auto-download=false` into a `gradle.properties` file.

NOTE

After disabling the auto provisioning, ensure that the specified JRE/JDK version in the build file is already installed locally. Then, stop the Gradle daemon so that it can be reinitialized for the next build. You can use the `./gradlew --stop` command to stop the daemon process.

Removing an auto-provisioned toolchain

When removing an auto-provisioned toolchain is necessary, remove the relevant toolchain located in the `/jdk` directory within the [Gradle User Home](#).

NOTE

The [Gradle Daemon](#) caches information about your project, including configuration details such as toolchain paths or versions. Changes to a project's toolchain configuration might only occur once the Gradle Daemon is restarted. It is recommended to [stop the Gradle Daemon](#) to ensure that Gradle updates the configuration for subsequent builds.

Custom toolchain locations

If auto-detecting local toolchains is not sufficient or disabled, there are additional ways you can let Gradle know about installed toolchains.

If your setup already provides environment variables pointing to installed JVMs, you can also let Gradle know about which environment variables to take into account. Assuming the environment variables `JDK8` and `JRE17` point to valid java installations, the following instructs Gradle to resolve those environment variables and consider those installations when looking for a matching toolchain.

```
org.gradle.java.installations.fromEnv=JDK8,JRE17
```

Additionally, you can provide a comma-separated list of paths to specific installations using the `org.gradle.java.installations.paths` property. For example, using the following in your `gradle.properties` will let Gradle know which directories to look at when detecting toolchains. Gradle will treat these directories as possible installations but will not descend into any nested directories.

```
org.gradle.java.installations.paths=/custom/path/jdk1.8,/shared/jre11
```

NOTE

Gradle does not prioritize custom toolchains over [auto-detected](#) toolchains. If you enable auto-detection in your build, custom toolchains extend the set of toolchain locations. Gradle picks a toolchain according to the [precedence rules](#).

Toolchain installations precedence

Gradle will sort all the JDK/JRE installations matching the toolchain specification of the build and will pick the first one. Sorting is done based on the following rules:

1. the installation currently running Gradle is preferred over any other
2. JDK installations are preferred over JRE ones
3. certain vendors take precedence over others; their ordering (from the highest priority to lowest):
 - a. ADOPTIUM
 - b. ADOPTOPENJDK
 - c. AMAZON
 - d. APPLE
 - e. AZUL
 - f. BELLSOFT
 - g. GRAAL_VM
 - h. HEWLETT_PACKARD

- i. IBM
 - j. JETBRAINS
 - k. MICROSOFT
 - l. ORACLE
 - m. SAP
 - n. TENCENT
 - o. everything else
4. higher major versions take precedence over lower ones
 5. higher minor versions take precedence over lower ones
 6. installation paths take precedence according to their lexicographic ordering (last resort criteria for deterministically deciding between installations of the same type, from the same vendor and with the same version)

All these rules are applied as multilevel sorting criteria, **in the order shown**. Let's illustrate with an example. A toolchain specification requests Java version 17. Gradle detects the following matching installations:

- Oracle JRE v17.0.1
- Oracle JDK v17.0.0
- Microsoft JDK 17.0.0
- Microsoft JRE 17.0.1
- Microsoft JDK 17.0.1

Assume that Gradle runs on a major Java version other than 17. Otherwise, that installation would have priority.

When we apply the above rules to sort this set we will end up with following ordering:

1. Microsoft JDK 17.0.1
2. Microsoft JDK 17.0.0
3. Oracle JDK v17.0.0
4. Microsoft JRE v17.0.1
5. Oracle JRE v17.0.1

Gradle prefers JDKs over JREs, so the JREs come last. Gradle prefers the Microsoft vendor over Oracle, so the Microsoft installations come first. Gradle prefers higher version numbers, so JDK 17.0.1 comes before JDK 17.0.0.

So Gradle picks the first match in this order: Microsoft JDK 17.0.1.

Toolchains for plugin authors

When creating a plugin or a task that uses toolchains, it is essential to provide sensible defaults and

allow users to override them.

For JVM projects, it is usually safe to assume that the **java** plugin has been applied to the project. The **java** plugin is automatically applied for the core Groovy and Scala plugins, as well as for the Kotlin plugin. In such a case, using the toolchain defined via the **java** extension as a default value for the tool property is appropriate. This way, the users will need to configure the toolchain only once on the project level.

The example below showcases how to use the default toolchain as convention while allowing users to individually configure the toolchain per task.

build.gradle.kts

```
abstract class CustomTaskUsingToolchains : DefaultTask() {

    @get:Nested
    abstract val launcher: Property<JavaLauncher> ❶

    init {
        val toolchain =
project.extensions.getByType<JavaPluginExtension>().toolchain ❷
        val defaultLauncher = javaToolchainService.launcherFor(toolchain) ❸
        launcher.convention(defaultLauncher) ❹
    }

    @TaskAction
    fun showConfiguredToolchain() {
        println(launcher.get().executablePath)
        println(launcher.get().metadata.installationPath)
    }

    @get:Inject
    protected abstract val javaToolchainService: JavaToolchainService
}
```

build.gradle

```
abstract class CustomTaskUsingToolchains extends DefaultTask {

    @Nested
    abstract Property<JavaLauncher> getLauncher() ❶

    CustomTaskUsingToolchains() {
        def toolchain = project.extensions.getByType(JavaPluginExtension
.class).toolchain ❷
        Provider<JavaLauncher> defaultLauncher = getJavaToolchainService()
.launcherFor(toolchain) ❸
    }
}
```

```

        launcher.convention(defaultLauncher) ④
    }

    @TaskAction
    def showConfiguredToolchain() {
        println launcher.get().executablePath
        println launcher.get().metadata.installationPath
    }

    @Inject
    protected abstract JavaToolchainService getJavaToolchainService()
}

```

- ① We declare a `JavaLauncher` property on the task. The property must be marked as a `@Nested input` to make sure the task is responsive to toolchain changes.
- ② We obtain the toolchain spec from the `java` extension to use it as a default.
- ③ Using the `JavaToolchainService` we get a provider of the `JavaLauncher` that matches the toolchain.
- ④ Finally, we wire the launcher provider as a convention for our property.

In a project where the `java` plugin was applied, we can use the task as follows:

build.gradle.kts

```

plugins {
    java
}

java {
    toolchain { ①
        languageVersion = JavaLanguageVersion.of(8)
    }
}

tasks.register<CustomTaskUsingToolchains>("showDefaultToolchain") ②

tasks.register<CustomTaskUsingToolchains>("showCustomToolchain") {
    launcher = javaToolchains.launcherFor { ③
        languageVersion = JavaLanguageVersion.of(17)
    }
}

```

build.gradle

```

plugins {

```

```

    id 'java'
}

java {
    toolchain { ①
        languageVersion = JavaLanguageVersion.of(8)
    }
}

tasks.register('showDefaultToolchain', CustomTaskUsingToolchains) ②

tasks.register('showCustomToolchain', CustomTaskUsingToolchains) {
    launcher = javaToolchains.launcherFor { ③
        languageVersion = JavaLanguageVersion.of(17)
    }
}

```

- ① The toolchain defined on the `java` extension is used by default to resolve the launcher.
- ② The custom task without additional configuration will use the default Java 8 toolchain.
- ③ The other task overrides the value of the launcher by selecting a different toolchain using `javaToolchains` service.

When a task needs access to toolchains without the `java` plugin being applied the toolchain service can be used directly. If an `unconfigured` toolchain spec is provided to the service, it will always return a tool provider for the toolchain that is running Gradle. This can be achieved by passing an empty lambda when requesting a tool: `javaToolchainService.launcherFor({})`.

You can find more details on defining custom tasks in the [Authoring tasks](#) documentation.

Toolchains limitations

Gradle may detect toolchains incorrectly when it's running in a JVM compiled against `musl`, an [alternative implementation](#) of the C standard library. JVMs compiled against `musl` can sometimes override the `LD_LIBRARY_PATH` environment variable to control dynamic library resolution. This can influence forked java processes launched by Gradle, resulting in unexpected behavior.

As a consequence, using multiple java toolchains is discouraged in environments with the `musl` library. This is the case in most Alpine distributions — consider using another distribution, like Ubuntu, instead. If you are using a single toolchain, the JVM running Gradle, to build and run your application, you can safely ignore this limitation.

Toolchain Resolver Plugins

In Gradle version 7.6 and above, Gradle provides a way to define Java toolchain auto-provisioning logic in plugins. This page explains how to author a toolchain resolver plugin. For details on how toolchain auto-provisioning interacts with these plugins, see [Toolchains](#).

Provide a download URI

Toolchain resolver plugins provide logic to map a [toolchain request](#) to a [download response](#). At the moment the download response only contains a download URL, but may be extended in the future.

WARNING

For the download URL only secure protocols like [https](#) are accepted. This is required to make sure no one can tamper with the download in flight.

The plugins provide the mapping logic via an implementation of [JavaToolchainResolver](#):

JavaToolchainResolverImplementation.java

```
public abstract class JavaToolchainResolverImplementation
    implements JavaToolchainResolver { ①

    public Optional<JavaToolchainDownload> resolve(JavaToolchainRequest request) { ②
        return Optional.empty(); // custom mapping logic goes here instead
    }
}
```

① This class is **abstract** because [JavaToolchainResolver](#) is a [build service](#). Gradle provides dynamic implementations for certain abstract methods at runtime.

② The mapping method returns a download response wrapped in an [Optional](#). If the resolver implementation can't provide a matching toolchain, the enclosing [Optional](#) contains an empty value.

Register the resolver in a plugin

Use a settings plugin ([Plugin<Settings>](#)) to register the [JavaToolchainResolver](#) implementation:

JavaToolchainResolverPlugin.java

```
public abstract class JavaToolchainResolverPlugin implements Plugin<Settings> { ①
    @Inject
    protected abstract JavaToolchainResolverRegistry getToolchainResolverRegistry();
    ②

    public void apply(Settings settings) {
        settings.getPluginManager().apply("jvm-toolchain-management"); ③

        JavaToolchainResolverRegistry registry = getToolchainResolverRegistry();
        registry.register(JavaToolchainResolverImplementation.class);
    }
}
```

① The plugin uses [property injection](#), so it must be **abstract** and a settings plugin.

② To register the resolver implementation, use [property injection](#) to access the [JavaToolchainResolverRegistry](#) Gradle service.

- ③ Resolver plugins must apply the `jvm-toolchain-management` base plugin. This dynamically adds the `jvm` block to `toolchainManagement`, which makes registered toolchain repositories usable from the build.

JVM PLUGINS

The Java Library Plugin

The Java Library plugin expands the capabilities of the [Java Plugin \(java\)](#) by providing specific knowledge about Java libraries. In particular, a Java library exposes an API to consumers (i.e., other projects using the Java or the Java Library plugin). All the source sets, tasks and configurations exposed by the Java plugin are implicitly available when using this plugin.

Usage

To use the Java Library plugin, include the following in your build script:

Example 112. Using the Java Library plugin

build.gradle.kts

```
plugins {  
    `java-library`  
}
```

build.gradle

```
plugins {  
    id 'java-library'  
}
```

API and implementation separation

The key difference between the standard Java plugin and the Java Library plugin is that the latter introduces the concept of an *API* exposed to consumers. A library is a Java component meant to be consumed by other components. It's a very common use case in multi-project builds, but also as soon as you have external dependencies.

The plugin exposes two [configurations](#) that can be used to declare dependencies: `api` and `implementation`. The `api` configuration should be used to declare dependencies which are exported by the library API, whereas the `implementation` configuration should be used to declare dependencies which are internal to the component.

Example 113. *Declaring API and implementation dependencies*

build.gradle.kts

```
dependencies {  
    api("org.apache.httpcomponents:httpclient:4.5.7")  
    implementation("org.apache.commons:commons-lang3:3.5")  
}
```

build.gradle

```
dependencies {  
    api 'org.apache.httpcomponents:httpclient:4.5.7'  
    implementation 'org.apache.commons:commons-lang3:3.5'  
}
```

Dependencies appearing in the **api** configurations will be transitively exposed to consumers of the library, and as such will appear on the compile classpath of consumers. Dependencies found in the **implementation** configuration will, on the other hand, not be exposed to consumers, and therefore not leak into the consumers' compile classpath. This comes with several benefits:

- dependencies do not leak into the compile classpath of consumers anymore, so you will never accidentally depend on a transitive dependency
- faster compilation thanks to reduced classpath size
- less recompilations when implementation dependencies change: consumers would not need to be recompiled
- cleaner publishing: when used in conjunction with the new **maven-publish** plugin, Java libraries produce POM files that distinguish exactly between what is required to compile against the library and what is required to use the library at runtime (in other words, don't mix what is needed to compile the library itself and what is needed to compile against the library).

NOTE

The **compile** and **runtime** configurations have been removed with Gradle 7.0. Please refer to the [upgrade guide](#) how to migrate to **implementation** and **api** configurations`.

If your build consumes a published module with POM metadata, the Java and Java Library plugins both honor api and implementation separation through the scopes used in the POM. Meaning that the compile classpath only includes Maven **compile** scoped dependencies, while the runtime classpath adds the Maven **runtime** scoped dependencies as well.

This often does not have an effect on modules published with Maven, where the POM that defines the project is directly published as metadata. There, the compile scope includes both dependencies that were required to compile the project (i.e. implementation dependencies) and dependencies

required to compile against the published library (i.e. API dependencies). For most published libraries, this means that all dependencies belong to the compile scope. If you encounter such an issue with an existing library, you can consider a [component metadata rule](#) to fix the incorrect metadata in your build. However, as mentioned above, if the library is published with Gradle, the produced POM file only puts `api` dependencies into the compile scope and the remaining `implementation` dependencies into the runtime scope.

If your build consumes modules with Ivy metadata, you might be able to activate api and implementation separation as described [here](#) if all modules follow a certain structure.

NOTE

Separating compile and runtime scope of modules is active by default in Gradle 5.0+. In Gradle 4.6+, you need to activate it by adding `enableFeaturePreview('IMPROVED_POM_SUPPORT')` in `settings.gradle`.

Recognizing API and implementation dependencies

This section will help you identify API and Implementation dependencies in your code using simple rules of thumb. The first of these is:

- Prefer the `implementation` configuration over `api` when possible

This keeps the dependencies off of the consumer's compilation classpath. In addition, the consumers will immediately fail to compile if any implementation types accidentally leak into the public API.

So when should you use the `api` configuration? An API dependency is one that contains at least one type that is exposed in the library binary interface, often referred to as its ABI (Application Binary Interface). This includes, but is not limited to:

- types used in super classes or interfaces
- types used in public method parameters, including generic parameter types (where *public* is something that is visible to compilers. I.e. , *public*, *protected* and *package private* members in the Java world)
- types used in public fields
- public annotation types

By contrast, any type that is used in the following list is irrelevant to the ABI, and therefore should be declared as an `implementation` dependency:

- types exclusively used in method bodies
- types exclusively used in private members
- types exclusively found in internal classes (future versions of Gradle will let you declare which packages belong to the public API)

The following class makes use of a couple of third-party libraries, one of which is exposed in the class's public API and the other is only used internally. The import statements don't help us determine which is which, so we have to look at the fields, constructors and methods instead:

Example: Making the difference between API and implementation

src/main/java/org/gradle/HttpClientWrapper.java

```
// The following types can appear anywhere in the code
// but say nothing about API or implementation usage
import org.apache.commons.lang3.exception.ExceptionUtils;
import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.HttpStatus;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpGet;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.UnsupportedEncodingException;

public class HttpClientWrapper {

    private final HttpClient client; // private member: implementation details

    // HttpClient is used as a parameter of a public method
    // so "leaks" into the public API of this component
    public HttpClientWrapper(HttpClient client) {
        this.client = client;
    }

    // public methods belongs to your API
    public byte[] doRawGet(String url) {
        HttpGet request = new HttpGet(url);
        try {
            HttpEntity entity = doGet(request);
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            entity.writeTo(baos);
            return baos.toByteArray();
        } catch (Exception e) {
            ExceptionUtils.rethrow(e); // this dependency is internal only
        } finally {
            request.releaseConnection();
        }
        return null;
    }

    // HttpGet and HttpEntity are used in a private method, so they don't belong to
    // the API
    private HttpEntity doGet(HttpGet get) throws Exception {
        HttpResponse response = client.execute(get);
        if (response.getStatusLine().getStatusCode() != HttpStatus.SC_OK) {
            System.err.println("Method failed: " + response.getStatusLine());
        }
        return response.getEntity();
    }
}
```

```
}  
}
```

The *public* constructor of `HttpClientWrapper` uses `HttpClient` as a parameter, so it is exposed to consumers and therefore belongs to the API. Note that `HttpGet` and `HttpEntity` are used in the signature of a *private* method, and so they don't count towards making `HttpClient` an API dependency.

On the other hand, the `ExceptionUtils` type, coming from the `commons-lang` library, is only used in a method body (not in its signature), so it's an implementation dependency.

Therefore, we can deduce that `httpClient` is an API dependency, whereas `commons-lang` is an implementation dependency. This conclusion translates into the following declaration in the build script:

Example 114. Declaring API and implementation dependencies

build.gradle.kts

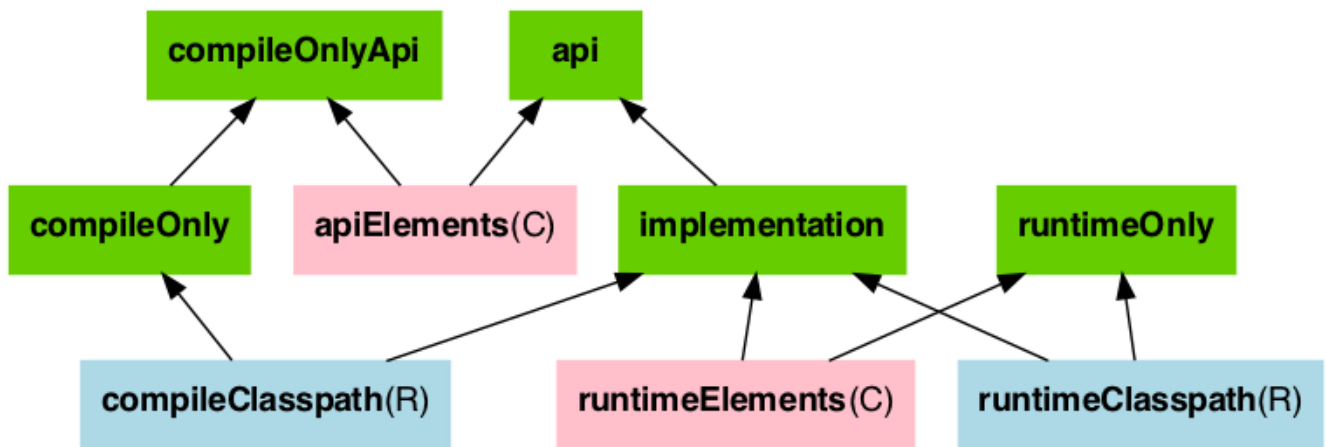
```
dependencies {  
    api("org.apache.httpcomponents:httpClient:4.5.7")  
    implementation("org.apache.commons:commons-lang3:3.5")  
}
```

build.gradle

```
dependencies {  
    api 'org.apache.httpcomponents:httpClient:4.5.7'  
    implementation 'org.apache.commons:commons-lang3:3.5'  
}
```

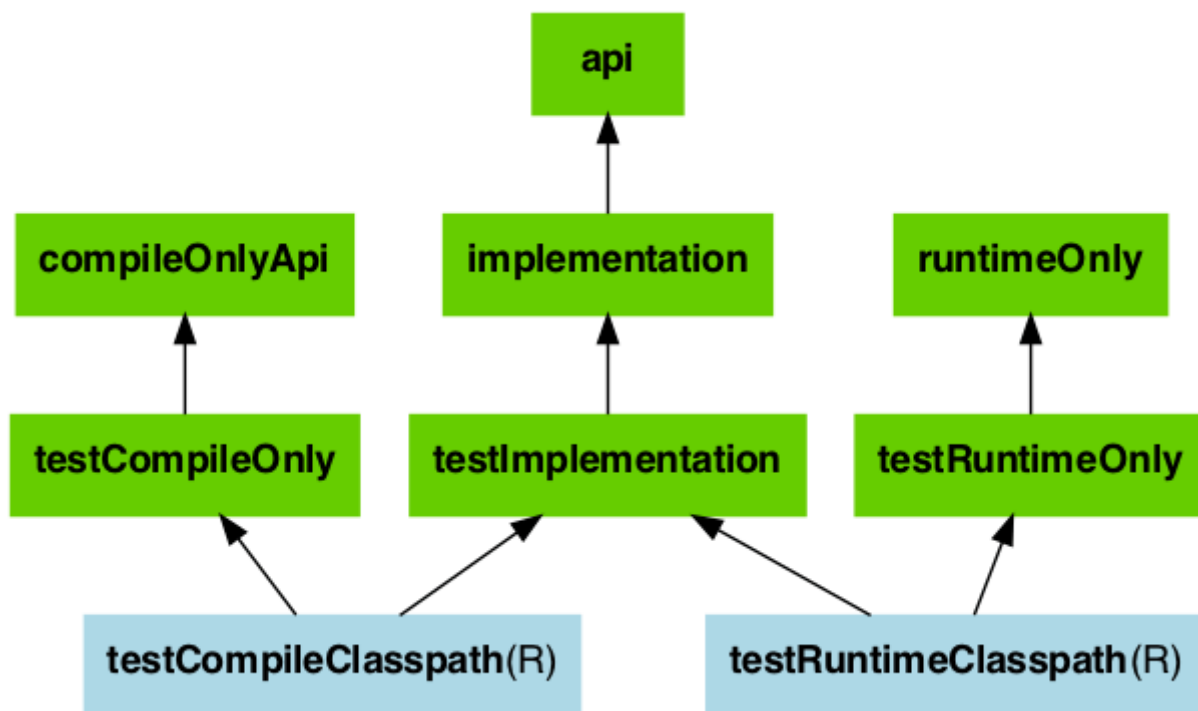
The Java Library plugin configurations

The following graph describes how configurations are setup when the Java Library plugin is in use.



- The configurations in *green* are the ones a user should use to declare dependencies
- The configurations in *pink* are the ones used when a component compiles, or runs against the library
- The configurations in *blue* are internal to the component, for its own use

And the next graph describes the test configurations setup:



The role of each configuration is described in the following tables:

Table 9. Java Library plugin - configurations used to declare dependencies

Configura tion name	Role	Consu mable?	Resolv able?	Description
annotation Processor	Declaring annotation processors	no	yes	This configuration is used to declare annotation processors, ensuring they are available during the compile phase for code generation.

Configura tion name	Role	Consu mable?	Resolv able?	Description
api	Declaring API dependencies	no	no	This is where you declare dependencies which are transitively exported to consumers, for compile time and runtime.
implementa tion	Declaring implementation dependencies	no	no	This is where you declare dependencies which are purely internal and not meant to be exposed to consumers (they are still exposed to consumers at runtime).
compileOnl y	Declaring compile only dependencies	no	no	This is where you declare dependencies which are required at compile time, but not at runtime. This typically includes dependencies which are shaded when found at runtime.
compileOnl yApi	Declaring compile only API dependencies	no	no	This is where you declare dependencies which are required at compile time by your module and consumers, but not at runtime. This typically includes dependencies which are shaded when found at runtime.
runtimeOnl y	Declaring runtime dependencies	no	no	This is where you declare dependencies which are only required at runtime, and not at compile time.
testImplem entation	Test dependencies	no	no	This is where you declare dependencies which are used to compile tests.
testCompil eOnly	Declaring test compile only dependencies	no	no	This is where you declare dependencies which are only required at test compile time, but should not leak into the runtime. This typically includes dependencies which are shaded when found at runtime.
testRuntim eOnly	Declaring test runtime dependencies	no	no	This is where you declare dependencies which are only required at test runtime, and not at test compile time.

Table 10. Java Library plugin — configurations used by consumers

Configurat ion name	Role	Consu mable?	Resolv able?	Description
apiElements	For compiling against this library	yes	no	This configuration is meant to be used by consumers, to retrieve all the elements necessary to compile against this library.
runtimeElem ents	For executing this library	yes	no	This configuration is meant to be used by consumers, to retrieve all the elements necessary to run against this library.

Table 11. Java Library plugin - configurations used by the library itself

Configuration name	Role	Consumable?	Resolvable?	Description
compileClasspath	For compiling this library	no	yes	This configuration contains the compile classpath of this library, and is therefore used when invoking the java compiler to compile it.
runtimeClasspath	For executing this library	no	yes	This configuration contains the runtime classpath of this library
testCompileClasspath	For compiling the tests of this library	no	yes	This configuration contains the test compile classpath of this library.
testRuntimeClasspath	For executing tests of this library	no	yes	This configuration contains the test runtime classpath of this library

Building Modules for the Java Module System

Since Java 9, Java itself offers a [module system](#) that allows for strict encapsulation during compile and runtime. You can turn a Java library into a *Java Module* by creating a `module-info.java` file in the `main/java` source folder.

```
src
├── main
│   └── java
│       └── module-info.java
```

In the module info file, you declare a *module name*, which packages of your module you want to *export* and which other modules you *require*.

module-info.java file

```
module org.gradle.sample {
    requires com.google.gson;           // real module
    requires org.apache.commons.lang3; // automatic module
    // commons-cli-1.4.jar is not a module and cannot be required
}
```

To tell the Java compiler that a Jar is a module, as opposed to a traditional Java library, Gradle needs to place it on the so called *module path*. It is an alternative to the *classpath*, which is the traditional way to tell the compiler about compiled dependencies. Gradle will automatically put a Jar of your dependencies on the module path, instead of the classpath, if these three things are true:

- `java.modularity.inferModulePath` is **not** turned off
- We are actually building a module (as opposed to a traditional library) which we expressed by adding the `module-info.java` file. (Another option is to add the `Automatic-Module-Name` Jar manifest attribute as [described further down](#).)
- The Jar our module depends on is itself a module, which Gradle decides based on the presence of a `module-info.class`—the compiled version of the module descriptor—in the Jar. (Or,

alternatively, the presence of an `Automatic-Module-Name` attribute the Jar manifest)

In the following, some more details about defining Java modules and how that interacts with Gradle's dependency management are described. You can also look at a [ready made example](#) to try out the Java Module support directly.

Declaring module dependencies

There is a direct relationship to the dependencies you declare in the build file and the module dependencies you declare in the `module-info.java` file. Ideally the declarations should be in sync as seen in the following table.

Table 12. Mapping between Java module directives and Gradle configurations to declare dependencies

Java Module Directive	Gradle Configuration	Purpose
<code>requires</code>	<code>implementation</code>	Declaring implementation dependencies
<code>requires transitive</code>	<code>api</code>	Declaring API dependencies
<code>requires static</code>	<code>compileOnly</code>	Declaring compile only dependencies
<code>requires static transitive</code>	<code>compileOnlyApi</code>	Declaring compile only API dependencies

Gradle currently does not automatically check if the dependency declarations are in sync. This may be added in future versions.

For more details on declaring module dependencies, please refer to [documentation on the Java Module System](#).

Declaring package visibility and services

The Java module system supports additional more fine granular encapsulation concepts than Gradle itself currently does. For example, you explicitly need to declare which packages are part of your API and which are only visible inside your module. Some of these capabilities might be added to Gradle itself in future versions. For now, please refer to [documentation on the Java Module System](#) to learn how to use these features in Java Modules.

Declaring module versions

Java Modules also have a version that is encoded as part of the module identity in the `module-info.class` file. This version can be inspected when a module is running.

Example 115. Declare the module version in the build script or directly as compile task option

build.gradle.kts

```
version = "1.2"

tasks.compileJava {
    // use the project's version or define one directly
    options.javaModuleVersion = provider { version as String }
```

```
}
```

build.gradle

```
version = '1.2'

tasks.named('compileJava') {
    // use the project's version or define one directly
    options.javaModuleVersion = provider { version }
}
```

Using libraries that are not modules

You probably want to use external libraries, like OSS libraries from Maven Central, in your modular Java project. Some libraries, in their newer versions, are already full modules with a module descriptor. For example, `com.google.code.gson:gson:2.8.9` that has the module name `com.google.gson`.

Others, like `org.apache.commons:commons-lang3:3.10`, may not offer a full module descriptor but will at least contain an `Automatic-Module-Name` entry in their manifest file to define the module's name (`org.apache.commons.lang3` in the example). Such modules, that only have a name as module description, are called *automatic module* that export **all** their packages and can read **all** modules on the module path.

A third case are traditional libraries that provide no module information at all—for example `commons-cli:commons-cli:1.4`. Gradle puts such libraries on the classpath instead of the module path. The classpath is then treated as one module (the so called *unnamed* module) by Java.

Example 116. Dependencies to modules and libraries declared in build file

build.gradle.kts

```
dependencies {
    implementation("com.google.code.gson:gson:2.8.9")           // real module
    implementation("org.apache.commons:commons-lang3:3.10")    // automatic
    module
    implementation("commons-cli:commons-cli:1.4")               // plain library
}
```

build.gradle

```
dependencies {
    implementation 'com.google.code.gson:gson:2.8.9'           // real module
}
```

```

        implementation 'org.apache.commons:commons-lang3:3.10' // automatic
    module
        implementation 'commons-cli:commons-cli:1.4'           // plain library
    }

```

Module dependencies declared in module-info.java file

```

module org.gradle.sample.lib {
    requires com.google.gson;           // real module
    requires org.apache.commons.lang3; // automatic module
    // commons-cli-1.4.jar is not a module and cannot be required
}

```

While a real module cannot directly depend on the unnamed module (only by adding command line flags), automatic modules can also see the unnamed module. Thus, if you cannot avoid to rely on a library without module information, you can wrap that library in an automatic module as part of your project. How you do that is described in the next section.

Another way to deal with non-modules is to enrich existing Jars with module descriptors yourself using [artifact transforms](#). [This sample](#) contains a small *buildSrc* plugin registering such a transform which you may use and adjust to your needs. This can be interesting if you want to build a fully [modular application](#) and want the java runtime to treat everything as a real module.

Disabling Java Module support

In rare cases, you might want to disable the built-in Java Module support and define the module path by other means. To achieve this, you can disable the functionality to automatically put any Jar on the module path. Then Gradle puts Jars with module information on the classpath, even if you have a `module-info.java` in your source set. This corresponds to the behaviour of Gradle versions <7.0.

To make this work, you need to set `modularity.inferModulePath = false` on the Java extension (for all tasks) or on individual tasks.

Example 117. [Disable Gradle's module path inference](#)

build.gradle.kts

```

java {
    modularity.inferModulePath = false
}

tasks.compileJava {
    modularity.inferModulePath = false
}

```

build.gradle

```
java {  
    modularity.inferModulePath = false  
}  
  
tasks.named('compileJava') {  
    modularity.inferModulePath = false  
}
```

Building an automatic module

If you can, you should always write complete `module-info.java` descriptors for your modules. Still, there are a few cases where you might consider to (initially) only provide a *module name* for an automatic module:

- You are working on a library that is **not** a module but you want to make it usable as such in the next release. Adding an `Automatic-Module-Name` is a good first step (most popular OSS libraries on Maven central have done it by now).
- As discussed in the previous section, an automatic module can be used as an adapter between your real modules and a traditional library on the classpath.

To turn a normal Java project into an *automatic module*, just add the manifest entry with the module name:

Example 118. Declare an automatic module name as Jar manifest attribute

build.gradle.kts

```
tasks.jar {  
    manifest {  
        attributes("Automatic-Module-Name" to "org.gradle.sample")  
    }  
}
```

build.gradle

```
tasks.named('jar') {  
    manifest {  
        attributes('Automatic-Module-Name': 'org.gradle.sample')  
    }  
}
```

NOTE

=== You can define an automatic module as part of a multi-project that otherwise defines real modules (e.g. as an adapter to another library). While this works fine in the Gradle build, such automatic module projects are not correctly recognized by IDEA/Eclipse at the moment. You can work around it by manually adding the Jar built for the automatic module to the dependencies of the project that does not find it in the IDE's UI. ===

Using classes instead of jar for compilation

A feature of the `java-library` plugin is that projects which consume the library only require the `classes` folder for compilation, instead of the full JAR. This enables lighter inter-project dependencies as resources processing (`processResources` task) and archive construction (`jar` task) are no longer executed when only Java code compilation is performed during development.

NOTE

The usage or not of the classes output instead of the JAR is a *consumer* decision. For example, Groovy consumers will request classes *and* processed resources as these may be needed for executing AST transformation as part of the compilation process.

Increased memory usage for consumers

An indirect consequence is that up-to-date checking will require more memory, because Gradle will snapshot individual class files instead of a single jar. This may lead to increased memory consumption for large projects, with the benefit of having the `compileJava` task up-to-date in more cases (e.g. changing resources no longer changes the input for `compileJava` tasks of upstream projects)

Significant build performance drop on Windows for huge multi-projects

Another side effect of the snapshotting of individual class files, only affecting Windows systems, is that the performance can significantly drop when processing a very large amount of class files on the compile classpath. This only concerns very large multi-projects where a lot of classes are present on the classpath by using many `api` dependencies. To mitigate this, you can set the `org.gradle.java.compile-classpath-packaging` system property to `true` to change the behavior of the Java Library plugin to use jars instead of class folders for everything on the compile classpath. Note, since this has other performance impacts and potentially side effects, by triggering all jar tasks at compile time, it is only recommended to activate this if you suffer from the described performance issue on Windows.

Distributing a library

Aside from [publishing](#) a library to a component repository, you may sometimes need to package a library and its dependencies in a distribution deliverable. The [Java Library Distribution Plugin](#) is there to help you do just that.

The Application Plugin

The Application plugin facilitates creating an executable JVM application. It makes it easy to start

the application locally during development, and to package the application as a TAR and/or ZIP including operating system specific start scripts.

Applying the Application plugin also implicitly applies the [Java plugin](#). The `main` source set is effectively the "application".

Applying the Application plugin also implicitly applies the [Distribution plugin](#). A `main` distribution is created that packages up the application, including code dependencies and generated start scripts.

Building JVM applications

To use the application plugin, include the following in your build script:

Example 119. Using the application plugin

build.gradle.kts

```
plugins {  
    application  
}
```

build.gradle

```
plugins {  
    id 'application'  
}
```

The only mandatory configuration for the plugin is the specification of the main class (i.e. entry point) of the application.

Example 120. Configure the application main class

build.gradle.kts

```
application {  
    mainClass = "org.gradle.sample.Main"  
}
```

build.gradle

```
application {  
    mainClass = 'org.gradle.sample.Main'
```

```
}
```

You can run the application by executing the `run` task (type: `JavaExec`). This will compile the main source set, and launch a new JVM with its classes (along with all runtime dependencies) as the classpath and using the specified main class. You can launch the application in debug mode with `gradle run --debug-jvm` (see `JavaExec.setDebug(boolean)`).

Since Gradle 4.9, the command line arguments can be passed with `--args`. For example, if you want to launch the application with command line arguments `foo --bar`, you can use `gradle run --args="foo --bar"` (see `JavaExec.setArgsString(java.lang.String)`).

If your application requires a specific set of JVM settings or system properties, you can configure the `applicationDefaultJvmArgs` property. These JVM arguments are applied to the `run` task and also considered in the generated start scripts of your distribution.

Example 121. Configure default JVM settings

build.gradle.kts

```
application {  
    applicationDefaultJvmArgs = listOf("-Dgreeting.language=en")  
}
```

build.gradle

```
application {  
    applicationDefaultJvmArgs = ['-Dgreeting.language=en']  
}
```

If your application's start scripts should be in a different directory than `bin`, you can configure the `executableDir` property.

Example 122. Configure custom directory for start scripts

build.gradle.kts

```
application {  
    executableDir = "custom_bin_dir"  
}
```

build.gradle

```
application {  
    executableDir = 'custom_bin_dir'  
}
```

Building applications using the Java Module System

Gradle supports the building of [Java Modules](#) as described in the [corresponding section of the Java Library plugin documentation](#). Java modules can also be runnable and you can use the application plugin to run and package such a modular application. For this, you need to do two things in addition to what you do for a non-modular application.

First, you need to add a `module-info.java` file to describe your application module. Please refer to the [Java Library plugin documentation](#) for more details on this topic.

Second, you need to tell Gradle the name of the module you want to run in addition to the main class name like this:

Example 123. Configure the modular application's main module

build.gradle.kts

```
application {  
    mainModule = "org.gradle.sample.app" // name defined in module-info.java  
    mainClass = "org.gradle.sample.Main"  
}
```

build.gradle

```
application {  
    mainModule = 'org.gradle.sample.app' // name defined in module-info.java  
    mainClass = 'org.gradle.sample.Main'  
}
```

That's all. If you run your application, by executing the `run` task or through a [generated start script](#), it will run as module and respect module boundaries at runtime. For example, reflective access to an internal package from another module can fail.

The configured *main class* is also baked into the `module-info.class` file of your application Jar. If you run the modular application directly using the `java` command, it is then sufficient to provide the module name.

You can also look at a [ready made example](#) that includes a modular application as part of a multi-project.

Building a distribution

A distribution of the application can be created, by way of the [Distribution plugin](#) (which is automatically applied). A `main` distribution is created with the following content:

Table 13. Distribution content

Location	Content
(root dir)	<code>src/dist</code>
<code>lib</code>	All runtime dependencies and main source set class files.
<code>bin</code>	Start scripts (generated by <code>startScripts</code> task).

Static files to be added to the distribution can be simply added to `src/dist`. More advanced customization can be done by configuring the [CopySpec](#) exposed by the main distribution.

Example 124. [Include output from other tasks in the application distribution](#)

build.gradle.kts

```
val createDocs by tasks.registering {
    val docs = layout.buildDirectory.dir("docs")
    outputs.dir(docs)
    doLast {
        docs.get().asFile.mkdirs()
        docs.get().file("readme.txt").asFile.writeText("Read me!")
    }
}

distributions {
    main {
        contents {
            from(createDocs) {
                into("docs")
            }
        }
    }
}
```

build.gradle

```
tasks.register('createDocs') {
    def docs = layout.buildDirectory.dir('docs')
    outputs.dir docs
    doLast {
```

```

        docs.get().asFile.mkdirs()
        docs.get().file('readme.txt').asFile.write('Read me!')
    }
}

distributions {
    main {
        contents {
            from(createDocs) {
                into 'docs'
            }
        }
    }
}
}

```

By specifying that the distribution should include the task's output files (see [incremental builds](#)), Gradle knows that the task that produces the files must be invoked before the distribution can be assembled and will take care of this for you.

You can run `gradle installDist` to create an image of the application in `build/install/projectName`. You can run `gradle distZip` to create a ZIP containing the distribution, `gradle distTar` to create an application TAR or `gradle assemble` to build both.

Customizing start script generation

The application plugin can generate Unix (suitable for Linux, macOS etc.) and Windows start scripts out of the box. The start scripts launch a JVM with the specified settings defined as part of the original build and runtime environment (e.g. `JAVA_OPTS` env var). The default script templates are based on the same scripts used to launch Gradle itself, that ship as part of a Gradle distribution.

The start scripts are completely customizable. Please refer to the documentation of [CreateStartScripts](#) for more details and customization examples.

Tasks

The Application plugin adds the following tasks to the project.

`run` — [JavaExec](#)

Depends on: `classes`

Starts the application.

`startScripts` — [CreateStartScripts](#)

Depends on: `jar`

Creates OS specific scripts to run the project as a JVM application.

installDist — **Sync**

Depends on: `jar`, `startScripts`

Installs the application into a specified directory.

distZip — **Zip**

Depends on: `jar`, `startScripts`

Creates a full distribution ZIP archive including runtime libraries and OS specific scripts.

distTar — **Tar**

Depends on: `jar`, `startScripts`

Creates a full distribution TAR archive including runtime libraries and OS specific scripts.

Application extension

The Application Plugin adds an extension to the project, which you can use to configure its behavior. See the [JavaApplication](#) DSL documentation for more information on the properties available on the extension.

You can configure the extension via the `application {}` block shown earlier, for example using the following in your build script:

build.gradle.kts

```
application {  
    executableDir = "custom_bin_dir"  
}
```

build.gradle

```
application {  
    executableDir = 'custom_bin_dir'  
}
```

License of start scripts

The start scripts generated for the application are licensed under the [Apache 2.0 Software License](#).

Convention properties (deprecated)

This plugin also adds some convention properties to the project, which you can use to configure its behavior. These are **deprecated** and superseded by the extension described above. See the [Project](#)

DSL documentation for information on them.

Unlike the extension properties, these properties appear as top-level project properties in the build script. For example, to change the application name you can just add the following to your build script:

build.gradle.kts

```
application.applicationName = "my-app"
```

build.gradle

```
application.applicationName = 'my-app'
```

The Java Platform Plugin

The Java Platform plugin brings the ability to declare platforms for the Java ecosystem. A platform can be used for different purposes:

- a description of modules which are published together (and for example, share the same version)
- a set of recommended versions for heterogeneous libraries. A typical example includes the [Spring Boot BOM](#)
- [sharing a set of dependency versions](#) between subprojects

A platform is a special kind of software component which doesn't contain any sources: it is only used to reference other libraries, so that they play well together during dependency resolution.

Platforms can be published as [Gradle Module Metadata](#) and [Maven BOMs](#).

NOTE

The `java-platform` plugin cannot be used in combination with the `java` or `java-library` plugins in a given project. Conceptually a project is either a platform, with no binaries, *or* produces binaries.

Usage

To use the Java Platform plugin, include the following in your build script:

Example 125. *Using the Java Platform plugin*

build.gradle.kts

```
plugins {  
    `java-platform`  
}
```

build.gradle

```
plugins {  
    id 'java-platform'  
}
```

API and runtime separation

A major difference between a Maven BOM and a Java platform is that in Gradle dependencies and [constraints](#) are declared and scoped to a configuration and the ones extending it. While many users will only care about declaring constraints for *compile time* dependencies, thus inherited by runtime and tests ones, it allows declaring dependencies or constraints that only apply to runtime or test.

For this purpose, the plugin exposes two [configurations](#) that can be used to declare dependencies: [api](#) and [runtime](#). The [api](#) configuration should be used to declare constraints and dependencies which should be used when compiling against the platform, whereas the [runtime](#) configuration should be used to declare constraints or dependencies which are visible at runtime.

Example 126. *Declaring API and runtime constraints*

build.gradle.kts

```
dependencies {  
    constraints {  
        api("commons-httpclient:commons-httpclient:3.1")  
        runtime("org.postgresql:postgresql:42.2.5")  
    }  
}
```

build.gradle

```
dependencies {  
    constraints {  
        api 'commons-httpclient:commons-httpclient:3.1'
```

```
        runtime 'org.postgresql:postgresql:42.2.5'
    }
}
```

Note that this example makes use of *constraints* and not dependencies. In general, this is what you would like to do: constraints will only apply if such a component is added to the dependency graph, either directly or transitively. This means that all constraints listed in a platform would not add a dependency unless another component brings it in: they can be seen as *recommendations*.

NOTE

For example, if a platform declares a constraint on `org:foo:1.1`, and that nothing else brings in a dependency on `foo`, `foo` will *not* appear in the graph. However, if `foo` appears, then usual conflict resolution would kick in. If a dependency brings in `org:foo:1.0`, then we would select `org:foo:1.1` to satisfy the platform constraint.

By default, in order to avoid the common mistake of adding a dependency in a platform instead of a constraint, Gradle will fail if you try to do so. If, for some reason, you also want to add *dependencies* in addition to constraints, you need to enable it explicitly:

Example 127. Allowing declaration of dependencies

build.gradle.kts

```
javaPlatform {
    allowDependencies()
}
```

build.gradle

```
javaPlatform {
    allowDependencies()
}
```

Local project constraints

If you have a multi-project build and want to publish a platform that links to subprojects, you can do it by declaring constraints on the subprojects which belong to the platform, as in the example below:

Example 128. Declaring constraints on subprojects

build.gradle.kts

```
dependencies {  
    constraints {  
        api(project(":core"))  
        api(project(":lib"))  
    }  
}
```

build.gradle

```
dependencies {  
    constraints {  
        api project(":core")  
        api project(":lib")  
    }  
}
```

The project notation will become a classical `group:name:version` notation in the published metadata.

Sourcing constraints from another platform

Sometimes the platform you define is an extension of another existing platform.

In order to have your platform include the constraints from that third party platform, it needs to be imported as a `platform` dependency:

Example 129. Importing a platform

build.gradle.kts

```
javaPlatform {  
    allowDependencies()  
}  
  
dependencies {  
    api(platform("com.fasterxml.jackson:jackson-bom:2.9.8"))  
}
```

build.gradle

```
javaPlatform {
    allowDependencies()
}

dependencies {
    api platform('com.fasterxml.jackson:jackson-bom:2.9.8')
}
```

Publishing platforms

Publishing Java platforms is done by applying the `maven-publish` plugin and configuring a Maven publication that uses the `javaPlatform` component:

Example 130. Publishing as a BOM

build.gradle.kts

```
publishing {
    publications {
        create<MavenPublication>("myPlatform") {
            from(components["javaPlatform"])
        }
    }
}
```

build.gradle

```
publishing {
    publications {
        myPlatform(MavenPublication) {
            from components.javaPlatform
        }
    }
}
```

This will generate a BOM file for the platform, with a `<dependencyManagement>` block where its `<dependencies>` correspond to the constraints defined in the platform module.

Consuming platforms

Because a Java Platform is a special kind of component, a dependency on a Java platform has to be declared using the `platform` or `enforcedPlatform` keyword, as explained in the [managing transitive dependencies](#) section. For example, if you want to share dependency versions between subprojects, you can define a platform module which would declare all versions:

Example 131. [Recommend versions in a platform module](#)

build.gradle.kts

```
dependencies {
    constraints {
        // Platform declares some versions of libraries used in subprojects
        api("commons-httpclient:commons-httpclient:3.1")
        api("org.apache.commons:commons-lang3:3.8.1")
    }
}
```

build.gradle

```
dependencies {
    constraints {
        // Platform declares some versions of libraries used in subprojects
        api 'commons-httpclient:commons-httpclient:3.1'
        api 'org.apache.commons:commons-lang3:3.8.1'
    }
}
```

And then have subprojects depend on the platform to get recommendations:

Example 132. [Get recommendations from a platform](#)

build.gradle.kts

```
dependencies {
    // get recommended versions from the platform project
    api(platform(project(":platform")))
    // no version required
    api("commons-httpclient:commons-httpclient")
}
```

build.gradle

```
dependencies {  
    // get recommended versions from the platform project  
    api platform(project(':platform'))  
    // no version required  
    api 'commons-httpclient:commons-httpclient'  
}
```

The Groovy Plugin

The Groovy plugin extends the [Java plugin](#) to add support for [Groovy](#) projects. It can deal with Groovy code, mixed Groovy and Java code, and even pure Java code (although we don't necessarily recommend to use it for the latter). The plugin supports *joint compilation*, which allows you to freely mix and match Groovy and Java code, with dependencies in both directions. For example, a Groovy class can extend a Java class that in turn extends a Groovy class. This makes it possible to use the best language for the job, and to rewrite any class in the other language if needed.

Note that if you want to benefit from the [API / implementation separation](#), you can also apply the [java-library](#) plugin to your Groovy project.

Usage

To use the Groovy plugin, include the following in your build script:

Example 133. [Using the Groovy plugin](#)

build.gradle.kts

```
plugins {  
    groovy  
}
```

build.gradle

```
plugins {  
    id 'groovy'  
}
```

Tasks

The Groovy plugin adds the following tasks to the project. Information about altering the dependencies to Java compile tasks are found [here](#).

`compileGroovy` — `GroovyCompile`

Depends on: `compileJava`

Compiles production Groovy source files.

`compileTestGroovy` — `GroovyCompile`

Depends on: `compileTestJava`

Compiles test Groovy source files.

`compileSourceSetGroovy` — `GroovyCompile`

Depends on: `compileSourceSetJava`

Compiles the given source set's Groovy source files.

`groovydoc` — `Groovydoc`

Generates API documentation for the production Groovy source files.

The Groovy plugin adds the following dependencies to tasks added by the Java plugin.

Table 14. Groovy plugin - additional task dependencies

Task name	Depends on
<code>classes</code>	<code>compileGroovy</code>
<code>testClasses</code>	<code>compileTestGroovy</code>
<code>sourceSetClasses</code>	<code>compileSourceSetGroovy</code>

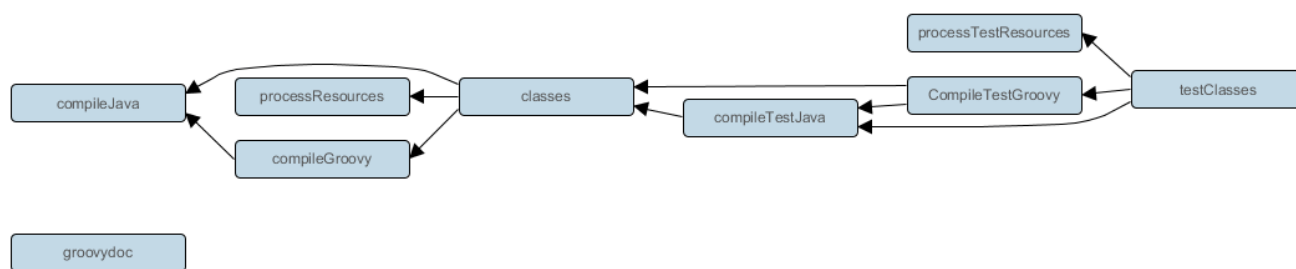


Figure 20. Groovy plugin - tasks

Project layout

The Groovy plugin assumes the project layout shown in [Groovy Layout](#). All the Groovy source directories can contain Groovy *and* Java code. The Java source directories may only contain Java source code.^[1] None of these directories need to exist or have anything in them; the Groovy plugin will simply compile whatever it finds.

src/main/java

Production Java source.

src/main/resources

Production resources, such as XML and properties files.

src/main/groovy

Production Groovy source. May also contain Java source files for joint compilation.

src/test/java

Test Java source.

src/test/resources

Test resources.

src/test/groovy

Test Groovy source. May also contain Java source files for joint compilation.

src/sourceSet/java

Java source for the source set named *sourceSet*.

src/sourceSet/resources

Resources for the source set named *sourceSet*.

src/sourceSet/groovy

Groovy source files for the given source set. May also contain Java source files for joint compilation.

Changing the project layout

Just like the Java plugin, the Groovy plugin allows you to configure custom locations for Groovy production and test source files.

Example 134. Custom Groovy source layout

build.gradle.kts

```
sourceSets {  
    main {  
        groovy {  
            setSrcDirs(listOf("src/groovy"))  
        }  
    }  
  
    test {  
        groovy {  
            setSrcDirs(listOf("test/groovy"))  
        }  
    }  
}
```

```
}
```

build.gradle

```
sourceSets {
    main {
        groovy {
            srcDirs = ['src/groovy']
        }
    }

    test {
        groovy {
            srcDirs = ['test/groovy']
        }
    }
}
```

Dependency management

Because Gradle's build language is based on Groovy, Gradle already ships with a Groovy library. Nevertheless, Groovy projects need to explicitly declare a Groovy dependency. This dependency will then be used on compile and runtime class paths. It will also be used to get hold of the Groovy compiler and Groovydoc tool, respectively.

If Groovy is used for production code, the Groovy dependency should be added to the **implementation** configuration:

Example 135. Configuration of Groovy dependency

build.gradle.kts

```
repositories {
    mavenCentral()
}

dependencies {
    implementation("org.codehaus.groovy:groovy-all:2.4.15")
}
```

build.gradle

```
repositories {
```

```

    mavenCentral()
}

dependencies {
    implementation 'org.codehaus.groovy:groovy-all:2.4.15'
}

```

If Groovy is only used for test code, the Groovy dependency should be added to the `testImplementation` configuration:

Example 136. Configuration of Groovy test dependency

build.gradle.kts

```

dependencies {
    testImplementation("org.codehaus.groovy:groovy-all:2.4.15")
}

```

build.gradle

```

dependencies {
    testImplementation 'org.codehaus.groovy:groovy-all:2.4.15'
}

```

To use the Groovy library that ships with Gradle, declare a `localGroovy()` dependency. Note that different Gradle versions ship with different Groovy versions; as such, using `localGroovy()` is less safe than declaring a regular Groovy dependency.

Example 137. Configuration of bundled Groovy dependency

build.gradle.kts

```

dependencies {
    implementation(localGroovy())
}

```

build.gradle

```

dependencies {
    implementation localGroovy()
}

```

```
}
```

Automatic configuration of groovyClasspath

The `GroovyCompile` and `Groovydoc` tasks consume Groovy code in two ways: on their `classpath`, and on their `groovyClasspath`. The former is used to locate classes referenced by the source code, and will typically contain the Groovy library along with other libraries. The latter is used to load and execute the Groovy compiler and Groovydoc tool, respectively, and should only contain the Groovy library and its dependencies.

Unless a task's `groovyClasspath` is configured explicitly, the Groovy (base) plugin will try to infer it from the task's `classpath`. This is done as follows:

- If a `groovy-all(-indy)` Jar is found on `classpath`, that jar will be added to `groovyClasspath`.
- If a `groovy(-indy)` jar is found on `classpath`, and the project has at least one repository declared, a corresponding `groovy(-indy)` repository dependency will be added to `groovyClasspath`.
- Otherwise, execution of the task will fail with a message saying that `groovyClasspath` could not be inferred.

Note that the “-indy” variation of each jar refers to the version with `invokedynamic` support.

Convention properties

The Groovy plugin does not add any convention properties to the project.

Source set properties

The Groovy plugin adds the following extensions to each source set in the project. You can use these properties in your build script as though they were properties of the source set object.

Groovy Plugin — source set properties

`groovy` — `GroovySourceDirectorySet` (read-only)

Default value: Not null

The Groovy source files of this source set. Contains all `.groovy` and `.java` files found in the Groovy source directories, and excludes all other types of files.

`groovy.srcDirs` — `Set<File>`

Default value: `[projectDir/src/name/groovy]`

The source directories containing the Groovy source files of this source set. May also contain Java source files for joint compilation. Can set using anything described in [Specifying Multiple Files](#).

allGroovy — FileTree (read-only)

Default value: Not null

All Groovy source files of this source set. Contains only the `.groovy` files found in the Groovy source directories.

These properties are provided by a convention object of type `GroovySourceSet`.

The Groovy plugin also modifies some source set properties:

Groovy Plugin - modified source set properties

Property name	Change
<code>allJava</code>	Adds all <code>.java</code> files found in the Groovy source directories.
<code>allSource</code>	Adds all source files found in the Groovy source directories.

GroovyCompile

The Groovy plugin adds a `GroovyCompile` task for each source set in the project. The task type shares much with the `JavaCompile` task by extending `AbstractCompile` (see [the relevant Java Plugin section](#)). The `GroovyCompile` task supports most configuration options of the official Groovy compiler. The task can also leverage the [Java toolchain support](#).

Table 15. Groovy plugin - GroovyCompile properties

Task Property	Type	Default Value
<code>classpath</code>	<code>FileCollection</code>	<code>sourceSet.compileClasspath</code>
<code>source</code>	<code>FileTree</code> . Can set using anything described in Specifying Multiple Files .	<code>sourceSet.groovy</code>
<code>destinationDirectory</code>	<code>File</code> .	<code>sourceSet.groovy.destinationDirectory</code>
<code>groovyClasspath</code>	<code>FileCollection</code>	<code>groovy</code> configuration if non-empty; Groovy library found on <code>classpath</code> otherwise
<code>javaLauncher</code>	<code>Property<JavaLauncher></code> , see the toolchain documentation .	None but will be configured if a toolchain is defined on the <code>java</code> extension.

Compilation avoidance

Caveat: Groovy compilation avoidance is an incubating feature since Gradle 5.6. There are known inaccuracies so please enable it at your own risk.

To enable the incubating support for Groovy compilation avoidance, add a `enableFeaturePreview` to your settings file:

settings.gradle

```
enableFeaturePreview('GROOVY_COMPILATION_AVOIDANCE')
```

settings.gradle.kts

```
enableFeaturePreview("GROOVY_COMPILATION_AVOIDANCE")
```

If a dependent project has changed in an [ABI-compatible](#) way (only its private API has changed), then Groovy compilation tasks will be up-to-date. This means that if project **A** depends on project **B** and a class in **B** is changed in an ABI-compatible way (typically, changing only the body of a method), then Gradle won't recompile **A**.

See [Java compile avoidance](#) for a detailed list of the types of changes that do not affect the ABI and are ignored.

However, similar to Java's annotation processing, there are various ways to [customize the Groovy compilation process](#), for which implementation details matter. Some well-known examples are [Groovy AST transformations](#). In these cases, these dependencies must be declared separately in a classpath called `astTransformationClasspath`:

Example 138. Declaring AST transformations

build.gradle.kts

```
val astTransformation by configurations.creating
dependencies {
    astTransformation(project(":ast-transformation"))
}
tasks.withType<GroovyCompile>().configureEach {
    astTransformationClasspath.from(astTransformation)
}
```

build.gradle

```
configurations { astTransformation }
dependencies {
    astTransformation(project(":ast-transformation"))
}
tasks.withType(GroovyCompile).configureEach {
    astTransformationClasspath.from(configurations.astTransformation)
}
```

```
}
```

Incremental Groovy compilation

Since 5.6, Gradle introduces an experimental incremental Groovy compiler. To enable incremental compilation for Groovy, you need:

- Enable [Groovy compilation avoidance](#).
- Explicitly enable incremental Groovy compilation in the build script:

Example 139. [Enable incremental Groovy compilation](#)

buildSrc/src/main/kotlin/myproject.groovy-conventions.gradle.kts

```
tasks.withType<GroovyCompile>().configureEach {  
    options.isIncremental = true  
    options.incrementalAfterFailure = true  
}
```

buildSrc/src/main/groovy/myproject.groovy-conventions.gradle

```
tasks.withType(GroovyCompile).configureEach {  
    options.incremental = true  
    options.incrementalAfterFailure = true  
}
```

This gives you the following benefits:

- Incremental builds are much faster.
- If only a small set of Groovy source files are changed, only the affected source files will be recompiled. Classes that don't need to be recompiled remain unchanged in the output directory. For example, if you only change a few Groovy test classes, you don't need to recompile all Groovy test source files — only the changed ones need to be recompiled.

To understand how incremental compilation works, see [Incremental Java compilation](#) for a detailed overview. Note that there're several differences from Java incremental compilation:

The Groovy compiler doesn't keep `@Retention` in generated annotation class bytecode ([GROOVY-9185](#)), thus all annotations are `RUNTIME`. This means that changes to source-retention annotations won't trigger a full recompilation.

Known issues

Also see [Known issues for incremental Java compilation](#).

- Changes to resources won't trigger a recompilation, this might result in some incorrectness — for example [Extension Modules](#).

Compiling and testing for Java 6 or Java 7

With [toolchain support](#) added to [GroovyCompile](#), it is possible to compile Groovy code using a different Java version than the one running Gradle. If you also have Java source files, this will also configure [JavaCompile](#) to use the right Java compiler is used, as can be seen in the [Java plugin](#) documentation.

Example: Configure Java 7 build for Groovy

build.gradle.kts

```
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(7)
    }
}
```

build.gradle

```
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(7)
    }
}
```

The Scala Plugin

The Scala plugin extends the [Java plugin](#) to add support for [Scala](#) projects. The plugin also supports *joint compilation*, which allows you to freely mix and match Scala and Java code with dependencies in both directions. For example, a Scala class can extend a Java class that in turn extends a Scala class. This makes it possible to use the best language for the job, and to rewrite any class in the other language if needed.

Note that if you want to benefit from the [API / implementation separation](#), you can also apply the [java-library](#) plugin to your Scala project.

Usage

To use the Scala plugin, include the following in your build script:

Example 140. Using the Scala plugin

build.gradle.kts

```
plugins {  
    id("scala")  
}
```

build.gradle

```
plugins {  
    id("scala")  
}
```

Tasks

The Scala plugin adds the following tasks to the project. Information about altering the dependencies to Java compile tasks are found [here](#).

compileScala — **ScalaCompile**

Depends on: **compileJava**

Compiles production Scala source files.

compileTestScala — **ScalaCompile**

Depends on: **compileTestJava**

Compiles test Scala source files.

compileSourceSetScala — **ScalaCompile**

Depends on: **compileSourceSetJava**

Compiles the given source set's Scala source files.

scaladoc — **ScalaDoc**

Generates API documentation for the production Scala source files.

The **ScalaCompile** and **ScalaDoc** tasks support [Java toolchains](#) out of the box.

The Scala plugin adds the following dependencies to tasks added by the Java plugin.

Table 16. Scala plugin - additional task dependencies

Task name	Depends on
<code>classes</code>	<code>compileScala</code>
<code>testClasses</code>	<code>compileTestScala</code>
<code>sourceSetClasses</code>	<code>compileSourceSetScala</code>

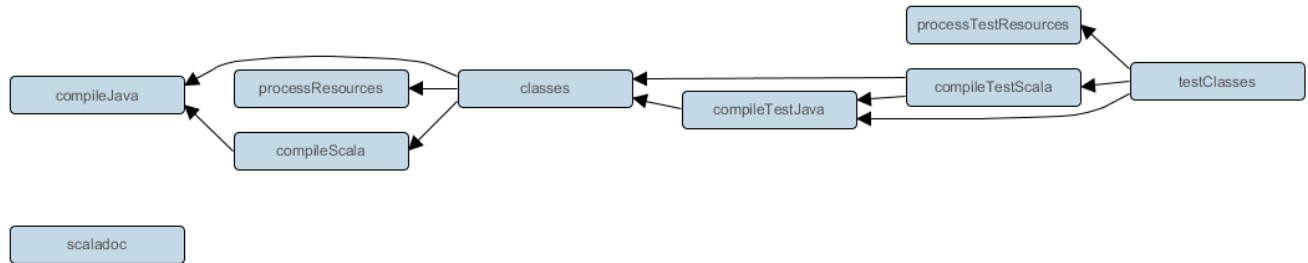


Figure 21. Scala plugin - tasks

Project layout

The Scala plugin assumes the project layout shown below. All the Scala source directories can contain Scala *and* Java code. The Java source directories may only contain Java source code. None of these directories need to exist or have anything in them; the Scala plugin will simply compile whatever it finds.

`src/main/java`

Production Java source.

`src/main/resources`

Production resources, such as XML and properties files.

`src/main/scala`

Production Scala source. May also contain Java source files for joint compilation.

`src/test/java`

Test Java source.

`src/test/resources`

Test resources.

`src/test/scala`

Test Scala source. May also contain Java source files for joint compilation.

`src/sourceSet/java`

Java source for the source set named *sourceSet*.

`src/sourceSet/resources`

Resources for the source set named *sourceSet*.

src/sourceSet/scala

Scala source files for the given source set. May also contain Java source files for joint compilation.

Changing the project layout

Just like the Java plugin, the Scala plugin allows you to configure custom locations for Scala production and test source files.

Example 141. Custom Scala source layout

build.gradle.kts

```
sourceSets {
    main {
        scala {
            setSrcDirs(listOf("src/scala"))
        }
    }
    test {
        scala {
            setSrcDirs(listOf("test/scala"))
        }
    }
}
```

build.gradle

```
sourceSets {
    main {
        scala {
            srcDirs = ['src/scala']
        }
    }
    test {
        scala {
            srcDirs = ['test/scala']
        }
    }
}
```

Scala version

The Scala version can be declared directly on the `scala` extension. Dependencies are automatically added to the `scalaToolchain` configuration based on the declared Scala version. The

`scalaToolchainRuntimeClasspath` configuration resolves the dependencies declared on the `scalaToolchain` configuration and contains files required to run the Scala compiler.

NOTE

Direct dependencies on the Scala SDK do not need to be declared when the `scalaVersion` property is set.

Example 142. Declaring a Scala 2 version

build.gradle.kts

```
repositories {
    mavenCentral()
}

scala {
    scalaVersion = "2.13.12"
}
```

build.gradle

```
repositories {
    mavenCentral()
}

scala {
    scalaVersion = "2.13.12"
}
```

Configuring Gradle to use Scala 3 is no different from Scala 2.

Example 143. Declaring a Scala 3 version

build.gradle.kts

```
repositories {
    mavenCentral()
}

scala {
    scalaVersion = "3.6.3"
}
```

build.gradle

```
repositories {  
    mavenCentral()  
}  
  
scala {  
    scalaVersion = "3.6.3"  
}
```

NOTE The `scalaVersion` property is incubating.

Declaring Scala dependencies

When not using the `scalaVersion` property, the Scala SDK dependency must be declared manually on the `implementation` configuration. This pattern is not preferred, as it relies on [inferring the Scala classpath](#) from the production runtime classpath. Omitting the Scala version from the `scala` extension will be deprecated in a future Gradle version.

Scala 2 projects need to declare a `scala-library` dependency.

Example 144. Declaring a Scala dependency for production code

build.gradle.kts

```
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    implementation("org.scala-lang:scala-library:2.13.12")  
}
```

build.gradle

```
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    implementation("org.scala-lang:scala-library:2.13.12")  
}
```

Scala 3 projects need to declare a `scala3-library_3` dependency instead:

Example 145. Declaring a Scala 3 dependency for production code

build.gradle.kts

```
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    implementation("org.scala-lang:scala3-library_3:3.6.3")  
}
```

build.gradle

```
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    implementation("org.scala-lang:scala3-library_3:3.6.3")  
}
```

If Scala is only used for test code, the `scala-library` dependency should be added to the `testImplementation` configuration:

Example 146. Declaring a Scala dependency for test code

build.gradle.kts

```
dependencies {  
    testImplementation("org.scala-lang:scala-library:2.13.12")  
}
```

build.gradle

```
dependencies {  
    testImplementation("org.scala-lang:scala-library:2.13.12")  
}
```

Automatic configuration of `scalaClasspath`

The `ScalaCompile` and `ScalaDoc` tasks consume Scala code in two ways: on their `classpath`, and on their `scalaClasspath`. The former is used to locate classes referenced by the source code, and will typically contain `scala-library` along with other libraries. The latter is used to load and execute the Scala compiler and Scaladoc tool, respectively, and should only contain the `scala-compiler` library and its dependencies.

If the `scala` extension's `scalaVersion` property is not set and the task's `scalaClasspath` is not configured explicitly, the Scala (base) plugin will try to infer the classpath from the task's `classpath`. This is done as follows:

- If a `scala-library` jar is found on `classpath`, and the project has at least one repository declared, a corresponding `scala-compiler` repository dependency will be added to `scalaClasspath`.
- Otherwise, execution of the task will fail with a message saying that `scalaClasspath` could not be inferred.

Configuring the Zinc compiler

The Scala plugin uses a configuration named `zinc` to resolve the `Zinc compiler` and its dependencies. Gradle will provide a default version of Zinc, but if you need to use a particular Zinc version, you can change it. Gradle supports version 1.6.0 of Zinc and above.

Example 147. Declaring a version of the Zinc compiler to use

build.gradle.kts

```
scala {  
    scalaVersion = "2.13.12"  
    zincVersion = "1.10.4"  
}
```

build.gradle

```
scala {  
    scalaVersion = "2.13.12"  
    zincVersion = "1.10.4"  
}
```

The Zinc compiler itself needs a compatible version of `scala-library` that may be different from the version required by your application. Gradle takes care of specifying a compatible version of `scala-library` for you.

You can diagnose problems with the version of the Zinc compiler selected by running `dependencyInsight` for the `zinc` configuration.

Table 17. Zinc compatibility table

Gradle version	Supported Zinc versions	Zinc coordinates	Required Scala version	Supported Scala compilation version
7.5 and newer	SBT Zinc . Versions 1.6.0 and above.	<code>org.scala-sbt:zinc_2.13</code>	Scala 2.13.x is required for <i>running</i> Zinc.	Scala 2.10.x through 3.x can be compiled.
6.0 to 7.5	SBT Zinc . Versions 1.2.0 and above.	<code>org.scala-sbt:zinc_2.12</code>	Scala 2.12.x is required for <i>running</i> Zinc.	Scala 2.10.x through 2.13.x can be compiled.
1.x through 5.x	Deprecated Typesafe Zinc compiler . Versions 0.3.0 and above, except for 0.3.2 through 0.3.5.2.	<code>com.typesafe.zinc:zinc</code>	Scala 2.10.x is required for <i>running</i> Zinc.	Scala 2.9.x through 2.12.x can be compiled.

Adding plugins to the Scala compiler

The Scala plugin adds a configuration named `scalaCompilerPlugins` which is used to declare and resolve optional compiler plugins.

Example 148. Adding a dependency on a Scala compiler plugin

build.gradle.kts

```
dependencies {
    scalaCompilerPlugins("org.typelevel:kind-projector_2.13.12:0.13.2")
}
```

build.gradle

```
dependencies {
    scalaCompilerPlugins("org.typelevel:kind-projector_2.13.12:0.13.2")
}
```

Convention properties

The Scala plugin does not add any convention properties to the project.

Source set properties

The Scala plugin adds the following extensions to each source set in the project. You can use these in your build script as though they were properties of the source set object.

scala — **SourceDirectorySet** (read-only)

The Scala source files of this source set. Contains all **.scala** and **.java** files found in the Scala source directories, and excludes all other types of files. *Default value:* non-null.

scala.srcDirs — **Set<File>**

The source directories containing the Scala source files of this source set. May also contain Java source files for joint compilation. Can set using anything described in [Understanding implicit conversion to file collections](#). *Default value:* `[projectDir/src/name/scala]`.

allScala — **FileTree** (read-only)

All Scala source files of this source set. Contains only the **.scala** files found in the Scala source directories. *Default value:* non-null.

These extensions are backed by an object of type [ScalaSourceSet](#).

The Scala plugin also modifies some source set properties:

Table 18. Scala plugin - source set properties

Property name	Change
allJava	Adds all .java files found in the Scala source directories.
allSource	Adds all source files found in the Scala source directories.

Target bytecode level and Java APIs version

When running the Scala compile task, Gradle will always add a parameter to configure the Java target for the Scala compiler that is derived from the Gradle configuration:

- When using toolchains, the **-release** option, or **target** for older Scala versions, is selected, with a version matching the Java language level of the toolchain configured.
- When not using toolchains, Gradle will always pass a **target** flag — with exact value dependent on the Scala version — to compile to Java 8 bytecode.

NOTE

This means that using toolchains with a recent Java version and an old Scala version can result in failures because Scala only supported Java 8 bytecode for some time. The solution is then to either use the right Java version in the toolchain or explicitly downgrade the target when needed.

The following table explains the values computed by Gradle:

Table 19. Scala target parameter based on project configuration

Scala version	Toolchain in use	Parameter value
version < 2.13.1	yes	-target:jvm-1.<java_version>
	no	-target:jvm-1.8
2.13.1 <= version < 2.13.9	yes	-target:<java_version>
	no	-target:8

Scala version	Toolchain in use	Parameter value
2.13.9 <= version < 3.0	yes	-release:<java_version>
	no	-target:8
3.0 <= version	yes	-release:<java_version>
	no	-Xtarget:8

Setting any of these flags explicitly, or using flags containing `java-output-version`, on `ScalaCompile.scalaCompileOptions.additionalParameters` disables that logic in favor of the explicit flag.

Compiling in external process

Scala compilation takes place in an external process.

Memory settings for the external process default to the defaults of the JVM. To adjust memory settings, configure the `scalaCompileOptions.forkOptions` property as needed:

Example 149. Adjusting memory settings

build.gradle.kts

```
tasks.withType<ScalaCompile>().configureEach {
    scalaCompileOptions.forkOptions.apply {
        memoryMaximumSize = "1g"
        jvmArgs = listOf("-XX:MaxMetaspaceSize=512m")
    }
}
```

build.gradle

```
tasks.withType(ScalaCompile) {
    scalaCompileOptions.forkOptions.with {
        memoryMaximumSize = '1g'
        jvmArgs = ['-XX:MaxMetaspaceSize=512m']
    }
}
```

Incremental compilation

By compiling only classes whose source code has changed since the previous compilation, and classes affected by these changes, incremental compilation can significantly reduce Scala compilation time. It is particularly effective when frequently compiling small code increments, as is often done at development time.

The Scala plugin defaults to incremental compilation by integrating with [Zinc](#), a standalone version of [sbt](#)'s incremental Scala compiler. If you want to disable the incremental compilation, set `force = true` in your build file:

Example 150. Forcing all code to be compiled

build.gradle.kts

```
tasks.withType<ScalaCompile>().configureEach {
    scalaCompileOptions.apply {
        isForce = true
    }
}
```

build.gradle

```
tasks.withType(ScalaCompile) {
    scalaCompileOptions.with {
        force = true
    }
}
```

Note: This will only cause all classes to be recompiled if at least one input source file has changed. If there are no changes to the source files, the `compileScala` task will still be considered **UP-TO-DATE** as usual.

The Zinc-based Scala Compiler supports joint compilation of Java and Scala code. By default, all Java and Scala code under `src/main/scala` will participate in joint compilation. Even Java code will be compiled incrementally.

Incremental compilation requires dependency analysis of the source code. The results of this analysis are stored in the file designated by `scalaCompileOptions.incrementalOptions.analysisFile` (which has a sensible default). In a multi-project build, analysis files are passed on to downstream `ScalaCompile` tasks to enable incremental compilation across project boundaries. For `ScalaCompile` tasks added by the Scala plugin, no configuration is necessary to make this work. For other `ScalaCompile` tasks that you might add, the property `scalaCompileOptions.incrementalOptions.publishedCode` needs to be configured to point to the classes folder or Jar archive by which the code is passed on to compile class paths of downstream `ScalaCompile` tasks. Note that if `publishedCode` is not set correctly, downstream tasks may not recompile code affected by upstream changes, leading to incorrect compilation results.

Note that Zinc's Nailgun based daemon mode is not supported. Instead, we plan to enhance Gradle's own compiler daemon to stay alive across Gradle invocations, reusing the same Scala compiler. This is expected to yield another significant speedup for Scala compilation.

Eclipse Integration

When the Eclipse plugin encounters a Scala project, it adds additional configuration to make the project work with Scala IDE out of the box. Specifically, the plugin adds a Scala nature and dependency container.

IntelliJ IDEA Integration

When the IDEA plugin encounters a Scala project, it adds additional configuration to make the project work with IDEA out of the box. Specifically, the plugin adds a Scala SDK (IntelliJ IDEA 14+) and a Scala compiler library that matches the Scala version on the project's class path. The Scala plugin is backwards compatible with earlier versions of IntelliJ IDEA and it is possible to add a Scala facet instead of the default Scala SDK by configuring `targetVersion` on `IdeaModel`.

Example 151. Explicitly specify a target IntelliJ IDEA version

build.gradle.kts

```
idea {  
    targetVersion = "13"  
}
```

build.gradle

```
idea {  
    targetVersion = '13'  
}
```

[1] Gradle uses the same conventions as introduced by Russel Winder's [Gant tool](#).

INTEGRATION

Gradle & Third-party Tools

Gradle can be integrated with many different third-party tools such as IDEs and continuous integration platforms. Here we look at some of the more common ones as well as how to integrate your own tool with Gradle.

IDEs

Android Studio

As a variant of IntelliJ IDEA, [Android Studio](#) has built-in support for importing and building Gradle projects. You can also use the [IDEA Plugin for Gradle](#) to fine-tune the import process if that's necessary.

This IDE also has an [extensive user guide](#) to help you get the most out of the IDE and Gradle.

Eclipse

If you want to work on a project within Eclipse that has a Gradle build, you should use the [Eclipse Buildship plugin](#). This will allow you to import and run Gradle builds. If you need to fine tune the import process so that the project loads correctly, you can use the [Eclipse Plugins for Gradle](#). See [the associated release announcement](#) for details on what fine tuning you can do.

IntelliJ IDEA

IDEA has built-in support for importing Gradle projects. If you need to fine tune the import process so that the project loads correctly, you can use the [IDEA Plugin for Gradle](#).

NetBeans

Built-in support for Gradle in [Apache NetBeans](#)

Visual Studio

For developing C++ projects, Gradle comes with a [Visual Studio plugin](#).

Xcode

For developing C++ projects, Gradle comes with a [Xcode plugin](#).

CLion

JetBrains supports building [C++ projects with Gradle](#).

Continuous integration

We have [dedicated guides](#) showing you how to integrate a Gradle project with several CI platforms.

How to integrate with Gradle

There are two main ways to integrate a tool with Gradle:

- The Gradle build uses the tool
- The tool executes the Gradle build

The former case is typically [implemented as a Gradle plugin](#). The latter can be accomplished by embedding Gradle through the [Tooling API](#).

Test integration

If your platform uses an external testing suite, and you want to integrate with the [HTML Test Report](#), you can use the [Test Reporting API](#).

Tooling API

Embedding Gradle using the Tooling API

Introduction to the Tooling API

Gradle provides a programmatic API called the Tooling API, which you can use for embedding Gradle into your own software. This API allows you to execute and monitor builds and to query Gradle about the details of a build. The main audience for this API is IDE, CI server, other UI authors; however, the API is open for anyone who needs to embed Gradle in their application.

- [Gradle TestKit](#) uses the Tooling API for functional testing of your Gradle plugins.
- [Eclipse Buildship](#) uses the Tooling API for importing your Gradle project and running tasks.
- [IntelliJ IDEA](#) uses the Tooling API for importing your Gradle project and running tasks.

Tooling API Features

A fundamental characteristic of the Tooling API is that it operates in a version independent way. This means that you can use the same API to work with builds that use different versions of Gradle, including versions that are newer or older than the version of the Tooling API that you are using. The Tooling API is Gradle wrapper aware and, by default, uses the same Gradle version as that used by the wrapper-powered build.

Some features that the Tooling API provides:

- Query the details of a build, including the project hierarchy and the project dependencies, external dependencies (including source and Javadoc jars), source directories and tasks of each project.
- Execute a build and listen to stdout and stderr logging and progress messages (e.g. the messages shown in the 'status bar' when you run on the command line).
- Execute a specific test class or test method.
- Receive interesting events as a build executes, such as project configuration, task execution or test execution.
- Cancel a build that is running.
- Combine multiple separate Gradle builds into a single composite build.

- The Tooling API can download and install the appropriate Gradle version, similar to the wrapper.
- The implementation is lightweight, with only a small number of dependencies. It is also a well-behaved library, and makes no assumptions about your classloader structure or logging configuration. This makes the API easy to embed in your application.

Tooling API and the Gradle Build Daemon

The Tooling API always uses the Gradle daemon. This means that subsequent calls to the Tooling API, be it model building requests or task executing requests will be executed in the same long-living process. [Gradle Daemon](#) contains more details about the daemon, specifically information on situations when new daemons are forked.

Quickstart

As the Tooling API is an interface for developers, the Javadoc is the main documentation for it.

To use the Tooling API, add the following repository and dependency declarations to your build script:

build.gradle.kts

```
repositories {
    maven { url = uri("https://repo.gradle.org/gradle/libs-releases") }
}

dependencies {
    implementation("org.gradle:gradle-tooling-api:$toolingApiVersion")
    // The tooling API need an SLF4J implementation available at runtime,
    // replace this with any other implementation
    runtimeOnly("org.slf4j:slf4j-simple:1.7.10")
}
```

build.gradle

```
repositories {
    maven { url = 'https://repo.gradle.org/gradle/libs-releases' }
}

dependencies {
    implementation "org.gradle:gradle-tooling-api:$toolingApiVersion"
    // The tooling API need an SLF4J implementation available at runtime,
    // replace this with any other implementation
    runtimeOnly 'org.slf4j:slf4j-simple:1.7.10'
}
```

The main entry point to the Tooling API is the [GradleConnector](#). You can navigate from there to find code samples and explore the available Tooling API models. You can use [GradleConnector.connect\(\)](#) to create a [ProjectConnection](#). A [ProjectConnection](#) connects to a single Gradle project. Using the connection you can execute tasks, tests and retrieve models relative to this project.

Compatibility of Java and Gradle versions

The following components should be considered when implementing Gradle integration: the Tooling API version, The JVM running the Tooling API client (i.e. the IDE process), the JVM running the Gradle daemon, and the Gradle version.

The Tooling API itself is a Java library published as part of the Gradle release. Each Gradle release has a corresponding Tooling API version with the same version number.

The Tooling API classes are loaded into the client's JVM, so they should have a matching version. The current version of the Tooling API library is compiled with Java 8 compatibility.

The JVM running the Tooling API client and the one running the daemon can be different. At the same time, classes that are sent to the build via custom build actions need to be targeted to the lowest supported Java version. The JVM versions supported by Gradle is version-specific. The upper bound is defined in the [compatibility matrix](#). The rule for the lower bound is the following:

- Gradle 3.x and 4.x require a minimum version of Java 7.
- Gradle 5 and above require a minimum version of Java 8.

The Tooling API version is guaranteed to support running builds with all Gradle versions for the last five major releases. For example, the Tooling API 8.0 release is compatible with Gradle versions ≥ 3.0 . Besides, the Tooling API is guaranteed to be compatible with future Gradle releases for the current and the next major. This means, for example, that the 8.1 version of the Tooling API will be able to run Gradle 9.x builds and *might* break with Gradle 10.0.

Test Reporting API

The Test Reporting API allows you to capture test execution events programmatically and produce results similar to Gradle's built-in [Test](#) task. It's particularly useful to integrate non-JVM test results into Gradle's testing infrastructure and publish them in Gradle's HTML test reports.

The API is primarily targeted at plugin developers and platform providers, with the [Javadoc](#) serving as the main reference.

Let's take a look at a quick example that defines a custom Gradle task, [CustomTest](#), which demonstrates how to capture and report test events:

```
src/main/java/com/example/CustomTest.java
```

```
package com.example;  
  
import org.gradle.api.DefaultTask;
```

```

import org.gradle.api.file.ProjectLayout;
import org.gradle.api.tasks.TaskAction;
import org.gradle.api.tasks.testing.GroupTestEventReporter;
import org.gradle.api.tasks.testing.TestEventReporter;
import org.gradle.api.tasks.testing.TestEventReporterFactory;
import org.junit.platform.launcher.Launcher;
import org.junit.platform.launcher.LauncherDiscoveryRequest;
import org.junit.platform.launcher.core.LauncherDiscoveryRequestBuilder;
import org.junit.platform.launcher.core.LauncherFactory;
import org.junit.platform.launcher.listeners.SummaryGeneratingListener;
import org.junit.platform.launcher.listeners.TestExecutionSummary;

import javax.inject.Inject;
import java.time.Instant;
import java.util.ArrayList;
import java.util.List;

import static
org.junit.platform.engine.discovery.DiscoverySelectors.selectClass;

/**
 * A custom task that demonstrates the {@code TestEventReporter} API.
 */
public abstract class CustomTest extends DefaultTask {

    @Inject
    public abstract ProjectLayout getLayout();

    @Inject
    protected abstract TestEventReporterFactory getTestEventReporterFactory(
    );

    @TaskAction
    void runTests() {
        try (GroupTestEventReporter root = getTestEventReporterFactory()
            .createTestEventReporter(
                "root",
                getLayout().getBuildDirectory().dir("test-results/custom-test")
                    .get(),
                getLayout().getBuildDirectory().dir("reports/tests/custom-test")
                    .get()
            )) {
            root.started(Instant.now());

            List<String> failedTests = new ArrayList<>();

            try (GroupTestEventReporter junittest = root.reportTestGroup(
                "CustomJUnitTestSuite")) {
                junittest.started(Instant.now());
            }
        }
    }
}

```

```

        LauncherDiscoveryRequest request =
LauncherDiscoveryRequestBuilder.request()
        .selectors(
            selectClass(MyTest1.class)
        )
        .build();

        Launcher launcher = LauncherFactory.create();
        TestExecutionSummary summary = executeTests(launcher,
request);

        summary.getFailures().forEach(result -> {
            try (TestEventReporter test = junittest.reportTest(
result.getTestIdentifier().getDisplayName(),
            result.getTestIdentifier().getLegacyReportingName()))
            {
                test.started(Instant.now());
                String testName = String.valueOf(result
.getTestIdentifier().getParentIdObject());
                failedTests.add(testName);
                test.metadata(Instant.now(), "Parent class:", String
.valueOf(result.getTestIdentifier().getParentId().get()));
                test.failed(Instant.now(), String.valueOf(result
.getException()));
            }
        });

        String failedTestsList = String.join(", ", failedTests);
        junittest.metadata(Instant.now(), "Tests that failed:",
failedTestsList);

        if (summary.getTestsFailedCount() > 0) {
            junittest.failed(Instant.now());
        } else {
            junittest.succeeded(Instant.now());
        }
    }

    if (!failedTests.isEmpty()) {
        root.failed(Instant.now());
    } else {
        root.succeeded(Instant.now());
    }
}

private TestExecutionSummary executeTests(Launcher launcher,
LauncherDiscoveryRequest request) {
    // Use SummaryGeneratingListener to collect test execution data
    SummaryGeneratingListener listener = new SummaryGeneratingListener();
    launcher.registerTestExecutionListeners(listener);

```

```
// Execute the tests
launcher.execute(request);

// Return the summary generated by the listener
return listener.getSummary();
}
}
```

1. The `GroupTestEventListener` is initialized to act as the root event reporter.

It creates directories for test results and reports under the project's build directory:

This establishes the reporting hierarchy. **Nesting is supported.**

2. The task uses JUnit Platform to dynamically discover and execute tests, **this would be replaced with your custom test system/platform.**

It selects the class `MyTest1` for testing.

JUnit's `Launcher` is used to execute the tests and collect a summary of the results.

3. The `TestEventListener` is used to record detailed events for each test.

For each failure, the test is reported with metadata and marked as failed.

4. Each test group (`junittest`) and the root (`root`) are finalized with success or failure based on the test results.

By using the Test Reporting API:

- **Integration:** Custom tests (even those outside JVM frameworks) can generate events compatible with Gradle's test reporting infrastructure.
- **Rich Metadata:** Developers can attach additional context (e.g., error messages or parent class details) to test events.
- **Consistency:** Results integrate seamlessly with Gradle's HTML reports, making it easier to debug and understand test outcomes.

For more details on the HTML report itself, see the [JVM Testing Documentation](#).

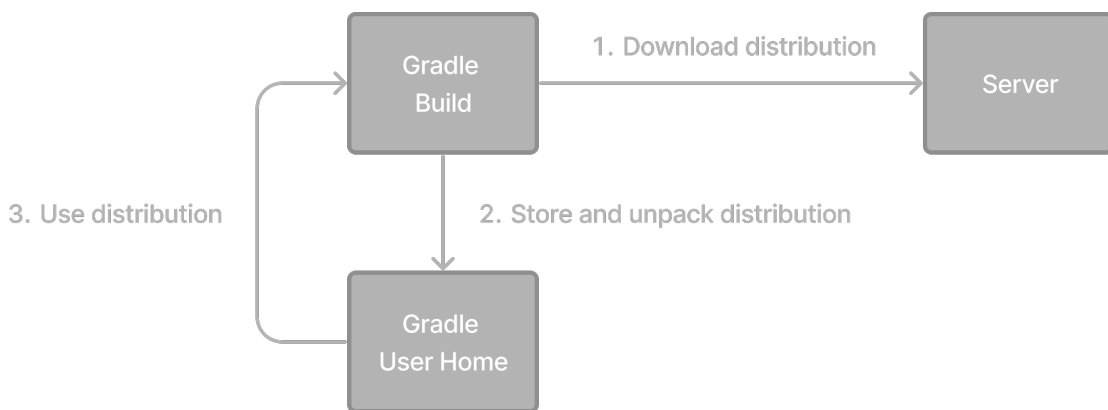
For a downloadable example, refer to the [Custom Test Task Sample](#).

REFERENCE

Gradle Wrapper Reference

The **recommended way to execute any Gradle build** is with the help of the Gradle Wrapper (referred to as "Wrapper").

The Wrapper is a script that invokes a declared version of Gradle, downloading it beforehand if necessary. As a result, developers can get up and running with a Gradle project quickly.



In a nutshell, you gain the following benefits:

- Standardizes a project on a given Gradle version for more reliable and robust builds.
- Provisioning the Gradle version for different users is done with a simple Wrapper definition change.
- Provisioning the Gradle version for different execution environments (e.g., IDEs or Continuous Integration servers) is done with a simple Wrapper definition change.

There are three ways to use the Wrapper:

1. You set up a new Gradle project and [add the Wrapper](#) to it.
2. You [run a project with the Wrapper](#) that already provides it.
3. You [upgrade the Wrapper](#) to a new version of Gradle.

The following sections explain each of these use cases in more detail.

Adding the Gradle Wrapper

Generating the Wrapper files requires an installed version of the Gradle runtime on your machine as described in [Installation](#). Thankfully, generating the initial Wrapper files is a one-time process.

Every vanilla Gradle build comes with a built-in task called `wrapper`. The task is listed under the

group "Build Setup tasks" when [listing the tasks](#).

Executing the `wrapper` task generates the necessary Wrapper files in the project directory:

```
$ gradle wrapper
```

```
> Task :wrapper
```

```
BUILD SUCCESSFUL in 0s
```

```
1 actionable task: 1 executed
```

TIP

To make the Wrapper files available to other developers and execution environments, you need to check them into version control. Wrapper files, including the JAR file, are small. Adding the JAR file to version control is expected. Some organizations do not allow projects to submit binary files to version control, and there is no workaround available.

The generated Wrapper properties file, `gradle/wrapper/gradle-wrapper.properties`, stores the information about the Gradle distribution:

- The **server hosting** the Gradle distribution.
- The **type of Gradle distribution**. By default, the `-bin` distribution contains only the runtime but no sample code and documentation.
- The **Gradle version** used for executing the build. By default, the `wrapper` task picks the same Gradle version used to generate the Wrapper files.
- Optionally, a **timeout** in ms used when downloading the Gradle distribution.
- Optionally, a **boolean** to set the **validation of the distribution** URL.

The following is an example of the generated distribution URL in `gradle/wrapper/gradle-wrapper.properties`:

```
distributionUrl=https\://services.gradle.org/distributions/gradle-8.13-bin.zip
```

All of those aspects are configurable at the time of generating the Wrapper files with the help of the following command line options:

--gradle-version

The Gradle version used for downloading and executing the Wrapper. The resulting distribution URL is validated before it is written to the properties file.

The following labels are allowed:

- `latest`
- `release-candidate`

- [nightly](#)
- [release-nightly](#)

--distribution-type

The Gradle distribution type used for the Wrapper. Available options are **bin** and **all**. The default value is **bin**.

--gradle-distribution-url

The full URL pointing to the Gradle distribution ZIP file. This option makes **--gradle-version** and **--distribution-type** obsolete, as the URL already contains this information. This option is valuable if you want to host the Gradle distribution inside your company's network. The URL is validated before it is written to the properties file.

--gradle-distribution-sha256-sum

The SHA256 hash sum used for [verifying the downloaded Gradle distribution](#).

--network-timeout

The network timeout to use when downloading the Gradle distribution, in ms. The default value is **10000**.

--no-validate-url

Disables the validation of the configured distribution URL.

--validate-url

Enables the validation of the configured distribution URL. Enabled by default.

If the distribution URL is configured with **--gradle-version** or **--gradle-distribution-url**, the URL is validated by sending a HEAD request in the case of the **https** scheme or by checking the existence of the file in the case of the **file** scheme.

Let's assume the following use-case to illustrate the use of the command line options. You would like to generate the Wrapper with version 8.13 and use the **-all** distribution to enable your IDE to enable code-completion and being able to navigate to the Gradle source code.

The following command-line execution captures those requirements:

```
$ gradle wrapper --gradle-version 8.13 --distribution-type all
> Task :wrapper

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

As a result, you can find the desired information (the generated distribution URL) in the Wrapper properties file:

```
distributionUrl=https\://services.gradle.org/distributions/gradle-8.13-all.zip
```

Let's have a look at the following project layout to illustrate the expected Wrapper files:

```
.
├── a-subproject
│   ├── build.gradle.kts
│   └── settings.gradle.kts
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
└── gradlew.bat
```

```
.
├── a-subproject
│   ├── build.gradle
│   └── settings.gradle
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
└── gradlew.bat
```

A Gradle project typically provides a `settings.gradle(.kts)` file and one `build.gradle(.kts)` file for each subproject. The Wrapper files live alongside in the `gradle` directory and the root directory of the project.

The following list explains their purpose:

`gradle-wrapper.jar`

The Wrapper JAR file containing code for downloading the Gradle distribution.

`gradle-wrapper.properties`

A properties file responsible for configuring the Wrapper runtime behavior e.g. the Gradle version compatible with this version. Note that more generic settings, like [configuring the Wrapper to use a proxy](#), need to go into a [different file](#).

`gradlew, gradlew.bat`

A shell script and a Windows batch script for executing the build with the Wrapper.

You can go ahead and [execute the build with the Wrapper](#) without installing the Gradle runtime. If the project you are working on does not contain those Wrapper files, you will need to [generate](#)

them.

Using the Gradle Wrapper

It is always recommended to execute a build with the Wrapper to ensure a reliable, controlled, and standardized execution of the build. Using the Wrapper looks like running the build with a Gradle installation. Depending on the operating system you either run `gradlew` or `gradlew.bat` instead of the `gradle` command.

The following console output demonstrates the use of the Wrapper on a Windows machine for a Java-based project:

```
$ gradlew.bat build
Downloading https://services.gradle.org/distributions/gradle-5.0-all.zip
.....
Unzipping C:\Documents and Settings\Claudia\.gradle\wrapper\dists\gradle-5.0-
all\ac27o8rbd0ic8ih41or9l32mv\gradle-5.0-all.zip to C:\Documents and
Settings\Claudia\.gradle\wrapper\dists\gradle-5.0-al\ac27o8rbd0ic8ih41or9l32mv
Set executable permissions for: C:\Documents and
Settings\Claudia\.gradle\wrapper\dists\gradle-5.0-
all\ac27o8rbd0ic8ih41or9l32mv\gradle-5.0\bin\gradle

BUILD SUCCESSFUL in 12s
1 actionable task: 1 executed
```

If the Gradle distribution is unavailable on the machine, the Wrapper will download it and store it in the local file system. Any subsequent build invocation will reuse the existing local distribution as long as the distribution URL in the Gradle properties doesn't change.

NOTE

The Wrapper shell script and batch file reside in the root directory of a single or multi-project Gradle build. You will need to reference the correct path to those files in case you want to execute the build from a subproject directory e.g. `../../gradlew tasks`.

Upgrading the Gradle Wrapper

Projects typically want to keep up with the times and upgrade their Gradle version to benefit from new features and improvements.

One way to upgrade the Gradle version is by manually changing the `distributionUrl` property in the Wrapper's `gradle-wrapper.properties` file.

The better and recommended option is to run the `wrapper` task and provide the target Gradle version as described in [Adding the Gradle Wrapper](#). Using the `wrapper` task ensures that any optimizations made to the Wrapper shell script or batch file with that specific Gradle version are applied to the project.

As usual, you should commit the changes to the Wrapper files to version control.

Note that running the wrapper task once will update `gradle-wrapper.properties` only, but leave the wrapper itself in `gradle-wrapper.jar` untouched. This is usually fine as new versions of Gradle can be run even with older wrapper files.

NOTE

If you want **all** the wrapper files to be completely up-to-date, you will need to run the `wrapper` task a second time.

The following command upgrades the Wrapper to the `latest` version:

```
$ ./gradlew wrapper --gradle-version latest
```

```
BUILD SUCCESSFUL in 4s  
1 actionable task: 1 executed
```

The following command upgrades the Wrapper to a specific version:

```
$ ./gradlew wrapper --gradle-version 8.13
```

```
BUILD SUCCESSFUL in 4s  
1 actionable task: 1 executed
```

Once you have upgraded the wrapper, you can check that it's the version you expected by executing `./gradlew --version`.

Don't forget to run the `wrapper` task again to download the Gradle distribution binaries (if needed) and update the `gradlew` and `gradlew.bat` files.

Customizing the Gradle Wrapper

Most users of Gradle are happy with the default runtime behavior of the Wrapper. However, organizational policies, security constraints or personal preferences might require you to dive deeper into customizing the Wrapper.

Thankfully, the built-in `wrapper` task exposes numerous options to bend the runtime behavior to your needs. Most configuration options are exposed by the underlying task type `Wrapper`.

Let's assume you grew tired of defining the `-all` distribution type on the command line every time you upgrade the Wrapper. You can save yourself some keyboard strokes by re-configuring the `wrapper` task.

build.gradle.kts

```
tasks.wrapper {  
    distributionType = Wrapper.DistributionType.ALL  
}
```

build.gradle

```
tasks.named('wrapper') {  
    distributionType = Wrapper.DistributionType.ALL  
}
```

With the configuration in place, running `./gradlew wrapper --gradle-version 8.13` is enough to produce a `distributionUrl` value in the Wrapper properties file that will request the `-all` distribution:

```
distributionUrl=https\://services.gradle.org/distributions/gradle-8.13-all.zip
```

Check out the [API documentation](#) for a more detailed description of the available configuration options. You can also find various samples for configuring the Wrapper in the Gradle distribution.

Authenticated Gradle distribution download

The Gradle `Wrapper` can download Gradle distributions from servers using HTTP Basic Authentication. This enables you to host the Gradle distribution on a private protected server.

You can specify a username and password in two different ways depending on your use case: as system properties or directly embedded in the `distributionUrl`. Credentials in system properties take precedence over the ones embedded in `distributionUrl`.

TIP

HTTP Basic Authentication should only be used with `HTTPS` URLs and not plain `HTTP` ones. With Basic Authentication, the user credentials are sent in clear text.

System properties can be specified in the `.gradle/gradle.properties` file in the user's home directory or by other [means](#).

To specify the HTTP Basic Authentication credentials, add the following lines to the system properties file:

```
systemProp.gradle.wrapperUser=username  
systemProp.gradle.wrapperPassword=password
```

Embedding credentials in the `distributionUrl` in the `gradle/wrapper/gradle-wrapper.properties` file also works. Please note that this file is to be committed into your source control system.

TIP

Shared credentials embedded in `distributionUrl` should only be used in a controlled environment.

To specify the HTTP Basic Authentication credentials in `distributionUrl`, add the following line:

```
distributionUrl=https://username:password@somehost/path/to/gradle-distribution.zip
```

This can be used in conjunction with a proxy, authenticated or not. See [Accessing the web via a proxy](#) for more information on how to configure the **Wrapper** to use a proxy.

Verification of downloaded Gradle distributions

The Gradle Wrapper allows for verification of the downloaded Gradle distribution via SHA-256 hash sum comparison. This increases security against targeted attacks by preventing a man-in-the-middle attacker from tampering with the downloaded Gradle distribution.

To enable this feature, download the **.sha256** file associated with the Gradle distribution you want to verify.

Downloading the SHA-256 file

You can download the **.sha256** file from the [stable releases](#) or [release candidate and nightly releases](#). The format of the file is a single line of text that is the SHA-256 hash of the corresponding zip file.

You can also reference the [list of Gradle distribution checksums](#).

Configuring checksum verification

Add the downloaded (SHA-256 checksum) hash sum to **gradle-wrapper.properties** using the **distributionSha256Sum** property or use **--gradle-distribution-sha256-sum** on the command-line:

```
distributionSha256Sum=371cb9fb9bbe9880d147f59bab36d61eee122854ef8c9ee1ecf12b82368bcf10
```

Gradle will report a build failure if the configured checksum does not match the checksum found on the server hosting the distribution. Checksum verification is only performed if the configured Wrapper distribution hasn't been downloaded yet.

NOTE

The **Wrapper** task fails if **gradle-wrapper.properties** contains **distributionSha256Sum**, but the task configuration does not define a sum. Executing the **Wrapper** task preserves the **distributionSha256Sum** configuration when the Gradle version does not change.

Verifying the integrity of the Gradle Wrapper JAR

The Wrapper JAR is a binary file that will be executed on the computers of developers and build servers. As with all such files, you should ensure it's trustworthy before executing it.

Since the Wrapper JAR is usually checked into a project's version control system, there is the potential for a malicious actor to replace the original JAR with a modified one by submitting a pull request that only upgrades the Gradle version.

To verify the integrity of the Wrapper JAR, Gradle has created a [GitHub Action](#) that automatically

checks Wrapper JARs in pull requests against a list of known good checksums.

Gradle also publishes the [checksums of all releases](#) (except for version 3.3 to 4.0.2, which did not generate reproducible JARs), so you can manually verify the integrity of the Wrapper JAR.

Automatically verifying the Gradle Wrapper JAR on GitHub

The [GitHub Action](#) is released separately from Gradle, so please check its documentation for how to apply it to your project.

Manually verifying the Gradle Wrapper JAR

You can manually verify the checksum of the Wrapper JAR to ensure that it has not been tampered with by running the following commands on one of the major operating systems.

Manually verifying the checksum of the Wrapper JAR on Linux:

```
$ cd gradle/wrapper
```

```
$ curl --location --output gradle-wrapper.jar.sha256 \  
      https://services.gradle.org/distributions/gradle-{gradleVersion}-  
wrapper.jar.sha256
```

```
$ echo " gradle-wrapper.jar" >> gradle-wrapper.jar.sha256
```

```
$ sha256sum --check gradle-wrapper.jar.sha256
```

```
gradle-wrapper.jar: OK
```

Manually verifying the checksum of the Wrapper JAR on macOS:

```
$ cd gradle/wrapper
```

```
$ curl --location --output gradle-wrapper.jar.sha256 \  
      https://services.gradle.org/distributions/gradle-{gradleVersion}-  
wrapper.jar.sha256
```

```
$ echo " gradle-wrapper.jar" >> gradle-wrapper.jar.sha256
```

```
$ shasum --check gradle-wrapper.jar.sha256
```

```
gradle-wrapper.jar: OK
```

Manually verifying the checksum of the Wrapper JAR on Windows (using PowerShell):

```
> $expected = Invoke-RestMethod -Uri https://services.gradle.org/distributions/gradle-8.13-wrapper.jar.sha256
```

```
> $actual = (Get-FileHash gradle\wrapper\gradle-wrapper.jar -Algorithm SHA256).Hash.ToLower()
```

```
> @{$true = 'OK: Checksum match'; $false = "ERROR: Checksum mismatch!\nExpected: $expected\nActual:   $actual"}[$actual -eq $expected]
```

```
OK: Checksum match
```

Troubleshooting a checksum mismatch

If the checksum does not match the one you expected, chances are the **wrapper** task wasn't executed with the upgraded Gradle distribution.

You should first check whether the actual checksum matches a different Gradle version.

Here are the commands you can run on the major operating systems to generate the actual checksum of the Wrapper JAR.

Generating the checksum of the Wrapper JAR on Linux:

```
$ sha256sum gradle/wrapper/gradle-wrapper.jar
d81e0f23ade952b35e55333dd5f1821585e887c6d24305aeea2fbc8dad564b95
gradle/wrapper/gradle-wrapper.jar
```

Generating the actual checksum of the Wrapper JAR on macOS:

```
$ shasum --algorithm=256 gradle/wrapper/gradle-wrapper.jar
d81e0f23ade952b35e55333dd5f1821585e887c6d24305aeea2fbc8dad564b95
gradle/wrapper/gradle-wrapper.jar
```

Generating the actual checksum of the Wrapper JAR on Windows (using PowerShell):

```
> (Get-FileHash gradle\wrapper\gradle-wrapper.jar -Algorithm SHA256).Hash.ToLower()  
d81e0f23ade952b35e55333dd5f1821585e887c6d24305aeea2fbc8dad564b95
```

Once you know the actual checksum, check whether it's listed on <https://gradle.org/release-checksums/>. If it is listed, you have verified the integrity of the Wrapper JAR. If the version of Gradle that generated the Wrapper JAR doesn't match the version in `gradle/wrapper/gradle-wrapper.properties`, it's safe to run the `wrapper` task again to update the Wrapper JAR.

If the checksum is not listed on the page, the Wrapper JAR might be from a milestone, release candidate, or nightly build or may have been generated by Gradle 3.3 to 4.0.2. Try to find out how it was generated but treat it as untrustworthy until proven otherwise. If you think the Wrapper JAR was compromised, please let the Gradle team know by sending an email to security@gradle.com.

Gradle Daemon

A daemon is a computer program that runs as a background process rather than being under the direct control of an interactive user.

Gradle runs on the Java Virtual Machine (JVM) and uses several supporting libraries with non-trivial initialization time. Startups can be slow. The **Gradle Daemon** solves this problem.

The Gradle Daemon is a long-lived background process that reduces the time it takes to run a build.

The Gradle Daemon reduces build times by:

- Caching project information across builds
- Running in the background so every Gradle build doesn't have to wait for JVM startup
- Benefiting from continuous runtime optimization in the JVM
- [Watching the file system](#) to calculate exactly what needs to be rebuilt before you run a build

Understanding the Daemon

The Gradle JVM client sends the Daemon build information such as command line arguments, project directories, and environment variables so that it can run the build. The Daemon is responsible for resolving dependencies, executing build scripts, creating and running tasks; when it is done, it sends the client the output. Communication between the client and the Daemon happens via a local socket connection.

Daemons use the JVM's default minimum heap size.

If the requested build environment does not specify a maximum heap size, the Daemon uses up to 512MB of heap. 512MB is adequate for most builds. Larger builds with hundreds of subprojects, configuration, and source code may benefit from a larger heap size.

Check Daemon status

To get a list of running Daemons and their statuses, use the `--status` command:

```
$ gradle --status
```

PID	STATUS	INFO
28486	IDLE	7.5
34247	BUSY	7.5

Currently, a given Gradle version can only connect to Daemons of the same version. This means the status output only shows Daemons spawned running the same version of Gradle as the current project.

Find Daemons

If you have installed the Java Development Kit (JDK), you can view live daemons with the `jps` command.

```
$ jps
```

```
33920 Jps
27171 GradleDaemon
22792
```

Live Daemons appear under the name `GradleDaemon`. Because this command uses the JDK, you can view Daemons running any version of Gradle.

Enable Daemon

Gradle enables the Daemon by default since Gradle 3.0. If your project doesn't use the Daemon, you can enable it for a single build with the `--daemon` flag when you run a build:

```
$ gradle <task> --daemon
```

This flag overrides any settings that disable the Daemon in your project or user `gradle.properties` files.

To enable the Daemon by default in older Gradle versions, add the following setting to the `gradle.properties` file in the project root or your Gradle User Home (`GRADLE_USER_HOME`):

gradle.properties

```
org.gradle.daemon=true
```

Disable Daemon

You can disable the Daemon in multiple ways but there are important considerations:

Single-use Daemon

If the JVM args of the client process don't match what the build requires, a single-used Daemon (disposable JVM) is created. This means the Daemon is required for the build, so it is created, used, and then stopped at the end of the build.

No Daemon

If the `JAVA_OPTS` and `GRADLE_OPTS` match `org.gradle.jvmargs`, the Daemon will not be used at all since the build happens in the client JVM.

Disable for a build

To disable the Daemon for a single build, pass the `--no-daemon` flag when you run a build:

```
$ gradle <task> --no-daemon
```

This flag overrides any settings that enable the Daemon in your project including the `gradle.properties` files.

Disable for a project

To disable the Daemon for all builds of a project, add `org.gradle.daemon=false` to the `gradle.properties` file in the project root.

Disable for a user

On Windows, this command disables the Daemon for the current user:

```
(if not exist "%USERPROFILE%\.gradle" mkdir "%USERPROFILE%\.gradle") && (echo. >>
"%USERPROFILE%\.gradle\gradle.properties" && echo org.gradle.daemon=false >>
"%USERPROFILE%\.gradle\gradle.properties")
```

On UNIX-like operating systems, the following Bash shell command disables the Daemon for the current user:

```
mkdir -p ~/.gradle && echo "org.gradle.daemon=false" >> ~/.gradle/gradle.properties
```

Disable globally

There are two recommended ways to disable the Daemon globally across an environment:

- add `org.gradle.daemon=false` to the `$GRADLE_USER_HOME/gradle.properties` file
- add the flag `-Dorg.gradle.daemon=false` to the `GRADLE_OPTS` environment variable

Don't forget to make sure your JVM arguments and `GRADLE_OPTS` / `JAVA_OPTS` match if you want to completely disable the Daemon and not simply invoke a single-use one.

Stop Daemon

It can be helpful to stop the Daemon when troubleshooting or debugging a failure.

Daemons automatically stop given any of the following conditions:

- Available system memory is low
- Daemon has been idle for 3 hours

To stop running Daemon processes, use the following command:

```
$ gradle --stop
```

This terminates all Daemon processes started with the same version of Gradle used to execute the command.

You can also kill Daemons manually with your operating system. To find the PIDs for all Daemons regardless of Gradle version, see [Find Daemons](#).

Configuring the Daemon JVM

NOTE

Daemon JVM discovery and criteria are [incubating](#) features and are subject to change in a future release.

By default, the Gradle Daemon runs with the same JVM installation that started the build. Gradle defaults to the current shell path and `JAVA_HOME` environment variable to locate a usable JVM.

Alternatively, a different JVM installation can be specified for the build using the `org.gradle.java.home` [Gradle property](#) or programmatically through the Tooling API.

Building on the [toolchain feature](#), you can now use declarative criteria to specify the JVM requirements for the build.

If the Daemon JVM criteria configuration is provided, it takes precedence over `JAVA_HOME` and `org.gradle.java.home`.

Daemon JVM criteria

The *Daemon JVM criteria* is controlled by the `updateDaemonJvm` task, similar to how the [wrapper task](#) updates the wrapper properties file.

WARNING

This process requires [toolchain download repositories](#) to be configured. See [below](#) for details.

When the task runs, it creates or updates the criteria in the `gradle/gradle-daemon-jvm.properties`

file.

To configure the generation, you can use command line options:

```
$ ./gradlew updateDaemonJvm --jvm-version=17
```

Or configure the task in the build script of the root project:

build.gradle.kts

```
tasks.named<UpdateDaemonJvm>("updateDaemonJvm") {  
    languageVersion = JavaLanguageVersion.of(17)  
}
```

build.gradle

```
tasks.named("updateDaemonJvm") {  
    languageVersion = JavaLanguageVersion.of(17)  
}
```

And then run the task:

```
$ ./gradlew updateDaemonJvm
```

Both of these actions will produce a file like the following one:

gradle/gradle-daemon-jvm.properties

```
#This file is generated by updateDaemonJvm  
toolchainUrl.FREE_BSD.AARCH64=https://example.com/...  
toolchainUrl.FREE_BSD.X86_64=https://example.com/...  
toolchainUrl.LINUX.AARCH64=https://example.com/...  
toolchainUrl.LINUX.X86_64=https://example.com/...  
toolchainUrl.MAC_OS.AARCH64=https://example.com/...  
toolchainUrl.MAC_OS.X86_64=https://example.com/...  
toolchainUrl.UNIX.AARCH64=https://example.com/...  
toolchainUrl.UNIX.X86_64=https://example.com/...  
toolchainUrl.WINDOWS.X86_64=https://example.com/...  
toolchainVersion=17
```

If you run the `updateDaemonJvm` task without any arguments, and the properties file does not exist,

then the version of the current JVM used by the Daemon will be used.

On the next execution of the build, the Gradle client will use this file to locate a compatible JVM installation and start the Daemon with it.

Similar to the wrapper, the generated `gradle-daemon-jvm.properties` file should be checked into version control. This ensures that any developer or CI server running the build will use the same JVM version.

Specifying a JVM vendor

The JVM vendor, like the JVM version, can be used as a criteria to select a compatible JVM installation for the build. If no vendor is specified, Gradle considers all vendors compatible.

By default, running `updateDaemonJvm` to create the `gradle-daemon-jvm.properties` file will not generate a JVM vendor criteria. To specify a vendor, either configure it in the build script, using [the same syntax as the Java toolchain spec](#), or pass it on the command line:

```
$ ./gradlew updateDaemonJvm --jvm-version=17 --jvm-vendor=adoptium
```

List of recognized vendors:

Known Vendors	Acceptable Strings	<code>toolchainVendor</code> Value
Adoptium / Eclipse Temurin	<code>adoptium</code> , <code>temurin</code> , <code>eclipse foundation</code>	ADOPTIUM
AdoptOpenJDK	<code>adoptopenjdk</code> , <code>aoj</code>	ADOPTOPENJDK
Amazon Corretto	<code>amazon</code> , <code>corretto</code>	AMAZON
Apple	<code>apple</code>	APPLE
Azul Zulu	<code>azul</code> , <code>zulu</code>	AZUL
BellSoft	<code>bellsoft</code> , <code>liberica</code>	BELLSOFT
GraalVM	<code>graalvm</code> , <code>graal vm</code>	GRAAL_VM
Hewlett Packard	<code>hp</code> , <code>hewlett</code>	HEWLETT_PACKARD
IBM	<code>ibm</code> , <code>semeru</code> , <code>international business machines corporation</code>	IBM
JetBrains	<code>jetbrains</code> , <code>jbr</code>	JETBRAINS
Microsoft	<code>microsoft</code>	MICROSOFT
Oracle	<code>oracle</code>	ORACLE
SAP	<code>sap</code>	SAP
Tencent	<code>tencent</code> , <code>kona</code>	TENCENT

Some vendors will be recognized from more than one set of characters. All vendor strings are case-insensitive. You can view the list of recognized vendors by running `./gradlew help --task updateDaemonJvm`.

If the specified vendor is not one of the recognized equivalents, Gradle will match it exactly. For example, "MyCustomJVM" would require an exact match of the vendor name.

Auto-detection and auto-provisioning

The Daemon JVM is *auto-detected* using the same logic as [project JVM toolchains](#).

With *auto-provisioning*, the logic is simpler, as Gradle can only look up a download URL matching the platform inside the `gradle-daemon-jvm.properties` file. The URL is then used to download a JDK if none can be found locally.

The properties used for disabling *auto-detection* and *auto-provisioning* affect the Daemon toolchain resolution logic:

```
org.gradle.java.installations.auto-detect=false
org.gradle.java.installations.auto-download=false
```

Configuring provisioning URLs

NOTE | There are currently no CLI options for configuring this.

By default, the `updateDaemonJvm` task attempts to generate download URLs for JDKs on various platforms (OS and architecture) that match the specified criteria. Gradle needs to consider more than the current running platform as the build could be run on different platforms.

Gradle sets, by convention, build platforms based on architectures `X86_64` and `AARCH64` for the following operating systems:

- [FreeBSD](#)
- [Linux](#)
- [macOS](#)
- [Solaris](#)
- [UNIX](#)
- [Windows](#)

These platforms can be configured through the `toolchainPlatforms` property of the `UpdateDaemonJvm` task.

build.gradle.kts

```
tasks.named<UpdateDaemonJvm>("updateDaemonJvm") {
    val myPlatforms = mutableListOf(
        BuildPlatformFactory.of(
            org.gradle.platform.Architecture.AARCH64,
            org.gradle.platform.OperatingSystem.MAC_OS
        )
    )
}
```

```

    )
    toolchainPlatforms.set(myPlatforms)
}

```

build.gradle

```

tasks.named("updateDaemonJvm") {
    def myPlatforms = [
        BuildPlatformFactory.of(
            org.gradle.platform.Architecture.AARCH64,
            org.gradle.platform.OperatingSystem.MAC_OS
        )
    ]
    toolchainPlatforms.set(myPlatforms)
}

```

Gradle resolves JDK download URLs for these platforms by using the [configured toolchain download repositories](#). If no such repositories are configured and the `toolchainPlatforms` property has at least one value, the `updateDaemonJvm` task will fail.

Alternatively, users can directly configure the JDK URLs for specific platforms using the `toolchainDownloadUrls` property. This property is a `Map<BuildPlatform, URI>` and can be configured as shown in the following example:

build.gradle.kts

```

tasks.named<UpdateDaemonJvm>("updateDaemonJvm") {
    toolchainDownloadUrls = mapOf(
        BuildPlatformFactory.of(org.gradle.platform.Architecture.AARCH64,
            org.gradle.platform.OperatingSystem.MAC_OS) to
        uri("https://server?platform=MAC_OS.AARCH64"),
        BuildPlatformFactory.of(org.gradle.platform.Architecture.AARCH64,
            org.gradle.platform.OperatingSystem.WINDOWS) to
        uri("https://server?platform=WINDOWS.AARCH64")
    )
}

```

build.gradle

```

tasks.named("updateDaemonJvm") {
    toolchainDownloadUrls = [(BuildPlatformFactory.of(org.gradle.platform
        .Architecture.AARCH64, org.gradle.platform.OperatingSystem.MAC_OS)) : uri(
        "https://server?platform=MAC_OS.AARCH64"),

```

```
(BuildPlatformFactory.of(org.gradle.platform
.Architecture.AARCH64, org.gradle.platform.OperatingSystem.WINDOWS)) : uri(
"https://server?platform=WINDOWS.AARCH64")]
```

TIP

A full package name is required for `org.gradle.platform.Architecture` and `org.gradle.platform.OperatingSystem` due to a naming conflict with other types in different packages, which are resolved first alphabetically.

Running `./gradlew updateDaemonJvm` produces the following:

gradle/gradle-daemon-jvm.properties

```
#This file is generated by updateDaemonJvm
toolchainUrl.MAC_OS.AARCH64=https\://server?platform\=MAC_OS.AARCH64
toolchainUrl.WINDOWS.AARCH64=https\://server?platform\=WINDOWS.AARCH64
toolchainVersion=17
```

If you want to disable the generation of URLs by the `updateDaemonJvm` task:

build.gradle.kts

```
tasks.named<UpdateDaemonJvm>("updateDaemonJvm") {
    toolchainDownloadUrls.empty()
}
```

build.gradle

```
tasks.named("updateDaemonJvm") {
    toolchainPlatforms = []
}
```

Removing all platforms means that there is no longer a need for the [toolchain download repositories](#) to be configured.

Tools & IDEs

The [Gradle Tooling API](#) used by IDEs and other tools to integrate with Gradle *always* uses the Gradle Daemon to execute builds. If you execute Gradle builds from within your IDE, you already use the Gradle Daemon. There is no need to enable it for your environment.

Continuous Integration

We recommend using the Daemon for developer machines and Continuous Integration (CI) servers.

Compatibility

Gradle starts a new Daemon if no idle or compatible Daemons exist.

The following values determine compatibility:

- **Requested build environment**, including the following:
 - Java version
 - JVM attributes
 - JVM properties
- Gradle version

Compatibility is based on exact matches of these values. For example:

- If a Daemon is available with a Java 8 runtime, but the requested build environment calls for Java 10, then the Daemon is not compatible.
- If a Daemon is available running Gradle 7.0, but the current build uses Gradle 7.4, then the Daemon is not compatible.

Certain properties of a Java runtime are *immutable*: they cannot be changed once the JVM has started. The following JVM system properties are immutable:

- `file.encoding`
- `user.language`
- `user.country`
- `user.variant`
- `java.io.tmpdir`
- `javax.net.ssl.keyStore`
- `javax.net.ssl.keyStorePassword`
- `javax.net.ssl.keyStoreType`
- `javax.net.ssl.trustStore`
- `javax.net.ssl.trustStorePassword`
- `javax.net.ssl.trustStoreType`
- `com.sun.management.jmxremote`

The following JVM attributes controlled by startup arguments are also immutable:

- The maximum heap size (the `-Xmx` JVM argument)
- The minimum heap size (the `-Xms` JVM argument)

- The boot classpath (the `-Xbootclasspath` argument)
- The "assertion" status (the `-ea` argument)

If the requested build environment requirements for any of these properties and attributes differ from the Daemon's JVM requirements, the Daemon is not compatible.

NOTE

For more information about build environments, see [the build environment documentation](#).

Performance Impact

The Daemon can reduce build times by 15-75% when you build the same project repeatedly.

In between builds, the Daemon waits idly for the next build. As a result, your machine only loads Gradle into memory once for multiple builds instead of once per build. This is a significant performance optimization.

Runtime Code Optimizations

The JVM gains significant performance from **runtime code optimization**: optimizations applied to code while it runs.

JVM implementations like OpenJDK's Hotspot progressively optimize code during execution. Consequently, subsequent builds can be faster purely due to this optimization process.

With the Daemon, perceived build times can drop dramatically between a project's 1st and 10th builds.

Memory Caching

The Daemon enables in-memory caching across builds. This includes classes for plugins and build scripts.

Similarly, the Daemon maintains in-memory caches of build data, such as the hashes of task inputs and outputs for incremental builds.

Performance Monitoring

Gradle actively monitors heap usage to detect memory leaks in the Daemon.

When a memory leak exhausts available heap space, the Daemon:

1. Finishes the currently running build.
2. Restarts before running the next build.

Gradle enables this monitoring by default.

To disable this monitoring, set the `org.gradle.daemon.performance.enable-monitoring` Daemon option to `false`.

You can do this on the command line with the following command:

```
$ gradle <task> -Dorg.gradle.daemon.performance.enable-monitoring=false
```

Or you can configure the property in the `gradle.properties` file in the project root or your `GRADLE_USER_HOME` (Gradle User Home):

gradle.properties

```
org.gradle.daemon.performance.enable-monitoring=false
```

Command-Line Interface Reference

The command-line interface is the **primary method of interacting with Gradle**.

The following is a reference for executing and customizing the Gradle command-line. It also serves as a reference when writing scripts or configuring continuous integration.

Use of the [Gradle Wrapper](#) is highly encouraged. Substitute `./gradlew` (in macOS / Linux) or `gradlew.bat` (in Windows) for `gradle` in the following examples.

Executing Gradle on the command-line conforms to the following structure:

```
gradle [taskName...] [--option-name...]
```

Options are allowed *before* and *after* task names.

```
gradle [--option-name...] [taskName...]
```

If multiple tasks are specified, you should separate them with a space.

```
gradle [taskName1 taskName2...] [--option-name...]
```

Options that accept values can be specified with or without `=` between the option and argument. The use of `=` is recommended.

```
gradle [...] --console=plain
```

Options that enable behavior have long-form options with inverses specified with `--no-`. The following are opposites.

```
gradle [...] --build-cache
```

```
gradle [...] --no-build-cache
```

Many long-form options have short-option equivalents. The following are equivalent:

```
gradle --help  
gradle -h
```

NOTE

Many command-line flags can be specified in `gradle.properties` to avoid needing to be typed. See the [Configuring build environment guide](#) for details.

Command-line usage

The following sections describe the use of the Gradle command-line interface.

Some plugins also add their own command line options. For example, `--tests`, which is added by [Java test filtering](#). For more information on exposing command line options for your own tasks, see [Declaring command-line options](#).

Executing tasks

You can learn about what projects and tasks are available in the [project reporting section](#).

Most builds support a common set of tasks known as *lifecycle tasks*. These include the `build`, `assemble`, and `check` tasks.

To execute a task called `myTask` on the root project, type:

```
$ gradle :myTask
```

This will run the single `myTask` and all of its [dependencies](#).

Specify options for tasks

To pass an option to a task, prefix the option name with `--` after the task name:

```
$ gradle exampleTask --exampleOption=exampleValue
```

Disambiguate task options from built-in options

Gradle does not prevent tasks from registering options that conflict with Gradle's built-in options, like `--profile` or `--help`.

You can fix conflicting task options from Gradle's built-in options with a `--` delimiter before the task name in the command:

```
$ gradle [--built-in-option-name...] -- [taskName...] [--task-option-name...]
```

Consider a task named `mytask` that accepts an option named `profile`:

- In `gradle mytask --profile`, Gradle accepts `--profile` as the built-in Gradle option.
- In `gradle -- mytask --profile=value`, Gradle passes `--profile` as a task option.

Executing tasks in multi-project builds

In a [multi-project build](#), subproject tasks can be executed with `:` separating the subproject name and task name. The following are equivalent when *run from the root project*:

```
$ gradle :subproject:taskName
```

```
$ gradle subproject:taskName
```

You can also run a task for *all* subprojects using a task *selector* that consists of only the task name.

The following command runs the `test` task for all subprojects when invoked from the *root project directory*:

```
$ gradle test
```

NOTE

Some tasks selectors, like `help` or `dependencies`, will only run the task on the project they are invoked on and not on all the subprojects.

When invoking Gradle from within a subproject, the project name should be omitted:

```
$ cd subproject
```

```
$ gradle taskName
```

TIP

When executing the Gradle Wrapper from a subproject directory, reference `gradlew` relatively. For example: `../gradlew taskName`.

Executing multiple tasks

You can also specify multiple tasks. The tasks' dependencies determine the precise order of execution, and a task having no dependencies may execute earlier than it is listed on the command-line.

For example, the following will execute the `test` and `deploy` tasks in the order that they are listed on

the command-line and will also execute the dependencies for each task.

```
$ gradle test deploy
```

Command line order safety

Although Gradle will always attempt to execute the build quickly, command line ordering safety will also be honored.

For example, the following will execute **clean** and **build** along with their dependencies:

```
$ gradle clean build
```

However, the intention implied in the command line order is that **clean** should run first and then **build**. It would be incorrect to execute **clean** *after* **build**, even if doing so would cause the build to execute faster since **clean** would remove what **build** created.

Conversely, if the command line order was **build** followed by **clean**, it would not be correct to execute **clean** before **build**. Although Gradle will execute the build as quickly as possible, it will also respect the safety of the order of tasks specified on the command line and ensure that **clean** runs before **build** when specified in that order.

Note that **command line order safety** relies on tasks properly declaring what they create, consume, or remove.

Excluding tasks from execution

You can exclude a task from being executed using the **-x** or **--exclude-task** command-line option and providing the name of the task to exclude:

```
$ gradle dist --exclude-task test
```

```
> Task :compile
compiling source

> Task :dist
building the distribution

BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

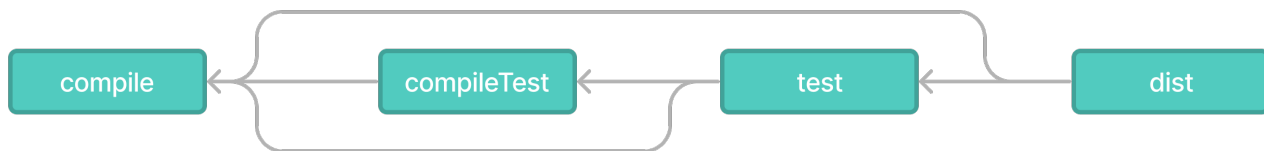


Figure 22. Simple Task Graph

You can see that the `test` task is not executed, even though the `dist` task depends on it. The `test` task's dependencies, such as `compileTest`, are not executed either. The dependencies of `test` that other tasks depend on, such as `compile`, are still executed.

Forcing tasks to execute

You can force Gradle to execute all tasks ignoring [up-to-date checks](#) using the `--rerun-tasks` option:

```
$ gradle test --rerun-tasks
```

This will force `test` and *all* task dependencies of `test` to execute. It is similar to running `gradle clean test`, but without the build's generated output being deleted.

Alternatively, you can tell Gradle to rerun a specific task using the `--rerun` built-in [task option](#).

Continue the build after a task failure

By default, Gradle aborts execution and fails the build when any task fails. This allows the build to complete sooner and prevents cascading failures from obfuscating the root cause of an error.

You can use the `--continue` option to force Gradle to execute every task when a failure occurs:

```
$ gradle test --continue
```

When executed with `--continue`, Gradle executes *every* task in the build if all the dependencies for that task are completed without failure.

For example, tests do not run if there is a compilation error in the code under test because the `test` task depends on the `compilation` task. Gradle outputs each of the encountered failures at the end of the build.

NOTE

If any tests fail, many test suites fail the entire `test` task. Code coverage and reporting tools frequently run after the test task, so "fail fast" behavior may halt execution before those tools run.

Name abbreviation

When you specify tasks on the command-line, you don't have to provide the full name of the task. You can provide enough of the task name to identify the task uniquely. For example, it is likely `gradle che` is enough for Gradle to identify the `check` task.

The same applies to project names. You can execute the `check` task in the `library` subproject with the `gradle lib:che` command.

You can use `camel case` patterns for more complex abbreviations. These patterns are expanded to match camel case and `kebab case` names. For example, the pattern `foBa` (or `fB`) matches `fooBar` and `foo-bar`.

More concretely, you can run the `compileTest` task in the `my-awesome-library` subproject with the command `gradle mAL:cT`.

```
$ gradle mAL:cT
```

```
> Task :my-awesome-library:compileTest
compiling unit tests

BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Abbreviations can also be used with the `-x` command-line option.

Tracing name expansion

For complex projects, it might be ambiguous if the intended tasks were executed. When using abbreviated names, a single typo can lead to the execution of unexpected tasks.

When `INFO`, or more `verbose logging` is enabled, the output will contain extra information about the project and task name expansion.

For example, when executing the `mAL:cT` command on the previous example, the following log messages will be visible:

```
No exact project with name ':mAL' has been found. Checking for abbreviated names.
Found exactly one project that matches the abbreviated name ':mAL': ':my-awesome-library'.
No exact task with name ':cT' has been found. Checking for abbreviated names.
Found exactly one task name, that matches the abbreviated name ':cT': ':compileTest'.
```

Common tasks

The following are task conventions applied by built-in and most major Gradle plugins.

Computing all outputs

It is common in Gradle builds for the `build` task to designate assembling all outputs and running all checks:

```
$ gradle build
```

Running applications

It is common for applications to run with the `run` task, which assembles the application and executes some script or binary:

```
$ gradle run
```

Running all checks

It is common for *all* verification tasks, including tests and linting, to be executed using the `check` task:

```
$ gradle check
```

Cleaning outputs

You can delete the contents of the build directory using the `clean` task. Doing so will cause pre-computed outputs to be lost, causing significant additional build time for the subsequent task execution:

```
$ gradle clean
```

Project reporting

Gradle provides several built-in tasks which show particular details of your build. This can be useful for understanding your build's structure and dependencies, as well as debugging problems.

Listing projects

Running the `projects` task gives you a list of the subprojects of the selected project, displayed in a hierarchy:

```
$ gradle projects
```

You also get a project report within [Build Scans](#).

Listing tasks

Running `gradle tasks` gives you a list of the main tasks of the selected project. This report shows the default tasks for the project, if any, and a description for each task:

```
$ gradle tasks
```

By default, this report shows only those tasks assigned to a task group.

Groups (such as verification, publishing, help, build...) are available as the header of each section when listing tasks:

```
> Task :tasks

Build tasks
-----
assemble - Assembles the outputs of this project.

Build Setup tasks
-----
init - Initializes a new Gradle build.

Distribution tasks
-----
assembleDist - Assembles the main distributions

Documentation tasks
-----
javadoc - Generates Javadoc API documentation for the main source code.
```

You can obtain more information in the task listing using the `--all` option:

```
$ gradle tasks --all
```

The option `--no-all` can limit the report to tasks assigned to a task group.

If you need to be more precise, you can display only the tasks from a specific group using the `--group` option:

```
$ gradle tasks --group="build setup"
```

Show task usage details

Running `gradle help --task someTask` gives you detailed information about a specific task:

```
$ gradle -q help --task libs
```

```
Detailed task information for libs
```

Paths

:api:libs
:webapp:libs

Type

Task (org.gradle.api.Task)

Options

--rerun Causes the task to be re-run even if up-to-date.

Description

Builds the JAR

Group

build

This information includes the full task path, the task type, possible [task-specific command line options](#), and the description of the given task.

You can get detailed information about the task class types using the `--types` option or using `--no-types` to hide this information.

Reporting dependencies

[Build Scans](#) give a full, visual report of what dependencies exist on which configurations, transitive dependencies, and dependency version selection. They can be invoked using the `--scan` options:

```
$ gradle myTask --scan
```

This will give you a link to a web-based report, where you can find [dependency information](#) like this:

Listing project dependencies

Running the **dependencies** task gives you a list of the dependencies of the selected project, broken down by configuration. For each configuration, the direct and transitive dependencies of that configuration are shown in a tree.

Below is an example of this report:

```
$ gradle dependencies
```

```
> Task :app:dependencies
```

```
-----
Project ':app'
-----
```

```
compileClasspath - Compile classpath for source set 'main'.
```

```
+--- project :model
```

```
|    \--- org.json:json:20220924
```

```
+--- com.google.inject:guice:5.1.0
```

```
|    +--- javax.inject:javax.inject:1
```

```
|    +--- aopalliance:aopalliance:1.0
```

```
|    \--- com.google.guava:guava:30.1-jre -> 28.2-jre
```

```
|         +--- com.google.guava:failureaccess:1.0.1
```

```
|         +--- com.google.guava:listenablefuture:9999.0-empty-to-avoid-conflict-with-guava
```

```
|         +--- com.google.code.findbugs:jsr305:3.0.2
```

```
|         +--- org.checkerframework:checker-qual:2.10.0 -> 3.28.0
```

```
|         +--- com.google.errorprone:error_prone_annotations:2.3.4
```

```
|         \--- com.google.j2objc:j2objc-annotations:1.3
```

```
+--- com.google.inject:guice:{strictly 5.1.0} -> 5.1.0 (c)
+--- org.json:json:{strictly 20220924} -> 20220924 (c)
+--- javax.inject:javax.inject:{strictly 1} -> 1 (c)
+--- aopalliance:aopalliance:{strictly 1.0} -> 1.0 (c)
+--- com.google.guava:guava:{strictly [28.0-jre, 28.5-jre]} -> 28.2-jre (c)
+--- com.google.guava:guava:{strictly 28.2-jre} -> 28.2-jre (c)
+--- com.google.guava:failureaccess:{strictly 1.0.1} -> 1.0.1 (c)
+--- com.google.guava:listenablefuture:{strictly 9999.0-empty-to-avoid-conflict-with-guava} -> 9999.0-empty-to-avoid-conflict-with-guava (c)
+--- com.google.code.findbugs:jsr305:{strictly 3.0.2} -> 3.0.2 (c)
+--- org.checkerframework:checker-qual:{strictly 3.28.0} -> 3.28.0 (c)
+--- com.google.errorprone:error_prone_annotations:{strictly 2.3.4} -> 2.3.4 (c)
\--- com.google.j2objc:j2objc-annotations:{strictly 1.3} -> 1.3 (c)
```

Concrete examples of build scripts and output available in [Viewing and debugging dependencies](#).

Running the `buildEnvironment` task visualises the buildscript dependencies of the selected project, similarly to how `gradle dependencies` visualizes the dependencies of the software being built:

```
$ gradle buildEnvironment
```

Running the `dependencyInsight` task gives you an insight into a particular dependency (or dependencies) that match specified input:

```
$ gradle dependencyInsight --dependency [...] --configuration [...]
```

The `--configuration` parameter restricts the report to a particular configuration such as `compileClasspath`.

Listing project properties

Running the `properties` task gives you a list of the properties of the selected project:

```
$ gradle -q api:properties
```

```
-----
Project ':api' - The shared API for the application
-----
```

```
allprojects: [project ':api']
ant: org.gradle.api.internal.project.DefaultAntBuilder@12345
antBuilderFactory: org.gradle.api.internal.project.DefaultAntBuilderFactory@12345
artifacts:
org.gradle.api.internal.artifacts.dsl.DefaultArtifactHandler_Decorated@12345
asDynamicObject: DynamicObject for project ':api'
baseClassLoaderScope:
```

```
org.gradle.api.internal.initialization.DefaultClassLoaderScope@12345
```

You can also query a single property with the optional `--property` argument:

```
$ gradle -q api:properties --property allprojects
```

```
-----  
Project ':api' - The shared API for the application  
-----
```

```
allprojects: [project ':api']
```

Command-line completion

Gradle provides `bash` and `zsh` tab completion support for tasks, options, and Gradle properties through [gradle-completion](#) (installed separately):

[gradle completion 4.0] | *gradle-completion-4.0.gif*

Debugging options

`-?, -h, --help`

Shows a help message with the built-in CLI options. To show project-contextual options, including help on a specific task, see the `help` task.

`-v, --version`

Prints Gradle, Groovy, Ant, Launcher & Daemon JVM, and operating system version information and exit without executing any tasks.

`-V, --show-version`

Prints Gradle, Groovy, Ant, Launcher & Daemon JVM, and operating system version information and continue execution of specified tasks.

`-S, --full-stacktrace`

Print out the full (very verbose) stacktrace for any exceptions. See also [logging options](#).

`-s, --stacktrace`

Print out the stacktrace also for user exceptions (e.g. compile error). See also [logging options](#).

`--scan`

Create a [Build Scan](#) with fine-grained information about all aspects of your Gradle build.

`-Dorg.gradle.debug=true`

A [Gradle property](#) that debugs the [Gradle Daemon](#) process. Gradle will wait for you to attach a debugger at `localhost:5005` by default.

-Dorg.gradle.debug.host=(host address)

A [Gradle property](#) that specifies the host address to listen on or connect to when debug is enabled. In the server mode on Java 9 and above, passing `*` for the host will make the server listen on all network interfaces. By default, no host address is passed to JDWP, so on Java 9 and above, the loopback address is used, while earlier versions listen on all interfaces.

-Dorg.gradle.debug.port=(port number)

A [Gradle property](#) that specifies the port number to listen on when debug is enabled. *Default is 5005.*

-Dorg.gradle.debug.server=(true,false)

A [Gradle property](#) that if set to `true` and debugging is enabled, will cause Gradle to run the build with the socket-attach mode of the debugger. Otherwise, the socket-listen mode is used. *Default is true.*

-Dorg.gradle.debug.suspend=(true,false)

A [Gradle property](#) that if set to `true` and debugging is enabled, the JVM running Gradle will suspend until a debugger is attached. *Default is true.*

-Dorg.gradle.daemon.debug=true

A [Gradle property](#) that debugs the [Gradle Daemon](#) process. (duplicate of `-Dorg.gradle.debug`)

Performance options

Try these options when optimizing and [improving](#) build performance.

Many of these options can be [specified](#) in the `gradle.properties` file, so command-line flags are unnecessary.

--build-cache, --no-build-cache

Toggles the [Gradle Build Cache](#). Gradle will try to reuse outputs from previous builds. *Default is off.*

--configuration-cache, --no-configuration-cache

Toggles the [Configuration Cache](#). Gradle will try to reuse the build configuration from previous builds. *Default is off.*

--configuration-cache-problems=(fail,warn)

Configures how the configuration cache handles problems. Default is `fail`.

Set to `warn` to report problems without failing the build.

Set to `fail` to report problems and fail the build if there are any problems.

--configure-on-demand, --no-configure-on-demand

Toggles configure-on-demand. Only relevant projects are configured in this build run. *Default is off.*

--max-workers

Sets the maximum number of workers that Gradle may use. *Default is number of processors.*

--parallel, --no-parallel

Build projects in parallel. For limitations of this option, see [Parallel Project Execution](#). *Default is off.*

--priority

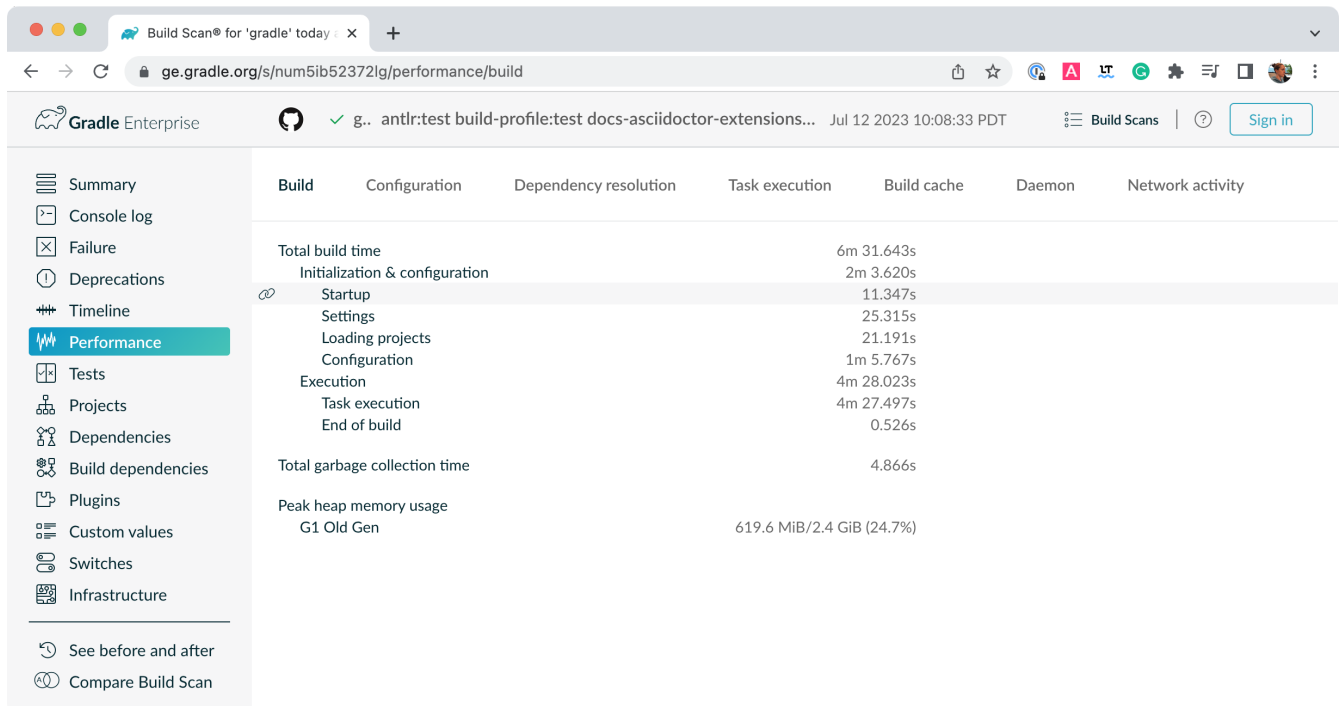
Specifies the scheduling priority for the Gradle daemon and all processes launched by it. Values are **normal** or **low**. *Default is normal.*

--profile

Generates a high-level performance report in the `layout.buildDirectory.dir("reports/profile")` directory. **--scan** is preferred.

--scan

Generate a build scan with detailed performance diagnostics.



--watch-fs, --no-watch-fs

Toggles [watching the file system](#). When enabled, Gradle reuses information it collects about the file system between builds. *Enabled by default on operating systems where Gradle supports this feature.*

Gradle daemon options

You can manage the [Gradle Daemon](#) through the following command line options.

--daemon, --no-daemon

Use the [Gradle Daemon](#) to run the build. Starts the daemon if not running or the existing daemon is busy. *Default is on.*

--foreground

Starts the Gradle Daemon in a foreground process.

--status (Standalone command)

Run `gradle --status` to list running and recently stopped Gradle daemons. It only displays daemons of the same Gradle version.

--stop (Standalone command)

Run `gradle --stop` to stop all Gradle Daemons of the same version.

-Dorg.gradle.daemon.idletimeout=(number of milliseconds)

A [Gradle property](#) wherein the Gradle Daemon will stop itself after this number of milliseconds of idle time. *Default is 10800000 (3 hours).*

Logging options

Setting log level

You can customize the [verbosity](#) of Gradle logging with the following options, ordered from least verbose to most verbose.

-Dorg.gradle.logging.level=(quiet,warn,lifecycle,info,debug)

A [Gradle property](#) that sets the logging level.

-q, --quiet

Log errors only.

-w, --warn

Set log level to warn.

-i, --info

Set log level to info.

-d, --debug

Log in debug mode (includes normal stacktrace).

Lifecycle is the default log level.

Customizing log format

You can control the use of rich output (colors and font variants) by specifying the console mode in the following ways:

-Dorg.gradle.console=(auto,plain,rich,verbose)

A [Gradle property](#) that specifies the console mode. Different modes are described immediately below.

--console=(auto,plain,rich,verbose)

Specifies which type of console output to generate.

Set to **plain** to generate plain text only. This option disables all color and other rich output in the console output. This is the default when Gradle is *not* attached to a terminal.

Set to **auto** (the default) to enable color and other rich output in the console output when the build process is attached to a console or to generate plain text only when not attached to a console. *This is the default when Gradle is attached to a terminal.*

Set to **rich** to enable color and other rich output in the console output, regardless of whether the build process is not attached to a console. When not attached to a console, the build output will use ANSI control characters to generate the rich output.

Set to **verbose** to enable color and other rich output like **rich** with output task names and outcomes at the lifecycle log level, (as is done by default in Gradle 3.5 and earlier).

Reporting problems

--no-problems-report

Disable the generation of **build/reports/problems-report.html**, by default this report is generated with problems provided to the [Problems API](#).

--problems-report

Enable the generation of **build/reports/problems-report.html**. This is the default behaviour. The report is generated with problems provided to the [Problems API](#).

Showing or hiding warnings

By default, Gradle won't display all warnings (e.g. deprecation warnings). Instead, Gradle will collect them and render a summary at the end of the build like:

```
Deprecated Gradle features were used in this build, making it incompatible with Gradle 5.0.
```

You can control the verbosity of warnings on the console with the following options:

-Dorg.gradle.warning.mode=(all,fail,none,summary)

A [Gradle property](#) that specifies the warning mode. Different modes are described immediately below.

--warning-mode=(all,fail,none,summary)

Specifies how to log warnings. Default is **summary**.

Set to **all** to log all warnings.

Set to **fail** to log all warnings and fail the build if there are any warnings.

Set to **summary** to suppress all warnings and log a summary at the end of the build.

Set to **none** to suppress all warnings, including the summary at the end of the build.

Rich console

Gradle's rich console displays extra information while builds are running.

```
solution — java -Xmx64m -Xms64m -Dorg.gradle.appname=gradlew -classpath ~/Documents/Training/training/build-tool-training-exercises/Jv...
> Task :groovy:processTestResources NO-SOURCE
> Task :scala:processTestResources NO-SOURCE
> Task :java:processTestResources NO-SOURCE

> Task :guava-old-version:compileJava
Note: /Users/lkassovic/Documents/Training/training/build-tool-training-exercises/Jvm_Builds_with_Gradle_Build_Tool/exercise4/solution/guava-old-version/src/main/java/com/gradle/lab/old/OldMessage.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

> Task :guava-old-version:processResources NO-SOURCE
> Task :guava-old-version:classes
> Task :guava-old-version:shadowJar
> Task :guava-old-version:jar
> Task :guava-old-version:assemble
> Task :guava-old-version:compileTestJava NO-SOURCE
> Task :guava-old-version:processTestResources NO-SOURCE
> Task :guava-old-version:testClasses UP-TO-DATE
> Task :guava-old-version:test NO-SOURCE
> Task :guava-old-version:check UP-TO-DATE
> Task :guava-old-version:build
<===-----> 23% EXECUTING [5s]
> IDLE
> IDLE
> :scala:compileJava
> :kotlin:compileKotlin > Resolve files of :kotlin:kotlinCompilerClasspath > kotlin-reflect-1.7.10.jar
> IDLE
> IDLE
> IDLE
> IDLE
> :groovy:compileJava
> IDLE
> :java:compileJava
> IDLE
```

Features:

- Progress bar and timer visually describe the overall status
- Parallel work-in-progress lines below describe what is happening now
- Colors and fonts are used to highlight significant output and errors

Execution options

The following options affect how builds are executed by changing what is built or how dependencies are resolved.

--include-build

Run the build as a [composite](#), including the specified build.

--offline

Specifies that the build should operate [without accessing network resources](#).

-U, --refresh-dependencies

Refresh the [state of dependencies](#).

--continue

[Continue task execution](#) after a task failure.

-m, --dry-run

Run Gradle with all task actions disabled. Use this to show which task would have executed.

-t, --continuous

Enables [continuous build](#). Gradle does not exit and will re-execute tasks when task file inputs

change.

--write-locks

Indicates that all resolved configurations that are *lockable* should have their [lock state](#) persisted.

--update-locks <group:name>[,<group:name>]*

Indicates that versions for the specified modules have to be updated in the [lock file](#).

This flag also implies **--write-locks**.

-a, --no-rebuild

Do not rebuild project dependencies. Useful for [debugging and fine-tuning buildSrc](#), but can lead to wrong results. Use with caution!

Dependency verification options

Learn more about this in [dependency verification](#).

-F=(strict, lenient, off), --dependency-verification=(strict, lenient, off)

Configures the [dependency verification mode](#).

The default mode is **strict**.

-M, --write-verification-metadata

Generates checksums for dependencies used in the project (comma-separated list) for [dependency verification](#).

--refresh-keys

Refresh the public keys used for dependency verification.

--export-keys

Exports the public keys used for dependency verification.

Environment options

You can [customize](#) many aspects of build scripts, settings, caches, and so on through the options below.

-b, --build-file (deprecated)

Specifies the build file. For example: **gradle --build-file=foo.gradle**. The default is **build.gradle**, then **build.gradle.kts**.

-c, --settings-file (deprecated)

Specifies the settings file. For example: **gradle --settings-file=somewhere/else/settings.gradle**

-g, --gradle-user-home

Specifies the Gradle User Home directory. The default is the **.gradle** directory in the user's home directory.

-p, --project-dir

Specifies the start directory for Gradle. Defaults to current directory.

--project-cache-dir

Specifies the project-specific cache directory. Default value is `.gradle` in the root project directory.

-D, --system-prop

Sets a [system property](#) of the JVM, for example `-Dmyprop=myvalue`.

-I, --init-script

Specifies an [initialization script](#).

-P, --project-prop

Sets a [project property](#) of the root project, for example `-Pmyprop=myvalue`.

-Dorg.gradle.jvmargs

A [Gradle property](#) that sets JVM arguments.

-Dorg.gradle.java.home

A [Gradle property](#) that sets the JDK home dir.

Task options

Tasks may define task-specific options which are different from most of the global options described in the sections above (which are interpreted by Gradle itself, can appear anywhere in the command line, and can be listed using the `--help` option).

Task options:

1. Are consumed and interpreted by the tasks themselves;
2. **Must** be specified immediately after the task in the command-line;
3. May be listed using `gradle help --task someTask` (see [Show task usage details](#)).

To learn how to declare command-line options for your own tasks, see [Declaring and Using Command Line Options](#).

Built-in task options

Built-in task options are options available as task options for all tasks. At this time, the following built-in task options exist:

--rerun

Causes the task to be rerun even if up-to-date. Similar to `--rerun-tasks`, but for a specific task.

Bootstrapping new projects

Creating new Gradle builds

Use the built-in `gradle init` task to create a new Gradle build, with new or existing projects.

```
$ gradle init
```

Most of the time, a project type is specified. Available types include `basic` (default), `java-library`, `java-application`, and more. See [init plugin documentation](#) for details.

```
$ gradle init --type java-library
```

Standardize and provision Gradle

The built-in `gradle wrapper` task generates a script, `gradlew`, that invokes a declared version of Gradle, downloading it beforehand if necessary.

```
$ gradle wrapper --gradle-version=8.1
```

You can also specify `--distribution-type=(bin|all)`, `--gradle-distribution-url`, `--gradle-distribution-sha256-sum` in addition to `--gradle-version`.

Full details on using these options are documented in the [Gradle wrapper section](#).

Continuous build

Continuous Build allows you to automatically re-execute the requested tasks when file inputs change. You can execute the build in this mode using the `-t` or `--continuous` command-line option.

Learn more in [Continuous Builds](#).

GRADLE DSL/API

A Groovy Build Script Primer

Ideally, a Groovy build script looks mostly like configuration: setting some properties of the project, configuring dependencies, declaring tasks, and so on. That configuration is based on Groovy language constructs. This primer aims to explain what those constructs are and — most importantly — how they relate to Gradle's API documentation.

The **Project** object

As Groovy is an object-oriented language based on Java, its properties and methods apply to objects. In some cases, the object is implicit — particularly at the top level of a build script, i.e. not nested inside a `{}` block.

Consider this fragment of build script, which contains an unqualified property and block:

```
version = '1.0.0.GA'

configurations {
    ...
}
```

Both `version` and `configurations {}` are part of `org.gradle.api.Project`.

This example reflects how every Groovy build script is backed by an implicit instance of `Project`. If you see an unqualified element and you don't know where it's defined, always check the `Project` API documentation to see if that's where it's coming from.

CAUTION

Avoid using `Groovy MetaClass` programming techniques in your build scripts. Gradle provides its own API for adding `dynamic runtime properties`.

Use of Groovy-specific metaprogramming can cause builds to retain large amounts of memory between builds that will eventually cause the Gradle daemon to run out-of-memory.

Properties

```
<obj>.<name>           // Get a property value
<obj>.<name> = <value>   // Set a property to a new value
"$<name>"              // Embed a property value in a string
"${<obj>.<name>}"       // Same as previous (embedded value)
```

Examples

```
version = '1.0.1'
```

```
myCopyTask.description = 'Copies some files'

file("${projectDir/src}")
println "Destination: ${myCopyTask.destinationDir}"
```

A property represents some state of an object. The presence of an `=` sign is a clear indicator that you're looking at a property. Otherwise, a qualified name — it begins with `<obj>`. — without any other decoration is also a property.

If the name is unqualified, then it may be one of the following:

- A task instance with that name.
- A property on [Project](#).
- An [extra property](#) defined elsewhere in the project.
- A property of an implicit object within a [block](#).
- A [local variable](#) defined earlier in the build script.

Note that plugins can add their own properties to the [Project](#) object. The [API documentation](#) lists all the properties added by core plugins. If you're struggling to find where a property comes from, check the documentation for the plugins that the build uses.

TIP

When referencing a project property in your build script that is added by a non-core plugin, consider prefixing it with `project.` — it's clear then that the property belongs to the project object.

Properties in the API documentation

The [Groovy DSL reference](#) shows properties as they are used in your build scripts, but the Javadocs only display methods. That's because properties are implemented as methods behind the scenes:

- A property can be *read* if there is a method named `get<PropertyName>` with zero arguments that returns the same type as the property.
- A property can be *modified* if there is a method named `set<PropertyName>` with one argument that has the same type as the property and a return type of `void`.

Note that property names usually start with a lower-case letter, but that letter is upper case in the method names. So the getter method `getProjectVersion()` corresponds to the property `projectVersion`. This convention does not apply when the name begins with at least two upper-case letters, in which case there is not change in case. For example, `getRAM()` corresponds to the property `RAM`.

Examples

```
project.getVersion()
project.version

project.setVersion('1.0.1')
```

```
project.version = '1.0.1'
```

Methods

```
<obj>.<name>()           // Method call with no arguments
<obj>.<name>(<arg>, <arg>) // Method call with multiple arguments
<obj>.<name> <arg>, <arg>  // Method call with multiple args (no parentheses)
```

Examples

```
myCopyTask.include '**/*.xml', '**/*.properties'

ext.resourceSpec = copySpec() // `copySpec()` comes from `Project`

file('src/main/java')
println 'Hello, World!'
```

A method represents some behavior of an object, although Gradle often uses methods to configure the state of objects as well. Methods are identifiable by their arguments or empty parentheses. Note that parentheses are sometimes required, such as when a method has zero arguments, so you may find it simplest to always use parentheses.

NOTE

Gradle has a convention whereby if a method has the same name as a collection-based property, then the method *appends* its values to that collection.

Blocks

Blocks are also [methods](#), just with specific types for the last argument.

```
<obj>.<name> {
    ...
}

<obj>.<name>(<arg>, <arg>) {
    ...
}
```

Examples

```
plugins {
    id 'java-library'
}

configurations {
    assets
}
```

```

sourceSets {
    main {
        java {
            srcDirs = ['src']
        }
    }
}

dependencies {
    implementation project(':util')
}

```

Blocks are a mechanism for configuring multiple aspects of a build element in one go. They also provide a way to nest configuration, leading to a form of structured data.

There are two important aspects of blocks that you should understand:

1. They are implemented as methods with specific signatures.
2. They can change the target ("delegate") of unqualified methods and properties.

Both are based on Groovy language features and we explain them in the following sections.

Block method signatures

You can easily identify a method as the implementation behind a block by its signature, or more specifically, its argument types. If a method corresponds to a block:

- It must have at least one argument.
- The *last* argument must be of type `groovy.lang.Closure` or `org.gradle.api.Action`.

For example, `Project.copy(Action)` matches these requirements, so you can use the syntax:

```

copy {
    into layout.buildDirectory.dir("tmp")
    from 'custom-resources'
}

```

That leads to the question of how `into()` and `from()` work. They're clearly methods, but where would you find them in the API documentation? The answer comes from understanding object *delegation*.

Delegation

The [section on properties](#) lists where unqualified properties might be found. One common place is on the `Project` object. But there is an alternative source for those unqualified properties and methods inside a block: the block's *delegate object*.

To help explain this concept, consider the last example from the previous section:

```
copy {  
    into layout.buildDirectory.dir("tmp")  
    from 'custom-resources'  
}
```

All the methods and properties in this example are unqualified. You can easily find `copy()` and `layout` in the [Project API documentation](#), but what about `into()` and `from()`? These are resolved against the delegate of the `copy {}` block. What is the type of that delegate? You'll need to [check the API documentation for that](#).

There are two ways to determine the delegate type, depending on the signature of the block method:

- For **Action** arguments, look at the type's parameter.

In the example above, the method signature is `copy(Action<? super CopySpec>)` and it's the bit inside the angle brackets that tells you the delegate type — `CopySpec` in this case.

- For **Closure** arguments, the documentation will explicitly say in the description what type is being configured or what type the delegate is (different terminology for the same thing).

Hence you can find both `into()` and `from()` on `CopySpec`. You might even notice that both of those methods have variants that take an **Action** as their last argument, which means you can use block syntax with them.

All new Gradle APIs declare an **Action** argument type rather than **Closure**, which makes it very easy to pick out the delegate type. Even older APIs have an **Action** variant in addition to the old **Closure** one.

Local variables

```
def <name> = <value>           // Untyped variable  
<type> <name> = <value>       // Typed variable
```

Examples

```
def i = 1  
String errorMsg = 'Failed, because reasons'
```

Local variables are a Groovy construct — unlike [extra properties](#) — that can be used to share values within a build script.

CAUTION

Avoid using local variables in the root of the project, i.e. as pseudo project properties. They cannot be read outside of the build script and Gradle has no knowledge of them.

Within a narrower context — such as configuring a task — local variables can

occasionally be helpful.

Gradle Kotlin DSL Primer

Gradle's Kotlin DSL offers an alternative to the traditional Groovy DSL, delivering an enhanced editing experience in supported IDEs with features like better content assist, refactoring, and documentation.

This chapter explores the key Kotlin DSL constructs and demonstrates how to use them to interact with the Gradle API.

TIP

If you are interested in migrating an existing Gradle build to the Kotlin DSL, please also check out the dedicated [migration page](#).

Prerequisites

- The embedded Kotlin compiler works on Linux, macOS, Windows, Cygwin, FreeBSD, and Solaris on x86-64 architectures.
- Familiarity with Kotlin syntax and basic language features is recommended. Refer to the [Kotlin documentation](#) and [Kotlin Koans](#) to learn the basics.
- Using the `plugins {}` block to declare Gradle plugins is highly recommended as it significantly improves the editing experience.

IDE support

The Kotlin DSL is fully supported by IntelliJ IDEA and Android Studio. While other IDEs lack advanced tools for editing Kotlin DSL files, you can still import Kotlin-DSL-based builds and work with them as usual.

	Build import	Syntax highlighting ¹	Semantic editor ²
IntelliJ IDEA	☑	☑	☑
Android Studio	☑	☑	☑
Eclipse IDE	☑	☑	☑
CLion	☑	☑	☑
Apache NetBeans	☑	☑	☑
Visual Studio Code ^(LSP)	☑	☑	☑
Visual Studio	☑	☑	☑

¹ Kotlin syntax highlighting in Gradle Kotlin DSL scripts

² Code completion, navigation to sources, documentation, refactorings etc... in Gradle Kotlin DSL scripts

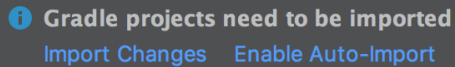
As noted in the limitations, you must [import your project using the Gradle model](#) to enable content assist and refactoring tools for Kotlin DSL scripts in IntelliJ IDEA.

Builds with slow configuration time might affect the IDE responsiveness, so please check out the [performance section](#) to help resolve such issues.

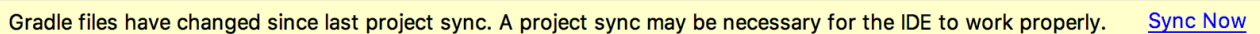
Automatic build import vs. automatic reloading of script dependencies

Both IntelliJ IDEA and Android Studio will detect when you make changes to your build logic and offer two suggestions:

1. Import the whole build again:




Gradle projects need to be imported
[Import Changes](#) [Enable Auto-Import](#)



Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly. [Sync Now](#)

2. Reload script dependencies when editing a build script:



There are new script dependencies available. [Apply dependencies](#) [Enable auto-reload](#)

We recommend *disabling automatic build import* while *enabling automatic reloading of script dependencies*. This approach provides early feedback when editing Gradle scripts while giving you control over when the entire build setup synchronizes with your IDE.

See the [Troubleshooting](#) section to learn more.

Kotlin DSL scripts

Just like its Groovy-based counterpart, the Kotlin DSL is built on Gradle's Java API. Everything in a Kotlin DSL script is Kotlin code, compiled and executed by Gradle. Many of the objects, functions, and properties in your build scripts come from the Gradle API and the APIs of applied plugins.

TIP Use the [Kotlin DSL reference](#) search to explore available members.

Script file names

- Groovy DSL script files use the `.gradle` file name extension.
- Kotlin DSL script files use the `.gradle.kts` file name extension.

To activate the Kotlin DSL, use the `.gradle.kts` extension for your build scripts instead of `.gradle`. This also applies to the [settings file](#) (e.g., `settings.gradle.kts`) and [initialization scripts](#).

You can mix Groovy DSL and Kotlin DSL scripts within the same build. For example, a Kotlin DSL build script can apply a Groovy DSL one, and different projects in a multi-project build can use either.

To improve IDE support, we recommend following these conventions:

- Name settings scripts (or any script backed by a Gradle `Settings` object) using the pattern `*.settings.gradle.kts`. This includes script plugins applied from settings scripts.

- Name [initialization scripts](#) using the pattern `*.init.gradle.kts` or simply `init.gradle.kts`.

This helps the IDE identify the object "backing" the script, whether it's [Project](#), [Settings](#), or [Gradle](#).

Implicit imports

All Kotlin DSL build scripts come with implicit imports, including:

- The [default Gradle API imports](#)
- The Kotlin DSL API, which includes types from the following packages:
 - `org.gradle.kotlin.dsl`
 - `org.gradle.kotlin.dsl.plugins.dsl`
 - `org.gradle.kotlin.dsl.precompile`

Avoid Using Internal Kotlin DSL APIs

Using internal Kotlin DSL APIs in plugins and build scripts can break builds when either Gradle or plugins are updated.

The [Kotlin DSL API](#) extends the public Gradle API with types listed in the [corresponding API docs](#) found in the packages above (but not in their subpackages).

Compilation warnings

Gradle Kotlin DSL scripts are compiled by Gradle during the [configuration phase](#) of your build.

Deprecation warnings found by the Kotlin compiler are reported on the console when compiling the scripts:

```
> Configure project :  
w: build.gradle.kts:4:5: 'getter for uploadTaskName: String!' is deprecated.  
Deprecated in Java
```

It is possible to configure your build to fail on any warning emitted during script compilation by [setting](#) the `org.gradle.kotlin.dsl.allWarningsAsErrors` Gradle property to `true`:

gradle.properties

```
org.gradle.kotlin.dsl.allWarningsAsErrors=true
```

Type-safe model accessors

The Groovy DSL allows you to reference many build model elements by name, even if they are defined at runtime, such as named configurations or source sets.

For example, when the `Java` plugin is applied, you can access the `implementation` configuration via `configurations.implementation`.

The Kotlin DSL replaces this dynamic resolution with type-safe model accessors, which work with model elements contributed by plugins.

Understanding when type-safe model accessors are available

The Kotlin DSL currently provides various sets of type-safe model accessors, each tailored to different scopes.

For the main project build scripts and precompiled project script plugins:

Type-safe model accessors	Example
Dependency and artifact configurations	<code>implementation</code> and <code>runtimeOnly</code> (contributed by the Java Plugin)
Project extensions and conventions, and extensions on them	<code>sourceSets</code>
Extensions on the <code>dependencies</code> and <code>repositories</code> containers, and extensions on them	<code>testImplementation</code> (contributed by the Java Plugin), <code>mavenCentral</code>
Elements in the <code>tasks</code> and <code>configurations</code> containers	<code>compileJava</code> (contributed by the Java Plugin), <code>test</code>
Elements in <code>project-extension containers</code>	Source sets contributed by the Java Plugin that are added to the <code>sourceSets</code> container: <code>sourceSets.main.java { setSrcDirs(listOf("src/main/java")) }</code>

For the main project settings script and precompiled settings script plugins:

Type-safe model accessors	Example
Project extensions and conventions, contributed by <code>Settings</code> plugins, and extensions on them	<code>pluginManagement</code> , <code>dependencyResolutionManagement</code>

IMPORTANT

Initialization scripts and script plugins do not have type-safe model accessors. These limitations will be removed in a future Gradle release.

The set of type-safe model accessors available is determined right before evaluating the script body, immediately after the `plugins {}` block. Model elements contributed after that point, such as configurations defined in your build script, **will not work** with type-safe model accessors:

build.gradle.kts

```
// Applies the Java plugin
plugins {
    id("java")
}

repositories {
    mavenCentral()
```

```

}

// Access to 'implementation' (contributed by the Java plugin) works here:
dependencies {
    implementation("org.apache.commons:commons-lang3:3.12.0")
    testImplementation("org.junit.jupiter:junit-jupiter:5.10.0")
    testRuntimeOnly("org.junit.platform:junit-platform-launcher") // Add this
    if needed for runtime
}

// Add a custom configuration
configurations.create("customConfiguration")
// Type-safe accessors for 'customConfiguration' will NOT be available
because it was created after the plugins block
dependencies {
    customConfiguration("com.google.guava:guava:32.1.2-jre") // ❌ Error: No
    type-safe accessor for 'customConfiguration'
}

```

However, this means you can use type-safe accessors for any model elements contributed by plugins that are *applied by parent projects*.

The following project build script demonstrates how you can access various configurations, extensions and other elements using type-safe accessors:

build.gradle.kts

```

plugins {
    `java-library`
}

dependencies {
    api("junit:junit:4.13")
    implementation("junit:junit:4.13")
    testImplementation("junit:junit:4.13")
}

configurations {
    implementation {
        resolutionStrategy.failOnVersionConflict()
    }
}

sourceSets {
    main {
        java.srcDir("src/core/java")
    }
}

```

```

}

java {
    sourceCompatibility = JavaVersion.VERSION_11
    targetCompatibility = JavaVersion.VERSION_11
}

tasks {
    test {
        testLogging.showExceptions = true
        useJUnit()
    }
}

```

- ① Uses type-safe accessors for the `api`, `implementation` and `testImplementation` dependency configurations contributed by the [Java Library Plugin](#)
- ② Uses an accessor to configure the `sourceSets` project extension
- ③ Uses an accessor to configure the `main` source set
- ④ Uses an accessor to configure the `java` source for the `main` source set
- ⑤ Uses an accessor to configure the `test` task

TIP

Your IDE is aware of the type-safe accessors and will include them in its suggestions.

This applies both at the top level of your build scripts, where most plugin extensions are added to the `Project` object, and within the blocks that configure an extension.

Note that accessors for elements of containers such as `configurations`, `tasks`, and `sourceSets` leverage Gradle's [configuration avoidance APIs](#). For example, on `tasks`, accessors are of type `TaskProvider<T>` and provide a lazy reference and lazy configuration of the underlying task.

Here are some examples illustrating when configuration avoidance applies:

build.gradle.kts

```

tasks.test {
    // lazy configuration
    useJUnitPlatform()
}

// Lazy reference
val testProvider: TaskProvider<Test> = tasks.test

testProvider {
    // lazy configuration
}

```

```
// Eagerly realized Test task, defeats configuration avoidance if done out of  
a lazy context  
val test: Test = tasks.test.get()
```

For all other containers, accessors for elements are of type `NamedDomainObjectProvider<T>`, providing the same behavior:

build.gradle.kts

```
val mainSourceSetProvider: NamedDomainObjectProvider<SourceSet> =  
sourceSets.named("main")
```

Understanding what to do when type-safe model accessors are not available

Consider the sample build script shown above, which demonstrates the use of type-safe accessors. The following sample is identical, except it uses the `apply()` method to apply the plugin.

In this case, the build script cannot use type-safe accessors because the `apply()` call occurs in the body of the build script. You must use another techniques instead, as demonstrated here:

build.gradle.kts

```
apply(plugin = "java-library")  
  
dependencies {  
    "api"("junit:junit:4.13")  
    "implementation"("junit:junit:4.13")  
    "testImplementation"("junit:junit:4.13")  
}  
  
configurations {  
    "implementation" {  
        resolutionStrategy.failOnVersionConflict()  
    }  
}  
  
configure<SourceSetContainer> {  
    named("main") {  
        java.srcDir("src/core/java")  
    }  
}
```

```

configure<JavaPluginExtension> {
    sourceCompatibility = JavaVersion.VERSION_11
    targetCompatibility = JavaVersion.VERSION_11
}

tasks {
    named<Test>("test") {
        testLogging.showExceptions = true
    }
}

```

Type-safe accessors are unavailable for model elements contributed by the following:

- Plugins applied via the `apply(plugin = "id")` method.
- The project build script.
- Script plugins, via `apply(from = "script-plugin.gradle.kts")`.
- Plugins applied via [cross-project configuration](#).

You cannot use type-safe accessors in [binary Gradle plugins](#) implemented in Kotlin.

If you can't find a type-safe accessor, *fall back to using the normal API* for the corresponding types. To do so, you need to know the names and/or types of the configured model elements. We will now show you how these can be discovered by examining the script in detail.

Artifact configurations

The following sample demonstrates how to reference and configure artifact configurations without type-safe accessors:

build.gradle.kts

```

apply(plugin = "java-library")

dependencies {
    "api"("junit:junit:4.13")
    "implementation"("junit:junit:4.13")
    "testImplementation"("junit:junit:4.13")
}

configurations {
    "implementation" {
        resolutionStrategy.failOnVersionConflict()
    }
}

```

The code looks similar to that of the type-safe accessors, except that the configuration names are string literals. You can use string literals for configuration names in dependency declarations and within the `configurations {}` block.

While the IDE won't be able to help you discover the available configurations, you can look them up either in the corresponding plugin's documentation or by running `./gradlew dependencies`.

Project extensions and conventions

Project extensions and `conventions` have both a name and a unique type. However, the Kotlin DSL only needs to know the type to configure them.

The following sample shows the `sourceSets {}` and `java {}` blocks from the original example build script. The `configure<T>()` function is used with the corresponding type:

build.gradle.kts

```
apply(plugin = "java-library")

configure<SourceSetContainer> {
    named("main") {
        java.srcDir("src/core/java")
    }
}

configure<JavaPluginExtension> {
    sourceCompatibility = JavaVersion.VERSION_11
    targetCompatibility = JavaVersion.VERSION_11
}
```

Note that `sourceSets` is a Gradle extension on `Project` of type `SourceSetContainer` and `java` is an extension on `Project` of type `JavaPluginExtension`.

You can discover available extensions and conventions by either reviewing the documentation for the applied plugins or running `./gradlew kotlinDslAccessorsReport`. The report generates the Kotlin code needed to access the model elements contributed by the applied plugins, providing both names and types.

As a last resort, you can check the plugin's source code, though this should not be necessary in most cases.

You can also use the `the<T>()` function if you only need a reference to the extension or convention without configuring it, or if you want to perform a one-line configuration:

build.gradle.kts

```
the<SourceSetContainer>()["main"].java.srcDir("src/main/java")
```

The snippet above also demonstrates one way to configure elements of a project extension that is a container.

Elements in project-extension containers

Container-based project extensions, such as `SourceSetContainer`, allow you to configure the elements they hold.

In our sample build script, we want to configure a source set named `main` within the source set container. We can do this by using the `named()` method instead of an accessor:

build.gradle.kts

```
apply(plugin = "java-library")

configure<SourceSetContainer> {
    named("main") {
        java.srcDir("src/core/java")
    }
}
```

All elements within a container-based project extension have a name, so you can use this technique in all such cases.

For project extensions and conventions, you can discover what elements are present in any container by either checking the documentation for the applied plugins or by running `./gradlew kotlinDslAccessorsReport`.

As a last resort, you may also review the plugin's source code to find out what it does.

Tasks

Tasks are not managed through a container-based project extension, but they are part of a container that behaves in a similar way.

This means that you can configure tasks in the same way as you do for source sets. The following example illustrates this approach:

build.gradle.kts

```
apply(plugin = "java-library")

tasks {
    named<Test>("test") {
        testLogging.showExceptions = true
    }
}
```

We are using the Gradle API to refer to tasks by name and type, rather than using accessors.

Note that it is necessary to specify the type of the task explicitly. If you don't, the script won't compile because the inferred type will be `Task`, not `Test`, and the `testLogging` property is specific to the `Test` task type.

However, you can omit the type if you only need to configure properties or call methods that are common to all tasks, i.e., those declared on the `Task` interface.

You can discover what tasks are available by running `./gradlew tasks`.

To find out the type of a given task, run `./gradlew help --task <taskName>`, as demonstrated here:

```
❯ ./gradlew help --task test
...
Type
    Test (org.gradle.api.tasks.testing.Test)
```

The IDE can assist you with the required imports, so you only need the simple names of the types, without the package name part. In this case, there's no need to import the `Test` task type, as it is part of the Gradle API and is therefore [imported implicitly](#).

About conventions

Some of the Gradle core plugins expose configurability with the help of a so-called *convention* object. These serve a similar purpose to — and have now been superseded by — *extensions*. Conventions are deprecated, so please avoid using convention objects when writing new plugins.

As mentioned above, the Kotlin DSL provides accessors only for convention objects on `Project`. However, there are situations where you may need to interact with a Gradle plugin that uses convention objects on other types. The Kotlin DSL provides the `withConvention(T::class) {}` extension function for this:

build.gradle.kts

```
sourceSets {  
    main {  
        withConvention(CustomSourceSetConvention::class) {  
            someOption = "some value"  
        }  
    }  
}
```

This technique is primarily necessary for source sets added by language plugins that have yet to be migrated to extensions.

Working with container objects

The Gradle build model makes extensive use of container objects (or simply "containers").

For example, `configurations` and `tasks` are containers that hold `Configuration` and `Task` objects, respectively. Community plugins also contribute containers, such as the `android.buildTypes` container contributed by the Android Plugin.

The Kotlin DSL provides multiple ways for build authors to interact with containers. We will explore each of these methods, using the `tasks` container as an example.

TIP

You can leverage the type-safe accessors described in [another section](#) when configuring existing elements on supported containers. That section also explains which containers support type-safe accessors.

Using the container API

All containers in Gradle implement `NamedDomainObjectContainer<DomainObjectType>`. Some containers can hold objects of different types and implement `PolymorphicDomainObjectContainer<BaseType>`. The simplest way to interact with containers is through these interfaces.

The following example demonstrates how you can use the `named()` method to configure existing tasks, and the `register()` method to create new tasks:

build.gradle.kts

```
tasks.named("check")           ①  
tasks.register("myTask1")       ②  
  
tasks.named<JavaCompile>("compileJava") ③  
tasks.register<Copy>("myCopy1")         ④
```

```

tasks.named("assemble") {           ⑤
    dependsOn(":myTask1")
}
tasks.register("myTask2") {          ⑥
    description = "Some meaningful words"
}

tasks.named<Test>("test") {           ⑦
    testLogging.showStackTraces = true
}
tasks.register<Copy>("myCopy2") {     ⑧
    from("source")
    into("destination")
}

```

- ① Gets a reference of type `Task` to the existing task named `check`
- ② Registers a new untyped task named `myTask1`
- ③ Gets a reference to the existing task named `compileJava` of type `JavaCompile`
- ④ Registers a new task named `myCopy1` of type `Copy`
- ⑤ Gets a reference to the existing (untyped) task named `assemble` and configures it — you can only configure properties and methods that are available on `Task` with this syntax
- ⑥ Registers a new untyped task named `myTask2` and configures it — you can only configure properties and methods that are available on `Task` in this case
- ⑦ Gets a reference to the existing task named `test` of type `Test` and configures it — in this case you have access to the properties and methods of the specified type
- ⑧ Registers a new task named `myCopy2` of type `Copy` and configures it

NOTE

The above sample relies on the [configuration avoidance APIs](#). If you need or want to eagerly configure or register container elements, simply replace `named()` with `getByName()` and `register()` with `create()`.

Using Kotlin delegated properties

Another way to interact with containers is via Kotlin delegated properties. These are particularly useful if you need a reference to a container element that you can use elsewhere in the build. Additionally, Kotlin delegated properties can easily be renamed via IDE refactoring.

The following example achieves the same result as the one in the previous section, but it uses delegated properties and reuses those references instead of string-literal task paths:

build.gradle.kts

```
val check by tasks.existing
val myTask1 by tasks.registering

val compileJava by tasks.existing(JavaCompile::class)
val myCopy1 by tasks.registering(Copy::class)

val assemble by tasks.existing {
    dependsOn(myTask1) ❶
}
val myTask2 by tasks.registering {
    description = "Some meaningful words"
}

val test by tasks.existing(Test::class) {
    testLogging.showStackTraces = true
}
val myCopy2 by tasks.registering(Copy::class) {
    from("source")
    into("destination")
}
```

❶ Uses the reference to the `myTask1` task rather than a task path

NOTE

The above sample relies on the [configuration avoidance APIs](#). If you need or want to eagerly configure or register container elements, simply replace `existing()` with `getting()` and `registering()` with `creating()`.

Configuring multiple container elements together

When configuring several elements of a container, you can group interactions in a block to avoid repeating the container's name on each interaction.

The following example demonstrates a combination of type-safe accessors, the container API, and Kotlin delegated properties:

build.gradle.kts

```
tasks {
    test {
        testLogging.showStackTraces = true
    }
    val myCheck by registering {
        doLast { /* assert on something meaningful */ }
    }
}
```

```
}
check {
    dependsOn(myCheck)
}
register("myHelp") {
    doLast { /* do something helpful */ }
}
}
```

Working with runtime properties

Gradle has two main sources of properties defined at runtime: [project properties](#) and [extra properties](#).

The Kotlin DSL provides specific syntax for working with these property types, which we will explore in the following sections.

Project properties

The Kotlin DSL allows you to access project properties by binding them via Kotlin delegated properties.

The following snippet demonstrates this technique for a couple of project properties, one of which *must* be defined:

build.gradle.kts

```
val myProperty: String by project ①
val myNullableProperty: String? by project ②
```

- ① Makes the `myProperty` project property available via a `myProperty` delegated property — the project property must exist in this case, otherwise the build will fail when the build script attempts to use the `myProperty` value
- ② Does the same for the `myNullableProperty` project property, but the build won't fail on using the `myNullableProperty` value as long as you check for null (standard [Kotlin rules for null safety](#) apply)

The same approach works in both settings and initialization scripts, except you use `by settings` and `by gradle` respectively in place of `by project`.

Extra properties

Extra properties are available on any object that implements the [ExtensionAware](#) interface.

In Kotlin DSL, you can access and create extra properties via delegated properties, using the **by extra** syntax as demonstrated in the following sample:

build.gradle.kts

```
val myNewProperty by extra("initial value") ①
val myOtherNewProperty by extra { "calculated initial value" } ②

val myExtraProperty: String by extra ③
val myExtraNullableProperty: String? by extra ④
```

- ① Creates a new extra property called **myNewProperty** in the current context (the project in this case) and initializes it with the value **"initial value"**, which also determines the property's *type*
- ② Create a new extra property whose initial value is calculated by the provided lambda
- ③ Binds an existing extra property from the current context (the project in this case) to a **myProperty** reference
- ④ Does the same as the previous line but allows the property to have a null value

This approach works for all Gradle scripts: project build scripts, script plugins, settings scripts, and initialization scripts.

You can also access extra properties on a root project from a subproject using the following syntax:

my-sub-project/build.gradle.kts

```
val myNewProperty: String by rootProject.extra ①
```

- ① Binds the root project's **myNewProperty** extra property to a reference of the same name

Extra properties aren't just limited to projects. For example, **Task** extends **ExtensionAware**, so you can attach extra properties to tasks as well.

Here's an example that defines a new **myNewTaskProperty** on the **test** task and then uses that property to initialize another task:

build.gradle.kts

```
tasks {
    test {
        val reportType by extra("dev") ①
        doLast {
```

```

        // Use 'suffix' for post-processing of reports
    }
}

register<Zip>("archiveTestReports") {
    val reportType: String by test.get().extra ②
    archiveAppendix = reportType
    from(test.get().reports.html.outputLocation)
}
}

```

- ① Creates a new `reportType` extra property on the `test` task
- ② Makes the `test` task's `reportType` extra property available to configure the `archiveTestReports` task

If you're happy to use eager configuration rather than the configuration avoidance APIs, you could use a single, "global" property for the report type, like this:

build.gradle.kts

```

tasks.test {
    doLast { /* ... */ }
}

val testReportType by tasks.test.get().extra("dev") ①

tasks.create<Zip>("archiveTestsReports") {
    archiveAppendix = testReportType ②
    from(test.reports.html.outputLocation)
}

```

- ① Creates and initializes an extra property on the `test` task, binding it to a "global" property
- ② Uses the "global" property to initialize the `archiveTestReports` task

There is one last syntax for extra properties that treats `extra` as a map. We generally recommend against using this, as it bypasses Kotlin's type checking and limits IDE support. However, it is more succinct than the delegated properties syntax and can be used if you only need to set an extra property without referencing it later.

Here is a simple example demonstrating how to set and read extra properties using the map syntax:

build.gradle.kts

```
extra["myNewProperty"] = "initial value" ❶

tasks.register("myTask") {
    doLast {
        println("Property: ${project.extra["myNewProperty"]}") ❷
    }
}
```

- ❶ Creates a new project extra property called `myNewProperty` and sets its value
- ❷ Reads the value from the project extra property we created — note the `project.` qualifier on `extra[...]`, otherwise Gradle will assume we want to read an extra property from the *task*

Working with Gradle types

`Property`, `Provider`, and `NamedDomainObjectProvider` are **types** that represent deferred and lazy evaluation of values and objects. The Kotlin DSL provides a specialized syntax for working with these types.

Using a `Property`

A property represents a value that can be set and read lazily:

- Setting a value: `property.set(value)` or `property = value`
- Accessing the value: `property.get()`
- Using the delegate syntax: `val propValue: String by property`

build.gradle.kts

```
val myProperty: Property<String> =
    project.objects.property(String::class.java)

myProperty.set("Hello, Gradle!") // Set the value
println(myProperty.get())        // Access the value

// Using delegate syntax
val propValue: String by myProperty
println(propValue)

// Using lazy syntax
myProperty = "Hi, Gradle!" // Set the value
println(myProperty.get())  // Access the value
```

Using a **Provider**

A provider represents a read-only, lazily-evaluated value:

- Accessing the value: `provider.get()`
- Chaining: `provider.map { transform(it) }`

build.gradle.kts

```
val versionProvider: Provider<String> = project.provider { "1.0.0" }

println(versionProvider.get()) // Access the value

// Chaining transformations
val majorVersion: Provider<String> = versionProvider.map { it.split(".")[0] }
println(majorVersion.get()) // Prints: "1"
```

Using a **NamedDomainObjectProvider**

A named domain object provider represents a lazily-evaluated named object from a Gradle container (like tasks or extensions):

- Accessing the object: `namedObjectProvider.get()`
- Configuring the object: `namedObjectProvider.configure { ... }`

build.gradle.kts

```
val myTaskProvider: NamedDomainObjectProvider<Task> = tasks.named("build")

// Configuring the task
myTaskProvider.configure {
    doLast {
        println("Build task completed!")
    }
}

// Accessing the task
val myTask: Task = myTaskProvider.get()
```

Lazy property assignment

Gradle's Kotlin DSL supports lazy property assignment using the `=` operator.

Lazy property assignment reduces verbosity when [lazy properties](#) are used. It works for properties that are publicly seen as `final` (without a setter) and have type `Property` or `ConfigurableFileCollection`. Since properties must be `final`, we generally recommend avoiding custom setters for properties with lazy types and, if possible, implementing such properties via an abstract getter.

Using the `=` operator is the preferred way to call `set()` in the Kotlin DSL:

build.gradle.kts

```
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(17)
    }
}

abstract class WriteJavaVersionTask : DefaultTask() {
    @get:Input
    abstract val javaVersion: Property<String>
    @get:OutputFile
    abstract val output: RegularFileProperty

    @TaskAction
    fun execute() {
        output.get().asFile.writeText("Java version: ${javaVersion.get()}")
    }
}

tasks.register<WriteJavaVersionTask>("writeJavaVersion") {
    javaVersion.set("17") ❶
    javaVersion = "17" ❷
    javaVersion = java.toolchain.languageVersion.map { it.toString() } ❸
    output = layout.buildDirectory.file("writeJavaVersion/javaVersion.txt")
}
```

- ❶ Set value with the `.set()` method
- ❷ Set value with lazy property assignment using the `=` operator
- ❸ The `=` operator can be used also for assigning lazy values

IDE support

Lazy property assignment is supported from IntelliJ 2022.3 and from Android Studio Giraffe.

Kotlin DSL Plugin

The Kotlin DSL Plugin provides a convenient way to develop Kotlin-based projects that contribute

build logic. This includes [buildSrc projects](#), [included builds](#), and [Gradle plugins](#).

The plugin achieves this by doing the following:

- Applies the [Kotlin Plugin](#), which adds support for compiling Kotlin source files.
- Adds the `kotlin-stdlib`, `kotlin-reflect`, and `gradleKotlinDsl()` dependencies to the `compileOnly` and `testImplementation` configurations, enabling the use of those Kotlin libraries and the Gradle API in your Kotlin code.
- Configures the Kotlin compiler with the same settings used for Kotlin DSL scripts, ensuring consistency between your build logic and those scripts:
 - Adds [Kotlin compiler arguments](#),
 - Registers the [SAM-with-receiver Kotlin compiler plugin](#).
- Enables support for [precompiled script plugins](#).

Each Gradle release is meant to be used with a specific version of the `kotlin-dsl` plugin. Compatibility between arbitrary Gradle releases and `kotlin-dsl` plugin versions is not guaranteed. Using an unexpected version of the `kotlin-dsl` plugin will emit a warning and can cause hard-to-diagnose problems.

This is the basic configuration you need to use the plugin:

buildSrc/build.gradle.kts

```
plugins {
    `kotlin-dsl`
}

repositories {
    // The org.jetbrains.kotlin.jvm plugin requires a repository
    // where to download the Kotlin compiler dependencies from.
    mavenCentral()
}
```

The Kotlin DSL Plugin leverages [Java Toolchains](#). By default, the code will target Java 8. You can change that by defining a Java toolchain to be used by the project:

buildSrc/src/main/kotlin/myproject.java-conventions.gradle.kts

```
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(11)
    }
}
```

Embedded Kotlin

Gradle embeds Kotlin in order to provide support for Kotlin-based scripts.

Kotlin versions

Gradle ships with `kotlin-compiler-embeddable` plus matching versions of `kotlin-stdlib` and `kotlin-reflect` libraries. For details, see the Kotlin section of Gradle's [compatibility matrix](#). The `kotlin` package from those modules is visible through the Gradle classpath.

The [compatibility](#) guarantees provided by Kotlin apply for both backward and forward compatibility.

Backward compatibility

Our approach is to only make backward-incompatible Kotlin upgrades with major Gradle releases. We clearly document the Kotlin version shipped with each release and announce upgrade plans ahead of major releases.

Plugin authors aiming to maintain compatibility with older Gradle versions must limit their API usage to what is supported by those versions. This is no different from working with any new API in Gradle. For example, if a new API for dependency resolution is introduced, a plugin must either drop support for older Gradle versions or organize its code to conditionally execute the new code path on compatible versions.

Forward compatibility

The primary compatibility concern lies between the external `kotlin-gradle-plugin` version and the `kotlin-stdlib` version shipped with Gradle. More broadly, this applies to any plugin that transitively depends on `kotlin-stdlib` and its version provided by Gradle. As long as the versions are compatible, everything should work as expected. This issue will diminish as the Kotlin language matures.

Kotlin compiler arguments

The following Kotlin compiler arguments are used for compiling Kotlin DSL scripts, as well as Kotlin sources and scripts in projects with the `kotlin-dsl` plugin applied:

`-java-parameters`

Generate metadata for Java ≥ 1.8 reflection on method parameters. See [Kotlin/JVM compiler options](#) in the Kotlin documentation for more information.

`-Xjvm-default=all`

Makes all non-abstract members of Kotlin interfaces default for the Java classes implementing them. This is to provide a better interoperability with Java and Groovy for plugins written in Kotlin. See [Default methods in interfaces](#) in the Kotlin documentation for more information.

`-Xsam-conversions=class`

Sets up the implementation strategy for SAM (single abstract method) conversion to always generate anonymous classes, instead of using the `invokedynamic` JVM instruction. This is to

provide a better support for configuration cache and incremental build. See [KT-44912](#) in the Kotlin issue tracker for more information.

-Xjsr305=strict

Sets up Kotlin's Java interoperability to strictly follow JSR-305 annotations for increased null safety. See [Calling Java code from Kotlin](#) in the Kotlin documentation for more information.

Interoperability

When mixing languages in your build logic, you may have to cross language boundaries. An extreme example would be a build that uses tasks and plugins that are implemented in Java, Groovy and Kotlin, while also using both Kotlin DSL and Groovy DSL build scripts.

Kotlin is designed with Java Interoperability in mind. Existing Java code can be called from Kotlin in a natural way, and Kotlin code can be used from Java rather smoothly as well.

— Kotlin reference documentation

Both [calling Java from Kotlin](#) and [calling Kotlin from Java](#) are very well covered in the Kotlin reference documentation.

The same mostly applies to interoperability with Groovy code. In addition, the Kotlin DSL provides several ways to opt into Groovy semantics, which we look at next.

Static extensions

Both the Groovy and Kotlin languages support extending existing classes via [Groovy Extension modules](#) and [Kotlin extensions](#).

To call a Kotlin extension function from Groovy, call it as a static function, passing the receiver as the first parameter:

build.gradle

```
TheTargetTypeKt.kotlinExtensionFunction(receiver, "parameters", 42, aReference)
```

Kotlin extension functions are package-level functions. You can learn how to locate the name of the type declaring a given Kotlin extension in the [Package-Level Functions](#) section of the Kotlin reference documentation.

To call a Groovy extension method from Kotlin, the same approach applies: call it as a static function passing the receiver as the first parameter:

build.gradle.kts

```
TheTargetTypeGroovyExtension.groovyExtensionMethod(receiver, "parameters",  
42, aReference)
```

Named parameters and default arguments

Both the Groovy and Kotlin languages support named function parameters and default arguments, although they are implemented very differently. Kotlin has fully-fledged support for both, as described in the Kotlin language reference under [named arguments](#) and [default arguments](#). Groovy implements [named arguments](#) in a non-type-safe way based on a `Map<String, ?>` parameter, which means they cannot be combined with [default arguments](#). In other words, you can only use one or the other in Groovy for any given method.

Calling Kotlin from Groovy

To call a Kotlin function that has named arguments from Groovy, just use a normal method call with positional parameters:

build.gradle

```
kotlinFunction("value1", "value2", 42)
```

There is no way to provide values by argument name.

To call a Kotlin function that has default arguments from Groovy, always pass values for all the function parameters.

Calling Groovy from Kotlin

To call a Groovy function with named arguments from Kotlin, you need to pass a `Map<String, ?>`, as shown in this example:

build.gradle.kts

```
groovyNamedArgumentTakingMethod(mapOf(  
    "parameterName" to "value",  
    "other" to 42,  
    "and" to aReference))
```

To call a Groovy function with default arguments from Kotlin, always pass values for all the parameters.

Groovy closures from Kotlin

You may sometimes have to call Groovy methods that take [Closure](#) arguments from Kotlin code. For

example, some third-party plugins written in Groovy expect closure arguments.

NOTE

Gradle plugins written in any language should prefer the type `Action<T>` type in place of closures. Groovy closures and Kotlin lambdas are automatically mapped to arguments of that type.

In order to provide a way to construct closures while preserving Kotlin's strong typing, two helper methods exist:

- `closureOf<T> {}`
- `delegateClosureOf<T> {}`

Both methods are useful in different circumstances and depend upon the method you are passing the `Closure` instance into.

Some plugins expect simple closures, as with the [Bintray](#) plugin:

build.gradle.kts

```
bintray {
    pkg(closureOf<PackageConfig> {
        // Config for the package here
    })
}
```

In other cases, like with the [Gretty Plugin](#) when configuring farms, the plugin expects a delegate closure:

build.gradle.kts

```
farms {
    farm("OldCoreWar", delegateClosureOf<FarmExtension> {
        // Config for the war here
    })
}
```

There sometimes isn't a good way to tell, from looking at the source code, which version to use. Usually, if you get a `NullPointerException` with `closureOf<T> {}`, using `delegateClosureOf<T> {}` will resolve the problem.

These two utility functions are useful for *configuration closures*, but some plugins might expect Groovy closures for other purposes. The `KotlinClosure0` to `KotlinClosure2` types allows adapting Kotlin functions to Groovy closures with more flexibility:

build.gradle.kts

```
somePlugin {  
  
    // Adapt parameter-less function  
    takingParameterLessClosure(KotlinClosure0({  
        "result"  
    })))  
  
    // Adapt unary function  
    takingUnaryClosure(KotlinClosure1<String, String>({  
        "result from single parameter $this"  
    })))  
  
    // Adapt binary function  
    takingBinaryClosure(KotlinClosure2<String, String, String>({ a, b ->  
        "result from parameters $a and $b"  
    })))  
}
```

The Kotlin DSL Groovy Builder

If some plugin makes heavy use of [Groovy metaprogramming](#), then using it from Kotlin or Java or any statically-compiled language can be very cumbersome.

The Kotlin DSL provides a `withGroovyBuilder {}` utility extension that attaches the Groovy metaprogramming semantics to objects of type `Any`.

The following example demonstrates several features of the method on the object `target`:

build.gradle.kts

```
target.withGroovyBuilder {  
  
    // GroovyObject methods available  
    if (hasProperty("foo")) { /*...*/ }  
    val foo = getProperty("foo")  
    setProperty("foo", "bar")  
    invokeMethod("name", arrayOf("parameters", 42, aReference))  
  
    // Kotlin DSL utilities  
    "name"("parameters", 42, aReference)  
        "blockName" {  
            // Same Groovy Builder semantics on `blockName`  
        }  
    "another"("name" to "example", "url" to "https://example.com/")
```

```
}
```

- ① The receiver is a `GroovyObject` and provides Kotlin helpers
- ② The `GroovyObject` API is available
- ③ Invoke the `methodName` method, passing some parameters
- ④ Configure the `blockName` property, maps to a `Closure` taking method invocation
- ⑤ Invoke `another` method taking named arguments, maps to a Groovy named arguments `Map<String, ?>` taking method invocation

Using a Groovy script

Another option when dealing with problematic plugins that assume a Groovy DSL build script is to configure them in a Groovy DSL build script that is applied from the main Kotlin DSL build script:

dynamic-groovy-plugin-configuration.gradle

```
native { ①
    dynamic {
        groovy as Usual
    }
}
```

build.gradle.kts

```
plugins {
    id("dynamic-groovy-plugin") version "1.0" ②
}
apply(from = "dynamic-groovy-plugin-configuration.gradle") ③
```

- ① The Groovy script uses dynamic Groovy to configure plugin
- ② The Kotlin build script requests and applies the plugin
- ③ The Kotlin build script applies the Groovy script

Troubleshooting

The IDE support is provided by two components:

1. Kotlin Plugin (used by IntelliJ IDEA/Android Studio).

2. Gradle.

The level of support varies based on the versions of each.

If you encounter issues, first run `./gradlew tasks` from the command line to determine if the problem is specific to the IDE. If the issue persists on the command line, it likely originates from the build itself rather than IDE integration.

However, if the build runs successfully on the command line but your script editor reports errors, try restarting your IDE and invalidating its caches.

If the issue persists, and you suspect a problem with the Kotlin DSL script editor, try the following:

- Run `./gradlew tasks` to gather more details.
- Check the logs in one of these locations:
 - `$HOME/Library/Logs/gradle-kotlin-dsl` on macOS
 - `$HOME/.gradle-kotlin-dsl/log` on Linux
 - `$HOME/AppData/Local/gradle-kotlin-dsl/log` on Windows
- Report the issue on the [Gradle issue tracker](#), including as much detail as possible.

From version 5.1 onward, the log directory is automatically cleaned. Logs are checked periodically (at most, every 24 hours), and files are deleted if unused for 7 days.

If this doesn't help pinpoint the problem, you can enable the `org.gradle.kotlin.dsl.logging.tapi` system property in your IDE. This causes the Gradle Daemon to log additional details in its log file located at `$HOME/.gradle/daemon`.

In IntelliJ IDEA, enable this property by navigating to `Help > Edit Custom VM Options...` and adding: `-Dorg.gradle.kotlin.dsl.logging.tapi=true`.

For IDE problems outside the Kotlin DSL script editor, please open issues in the corresponding IDE's issue tracker:

- [JetBrains's IDEA issue tracker](#)
- [Google's Android Studio issue tracker](#)

Lastly, if you face problems with Gradle itself or with the Kotlin DSL, please open issues on the [Gradle issue tracker](#).

Limitations

- The Kotlin DSL is [known to be slower than the Groovy DSL](#) on first use, for example with clean checkouts or on ephemeral continuous integration agents. Changing something in the `buildSrc` directory also has an impact as it invalidates build-script caching. The main reason for this is the slower script compilation for Kotlin DSL.
- In IntelliJ IDEA, you must [import your project from the Gradle model](#) in order to get content assist and refactoring support for your Kotlin DSL build scripts.

- Kotlin DSL script compilation avoidance has known issues. If you encounter problems, it can be disabled by [setting](#) the `org.gradle.kotlin.dsl.scriptCompilationAvoidance` system property to `false`.
- The Kotlin DSL will not support the `model {}` block, which is part of the [discontinued Gradle Software Model](#).

If you run into trouble or discover a suspected bug, please report the issue in the [Gradle issue tracker](#).

CORE PLUGINS

Gradle Plugin Reference

This page contains links and short descriptions for all the core plugins provided by Gradle itself.

JVM languages and frameworks

Java

Provides support for building any type of Java project.

Java Library

Provides support for building a Java library.

Java Platform

Provides support for building a Java platform.

Groovy

Provides support for building any type of [Groovy](#) project.

Scala

Provides support for building any type of [Scala](#) project.

ANTLR

Provides support for generating parsers using [ANTLR](#).

JVM Test Suite

Provides support for modeling and configuring multiple test suite invocations.

Test Report Aggregation

Aggregates the results of multiple [Test](#) task invocations (potentially spanning multiple Gradle projects) into a single HTML report.

Native languages

C++ Application

Provides support for building C++ applications on Windows, Linux, and macOS.

C++ Library

Provides support for building C++ libraries on Windows, Linux, and macOS.

C++ Unit Test

Provides support for building and running C++ executable-based tests on Windows, Linux, and macOS.

Swift Application

Provides support for building Swift applications on Linux and macOS.

Swift Library

Provides support for building Swift libraries on Linux and macOS.

XCTest

Provides support for building and running XCTest-based tests on Linux and macOS.

Packaging and distribution

Application

Provides support for building JVM-based, runnable applications.

WAR

Provides support for building and packaging WAR-based Java web applications.

EAR

Provides support for building and packaging Java EE applications.

Maven Publish

Provides support for [publishing artifacts](#) to Maven-compatible repositories.

Ivy Publish

Provides support for [publishing artifacts](#) to Ivy-compatible repositories.

Distribution

Makes it easy to create ZIP and tarball distributions of your project.

Java Library Distribution

Provides support for creating a ZIP distribution of a Java library project that includes its runtime dependencies.

Code analysis

Checkstyle

Performs quality checks on your project's Java source files using [Checkstyle](#) and generates associated reports.

PMD

Performs quality checks on your project's Java source files using [PMD](#) and generates associated reports.

JaCoCo

Provides code coverage metrics for your Java project using [JaCoCo](#).

JaCoCo Report Aggregation

Aggregates the results of multiple JaCoCo code coverage reports (potentially spanning multiple Gradle projects) into a single HTML report.

CodeNarc

Performs quality checks on your Groovy source files using [CodeNarc](#) and generates associated reports.

IDE integration

Eclipse

Generates Eclipse project files for the build that can be opened by the IDE. This set of plugins can also be used to fine tune [Buildship's](#) import process for Gradle builds.

IntelliJ IDEA

Generates IDEA project files for the build that can be opened by the IDE. It can also be used to fine tune IDEA's import process for Gradle builds.

Visual Studio

Generates Visual Studio solution and project files for build that can be opened by the IDE.

Xcode

Generates Xcode workspace and project files for the build that can be opened by the IDE.

Utility

Base

Provides common lifecycle tasks, such as `clean`, and other features common to most builds.

Build Init

Generates a new Gradle build of a specified type, such as a Java library. It can also generate a build script from a Maven POM — see [Migrating from Maven to Gradle](#) for more details.

Signing

Provides support for digitally signing generated files and artifacts.

Plugin Development

Makes it easier to develop and publish a Gradle plugin.

Project Report Plugin

Helps to generate reports containing useful information about your build.

HOW TO GUIDES

How to share outputs between projects

A common pattern, in multi-project builds, is that one project consumes the artifacts of another project.

In general, the simplest consumption form in the Java ecosystem is that when **A** depends on **B**, then **A** would depend on the **jar** produced by project **B**.

Considerations and possible solutions

A frequent anti-pattern to declare cross-project dependencies is:

```
dependencies {  
    // this is unsafe!  
    implementation project(":other").tasks.someOtherJar  
}
```

This publication model is *unsafe* and can lead to non-reproducible and hard to parallelize builds.

WARNING

Don't reference other project tasks directly!

You could define a configuration on the producer side which serves as an exchange of artifacts between the producer and the consumer.

consumer/build.gradle

```
dependencies {  
    instrumentedClasspath(project(path: ":producer", configuration:  
    'instrumentedJars'))  
}
```

consumer/build.gradle.kts

```
dependencies {  
    instrumentedClasspath(project(mapOf(  
        "path" to ":producer",  
        "configuration" to "instrumentedJars")))  
}
```

However, the consumer has to explicitly tell which configuration it depends on and this is *not*

recommended. If you plan to publish the component which has this dependency, it will likely lead to broken metadata.

This section explains how to *properly create cross-project boundaries* by defining "exchanges" between projects by using *variants*.

Variant-aware sharing of artifacts

Gradle's [variant model](#) allows consumers to specify requirements using *attributes*, while producers provide appropriate outgoing variants using attributes as well.

For example, a single dependency declaration like `project(":myLib")` can select either the `arm64` or `i386` version of `myLib`, based on the architecture.

To achieve this, attributes must be defined on both the consumer and producer configurations.

WARNING

When configurations have attributes, they participate in *variant-aware resolution*. This means they become candidates for resolution whenever any dependency declaration, such as `project(":myLib")`, is used.

Attributes on producer configurations *must be consistent with other variants provided by the same project*. Introducing inconsistent or ambiguous attributes can lead to resolution failures.

In practice, the attributes you define will often depend on the *ecosystem* (e.g., Java, C++) because ecosystem-specific plugins typically apply different attribute conventions.

Consider an example of a Java Library project. Java libraries typically expose two variants to consumers: `apiElements` and `runtimeElements`. In this case, we are adding a third variant, `instrumentedJars`.

To correctly configure this new variant, we need to understand its purpose and set appropriate attributes. Here are the attributes on the `runtimeElements` configuration of the producer:

```
$ ./gradle outgoingVariants --variant runtimeElements
```

Attributes

```
- org.gradle.category           = library
- org.gradle.dependency.bundling = external
- org.gradle.jvm.version        = 11
- org.gradle.libraryelements     = jar
- org.gradle.usage               = java-runtime
```

This tells us that the `runtimeElements` configuration includes 5 attributes:

1. `org.gradle.category` indicates that this variant represents a *library*.
2. `org.gradle.dependency.bundling` specifies that dependencies are external jars (not repackaged inside the jar).

3. `org.gradle.jvm.version` denotes the minimum Java version supported, which is Java 11.
4. `org.gradle.libraryelements` shows that this variant contains all elements typically found in a jar (classes and resources).
5. `org.gradle.usage` defines the variant as a Java runtime, suitable for both compilation and runtime.

To ensure that the `instrumentedJars` variant is used in place of `runtimeElements` when executing tests, we must attach similar attributes to this new variant.

The key attribute for this configuration is `org.gradle.libraryelements`, as it describes *what the variant contains*. We can set up the `instrumentedJars` variant accordingly:

producer/build.gradle

```
configurations {
    instrumentedJars {
        canBeConsumed = true
        canBeResolved = false
        attributes {
            attribute(Category.CATEGORY_ATTRIBUTE, objects.named(Category,
Category.LIBRARY))
            attribute(Usage.USAGE_ATTRIBUTE, objects.named(Usage, Usage
.JAVA_RUNTIME))
            attribute(Bundling.BUNDLING_ATTRIBUTE, objects.named(Bundling,
Bundling.EXTERNAL))
            attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
JavaVersion.current().majorVersion.toInteger())
            attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE, objects
.named(LibraryElements, 'instrumented-jar'))
        }
    }
}
```

producer/build.gradle.kts

```
val instrumentedJars by configurations.creating {
    isCanBeConsumed = true
    isCanBeResolved = false
    attributes {
        attribute(Category.CATEGORY_ATTRIBUTE,
objects.named(Category.LIBRARY))
        attribute(Usage.USAGE_ATTRIBUTE, objects.named(Usage.JAVA_RUNTIME))
        attribute(Bundling.BUNDLING_ATTRIBUTE,
objects.named(Bundling.EXTERNAL))
        attribute(TargetJvmVersion.TARGET_JVM_VERSION_ATTRIBUTE,
JavaVersion.current().majorVersion.toInt())
    }
}
```

```
        attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE,
objects.named("instrumented-jar"))
    }
}
```

This ensures that the `instrumentedJars` variant is correctly identified as containing elements similar to a jar, allowing it to be selected appropriately.

NOTE

Selecting the right attributes is the most challenging part of this process, as they define the semantics of the variant. Before introducing *new attributes*, always consider whether an existing attribute already conveys the required semantics. If no suitable attribute exists, you can create a new one. However, be cautious—adding a new attribute may introduce ambiguity during variant selection. In many cases, adding an attribute requires applying it consistently across *all* existing variants.

We've introduced a *new* variant for *runtime* that provides instrumented classes instead of the normal ones. As a result, consumers now face a choice between two runtime variants:

1. `runtimeElements` - the default runtime variant provided by the `java-library` plugin.
2. `instrumentedJars` - the custom variant we've added.

If we want the instrumented classes to be included on the test runtime classpath, we can now declare the dependency on the consumer as a regular project dependency:

consumer/build.gradle

```
dependencies {
    testImplementation 'junit:junit:4.13'
    testImplementation project(':producer')
}
```

consumer/build.gradle.kts

```
dependencies {
    testImplementation("junit:junit:4.13")
    testImplementation(project(":producer"))
}
```

If we stop here, Gradle will still resolve the `runtimeElements` variant instead of the `instrumentedJars` variant.

This happens because the `testRuntimeClasspath` configuration requests a variant with the `libraryelements` attribute set to `jar`, and our `instrumented-jars` value does not match.

To fix this, we need to update the requested attributes to specifically target instrumented jars:

consumer/build.gradle

```
configurations {
    testRuntimeClasspath {
        attributes {
            attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE, objects
                .named(LibraryElements, 'instrumented-jar'))
        }
    }
}
```

consumer/build.gradle.kts

```
configurations {
    testRuntimeClasspath {
        attributes {
            attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE,
                objects.named(LibraryElements::class.java, "instrumented-jar"))
        }
    }
}
```

We can look at another report *on the consumer side* to view exactly what attributes of each dependency will be requested:

```
$ .gradle resolvableConfigurations --configuration testRuntimeClasspath
```

Attributes

- org.gradle.category	= library
- org.gradle.dependency.bundling	= external
- org.gradle.jvm.version	= 11
- org.gradle.libraryelements	= instrumented-jar
- org.gradle.usage	= java-runtime

The `resolvableConfigurations` report is the complement of the `outgoingVariants` report we ran previously.

By running both of these reports on the consumer and producer sides of a relationship, respectively, you can see exactly what attributes are involved in matching during dependency

resolution and better predict the outcome when configurations are resolved.

At this point, we're specifying that the test runtime classpath should resolve variants with *instrumented classes*.

However, there's an issue: some dependencies, like **JUnit**, don't provide instrumented classes. If we stop here, Gradle will fail, stating that no compatible variant of **JUnit** exists.

This happens because we haven't told Gradle that it's acceptable to fall back to the regular jar when an instrumented variant isn't available. To resolve this, we need to define a *compatibility rule*:

consumer/build.gradle

```
abstract class InstrumentedJarsRule implements AttributeCompatibilityRule<LibraryElements> {

    @Override
    void execute(CompatibilityCheckDetails<LibraryElements> details) {
        if (details.consumerValue.name == 'instrumented-jar' && details
            .producerValue.name == 'jar') {
            details.compatible()
        }
    }
}
```

consumer/build.gradle.kts

```
abstract class InstrumentedJarsRule:
    AttributeCompatibilityRule<LibraryElements> {
    override fun execute(details: CompatibilityCheckDetails<LibraryElements>)
= details.run {
        if (consumerValue?.name == "instrumented-jar" && producerValue?.name
== "jar") {
            compatible()
        }
    }
}
```

We then declare this rule on the attributes schema:

consumer/build.gradle

```
dependencies {
    attributesSchema {
```

```
        attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE) {
            compatibilityRules.add(InstrumentedJarsRule)
        }
    }
}
```

consumer/build.gradle.kts

```
dependencies {
    attributesSchema {
        attribute(LibraryElements.LIBRARY_ELEMENTS_ATTRIBUTE) {
            compatibilityRules.add(InstrumentedJarsRule::class.java)
        }
    }
}
```

And that's it! Now we have:

- Added a variant which provides instrumented jars.
- Specified that this variant is a substitute for the runtime.
- Defined that the consumer needs this variant only for *test runtime*.

Gradle provides a powerful mechanism for selecting the right variants based on preferences and compatibility. For more details, check out the [variant aware plugins section](#) of the documentation.

WARNING

By adding a value to an existing attribute or defining new attributes, we are extending the model. This means that *all consumers* must be aware of this extended model.

For local consumers, this is usually not a problem because all projects share the same schema. However, if you need to publish this new variant to an external repository, external consumers must also add the same rules to their builds for them to work.

This is generally not an issue for *ecosystem plugins* (e.g., the Kotlin plugin), where consumption is not possible without applying the plugin. However, it becomes problematic if you add custom values or attributes.

Therefore, **avoid publishing custom variants** if they are intended for internal use only.

LICENSE INFORMATION

License Information

Gradle Documentation

Copyright © 2024 Gradle, Inc. All rights reserved. Gradle is a trademark of Gradle, Inc.

Gradle's Build Tool source code is open-source and licensed under the [Apache License 2.0](#).

Gradle's User Manual and DSL Reference Manual are licensed under [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#).

Gradle Build Scan Plugin

Use of the [Build Scan plugin](#) is subject to [Gradle's Terms of Service](#).