Docs

[User Manual](http://docs.google.com/userguide/userguide.html)

[Guides and Tutorials](https://guides.gradle.org)

[DSL Reference](http://docs.google.com/dsl/)

[Javadoc](http://docs.google.com/javadoc/)

[Release Notes](http://docs.google.com/release-notes.html)

[Forums](https://discuss.gradle.org/)

[Training](https://gradle.org/training/)

[Try Gradle Enterprise](https://gradle.com/enterprise)

[PDF](http://docs.google.com/userguide/userguide.pdf)

* [User Manual Home](http://docs.google.com/userguide/userguide.html)
* [Release Notes](http://docs.google.com/release-notes.html)
* [Installing Gradle](http://docs.google.com/userguide/installation.html)
* [Tutorials](https://guides.gradle.org/)

### Reference

* [Groovy DSL Reference](http://docs.google.com/dsl/)
* [Gradle API Javadoc](http://docs.google.com/javadoc/)
* [Core Plugins](http://docs.google.com/userguide/plugin_reference.html)
* [Gradle & Third-party Tools](http://docs.google.com/userguide/third_party_integration.html)

### Getting Started

* [Creating New Gradle Builds](https://guides.gradle.org/creating-new-gradle-builds/)
* [Creating Build Scans](https://guides.gradle.org/creating-build-scans/)
* [Migrating From Maven](https://guides.gradle.org/migrating-from-maven/)

### Running Gradle Builds

* [Command-Line Interface](http://docs.google.com/userguide/command_line_interface.html)
* [Customizing Execution](#gjdgxs)
  + [Configuring the Build Environment](http://docs.google.com/userguide/build_environment.html)
  + [Configuring the Gradle Daemon](http://docs.google.com/userguide/gradle_daemon.html)
  + [Initialization Scripts](http://docs.google.com/userguide/init_scripts.html)
* [Directory Layout](http://docs.google.com/userguide/directory_layout.html)
* [Executing Multi-Project Builds](http://docs.google.com/userguide/intro_multi_project_builds.html)
* [Gradle Wrapper](http://docs.google.com/userguide/gradle_wrapper.html)
* [Troubleshooting](http://docs.google.com/userguide/troubleshooting.html)
* [Using Build Scans](https://docs.gradle.com/build-scan-plugin)
* [Enabling and Configuring the Build Cache](http://docs.google.com/userguide/build_cache.html)
* [Integrating Separate Gradle Builds (Composite Builds)](http://docs.google.com/userguide/composite_builds.html)

### Authoring Gradle Builds

* [Fundamentals](#30j0zll)
  + [Introducing the Basics of Build Scripts](http://docs.google.com/userguide/tutorial_using_tasks.html)
  + [Working with Tasks](http://docs.google.com/userguide/more_about_tasks.html)
  + [Learning More About Build Scripts](http://docs.google.com/userguide/writing_build_scripts.html)
  + [Working with Files](http://docs.google.com/userguide/working_with_files.html)
  + [Creating Custom Task Types](http://docs.google.com/userguide/custom_tasks.html)
  + [Using Gradle Plugins](http://docs.google.com/userguide/plugins.html)
  + [The Standard Gradle Plugins](http://docs.google.com/userguide/standard_plugins.html)
  + [Understanding the Build Lifecycle](http://docs.google.com/userguide/build_lifecycle.html)
  + [Working with Logging](http://docs.google.com/userguide/logging.html)
  + [Configuring Multi-Project Builds](http://docs.google.com/userguide/multi_project_builds.html)
* [Best Practices](#1fob9te)
  + [Authoring Maintainable Build Scripts](http://docs.google.com/userguide/authoring_maintainable_build_scripts.html)
  + [Organizing Gradle Projects](http://docs.google.com/userguide/organizing_gradle_projects.html)
  + [Optimizing Build Performance](https://guides.gradle.org/performance/)
  + [Using the Build Cache](https://guides.gradle.org/using-build-cache/)
* [Dependency Management](#3znysh7)
  + [Introduction to Dependency Management](http://docs.google.com/userguide/introduction_dependency_management.html)
  + [Dependency Management Terminology](http://docs.google.com/userguide/dependency_management_terminology.html)
  + [Dependency Types](http://docs.google.com/userguide/dependency_types.html)
  + [Repository Types](http://docs.google.com/userguide/repository_types.html)
  + [Declaring Dependencies](http://docs.google.com/userguide/declaring_dependencies.html)
  + [Declaring Repositories](http://docs.google.com/userguide/declaring_repositories.html)
  + [Inspecting Dependencies](http://docs.google.com/userguide/inspecting_dependencies.html)
  + [Managing Dependency Configurations](http://docs.google.com/userguide/managing_dependency_configurations.html)
  + [Managing Transitive Dependencies](http://docs.google.com/userguide/managing_transitive_dependencies.html)
  + [Dependency Locking](http://docs.google.com/userguide/dependency_locking.html)
  + [Troubleshooting Dependency Resolution](http://docs.google.com/userguide/troubleshooting_dependency_resolution.html)
  + [Customizing Dependency Resolution Behavior](http://docs.google.com/userguide/customizing_dependency_resolution_behavior.html)
  + [Dependency Cache Internals](http://docs.google.com/userguide/dependency_cache.html)
  + [Working with Dependencies](http://docs.google.com/userguide/working_with_dependencies.html)
* [Publishing Artifacts](http://docs.google.com/userguide/artifact_management.html)
* [C++ Projects](#2et92p0)
  + [Building Native Software](http://docs.google.com/userguide/native_software.html)
  + [Software Model Concepts](http://docs.google.com/userguide/software_model_concepts.html)
  + [Rule-based Model Configuration](http://docs.google.com/userguide/software_model.html)
  + [Implementing Model Rules in a Plugin](http://docs.google.com/userguide/rule_source.html)
  + [Extending the Software Model](http://docs.google.com/userguide/software_model_extend.html)
* [Java Projects](#tyjcwt)
  + [Building Java & JVM projects](http://docs.google.com/userguide/building_java_projects.html)
  + [Testing Java & JVM projects](http://docs.google.com/userguide/java_testing.html)
* [Advanced Techniques](#3dy6vkm)
  + [Configuring Tasks Lazily](http://docs.google.com/userguide/lazy_configuration.html)
  + [Developing Parallel Tasks](https://guides.gradle.org/using-the-worker-api/)
  + [Testing Your Build with TestKit](http://docs.google.com/userguide/test_kit.html)
  + [Using Ant from Gradle](http://docs.google.com/userguide/ant.html)
* [Sample Gradle builds](#1t3h5sf)
  + [Groovy DSL Samples](https://github.com/gradle/gradle/tree/master/subprojects/docs/src/samples)
  + [Kotlin DSL Samples](https://github.com/gradle/kotlin-dsl/tree/master/samples)

### Extending Gradle

* [Writing Custom Plugins](http://docs.google.com/userguide/custom_plugins.html)
* [Plugin Development Guides](https://gradle.org/guides/?q=Plugin+Development)

[Edit this page](https://github.com/gradle/gradle/edit/master/subprojects/docs/src/docs/userguide/)

# Gradle User Manual: Version 4.10.1

Table of Contents

[About Gradle](#4d34og8)

[Overview](#2s8eyo1)

[Getting Started](#17dp8vu)

[Installing Gradle](#3rdcrjn)

[Using Gradle Builds](#26in1rg)

[Command-Line Interface](#lnxbz9)

[Build Environment](#35nkun2)

[Directory Layout](#1ksv4uv)

[The Gradle Daemon](#44sinio)

[Initialization Scripts](#2jxsxqh)

[Executing Multi-Project Builds](#z337ya)

[The Gradle Wrapper](#3j2qqm3)

[Troubleshooting](#1y810tw)

[Authoring Gradle Builds](#4i7ojhp)

[The Feature Lifecycle](#2xcytpi)

[Authoring Maintainable Build Scripts](#1ci93xb)

[Organizing Gradle Projects](#3whwml4)

[Build Cache](#2bn6wsx)

[Build Init Plugin](#qsh70q)

[Build Lifecycle](#3as4poj)

[Build Script Basics](#1pxezwc)

[Composite builds](#49x2ik5)

[Authoring Multi-Project Builds](#2p2csry)

[Authoring Tasks](#147n2zr)

[Logging](#3o7alnk)

[Standard Gradle plugins](#23ckvvd)

[Testing Build Logic with TestKit](#ihv636)

[Using Gradle Plugins](#32hioqz)

[Working With Files](#1hmsyys)

[Writing Build Scripts](#41mghml)

[Writing Custom Task Classes](#2grqrue)

[The Base Plugin](#vx1227)

[Dependency Management](#3fwokq0)

[Introduction to Dependency Management](#1v1yuxt)

[Dependency Management Terminology](#4f1mdlm)

[Dependency Types](#2u6wntf)

[Repository Types](#19c6y18)

[Declaring Dependencies](#3tbugp1)

[Declaring Repositories](#28h4qwu)

[Inspecting Dependencies](#nmf14n)

[Managing Dependency Configurations](#37m2jsg)

[Managing Transitive Dependencies](#1mrcu09)

[Dependency Locking](#46r0co2)

[Troubleshooting Dependency Resolution](#2lwamvv)

[Customizing Dependency Resolution Behavior](#111kx3o)

[The Dependency Cache](#3l18frh)

[Working with Dependencies](#206ipza)

[Publishing Artifacts](#4k668n3)

[Publishing](#2zbgiuw)

[Maven Publish Plugin](#1egqt2p)

[Ivy Publish Plugin](#3ygebqi)

[Legacy publishing](#2dlolyb)

[Maven Plugin](#sqyw64)

[The Signing Plugin](#3cqmetx)

[The Distribution Plugin](#1rvwp1q)

[Native Projects](#4bvk7pj)

[Building native software](#2r0uhxc)

[Software model concepts](#1664s55)

[Rule based model configuration](#3q5sasy)

[Implementing model rules in a plugin](#25b2l0r)

[Extending the software model](#kgcv8k)

[Groovy Projects](#34g0dwd)

[Groovy Quickstart](#1jlao46)

[The Groovy Plugin](#43ky6rz)

[The CodeNarc Plugin](#2iq8gzs)

[Java Projects](#xvir7l)

[Java Quickstart](#3hv69ve)

[Building Java & JVM projects](#1x0gk37)

[Testing in Java & JVM projects](#4h042r0)

[The Java Plugin](#2w5ecyt)

[The Java Library Plugin](#1baon6m)

[The Java Library Distribution Plugin](#3vac5uf)

[Dependency Management for Java Projects](#2afmg28)

[Using Ant from Gradle](#pkwqa1)

[The ANTLR Plugin](#39kk8xu)

[The Application Plugin](#1opuj5n)

[The Checkstyle Plugin](#48pi1tg)

[The FindBugs Plugin](#2nusc19)

[The JaCoCo Plugin](#1302m92)

[The JDepend Plugin](#3mzq4wv)

[The OSGi Plugin](#2250f4o)

[The PMD Plugin](#haapch)

[Java Web Projects](#319y80a)

[The Ear Plugin](#1gf8i83)

[Building Play applications](#40ew0vw)

[The War Plugin](#2fk6b3p)

[Scala Projects](#upglbi)

[The Scala Plugin](#3ep43zb)

[Integrating Gradle](#1tuee74)

[The Eclipse Plugins](#4du1wux)

[The IDEA Plugin](#2szc72q)

[Embedding Gradle using the Tooling API](#184mhaj)

[Extending Gradle](#3s49zyc)

[Writing Custom Plugins](#279ka65)

[Gradle Plugin Development Plugin](#meukdy)

[Lazy Configuration](#36ei31r)

[Licenses](#1ljsd9k)

[Documentation licenses](#45jfvxd)

[Gradle Documentation](#2koq656)

[About Gradle](#4d34og8)

[Overview](#2s8eyo1)

[Features](#zu0gcz)

Here is a list of some of Gradle’s features.

Declarative builds and build-by-convention

At the heart of Gradle lies a rich extensible Domain Specific Language (DSL) based on Groovy. Gradle pushes declarative builds to the next level by providing declarative language elements that you can assemble as you like. Those elements also provide build-by-convention support for Java, Groovy, OSGi, Web and Scala projects. Even more, this declarative language is extensible. Add your own new language elements or enhance the existing ones, thus providing concise, maintainable and comprehensible builds.

Language for dependency based programming

The declarative language lies on top of a general purpose task graph, which you can fully leverage in your builds. It provides utmost flexibility to adapt Gradle to your unique needs.

Structure your build

The suppleness and richness of Gradle finally allows you to apply common design principles to your build. For example, it is very easy to compose your build from reusable pieces of build logic. Inline stuff where unnecessary indirections would be inappropriate. Don’t be forced to tear apart what belongs together (e.g. in your project hierarchy). Avoid smells like shotgun changes or divergent change that turn your build into a maintenance nightmare. At last you can create a well structured, easily maintained, comprehensible build.

Deep API

From being a pleasure to be used embedded to its many hooks over the whole lifecycle of build execution, Gradle allows you to monitor and customize its configuration and execution behavior to its very core.

Gradle scales

Gradle scales very well. It significantly increases your productivity, from simple single project builds up to huge enterprise multi-project builds. This is true for structuring the build. With the state-of-art incremental build function, this is also true for tackling the performance pain many large enterprise builds suffer from.

Multi-project builds

Gradle’s support for multi-project build is outstanding. Project dependencies are first class citizens. We allow you to model the project relationships in a multi-project build as they really are for your problem domain. Gradle follows your layout not vice versa.

Gradle provides partial builds. If you build a single subproject Gradle takes care of building all the subprojects that subproject depends on. You can also choose to rebuild the subprojects that depend on a particular subproject. Together with incremental builds this is a big time saver for larger builds.

Many ways to manage your dependencies

Different teams prefer different ways to manage their external dependencies. Gradle provides convenient support for any strategy. From transitive dependency management with remote Maven and Ivy repositories to jars or directories on the local file system.

Gradle is the first build integration tool

Ant tasks are first class citizens. Even more interesting, Ant projects are first class citizens as well. Gradle provides a deep import for any Ant project, turning Ant targets into native Gradle tasks at runtime. You can depend on them from Gradle, you can enhance them from Gradle, you can even declare dependencies on Gradle tasks in your build.xml. The same integration is provided for properties, paths, etc …​

Gradle fully supports your existing Maven or Ivy repository infrastructure for publishing and retrieving dependencies. Gradle also provides a converter for turning a Maven pom.xml into a Gradle script. Runtime imports of Maven projects will come soon.

Ease of migration

Gradle can adapt to any structure you have. Therefore you can always develop your Gradle build in the same branch where your production build lives and both can evolve in parallel. We usually recommend to write tests that make sure that the produced artifacts are similar. That way migration is as less disruptive and as reliable as possible. This is following the best-practices for refactoring by applying baby steps.

Groovy

Gradle’s build scripts are written in Groovy or Kotlin, not XML. But unlike other approaches this is not for simply exposing the raw scripting power of a dynamic language. That would just lead to a very difficult to maintain build. The whole design of Gradle is oriented towards being used as a language, not as a rigid framework. And Groovy is our glue that allows you to tell your individual story with the abstractions Gradle (or you) provide. Gradle provides some standard stories but they are not privileged in any form. This is for us a major distinguishing feature compared to other declarative build systems. Our Groovy support is not just sugar coating. The whole Gradle API is fully Groovy-ized. Adding Groovy results in an enjoyable and productive experience.

The Gradle wrapper

The Gradle Wrapper allows you to execute Gradle builds on machines where Gradle is not installed. This is useful for example for some continuous integration servers. It is also useful for an open source project to keep the barrier low for building it. The wrapper is also very interesting for the enterprise. It is a zero administration approach for the client machines. It also enforces the usage of a particular Gradle version thus minimizing support issues.

Free and open source

Gradle is an open source project, and is licensed under the [Apache License 2.0](https://github.com/gradle/gradle/blob/master/LICENSE).

[Why Groovy?](#3jtnz0s)

We think the advantages of an internal DSL (based on a dynamic language) over XML are tremendous when used in *build scripts*. There are a couple of dynamic languages out there. Why Groovy? The answer lies in the context Gradle is operating in. Although Gradle is a general purpose build tool at its core, its main focus are Java projects. In such projects the team members will be very familiar with Java. We think a build should be as transparent as possible to *all* team members.

In that case, you might argue why we don’t just use Java as the language for build scripts. We think this is a valid question. It would have the highest transparency for your team and the lowest learning curve, but because of the limitations of Java, such a build language would not be as nice, expressive and powerful as it could be.[[1](#1yyy98l)] Languages like Python, Groovy or Ruby do a much better job here. We have chosen Groovy as it offers by far the greatest transparency for Java people. Its base syntax is the same as Java’s as well as its type system, its package structure and other things. Groovy provides much more on top of that, but with the common foundation of Java.

For Java developers with Python or Ruby knowledge or the desire to learn them, the above arguments don’t apply. The Gradle design is well-suited for creating another build script engine in JRuby or Jython. It just doesn’t have the highest priority for us at the moment. We happily support any community effort to create additional build script engines.

[Getting Started](#17dp8vu)

[Installing Gradle](#3rdcrjn)

You can install the Gradle build tool on Linux, macOS, or Windows. This document covers installing using a package manager like SDKMAN!, Homebrew, or Scoop, as well as manual installation.

Use of the [Gradle Wrapper](#4iylrwe) is the recommended way to upgrade Gradle.

You can find all releases and their checksums on the [releases page](https://gradle.org/releases).

[Prerequisites](#2y3w247)

Gradle runs on all major operating systems and requires only a [Java JDK](http://www.oracle.com/technetwork/java/javase/downloads/index.html) version 7 or higher to run. To check, run java -version. You should see something like this:

❯ java -version  
java version "1.8.0\_151"  
Java(TM) SE Runtime Environment (build 1.8.0\_151-b12)  
Java HotSpot(TM) 64-Bit Server VM (build 25.151-b12, mixed mode)

Gradle ships with its own Groovy library, therefore Groovy does not need to be installed. Any existing Groovy installation is ignored by Gradle.

Gradle uses whatever JDK it finds in your path. Alternatively, you can set the JAVA\_HOME environment variable to point to the installation directory of the desired JDK.

[Installing with a package manager](#1d96cc0)

[SDKMAN!](http://sdkman.io) is a tool for managing parallel versions of multiple Software Development Kits on most Unix-based systems.

❯ sdk install gradle

[Homebrew](http://brew.sh) is "the missing package manager for macOS".

❯ brew install gradle

[Scoop](http://scoop.sh) is a command-line installer for Windows inspired by Homebrew.

❯ scoop install gradle

[Chocolatey](https://chocolatey.org) is "the package manager for Windows".

❯ choco install gradle

[MacPorts](https://www.macports.org) is a system for managing tools on macOS:

❯ sudo port install gradle

[↓ Proceed to next steps](#3x8tuzt)

[Installing manually](#2ce457m)

[Step 1.](#rjefff) [Download](https://gradle.org/releases) the latest Gradle distribution

The distribution ZIP file comes in two flavors:

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

Need to work with an older version? See the [releases page](https://gradle.org/releases).

[Step 2. Unpack the distribution](#3bj1y38)

[Linux & MacOS users](#1qoc8b1)

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

❯ mkdir /opt/gradle  
❯ unzip -d /opt/gradle gradle-4.10.1-bin.zip  
❯ ls /opt/gradle/gradle-4.10.1  
LICENSE NOTICE bin getting-started.html init.d lib media

[Microsoft Windows users](#4anzqyu)

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-4.10.1 to your newly created C:\Gradle folder.

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

[Step 3. Configure your system environment](#2pta16n)

For running Gradle, firstly add the environment variable GRADLE\_HOME. This should point to the unpacked files from the Gradle website. Next add *GRADLE\_HOME*/bin to your PATH environment variable. Usually, this is sufficient to run Gradle.

[Linux & MacOS users](#14ykbeg)

Configure your PATH environment variable to include the bin directory of the unzipped distribution, e.g.:

❯ export PATH=$PATH:/opt/gradle/gradle-4.10.1/bin

[Microsoft Windows users](#3oy7u29)

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-4.10.1\bin. Click OK to save.

[↓ Proceed to next steps](#3x8tuzt)

[Verifying installation](#243i4a2)

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

❯ gradle -v  
  
------------------------------------------------------------  
Gradle 4.10.1  
------------------------------------------------------------  
  
Build time: 2018-02-21 15:28:42 UTC  
Revision: 819e0059da49f469d3e9b2896dc4e72537c4847d  
  
Groovy: 2.4.15  
Ant: Apache Ant(TM) version 1.9.9 compiled on February 2 2017  
JVM: 1.8.0\_151 (Oracle Corporation 25.151-b12)  
OS: Mac OS X 10.13.3 x86\_64

If you run into any trouble, see the [section on troubleshooting installation](#j8sehv).

You can verify the integrity of the Gradle distribution by downloading the SHA-256 file (available from the [releases page](https://gradle.org/releases)) and following these [verification instructions](#338fx5o).

[Next steps](#3x8tuzt)

Now that you have Gradle installed, use these resources for getting started:

* Create your first Gradle project by following the [Creating New Gradle Builds](https://guides.gradle.org/creating-new-gradle-builds/) tutorial.
* Sign up for a [live introductory Gradle training](https://gradle.org/training/intro-to-gradle/) with a core engineer.
* Learn how to achieve common tasks through the [command-line interface](#lnxbz9).
* [Configure Gradle execution](#35nkun2), such as use of an HTTP proxy for downloading dependencies.
* Subscribe to the [Gradle Newsletter](https://newsletter.gradle.com/) for monthly release and community updates.

[Using Gradle Builds](#26in1rg)

[Command-Line Interface](#lnxbz9)

The command-line interface is one of the primary methods of interacting with Gradle. The following serves as a reference of executing and customizing Gradle use of a command-line or when writing scripts or configuring continuous integration.

Use of the [Gradle Wrapper](#3j2qqm3) is highly encouraged. You should substitute ./gradlew or gradlew.bat for gradle in all following examples when using the Wrapper.

Executing Gradle on the command-line conforms to the following structure. Options are allowed before and after task names.

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

If multiple tasks are specified, they should be separated with a space.

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

--console=plain

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

--build-cache  
--no-build-cache

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

--help  
-h

| **Note** | Many command-line flags can be specified in gradle.properties to avoid needing to be typed. See the [configuring build environment guide](#1idq7dh) for details. |
| --- | --- |

The following sections describe use of the Gradle command-line interface, grouped roughly by user goal. Some plugins also add their own command line options, for example [--tests for Java test filtering](#42ddq1a). For more information on exposing command line options for your own tasks, see [Declaring and using command-line options](#2hio093).

[Executing tasks](#wnyagw)

You can run a task and all of its [dependencies](#3gnlt4p).

❯ gradle myTask

You can learn about what projects and tasks are available in the [project reporting section](#1vsw3ci).

[Executing tasks in multi-project builds](#4fsjm0b)

In a [multi-project build](#z337ya), subproject tasks can be executed with ":" separating subproject name and task name. The following are equivalent *when run from the root project*.

❯ gradle :mySubproject:taskName  
❯ gradle mySubproject:taskName

You can also run a task for all subprojects using the task name only. For example, this will run the "test" task for all subprojects when invoked from the root project directory.

❯ gradle test

When invoking Gradle from within a subproject, the project name should be omitted:

❯ cd mySubproject  
❯ gradle taskName

| **Note** | When executing the Gradle Wrapper from subprojects, one must reference gradlew relatively. For example: ../gradlew taskName. The community [gdub project](http://www.gdub.rocks/) aims to make this more convenient. |
| --- | --- |

[Executing multiple tasks](#2uxtw84)

You can also specify multiple tasks. 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

[Excluding tasks from execution](#1a346fx)

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.



*Figure 1. Example Task Graph*

[Example: Excluding tasks](#3u2rp3q)

**Output of** gradle dist --exclude-task test

> gradle dist --exclude-task test  
  
> Task :compile  
compiling source  
  
> Task :dist  
building the distribution  
  
BUILD SUCCESSFUL in 0s  
2 actionable tasks: 2 executed

You can see that the test task is not executed, even though it is a dependency of the dist task. The test task’s dependencies such as compileTest are not executed either. Those dependencies of test that are required by another task, such as compile, are still executed.

[Forcing tasks to execute](#2981zbj)

You can force Gradle to execute all tasks ignoring [up-to-date checks](#odc9jc) using the --rerun-tasks option:

❯ gradle test --rerun-tasks

This will force test and *all* task dependencies of test to execute. It’s a little like running gradle clean test, but without the build’s generated output being deleted.

[Continuing the build when a failure occurs](#38czs75)

By default, Gradle will abort execution and fail the build as soon as any task fails. This allows the build to complete sooner, but hides other failures that would have occurred. In order to discover as many failures as possible in a single build execution, you can use the --continue option.

❯ gradle test --continue

When executed with --continue, Gradle will execute *every* task to be executed where all of the dependencies for that task completed without failure, instead of stopping as soon as the first failure is encountered. Each of the encountered failures will be reported at the end of the build.

If a task fails, any subsequent tasks that were depending on it will not be executed. For example, tests will not run if there is a compilation failure in the code under test; because the test task will depend on the compilation task (either directly or indirectly).

[Task name abbreviation](#1nia2ey)

When you specify tasks on the command-line, you don’t have to provide the full name of the task. You only need to provide enough of the task name to uniquely identify the task. For example, it’s likely gradle che is enough for Gradle to identify the check task.

You can also abbreviate each word in a camel case task name. For example, you can execute task compileTest by running gradle compTest or even gradle cT.

[Example: Abbreviated camel case task name](#47hxl2r)

**Output of** gradle cT

> gradle cT  
  
> Task :compile  
compiling source  
  
> Task :compileTest  
compiling unit tests  
  
BUILD SUCCESSFUL in 0s  
2 actionable tasks: 2 executed

You can also use these abbreviations with the -x command-line option.

[Common tasks](#2mn7vak)

The following are task conventions applied by built-in and most major Gradle plugins.

[Computing all outputs](#11si5id)

It is common in Gradle builds for the build task to designate assembling all outputs and running all checks.

❯ gradle build

[Running applications](#3ls5o66)

It is common for applications to be run with the run task, which assembles the application and executes some script or binary.

❯ gradle run

[Running all checks](#20xfydz)

It is common for *all* verification tasks, including tests and linting, to be executed using the check task.

❯ gradle check

[Cleaning outputs](#4kx3h1s)

You can delete the contents of the build directory using the clean task, though doing so will cause pre-computed outputs to be lost, causing significant additional build time for the subsequent task execution.

❯ gradle clean

[Project reporting](#1vsw3ci)

Gradle provides several built-in tasks which show particular details of your build. This can be useful for understanding the structure and dependencies of your build, and for debugging problems.

You can get basic help about available reporting options using gradle help.

[Listing projects](#302dr9l)

Running gradle projects gives you a list of the sub-projects of the selected project, displayed in a hierarchy.

❯ gradle projects

You also get a project report within build scans. Learn more about [creating build scans](https://guides.gradle.org/creating-build-scans/).

[Listing tasks](#1f7o1he)

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 which have been assigned to a task group. You can obtain more information in the task listing using the --all option.

❯ gradle tasks --all

[Show task usage details](#3z7bk57)

Running gradle help --task someTask gives you detailed information about a specific task.

[Example: Obtaining detailed help for tasks](#2eclud0)

**Output of** gradle -q help --task libs

> gradle -q help --task libs  
Detailed task information for libs  
  
Paths  
 :api:libs  
 :webapp:libs  
  
Type  
 Task (org.gradle.api.Task)  
  
Description  
 Builds the JAR  
  
Group  
 build

This information includes the full task path, the task type, possible command line options and the description of the given task.

[Reporting dependencies](#thw4kt)

Build scans give a full, visual report of what dependencies exist on which configurations, transitive dependencies, and dependency version selection.

❯ gradle myTask --scan

This will give you a link to a web-based report, where you can find dependency information like this.



Learn more in [Inspecting Dependencies](#nmf14n).

[Listing project dependencies](#3dhjn8m)

Running gradle dependencies 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

Concrete examples of build scripts and output available in the [Inspecting Dependencies](#nmf14n).

Running gradle buildEnvironment visualises the buildscript dependencies of the selected project, similarly to how gradle dependencies visualizes the dependencies of the software being built.

❯ gradle buildEnvironment

Running gradle dependencyInsight gives you an insight into a particular dependency (or dependencies) that match specified input.

❯ gradle dependencyInsight

Since a dependency report can get large, it can be useful to restrict the report to a particular configuration. This is achieved with the optional --configuration parameter:

[Listing project properties](#1smtxgf)

Running gradle properties gives you a list of the properties of the selected project.

[Example: Information about properties](#4cmhg48)

**Output of gradle -q api:properties**

> 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

[Software Model reports](#2rrrqc1)

You can get a hierarchical view of elements for [software model](#3q5sasy) projects using the model task:

❯ gradle model

Learn more about [the model report](#16x20ju) in the software model documentation.

[Command-line completion](#3qwpj7n)

Gradle provides bash and zsh tab completion support for tasks, options, and Gradle properties through [gradle-completion](https://github.com/gradle/gradle-completion), installed separately.

[Debugging options](#261ztfg)

-?, -h, --help

Shows a help message with all available CLI options.

-v, --version

Prints Gradle, Groovy, Ant, JVM, and operating system version information.

-S, --full-stacktrace

Print out the full (very verbose) stacktrace for any exceptions. See also [logging options](#l7a3n9).

-s, --stacktrace

Print out the stacktrace also for user exceptions (e.g. compile error). See also [logging options](#l7a3n9).

--scan

Create a [build scan](https://gradle.com/build-scans) with fine-grained information about all aspects of your Gradle build.

-Dorg.gradle.debug=true

Debug Gradle client (non-Daemon) process. Gradle will wait for you to attach a debugger at localhost:5005 by default.

-Dorg.gradle.daemon.debug=true

Debug [Gradle Daemon](#44sinio) process.

[Performance options](#356xmb2)

Try these options when optimizing build performance. Learn more about [improving performance of Gradle builds here](https://guides.gradle.org/performance/).

Many of these options can be specified in gradle.properties so command-line flags are not necessary. See the [configuring build environment guide](#1idq7dh).

--build-cache, --no-build-cache

Toggles the [Gradle build cache](#2bn6wsx). Gradle will try to reuse outputs from previous builds. *Default is off*.

--configure-on-demand, --no-configure-on-demand

Toggles [Configure-on-demand](#1kc7wiv). Only relevant projects are configured in this build run. *Default is off*.

--max-workers

Sets 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](#44bvf6o). *Default is off*.

--profile

Generates a high-level performance report in the $buildDir/reports/profile directory. --scan is preferred.

--scan

Generate a build scan with detailed performance diagnostics.



[Gradle daemon options](#2jh5peh)

You can manage the [Gradle Daemon](#44sinio) through the following command line options.

--daemon, --no-daemon

Use the [Gradle Daemon](#44sinio) to run the build. Starts the daemon if not running or existing daemon 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. 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)

Gradle Daemon will stop itself after this number of milliseconds of idle time. *Default is 10800000* (3 hours).

[Logging options](#l7a3n9)

[Setting log level](#ymfzma)

You can customize the verbosity of Gradle logging with the following options, ordered from least verbose to most verbose. Learn more in the [logging documentation](#3o7alnk).

-Dorg.gradle.logging.level=(quiet,warn,lifecycle,info,debug)

Set logging level via Gradle properties.

-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](#3im3ia3)

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)

Specify console mode via Gradle properties. Different modes 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 the rich, but output task names and outcomes at the lifecycle log level, as is done by default in Gradle 3.5 and earlier.

[Showing or hiding warnings](#1xrdshw)

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,none,summary)

Specify warning mode via [Gradle properties](#1idq7dh). Different modes described immediately below.

--warning-mode=(all,none,summary)

Specifies how to log warnings. Default is summary.

Set to all to log all 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](#4hr1b5p)

Gradle’s rich console displays extra information while builds are running.



Features:

* Progress bar and timer visually describe overall status
* Parallel work-in-progress lines below describe what is happening now
* Colors and fonts are used to highlight important output and errors

[Execution options](#2wwbldi)

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. See [Composite Builds](#49x2ik5).

--offline

Specifies that the build should operate without accessing network resources. Learn more about [options to override dependency caching](#1c1lvlb).

--refresh-dependencies

Refresh the state of dependencies. Learn more about how to use this in the [dependency management docs](#1c1lvlb).

--dry-run

Run Gradle with all task actions disabled. Use this to show which task would have executed.

--write-locks

Indicates that all resolved configurations that are *lockable* should have their lock state persisted. Learn more about this in [dependency locking](#46r0co2).

--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. Learn more about this in [dependency locking](#46r0co2).

[Environment options](#3w19e94)

You can customize many aspects about where build scripts, settings, caches, and so on through the options below. Learn more about customizing your [build environment](#35nkun2).

-b, --build-file

Specifies the build file. For example: gradle --build-file=foo.gradle. The default is build.gradle, then build.gradle.kts, then myProjectName.gradle.

-c, --settings-file

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.

-u, --no-search-upward (deprecated)

Don’t search in parent directories for a settings.gradle file.

-D, --system-prop

Sets a system property of the JVM, for example -Dmyprop=myvalue. See [System Properties](#2b6jogx).

-I, --init-script

Specifies an initialization script. See [Init Scripts](#2jxsxqh).

-P, --project-prop

Sets a project property of the root project, for example -Pmyprop=myvalue. See [System Properties](#qbtyoq).

-Dorg.gradle.jvmargs

Set JVM arguments.

-Dorg.gradle.java.home

Set JDK home dir.

[Bootstrapping new projects](#3abhhcj)

[Creating new Gradle builds](#1pgrrkc)

Use the built-in gradle init task to create a new Gradle builds, with new or existing projects.

❯ gradle init

Most of the time you’ll want to specify a project type. Available types include basic (default), java-library, java-application, and more. See [init plugin documentation](#qsh70q) for details.

❯ gradle init --type java-library

[Standardize and provision Gradle](#49gfa85)

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=4.4

You can also specify --distribution-type=(bin|all), --gradle-distribution-url, --gradle-distribution-sha256-sum in addition to --gradle-version. Full details on how to use these options are documented in the [Gradle wrapper section](#3j2qqm3).

[Continuous Build](#2olpkfy)

Continuous Build allows you to automatically re-execute the requested tasks when task inputs change.

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 that unrelated changes (such as changes to build scripts) will not trigger a rebuild. In order to incorporate build logic changes, the continuous build must be restarted manually.

[Terminating Continuous Build](#13qzunr)

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 and quirks](#3nqndbk)

| **Note** | Continuous build is an [incubating](#2xcytpi) feature. |
| --- | --- |

There are several issues to be aware with the current implementation of continuous build. These are likely to be addressed in future Gradle releases.

[Build cycles](#22vxnjd)

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.

[Restrictions with Java 9](#i17xr6)

Due to class access restrictions related to Java 9, Gradle cannot set some operating system specific options, which means that:

* On macOS, Gradle will poll for file changes every 10 seconds instead of every 2 seconds.
* On Windows, Gradle must use individual file watches (like on Linux/Mac OS), which may cause continuous build to no longer work on very large projects.

[Performance and stability](#320vgez)

The JDK file watching facility relies on inefficient file system polling on macOS (see: [JDK-7133447](https://bugs.openjdk.java.net/browse/JDK-7133447)). This can significantly delay notification of changes on large projects with many source files.

Additionally, the watching mechanism may deadlock under *heavy* load on macOS (see: [JDK-8079620](https://bugs.openjdk.java.net/browse/JDK-8079620)). This will manifest as Gradle appearing not to notice file changes. If you suspect this is occurring, exit continuous build and start again.

On Linux, OpenJDK’s implementation of the file watch service can sometimes miss file system events (see: [JDK-8145981](https://bugs.openjdk.java.net/browse/JDK-8145981)).

[Changes to symbolic links](#1h65qms)

* Creating or removing symbolic link to files will initiate a build.
* Modifying the target of a symbolic link will not cause a rebuild.
* Creating or removing symbolic links to directories will not cause rebuilds.
* Creating new files in the target directory of a symbolic link will not cause a rebuild.
* Deleting the target directory will not cause a rebuild.

[Changes to build logic are not considered](#415t9al)

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.

[Build Environment](#35nkun2)

Gradle provides multiple mechanisms for configuring behavior of Gradle itself and specific projects. The following is a reference for using these mechanisms.

When configuring Gradle behavior you can use these methods, listed in order of highest to lowest precedence (first one wins):

* [Command-line flags](#lnxbz9) such as --build-cache. These have precedence over properties and environment variables.
* [System properties](#2b6jogx) such as systemProp.http.proxyHost=somehost.org stored in a gradle.properties file.
* [Gradle properties](#1idq7dh) such as org.gradle.caching=true that are typically stored in a gradle.properties file in a project root directory or GRADLE\_USER\_HOME environment variable.
* [Environment variables](#2gb3jie) such as GRADLE\_OPTS sourced by the environment that executes Gradle.

Aside from configuring the build environment, you can configure a given project build using [Project properties](#qbtyoq) such as -PreleaseType=final.

[Gradle properties](#1idq7dh)

Gradle provides several options that make it easy to configure the Java process that will be used to execute your build. While it’s possible to configure these in your local environment via GRADLE\_OPTS or JAVA\_OPTS, it is useful to store certain settings like JVM memory configuration and Java home location in version control so that an entire team can work with a consistent environment.

Setting up a consistent environment for your build is as simple as placing these settings into a gradle.properties file. The configuration is applied in following order (if an option is configured in multiple locations the *last one wins*):

* gradle.properties in project root directory.
* gradle.properties in GRADLE\_USER\_HOME directory.
* system properties, e.g. when -Dgradle.user.home is set on the command line.

The following properties can be used to configure the Gradle build environment:

org.gradle.caching=(true,false)

When set to true, Gradle will reuse task outputs from any previous build, when possible, resulting is much faster builds. Learn more about [using the build cache](#2bn6wsx).

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. Learn more about [task output caching](#vgdtq7).

org.gradle.configureondemand=(true,false)

Enables incubating [configuration on demand](#1kc7wiv), where Gradle will attempt to configure only necessary projects.

org.gradle.console=(auto,plain,rich,verbose)

Customize console output coloring or verbosity. Default depends on how Gradle is invoked. See [command-line logging](#l7a3n9) for additional details.

org.gradle.daemon=(true,false)

When set to true the [Gradle Daemon](#44sinio) is used to run the build. Default is true.

org.gradle.daemon.idletimeout=(# of idle millis)

Gradle Daemon will terminate itself after 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 the equivalent of 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, depending on what your build does, using a JDK is safer. A reasonable default is used 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](#3fg1ce0) for build performance.

org.gradle.logging.level=(quiet,warn,lifecycle,info,debug)

When set to quiet, warn, lifecycle, info, or debug, Gradle will use this log level. The values are not case sensitive. The lifecycle level is the default. See [Choosing a log level](#1ulbmlt).

org.gradle.parallel=(true,false)

When configured, Gradle will fork up to org.gradle.workers.max JVMs to execute projects in parallel. To learn more about parallel task execution, see [the Gradle performance guide](https://guides.gradle.org/performance/#parallel_execution).

org.gradle.warning.mode=(all,none,summary)

When set to all, summary or none, Gradle will use different warning type display. See [Command-line logging options](#l7a3n9) for details.

org.gradle.workers.max=(max # of worker processes)

When configured, Gradle will use a maximum of the given number of workers. Default is number of CPU processors. See also [performance command-line options](#356xmb2).

The following example demonstrates usage of various properties.

[Example: Setting properties with a gradle.properties file](#4ekz59m)

**gradle.properties**

gradlePropertiesProp=gradlePropertiesValue  
sysProp=shouldBeOverWrittenBySysProp  
systemProp.system=systemValue

**build.gradle**

task printProps {  
 doLast {  
 println commandLineProjectProp  
 println gradlePropertiesProp  
 println systemProjectProp  
 println System.properties['system']  
 }  
}

**Output of gradle -q -PcommandLineProjectProp=commandLineProjectPropValue -Dorg.gradle.project.systemProjectProp=systemPropertyValue printProps**

> gradle -q -PcommandLineProjectProp=commandLineProjectPropValue -Dorg.gradle.project.systemProjectProp=systemPropertyValue printProps  
commandLineProjectPropValue  
gradlePropertiesValue  
systemPropertyValue  
systemValue

[System properties](#2b6jogx)

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.

You can also set system properties in gradle.properties files with the prefix systemProp.

[Example: Specifying system properties in gradle.properties](#2tq9fhf)

systemProp.gradle.wrapperUser=myuser  
systemProp.gradle.wrapperPassword=mypassword

The following system properties are available. Note that command-line options take precedence over system properties.

gradle.wrapperUser=(myuser)

Specify user name to download Gradle distributions from servers using HTTP Basic Authentication. Learn more in [Authenticated wrapper downloads](#18vjpp8).

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.

In a multi project build, “systemProp.” properties set in any project except the root will be ignored. That is, only the root project’s gradle.properties file will be checked for properties that begin with the “systemProp.” prefix.

[Environment variables](#2gb3jie)

The following environment variables are available for the gradle command. Note that command-line options and system properties take precedence over environment variables.

GRADLE\_OPTS

Specifies [command-line arguments](#lnxbz9) to use when starting the Gradle client. This can be useful for setting the properties to use when running Gradle.

GRADLE\_USER\_HOME

Specifies the Gradle user home directory (which defaults to $USER\_HOME/.gradle if not set).

JAVA\_HOME

Specifies the JDK installation directory to use.

[Project properties](#qbtyoq)

You can add properties directly to your [Project](http://docs.google.com/dsl/org.gradle.api.Project.html) object via the -P command line option.

Gradle can also set project properties when it sees specially-named system properties or environment variables. If the environment variable name looks like ORG\_GRADLE\_PROJECT*\_prop*=somevalue, then Gradle will set a prop property on your project object, with the value of somevalue. Gradle also supports this for system properties, but with a different naming pattern, which looks like org.gradle.project.*prop*. Both of the following will set the foo property on your Project object to "bar".

[Example: Setting a project property via gradle.properties](#3sv78d1)

org.gradle.project.foo=bar

[Example: Setting a project property via environment variable](#280hiku)

ORG\_GRADLE\_PROJECT\_foo=bar

| **Note** | The properties file in the user’s home directory has precedence over property files in the project directories. |
| --- | --- |

This feature is very useful when you don’t have admin rights to a continuous integration server and you need to set property values that should not be easily visible. Since you cannot use the -P option in that scenario, nor change the system-level configuration files, the correct strategy is to change the configuration of your continuous integration build job, adding an environment variable setting that matches an expected pattern. This won’t be visible to normal users on the system.

You can access a project property in your build script simply by using its name as you would use a variable.

| **Note** | If a project property is referenced but does not exist, an exception will be thrown and the build will fail.  You should check for existence of optional project properties before you access them using the [Project.hasProperty(java.lang.String)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:hasProperty(java.lang.String)) method. |
| --- | --- |

[Configuring JVM memory](#3fg1ce0)

Gradle defaults to 1024 megabytes maximum heap per JVM process (-Xmx1024m), however, that may be too much or too little depending on the size of your project. There are many JVM options (this [blog post on Java performance tuning](https://dzone.com/articles/java-performance-tuning) and [this reference](http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html) may be helpful).

You can adjust JVM options for Gradle in the following ways:

The JAVA\_OPTS environment variable is used for the Gradle client, but not forked JVMs.

[Example: Changing JVM settings for Gradle client JVM](#n5rssn)

JAVA\_OPTS="-Xmx2g -XX:MaxPermSize=256m -XX:+HeapDumpOnOutOfMemoryError -Dfile.encoding=UTF-8"

You need to use the org.gradle.jvmargs Gradle property to configure JVM settings for the [Gradle Daemon](#44sinio).

[Example: Changing JVM settings for forked Gradle JVMs](#375fbgg)

org.gradle.jvmargs=-Xmx2g -XX:MaxPermSize=256m -XX:+HeapDumpOnOutOfMemoryError -Dfile.encoding=UTF-8

| **Note** | Many settings (like the Java version and maximum heap size) can only be specified when launching a new JVM for the build process. This means that Gradle must launch a separate JVM process to execute the build after parsing the various gradle.properties files.  When running with the [Gradle Daemon](#44sinio), a JVM with the correct parameters is started once and reused for each daemon build execution. When Gradle is executed without the daemon, then a new JVM must be launched for every build execution, unless the JVM launched by the Gradle start script happens to have the same parameters. |
| --- | --- |

Certain tasks in Gradle also fork additional JVM processes, like the test task when using [Test.setMaxParallelForks(int)](http://docs.google.com/javadoc/org/gradle/api/tasks/testing/Test.html#setMaxParallelForks-int-) for JUnit or TestNG tests. You must configure these through the tasks themselves.

[Example: Set Java compile options for](#1maplo9) [JavaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.JavaCompile.html) tasks

**build.gradle**

apply plugin: "java"  
  
tasks.withType(JavaCompile) {  
 options.compilerArgs += ["-Xdoclint:none", "-Xlint:none", "-nowarn"]  
}

See other examples in the [Test](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html) API documentation and [test execution in the Java plugin reference](#46ad4c2).

[Build scans](https://scans.gradle.com) will tell you information about the JVM that executed the build when you use the --scan option.



[Configuring a task using project properties](#2lfnejv)

It’s 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.

[Example: Prevent releasing outside of CI](#10kxoro)

**build.gradle**

task performRelease {  
 doLast {  
 **if** (project.hasProperty("isCI")) {  
 println("Performing release actions")  
 } **else** {  
 **throw** **new** InvalidUserDataException("Cannot perform release outside of CI")  
 }  
 }  
}

**Output of** gradle performRelease -PisCI=true --quiet

> gradle performRelease -PisCI=true --quiet  
Performing release actions

[Accessing the web through a HTTP proxy](#3kkl7fh)

Configuring an HTTP or HTTPS proxy (for downloading dependencies, for example) is done via standard JVM system properties. These properties can be set directly in the build script; for example, setting the HTTP proxy host would be done with System.setProperty('http.proxyHost', 'www.somehost.org'). Alternatively, the properties can be [specified in gradle.properties](#1idq7dh).

**Configuring an HTTP proxy using gradle.properties**

systemProp.http.proxyHost=www.somehost.org  
systemProp.http.proxyPort=8080  
systemProp.http.proxyUser=userid  
systemProp.http.proxyPassword=password  
systemProp.http.nonProxyHosts=\*.nonproxyrepos.com|localhost

There are separate settings for HTTPS.

**Configuring an HTTPS proxy using gradle.properties**

systemProp.https.proxyHost=www.somehost.org  
systemProp.https.proxyPort=8080  
systemProp.https.proxyUser=userid  
systemProp.https.proxyPassword=password  
systemProp.https.nonProxyHosts=\*.nonproxyrepos.com|localhost

You may need to set other properties to access other networks. Here are 2 references that may be helpful:

* [ProxySetup.java in the Ant codebase](https://git-wip-us.apache.org/repos/asf?p=ant.git;a%3Dblob;f%3Dsrc/main/org/apache/tools/ant/util/ProxySetup.java;hb%3DHEAD)
* [JDK 7 Networking Properties](http://download.oracle.com/javase/7/docs/technotes/guides/net/properties.html)

[NTLM Authentication](#1zpvhna)

If your proxy requires NTLM authentication, you may need to provide the authentication domain as well as the username and password. There are 2 ways that you can provide the domain for authenticating to a NTLM proxy:

* Set the http.proxyUser system property to a value like *domain*/*username*.
* Provide the authentication domain via the http.auth.ntlm.domain system property.

[Directory Layout](#1ksv4uv)

Gradle uses two main directories to perform and manage its work: the [Gradle user home directory](#4jpj0b3) and the [Project root directory](#2yutaiw). The following two sections describe what is stored in each of them and how transient files and directories are cleaned up.

[Gradle user home directory](#4jpj0b3)

The Gradle user home directory ($USER\_HOME/.gradle by default) is used to store global configuration properties and initialization scripts as well as caches and log files. 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  
├── wrapper  
│ └── dists (6)  
│ ├── ⋮  
│ ├── gradle-4.8-bin  
│ ├── gradle-4.9-all  
│ └── gradle-4.9-bin  
└── gradle.properties (7)

1. Global cache directory (for everything that’s 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](#44sinio)
5. Global [initialization scripts](#2jxsxqh)
6. Distributions downloaded by the [Gradle Wrapper](#3j2qqm3)
7. Global [Gradle configuration properties](#1idq7dh)

[Cleanup of caches and distributions](#1e03kqp)

From version 4.10 onwards, Gradle automatically cleans its user home directory. The cleanup runs in the background when the Gradle daemon is stopped or shuts down. If using --no-daemon, it runs in the foreground after the build session with a visual progress indicator.

The following cleanup strategies are applied periodically (at most every 24 hours):

* Version-specific caches in caches/<gradle-version>/ are checked for whether they are still in use. If not, directories for release versions are deleted after 30 days of inactivity, snapshot versions after 7 days of inactivity.
* Shared caches in caches/ (e.g. jars-\*) are checked for whether they are still in use. If there’s no Gradle version that 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 would have to be downloaded from a remote repository again, it will be deleted after 7 or 30 days of not being accessed, 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.

[Project root directory](#2yutaiw)

The project root directory contains all source files that are part of your project. In addition, it contains files and directories that are generated by Gradle such as .gradle and build. While the former are usually checked in to source control, the latter are transient files used by Gradle to support features like incremental builds. Overall, the anatomy of a typical project root directory looks roughly as follows:

├── .gradle (1)  
│   ├── 4.8 (2)  
│   ├── 4.9 (2)  
│   └── ⋮  
├── build (3)  
├── gradle  
│   └── wrapper (4)  
├── build.gradle *or* build.gradle.kts (5)  
├── gradle.properties (6)  
├── gradlew (7)  
├── gradlew.bat (7)  
└── settings.gradle *or* settings.gradle.kts (8)

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](#3j2qqm3)
5. The project’s Gradle build script
6. Project-specific [Gradle configuration properties](#1idq7dh)
7. Scripts for executing builds using the [Gradle Wrapper](#3j2qqm3)
8. The project’s [settings file](#3xzr3ei)

[Project cache cleanup](#2d51dmb)

From version 4.10 onwards, Gradle automatically cleans the project-specific cache directory. After building the project, version-specific cache directories in .gradle/<gradle-version>/ 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.

[The Gradle Daemon](#44sinio)

*A daemon is a computer program that runs as a background process, rather than being under the direct control of an interactive user.*

*— Wikipedia*

Gradle runs on the Java Virtual Machine (JVM) and uses several supporting libraries that require a non-trivial initialization time. As a result, it can sometimes seem a little slow to start. The solution to this problem is the Gradle *Daemon*: a long-lived background process that executes your builds much more quickly than would otherwise be the case. We accomplish this by avoiding the expensive bootstrapping process as well as leveraging caching, by keeping data about your project in memory. Running Gradle builds with the Daemon is no different than without. Simply configure whether you want to use it or not - everything else is handled transparently by Gradle.

[Why the Gradle Daemon is important for performance](#sabnu4)

The Daemon is a long-lived process, so not only are we able to avoid the cost of JVM startup for every build, but we are able to cache information about project structure, files, tasks, and more in memory.

The reasoning is simple: improve build speed by reusing computations from previous builds. However, the benefits are dramatic: we typically measure build times reduced by 15-75% on subsequent builds. We recommend profiling your build by using --profile to get a sense of how much impact the Gradle Daemon can have for you.

The Gradle Daemon is enabled by default starting with Gradle 3.0, so you don’t have to do anything to benefit from it.

If you run CI builds in ephemeral environments (such as containers) that do not reuse any processes, use of the Daemon will slightly decrease performance (due to caching additional information) for no benefit, and may be disabled.

[Running Daemon Status](#3c9z6hx)

To get a list of running Gradle Daemons and their statuses use the --status command.

Sample output:

PID VERSION STATUS  
 28411 3.0 IDLE  
 34247 3.0 BUSY

Currently, a given Gradle version can only connect to daemons of the same version. This means the status output will only show Daemons for the version of Gradle being invoked and not for any other versions. Future versions of Gradle will lift this constraint and will show the running Daemons for all versions of Gradle.

[Disabling the Daemon](#1rf9gpq)

The Gradle Daemon is enabled by default, and we recommend always enabling it. There are several ways to disable the Daemon, but the most common one is to add the line

org.gradle.daemon=false

to the file «USER\_HOME»/.gradle/gradle.properties, where «USER\_HOME» is your home directory. That’s typically one of the following, depending on your platform:

* C:\Users\<username> (Windows Vista & 7+)
* /Users/<username> (macOS)
* /home/<username> (Linux)

If that file doesn’t exist, just create it using a text editor. You can find details of other ways to disable (and enable) the Daemon in [Daemon FAQ](#4bewzdj) further down. That section also contains more detailed information on how the Daemon works.

Note that having the Daemon enabled, all your builds will take advantage of the speed boost, regardless of the version of Gradle a particular build uses.

| **Tip** | Continuous integration  Since Gradle 3.0, we enable Daemon by default and recommend using it for both developers' machines and Continuous Integration servers. However, if you suspect that Daemon makes your CI builds unstable, you can disable it to use a fresh runtime for each build since the runtime is *completely* isolated from any previous builds. |
| --- | --- |

[Stopping an existing Daemon](#2qk79lc)

As mentioned, the Daemon is a background process. You needn’t worry about a build up of Gradle processes on your machine, though. Every Daemon monitors its memory usage compared to total system memory and will stop itself if idle when available system memory is low. If you want to explicitly stop running Daemon processes for any reason, just use the command gradle --stop.

This will terminate all Daemon processes that were started with the same version of Gradle used to execute the command. If you have the Java Development Kit (JDK) installed, you can easily verify that a Daemon has stopped by running the jps command. You’ll see any running Daemons listed with the name GradleDaemon.

[FAQ](#4bewzdj)

[How do I disable the Gradle Daemon?](#15phjt5)

There are two recommended ways to disable the Daemon persistently for an environment:

* Via environment variables: add the flag -Dorg.gradle.daemon=false to the GRADLE\_OPTS environment variable
* Via properties file: add org.gradle.daemon=false to the «GRADLE\_USER\_HOME»/gradle.properties file

| **Note** | Note, «GRADLE\_USER\_HOME» defaults to «USER\_HOME»/.gradle, where «USER\_HOME» is the home directory of the current user. This location can be configured via the -g and --gradle-user-home command line switches, as well as by the GRADLE\_USER\_HOME environment variable and org.gradle.user.home JVM system property. |
| --- | --- |

Both approaches have the same effect. Which one to use is up to personal preference. Most Gradle users choose the second option and add the entry to the user gradle.properties file.

On Windows, this command will disable 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 will disable the Daemon for the current user:

mkdir -p ~/.gradle && echo "org.gradle.daemon=false" >> ~/.gradle/gradle.properties

Once the Daemon is disabled for a build environment in this way, a Gradle Daemon will not be started unless explicitly requested using the --daemon option.

The --daemon and --no-daemon command line options enable and disable usage of the Daemon for individual build invocations when using the Gradle command line interface. These command line options have the *highest* precedence when considering the build environment. Typically, it is more convenient to enable the Daemon for an environment (e.g. a user account) so that all builds use the Daemon without requiring to remember to supply the --daemon option.

[Why is there more than one Daemon process on my machine?](#3pp52gy)

There are several reasons why Gradle will create a new Daemon, instead of using one that is already running. The basic rule is that Gradle will start a new Daemon if there are no existing idle or compatible Daemons available. Gradle will kill any Daemon that has been idle for 3 hours or more, so you don’t have to worry about cleaning them up manually.

idle

An idle Daemon is one that is not currently executing a build or doing other useful work.

compatible

A compatible Daemon is one that can (or can be made to) meet the requirements of the requested build environment. The Java runtime used to execute the build is an example aspect of the build environment. Another example is the set of JVM system properties required by the build runtime.

Some aspects of the requested build environment may not be met by an Daemon. If the Daemon is running with a Java 7 runtime, but the requested environment calls for Java 8, then the Daemon is not compatible and another must be started. Moreover, certain properties of a Java runtime cannot be changed once the JVM has started. For example, it is not possible to change the memory allocation (e.g. -Xmx1024m), default text encoding, default locale, etc of a running JVM.

The “requested build environment” is typically constructed implicitly from aspects of the build client’s (e.g. Gradle command line client, IDE etc.) environment and explicitly via command line switches and settings. See [Build Environment](#35nkun2) for details on how to specify and control the build environment.

The following JVM system properties are effectively immutable. If the requested build environment requires any of these properties, with a different value than a Daemon’s JVM has for this property, the Daemon is not compatible.

* 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 effectively immutable. The corresponding attributes of the requested build environment and the Daemon’s environment must match exactly in order for a Daemon to be compatible.

* The maximum heap size (i.e. the -Xmx JVM argument)
* The minimum heap size (i.e. the -Xms JVM argument)
* The boot classpath (i.e. the -Xbootclasspath argument)
* The “assertion” status (i.e. the -ea argument)

The required Gradle version is another aspect of the requested build environment. Daemon processes are coupled to a specific Gradle runtime. Working on multiple Gradle projects during a session that use different Gradle versions is a common reason for having more than one running Daemon process.

[How much memory does the Daemon use and can I give it more?](#24ufcor)

If the requested build environment does not specify a maximum heap size, the Daemon will use up to 1GB of heap. It will use the JVM’s default minimum heap size. 1GB is more than enough for most builds. Larger builds with hundreds of subprojects, lots of configuration, and source code may require, or perform better, with more memory.

To increase the amount of memory the Daemon can use, specify the appropriate flags as part of the requested build environment. Please see [Build Environment](#35nkun2) for details.

[How can I stop a Daemon?](#jzpmwk)

Daemon processes will automatically terminate themselves after 3 hours of inactivity or less. If you wish to stop a Daemon process before this, you can either kill the process via your operating system or run the gradle --stop command. The --stop switch causes Gradle to request that *all* running Daemon processes, *of the same Gradle version used to run the command*, terminate themselves.

[What can go wrong with Daemon?](#33zd5kd)

Considerable engineering effort has gone into making the Daemon robust, transparent and unobtrusive during day to day development. However, Daemon processes can occasionally be corrupted or exhausted. A Gradle build executes arbitrary code from multiple sources. While Gradle itself is designed for and heavily tested with the Daemon, user build scripts and third party plugins can destabilize the Daemon process through defects such as memory leaks or global state corruption.

It is also possible to destabilize the Daemon (and build environment in general) by running builds that do not release resources correctly. This is a particularly poignant problem when using Microsoft Windows as it is less forgiving of programs that fail to close files after reading or writing.

Gradle actively monitors heap usage and attempts to detect when a leak is starting to exhaust the available heap space in the daemon. When it detects a problem, the Gradle daemon will finish the currently running build and proactively restart the daemon on the next build. This monitoring is enabled by default, but can be disabled by setting the org.gradle.daemon.performance.enable-monitoring system property to false.

If it is suspected that the Daemon process has become unstable, it can simply be killed. Recall that the --no-daemon switch can be specified for a build to prevent use of the Daemon. This can be useful to diagnose whether or not the Daemon is actually the culprit of a problem.

[Tools & IDEs](#1j4nfs6)

The [Gradle Tooling API](#184mhaj) that is used by IDEs and other tools to integrate with Gradle *always* uses the Gradle Daemon to execute builds. If you are executing Gradle builds from within your IDE you are using the Gradle Daemon and do not need to enable it for your environment.

[How does the Gradle Daemon make builds faster?](#434ayfz)

The Gradle Daemon is a *long lived* build process. In between builds it waits idly for the next build. This has the obvious benefit of only requiring Gradle to be loaded into memory once for multiple builds, as opposed to once for each build. This in itself is a significant performance optimization, but that’s not where it stops.

A significant part of the story for modern JVM performance is runtime code optimization. For example, HotSpot (the JVM implementation provided by Oracle and used as the basis of OpenJDK) applies optimization to code while it is running. The optimization is progressive and not instantaneous. That is, the code is progressively optimized during execution which means that subsequent builds can be faster purely due to this optimization process. Experiments with HotSpot have shown that it takes somewhere between 5 and 10 builds for optimization to stabilize. The difference in perceived build time between the first build and the 10th for a Daemon can be quite dramatic.

The Daemon also allows more effective in memory caching across builds. For example, the classes needed by the build (e.g. plugins, build scripts) can be held in memory between builds. Similarly, Gradle can maintain in-memory caches of build data such as the hashes of task inputs and outputs, used for incremental building.

[Initialization Scripts](#2jxsxqh)

Gradle provides a powerful mechanism to allow customizing the build based on the current environment. This mechanism also supports tools that wish to integrate with Gradle.

Note that this is completely different from the “init” task provided by the “build-init” incubating plugin (see [Build Init Plugin](#qsh70q)).

[Basic usage](#2i9l8ns)

Initialization scripts (a.k.a. *init scripts*) are similar to other scripts in Gradle. These scripts, however, are run before the build starts. Here are several possible uses:

* Set up enterprise-wide configuration, such as where to find custom plugins.
* Set up properties based on the current environment, such as a developer’s machine vs. a continuous integration server.
* Supply personal information about the user that is required by the build, such as repository or database authentication credentials.
* Define machine specific details, such as where JDKs are installed.
* Register build listeners. External tools that wish to listen to Gradle events might find this useful.
* Register build loggers. You might wish to 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 (see [Using buildSrc to extract imperative logic](#xevivl) for details of this feature).

[Using an init script](#3hej1je)

There are several ways to use an init script:

* Specify a file on the command line. The command line option is -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 of the files specified on the command line does not exist.
* Put a file called init.gradle in the *USER\_HOME*/.gradle/ directory.
* Put a file that ends with .gradle in the *USER\_HOME*/.gradle/init.d/ directory.
* Put a file that ends with .gradle in the *GRADLE\_HOME*/init.d/ directory, in the Gradle distribution. This allows you to package up a custom Gradle distribution containing some custom build logic and plugins. You can combine this with the [Gradle wrapper](#3j2qqm3) as a way 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. This allows, for example, a tool to specify an init script on the command line and the user to put one in their home directory for defining the environment and both scripts will run when Gradle is executed.

[Writing an init script](#1wjtbr7)

Similar to a Gradle build script, an init script is a Groovy script. Each init script has a [Gradle](http://docs.google.com/dsl/org.gradle.api.invocation.Gradle.html) instance associated with it. Any property reference and method call in the init script will delegate to this Gradle instance.

Each init script also implements the [Script](http://docs.google.com/dsl/org.gradle.api.Script.html) interface.

[Configuring projects from an init script](#4gjguf0)

You can use an init script to configure the projects in the build. This works in a similar way 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. This sample uses this feature to configure an extra repository to be used only for certain environments.

[Example: Using init script to perform extra configuration before projects are evaluated](#2vor4mt)

**build.gradle**

repositories {  
 mavenCentral()  
}  
  
task showRepos {  
 doLast {  
 println "All repos:"  
 println repositories.collect { it.name }  
 }  
}

**init.gradle**

allprojects {  
 repositories {  
 mavenLocal()  
 }  
}

**Output of gradle --init-script init.gradle -q showRepos**

> gradle --init-script init.gradle -q showRepos  
All repos:  
[MavenLocal, MavenRepo]

[External dependencies for the init script](#1au1eum)

In [External dependencies for the build script](#3utoxif) it was explained how to add external dependencies to a build script. Init scripts can also declare dependencies. You do this with the initscript() method, passing in a closure which declares the init script classpath.

[Example: Declaring external dependencies for an init script](#29yz7q8)

**init.gradle**

initscript {  
 repositories {  
 mavenCentral()  
 }  
 dependencies {  
 classpath group: 'org.apache.commons', name: 'commons-math', version: '2.0'  
 }  
}

The closure passed to the initscript() method configures a [ScriptHandler](http://docs.google.com/javadoc/org/gradle/api/initialization/dsl/ScriptHandler.html) 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](#3tbugp1), 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.

[Example: An init script with external dependencies](#p49hy1)

**init.gradle**

**import** org.apache.commons.math.fraction.Fraction  
  
initscript {  
 repositories {  
 mavenCentral()  
 }  
 dependencies {  
 classpath group: 'org.apache.commons', name: 'commons-math', version: '2.0'  
 }  
}  
  
println Fraction.ONE\_FIFTH.multiply(2)

**Output of** gradle --init-script init.gradle -q doNothing

> gradle --init-script init.gradle -q doNothing  
2 / 5

[Init script plugins](#393x0lu)

Similar to a Gradle build script or a Gradle settings file, plugins can be applied on init scripts.

[Example: Using plugins in init scripts](#1o97atn)

**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()  
}  
  
 task showRepositories {  
 doLast {  
 repositories.each {  
 println "repository: ${it.name} ('${it.url}')"  
 }  
 }  
}

**Output of gradle --init-script init.gradle -q showRepositories**

> 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)](http://docs.google.com/javadoc/org/gradle/api/Plugin.html#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](#1au1eum)

[Executing Multi-Project Builds](#z337ya)

Only the smallest of projects has a single build file and source tree, unless it happens to be a massive, monolithic application. It’s often much easier to digest and understand a project that has been split into smaller, inter-dependent modules. The word “inter-dependent” is important, though, and is why you typically want to link the modules together through a single build.

Gradle supports this scenario through *multi-project* builds.

[Structure of a multi-project build](#488uthg)

Such builds come in all shapes and sizes, but they do have some common characteristics:

* A settings.gradle file in the root or master directory of the project
* A build.gradle file in the root or master directory
* Child directories that have their own \*.gradle build files (some multi-project builds may omit child project build scripts)

The settings.gradle file tells Gradle how the project and subprojects are structured. Fortunately, you don’t have to read this file simply to learn what the project structure is as you can run the command gradle projects. Here’s the output from using that command on the Java *multiproject* build in the Gradle samples:

[Example: Listing the projects in a build](#2ne53p9)

**Output of** gradle -q projects

> gradle -q projects  
  
------------------------------------------------------------  
Root project  
------------------------------------------------------------  
  
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

This tells you that *multiproject* has three immediate child projects: *api*, *services* and *shared*. The *services* project then has its own children, *shared* and *webservice*. These map to the directory structure, so it’s easy to find them. For example, you can find *webservice* in <root>/services/webservice.

By default, Gradle uses the name of the directory it finds the settings.gradle as the name of the root project. This usually doesn’t cause problems since all developers check out the same directory name when working on a project. On Continuous Integration servers, like Jenkins, the directory name may be auto-generated and not match the name in your VCS. For that reason, it’s recommended that you always set the root project name to something predictable, even in single project builds. You can configure the root project name by setting rootProject.name.

Each project will usually have its own build file, but that’s not necessarily the case. In the above example, the *services* project is just a container or grouping of other subprojects. There is no build file in the corresponding directory. However, *multiproject* does have one for the root project.

The root build.gradle is often used to share common configuration between the child projects, for example by applying the same sets of plugins and dependencies to all the child projects. It can also be used to configure individual subprojects when it is preferable to have all the configuration in one place. This means you should always check the root build file when discovering how a particular subproject is being configured.

Another thing to bear in mind is that the build files might not be called build.gradle. Many projects will name the build files after the subproject names, such as api.gradle and services.gradle from the previous example. Such an approach helps a lot in IDEs because it’s tough to work out which build.gradle file out of twenty possibilities is the one you want to open. This little piece of magic is handled by the settings.gradle file, but as a build user you don’t need to know the details of how it’s done. Just have a look through the child project directories to find the files with the .gradle suffix.

Once you know what subprojects are available, the key question for a build user is how to execute the tasks within the project.

[Executing a multi-project build](#12jfdx2)

From a user’s perspective, multi-project builds are still collections of tasks you can run. The difference is that you may want to control *which* project’s tasks get executed. You have two options here:

* Change to the directory corresponding to the subproject you’re interested in and just execute gradle <task> as normal.
* Use a qualified task name from any directory, although this is usually done from the root. For example: gradle :services:webservice:build will build the *webservice* subproject and any subprojects it depends on.

The first approach is similar to the single-project use case, but Gradle works slightly differently in the case of a multi-project build. The command gradle test will execute the test task in any subprojects, relative to the current working directory, that have that task. So if you run the command from the root project directory, you’ll run test in *api*, *shared*, *services:shared* and *services:webservice*. If you run the command from the services project directory, you’ll only execute the task in *services:shared* and *services:webservice*.

For more control over what gets executed, use qualified names (the second approach mentioned). These are paths just like directory paths, but use ‘:’ instead of ‘/’ or ‘\’. If the path begins with a ‘:’, then the path is resolved relative to the root project. In other words, the leading ‘:’ represents the root project itself. All other colons are path separators.

This approach works for any task, so if you want to know what tasks are in a particular subproject, just use the tasks task, e.g. gradle :services:webservice:tasks .

Regardless of which technique you use to execute tasks, Gradle will take care of building any subprojects that the target depends on. You don’t have to worry about the inter-project dependencies yourself. If you’re interested in how this is configured, you can read about writing multi-project builds [later in the user guide](#2p2csry).

There’s one last thing to note. When you’re using the Gradle wrapper, the first approach doesn’t work well because you have to specify the path to the wrapper script if you’re not in the project root. For example, if you’re in the *webservice* subproject directory, you would have to run ../../gradlew build.

That’s all you really need to know about multi-project builds as a build user. You can now identify whether a build is a multi-project one and you can discover its structure. And finally, you can execute tasks within specific subprojects.

[The Gradle Wrapper](#3j2qqm3)

The recommended way to execute any Gradle build is with the help of the Gradle Wrapper (in short just “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 without having to follow manual installation processes saving your company time and money.



*Figure 2. The Wrapper workflow*

In a nutshell you gain the following benefits:

* Standardizes a project on a given Gradle version, leading to more reliable and robust builds.
* Provisioning a new Gradle version to different users and execution environment (e.g. IDEs or Continuous Integration servers) is as simple as changing the Wrapper definition.

So how does it work? For a user there are typically three different workflows:

* You set up a new Gradle project and want to [add the Wrapper](#3mj2wkv) to it.
* You want to [run a project with the Wrapper](#21od6so) that already provides it.
* You want to [upgrade the Wrapper](#4iylrwe) to a new version of Gradle.

The following sections explain each of these use cases in more detail.

[Adding the Gradle Wrapper](#3mj2wkv)

Generating the Wrapper files requires an installed version of the Gradle runtime on your machine as described in [Installation](#3rdcrjn). Thankfully, generating the initial Wrapper files is a one-time process.

Every vanilla Gradle build comes with a built-in task called wrapper. You’ll be able to find the task listed under the group "Build Setup tasks" when [listing the tasks](#1f7o1he). Executing the wrapper task generates the necessary Wrapper files in the project directory.

[Example: Running the Wrapper task](#gtnh0h)

**Output of** gradle wrapper

> gradle wrapper  
> Task :wrapper  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

| **Note** | To make the Wrapper files available to other developers and execution environments you’ll need to check them into version control. All Wrapper files including the JAR file are very small in size. Adding the JAR file to version control is expected. Some organizations do not allow projects to submit binary files to version control. At the moment there are no alternative options to the approach. |
| --- | --- |

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 that’s the -bin distribution containing only the runtime but no sample code and documentation.
* The Gradle version used for executing the build. By default the wrapper task picks the exact same Gradle version that was used to generate the Wrapper files.

**gradle/wrapper/gradle-wrapper.properties**

distributionUrl=https\://services.gradle.org/distributions/gradle-4.3.1-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.

--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 Gradle distribution ZIP file. Using this option makes --gradle-version and --distribution-type obsolete as the URL already contains this information. This option is extremely valuable if you want to host the Gradle distribution inside your company’s network.

--gradle-distribution-sha256-sum

The SHA256 hash sum used for [verifying the downloaded Gradle distribution](#338fx5o).

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 4.0 and use the -all distribution to enable your IDE to enable code-completion and being able to navigate to the Gradle source code. Those requirements are captured by the following command line execution:

[Example: Providing options to Wrapper task](#30tazoa)

**Output of** gradle wrapper --gradle-version 4.0 --distribution-type all

> gradle wrapper --gradle-version 4.0 --distribution-type all  
> Task :wrapper  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

As a result you can find the desired information in the Wrapper properties file.

**gradle/wrapper/gradle-wrapper.properties**

distributionUrl=https\://services.gradle.org/distributions/gradle-4.0-all.zip

Let’s have a look at the following project layout to illustrate the expected Wrapper files:

.  
├── build.gradle  
├── settings.gradle  
├── gradle  
│ └── wrapper  
│ ├── gradle-wrapper.jar  
│ └── gradle-wrapper.properties  
├── gradlew  
└── gradlew.bat

A Gradle project typically provides a build.gradle and a settings.gradle file. 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.

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](#21od6so) without having to install the Gradle runtime. If the project you are working on does not contain those Wrapper files then you’ll need to [generate them](#3mj2wkv).

[Using the Gradle Wrapper](#21od6so)

It is recommended to always execute a build with the Wrapper to ensure a reliable, controlled and standardized execution of the build. Using the Wrapper looks almost exactly 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 demonstrate the use of the Wrapper on a Windows machine for a Java-based project.

[Example: Executing the build with the Wrapper batch file](#1fyl9w3)

**Output of gradlew.bat build**

> gradlew.bat build  
Downloading https://services.gradle.org/distributions/gradle-4.0-all.zip  
.....................................................................................  
Unzipping C:\Documents and Settings\Claudia\.gradle\wrapper\dists\gradle-4.0-all\ac27o8rbd0ic8ih41or9l32mv\gradle-4.0-all.zip to C:\Documents and Settings\Claudia\.gradle\wrapper\dists\gradle-4.0-al\ac27o8rbd0ic8ih41or9l32mv  
Set executable permissions for: C:\Documents and Settings\Claudia\.gradle\wrapper\dists\gradle-4.0-all\ac27o8rbd0ic8ih41or9l32mv\gradle-4.0\bin\gradle  
  
BUILD SUCCESSFUL in 12s  
1 actionable task: 1 executed

In case the Gradle distribution is not available on the machine, the Wrapper will download it and store in the local file system. Any subsequent build invocation is going to 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](#4iylrwe)

Projects will 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 manually change the distributionUrl property in the Wrapper property 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](#3mj2wkv). 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’d commit the changes to the Wrapper files to version control.

Use the Gradle wrapper task to generate the wrapper, specifying a version. The default is the current version, which you can check by executing ./gradlew --version.

[Example: Upgrading the Wrapper version](#3zy8sjw)

**Output of ./gradlew wrapper --gradle-version 4.2.1**

> ./gradlew wrapper --gradle-version 4.2.1  
  
BUILD SUCCESSFUL in 4s  
1 actionable task: 1 executed

[Example: Checking the Wrapper version after upgrading](#2f3j2rp)

**Output of ./gradlew -v**

> ./gradlew -v  
Downloading https://services.gradle.org/distributions/gradle-4.2.1-bin.zip  
...................................................................  
Unzipping /Users/claudia/.gradle/wrapper/dists/gradle-4.2.1-bin/dajvke9o8kmaxbu0kc5gcgeju/gradle-4.2.1-bin.zip to /Users/claudia/.gradle/wrapper/dists/gradle-4.2.1-bin/dajvke9o8kmaxbu0kc5gcgeju  
Set executable permissions for: /Users/claudia/.gradle/wrapper/dists/gradle-4.2.1-bin/dajvke9o8kmaxbu0kc5gcgeju/gradle-4.2.1/bin/gradle  
  
------------------------------------------------------------  
Gradle 4.2.1  
------------------------------------------------------------  
  
Build time: 2017-10-02 15:36:21 UTC  
Revision: a88ebd6be7840c2e59ae4782eb0f27fbe3405ddf  
  
Groovy: 2.4.12  
Ant: Apache Ant(TM) version 1.9.6 compiled on June 29 2015  
JVM: 1.8.0\_60 (Oracle Corporation 25.60-b23)  
OS: Mac OS X 10.13.1 x86\_64

[Customizing the Gradle Wrapper](#u8tczi)

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](http://docs.google.com/dsl/org.gradle.api.tasks.wrapper.Wrapper.html).

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.

[Example: Customizing the Wrapper task](#3e8gvnb)

**build.gradle**

wrapper {  
 distributionType = Wrapper.DistributionType.ALL  
}

With the configuration in place running ./gradlew wrapper --gradle-version 4.1 is enough to produce a distributionUrl value in the Wrapper properties file that will request the -all distribution.

**gradle/wrapper/gradle-wrapper.properties**

distributionUrl=https\://services.gradle.org/distributions/gradle-4.1-all.zip

Check out the API documentation for more detail descriptions of the available configuration options. You can also find various samples for configuring the Wrapper in the Gradle distribution.

[Authenticated Gradle distribution download](#18vjpp8)

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** | Security Warning  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. |
| --- | --- |

Using system properties can be done in the .gradle/gradle.properties file in the user’s home directory, or by other means, see [Gradle Configuration Properties](#1idq7dh).

**gradle.properties**

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. Shared credentials embedded in distributionUrl should only be used in a controlled environment.

**gradle/wrapper/gradle-wrapper.properties**

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](#3kkl7fh) for more information on how to configure the Wrapper to use a proxy.

[Verification of downloaded Gradle distributions](#338fx5o)

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](#1tdr5v4)

You can download the .sha256 file from the [stable releases](https://services.gradle.org/distributions/) or [release candidate and nightly releases](https://services.gradle.org/distributions-snapshots/). The format of the file is a single line of text that is the SHA-256 hash of the corresponding zip file.

[Configuring checksum verification](#4ddeoix)

Add the downloaded hash sum to gradle-wrapper.properties using the distributionSha256Sum property or use --gradle-distribution-sha256-sum on the command-line.

**gradle/wrapper/gradle-wrapper.properties**

distributionSha256Sum=371cb9fbebbe9880d147f59bab36d61eee122854ef8c9ee1ecf12b82368bcf10

Gradle will report a build failure in case the configured checksum does not match the checksum found on the server for hosting the distribution. Checksum Verification is only performed if the configured Wrapper distribution hasn’t been downloaded yet.

[Troubleshooting](#1y810tw)

The following is a collection of common issues and suggestions for addressing them. You can get other tips and search the [Gradle forums](https://discuss.gradle.org/c/help-discuss) and [StackOverflow #gradle](https://stackoverflow.com/questions/tagged/gradle) answers, as well as Gradle documentation from [help.gradle.org](https://help.gradle.org/).

[Troubleshooting Gradle installation](#j8sehv)

If you followed the [installation instructions](#3rdcrjn), and aren’t able to execute your Gradle build, here are some tips that may help.

If you installed Gradle outside of just invoking the [Gradle Wrapper](#3j2qqm3), you can check your Gradle installation by running gradle --version in a terminal.

You should see something like this:

❯ gradle --version  
  
-----------------------------------------------------------  
Gradle 4.6  
------------------------------------------------------------  
  
Build time: 2018-02-21 15:28:42 UTC  
Revision: 819e0059da49f469d3e9b2896dc4e72537c4847d  
  
Groovy: 2.4.12  
Ant: Apache Ant(TM) version 1.9.9 compiled on February 2 2017  
JVM: 1.8.0\_151 (Oracle Corporation 25.151-b12)  
OS: Mac OS X 10.13.3 x86\_64

If not, here are some things you might see instead.

[Command not found: gradle](#2sioyqq)

If you get "command not found: gradle", you need to ensure that Gradle is properly added to your PATH.

[JAVA\_HOME is set to an invalid directory](#17nz8yj)

If you get something like:

ERROR: JAVA\_HOME is set to an invalid directory  
  
Please set the JAVA\_HOME variable in your environment to match the location of your Java installation.

You’ll need to ensure that a [Java Development Kit](http://www.oracle.com/technetwork/java/javase/downloads/index.html) version 7 or higher is [properly installed](https://www.java.com/en/download/help/index_installing.xml), the JAVA\_HOME environment variable is set, and [Java is added to your PATH](https://www.java.com/en/download/help/path.xml).

[Permission denied](#3rnmrmc)

If you get "permission denied", that means that Gradle likely exists in the correct place, but it is not executable. You can fix this using chmod +x path/to/executable on \*nix-based systems.

[Other installation failures](#26sx1u5)

If gradle --version works, but all of your builds fail with the same error, it is possible there is a problem with one of your Gradle build configuration scripts.

You can verify the problem is with Gradle scripts by running gradle help which executes configuration scripts, but no Gradle tasks. If the error persists, build configuration is problematic. If not, then the problem exists within the execution of one or more of the requested tasks (Gradle executes configuration scripts first, and then executes build steps).

[Debugging dependency resolution](#ly7c1y)

Common dependency resolution issues such as resolving version conflicts are covered in [Troubleshooting Dependency Resolution](#2lwamvv).

You can see a dependency tree and see which resolved dependency versions differed from what was requested by clicking the *Dependencies* view and using the search functionality, specifying the resolution reason.



*Figure 3. Debugging dependency conflicts with build scans*

The [actual build scan](https://scans.gradle.com/s/sample/troubleshooting-userguide/dependencies?expandAll&filters=WzFd&toggled=W1swXSxbMF0sWzAsMF0sWzAsMV1d) with filtering criteria is available for exploration.

[Troubleshooting slow Gradle builds](#35xuupr)

For build performance issues (including “slow sync time”), see the guide to [Improving the Performance of Gradle Builds](https://guides.gradle.org/performance/).

Android developers should watch a presentation by the Android SDK Tools team about [Speeding Up Your Android Gradle Builds](https://youtu.be/7ll-rkLCtyk). Many tips are also covered in the Android Studio user guide [on optimizing build speed](https://developer.android.com/studio/build/optimize-your-build.html).

[Debugging build logic](#1l354xk)

[Attaching a debugger to your build](#452snld)

You can set breakpoints and debug [buildSrc and standalone plugins](#2k82xt6) in your Gradle build itself by setting the org.gradle.debug property to “true” and then attaching a remote debugger to port 5005.

❯ gradle help -Dorg.gradle.debug=true --no-daemon

In addition, if you’ve adopted the Kotlin DSL, you can also debug build scripts themselves.

| **Note** | You must either stop running Gradle Daemons or run with --no-daemon when using debug mode. |
| --- | --- |

[Adding and changing logging](#zdd80z)

In addition to [controlling logging verbosity](#l7a3n9), you can also control display of task outcomes (e.g. “UP-TO-DATE”) in lifecycle logging using the [--console=verbose flag](#3im3ia3).

You can also replace much of Gradle’s logging with your own by registering various event listeners. One example of a [custom event logger is explained in the logging documentation](#3jd0qos). You can also [control logging from external tools](#1yib0wl), making them more verbose in order to debug their execution.

| **Note** | Additional logs from the [Gradle Daemon](#44sinio) can be found under GRADLE\_USER\_HOME/daemon/<gradle-version>/. |
| --- | --- |

[Task executed when it should have been UP-TO-DATE](#4ihyjke)

--info logs explain why a task was executed, though build scans do this in a searchable, visual way by going to the *Timeline* view and clicking on the task you want to inspect.



*Figure 4. Debugging incremental build with a build scan*

You can learn what the task outcomes mean from [this listing](#2xn8ts7).

[Debugging IDE integration](#1csj400)

Many infrequent errors within IDEs can be solved by "refreshing" Gradle. See also more documentation on working with Gradle [in IntelliJ IDEA](https://www.jetbrains.com/help/idea/gradle.html) and [in Eclipse](http://www.vogella.com/tutorials/EclipseGradle/article.html).

[Refreshing IntelliJ IDEA](#3ws6mnt)

NOTE: This only works for Gradle projects [linked to IntelliJ](https://www.jetbrains.com/help/idea/gradle.html#link_gradle_project).

From the main menu, go to View > Tool Windows > Gradle. Then click on the *Refresh* icon.



*Figure 5. Refreshing a Gradle project in IntelliJ IDEA*

[Refreshing Eclipse (using Buildship)](#2bxgwvm)

If you’re using [Buildship](https://projects.eclipse.org/projects/tools.buildship) for the Eclipse IDE, you can re-synchronize your Gradle build by opening the "Gradle Tasks" view and clicking the "Refresh" icon, or by executing the Gradle > Refresh Gradle Project command from the context menu while editing a Gradle script.



*Figure 6. Refreshing a Gradle project in Eclipse Buildship*

[Getting additional help](#r2r73f)

If you didn’t find a fix for your issue here, please reach out to the Gradle community on the [help forum](https://discuss.gradle.org/c/help-discuss) or search relevant developer resources using [help.gradle.org](https://help.gradle.org/).

If you believe you’ve found a bug in Gradle, please [file an issue](https://github.com/gradle/gradle/issues) on GitHub.

[Authoring Gradle Builds](#4i7ojhp)

[The Feature Lifecycle](#2xcytpi)

Gradle is under constant development and improvement. New versions are delivered on a regular and frequent basis (approximately every 6 weeks). Continuous improvement combined with frequent delivery allows new features to be made available to users early and for invaluable real world feedback to be 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 is taken very seriously and is also considered a core value of the Gradle platform. This is something that is engineered into the development process by design choices and automated testing, and is formalised by [the section on backwards compatibility](#3b2epr8).

The Gradle *feature lifecycle* has been designed to meet these goals. It also serves to clearly communicate 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 kinds of features.

[States](#1q7ozz1)

Features can be in one of 4 states:

* Internal
* Incubating
* Public
* Deprecated

[Internal](#4a7cimu)

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 Guide, the DSL Reference or the API Reference documentation then the feature is not internal.

Internal features may evolve into public features.

[Incubating](#2pcmsun)

Features are introduced in the *incubating* state to allow real world feedback to be incorporated into the feature before it is made public and locked down to provide backwards compatibility. It also gives users who are willing to accept potential future changes early access to the feature so they can put it into use immediately.

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 scope, complexity and nature of the feature.

Features in incubation are clearly indicated to be so. In the source code, all methods/properties/classes that are incubating are annotated with [Incubating](http://docs.google.com/javadoc/org/gradle/api/Incubating.html), which is also used to specially mark them in the DSL and API references. If an incubating feature is discussed in this User Guide, it will be explicitly said to be in the incubating state.

[Public](#14hx32g)

The default state for a non-internal feature is *public*. Anything that is documented in the User Guide, DSL Reference or API references 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 backwards compatibility policy.

[Deprecated](#3ohklq9)

Some features will become superseded or irrelevant due to the natural evolution of Gradle. Such features will eventually be removed from Gradle after being *deprecated*. A deprecated feature will *never* be changed, until it is finally removed according to the backwards compatibility policy.

Deprecated features are clearly 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, and this will be described in the documentation. Using a deprecated feature will also result in a runtime warning in Gradle’s output.

Use of deprecated features should be avoided. The release notes for each release indicate any features that are being deprecated by the release.

[Backwards Compatibility Policy](#3b2epr8)

Gradle provides backwards compatibility across major versions (e.g. 1.x, 2.x, etc.). Once a public feature is introduced or promoted in a Gradle release it will remain indefinitely or until it is 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.

[Authoring Maintainable Build Scripts](#1ci93xb)

Gradle build scripts combine the qualities of declarative build logic, expressiveness as well as flexibility and rigidity as needed. As a build script author it is easy to fall into the trap of striking the wrong balance or applying poor coding habits. This chapter describes best practices for writing your build script in a meaningful, yet flexible and efficient way.

| **Note** | The third-party [Gradle lint plugin](https://github.com/nebula-plugins/gradle-lint-plugin) helps with enforcing a desired code style in a build script if you are looking for appropriate linting automation. |
| --- | --- |

[Avoiding imperative logic in scripts](#23muvy2)

The Gradle runtime does not enforce a specific style for build logic. For that very reason, it’s easy to end up with a build script that mixes declarative DSL elements with imperative, procedural code. Let’s talk about some concrete examples.

* *Declarative code:* Built-in, language-agnostic DSL elements (e.g. [Project.dependencies{}](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:dependencies(groovy.lang.Closure)) or [Project.repositories{}](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:repositories(groovy.lang.Closure))) or DSLs exposed by plugins
* *Imperative code:* Conditional logic or very complex task action implementations

The end goal of every build script should be to only contain declarative language elements which makes the code easier to understand and maintain. Imperative logic should live in binary plugins and which in turn is applied to the build script. As a side product, you automatically enable your team to [reuse the plugin logic in other projects](https://guides.gradle.org/designing-gradle-plugins/#reusable_logic_should_be_written_as_binary_plugin) if you publish the artifact to a binary repository.

The following sample build shows a negative example of using conditional logic directly in the build script. While this code snippet is small, it is easy to imagine a full-blown build script using numerous procedural statements and the impact it would have on readability and maintainability. By moving the code into a class [testability](https://guides.gradle.org/testing-gradle-plugins/) also becomes a valid option.

[Example: A build script using conditional logic to create a task](#is565v)

**build.gradle**

**if** (project.findProperty('releaseEngineer')) {  
 task release {  
 doLast {  
 logger.quiet 'Releasing to production...'  
  
 *// release the artifact to production*  
 }  
 }  
}

Let’s compare the build script with the same logic implemented as a binary plugin. The code might look more involved at first but clearly looks more like typical application code. This particular plugin class lives in the [buildSrc directory](#xevivl) which makes it available to the build script automatically.

[Example: A binary plugin implementing imperative logic](#32rsoto)

**ReleasePlugin.java**

**package** com.enterprise;  
  
**import** org.gradle.api.Action;  
**import** org.gradle.api.Plugin;  
**import** org.gradle.api.Project;  
**import** org.gradle.api.Task;  
  
**public** **class** **ReleasePlugin** **implements** Plugin<Project> {  
 **private** **static** **final** String RELEASE\_ENG\_ROLE\_PROP = "releaseEngineer";  
 **private** **static** **final** String RELEASE\_TASK\_NAME = "release";  
  
 @Override  
 **public** **void** apply(Project project) {  
 **if** (project.findProperty(RELEASE\_ENG\_ROLE\_PROP) != null) {  
 Task task = project.getTasks().create(RELEASE\_TASK\_NAME);  
  
 task.doLast(**new** Action<Task>() {  
 @Override  
 **public** **void** execute(Task task) {  
 task.getLogger().quiet("Releasing to production...");  
  
 *// release the artifact to production*  
 }  
 });  
 }  
 }  
}

Now that the build logic has been translated into a plugin, you can apply it in the build script. The build script has been shrunk from 8 lines of code to a one liner.

[Example: A build script applying a plugin that encapsulates imperative logic](#1hx2z1h)

**build.gradle**

apply plugin: 'com.enterprise.release'

[Avoiding Gradle internal APIs](#41wqhpa)

Use of Gradle internal APIs in plugins and build scripts has the potential to break builds when either Gradle or plugins change.

The following packages are listed in the [Gradle public API definition](https://github.com/gradle/gradle/blob/180b9d3fa84b91768364c603380e82947437eda1/buildSrc/subprojects/configuration/src/main/kotlin/org/gradle/gradlebuild/public-api.kt), with the exception of any subpackage with internal in the name:

org/gradle/\*  
org/gradle/api/\*\*  
org/gradle/authentication/\*\*  
org/gradle/buildinit/\*\*  
org/gradle/caching/\*\*  
org/gradle/concurrent/\*\*  
org/gradle/deployment/\*\*  
org/gradle/external/javadoc/\*\*  
org/gradle/ide/\*\*  
org/gradle/includedbuild/\*\*  
org/gradle/ivy/\*\*  
org/gradle/jvm/\*\*  
org/gradle/language/\*\*  
org/gradle/maven/\*\*  
org/gradle/nativeplatform/\*\*  
org/gradle/normalization/\*\*  
org/gradle/platform/\*\*  
org/gradle/play/\*\*  
org/gradle/plugin/devel/\*\*  
org/gradle/plugin/repository/\*  
org/gradle/plugin/use/\*  
org/gradle/plugin/management/\*  
org/gradle/plugins/\*\*  
org/gradle/process/\*\*  
org/gradle/testfixtures/\*\*  
org/gradle/testing/jacoco/\*\*  
org/gradle/tooling/\*\*  
org/gradle/swiftpm/\*\*  
org/gradle/model/\*\*  
org/gradle/testkit/\*\*  
org/gradle/testing/\*\*  
org/gradle/vcs/\*\*  
org/gradle/workers/\*\*

[Alternatives for oft-used internal APIs](#2h20rx3)

To provide a nested DSL for your custom task, don’t use org.gradle.internal.reflect.Instantiator; use [ObjectFactory](http://docs.google.com/javadoc/org/gradle/api/model/ObjectFactory.html) instead. It may also be helpful to read [the chapter on lazy configuration](#36ei31r).

Don’t use org.gradle.api.internal.ConventionMapping. Use [Provider](http://docs.google.com/javadoc/org/gradle/api/provider/Provider.html) and/or [Property](http://docs.google.com/javadoc/org/gradle/api/provider/Property.html). You can find an example for capturing user input to configure runtime behavior in the [implementing plugins guide](https://guides.gradle.org/implementing-gradle-plugins/#capturing_user_input_to_configure_plugin_runtime_behavior).

Instead of org.gradle.internal.os.OperatingSystem, use another method to detect operating system, such as [Apache commons-lang SystemUtils](https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/SystemUtils.html) or System.getProperty("os.name").

Use other collections or I/O frameworks instead of org.gradle.util.CollectionUtils, org.gradle.util.GFileUtils, and other classes under org.gradle.util.\*.

Gradle plugin authors may find the Designing Gradle Plugins subsection on [restricting the plugin implementation to Gradle’s public API](https://guides.gradle.org/designing-gradle-plugins/#restricting_the_plugin_implementation_to_gradle_s_public_api) helpful.

[Declaring tasks in a build script](#w7b24w)

The task API gives a build author a lot of flexibility to declare tasks in a build script. For optimal readability and maintainability follow these rules:

* The task type should be the only key-value pair that belongs within the parentheses after the task name.
* Any other configuration logic for a task should be defined as part of [Project.task(java.lang.String, groovy.lang.Closure)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:task(java.lang.String,%20groovy.lang.Closure)) if possible.
* [Task actions](#3g6yksp) should only be declared with the methods [Task.doFirst{}](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:doFirst(org.gradle.api.Action)) or [Task.doLast{}](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:doLast(org.gradle.api.Action)).
* [Task.doLast{}](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:doLast(org.gradle.api.Action)) should be used if the task only defines a single action.
* A task should [define a group and description](#1vc8v0i).

[Example: Definition of tasks following best practices](#4fbwdob)

**build.gradle**

**import** com.enterprise.DocsGenerate  
  
task generateHtmlDocs(type: DocsGenerate) {  
 group = JavaBasePlugin.DOCUMENTATION\_GROUP  
 description = 'Generates the HTML documentation for this project.'  
 title = 'Project docs'  
 outputDir = file("${buildDir}/docs")  
}  
  
task allDocs {  
 group = JavaBasePlugin.DOCUMENTATION\_GROUP  
 description = 'Generates all documentation for this project.'  
 dependsOn generateHtmlDocs  
  
 doLast {  
 logger.quiet('Generating all documentation...')  
 }  
}

[Improving task discoverability](#1vc8v0i)

Even new users to a build should to be able to find crucial information quickly and effortlessly. In Gradle you can declare a [group](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:group) and a [description](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:description) for any task of the build. The [tasks report](#1f7o1he) uses the assigned values to organize and render the task for easy discoverability. Assigning a group and description is most helpful for any task that you expect build users to invoke.

The example task generateDocs generates documentation for a project in the form of HTML pages. The task should be organized underneath the bucket Documentation. The description should express its intent.

[Example: A task declaring the group and description](#2uh6nw4)

**build.gradle**

task generateDocs {  
 group = 'Documentation'  
 description = 'Generates the HTML documentation for this project.'  
  
 doLast {  
 *// action implementation*  
 }  
}

The output of the tasks report reflects the assigned values.

> gradle tasks  
  
> Task :tasks  
  
Documentation tasks  
-------------------  
generateDocs - Generates the HTML documentation for this project.

[Minimize logic executed during the configuration phase](#19mgy3x)

It’s important for every build script developer to understand the different phases of the [build lifecycle](#3as4poj) and their implications on performance and evaluation order of build logic. During the configuration phase the project and its domain objects should be *configured*, whereas the execution phase only executes the actions of the task(s) requested on the command line plus their dependencies. Be aware that any code that is not part of a task action will be executed with *every single run* of the build. A [build scan](https://scans.gradle.com/get-started) can help you with identifying the time spent during each of the lifecycle phases. It’s an invaluable tool for diagnosing common performance issues.

Let’s consider the following incantation of the anti-pattern described above. In the build script you can see that the dependencies assigned to the configuration printArtifactNames are resolved outside of the task action.

[Example: Executing logic during configuration should be avoided](#3tm4grq)

**build.gradle**

dependencies {  
 implementation 'log4j:log4j:1.2.17'  
}  
  
task printArtifactNames {  
 *// always executed*  
 **def** libraryNames = configurations.compileClasspath.collect { it.name }  
  
 doLast {  
 logger.quiet libraryNames  
 }  
}

The code for resolving the dependencies should be moved into the task action to avoid the performance impact of resolving the dependencies before they are actually needed.

[Example: Executing logic during execution phase is preferred](#28reqzj)

**build.gradle**

dependencies {  
 implementation 'log4j:log4j:1.2.17'  
}  
  
task printArtifactNames {  
 doLast {  
 **def** libraryNames = configurations.compileClasspath.collect { it.name }  
 logger.quiet libraryNames  
 }  
}

[Avoiding the use of GradleBuild](#nwp17c)

The [GradleBuild](http://docs.google.com/dsl/org.gradle.api.tasks.GradleBuild.html) task type allows a build script to define a task that invokes another Gradle build. The use of this type is generally discouraged. There are some corner cases where the invoked build doesn’t expose the same runtime behavior as from the command line or through the Tooling API leading to unexpected results.

Usually, there’s a better way to model the requirement. The appropriate approach depends on the problem at hand. Here’re some options:

* Model the build as [multi-project build](#2p2csry) if the intention is to execute tasks from different modules as unified build.
* Use [composite builds](#49x2ik5) for projects that are physically separated but should occasionally be built as a single unit.

[Avoiding inter-project configuration](#37wcjv5)

Gradle does not restrict build script authors from reaching into the domain model from one project into another one in a [multi-project build](#2p2csry). Strongly-coupled projects hurts [build execution performance](#44bvf6o) as well as readability and maintainability of code.

The following practices should be avoided:

* Explicitly depending on a task from another project via [Task.dependsOn(java.lang.Object...)](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:dependsOn(java.lang.Object%5B%5D)).
* Setting property values or calling methods on domain objects from another project.
* Executing another portion of the build with [GradleBuild](#nwp17c).
* Declaring unnecessary [project dependencies](#1n1mu2y).

[Avoiding passwords in plain text](#471acqr)

Most builds need to consume one or many passwords. The reasons for this need may vary. Some builds need a password for publishing artifacts to a secured binary repository, other builds need a password for downloading binary files. Passwords should always kept safe to prevent fraud. Under no circumstance should you add the password to the build script as property in plain text or declare it in a gradle.properties. Those files usually live in a version control repository and can be viewed by anyone that has access to it.

Passwords should be stored in encrypted fashion. At the moment Gradle does not provide a built-in mechanism for encrypting, storing and accessing passwords. A good solution for solving this problem is the [Gradle Credentials plugin](https://github.com/etiennestuder/gradle-credentials-plugin).

[Organizing Gradle Projects](#3whwml4)

Source code and build logic of every software project should be organized in a meaningful way. This page lays out the best practices that lead to readable, maintainable projects. The following sections also touch on common problems and how to avoid them.

[Separate language-specific source files](#2m6kmyk)

Gradle’s language plugins establish conventions for discovering and compiling source code. For example, a project applying the [Java plugin](#2w5ecyt) will automatically compile the code in the directory src/main/java. Other language plugins follow the same pattern. The last portion of the directory path usually indicates the expected language of the source files.

Some compilers are capable of cross-compiling multiple languages in the same source directory. The Groovy compiler can handle the scenario of mixing Java and Groovy source files located in src/main/groovy. Gradle recommends that you place sources in directories according to their language, because builds are more performant and both the user and build can make stronger assumptions.

The following source tree contains Java and Kotlin source files. Java source files live in src/main/java, whereas Kotlin source files live in src/main/kotlin.

.  
├── build.gradle  
├── settings.gradle  
└── src  
 └── main  
 ├── java  
 │   └── HelloWorld.java  
 └── kotlin  
 └── Utils.kt

[Separate source files per test type](#11bux6d)

It’s very common that a project defines and executes different types of tests e.g. unit tests, integration tests, functional tests or smoke tests. Optimally, the test source code for each test type should be stored in dedicated source directories. Separated test source code has a positive impact on maintainability and separation of concerns as you can run test types independent from each other.

The following source tree demonstrates how to separate unit from integration tests in a Java-based project.

.  
├── build.gradle  
├── gradle  
│   └── integration-test.gradle  
├── settings.gradle  
└── src  
 ├── integTest  
 │   └── java  
 │   └── DefaultFileReaderIntegrationTest.java  
 ├── main  
 │   └── java  
 │   ├── DefaultFileReader.java  
 │   ├── FileReader.java  
 │   └── StringUtils.java  
 └── test  
 └── java  
 └── StringUtilsTest.java

Gradle models source code directories with the help of the [source set concept](#3lbifu6). By pointing an instance of a source set to one or many source code directories, Gradle will automatically create a corresponding compilation task out-of-the-box.

[Example: Integration test source set](#20gsq1z)

**gradle/integration-test.gradle**

sourceSets {  
 integTest {  
 java.srcDir file('src/integTest/java')  
 resources.srcDir file('src/integTest/resources')  
 compileClasspath += sourceSets.main.output + configurations.testRuntimeClasspath  
 runtimeClasspath += output + compileClasspath  
 }  
}

Source sets are only responsible for compiling source code, but do not deal with executing the byte code. For the purpose of test execution, a corresponding task of type [Test](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html) needs to be established.

[Example: Integration test task](#4kgg8ps)

**gradle/integration-test.gradle**

task integTest(type: Test) {  
 description = 'Runs the integration tests.'  
 group = 'verification'  
 testClassesDirs = sourceSets.integTest.output.classesDirs  
 classpath = sourceSets.integTest.runtimeClasspath  
 mustRunAfter test  
}  
  
check.dependsOn integTest

[Use standard conventions as much as possible](#2zlqixl)

All Gradle core plugins follow the software engineering paradigm [convention over configuration](https://en.wikipedia.org/wiki/Convention_over_configuration). The plugin logic provides users with sensible defaults and standards, the conventions, in a certain context. Let’s take the [Java plugin](#2w5ecyt) as an example.

* It defines the directory src/main/java as the default source directory for compilation.
* The output directory for compiled source code and other artifacts (like the JAR file) is build.

By sticking to the default conventions, new developers to the project immediately know how to find their way around. While those conventions can be reconfigured, it makes it harder to build script users and authors to manage the build logic and its outcome. Try to stick to the default conventions as much as possible except if you need to adapt to the layout of a legacy project. Refer to the reference page of the relevant plugin to learn about its default conventions.

[Always define a settings file](#1er0t5e)

Gradle tries to locate a settings.gradle (Groovy DSL) or a settings.gradle.kts (Kotlin DSL) file with every invocation of the build. For that purpose, the runtime walks the hierarchy of the directory tree up to the root directory. The algorithm stops searching as soon as it finds the settings file.

Always add a settings.gradle to the root directory of your build to avoid the initial performance impact. This recommendation applies to single project builds as well as multi-project builds. The file can either be empty or define the desired name of the project.

A typical Gradle project with a settings file look as such:

.  
├── build.gradle  
└── settings.gradle

[Use buildSrc to abstract imperative logic](#xevivl)

Complex build logic is usually a good candidate for being encapsulated either as custom task or binary plugin. Custom task and plugin implementations should not live in the build script. It is very convenient to use buildSrc for that purpose as long as the code does not need to be shared among multiple, independent projects.

The directory buildSrc is treated as an [included build](#3yqobt7). Upon discovery of the directory, Gradle automatically compiles and tests this code and puts it in the classpath of your build script. For multi-project builds there can be only one buildSrc directory, which has to sit in the root project directory. buildSrc should be preferred over [script plugins](#2dvym10) as it is easier to maintain, refactor and test the code.

buildSrc uses the same [source code conventions](#t18w8t) applicable to Java and Groovy projects. It also provides direct access to the Gradle API. Additional dependencies can be declared in a dedicated build.gradle under buildSrc.

[Example: Custom buildSrc build script](#3d0wewm)

**buildSrc/build.gradle**

repositories {  
 mavenCentral()  
}  
  
dependencies {  
 testCompile 'junit:junit:4.12'  
}

A typical project including buildSrc has the following layout. Any code under buildSrc should use a package similar to application code. Optionally, the buildSrc directory can host a build script if additional configuration is needed (e.g. to apply plugins or to declare dependencies).

.  
├── build.gradle  
├── buildSrc  
│   ├── build.gradle  
│   └── src  
│   ├── main  
│   │   └── java  
│   │   └── com  
│   │   └── enterprise  
│   │   ├── Deploy.java  
│   │   └── DeploymentPlugin.java  
│   └── test  
│   └── java  
│   └── com  
│   └── enterprise  
│   └── DeploymentPluginTest.java  
└── settings.gradle

[Declare properties in gradle.properties file](#1s66p4f)

In Gradle, properties can be define in the build script, in a gradle.properties file or as parameters on the command line.

It’s common to declare properties on the command line for ad-hoc scenarios. For example you may want to pass in a specific property value to control runtime behavior just for this one invocation of the build. Properties in a build script can easily become a maintenance headache and convolute the build script logic. The gradle.properties helps with keeping properties separate from the build script and should be explored as viable option. It’s a good location for placing [properties that control the build environment](#1idq7dh).

A typical project setup places the gradle.properties file in the root directory of the build. Alternatively, the file can also live in the GRADLE\_USER\_HOME directory if you want to it apply to all builds on your machine.

.  
├── build.gradle  
├── gradle.properties  
└── settings.gradle

[Avoid overlapping task outputs](#4c5u7s8)

Tasks should define inputs and outputs to get the performance benefits of [incremental build functionality](#odc9jc). When declaring the outputs of a task, make sure that the directory for writing outputs is unique among all the tasks in your project.

Intermingling or overwriting output files produced by different tasks compromises up-to-date checking causing slower builds. In turn, these filesystem changes may prevent Gradle’s [build cache](#2bn6wsx) from properly identifying and caching what would otherwise be cacheable tasks.

[Standardizing builds with a custom Gradle distribution](#2rb4i01)

Often enterprises want to standardize the build platform for all projects in the organization by defining common conventions or rules. You can achieve that with the help of initialization scripts. [Initialization scripts](#2jxsxqh) make it extremely easy to apply build logic across all projects on a single machine. For example, to declare a in-house repository and its credentials.

There are some drawbacks to the approach. First of all, you will have to communicate the setup process across all developers in the company. Furthermore, updating the initialization script logic uniformly can prove challenging.

Custom Gradle distributions are a practical solution to this very problem. A custom Gradle distribution is comprised of the standard Gradle distribution plus one or many custom initialization scripts. The initialization scripts come bundled with the distribution and are applied every time the build is run. Developers only need to point their checked-in [Wrapper](#3j2qqm3) files to the URL of the custom Gradle distribution.

The following steps are typical for creating a custom Gradle distribution:

1. Implement logic for downloading and repackaging a Gradle distribution.
2. Define one or many initialization scripts with the desired logic.
3. Bundle the initialization scripts with the Gradle distribution.
4. Upload the Gradle distribution archive to a HTTP server.
5. Change the Wrapper files of all projects to point to the URL of the custom Gradle distribution.

You can find a sample project that covers steps one to three in the samples directory of the standard -all Gradle distribution.

[Build Cache](#2bn6wsx)

| **Note** | The build cache feature described here is different from the [Android plugin build cache](https://developer.android.com/studio/build/build-cache.html). |
| --- | --- |

[Overview](#16ges7u)

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](#odc9jc) 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 task output caching, we expect other features to use the build cache in the future.

| **Note** | A complete guide is available about [using the build cache](https://guides.gradle.org/using-build-cache/). It covers the different scenarios caching can improve, and detailed discussions of the different caveats you need to be aware of when enabling caching for a build. |
| --- | --- |

[Enable the Build Cache](#3qg2avn)

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](#25lcl3g).

[Task Output Caching](#vgdtq7)

Beyond incremental builds described in [up-to-date checks](#odc9jc), 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](#odc9jc) 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](#3qg2avn).

[What does it look like](#kqmvb9)

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](#34qadz2) and :processResources and :jar being Copy-like tasks which are not cacheable since it is generally faster to execute them.

[Cacheable tasks](#1jvko6v)

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](#43v86uo):

* 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"](#2j0ih2h)
* The names and values of properties added by the DSL via [TaskInputs](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskInputs.html)
* 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](http://docs.google.com/javadoc/org/gradle/api/tasks/CacheableTask.html) annotation. Note that [@CacheableTask](http://docs.google.com/javadoc/org/gradle/api/tasks/CacheableTask.html) is not inherited by subclasses. Custom task types are *not* cacheable by default.

[Built-in cacheable tasks](#y5sraa)

Currently, the following built-in Gradle tasks are cacheable:

* Java toolchain: [JavaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.JavaCompile.html), [Javadoc](http://docs.google.com/dsl/org.gradle.api.tasks.javadoc.Javadoc.html)
* Groovy toolchain: [GroovyCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.GroovyCompile.html), [Groovydoc](http://docs.google.com/dsl/org.gradle.api.tasks.javadoc.Groovydoc.html)
* Scala toolchain: [ScalaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.scala.ScalaCompile.html), [PlatformScalaCompile](http://docs.google.com/javadoc/org/gradle/language/scala/tasks/PlatformScalaCompile.html), [ScalaDoc](http://docs.google.com/dsl/org.gradle.api.tasks.scala.ScalaDoc.html)
* Native toolchain: [CppCompile](http://docs.google.com/javadoc/org/gradle/language/cpp/tasks/CppCompile.html), [CCompile](http://docs.google.com/javadoc/org/gradle/language/c/tasks/CCompile.html), [SwiftCompile](http://docs.google.com/javadoc/org/gradle/language/swift/tasks/SwiftCompile.html)
* Testing: [Test](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html)
* Code quality tasks: [Checkstyle](http://docs.google.com/dsl/org.gradle.api.plugins.quality.Checkstyle.html), [CodeNarc](http://docs.google.com/dsl/org.gradle.api.plugins.quality.CodeNarc.html), [FindBugs](http://docs.google.com/dsl/org.gradle.api.plugins.quality.FindBugs.html), [JDepend](http://docs.google.com/dsl/org.gradle.api.plugins.quality.JDepend.html), [Pmd](http://docs.google.com/dsl/org.gradle.api.plugins.quality.Pmd.html)
* JaCoCo: [JacocoMerge](http://docs.google.com/dsl/org.gradle.testing.jacoco.tasks.JacocoMerge.html), [JacocoReport](http://docs.google.com/dsl/org.gradle.testing.jacoco.tasks.JacocoReport.html)
* Other tasks: [AntlrTask](http://docs.google.com/dsl/org.gradle.api.plugins.antlr.AntlrTask.html), [ValidateTaskProperties](http://docs.google.com/javadoc/org/gradle/plugin/devel/tasks/ValidateTaskProperties.html), [WriteProperties](http://docs.google.com/dsl/org.gradle.api.tasks.WriteProperties.html)

All other built-in tasks are currently not cacheable.

Some tasks, like [Copy](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html) or [Jar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Jar.html), 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](#3i5g9y3)

There are third party plugins that work well with the build cache. The most prominent examples are the [Android plugin 3.1+](https://developer.android.com/studio/releases/gradle-plugin.html) and the [Kotlin plugin 1.2.21+](https://blog.gradle.org/kotlin-build-cache-use). For other third party plugins, check their documentation to find out whether they support the build cache.

[Declaring task inputs and outputs](#1xaqk5w)

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"](#4hae2tp) 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](http://docs.google.com/javadoc/org/gradle/api/tasks/PathSensitive.html)).

In order to handle volatile inputs for your tasks consider [configuring input normalization](#2wfod1i).

[Configure the Build Cache](#25lcl3g)

You can configure the build cache by using the [Settings.buildCache(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.initialization.Settings.html#org.gradle.api.initialization.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()](http://docs.google.com/javadoc//org/gradle/caching/configuration/BuildCache.html#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](http://docs.google.com/dsl/org.gradle.caching.local.DirectoryBuildCache.html) 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)](http://docs.google.com/dsl/org.gradle.caching.configuration.BuildCacheConfiguration.html#org.gradle.caching.configuration.BuildCacheConfiguration:remote(java.lang.Class))).

[Built-in local build cache](#1bkyn9b)

The built-in local build cache, [DirectoryBuildCache](http://docs.google.com/dsl/org.gradle.caching.local.DirectoryBuildCache.html), uses a directory to store build cache artifacts. By default, this directory resides in the Gradle user home directory, but its location is configurable.

Gradle will periodically clean-up the local cache directory by removing entries that have not been used recently to conserve disk space.

For more details on the configuration options refer to the DSL documentation of [DirectoryBuildCache](http://docs.google.com/dsl/org.gradle.caching.local.DirectoryBuildCache.html). Here is an example of the configuration.

[Example: Configure the local cache](#3vkm5x4)

**settings.gradle**

buildCache {  
 local(DirectoryBuildCache) {  
 directory = **new** File(rootDir, 'build-cache')  
 removeUnusedEntriesAfterDays = 30  
 }  
}

[Remote HTTP build cache](#2apwg4x)

Gradle has built-in support for connecting to a remote build cache backend via HTTP. For more details on what the protocol looks like see [HttpBuildCache](http://docs.google.com/dsl/org.gradle.caching.http.HttpBuildCache.html). Note that by using 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: Load from HttpBuildCache](#pv6qcq)

**settings.gradle**

buildCache {  
 remote(HttpBuildCache) {  
 url = 'https://example.com:8123/cache/'  
 }  
}

You can configure the credentials the [HttpBuildCache](http://docs.google.com/dsl/org.gradle.caching.http.HttpBuildCache.html) uses to access the build cache server as shown in the following example.

[Example: Configure remote HTTP cache](#39uu90j)

**settings.gradle**

buildCache {  
 remote(HttpBuildCache) {  
 url = 'http://example.com:8123/cache/'  
 credentials {  
 username = 'build-cache-user'  
 password = 'some-complicated-password'  
 }  
 }  
}

| **Note** | You may encounter problems with an untrusted SSL certificate when you try to use a build cache backend with an HTTPS URL. The ideal solution is for someone to add a valid SSL certificate to the build cache backend, but we recognize that you may not be able to do that. In that case, set [HttpBuildCache.isAllowUntrustedServer()](http://docs.google.com/dsl/org.gradle.caching.http.HttpBuildCache.html#org.gradle.caching.http.HttpBuildCache:allowUntrustedServer) to true:  **settings.gradle**  buildCache {  remote(HttpBuildCache) {  url = 'https://example.com:8123/cache/'  allowUntrustedServer = true  } }  This is a convenient workaround, but you shouldn’t use it as a long-term solution. |
| --- | --- |

[Configuration use cases](#1p04j8c)

The recommended use case for the build cache is that your continuous integration server populates the remote build cache from clean builds while developers load from the remote build cache and store in the local build cache. The configuration would then look as follows.

[Example: Recommended setup for CI push use case](#48zs1w5)

**settings.gradle**

ext.isCiServer = System.getenv().containsKey("CI")  
  
buildCache {  
 local {  
 enabled = !isCiServer  
 }  
 remote(HttpBuildCache) {  
 url = 'https://example.com:8123/cache/'  
 push = isCiServer  
 }  
}

If you use a buildSrc directory, you should make sure that it uses the same build cache configuration as the main build. This can be achieved by applying the same script to buildSrc/settings.gradle and settings.gradle as shown in the following example.

[Example: Consistent setup for buildSrc and main build](#2o52c3y)

**settings.gradle**

apply from: **new** File(settingsDir, 'gradle/buildCacheSettings.gradle')

**gradle/buildCacheSettings.gradle**

ext.isCiServer = System.getenv().containsKey("CI")  
  
buildCache {  
 local {  
 enabled = !isCiServer  
 }  
 remote(HttpBuildCache) {  
 url = 'https://example.com:8123/cache/'  
 push = isCiServer  
 }  
}

**buildSrc/settings.gradle**

apply from: **new** File(settingsDir, '../gradle/buildCacheSettings.gradle')

It is also possible to configure the build cache from an [init script](#3hej1je), which can be used from the command line, added to your Gradle user home or be a part of your custom Gradle distribution.

[Example: Init script to configure the build cache](#13acmbr)

**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 and composite builds](#3na04zk)

Gradle’s [composite build feature](#49x2ik5) 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.

[How to set up an HTTP build cache backend](#22faf7d)

Gradle provides a Docker image for a [build cache node](https://hub.docker.com/r/gradle/build-cache-node/), which can connect with Gradle Enterprise for centralized management. The cache node can also be used without a Gradle Enterprise installation with restricted functionality.

[Implement your own Build Cache](#hkkpf6)

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)](http://docs.google.com/javadoc/org/gradle/caching/configuration/BuildCacheConfiguration.html#registerBuildCacheService-java.lang.Class-java.lang.Class-).

[Gradle Enterprise](https://gradle.com/build-cache) includes a high-performance, easy to install and operate, shared build cache backend.

[Build Init Plugin](#qsh70q)

| **Note** | The Build Init plugin is currently [incubating](#2xcytpi). Please be aware that the DSL and other configuration may change in later Gradle versions. |
| --- | --- |

The Gradle Build Init plugin can be used to bootstrap the process of creating a new Gradle build. It supports creating brand new projects of different types as well as converting existing builds (e.g. An Apache Maven build) to be Gradle builds.

Gradle plugins typically need to be *applied* to a project before they can be used (see [Using plugins](#31k882z)). The Build Init plugin is an automatically applied plugin, which means you do not need to apply it explicitly. To use the plugin, simply execute the task named init where you would like to create the Gradle build. There is no need to create a “stub” build.gradle file in order to apply the plugin.

It also leverages the wrapper task to [generate the Gradle Wrapper files](#3mj2wkv) for the project.

[Tasks](#1gpiias)

The plugin adds the following tasks to the project:

init — [InitBuild](http://docs.google.com/dsl/org.gradle.buildinit.tasks.InitBuild.html)

*Depends on*: wrapper

Generates a Gradle project.

wrapper — [Wrapper](http://docs.google.com/dsl/org.gradle.api.tasks.wrapper.Wrapper.html)

Generates Gradle wrapper files.

[What to set up](#40p60yl)

The init supports different build setup *types*. The type is specified by supplying a --type argument value. For example, to create a Java library project simply execute: gradle init --type java-library.

If a --type parameter is not supplied, Gradle will attempt to infer the type from the environment. For example, it will infer a type value of “pom” if it finds a pom.xml to convert to a Gradle build.

If the type could not be inferred, the type “basic” will be used.

The init plugin also supports generating build scripts using either the Gradle Groovy DSL or the Gradle Kotlin DSL. The build script DSL to use defaults to the Groovy DSL and is specified by supplying a --dsl argument value. For example, to create a Java library project with Kotlin DSL build scripts simply execute: gradle init --type java-library --dsl kotlin.

All build setup types include the setup of the Gradle Wrapper.

Note that the migration from Maven builds only supports the Groovy DSL for generated build scripts.

[Build init types](#2fugb6e)

| **Note** | As this plugin is currently [incubating](#2xcytpi), only a few build init types are currently supported. More types will be added in future Gradle releases. |
| --- | --- |

[pom (Maven conversion)](#uzqle7)

The “pom” type can be used to convert an Apache Maven build to a Gradle build. This works by converting the POM to one or more Gradle files. It is only able to be used if there is a valid “pom.xml” file in the directory that the init task is invoked in or, if invoked via the “-p” [command line option](#lnxbz9), in the specified project directory. This “pom” type will be automatically inferred if such a file exists.

The Maven conversion implementation was inspired by the [maven2gradle tool](https://github.com/jbaruch/maven2gradle) that was originally developed by Gradle community members.

The conversion process has the following features:

* Uses effective POM and effective settings (support for POM inheritance, dependency management, properties)
* Supports both single module and multimodule projects
* Supports custom module names (that differ from directory names)
* Generates general metadata - id, description and version
* Applies maven, java and war plugins (as needed)
* Supports packaging war projects as jars if needed
* Generates dependencies (both external and inter-module)
* Generates download repositories (inc. local Maven repository)
* Adjusts Java compiler settings
* Supports packaging of sources and tests
* Supports TestNG runner
* Generates global exclusions from Maven enforcer plugin settings

[java-application](#3eze420)

The “java-application” build init type is not inferable. It must be explicitly specified.

It has the following features:

* Uses the “application” plugin to produce a command-line application implemented using Java
* Uses the “jcenter” dependency repository
* Uses [JUnit](http://junit.org) for testing
* Has directories in the conventional locations for source code
* Contains a sample class and unit test, if there are no existing source or test files

Alternative test framework can be specified by supplying a --test-framework argument value. To use a different test framework, execute one of the following commands:

* gradle init --type java-application --test-framework spock: Uses [Spock](http://code.google.com/p/spock/) for testing instead of JUnit
* gradle init --type java-application --test-framework testng: Uses [TestNG](http://testng.org/doc/index.html) for testing instead of JUnit

[java-library](#1u4oe9t)

The “java-library” build init type is not inferable. It must be explicitly specified.

It has the following features:

* Uses the “java” plugin to produce a library Jar
* Uses the “jcenter” dependency repository
* Uses [JUnit](http://junit.org) for testing
* Has directories in the conventional locations for source code
* Contains a sample class and unit test, if there are no existing source or test files

Alternative test framework can be specified by supplying a --test-framework argument value. To use a different test framework, execute one of the following commands:

* gradle init --type java-library --test-framework spock: Uses [Spock](http://code.google.com/p/spock/) for testing instead of JUnit
* gradle init --type java-library --test-framework testng: Uses [TestNG](http://testng.org/doc/index.html) for testing instead of JUnit

[scala-library](#4e4bwxm)

The “scala-library” build init type is not inferable. It must be explicitly specified.

It has the following features:

* Uses the “scala” plugin to produce a library Jar
* Uses the “jcenter” dependency repository
* Uses Scala 2.10
* Uses [ScalaTest](http://www.scalatest.org) for testing
* Has directories in the conventional locations for source code
* Contains a sample scala class and an associated ScalaTest test suite, if there are no existing source or test files
* Uses the Zinc Scala compiler by default

[groovy-library](#2t9m75f)

The “groovy-library” build init type is not inferable. It must be explicitly specified.

It has the following features:

* Uses the “groovy” plugin to produce a library Jar
* Uses the “jcenter” dependency repository
* Uses Groovy 2.x
* Uses [Spock testing framework](http://spockframework.org) for testing
* Has directories in the conventional locations for source code
* Contains a sample Groovy class and an associated Spock specification, if there are no existing source or test files

[groovy-application](#18ewhd8)

The “groovy-application” build init type is not inferable. It must be explicitly specified.

It has the following features:

* Uses the “groovy” plugin
* Uses the “application” plugin to produce a command-line application implemented using Groovy
* Uses the “jcenter” dependency repository
* Uses Groovy 2.x
* Uses [Spock testing framework](http://spockframework.org) for testing
* Has directories in the conventional locations for source code
* Contains a sample Groovy class and an associated Spock specification, if there are no existing source or test files

[basic](#3sek011)

The “basic” build init type is useful for creating a fresh new Gradle project. It creates a sample build.gradle file, with comments and links to help get started.

This type is used when no type was explicitly specified, and no type could be inferred.

[Build Lifecycle](#3as4poj)

We said earlier that the core of Gradle is a language for dependency based programming. In Gradle terms this means that you can define tasks and dependencies between tasks. Gradle guarantees that these tasks are executed in the order of their dependencies, and that each task is executed only once. These tasks form a [Directed Acyclic Graph](http://en.wikipedia.org/wiki/Directed_acyclic_graph). There are build tools that build up such a dependency graph as they execute their tasks. Gradle builds the complete dependency graph *before* any task is executed. This lies at the heart of Gradle and makes many things possible which would not be possible otherwise.

Your build scripts configure this dependency graph. Therefore they are strictly speaking *build configuration scripts*.

[Build phases](#27jua8u)

A Gradle build has three distinct phases.

Initialization

Gradle supports single and multi-project builds. During the initialization phase, Gradle determines which projects are going to take part in the build, and creates a [Project](http://docs.google.com/dsl/org.gradle.api.Project.html) instance for each of these projects.

Configuration

During this phase the project objects are configured. The build scripts of *all* projects which are part of the build are executed.

Execution

Gradle determines the subset of the tasks, created and configured during the configuration phase, to be executed. The subset is determined by the task name arguments passed to the gradle command and the current directory. Gradle then executes each of the selected tasks.

[Settings file](#3xzr3ei)

Beside the build script files, Gradle defines a settings file. The settings file is determined by Gradle via a naming convention. The default name for this file is settings.gradle. Later in this chapter we explain how Gradle looks for a settings file.

The settings file is executed during the initialization phase. A multi-project build must have a settings.gradle file in the root project of the multi-project hierarchy. It is required because the settings file defines which projects are taking part in the multi-project build (see [Authoring Multi-Project Builds](#2p2csry)). For a single-project build, a settings file is optional. Besides defining the included projects, you might need it to add libraries to your build script classpath (see [Organizing Gradle Projects](#3whwml4)). Let’s first do some introspection with a single project build:

[Example: Single project build](#mp4kgn)

**settings.gradle**

println 'This is executed during the initialization phase.'

**build.gradle**

println 'This is executed during the configuration phase.'  
  
task configured {  
 println 'This is also executed during the configuration phase.'  
}  
  
task test {  
 doLast {  
 println 'This is executed during the execution phase.'  
 }  
}  
  
task 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.'  
}

**Output of gradle test testBoth**

> gradle test testBoth  
This is executed during the initialization phase.  
  
> Configure project :  
This is executed during the configuration phase.  
This is also executed during the configuration phase.  
This is executed during the configuration phase as well.  
  
> 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

For a build script, the property access and method calls are delegated to a project object. Similarly property access and method calls within the settings file is delegated to a settings object. Look at the [Settings](http://docs.google.com/dsl/org.gradle.api.initialization.Settings.html) class in the API documentation for more information.

[Multi-project builds](#36os34g)

A multi-project build is a build where you build more than one project during a single execution of Gradle. You have to declare the projects taking part in the multi-project build in the settings file. There is much more to say about multi-project builds in the chapter dedicated to this topic (see [Authoring Multi-Project Builds](#2p2csry)).

[Project locations](#1lu2dc9)

Multi-project builds are always represented by a tree with a single root. Each element in the tree represents a project. A project has a path which denotes the position of the project in the multi-project build tree. In most cases the project path is consistent with the physical location of the project in the file system. However, this behavior is configurable. The project tree is created in the settings.gradle file. By default it is assumed that the location of the settings file is also the location of the root project. But you can redefine the location of the root project in the settings file.

[Building the tree](#45tpw02)

In the settings file you can use a set of methods to build the project tree. Hierarchical and flat physical layouts get special support.

[Hierarchical layouts](#2kz067v)

[Example: Hierarchical layout](#104agfo)

**settings.gradle**

include 'project1', 'project2:child', '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 from the project root). You only need to specify the leaves of the tree. This means that the inclusion of the path 'services:hotels:api' will result in creating 3 projects: 'services', 'services:hotels' and 'services:hotels:api'. More examples of how to work with the project path can be found in the DSL documentation of [Settings.include(java.lang.String[])](http://docs.google.com/dsl/org.gradle.api.initialization.Settings.html#org.gradle.api.initialization.Settings:include(java.lang.String%5B%5D)).

[Flat layouts](#3k3xz3h)

[Example: Flat layout](#1z989ba)

**settings.gradle**

includeFlat 'project3', 'project4'

The includeFlat method takes directory names as an argument. These directories need to exist as siblings of the root project directory. The location of these directories are considered as child projects of the root project in the multi-project tree.

[Modifying elements of the project tree](#4j8vrz3)

The multi-project tree created in the settings file is made up of so called *project descriptors*. You can modify these descriptors in the settings file at any time. To access a descriptor you can do:

[Example: Lookup of elements of the project tree](#2ye626w)

**settings.gradle**

println rootProject.name  
println project(':projectA').name

Using this descriptor you can change the name, project directory and build file of a project.

[Example: Modification of elements of the project tree](#1djgcep)

**settings.gradle**

rootProject.name = 'main'  
project(':projectA').projectDir = **new** File(settingsDir, '../my-project-a')  
project(':projectA').buildFileName = 'projectA.gradle'

Look at the [ProjectDescriptor](http://docs.google.com/javadoc/org/gradle/api/initialization/ProjectDescriptor.html) class in the API documentation for more information.

[Initialization](#3xj3v2i)

How does Gradle know whether to do a single or multi-project build? If you trigger a multi-project build from a directory with a settings file, things are easy. But Gradle also allows you to execute the build from within any subproject taking part in the build.[[2](#2coe5ab)] If you execute Gradle from within a project with no settings.gradle file, Gradle looks for a settings.gradle file in the following way:

* It looks in a directory called master which has the same nesting level as the current dir.
* If not found yet, it searches parent directories.
* If not found yet, the build is executed as a single project build.
* If a settings.gradle file is found, Gradle checks if the current project is part of the multi-project hierarchy defined in the found settings.gradle file. If not, the build is executed as a single project build. Otherwise a multi-project build is executed.

What is the purpose of this behavior? Gradle needs to determine whether the project you are in is a subproject of a multi-project build or not. Of course, if it is a subproject, only the subproject and its dependent projects are built, but Gradle needs to create the build configuration for the whole multi-project build (see [Authoring Multi-Project Builds](#2p2csry)). You can use the -u command line option to tell Gradle not to look in the parent hierarchy for a settings.gradle file. The current project is then always built as a single project build. If the current project contains a settings.gradle file, the -u option has no meaning. Such a build is always executed as:

* a single project build, if the settings.gradle file does not define a multi-project hierarchy
* a multi-project build, if the settings.gradle file does define a multi-project hierarchy.

The automatic search for a settings.gradle file only works for multi-project builds with a physical hierarchical or flat layout. For a flat layout you must additionally follow the naming convention described above (“master”). Gradle supports arbitrary physical layouts for a multi-project build, but for such arbitrary layouts you need to execute the build from the directory where the settings file is located. For information on how to run partial builds from the root, see [Running tasks by their absolute path](#rtofi4).

Gradle creates a Project object for every project taking part in the build. For a multi-project build these are the projects specified in the Settings object (plus the root project). Each project object has by default a name equal to the name of its top level directory, and every project except the root project has a parent project. Any project may have child projects.

[Configuration and execution of a single project build](#3btby5x)

For a single project build, the workflow of the *after initialization* phases are pretty simple. The build script is executed against the project object that was created during the initialization phase. Then Gradle looks for tasks with names equal to those passed as command line arguments. If these task names exist, they are executed as a separate build in the order you have passed them. The configuration and execution for multi-project builds is discussed in [Authoring Multi-Project Builds](#2p2csry).

[Responding to the lifecycle in the build script](#1qym8dq)

Your build script can receive notifications as the build progresses through its lifecycle. These notifications generally take two forms: You can either implement a particular listener interface, or you can provide a closure to execute when the notification is fired. The examples below use closures. For details on how to use the listener interfaces, refer to the API documentation.

[Project evaluation](#4ay9r1j)

You can receive a notification immediately before and after a project is evaluated. This can be used to do things like performing additional configuration once all the definitions in a build script have been applied, or for some custom logging or profiling.

Below is an example which adds a test task to each project which has a hasTests property value of true.

[Example: Adding of test task to each project which has certain property set](#2q3k19c)

**build.gradle**

allprojects {  
 afterEvaluate { project ->  
 **if** (project.hasTests) {  
 println "Adding test task to $project"  
 project.task('test') {  
 doLast {  
 println "Running tests for $project"  
 }  
 }  
 }  
 }  
}

**projectA.gradle**

hasTests = true

**Output of gradle -q test**

> gradle -q test  
Adding test task to project ':projectA'  
Running tests for project ':projectA'

This example uses method Project.afterEvaluate() to add a closure which is executed after the project is evaluated.

It is also possible to receive notifications when any project is evaluated. This example performs some custom logging of project evaluation. Notice that the afterProject notification is received regardless of whether the project evaluates successfully or fails with an exception.

[Example: Notifications](#158ubh5)

**build.gradle**

gradle.afterProject {project, projectState ->  
 **if** (projectState.failure) {  
 println "Evaluation of $project FAILED"  
 } **else** {  
 println "Evaluation of $project succeeded"  
 }  
}

**Output of** gradle -q test

> gradle -q test  
Evaluation of root project 'buildProjectEvaluateEvents' succeeded  
Evaluation of project ':projectA' succeeded  
Evaluation of project ':projectB' FAILED  
  
FAILURE: Build failed with an exception.  
  
\* Where:  
Build file '/home/user/gradle/samples/projectB.gradle' line: 1  
  
\* What went wrong:  
A problem occurred evaluating project ':projectB'.  
> broken  
  
\* 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

You can also add a [ProjectEvaluationListener](http://docs.google.com/javadoc/org/gradle/api/ProjectEvaluationListener.html) to the [Gradle](http://docs.google.com/dsl/org.gradle.api.invocation.Gradle.html) to receive these events.

[Task creation](#3p8hu4y)

You can receive a notification immediately after a task is added to a project. This can be used to set some default values or add behaviour before the task is made available in the build file.

The following example sets the srcDir property of each task as it is created.

[Example: Setting of certain property to all tasks](#24ds4cr)

**build.gradle**

tasks.whenTaskAdded { task ->  
 task.ext.srcDir = 'src/main/java'  
}  
  
task a  
  
println "source dir is $a.srcDir"

**Output of** gradle -q a

> gradle -q a  
source dir is src/main/java

You can also add an [Action](http://docs.google.com/javadoc/org/gradle/api/Action.html) to a [TaskContainer](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskContainer.html) to receive these events.

[Task execution graph ready](#jj2ekk)

You can receive a notification immediately after the task execution graph has been populated (See [Configure by DAG](#33ipx8d)).

You can also add a [TaskExecutionGraphListener](http://docs.google.com/javadoc/org/gradle/api/execution/TaskExecutionGraphListener.html) to the [TaskExecutionGraph](http://docs.google.com/javadoc/org/gradle/api/execution/TaskExecutionGraph.html) to receive these events.

[Task execution](#1io07g6)

You can receive a notification immediately before and after any task is executed.

The following example logs the start and end of each task execution. Notice that the afterTask notification is received regardless of whether the task completes successfully or fails with an exception.

[Example: Logging of start and end of each task execution](#42nnq3z)

**build.gradle**

task ok  
  
task broken(dependsOn: ok) {  
 doLast {  
 **throw** **new** RuntimeException('broken')  
 }  
}  
  
gradle.taskGraph.beforeTask { Task task ->  
 println "executing $task ..."  
}  
  
gradle.taskGraph.afterTask { Task task, TaskState state ->  
 **if** (state.failure) {  
 println "FAILED"  
 }  
 **else** {  
 println "done"  
 }  
}

**Output of** gradle -q broken

> gradle -q broken  
executing task ':ok' ...  
done  
executing task ':broken' ...  
FAILED  
  
FAILURE: Build failed with an exception.  
  
\* Where:  
Build file '/home/user/gradle/samples/build.gradle' line: 5  
  
\* What went wrong:  
Execution failed for task ':broken'.  
> broken  
  
\* 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

You can also use a [TaskExecutionListener](http://docs.google.com/javadoc/org/gradle/api/execution/TaskExecutionListener.html) to the [TaskExecutionGraph](http://docs.google.com/javadoc/org/gradle/api/execution/TaskExecutionGraph.html) to receive these events.

[Build Script Basics](#1pxezwc)

[Projects and tasks](#2hsy0bs)

Everything in Gradle sits on top of two basic concepts: *projects* and *tasks*.

Every Gradle build is made up of one or more *projects*. What a project represents depends on what it is that you are doing with Gradle. For example, a project might represent a library JAR or a web application. It might represent a distribution ZIP assembled from the JARs produced by other projects. A project does not necessarily represent a thing to be built. It might represent a thing to be done, such as deploying your application to staging or production environments. Don’t worry if this seems a little vague for now. Gradle’s build-by-convention support adds a more concrete definition for what a project is.

Each project is made up of one or more *tasks*. A task represents some atomic piece of work which a build performs. This might be compiling some classes, creating a JAR, generating Javadoc, or publishing some archives to a repository.

For now, we will look at defining some simple tasks in a build with one project. Later chapters will look at working with multiple projects and more about working with projects and tasks.

[Hello world](#3g6yksp)

You run a Gradle build using the gradle command. The gradle command looks for a file called build.gradle in the current directory.[[3](#wy8ajl)] We call this build.gradle file a *build script*, although strictly speaking it is a build configuration script, as we will see later. The build script defines a project and its tasks.

To try this out, create the following build script named build.gradle.

[Example: Your first build script](#3gxvt7e)

**build.gradle**

task hello {  
 doLast {  
 println 'Hello world!'  
 }  
}

In a command-line shell, move to the containing directory and execute the build script with gradle -q hello:

| **Tip** | What does -q do?  Most of the examples in this user guide are run with the -q command-line option. This suppresses Gradle’s log messages, so that only the output of the tasks is shown. This keeps the example output in this user guide a little clearer. You don’t need to use this option if you don’t want to. See [Logging](#3o7alnk) for more details about the command-line options which affect Gradle’s output. |
| --- | --- |

[Example: Execution of a build script](#1w363f7)

**Output of** gradle -q hello

> gradle -q hello  
Hello world!

What’s going on here? This build script defines a single task, called hello, and adds an action to it. When you run gradle hello, Gradle executes the hello task, which in turn executes the action you’ve provided. The action is simply a closure containing some Groovy code to execute.

If you think this looks similar to Ant’s targets, you would be right. Gradle tasks are the equivalent to Ant targets, but as you will see, they are much more powerful. We have used a different terminology than Ant as we think the word *task* is more expressive than the word *target*. Unfortunately this introduces a terminology clash with Ant, as Ant calls its commands, such as javac or copy, tasks. So when we talk about tasks, we *always* mean Gradle tasks, which are the equivalent to Ant’s targets. If we talk about Ant tasks (Ant commands), we explicitly say *Ant task*.

[A shortcut task definition](#4g2tm30)

| **Note** | This functionality is deprecated and will be removed in Gradle 5.0 without replacement. Use the methods [Task.doFirst(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:doFirst(org.gradle.api.Action)) and [Task.doLast(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:doLast(org.gradle.api.Action)) to define an action instead, as demonstrated by the rest of the examples in this chapter. |
| --- | --- |

There is a shorthand way to define a task like our hello task above, which is more concise.

[Example: A task definition shortcut](#2v83wat)

**build.gradle**

task hello << {  
 println 'Hello world!'  
}

Again, this defines a task called hello with a single closure to execute. The << operator is simply an alias for doLast.

[Build scripts are code](#1ade6im)

Gradle’s build scripts give you the full power of Groovy. As an appetizer, have a look at this:

[Example: Using Groovy in Gradle’s tasks](#3ud1p6f)

**build.gradle**

task upper {  
 doLast {  
 String someString = 'mY\_nAmE'  
 println "Original: " + someString  
 println "Upper case: " + someString.toUpperCase()  
 }  
}

**Output of** gradle -q upper

> gradle -q upper  
Original: mY\_nAmE  
Upper case: MY\_NAME

or

[Example: Using Groovy in Gradle’s tasks](#29ibze8)

**build.gradle**

task count {  
 doLast {  
 4.times { print "$it " }  
 }  
}

**Output of** gradle -q count

> gradle -q count  
0 1 2 3

[Task dependencies](#3gnlt4p)

As you probably have guessed, you can declare tasks that depend on other tasks.

[Example: Declaration of task that depends on other task](#onm9m1)

**build.gradle**

task hello {  
 doLast {  
 println 'Hello world!'  
 }  
}  
task intro(dependsOn: hello) {  
 doLast {  
 println "I'm Gradle"  
 }  
}

**Output of** gradle -q intro

> gradle -q intro  
Hello world!  
I'm Gradle

To add a dependency, the corresponding task does not need to exist.

[Example: Lazy dependsOn - the other task does not exist (yet)](#38n9s9u)

**build.gradle**

task taskX(dependsOn: 'taskY') {  
 doLast {  
 println 'taskX'  
 }  
}  
task taskY {  
 doLast {  
 println 'taskY'  
 }  
}

**Output of** gradle -q taskX

> gradle -q taskX  
taskY  
taskX

The dependency of taskX to taskY is declared before taskY is defined. This is very important for multi-project builds. Task dependencies are discussed in more detail in [Adding dependencies to a task](#1nsk2hn).

Please notice that you can’t use [shortcut notation](#47s7l5g) when referring to a task that is not yet defined.

[Dynamic tasks](#2mxhvd9)

The power of Groovy can be used for more than defining what a task does. For example, you can also use it to dynamically create tasks.

[Example: Dynamic creation of a task](#122s5l2)

**build.gradle**

4.times { counter ->  
 task "task$counter" {  
 doLast {  
 println "I'm task number $counter"  
 }  
 }  
}

**Output of** gradle -q task1

> gradle -q task1  
I'm task number 1

[Manipulating existing tasks](#3m2fo8v)

Once tasks are created they can be accessed via an *API*. For instance, you could use this to dynamically add dependencies to a task, at runtime. Ant doesn’t allow anything like this.

[Example: Accessing a task via API - adding a dependency](#217pygo)

**build.gradle**

4.times { counter ->  
 task "task$counter" {  
 doLast {  
 println "I'm task number $counter"  
 }  
 }  
}  
task0.dependsOn task2, task3

**Output of** gradle -q task0

> 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.

[Example: Accessing a task via API - adding behaviour](#4l7dh4h)

**build.gradle**

task hello {  
 doLast {  
 println 'Hello Earth'  
 }  
}  
hello.doFirst {  
 println 'Hello Venus'  
}  
hello.doLast {  
 println 'Hello Mars'  
}  
hello {  
 doLast {  
 println 'Hello Jupiter'  
 }  
}

**Output of** gradle -q hello

> gradle -q hello  
Hello Venus  
Hello Earth  
Hello Mars  
Hello Jupiter

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.

[Shortcut notations](#47s7l5g)

There is a convenient notation for accessing an *existing* task. Each task is available as a property of the build script:

[Example: Accessing task as a property of the build script](#30cnrca)

**build.gradle**

task hello {  
 doLast {  
 println 'Hello world!'  
 }  
}  
hello.doLast {  
 println "Greetings from the $hello.name task."  
}

**Output of** gradle -q hello

> gradle -q hello  
Hello world!  
Greetings from the hello task.

This enables very readable code, especially when using the tasks provided by the plugins, like the compile task.

[Extra task properties](#1fhy1k3)

You can add your own properties to a task. To add a property named myProperty, set ext.myProperty to an initial value. From that point on, the property can be read and set like a predefined task property.

[Example: Adding extra properties to a task](#3zhlk7w)

**build.gradle**

task myTask {  
 ext.myProperty = "myValue"  
}  
  
task printTaskProperties {  
 doLast {  
 println myTask.myProperty  
 }  
}

**Output of** gradle -q printTaskProperties

> gradle -q printTaskProperties  
myValue

Extra properties aren’t limited to tasks. You can read more about them in [Extra properties](#2emvufp).

[Using Ant Tasks](#ts64ni)

Ant tasks are first-class citizens in Gradle. Gradle provides excellent integration for Ant tasks by simply relying on Groovy. Groovy is shipped with the fantastic AntBuilder. Using Ant tasks from Gradle is as convenient and more powerful than using Ant tasks from a build.xml file. From the example below, you can learn how to execute Ant tasks and how to access Ant properties:

[Example: Using AntBuilder to execute ant.loadfile target](#3drtnbb)

**build.gradle**

task loadfile {  
 doLast {  
 **def** files = file('./antLoadfileResources').listFiles().sort()  
 files.each { File file ->  
 **if** (file.isFile()) {  
 ant.loadfile(srcFile: file, property: file.name)  
 println " \*\*\* $file.name \*\*\*"  
 println "${ant.properties[file.name]}"  
 }  
 }  
 }  
}

**Output of** gradle -q loadfile

> gradle -q loadfile  
 \*\*\* agile.manifesto.txt \*\*\*  
Individuals and interactions over processes and tools  
Working software over comprehensive documentation  
Customer collaboration over contract negotiation  
Responding to change over following a plan  
 \*\*\* gradle.manifesto.txt \*\*\*  
Make the impossible possible, make the possible easy and make the easy elegant.  
(inspired by Moshe Feldenkrais)

There is lots more you can do with Ant in your build scripts. You can find out more in [Ant](#pkwqa1).

[Using methods](#1sx3xj4)

Gradle scales in how you can organize your build logic. The first level of organizing your build logic for the example above, is extracting a method.

[Example: Using methods to organize your build logic](#4cwrg6x)

**build.gradle**

task checksum {  
 doLast {  
 fileList('./antLoadfileResources').each { File file ->  
 ant.checksum(file: file, property: "cs\_$file.name")  
 println "$file.name Checksum: ${ant.properties["cs\_$file.name"]}"  
 }  
 }  
}  
  
task loadfile {  
 doLast {  
 fileList('./antLoadfileResources').each { File file ->  
 ant.loadfile(srcFile: file, property: file.name)  
 println "I'm fond of $file.name"  
 }  
 }  
}  
  
File**[]** fileList(String dir) {  
 file(dir).listFiles({file -> file.isFile() } **as** FileFilter).sort()  
}

**Output of** gradle -q loadfile

> gradle -q loadfile  
I'm fond of agile.manifesto.txt  
I'm fond of gradle.manifesto.txt

Later you will see that such methods can be shared among subprojects in multi-project builds. If your build logic becomes more complex, Gradle offers you other very convenient ways to organize it. We have devoted a whole chapter to this. See [Organizing Gradle Projects](#3whwml4).

[Default tasks](#2s21qeq)

Gradle allows you to define one or more default tasks that are executed if no other tasks are specified.

[Example: Defining a default task](#177c0mj)

**build.gradle**

defaultTasks 'clean', 'run'  
  
task clean {  
 doLast {  
 println 'Default Cleaning!'  
 }  
}  
  
task run {  
 doLast {  
 println 'Default Running!'  
 }  
}  
  
task other {  
 doLast {  
 println "I'm not a default task!"  
 }  
}

**Output of** gradle -q

> gradle -q  
Default Cleaning!  
Default Running!

This is equivalent to running gradle clean run. In a multi-project build every subproject can have its own specific default tasks. If a subproject does not specify default tasks, the default tasks of the parent project are used (if defined).

[Configure by DAG](#33ipx8d)

As we later describe in full detail (see [Build Lifecycle](#3as4poj)), Gradle has a configuration phase and an execution phase. After the configuration phase, Gradle knows all tasks that should be executed. Gradle offers you a hook to make use of this information. A use-case for this would be to check if the release task is among the tasks to be executed. Depending on this, you can assign different values to some variables.

In the following example, execution of the distribution and release tasks results in different value of the version variable.

[Example: Different outcomes of build depending on chosen tasks](#3r6zjac)

**build.gradle**

task distribution {  
 doLast {  
 println "We build the zip with version=$version"  
 }  
}  
  
task release(dependsOn: 'distribution') {  
 doLast {  
 println 'We release now'  
 }  
}  
  
gradle.taskGraph.whenReady {taskGraph ->  
 **if** (taskGraph.hasTask(release)) {  
 version = '1.0'  
 } **else** {  
 version = '1.0-SNAPSHOT'  
 }  
}

**Output of gradle -q distribution**

> gradle -q distribution  
We build the zip with version=1.0-SNAPSHOT

**Output of gradle -q release**

> gradle -q release  
We build the zip with version=1.0  
We release now

The important thing is that whenReady affects the release task *before* the release task is executed. This works even when the release task is not the *primary* task (i.e., the task passed to the gradle command).

[External dependencies for the build script](#3utoxif)

If your build script needs to use external libraries, you can add them to the script’s classpath in the build script itself. You do this using the buildscript() method, passing in a closure which declares the build script classpath.

[Example: Declaring external dependencies for the build script](#26c9ti5)

**build.gradle**

buildscript {  
 repositories {  
 mavenCentral()  
 }  
 dependencies {  
 classpath group: 'commons-codec', name: 'commons-codec', version: '1.2'  
 }  
}

The closure passed to the buildscript() method configures a [ScriptHandler](http://docs.google.com/javadoc/org/gradle/api/initialization/dsl/ScriptHandler.html) instance. You declare the build 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](#2u6wntf) except project dependencies.

Having declared the build script classpath, you can use the classes in your build script as you would any other classes on the classpath. The following example adds to the previous example, and uses classes from the build script classpath.

[Example: A build script with external dependencies](#lhk3py)

**build.gradle**

**import** org.apache.commons.codec.binary.Base64  
  
buildscript {  
 repositories {  
 mavenCentral()  
 }  
 dependencies {  
 classpath group: 'commons-codec', name: 'commons-codec', version: '1.2'  
 }  
}  
  
task encode {  
 doLast {  
 **def** **byte[]** encodedString = **new** Base64().encode('hello world\n'.getBytes())  
 println **new** String(encodedString)  
 }  
}

**Output of** gradle -q encode

> gradle -q encode  
aGVsbG8gd29ybGQK

For multi-project builds, the dependencies declared with a project’s buildscript() method are available to the build scripts of all its sub-projects.

Build script dependencies may be Gradle plugins. Please consult [Using Gradle Plugins](#32hioqz) for more information on Gradle plugins.

Every project automatically has a buildEnvironment task of type [BuildEnvironmentReportTask](http://docs.google.com/dsl/org.gradle.api.tasks.diagnostics.BuildEnvironmentReportTask.html) that can be invoked to report on the resolution of the build script dependencies.

[Where to next?](#35h7mdr)

In this chapter, we have had a first look at tasks. But this is not the end of the story for tasks. If you want to jump into more of the details, have a look at [More About Tasks](#147n2zr).

Otherwise, continue on to [the tutorials](#3hv69ve) and [Dependency Management for Java Projects](#2afmg28).

[Composite builds](#49x2ik5)

| **Note** | Composite build is an [incubating](#2xcytpi) feature. While useful for many use cases, there are bugs to be discovered, rough edges to smooth, and enhancements we plan to make. Thanks for trying it out! |
| --- | --- |

[What is a composite build?](#3yqobt7)

A composite build is simply a build that includes other builds. In many ways a composite build is similar to a Gradle multi-project build, except that instead of including single projects, complete 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 in independently or together as needed

A build that is included in a composite build is referred to, naturally enough, 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.

Included builds interact with other builds via dependency substitution. If any build in the composite has a dependency that can be satisfied by the included build, then that dependency will be replaced by a project dependency on the included build.

By default, Gradle will attempt to determine the dependencies that can be substituted by an included build. However for more flexibility, it is possible to explicitly declare these substitutions if the default ones determined by Gradle are not correct for the composite. See [Declaring substitutions](#1kmhwlk).

As well as consuming outputs via project dependencies, a composite build can directly declare task dependencies on included builds. Included builds are isolated, and are not able to declare task dependencies on the composite build or on other included builds. See [Depending on tasks in an included build](#44m5f9d).

[Defining a composite build](#2jrfph6)

The following examples demonstrate the various ways that 2 Gradle builds that are normally developed separately can be combined into a composite build. For these examples, the my-utils multi-project build produces 2 different java libraries (number-utils and string-utils), and the my-app build produces an executable using functions from those libraries.

The my-app build does not have direct dependencies on my-utils. Instead, it declares binary dependencies on the libraries produced by my-utils.

[Example: Dependencies of my-app](#ywpzoz)

**my-app/build.gradle**

apply plugin: 'java'  
apply plugin: 'application'  
apply plugin: 'idea'  
  
group "org.sample"  
version "1.0"  
  
mainClassName = "org.sample.myapp.Main"  
  
dependencies {  
 compile "org.sample:number-utils:1.0"  
 compile "org.sample:string-utils:1.0"  
}  
  
repositories {  
 jcenter()  
}

| **Note** | The code for this example can be found at samples/compositeBuilds/basic in the ‘-all’ distribution of Gradle. |
| --- | --- |

[Defining a composite build via --include-build](#3iwdics)

The --include-build command-line argument turns the executed build into a composite, substituting dependencies from the included build into the executed build.

[Example: Declaring a command-line composite](#1y1nskl)

**Output of** gradle --include-build ../my-utils run

> gradle --include-build ../my-utils run  
> Task :processResources NO-SOURCE  
> Task :my-utils:string-utils:compileJava  
> Task :my-utils:string-utils:processResources NO-SOURCE  
> Task :my-utils:string-utils:classes  
> Task :my-utils:string-utils:jar  
> Task :my-utils:number-utils:compileJava  
> Task :my-utils:number-utils:processResources NO-SOURCE  
> Task :my-utils:number-utils:classes  
> Task :my-utils:number-utils:jar  
> Task :compileJava  
> Task :classes  
  
> Task :run  
The answer is 42  
  
  
BUILD SUCCESSFUL in 0s  
2 actionable tasks: 2 executed

[Defining a composite build via settings.gradle](#4i1bb8e)

It’s possible to make the above arrangement persistent, by using [Settings.includeBuild(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.initialization.Settings.html#org.gradle.api.initialization.Settings:includeBuild(java.lang.Object)) to declare the included build in the settings.gradle file. The settings.gradle file can be used to add subprojects and included builds at the same time. Included builds are added by location. See the examples below for more details.

[Defining a separate composite build](#2x6llg7)

One downside of the above approach is that it requires you to modify an existing build, rendering it less useful as a standalone build. One way to avoid this is to define a separate composite build, whose only purpose is to combine otherwise separate builds.

[Example: Declaring a separate composite](#1cbvvo0)

**settings.gradle**

rootProject.name='adhoc'  
  
includeBuild '../my-app'  
includeBuild '../my-utils'

In this scenario, the 'main' build that is executed is the composite, and it doesn’t define any useful tasks to execute itself. In order to execute the 'run' task in the 'my-app' build, the composite build must define a delegating task.

[Example: Depending on task from included build](#3wbjebt)

**build.gradle**

task run {  
 dependsOn gradle.includedBuild('my-app').task(':run')  
}

More details tasks that depend on included build tasks below.

[Restrictions on included builds](#2bgtojm)

Most builds can be included into a composite, including other composite builds. However there are some limitations.

Every included build:

* must not have a rootProject.name the same as another included build.
* must not have a rootProject.name the same as a top-level project of the composite build.
* must not have a rootProject.name the same as the composite build rootProject.name.

[Interacting with a composite build](#qm3yrf)

In general, interacting with a composite build is much the same as a regular multi-project build. Tasks can be executed, tests can be run, and builds can be imported into the IDE.

[Executing tasks](#3alrhf8)

Tasks from the composite build can be executed from the command line, or from you IDE. Executing a task will result in direct task dependencies being executed, as well as those tasks required to build dependency artifacts from included builds.

| **Note** | There is not (yet) any means to directly execute a task from an included build via the command line. Included build tasks are automatically executed in order to generate required dependency artifacts, or the [including build can declare a dependency on a task from an included build](#44m5f9d). |
| --- | --- |

[Importing into the IDE](#1pr1rn1)

One of the most useful features of composite builds is IDE integration. By applying the [idea](#2szc72q) or [eclipse](#4du1wux) plugin to your build, it is possible to generate a single IDEA or Eclipse project that permits all builds in the composite to be developed together.

In addition to these Gradle plugins, recent versions of [IntelliJ IDEA](https://www.jetbrains.com/idea/) and [Eclipse Buildship](https://projects.eclipse.org/projects/tools.buildship) support direct import of a composite build.

Importing a composite build permits sources from separate Gradle builds to be easily developed together. For every included build, each sub-project is included as an IDEA Module or Eclipse Project. Source dependencies are configured, providing cross-build navigation and refactoring.

[Declaring the dependencies substituted by an included build](#1kmhwlk)

By default, Gradle will configure each included build in order to determine the dependencies it can provide. The algorithm for doing this is very 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}.

There are cases when the default substitutions determined by Gradle are not sufficient, or they are not correct for a particular composite. For these cases it is possible to explicitly declare the substitutions for an included build. Take for example a single-project build 'unpublished', that produces a java utility library but does not declare a value for the group attribute:

[Example: Build that does not declare group attribute](#49qpaau)

**build.gradle**

apply plugin: 'java'

When this build is included in a composite, it will attempt to substitute for the dependency module "undefined:unpublished" ("undefined" being the default value for project.group, and 'unpublished' being the root project name). Clearly this isn’t going to be very useful in a composite build. To use the unpublished library unmodified in a composite build, the composing build can explicitly declare the substitutions that it provides:

[Example: Declaring the substitutions for an included build](#2ovzkin)

**settings.gradle**

rootProject.name = 'app'  
  
includeBuild('../anonymous-library') {  
 dependencySubstitution {  
 substitute module('org.sample:number-utils') with 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 "unpublished".

[Cases where included build substitutions must be declared](#1419uqg)

Many builds that use the uploadArchives task to publish artifacts 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: this usually means a task other than uploadArchives is used.
* 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](#3o0xde9)

Some builds won’t function correctly when included in a composite, even when dependency substitutions are explicitly declared. This limitation is due to the fact that a project dependency that is substituted will always point to the default configuration of the target project. Any time that the artifacts and dependencies specified for the default configuration of a project don’t match what is actually published to a repository, then the composite build may exhibit different behaviour.

Here are some cases where the publish 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. We plan to improve this in the future.

[Depending on tasks in an included build](#44m5f9d)

While included builds are isolated from one another and cannot declare direct dependencies, a composite build is able to declare task dependencies on its included builds. The included builds are accessed using [Gradle.getIncludedBuilds()](http://docs.google.com/dsl/org.gradle.api.invocation.Gradle.html#org.gradle.api.invocation.Gradle:includedBuilds) or [Gradle.includedBuild(java.lang.String)](http://docs.google.com/dsl/org.gradle.api.invocation.Gradle.html#org.gradle.api.invocation.Gradle:includedBuild(java.lang.String)), and a task reference is obtained via the [IncludedBuild.task(java.lang.String)](http://docs.google.com/dsl/org.gradle.api.initialization.IncludedBuild.html#org.gradle.api.initialization.IncludedBuild:task(java.lang.String)) method.

Using these APIs, it is possible to declare a dependency on a task in a particular included build, or tasks with a certain path in all or some of the included builds.

[Example: Depending on a single task from an included build](#2367nm2)

**build.gradle**

task run {  
 dependsOn gradle.includedBuild('my-app').task(':run')  
}

[Example: Depending on a tasks with path in all included builds](#ibhxtv)

**build.gradle**

task publishDeps {  
 dependsOn gradle.includedBuilds\*.task(':uploadArchives')  
}

[Current limitations and future plans for composite builds](#32b5gho)

We think composite builds are pretty useful already. However, there are some things that don’t yet work the way we’d like, and other improvements that we think will make things work even better.

Limitations of the current implementation include:

* No support for included builds that have publications that don’t mirror the project default configuration. See [Cases where composite builds won’t work](#3o0xde9).
* Native builds are not supported. (Binary dependencies are not yet supported for native builds).
* Substituting plugins only works with the buildscript block but not with the plugins block.

Improvements we have planned for upcoming releases include:

* Better detection of dependency substitution, for build that publish with custom coordinates, builds that produce multiple components, etc. This will reduce the cases where dependency substitution needs to be explicitly declared for an included build.
* The ability to target a task or tasks in an included build directly from the command line. We are currently exploring syntax options for allowing this functionality, which will remove many cases where a delegating task is required in the composite.
* Make the plugins {} block consider included builds when locating plugins and their dependencies.
* Making the implicit buildSrc project an included build.

[Authoring Multi-Project Builds](#2p2csry)

The powerful support for multi-project builds is one of Gradle’s unique selling points. This topic is also the most intellectually challenging.

A multi-project build in gradle consists of one root project, and one or more subprojects that may also have subprojects.

[Cross project configuration](#1hgfqph)

While each subproject could configure itself in complete isolation of the other subprojects, it is common that subprojects share common traits. It is then usually preferable to share configurations among projects, so the same configuration affects several subprojects.

Let’s start with a very simple multi-project build. Gradle is a general purpose build tool at its core, so the projects don’t have to be Java projects. Our first examples are about marine life.

[Configuration and execution](#41g39da)

[Build phases](#27jua8u) describes the phases of every Gradle build. Let’s zoom into the configuration and execution phases of a multi-project build. Configuration here means executing the build.gradle file of a project, which implies e.g. downloading all plugins that were declared using ‘apply plugin’. By default, the configuration of all projects happens before any task is executed. This means that when a single task, from a single project is requested, *all* projects of multi-project build are configured first. The reason every project needs to be configured is to support the flexibility of accessing and changing any part of the Gradle project model.

[Configuration on demand](#1kc7wiv)

The *Configuration injection* feature and access to the complete project model are possible because every project is configured before the execution phase. Yet, this approach may not be the most efficient in a very large multi-project build. There are Gradle builds with a hierarchy of hundreds of subprojects. The configuration time of huge multi-project builds may become noticeable. Scalability is an important requirement for Gradle. Hence, starting from version 1.4 a new incubating 'configuration on demand' mode is introduced.

Configuration on demand mode attempts to configure only projects that are relevant for requested tasks, i.e. it only executes the build.gradle file of projects that are participating in the build. This way, the configuration time of a large multi-project build can be reduced. In the long term, this mode will become the default mode, possibly the only mode for Gradle build execution. The configuration on demand feature is incubating so not every build is guaranteed to work correctly. The feature should work very well for multi-project builds that have [decoupled projects](#2gldjl3). In “configuration on demand” mode, projects are configured as follows:

* The root project is always configured. This way the typical common configuration is supported (allprojects or subprojects script blocks).
* 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 makes relevant projects configured. If project A has a compile dependency on project B then building A causes configuration of both projects.
* The task dependencies declared via task path are supported and cause relevant projects to be configured. Example: someTask.dependsOn(":someOtherProject:someOtherTask")
* A task requested via task path from the command line (or Tooling API) causes the relevant project to be configured. For example, building 'projectA:projectB:someTask' causes configuration of projectB.

Eager to try out this new feature? To configure on demand with every build run see [Gradle properties](#1idq7dh). To configure on demand just for a given build, see [command-line performance-oriented options](#356xmb2).

[Defining common behavior](#vqntsw)

Let’s look at some examples with the following project tree. This is a multi-project build with a root project named water and a subproject named bluewhale.

[Example: Multi-project tree - water & bluewhale projects](#3fqbcgp)

**Project layout**

.  
├── bluewhale/  
├── build.gradle  
└── settings.gradle

| **Note** | The code for this example can be found at samples/userguide/multiproject/firstExample/water in the ‘-all’ distribution of Gradle. |
| --- | --- |

**settings.gradle**

rootProject.name = 'water'  
include 'bluewhale'

And where is the build script for the bluewhale project? In Gradle build scripts are optional. Obviously for a single project build, a project without a build script doesn’t make much sense. For multiproject builds the situation is different. Let’s look at the build script for the water project and execute it:

[Example: Build script of water (parent) project](#1uvlmoi)

**build.gradle**

Closure cl = { task -> println "I'm $task.project.name" }  
task('hello').doLast(cl)  
project(':bluewhale') {  
 task('hello').doLast(cl)  
}

**Output of** gradle -q hello

> gradle -q hello  
I'm water  
I'm bluewhale

Gradle allows you to access any project of the multi-project build from any build script. The Project API provides a method called project(), which takes a path as an argument and returns the Project object for this path. The capability to configure a project build from any build script we call *cross project configuration*. Gradle implements this via *configuration injection*.

We are not that happy with the build script of the water project. It is inconvenient to add the task explicitly for every project. We can do better. Let’s first add another project called krill to our multi-project build.

[Example: Multi-project tree - water, bluewhale & krill projects](#4ev95cb)

**Project layout**

.  
├── bluewhale/  
├── build.gradle  
├── krill/  
└── settings.gradle

| **Note** | The code for this example can be found at samples/userguide/multiproject/addKrill/water in the ‘-all’ distribution of Gradle. |
| --- | --- |

**settings.gradle**

rootProject.name = 'water'  
  
include 'bluewhale', 'krill'

Now we rewrite the water build script and boil it down to a single line.

[Example: Water project build script](#2u0jfk4)

**build.gradle**

allprojects {  
 task hello {  
 doLast { task ->  
 println "I'm $task.project.name"  
 }  
 }  
}

**Output of** gradle -q hello

> gradle -q hello  
I'm water  
I'm bluewhale  
I'm krill

Is this cool or is this cool? And how does this work? The Project API provides a property allprojects which returns a list with the current project and all its subprojects underneath it. If you call allprojects with a closure, the statements of the closure are delegated to the projects associated with allprojects. You could also do an iteration via allprojects.each, but that would be more verbose.

Other build systems use inheritance as the primary means for defining common behavior. We also offer inheritance for projects as you will see later. But Gradle uses configuration injection as the usual way of defining common behavior. We think it provides a very powerful and flexible way of configuring multiproject builds.

Another possibility for sharing configuration is to use a common external script.

[Subproject configuration](#195tprx)

The Project API also provides a property for accessing the subprojects only.

[Defining common behavior](#3t5h8fq)

[Example: Defining common behavior of all projects and subprojects](#28arinj)

**build.gradle**

allprojects {  
 task hello {  
 doLast { task ->  
 println "I'm $task.project.name"  
 }  
 }  
}  
subprojects {  
 hello {  
 doLast {  
 println "- I depend on water"  
 }  
 }  
}

**Output of** gradle -q hello

> gradle -q hello  
I'm water  
I'm bluewhale  
- I depend on water  
I'm krill  
- I depend on water

You may notice that there are two code snippets referencing the “hello” task. The first one, which uses the “task” keyword, constructs the task and provides it’s base configuration. The second piece doesn’t use the “task” keyword, as it is further configuring the existing “hello” task. You may only construct a task once in a project, but you may add any number of code blocks providing additional configuration.

[Adding specific behavior](#ng1svc)

You can add specific behavior on top of the common behavior. Usually we put the project specific behavior in the build script of the project where we want to apply this specific behavior. But as we have already seen, we don’t have to do it this way. We could add project specific behavior for the bluewhale project like this:

[Example: Defining specific behaviour for particular project](#37fpbj5)

**build.gradle**

allprojects {  
 task hello {  
 doLast { task ->  
 println "I'm $task.project.name"  
 }  
 }  
}  
subprojects {  
 hello {  
 doLast {  
 println "- I depend on water"  
 }  
 }  
}  
project(':bluewhale').hello {  
 doLast {  
 println "- I'm the largest animal that has ever lived on this planet."  
 }  
}

**Output of** gradle -q hello

> gradle -q hello  
I'm water  
I'm bluewhale  
- I depend on water  
- I'm the largest animal that has ever lived on this planet.  
I'm krill  
- I depend on water

As we have said, we usually prefer to put project specific behavior into the build script of this project. Let’s refactor and also add some project specific behavior to the krill project.

[Example: Defining specific behaviour for project krill](#1mkzlqy)

**Project layout**

.  
├── bluewhale  
│   └── build.gradle  
├── build.gradle  
├── krill  
│   └── build.gradle  
└── settings.gradle

| **Note** | The code for this example can be found at samples/userguide/multiproject/spreadSpecifics/water in the ‘-all’ distribution of Gradle. |
| --- | --- |

**settings.gradle**

rootProject.name = 'water'  
include 'bluewhale', 'krill'

**bluewhale/build.gradle**

hello.doLast {  
 println "- I'm the largest animal that has ever lived on this planet."  
}

**krill/build.gradle**

hello.doLast {  
 println "- The weight of my species in summer is twice as heavy as all human beings."  
}

**build.gradle**

allprojects {  
 task hello {  
 doLast { task ->  
 println "I'm $task.project.name"  
 }  
 }  
}  
subprojects {  
 hello {  
 doLast {  
 println "- I depend on water"  
 }  
 }  
}

**Output of gradle -q hello**

> gradle -q hello  
I'm water  
I'm bluewhale  
- I depend on water  
- I'm the largest animal that has ever lived on this planet.  
I'm krill  
- I depend on water  
- The weight of my species in summer is twice as heavy as all human beings.

[Project filtering](#46kn4er)

To show more of the power of configuration injection, let’s add another project called tropicalFish and add more behavior to the build via the build script of the water project.

[Filtering by name](#2lpxemk)

[Example: Adding custom behaviour to some projects (filtered by project name)](#10v7oud)

**Project layout**

.  
├── bluewhale/  
│   └── build.gradle  
├── build.gradle  
├── krill/  
│   └── build.gradle  
├── settings.gradle  
└── tropicalFish/

| **Note** | The code for this example can be found at samples/userguide/multiproject/addTropical/water in the ‘-all’ distribution of Gradle. |
| --- | --- |

**settings.gradle**

rootProject.name = 'water'  
include 'bluewhale', 'krill', 'tropicalFish'

**build.gradle**

allprojects {  
 task hello {  
 doLast { task ->  
 println "I'm $task.project.name"  
 }  
 }  
}  
subprojects {  
 hello {  
 doLast {  
 println "- I depend on water"  
 }  
 }  
}  
configure(subprojects.findAll {it.name != 'tropicalFish'}) {  
 hello {  
 doLast {  
 println '- I love to spend time in the arctic waters.'  
 }  
 }  
}

**Output of gradle -q hello**

> gradle -q hello  
I'm water  
I'm bluewhale  
- I depend on water  
- I love to spend time in the arctic waters.  
- I'm the largest animal that has ever lived on this planet.  
I'm krill  
- I depend on water  
- I love to spend time in the arctic waters.  
- The weight of my species in summer is twice as heavy as all human beings.  
I'm tropicalFish  
- I depend on water

The configure() method takes a list as an argument and applies the configuration to the projects in this list.

[Filtering by properties](#3kuv7i6)

Using the project name for filtering is one option. Using [extra project properties](#2emvufp) is another. for more information on extra properties.)

[Example: Adding custom behaviour to some projects (filtered by project properties)](#2005hpz)

**Project layout**

.  
├── bluewhale  
│   └── build.gradle  
├── build.gradle  
├── krill  
│   └── build.gradle  
├── settings.gradle  
└── tropicalFish  
 └── build.gradle

| **Note** | The code for this example can be found at samples/userguide/multiproject/tropicalWithProperties/water in the ‘-all’ distribution of Gradle. |
| --- | --- |

**settings.gradle**

rootProject.name = 'water'  
include 'bluewhale', 'krill', 'tropicalFish'

**bluewhale/build.gradle**

ext.arctic = true  
hello.doLast {  
 println "- I'm the largest animal that has ever lived on this planet."  
}

**krill/build.gradle**

ext.arctic = true  
hello.doLast {  
 println "- The weight of my species in summer is twice as heavy as all human beings."  
}

**build.gradle**

allprojects {  
 task hello {  
 doLast { task ->  
 println "I'm $task.project.name"  
 }  
 }  
}  
subprojects {  
 hello {  
 doLast {println "- I depend on water"}  
 afterEvaluate { Project project ->  
 **if** (project.arctic) { doLast {  
 println '- I love to spend time in the arctic waters.' }  
 }  
 }  
 }  
}

**tropicalFish/build.gradle**

ext.arctic = false

**Output of gradle -q hello**

> gradle -q hello  
I'm water  
I'm bluewhale  
- I depend on water  
- I'm the largest animal that has ever lived on this planet.  
- I love to spend time in the arctic waters.  
I'm krill  
- I depend on water  
- The weight of my species in summer is twice as heavy as all human beings.  
- I love to spend time in the arctic waters.  
I'm tropicalFish  
- I depend on water

In the build file of the water project we use an afterEvaluate notification. This means that the closure we are passing gets evaluated *after* the build scripts of the subproject are evaluated. As the property arctic is set in those build scripts, we have to do it this way. You will find more on this topic in [Dependencies — Which Dependencies?](#4jzt0ds)

[Execution rules for multi-project builds](#2z53all)

When we executed the hello task from the root project dir, things behaved in an intuitive way. All the hello tasks of the different projects were executed. Let’s switch to the bluewhale dir and see what happens if we execute Gradle from there.

[Example: Running build from subproject](#1eadkte)

**Output of** gradle -q hello

> gradle -q hello  
I'm bluewhale  
- I depend on water  
- I'm the largest animal that has ever lived on this planet.  
- I love to spend time in the arctic waters.

The basic rule behind Gradle’s behavior is simple. Gradle looks down the hierarchy, starting with the *current dir*, for tasks with the name hello and executes them. One thing is very important to note. Gradle *always* evaluates *every* project of the multi-project build and creates all existing task objects. Then, according to the task name arguments and the current dir, Gradle filters the tasks which should be executed. Because of Gradle’s cross project configuration *every* project has to be evaluated before *any* task gets executed. We will have a closer look at this in the next section. Let’s now have our last marine example. Let’s add a task to bluewhale and krill.

[Example: Evaluation and execution of projects](#3ya13h7)

**bluewhale/build.gradle**

ext.arctic = true  
hello {  
 doLast {  
 println "- I'm the largest animal that has ever lived on this planet."  
 }  
}  
  
task distanceToIceberg {  
 doLast {  
 println '20 nautical miles'  
 }  
}

**krill/build.gradle**

ext.arctic = true  
hello {  
 doLast {  
 println "- The weight of my species in summer is twice as heavy as all human beings."  
 }  
}  
  
task distanceToIceberg {  
 doLast {  
 println '5 nautical miles'  
 }  
}

**Output of gradle -q distanceToIceberg**

> gradle -q distanceToIceberg  
20 nautical miles  
5 nautical miles

Here’s the output without the -q option:

[Example: Evaluation and execution of projects](#2dfbdp0)

**Output of** gradle distanceToIceberg

> gradle distanceToIceberg  
  
> Task :bluewhale:distanceToIceberg  
20 nautical miles  
  
> Task :krill:distanceToIceberg  
5 nautical miles  
  
BUILD SUCCESSFUL in 0s  
2 actionable tasks: 2 executed

The build is executed from the water project. Neither water nor tropicalFish have a task with the name distanceToIceberg. Gradle does not care. The simple rule mentioned already above is: Execute all tasks down the hierarchy which have this name. Only complain if there is *no* such task!

[Running tasks by their absolute path](#rtofi4)

As we have seen, you can run a multi-project build by entering any subproject dir and execute the build from there. All matching task names of the project hierarchy starting with the current dir are executed. But Gradle also offers to execute tasks by their absolute path (see also [Project and task paths](#sklnwt)):

[Example: Running tasks by their absolute path](#3ck96km)

**Output of** gradle -q :hello :krill:hello hello

> gradle -q :hello :krill:hello hello  
I'm water  
I'm krill  
- I depend on water  
- The weight of my species in summer is twice as heavy as all human beings.  
- I love to spend time in the arctic waters.  
I'm tropicalFish  
- I depend on water

The build is executed from the tropicalFish project. We execute the hello tasks of the water, the krill and the tropicalFish project. The first two tasks are specified by their absolute path, the last task is executed using the name matching mechanism described above.

[Project and task paths](#sklnwt)

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 that is 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.

The path of a task is simply its project path plus the task name, like “:bluewhale:hello”. Within a project you can address a task of the same project just by its name. This is interpreted as a relative path.

[Dependencies - Which dependencies?](#4jzt0ds)

The examples from the last section were special, as the projects had no *Execution Dependencies*. They had only *Configuration Dependencies*. The following sections illustrate the differences between these two types of dependencies.

[Execution dependencies](#1rpjgsf)

[Dependencies and execution order](#4bp6zg8)

[Example: Dependencies and execution order](#2quh9o1)

**Project layout**

.  
├── build.gradle  
├── consumer  
│   └── build.gradle  
├── producer  
│   └── build.gradle  
└── settings.gradle

| **Note** | The code for this example can be found at samples/userguide/multiproject/dependencies/firstMessages/messages in the ‘-all’ distribution of Gradle. |
| --- | --- |

**build.gradle**

ext.producerMessage = null

**settings.gradle**

include 'consumer', 'producer'

**consumer/build.gradle**

task action {  
 doLast {  
 println("Consuming message: ${rootProject.producerMessage}")  
 }  
}

**producer/build.gradle**

task action {  
 doLast {  
 println "Producing message:"  
 rootProject.producerMessage = 'Watch the order of execution.'  
 }  
}

**Output of gradle -q action**

> gradle -q action  
Consuming message: null  
Producing message:

This didn’t quite do what we want. If nothing else is defined, Gradle executes the task in alphanumeric order. Therefore, Gradle will execute “:consumer:action” before “:producer:action”. Let’s try to solve this with a hack and rename the producer project to “aProducer”.

[Example: Dependencies and execution order](#15zrjvu)

**Project layout**

.  
├── aProducer  
│   └── build.gradle  
├── build.gradle  
├── consumer  
│   └── build.gradle  
└── settings.gradle

**build.gradle**

ext.producerMessage = null

**settings.gradle**

include 'consumer', 'aProducer'

**aProducer/build.gradle**

task action {  
 doLast {  
 println "Producing message:"  
 rootProject.producerMessage = 'Watch the order of execution.'  
 }  
}

**consumer/build.gradle**

task action {  
 doLast {  
 println("Consuming message: ${rootProject.producerMessage}")  
 }  
}

**Output of gradle -q action**

> gradle -q action  
Producing message:  
Consuming message: Watch the order of execution.

We can show where this hack doesn’t work if we now switch to the consumer dir and execute the build.

[Example: Dependencies and execution order](#3pzf2jn)

**Output of** gradle -q action

> gradle -q action  
Consuming message: null

The problem is that the two “action” tasks are unrelated. If you execute the build from the “messages” project Gradle executes them both because they have the same name and they are down the hierarchy. In the last example only one “action” task was down the hierarchy and therefore it was the only task that was executed. We need something better than this hack.

[Real life examples](#254pcrg)

Gradle’s multi-project features are driven by real life use cases. One good example consists of two web application projects and a parent project that creates a distribution including the two web applications.[[4](#k9zmz9)] For the example we use only one build script and do *cross project configuration*.

[Example: Dependencies - real life example - crossproject configuration](#349n5n2)

**Project layout**

.  
├── build.gradle  
├── date  
│   └── src  
│   └── main  
│   ├── java  
│   │   └── org  
│   │   └── gradle  
│   │   └── sample  
│   │   └── DateServlet.java  
│   └── webapp  
│   └── web.xml  
├── hello  
│   └── src  
│   └── main  
│   ├── java  
│   │   └── org  
│   │   └── gradle  
│   │   └── sample  
│   │   └── HelloServlet.java  
│   └── webapp  
│   └── web.xml  
└── settings.gradle

| **Note** | The code for this example can be found at samples/userguide/multiproject/dependencies/webDist in the ‘-all’ distribution of Gradle. |
| --- | --- |

**settings.gradle**

include 'date', 'hello'

**build.gradle**

allprojects {  
 apply plugin: 'java'  
 group = 'org.gradle.sample'  
 version = '1.0'  
}  
  
subprojects {  
 apply plugin: 'war'  
 repositories {  
 mavenCentral()  
 }  
 dependencies {  
 compile "javax.servlet:servlet-api:2.5"  
 }  
}  
  
task explodedDist(type: Copy) {  
 into "$buildDir/explodedDist"  
 subprojects {  
 from tasks.withType(War)  
 }  
}

We have an interesting set of dependencies. Obviously the date and hello projects have a *configuration* dependency on webDist, as all the build logic for the webapp projects is injected by webDist. The *execution* dependency is in the other direction, as webDist depends on the build artifacts of date and hello. There is even a third dependency. webDist has a *configuration* dependency on date and hello because it needs to know the archivePath. But it asks for this information at *execution time*. Therefore we have no circular dependency.

Such dependency patterns are daily bread in the problem space of multi-project builds. If a build system does not support these patterns, you either can’t solve your problem or you need to do ugly hacks which are hard to maintain and massively impair your productivity as a build master.

[Project lib dependencies](#1jexfuv)

What if one project needs the jar produced by another project in its compile path, and not just the jar but also the transitive dependencies of this jar? Obviously this is a very common use case for Java multi-project builds. As mentioned in [Project dependencies](#43ekyio), Gradle offers project lib dependencies for this.

**Project lib dependencies**

.  
├── api  
│   └── src  
│   ├── main  
│   │   └── java  
│   │   └── org  
│   │   └── gradle  
│   │   └── sample  
│   │   ├── api  
│   │   │   └── Person.java  
│   │   └── apiImpl  
│   │   └── PersonImpl.java  
│   └── test  
│   └── java  
│   └── org  
│   └── gradle  
│   └── PersonTest.java  
├── build.gradle  
├── services  
│   └── personService  
│   └── src  
│   ├── main  
│   │   └── java  
│   │   └── org  
│   │   └── gradle  
│   │   └── sample  
│   │   └── services  
│   │   └── PersonService.java  
│   └── test  
│   └── java  
│   └── org  
│   └── gradle  
│   └── sample  
│   └── services  
│   └── PersonServiceTest.java  
├── settings.gradle  
└── shared  
 └── src  
 └── main  
 └── java  
 └── org  
 └── gradle  
 └── sample  
 └── shared  
 └── Helper.java

| **Note** | The code for this example can be found at samples/userguide/multiproject/dependencies/java in the ‘-all’ distribution of Gradle. |
| --- | --- |

We have the projects “shared”, “api” and “personService”. The “personService” project has a lib dependency on the other two projects. The “api” project has a lib dependency on the “shared” project. “services” is also a project, but we use it just as a container. It has no build script and gets nothing injected by another build script. We use the : separator to define a project path. Consult the DSL documentation of [Settings.include(java.lang.String[])](http://docs.google.com/dsl/org.gradle.api.initialization.Settings.html#org.gradle.api.initialization.Settings:include(java.lang.String%5B%5D)) for more information about defining project paths.

[Example: Project lib dependencies](#2ijv8qh)

**settings.gradle**

include 'api', 'shared', 'services:personService'

**build.gradle**

subprojects {  
 apply plugin: 'java'  
 group = 'org.gradle.sample'  
 version = '1.0'  
 repositories {  
 mavenCentral()  
 }  
 dependencies {  
 testCompile "junit:junit:4.12"  
 }  
}  
  
project(':api') {  
 dependencies {  
 compile project(':shared')  
 }  
}  
  
project(':services:personService') {  
 dependencies {  
 compile project(':shared'), project(':api')  
 }  
}

All the build logic is in the “build.gradle” file of the root project.[[5](#xp5iya)] A “*lib*” dependency is a special form of an execution dependency. It causes the other project to be built first and adds the jar with the classes of the other project to the classpath. It also adds the dependencies of the other project to the classpath. So you can enter the “api” directory and trigger a “gradle compile”. First the “shared” project is built and then the “api” project is built. Project dependencies enable partial multi-project builds.

If you come from Maven land you might be perfectly happy with this. If you come from Ivy land, you might expect some more fine grained control. Gradle offers this to you:

[Example: Fine grained control over dependencies](#3hot1m3)

**build.gradle**

subprojects {  
 apply plugin: 'java'  
 group = 'org.gradle.sample'  
 version = '1.0'  
}  
  
project(':api') {  
 configurations {  
 spi  
 }  
 dependencies {  
 compile project(':shared')  
 }  
 task spiJar(type: Jar) {  
 baseName = 'api-spi'  
 from sourceSets.main.output  
 include('org/gradle/sample/api/\*\*')  
 }  
 artifacts {  
 spi spiJar  
 }  
}  
  
project(':services:personService') {  
 dependencies {  
 compile project(':shared')  
 compile project(path: ':api', configuration: 'spi')  
 testCompile "junit:junit:4.12", project(':api')  
 }  
}

The Java plugin adds per default a jar to your project libraries which contains all the classes. In this example we create an *additional* library containing only the interfaces of the “api” project. We assign this library to a new *dependency configuration*. For the person service we declare that the project should be compiled only against the “api” interfaces but tested with all classes from “api”.

[Depending on the task output produced by another project](#1wu3btw)

[Project dependencies](#1jexfuv) model dependencies between modules. Effectively, you are saying that you depend on the main output of another project. In a Java-based project that’s usually a JAR file.

Sometimes you may want to depend on an output produced by another task. In turn you’ll want to make sure that the task is executed beforehand to produce that very output. Declaring a task dependency from one project to another is a poor way to model this kind of relationship and introduces unnecessary coupling. The recommended way to model such a dependency is to produce the output, mark it as an "outgoing" artifact or add it to the output of the main source set which you can depend on in the consuming project.

Let’s say you are working in a multi-project build with the two subprojects producer and consumer. The subproject producer defines a task named buildInfo that generates a properties file containing build information e.g. the project version. The attribute builtBy takes care of establishing an inferred task dependency. For more information on builtBy, see [SourceSetOutput](http://docs.google.com/dsl/org.gradle.api.tasks.SourceSetOutput.html).

[Example: Task generating a property file containing build information](#4gtquhp)

**build.gradle**

task buildInfo(type: BuildInfo) {  
 version = project.version  
 outputFile = file("${buildDir}/generated-resources/build-info.properties")  
}  
  
sourceSets {  
 main {  
 output.dir(buildInfo.outputFile.parentFile, builtBy: buildInfo)  
 }  
}

The consuming project is supposed to be able to read the properties file at runtime. Declaring a project dependency on the producing project takes care of creating the properties beforehand and making it available to the runtime classpath.

[Example: Declaring a project dependency on the project producing the properties file](#2vz14pi)

**build.gradle**

dependencies {  
 runtime project(':producer')  
}

In the example above, the consumer now declares a dependency on the outputs of the producer project.

[Parallel project execution](#44bvf6o)

With more and more CPU cores available on developer desktops and CI servers, it is important that Gradle is able to fully utilise these processing resources. More specifically, parallel execution attempts to:

* Reduce total build time for a multi-project build where execution is IO bound or otherwise does not consume all available CPU resources.
* Provide faster feedback for execution of small projects without awaiting completion of other projects.

Although Gradle already offers parallel test execution via [Test.setMaxParallelForks(int)](http://docs.google.com/javadoc/org/gradle/api/tasks/testing/Test.html#setMaxParallelForks-int-) the feature described in this section is parallel execution at a project level. Parallel execution is an incubating feature. Please use it and let us know how it works for you.

Parallel project execution allows the separate projects in a decoupled multi-project build to be executed in parallel (see also [Decoupled projects](#2gldjl3)). While parallel execution does not strictly require decoupling at configuration time, the long-term goal is to provide a powerful set of features that will be available for fully decoupled projects. Such features include:

* [Configuration on-demand](#1kc7wiv).
* Configuration of projects in parallel.
* Re-use of configuration for unchanged projects.
* Project-level up-to-date checks.
* Using pre-built artifacts in the place of building dependent projects.

How does parallel execution work? First, you need to tell Gradle to use parallel mode. You can use the [--parallel command line argument](#356xmb2) or configure your build environment ([Gradle properties](#1idq7dh)). Unless you provide a specific number of parallel threads, Gradle attempts to choose the right number based on available CPU cores. Every parallel worker exclusively owns a given project while executing a task. Task dependencies are fully supported and parallel workers will start executing upstream tasks first. Bear in mind that the alphabetical ordering of decoupled tasks, as can be seen during sequential execution, is not guaranteed in parallel mode. In other words, in parallel mode tasks will run as soon as their dependencies complete *and a task worker is available to run them*, which may be earlier than they would start during a sequential build. You should make sure that task dependencies and task inputs/outputs are declared correctly to avoid ordering issues.

[Decoupled Projects](#2gldjl3)

Gradle allows any project to access any other project during both the configuration and execution phases. While this provides a great deal of power and flexibility to the build author, it also limits the flexibility that Gradle has when building those projects. For instance, this effectively prevents Gradle from correctly building multiple projects in parallel, configuring only a subset of projects, or from substituting a pre-built artifact in place of a project dependency.

Two projects are said to be *decoupled* if they do not directly access each other’s project model. Decoupled projects may only interact in terms of declared dependencies: [project dependencies](#43ekyio) and/or [task dependencies](#3gnlt4p). Any other form of project interaction (i.e. by modifying another project object or by reading a value from another project object) causes the projects to be coupled. The consequence of coupling during the configuration phase is that if gradle is invoked with the 'configuration on demand' option, the result of the build can be flawed in several ways. The consequence of coupling during execution phase is that if gradle is invoked with the parallel option, one project task runs too late to influence a task of a project building in parallel. Gradle does not attempt to detect coupling and warn the user, as there are too many possibilities to introduce coupling.

A very common way for projects to be coupled is by using [configuration injection](#1hgfqph). It may not be immediately apparent, but using key Gradle features like the allprojects and subprojects keywords automatically cause your projects to be coupled. This is because these keywords are used in a build.gradle file, which defines a project. Often this is a “root project” that does nothing more than define common configuration, but as far as Gradle is concerned this root project is still a fully-fledged project, and by using allprojects that project is effectively coupled to all other projects. Coupling of the root project to subprojects does not impact 'configuration on demand', but using the allprojects and subprojects in any subproject’s build.gradle file will have an impact.

This means that using any form of shared build script logic or configuration injection (allprojects, subprojects, etc.) will cause your projects to be coupled. As we extend the concept of project decoupling and provide features that take advantage of decoupled projects, we will also introduce new features to help you to solve common use cases (like configuration injection) without causing your projects to be coupled.

In order to make good use of cross project configuration without running into issues for parallel and 'configuration on demand' options, follow these recommendations:

* Avoid a subproject’s build.gradle referencing other subprojects; preferring cross configuration from the root project.
* Avoid changing the configuration of other projects at execution time.

[Multi-Project Building and Testing](#1b4bexb)

The build task of the Java plugin is typically used to compile, test, and perform code style checks (if the CodeQuality plugin is used) of a single project. In multi-project builds you may often want to do all of these tasks across a range of projects. The buildNeeded and buildDependents tasks can help with this.

In [this example](#2ijv8qh), the “:services:personservice” project depends on both the “:api” and “:shared” projects. The “:api” project also depends on the “:shared” project.

Assume 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 project you have changed. The build task does this.

[Example: Build and Test Single Project](#3v3yxl4)

**Output of** gradle :api:build

> 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  
9 actionable tasks: 9 executed

If you have just gotten the latest version of source from your version control system which included changes in other projects that “:api” depends on, you might want to not only build all the projects you depend on, but test them as well. The buildNeeded task also tests all the projects from the project lib dependencies of the testRuntime configuration.

[Example: Build and Test Depended On Projects](#2a997sx)

**Output of** gradle :api:buildNeeded

> 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  
12 actionable tasks: 12 executed

You also might want to refactor some part of the “:api” project that is used in other projects. If you make these types of changes, it is not sufficient to test just the “:api” project, you also need to test all projects that depend on the “:api” project. The buildDependents task also tests all the projects that have a project lib dependency (in the testRuntime configuration) on the specified project.

[Example: Build and Test Dependent Projects](#peji0q)

**Output of** gradle :api:buildDependents

> 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:personService:compileJava  
> Task :services:personService:processResources  
> Task :services:personService:classes  
> Task :services:personService:jar  
> Task :services:personService:assemble  
> Task :services:personService:compileTestJava  
> Task :services:personService:processTestResources  
> Task :services:personService:testClasses  
> Task :services:personService:test  
> Task :services:personService:check  
> Task :services:personService:build  
> Task :services:personService:buildDependents  
> Task :api:buildDependents  
  
BUILD SUCCESSFUL in 0s  
17 actionable tasks: 17 executed

Finally, you may want to 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. So you can just run “gradle build” to build and test all projects.

[Multi Project and buildSrc](#39e70oj)

[Using buildSrc to organize build logic](#xevivl) tells us that we can place build logic to be compiled and tested in the special buildSrc directory. In a multi project build, there can only be one buildSrc directory which must be located in the root directory.

[Authoring Tasks](#147n2zr)

In the [introductory tutorial](#1pxezwc) you learned how to create simple tasks. You also learned how to add additional behavior to these tasks later on, and you learned how to create dependencies between tasks. This was all about simple tasks, but Gradle takes the concept of tasks further. Gradle supports *enhanced tasks*, which are tasks that have their own properties and methods. This is really different from what you are used to with Ant targets. Such enhanced tasks are either provided by you or built into Gradle.

[Task outcomes](#2xn8ts7)

When Gradle executes a task, it can label the task with different outcomes in the console UI and via the [Tooling API](#184mhaj). These labels are based on if a task has actions to execute, if it should execute those actions, if it did execute those actions and if those actions made any changes.

(no label) or EXECUTED

Task executed its actions.

* Task has actions and Gradle has determined they should be executed as part of a build.
* Task has no actions and some dependencies, and any of the dependencies are executed. See also [Lifecycle Tasks](#34qadz2).

UP-TO-DATE

Task’s outputs did not change.

* Task has outputs and inputs and they have not changed. See [Incremental Builds](#odc9jc).
* Task has actions, but the task tells Gradle it did not change its outputs.
* Task has no actions and some dependencies, but all of the dependencies are up-to-date, skipped or from cache. See also [Lifecycle Tasks](#34qadz2).
* 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](#2bn6wsx).

SKIPPED

Task did not execute its actions.

* Task has been explicitly excluded from the command-line. See [Excluding tasks from execution](#1a346fx).
* Task has an onlyIf predicate return false. See [Using a predicate](#1ojhawc).

NO-SOURCE

Task did not need to execute its actions.

* Task has inputs and outputs, but [no sources](#48j4tk5). For example, source files are .java files for [JavaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.JavaCompile.html).

[Defining tasks](#2nof3ry)

We have already seen how to define tasks using a keyword style in [this chapter](#1pxezwc). There are a few variations on this style, which you may need to use in certain situations. For example, the keyword style does not work in expressions.

[Example: Defining tasks](#12tpdzr)

**build.gradle**

task(hello) {  
 doLast {  
 println "hello"  
 }  
}  
  
task(copy, type: Copy) {  
 from(file('srcDir'))  
 into(buildDir)  
}

You can also use strings for the task names:

[Example: Defining tasks - using strings for task names](#3mtcwnk)

**build.gradle**

task('hello') {  
 doLast {  
 println "hello"  
 }  
}  
  
task('copy', type: Copy) {  
 from(file('srcDir'))  
 into(buildDir)  
}

There is an alternative syntax for defining tasks, which you may prefer to use:

[Example: Defining tasks with alternative syntax](#21yn6vd)

**build.gradle**

tasks.create('hello') {  
 doLast {  
 println "hello"  
 }  
}  
  
tasks.create('copy', Copy) {  
 from(file('srcDir'))  
 into(buildDir)  
}

Here we add tasks to the tasks collection. Have a look at [TaskContainer](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskContainer.html) for more variations of the create() method.

[Locating tasks](#h3xh36)

You often need to locate the tasks that you have defined in the build file, for example, to configure them or use them for dependencies. There are a number of ways of doing this. Firstly, each task is available as a property of the project, using the task name as the property name:

[Example: Accessing tasks as properties](#313kzqz)

**build.gradle**

task hello  
  
println hello.name  
println project.hello.name

Tasks are also available through the tasks collection.

[Example: Accessing tasks via tasks collection](#1g8v9ys)

**build.gradle**

task hello  
  
println tasks.hello.name  
println tasks['hello'].name

You can access tasks from any project using the task’s path using the tasks.getByPath() method. You can call the getByPath() method with a task name, or a relative path, or an absolute path.

[Example: Accessing tasks by path](#408isml)

**build.gradle**

project(':projectA') {  
 task hello  
}  
  
task hello  
  
println tasks.getByPath('hello').path  
println tasks.getByPath(':hello').path  
println tasks.getByPath('projectA:hello').path  
println tasks.getByPath(':projectA:hello').path

**Output of** gradle -q hello

> gradle -q hello  
:hello  
:hello  
:projectA:hello  
:projectA:hello

Have a look at [TaskContainer](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskContainer.html) for more options for locating tasks.

[Configuring tasks](#2fdt2ue)

As an example, let’s look at the Copy task provided by Gradle. To create a Copy task for your build, you can declare in your build script:

[Example: Creating a copy task](#uj3d27)

**build.gradle**

task myCopy(type: Copy)

This creates a copy task with no default behavior. The task can be configured using its API (see [Copy](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html)). The following examples show several different ways to achieve the same configuration.

Just to be clear, realize that the name of this task is “myCopy”, but it is of *type* “Copy”. You can have multiple tasks of the same *type*, but with different names. You’ll find this gives you a lot of power to implement cross-cutting concerns across all tasks of a particular type.

[Example: Configuring a task - various ways](#3eiqvq0)

**build.gradle**

Copy myCopy = task(myCopy, type: Copy)  
myCopy.from 'resources'  
myCopy.into 'target'  
myCopy.include('\*\*/\*.txt', '\*\*/\*.xml', '\*\*/\*.properties')

This is similar to the way we would configure objects in Java. You have to repeat the context (myCopy) in the configuration statement every time. This is a redundancy and not very nice to read.

There is another way of configuring a task. It also preserves the context and it is arguably the most readable. It is usually our favorite.

[Example: Configuring a task - with closure](#1to15xt)

**build.gradle**

task myCopy(type: Copy)  
  
myCopy {  
 from 'resources'  
 into 'target'  
 include('\*\*/\*.txt', '\*\*/\*.xml', '\*\*/\*.properties')  
}

This works for *any* task. Line 3 of the example is just a shortcut for the tasks.getByName() method. It is important to note that if you pass a closure to the getByName() method, this closure is applied to *configure* the task, not when the task executes.

You can also use a configuration closure when you define a task.

[Example: Defining a task with closure](#4dnoolm)

**build.gradle**

task copy(type: Copy) {  
 from 'resources'  
 into 'target'  
 include('\*\*/\*.txt', '\*\*/\*.xml', '\*\*/\*.properties')  
}

| **Tip** | Don’t forget about the build phases  A task has both configuration and actions. When using the doLast, you are simply using a shortcut to define an action. Code defined in the configuration section of your task will get executed during the configuration phase of the build regardless of what task was targeted. See [Build Lifecycle](#3as4poj) for more details about the build lifecycle. |
| --- | --- |

[Passing arguments to a task constructor](#2ssyytf)

As opposed to configuring the mutable properties of a Task after creation, you can pass argument values to the Task class’s constructor. In order to pass values to the Task constructor, you must annotate the relevant constructor with @javax.inject.Inject.

[Example: Task class with @Inject constructor](#17y9918)

**build.gradle**

**class** **CustomTask** **extends** DefaultTask {  
 **final** String message  
 **final** **int** number  
  
 @Inject  
 CustomTask(String message, **int** number) {  
 this.message = message  
 this.number = number  
 }

You can then create a task, passing the constructor arguments at the end of the parameter list.

[Example: Creating a task with constructor arguments using TaskContainer](#3rxwrp1)

**build.gradle**

tasks.create('myTask', CustomTask, 'hello', 42)

In a Groovy build script, you can create the task using constructorArgs.

[Example: Creating a task with constructor arguments using Map](#27371wu)

**build.gradle**

task myTask(type: CustomTask, constructorArgs: ['hello', 42])

In a Kotlin build script, you can pass constructor arguments using the reified extension function on the tasks TaskContainer.

[Example: Creating a task with constructor arguments using Kotlin DSL](#m8hc4n)

**build.gradle.kts**

open **class** **CustomTask** @Inject constructor(**private** val message: String, **private** val number: Int) : DefaultTask() {  
 @TaskAction fun run() = println("$message $number")  
}  
  
tasks.create<CustomTask>("myTask", "hello", 42)

In all circumstances, the values passed as constructor arguments must be non-null. If you attempt to pass a null value, Gradle will throw a NullPointerException indicating which runtime value is null.

[Adding dependencies to a task](#1nsk2hn)

There are several ways you can define the dependencies of a task. In [Task dependencies](#3gnlt4p) you were introduced to defining dependencies using task names. Task names can refer to tasks in the same project as the task, or to tasks in other projects. To refer to a task in another project, you prefix the name of the task with the path of the project it belongs to. The following is an example which adds a dependency from projectA:taskX to projectB:taskY:

[Example: Adding dependency on task from another project](#3684usg)

**build.gradle**

project('projectA') {  
 task taskX(dependsOn: ':projectB:taskY') {  
 doLast {  
 println 'taskX'  
 }  
 }  
}  
  
project('projectB') {  
 task taskY {  
 doLast {  
 println 'taskY'  
 }  
 }  
}

**Output of** gradle -q taskX

> gradle -q taskX  
taskY  
taskX

Instead of using a task name, you can define a dependency using a Task object, as shown in this example:

[Example: Adding dependency using task object](#1ldf509)

**build.gradle**

task taskX {  
 doLast {  
 println 'taskX'  
 }  
}  
  
task taskY {  
 doLast {  
 println 'taskY'  
 }  
}  
  
taskX.dependsOn taskY

**Output of** gradle -q taskX

> gradle -q taskX  
taskY  
taskX

For more advanced uses, you can define a task dependency using a closure. When evaluated, the closure is passed the task whose dependencies are being calculated. The closure should return a single Task or collection of Task objects, which are then treated as dependencies of the task. The following example adds a dependency from taskX to all the tasks in the project whose name starts with lib:

[Example: Adding dependency using closure](#45d2no2)

**build.gradle**

task taskX {  
 doLast {  
 println 'taskX'  
 }  
}  
  
taskX.dependsOn {  
 tasks.findAll { task -> task.name.startsWith('lib') }  
}  
  
task lib1 {  
 doLast {  
 println 'lib1'  
 }  
}  
  
task lib2 {  
 doLast {  
 println 'lib2'  
 }  
}  
  
task notALib {  
 doLast {  
 println 'notALib'  
 }  
}

**Output of** gradle -q taskX

> gradle -q taskX  
lib1  
lib2  
taskX

For more information about task dependencies, see the [Task](http://docs.google.com/dsl/org.gradle.api.Task.html) API.

[Ordering tasks](#2kicxvv)

| **Note** | Task ordering is an [incubating](#2xcytpi) feature. Please be aware that this feature may change in later Gradle versions. |
| --- | --- |

In some cases it is useful to control the *order* in which 2 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.

There are two ordering rules available: “*must run after*” and “*should run after*”.

When you use the “must run after” ordering rule you specify that taskB must always run after taskA, whenever both taskA and taskB will be run. This is expressed as taskB.mustRunAfter(taskA). The “should run after” ordering rule is similar but less strict as it will be ignored in two situations. Firstly if using that rule introduces an ordering cycle. Secondly when using parallel execution and all dependencies of a task have been satisfied apart from the “should run after” task, then this task will be run regardless of whether its “should run after” dependencies have been run or not. You should use “should run after” where the ordering is helpful but not strictly required.

With these rules present it is still possible to execute taskA without taskB and vice-versa.

[Example: Adding a 'must run after' task ordering](#znn83o)

**build.gradle**

task taskX {  
 doLast {  
 println 'taskX'  
 }  
}  
task taskY {  
 doLast {  
 println 'taskY'  
 }  
}  
taskY.mustRunAfter taskX

**Output of** gradle -q taskY taskX

> gradle -q taskY taskX  
taskX  
taskY

[Example: Adding a 'should run after' task ordering](#3jnaqrh)

**build.gradle**

task taskX {  
 doLast {  
 println 'taskX'  
 }  
}  
task taskY {  
 doLast {  
 println 'taskY'  
 }  
}  
taskY.shouldRunAfter taskX

**Output of** gradle -q taskY taskX

> gradle -q taskY taskX  
taskX  
taskY

In the examples above, it is still possible to execute taskY without causing taskX to run:

[Example: Task ordering does not imply task execution](#1ysl0za)

**Output of** gradle -q taskY

> gradle -q taskY  
taskY

To specify a “must run after” or “should run after” ordering between 2 tasks, you use the [Task.mustRunAfter(java.lang.Object...)](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:mustRunAfter(java.lang.Object%5B%5D)) and [Task.shouldRunAfter(java.lang.Object...)](http://docs.google.com/javadoc/org/gradle/api/Task.html#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...)](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:dependsOn(java.lang.Object%5B%5D)).

Note that “B.mustRunAfter(A)” or “B.shouldRunAfter(A)” does not imply any execution dependency between the tasks:

* It is possible to execute tasks A and B independently. The ordering rule only has an effect when both tasks are scheduled for execution.
* When run with --continue, it is possible for B to execute in the event that A fails.

As mentioned before, the “should run after” ordering rule will be ignored if it introduces an ordering cycle:

[Example: A 'should run after' task ordering is ignored if it introduces an ordering cycle](#4is8jn3)

**build.gradle**

task taskX {  
 doLast {  
 println 'taskX'  
 }  
}  
task taskY {  
 doLast {  
 println 'taskY'  
 }  
}  
task taskZ {  
 doLast {  
 println 'taskZ'  
 }  
}  
taskX.dependsOn taskY  
taskY.dependsOn taskZ  
taskZ.shouldRunAfter taskX

**Output of** gradle -q taskX

> gradle -q taskX  
taskZ  
taskY  
taskX

[Adding a description to a task](#2xxituw)

You can add a description to your task. This description is displayed when executing gradle tasks.

[Example: Adding a description to a task](#1d2t42p)

**build.gradle**

task copy(type: Copy) {  
 description 'Copies the resource directory to the target directory.'  
 from 'resources'  
 into 'target'  
 include('\*\*/\*.txt', '\*\*/\*.xml', '\*\*/\*.properties')  
}

[Replacing tasks](#3x2gmqi)

Sometimes you want to replace a task. For example, if you want to exchange a task added by the Java plugin with a custom task of a different type. You can achieve this with:

[Example: Overwriting a task](#2c7qwyb)

**build.gradle**

task copy(type: Copy)  
  
task copy(overwrite: true) {  
 doLast {  
 println('I am the new one.')  
 }  
}

**Output of** gradle -q copy

> gradle -q copy  
I am the new one.

This will replace a task of type Copy with the task you’ve defined, because it uses the same name. When you define the new task, you have to set the overwrite property to true. Otherwise Gradle throws an exception, saying that a task with that name already exists.

[Skipping tasks](#rd1764)

Gradle offers multiple ways to skip the execution of a task.

[Using a predicate](#1ojhawc)

You can use the onlyIf() method to attach a predicate to a task. The task’s actions are only executed if the predicate evaluates to true. You implement the predicate as a closure. The closure is passed the task as a parameter, and should return true if the task should execute and false if the task should be skipped. The predicate is evaluated just before the task is due to be executed.

[Example: Skipping a task using a predicate](#3bcoptx)

**build.gradle**

task hello {  
 doLast {  
 println 'hello world'  
 }  
}  
  
hello.onlyIf { !project.hasProperty('skipHello') }

**Output of** gradle hello -PskipHello

> gradle hello -PskipHello  
> Task :hello SKIPPED  
  
BUILD SUCCESSFUL in 0s

[Using StopExecutionException](#1qhz01q)

If the logic for skipping a task can’t be expressed with a predicate, you can use the [StopExecutionException](http://docs.google.com/javadoc/org/gradle/api/tasks/StopExecutionException.html). If this exception is thrown by an action, the further execution of this action as well as the execution of any following action of this task is skipped. The build continues with executing the next task.

[Example: Skipping tasks with StopExecutionException](#4ahmipj)

**build.gradle**

task compile {  
 doLast {  
 println 'We are doing the compile.'  
 }  
}  
  
compile.doFirst {  
 *// Here you would put arbitrary conditions in real life.*  
 *// But this is used in an integration test so we want defined behavior.*  
 **if** (true) { **throw** **new** StopExecutionException() }  
}  
task myTask(dependsOn: 'compile') {  
 doLast {  
 println 'I am not affected'  
 }  
}

**Output of** gradle -q myTask

> 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.[[6](#2pmwsxc)]

[Enabling and disabling tasks](#14s7355)

Every task has an enabled flag which defaults to true. Setting it to false prevents the execution of any of the task’s actions. A disabled task will be labelled SKIPPED.

[Example: Enabling and disabling tasks](#3orulsy)

**build.gradle**

task disableMe {  
 doLast {  
 println 'This should not be printed if the task is disabled.'  
 }  
}  
disableMe.enabled = false

**Output of** gradle disableMe

> gradle disableMe  
> Task :disableMe SKIPPED  
  
BUILD SUCCESSFUL in 0s

[Up-to-date checks (AKA Incremental Build)](#odc9jc)

An important part of any build tool is the ability to avoid doing work that has already been done. Consider the process of compilation. Once your source files have been compiled, there should be no need to recompile them unless something has changed that affects the output, such as the modification of a source file or the removal of an output file. And compilation can take a significant amount of time, so skipping the step when it’s not needed saves a lot of time.

Gradle supports this behavior out of the box through a feature it calls incremental build. You have almost certainly already seen it in action: it’s active nearly every time the UP-TO-DATE text appears next to the name of a task when you run a build. Task outcomes are described in [Task outcomes](#2xn8ts7).

How does incremental build work? And what does it take to make use of it in your own tasks? Let’s take a look.

[Task inputs and outputs](#4hae2tp)

In the most common case, a task takes some inputs and generates some outputs. If we use the compilation example from earlier, we can see that the source files are the inputs and, in the case of Java, the generated class files are the outputs. Other inputs might include things like whether debug information should be included.



*Figure 7. Example task inputs and outputs*

An important characteristic of an input is that it affects one or more outputs, as you can see from the previous figure. Different bytecode is generated depending on the content of the source files and the minimum version of the Java runtime you want to run the code on. That makes them task inputs. But whether compilation has 500MB or 600MB of maximum memory available, determined by the memoryMaximumSize property, has no impact on what bytecode gets generated. In Gradle terminology, memoryMaximumSize is just an internal task property.

As part of incremental build, Gradle tests whether any of the task inputs or outputs have changed since the last build. If they haven’t, Gradle can consider the task up to date and therefore skip executing its actions. Also note that incremental build won’t work unless a task has at least one task output, although tasks usually have at least one input as well.

What this means for build authors is simple: you need to tell Gradle which task properties are inputs and which are outputs. If a task property affects the output, be sure to register it as an input, otherwise the task will be considered up to date when it’s not. Conversely, don’t register properties as inputs if they don’t affect the output, otherwise the task will potentially execute when it doesn’t need to. Also be careful of non-deterministic tasks that may generate different output for exactly the same inputs: these should not be configured for incremental build as the up-to-date checks won’t work.

Let’s now look at how you can register task properties as inputs and outputs.

[Custom task types](#2j0ih2h)

If you’re implementing a custom task as a class, then it takes just two steps to make it work with incremental build:

1. Create typed properties (via getter methods) for each of your task inputs and outputs
2. Add the appropriate annotation to each of those properties

| **Note** | Annotations must be placed on getters or on Groovy properties. Annotations placed on setters, or on a Java field without a corresponding annotated getter are ignored. |
| --- | --- |

Gradle supports three main categories of inputs and outputs:

* Simple values  
  Things like strings and numbers. More generally, a simple value can have any type that implements Serializable.
* Filesystem types  
  These consist of the standard File class but also derivatives of Gradle’s [FileCollection](http://docs.google.com/javadoc/org/gradle/api/file/FileCollection.html) type and anything else that can be passed to either the [Project.file(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:file(java.lang.Object)) method - for single file/directory properties - or the [Project.files(java.lang.Object...)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:files(java.lang.Object%5B%5D)), [ProjectLayout.files(java.lang.Object…​)](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#files-java.lang.Object...-), and [ProjectLayout.configurableFiles(java.lang.Object…​)](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#configurableFiles-java.lang.Object...-) methods.
* Nested values  
  Custom types that don’t conform to the other two categories but have their own properties that are inputs or outputs. In effect, the task inputs or outputs are nested inside these custom types.

As an example, imagine you have a task that processes templates of varying types, such as FreeMarker, Velocity, Moustache, etc. It takes template source files and combines them with some model data to generate populated versions of the template files.

This task will have three inputs and one output:

* Template source files
* Model data
* Template engine
* Where the output files are written

When you’re writing a custom task class, it’s easy to register properties as inputs or outputs via annotations. To demonstrate, here is a skeleton task implementation with some suitable inputs and outputs, along with their annotations:

[Example: Custom task class](#23x4w0r)

**buildSrc/src/main/java/org/example/ProcessTemplates.java**

**package** org.example;  
  
**import** java.io.File;  
**import** java.util.HashMap;  
**import** org.gradle.api.\*;  
**import** org.gradle.api.file.\*;  
**import** org.gradle.api.tasks.\*;  
  
**public** **class** **ProcessTemplates** **extends** DefaultTask {  
 **private** TemplateEngineType templateEngine;  
 **private** FileCollection sourceFiles;  
 **private** TemplateData templateData;  
 **private** File outputDir;  
  
 @Input  
 **public** TemplateEngineType getTemplateEngine() {  
 **return** this.templateEngine;  
 }  
  
 @InputFiles  
 **public** FileCollection getSourceFiles() {  
 **return** this.sourceFiles;  
 }  
  
 @Nested  
 **public** TemplateData getTemplateData() {  
 **return** this.templateData;  
 }  
  
 @OutputDirectory  
 **public** File getOutputDir() { **return** this.outputDir; }  
  
 *// + setter methods for the above - assume we’ve defined them*  
  
 @TaskAction  
 **public** **void** processTemplates() {  
 *// ...*  
 }  
}

**buildSrc/src/main/java/org/example/TemplateData.java**

**package** org.example;  
  
**import** java.util.HashMap;  
**import** java.util.Map;  
**import** org.gradle.api.tasks.Input;  
  
**public** **class** **TemplateData** {  
 **private** String name;  
 **private** Map<String, String> variables;  
  
 **public** TemplateData(String name, Map<String, String> variables) {  
 this.name = name;  
 this.variables = **new** HashMap<>(variables);  
 }  
  
 @Input  
 **public** String getName() { **return** this.name; }  
  
 @Input  
 **public** Map<String, String> getVariables() {  
 **return** this.variables;  
 }  
}

**Output of gradle processTemplates**

> gradle processTemplates  
> Task :processTemplates  
  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

**Output of gradle processTemplates (run again)**

> gradle processTemplates  
> Task :processTemplates UP-TO-DATE  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 up-to-date

There’s plenty to talk about in this example, so let’s work through each of the input and output properties in turn:

* templateEngine  
  Represents which engine to use when processing the source templates, e.g. FreeMarker, Velocity, etc. You could implement this as a string, but in this case we have gone for a custom enum as it provides greater type information and safety. Since enums implement Serializable automatically, we can treat this as a simple value and use the @Input annotation, just as we would with a String property.
* sourceFiles  
  The source templates that the task will be processing. Single files and collections of files need their own special annotations. In this case, we’re dealing with a collection of input files and so we use the @InputFiles annotation. You’ll see more file-oriented annotations in a table later.
* templateData  
  For this example, we’re using a custom class to represent the model data. However, it does not implement Serializable, so we can’t use the @Input annotation. That’s not a problem as the properties within TemplateData - a string and a hash map with serializable type parameters - are serializable and can be annotated with @Input. We use @Nested on templateData to let Gradle know that this is a value with nested input properties.
* outputDir  
  The directory where the generated files go. As with input files, there are several annotations for output files and directories. A property representing a single directory requires @OutputDirectory. You’ll learn about the others soon.

These annotated properties mean that Gradle will skip the task if none of the source files, template engine, model data or generated files have changed since the previous time Gradle executed the task. This will often save a significant amount of time. You can learn how Gradle detects [changes later](#43v86uo).

This example is particularly interesting because it works with collections of source files. What happens if only one source file changes? Does the task process all the source files again or just the modified one? That depends on the task implementation. If the latter, then the task itself is incremental, but that’s a different feature to the one we’re discussing here. Gradle does help task implementers with this via its [incremental task inputs](#j2f68k) feature.

Now that you have seen some of the input and output annotations in practice, let’s take a look at all the annotations available to you and when you should use them. The table below lists the available annotations and the corresponding property type you can use with each one.

Table 1. Incremental build property type annotations

| **Annotation** | **Expected property type** | **Description** |
| --- | --- | --- |
| @[Input](http://docs.google.com/javadoc/org/gradle/api/tasks/Input.html) | Any Serializable type | A simple input value |
| @[InputFile](http://docs.google.com/javadoc/org/gradle/api/tasks/InputFile.html) | File\* | A single input file (not directory) |
| @[InputDirectory](http://docs.google.com/javadoc/org/gradle/api/tasks/InputDirectory.html) | File\* | A single input directory (not file) |
| @[InputFiles](http://docs.google.com/javadoc/org/gradle/api/tasks/InputFiles.html) | Iterable<File>\* | An iterable of input files and directories |
| @[Classpath](http://docs.google.com/javadoc/org/gradle/api/tasks/Classpath.html) | Iterable<File>\* | An iterable of input files and directories that represent a Java classpath. This allows the task to ignore irrelevant changes to the property, such as different names for the same files. It is similar to annotating the property @PathSensitive(RELATIVE) but it will ignore the names of JAR files directly added to the classpath, and it will consider changes in the order of the files as a change in the classpath. Gradle will inspect the contents of jar files on the classpath and ignore changes that do not affect the semantics of the classpath (such as file dates and entry order). See also [Using the classpath annotations](#3322owd).  Note: The @Classpath annotation was introduced in Gradle 3.2. To stay compatible with earlier Gradle versions, classpath properties should also be annotated with @InputFiles. |
| @[CompileClasspath](http://docs.google.com/javadoc/org/gradle/api/tasks/CompileClasspath.html) | Iterable<File>\* | An iterable of input files and directories that represent a Java compile classpath. This allows the task to ignore irrelevant changes that do not affect the API of the classes in classpath. See also [Using the classpath annotations](#3322owd).  The following kinds of changes to the classpath will be ignored:   * Changes to the path of jar or top level directories. * Changes to timestamps and the order of entries in Jars. * Changes to resources and Jar manifests, including adding or removing resources. * Changes to private class elements, such as private fields, methods and inner classes. * Changes to code, such as method bodies, static initializers and field initializers (except for constants). * Changes to debug information, for example when a change to a comment affects the line numbers in class debug information. * Changes to directories, including directory entries in Jars.  | **Note** | The @CompileClasspath annotation was introduced in Gradle 3.4. To stay compatible with Gradle 3.3 and 3.2, compile classpath properties should also be annotated with @Classpath. For compatibility with Gradle versions before 3.2 the property should also be annotated with @InputFiles. | | --- | --- | |
| @[OutputFile](http://docs.google.com/javadoc/org/gradle/api/tasks/OutputFile.html) | File\* | A single output file (not directory) |
| @[OutputDirectory](http://docs.google.com/javadoc/org/gradle/api/tasks/OutputDirectory.html) | File\* | A single output directory (not file) |
| @[OutputFiles](http://docs.google.com/javadoc/org/gradle/api/tasks/OutputFiles.html) | Map<String, File>\*\* or Iterable<File>\* | An iterable of output files (no directories). The task outputs can only be [cached](#vgdtq7) if a Map is provided. |
| @[OutputDirectories](http://docs.google.com/javadoc/org/gradle/api/tasks/OutputDirectories.html) | Map<String, File>\*\* or Iterable<File>\* | An iterable of output directories (no files). The task outputs can only be [cached](#vgdtq7) if a Map is provided. |
| @[Destroys](http://docs.google.com/javadoc/org/gradle/api/tasks/Destroys.html) | File or Iterable<File>\* | Specifies one or more files that are removed by this task. Note that a task can define either inputs/outputs or destroyables, but not both. |
| @[LocalState](http://docs.google.com/javadoc/org/gradle/api/tasks/LocalState.html) | File or Iterable<File>\* | Specifies one or more files that represent the [local state of the task](#1i7cz46). These files are removed when the task is loaded from cache. |
| @[Nested](http://docs.google.com/javadoc/org/gradle/api/tasks/Nested.html) | Any custom type | A custom type that may not implement Serializable but does have at least one field or property marked with one of the annotations in this table. It could even be another @Nested. |
| @[Console](http://docs.google.com/javadoc/org/gradle/api/tasks/Console.html) | Any type | Indicates that the property is neither an input nor an output. It simply affects the console output of the task in some way, such as increasing or decreasing the verbosity of the task. |
| @[Internal](http://docs.google.com/javadoc/org/gradle/api/tasks/Internal.html) | Any type | Indicates that the property is used internally but is neither an input nor an output. |
| @[SkipWhenEmpty](http://docs.google.com/javadoc/org/gradle/api/tasks/SkipWhenEmpty.html) | File\* | Used with @InputFiles or @InputDirectory to tell Gradle to skip the task if the corresponding files or directory are empty, along with all other input files declared with this annotation. Tasks that have been skipped due to all of their input files that were declared with this annotation being empty will result in a distinct “no source” outcome. For example, NO-SOURCE will be emitted in the console output. |
| @[Optional](http://docs.google.com/javadoc/org/gradle/api/tasks/Optional.html) | Any type | Used with any of the property type annotations listed in the [Optional](http://docs.google.com/javadoc/org/gradle/api/tasks/Optional.html) API documentation. This annotation disables validation checks on the corresponding property. See [the section on validation](#4270hrz) for more details. |
| @[PathSensitive](http://docs.google.com/javadoc/org/gradle/api/tasks/PathSensitive.html) | File\* | Used with any input file property to tell Gradle to only consider the given part of the file paths as important. For example, if a property is annotated with @PathSensitive(PathSensitivity.NAME\_ONLY), then moving the files around without changing their contents will not make the task out-of-date. |

| **Note** | | **\*** | In fact, File can be any type accepted by [Project.file(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:file(java.lang.Object)) and Iterable<File> can be any type accepted by [Project.files(java.lang.Object…​)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:files(java.lang.Object%5B%5D)), [ProjectLayout.files(java.lang.Object...)](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#files-java.lang.Object...-), or [ProjectLayout.configurableFiles(java.lang.Object...)](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#configurableFiles-java.lang.Object...-). This includes instances of Callable, such as closures, allowing for lazy evaluation of the property values. Be aware that the types FileCollection and FileTree are Iterable<File>s. | | --- | --- | | **\*\*** | Similar to the above, File can be any type accepted by [Project.file(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:file(java.lang.Object)). The Map itself can be wrapped in Callables, such as closures. | |
| --- | --- | --- | --- | --- | --- |

Annotations are inherited from all parent types including implemented interfaces. Property type annotations override any other property type annotation declared in a parent type. This way an @InputFile property can be turned into an @InputDirectory property in a child task type.

Annotations on a property declared in a type override similar annotations declared by the superclass and in any implemented interfaces. Superclass annotations take precedence over annotations declared in implemented interfaces.

The [Console](http://docs.google.com/javadoc/org/gradle/api/tasks/Console.html) and [Internal](http://docs.google.com/javadoc/org/gradle/api/tasks/Internal.html) annotations in the table are special cases as they don’t declare either task inputs or task outputs. So why use them? It’s so that you can take advantage of the [Java Gradle Plugin Development plugin](#meukdy) to help you develop and publish your own plugins. This plugin checks whether any properties of your custom task classes lack an incremental build annotation. This protects you from forgetting to add an appropriate annotation during development.

[Using the classpath annotations](#3322owd)

Besides @InputFiles, for JVM-related tasks Gradle understands the concept of classpath inputs. Both runtime and compile classpaths are treated differently when Gradle is looking for changes.

As opposed to input properties annotated with @[InputFiles](http://docs.google.com/javadoc/org/gradle/api/tasks/InputFiles.html), for classpath properties the order of the entries in the file collection matter. On the other hand, the names and paths of the directories and jar files on the classpath itself are ignored. Timestamps and the order of class files and resources inside jar files on a classpath are ignored, too, thus recreating a jar file with different file dates will not make the task out of date.

Runtime classpaths are marked with @[Classpath](http://docs.google.com/javadoc/org/gradle/api/tasks/Classpath.html), and they offer further customization via [classpath normalization](#2wfod1i).

Input properties annotated with @[CompileClasspath](http://docs.google.com/javadoc/org/gradle/api/tasks/CompileClasspath.html) are considered Java compile classpaths. Additionally to the aforementioned general classpath rules, compile classpaths ignore changes to everything but class files. Gradle uses the same class analysis described in [Java compile avoidance](#2hcarzs) to further filter changes that don’t affect the class' ABIs. This means that changes which only touch the implementation of classes do not make the task out of date.

[Nested inputs](#whl27l)

When analyzing @[Nested](http://docs.google.com/javadoc/org/gradle/api/tasks/Nested.html) task properties for declared input and output sub-properties Gradle uses the type of the actual value. Hence it can discover all sub-properties declared by a runtime sub-type.

When adding @[Nested](http://docs.google.com/javadoc/org/gradle/api/tasks/Nested.html) to a @[Provider](http://docs.google.com/javadoc/org/gradle/api/provider/Provider.html), the value of the Provider is treated as a nested input.

When adding @[Nested](http://docs.google.com/javadoc/org/gradle/api/tasks/Nested.html) to an iterable, each element is treated as a separate nested input. Each nested input in the iterable is assigned a name, which by default is the dollar sign followed by the index in the iterable, e.g. $2. If an element of the iterable implements [Named](http://docs.google.com/javadoc/org/gradle/api/Named.html), then the name is used as property name. The ordering of the elements in the iterable is crucial for for reliable up-to-date checks and caching if not all of the elements implement [Named](http://docs.google.com/javadoc/org/gradle/api/Named.html). Multiple elements which have the same name are not allowed.

When adding @[Nested](http://docs.google.com/javadoc/org/gradle/api/tasks/Nested.html) to a map, then for each value a nested input is added, using the key as name.

The type and classpath of nested inputs is tracked, too. This ensures that changes to the implementation of a nested input causes the build to be out of date. By this it is also possible to add user provided code as an input, e.g. by annotating an @[Action](http://docs.google.com/javadoc/org/gradle/api/Action.html) property with @[Nested](http://docs.google.com/javadoc/org/gradle/api/tasks/Nested.html). Note that any inputs to such actions should be tracked, either by annotated properties on the action or by manually registering them with the task.

Using nested inputs allows richer modeling and extensibility for tasks, as e.g. shown by [Test.getJvmArgumentProviders()](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html#org.gradle.api.tasks.testing.Test:jvmArgumentProviders).

This allows us to [model the JaCoCo Java agent](https://github.com/gradle/gradle/blob/7b047c7cbb4932743243a76123f5347be6d07856/subprojects/jacoco/src/main/java/org/gradle/testing/jacoco/plugins/JacocoPluginExtension.java#L138-L157), thus declaring the necessary JVM arguments and providing the inputs and outputs to Gradle:

**JacocoAgent.java**

**class** **JacocoAgent** **implements** CommandLineArgumentProvider {  
 **private** **final** JacocoTaskExtension jacoco;  
  
 **public** JacocoAgent(JacocoTaskExtension jacoco) {  
 this.jacoco = jacoco;  
 }  
  
 @Nested  
 @Optional  
 **public** JacocoTaskExtension getJacoco() {  
 **return** jacoco.isEnabled() ? jacoco : null;  
 }  
  
 @Override  
 **public** Iterable<String> asArguments() {  
 **return** jacoco.isEnabled() ? ImmutableList.of(jacoco.getAsJvmArg()) : Collections.<String>emptyList();  
 }  
}  
  
test.getJvmArgumentProviders().add(**new** JacocoAgent(extension));

For this to work, [JacocoTaskExtension](http://docs.google.com/dsl/org.gradle.testing.jacoco.plugins.JacocoTaskExtension.html) needs to have the correct input and output annotations.

The approach works for Test JVM arguments, since [Test.getJvmArgumentProviders()](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html#org.gradle.api.tasks.testing.Test:jvmArgumentProviders) is an Iterable annotated with @[Nested](http://docs.google.com/javadoc/org/gradle/api/tasks/Nested.html).

There are other task types where this kind of nested inputs are available:

* [JavaExec.getArgumentProviders()](http://docs.google.com/dsl/org.gradle.api.tasks.JavaExec.html#org.gradle.api.tasks.JavaExec:argumentProviders) - model e.g. custom tools
* [JavaExec.getJvmArgumentProviders()](http://docs.google.com/dsl/org.gradle.api.tasks.JavaExec.html#org.gradle.api.tasks.JavaExec:jvmArgumentProviders) - used for Jacoco Java agent
* [CompileOptions.getCompilerArgumentProviders()](http://docs.google.com/dsl/org.gradle.api.tasks.compile.CompileOptions.html#org.gradle.api.tasks.compile.CompileOptions:compilerArgumentProviders) - model e.g annotation processors
* [Exec.getArgumentProviders()](http://docs.google.com/dsl/org.gradle.api.tasks.Exec.html#org.gradle.api.tasks.Exec:argumentProviders) - model e.g custom tools

In the same way, this kind of modelling is available to custom tasks.

[Runtime API](#3gh8kve)

Custom task classes are an easy way to bring your own build logic into the arena of incremental build, but you don’t always have that option. That’s why Gradle also provides an alternative API that can be used with any tasks, which we look at next.

When you don’t have access to the source for a custom task class, there is no way to add any of the annotations we covered in the previous section. Fortunately, Gradle provides a runtime API for scenarios just like that. It can also be used for ad-hoc tasks, as you’ll see next.

[Using it for ad-hoc tasks](#1vmiv37)

This runtime API is provided through a couple of aptly named properties that are available on every Gradle task:

* [Task.getInputs()](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:inputs) of type [TaskInputs](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskInputs.html)
* [Task.getOutputs()](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:outputs) of type [TaskOutputs](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskOutputs.html)
* [Task.getDestroyables()](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:destroyables) of type [TaskDestroyables](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskDestroyables.html)

These objects have methods that allow you to specify files, directories and values which constitute the task’s inputs and outputs. In fact, the runtime API has almost feature parity with the annotations. All it lacks is validation of whether declared files are actually files and declared directories are directories. Nor will it create output directories if they don’t exist. But that’s it.

Let’s take the template processing example from before and see how it would look as an ad-hoc task that uses the runtime API:

[Example: Ad-hoc task](#4fm6dr0)

**build.gradle**

task processTemplatesAdHoc {  
 inputs.property("engine", TemplateEngineType.FREEMARKER)  
 inputs.files(fileTree("src/templates"))  
 inputs.property("templateData.name", "docs")  
 inputs.property("templateData.variables", [year: 2013])  
 outputs.dir("$buildDir/genOutput2")  
  
 doLast {  
 *// Process the templates here*  
 }  
}

**Output of** gradle processTemplatesAdHoc

> gradle processTemplatesAdHoc  
> Task :processTemplatesAdHoc  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

As before, there’s much to talk about. To begin with, you should really write a custom task class for this as it’s a non-trivial implementation that has several configuration options. In this case, there are no task properties to store the root source folder, the location of the output directory or any of the other settings. That’s deliberate to highlight the fact that the runtime API doesn’t require the task to have any state. In terms of incremental build, the above ad-hoc task will behave the same as the custom task class.

All the input and output definitions are done through the methods on inputs and outputs, such as property(), files(), and dir(). Gradle performs up-to-date checks on the argument values to determine whether the task needs to run again or not. Each method corresponds to one of the incremental build annotations, for example inputs.property() maps to @Input and outputs.dir() maps to @OutputDirectory. The only difference is that the file(), files(), dir() and dirs() methods don’t validate the type of file object at the given path (file or directory), unlike the annotations.

The files that a task removes can be specified through destroyables.register().

[Example: Ad-hoc task declaring a destroyable](#2urgnyt)

**build.gradle**

task removeTempDir {  
 destroyables.register("$projectDir/tmpDir")  
 doLast {  
 delete("$projectDir/tmpDir")  
 }  
}

One notable difference between the runtime API and the annotations is the lack of a method that corresponds directly to @Nested. That’s why the example uses two property() declarations for the template data, one for each TemplateData property. You should utilize the same technique when using the runtime API with nested values. Any given task can either declare destroyables or inputs/outputs, but cannot declare both.

[Using it for custom task types](#19wqy6m)

Another type of example involves adding input and output definitions to instances of a custom task class that lacks the requisite annotations. For example, imagine that the ProcessTemplates task is provided by a plugin and that it’s missing the incremental build annotations. In order to make up for that deficiency, you can use the runtime API:

[Example: Using runtime API with custom task type](#3tweguf)

**build.gradle**

task processTemplatesRuntime(type: ProcessTemplatesNoAnnotations) {  
 templateEngine = TemplateEngineType.FREEMARKER  
 sourceFiles = fileTree("src/templates")  
 templateData = **new** TemplateData("test", [year: 2014])  
 outputDir = file("$buildDir/genOutput3")  
  
 inputs.property("engine",templateEngine)  
 inputs.files(sourceFiles)  
 inputs.property("templateData.name", templateData.name)  
 inputs.property("templateData.variables", templateData.variables)  
 outputs.dir(outputDir)  
}

**Output of gradle processTemplatesRuntime**

> gradle processTemplatesRuntime  
> Task :processTemplatesRuntime  
  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

**Output of gradle processTemplatesRuntime (run again)**

> gradle processTemplatesRuntime  
> Task :processTemplatesRuntime UP-TO-DATE  
  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 up-to-date

As you can see, we can both configure the tasks properties and use those properties as arguments to the incremental build runtime API. Using the runtime API like this is a little like using doLast() and doFirst() to attach extra actions to a task, except in this case we’re attaching information about inputs and outputs. Note that if the task type is already using the incremental build annotations, the runtime API will add inputs and outputs rather than replace them.

[Fine-grained configuration](#291or28)

The runtime API methods only allow you to declare your inputs and outputs in themselves. However, the file-oriented ones return a builder - of type [TaskInputFilePropertyBuilder](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskInputFilePropertyBuilder.html) - that lets you provide additional information about those inputs and outputs.

You can learn about all the options provided by the builder in its API documentation, but we’ll show you a simple example here to give you an idea of what you can do.

Let’s say we don’t want to run the processTemplates task if there are no source files, regardless of whether it’s a clean build or not. After all, if there are no source files, there’s nothing for the task to do. The builder allows us to configure this like so:

[Example: Using skipWhenEmpty() via the runtime API](#o6z1a1)

**build.gradle**

task processTemplatesRuntimeConf(type: ProcessTemplatesNoAnnotations) {  
 *// ...*  
 sourceFiles = fileTree("src/templates") {  
 include "\*\*/\*.fm"  
 }  
  
 inputs.files(sourceFiles).skipWhenEmpty()  
 *// ...*  
}

**Output of** gradle clean processTemplatesRuntimeConf

> gradle clean processTemplatesRuntimeConf  
> Task :processTemplatesRuntimeConf NO-SOURCE  
  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 up-to-date

The TaskInputs.files() method returns a builder that has a skipWhenEmpty() method. Invoking this method is equivalent to annotating to the property with [@SkipWhenEmpty](#48j4tk5).

Prior to Gradle 3.0, you had to use the TaskInputs.source() and TaskInputs.sourceDir() methods to get the same behavior as with skipWhenEmpty(). These methods are now deprecated and should not be used with Gradle 3.0 and above.

Now that you have seen both the annotations and the runtime API, you may be wondering which API you should be using. Our recommendation is to use the annotations wherever possible, and it’s sometimes worth creating a custom task class just so that you can make use of them. The runtime API is more for situations in which you can’t use the annotations.

[Important beneficial side effects](#386mjxu)

Once you declare a task’s formal inputs and outputs, Gradle can then infer things about those properties. For example, if an input of one task is set to the output of another, that means the first task depends on the second, right? Gradle knows this and can act upon it.

We’ll look at this feature next and also some other features that come from Gradle knowing things about inputs and outputs.

[Inferred task dependencies](#1nbwu5n)

Consider an archive task that packages the output of the processTemplates task. A build author will see that the archive task obviously requires processTemplates to run first and so may add an explicit dependsOn. However, if you define the archive task like so:

[Example: Inferred task dependency via task outputs](#47bkctg)

**build.gradle**

task packageFiles(type: Zip) {  
 from processTemplates.outputs  
}

**Output of** gradle clean packageFiles

> gradle clean packageFiles  
> Task :processTemplates  
> Task :packageFiles  
  
  
BUILD SUCCESSFUL in 0s  
3 actionable tasks: 2 executed, 1 up-to-date

Gradle will automatically make packageFiles depend on processTemplates. It can do this because it’s aware that one of the inputs of packageFiles requires the output of the processTemplates task. We call this an inferred task dependency.

The above example can also be written as

[Example: Inferred task dependency via a task argument](#2mgun19)

**build.gradle**

task packageFiles2(type: Zip) {  
 from processTemplates  
}

**Output of** gradle clean packageFiles2

> gradle clean packageFiles2  
> Task :processTemplates  
> Task :packageFiles2  
  
  
BUILD SUCCESSFUL in 0s  
3 actionable tasks: 2 executed, 1 up-to-date

This is because the from() method can accept a task object as an argument. Behind the scenes, from() uses the project.files() method to wrap the argument, which in turn exposes the task’s formal outputs as a file collection. In other words, it’s a special case!

[Input and output validation](#4270hrz)

The incremental build annotations provide enough information for Gradle to perform some basic validation on the annotated properties. In particular, it does the following for each property before the task executes:

* @InputFile - verifies that the property has a value and that the path corresponds to a file (not a directory) that exists.
* @InputDirectory - same as for @InputFile, except the path must correspond to a directory.
* @OutputDirectory - verifies that the path doesn’t match a file and also creates the directory if it doesn’t already exist.

Such validation improves the robustness of the build, allowing you to identify issues related to inputs and outputs quickly.

You will occasionally want to disable some of this validation, specifically when an input file may validly not exist. That’s why Gradle provides the @Optional annotation: you use it to tell Gradle that a particular input is optional and therefore the build should not fail if the corresponding file or directory doesn’t exist.

[Continuous build](#11m4x92)

Another benefit of defining task inputs and outputs is continuous build. Since Gradle knows what files a task depends on, it can automatically run a task again if any of its inputs change. By activating continuous build when you run Gradle - through the --continuous or -t options - you will put Gradle into a state in which it continually checks for changes and executes the requested tasks when it encounters such changes.

You can find out more about this feature in [Continuous build](#2olpkfy).

[Task parallelism](#3llsfwv)

One last benefit of defining task inputs and outputs is that Gradle can use this information to make decisions about how to run tasks when the "--parallel" option is used. For instance, Gradle will inspect the outputs of tasks when selecting the next task to run and will avoid concurrent execution of tasks that write to the same output directory. Similarly, Gradle will use the information about what files a task destroys (e.g. specified by the Destroys annotation) and avoid running a task that removes a set of files while another task is running that consumes or creates those same files (and vice versa). It can also determine that a task that creates a set of files has already run and that a task that consumes those files has yet to run and will avoid running a task that removes those files in between. By providing task input and output information in this way, Gradle can infer creation/consumption/destruction relationships between tasks and can ensure that task execution does not violate those relationships.

[How does it work?](#43v86uo)

Before a task is executed for the first time, Gradle takes a snapshot of the inputs. This snapshot contains the paths of input files and a hash of the contents of each file. Gradle then executes the task. If the task completes successfully, Gradle takes a snapshot of the outputs. This snapshot contains the set of output files and a hash of the contents of each file. Gradle persists both snapshots for the next time the task is executed.

Each time after that, before the task is executed, Gradle takes a new snapshot of the inputs and outputs. If the new snapshots are the same as the previous snapshots, Gradle assumes that the outputs are up to date and skips the task. If they are not the same, Gradle executes the task. Gradle persists both snapshots for the next time the task is executed.

Gradle also considers the *code* of the task as part of the inputs to the task. When a task, its actions, or its dependencies change between executions, Gradle considers the task as out-of-date.

Gradle understands if a file property (e.g. one holding a Java classpath) is order-sensitive. When comparing the snapshot of such a property, even a change in the order of the files will result in the task becoming out-of-date.

Note that if a task has an output directory specified, any files added to that directory since the last time it was executed are ignored and will NOT cause the task to be out of date. This is so unrelated tasks may share an output directory without interfering with each other. If this is not the behaviour you want for some reason, consider using [TaskOutputs.upToDateWhen(groovy.lang.Closure)](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskOutputs.html#upToDateWhen-groovy.lang.Closure-)

The inputs for the task are also used to calculate the [build cache](#2bn6wsx) key used to load task outputs when enabled. For more details see [Task output caching](#vgdtq7).

[Advanced techniques](#20r2q4o)

Everything you’ve seen so far in this section will cover most of the use cases you’ll encounter, but there are some scenarios that need special treatment. We’ll present a few of those next with the appropriate solutions.

[Adding your own cached input/output methods](#4kqq8sh)

Have you ever wondered how the from() method of the Copy task works? It’s not annotated with @InputFiles and yet any files passed to it are treated as formal inputs of the task. What’s happening?

The implementation is quite simple and you can use the same technique for your own tasks to improve their APIs. Write your methods so that they add files directly to the appropriate annotated property. As an example, here’s how to add a sources() method to the custom ProcessTemplates class we introduced earlier:

[Example: Declaring a method to add task inputs](#2zw0j0a)

**ProcessTemplates.java**

**public** **class** **ProcessTemplates** **extends** DefaultTask {  
 *// ...*  
 **private** FileCollection sourceFiles = getProject().getLayout().files();  
  
 @SkipWhenEmpty  
 @InputFiles  
 @PathSensitive(PathSensitivity.NONE)  
 **public** FileCollection getSourceFiles() {  
 **return** this.sourceFiles;  
 }  
  
 **public** **void** sources(FileCollection sourceFiles) {  
 this.sourceFiles = this.sourceFiles.plus(sourceFiles);  
 }  
  
 *// ...*  
}

**build.gradle**

task processTemplates(type: ProcessTemplates) {  
 templateEngine = TemplateEngineType.FREEMARKER  
 templateData = **new** TemplateData("test", [year: 2012])  
 outputDir = file("$buildDir/genOutput")  
  
 sources fileTree("src/templates")  
}

**Output of gradle processTemplates**

> gradle processTemplates  
> Task :processTemplates  
  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

In other words, as long as you add values and files to formal task inputs and outputs during the configuration phase, they will be treated as such regardless from where in the build you add them.

If we want to support tasks as arguments as well and treat their outputs as the inputs, we can use the project.layout.files() method like so:

[Example: Declaring a method to add a task as an input](#1f1at83)

**ProcessTemplates.java**

*// ...*  
 **public** **void** sources(Task inputTask) {  
 this.sourceFiles = this.sourceFiles.plus(getProject().getLayout().files(inputTask));  
 }  
 *// ...*

**build.gradle**

task copyTemplates(type: Copy) {  
 into "$buildDir/tmp"  
 from "src/templates"  
}  
  
task processTemplates2(type: ProcessTemplates) {  
 *// ...*  
 sources copyTemplates  
}

**Output of gradle processTemplates2**

> gradle processTemplates2  
> Task :copyTemplates  
> Task :processTemplates2  
  
  
BUILD SUCCESSFUL in 0s  
2 actionable tasks: 2 executed

This technique can make your custom task easier to use and result in cleaner build files. As an added benefit, our use of getProject().getLayout().files() means that our custom method can set up an inferred task dependency.

One last thing to note: if you are developing a task that takes collections of source files as inputs, like this example, consider using the built-in [SourceTask](http://docs.google.com/dsl/org.gradle.api.tasks.SourceTask.html). It will save you having to implement some of the plumbing that we put into ProcessTemplates.

[Linking an @OutputDirectory to an @InputFiles](#3z0ybvw)

When you want to link the output of one task to the input of another, the types often match and a simple property assignment will provide that link. For example, a File output property can be assigned to a File input.

Unfortunately, this approach breaks down when you want the files in a task’s @OutputDirectory (of type File) to become the source for another task’s @InputFiles property (of type FileCollection). Since the two have different types, property assignment won’t work.

As an example, imagine you want to use the output of a Java compilation task - via the destinationDir property - as the input of a custom task that instruments a set of files containing Java bytecode. This custom task, which we’ll call Instrument, has a classFiles property annotated with @InputFiles. You might initially try to configure the task like so:

[Example: Failed attempt at setting up an inferred task dependency](#2e68m3p)

**build.gradle**

apply plugin: "java"  
  
task badInstrumentClasses(type: Instrument) {  
 classFiles = fileTree(compileJava.destinationDir)  
 destinationDir = file("$buildDir/instrumented")  
}

**Output of** gradle clean badInstrumentClasses

> gradle clean badInstrumentClasses  
> Task :clean UP-TO-DATE  
> Task :badInstrumentClasses NO-SOURCE  
  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 up-to-date

There’s nothing obviously wrong with this code, but you can see from the console output that the compilation task is missing. In this case you would need to add an explicit task dependency between instrumentClasses and compileJava via dependsOn. The use of fileTree() means that Gradle can’t infer the task dependency itself.

One solution is to use the TaskOutputs.files property, as demonstrated by the following example:

[Example: Setting up an inferred task dependency between output dir and input files](#tbiwbi)

**build.gradle**

task instrumentClasses(type: Instrument) {  
 classFiles = compileJava.outputs.files  
 destinationDir = file("$buildDir/instrumented")  
}

**Output of** gradle clean instrumentClasses

> gradle clean instrumentClasses  
> Task :clean UP-TO-DATE  
> Task :compileJava  
> Task :instrumentClasses  
  
  
BUILD SUCCESSFUL in 0s  
3 actionable tasks: 2 executed, 1 up-to-date

Alternatively, you can get Gradle to access the appropriate property itself by using one of project.files(), project.layout.files() or project.layout.configurableFiles() in place of project.fileTree():

[Example: Setting up an inferred task dependency with filesFor()](#3db6ezb)

**build.gradle**

task instrumentClasses2(type: Instrument) {  
 classFiles = layout.files(compileJava)  
 destinationDir = file("$buildDir/instrumented")  
}

**Output of** gradle clean instrumentClasses2

> gradle clean instrumentClasses2  
> Task :clean UP-TO-DATE  
> Task :compileJava  
> Task :instrumentClasses2  
  
  
BUILD SUCCESSFUL in 0s  
3 actionable tasks: 2 executed, 1 up-to-date

Remember that files(), layout.files() and layout.configurableFiles() can take tasks as arguments, whereas fileTree() cannot.

The downside of this approach is that all file outputs of the source task become the input files of the target - instrumentClasses in this case. That’s fine as long as the source task only has a single file-based output, like the JavaCompile task. But if you have to link just one output property among several, then you need to explicitly tell Gradle which task generates the input files using the builtBy method:

[Example: Setting up an inferred task dependency with builtBy()](#1sggp74)

**build.gradle**

task instrumentClassesBuiltBy(type: Instrument) {  
 classFiles = fileTree(compileJava.destinationDir) {  
 builtBy compileJava  
 }  
 destinationDir = file("$buildDir/instrumented")  
}

**Output of** gradle clean instrumentClassesBuiltBy

> gradle clean instrumentClassesBuiltBy  
> Task :clean UP-TO-DATE  
> Task :compileJava  
> Task :instrumentClassesBuiltBy  
  
  
BUILD SUCCESSFUL in 0s  
3 actionable tasks: 2 executed, 1 up-to-date

You can of course just add an explicit task dependency via dependsOn, but the above approach provides more semantic meaning, explaining why compileJava has to run beforehand.

[Providing custom up-to-date logic](#4cg47ux)

Gradle automatically handles up-to-date checks for output files and directories, but what if the task output is something else entirely? Perhaps it’s an update to a web service or a database table. Gradle has no way of knowing how to check whether the task is up to date in such cases.

That’s where the upToDateWhen() method on TaskOutputs comes in. This takes a predicate function that is used to determine whether a task is up to date or not. One use case is to disable up-to-date checks completely for a task, like so:

[Example: Ignoring up-to-date checks](#2rlei2q)

**build.gradle**

task alwaysInstrumentClasses(type: Instrument) {  
 classFiles = layout.files(compileJava)  
 destinationDir = file("$buildDir/instrumented")  
 outputs.upToDateWhen { false }  
}

**Output of gradle clean alwaysInstrumentClasses**

> gradle clean alwaysInstrumentClasses  
> Task :compileJava  
> Task :alwaysInstrumentClasses  
  
  
BUILD SUCCESSFUL in 0s  
3 actionable tasks: 2 executed, 1 up-to-date

**Output of gradle alwaysInstrumentClasses**

> gradle alwaysInstrumentClasses  
> Task :compileJava UP-TO-DATE  
> Task :alwaysInstrumentClasses  
  
  
BUILD SUCCESSFUL in 0s  
2 actionable tasks: 1 executed, 1 up-to-date

The { false } closure ensures that copyResources will always perform the copy, irrespective of whether there is no change in the inputs or outputs.

You can of course put more complex logic into the closure. You could check whether a particular record in a database table exists or has changed for example. Just be aware that up-to-date checks should *save* you time. Don’t add checks that cost as much or more time than the standard execution of the task. In fact, if a task ends up running frequently anyway, because it’s rarely up to date, then it may not be worth having an up-to-date check at all. Remember that your checks will always run if the task is in the execution task graph.

One common mistake is to use upToDateWhen() instead of Task.onlyIf(). If you want to skip a task on the basis of some condition unrelated to the task inputs and outputs, then you should use onlyIf(). For example, in cases where you want to skip a task when a particular property is set or not set.

[Configure input normalization](#2wfod1i)

For up to date checks and the [build cache](#2bn6wsx) Gradle needs to determine if two task input properties have the same value. In order to do so, Gradle first normalizes both inputs and then compares the result. For example, for a compile classpath, Gradle extracts the ABI signature from the classes on the classpath and then compares signatures between the last Gradle run and the current Gradle run as described in [Java compile avoidance](#2hcarzs).

It is possible to customize Gradle’s built-in strategy for runtime classpath normalization. All inputs annotated with @[Classpath](http://docs.google.com/javadoc/org/gradle/api/tasks/Classpath.html) are considered to be runtime classpaths.

Let’s say you want to add a file build-info.properties to all your produced jar files which contains 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 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 and changes on every build invocation. Therefore, the test would be never up-to-date or pulled from the build cache. In order to benefit from incremental builds again, you are able tell Gradle to ignore this file on the runtime classpath at the project level by using [Project.normalization(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:normalization(org.gradle.api.Action)):

[Example: Runtime classpath normalization](#16qosaj)

**build.gradle**

normalization {  
 runtimeClasspath {  
 ignore 'build-info.properties'  
 }  
}

The effect of this configuration would be that changes to build-info.properties would be ignored for up-to-date checks and [build cache](#2bn6wsx) key calculations. Note that 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 is still the same as before.

[Stale task outputs](#3qqcayc)

When the Gradle version changes, Gradle detects that outputs from tasks that ran with older versions of Gradle need to be removed to ensure that the newest version of the tasks are starting from a known clean state.

| **Note** | Automatic clean-up of stale output directories has only been implemented for the output of source sets (Java/Groovy/Scala compilation). |
| --- | --- |

[Task rules](#25vml65)

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:

[Example: Task rule](#l0wvdy)

**build.gradle**

tasks.addRule("Pattern: ping<ID>") { String taskName ->  
 **if** (taskName.startsWith("ping")) {  
 task(taskName) {  
 doLast {  
 println "Pinging: " + (taskName - 'ping')  
 }  
 }  
 }  
}

**Output of** gradle -q pingServer1

> gradle -q pingServer1  
Pinging: Server1

The String parameter is used as a description for the rule, which is shown with gradle tasks.

Rules are not only used when calling tasks from the command line. You can also create dependsOn relations on rule based tasks:

[Example: Dependency on rule based tasks](#350ke1r)

**build.gradle**

tasks.addRule("Pattern: ping<ID>") { String taskName ->  
 **if** (taskName.startsWith("ping")) {  
 task(taskName) {  
 doLast {  
 println "Pinging: " + (taskName - 'ping')  
 }  
 }  
 }  
}  
  
task groupPing {  
 dependsOn pingServer1, pingServer2  
}

**Output of** gradle -q groupPing

> gradle -q groupPing  
Pinging: Server1  
Pinging: Server2

If you run “gradle -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.

[Finalizer tasks](#1k5uo9k)

| **Note** | Finalizers tasks are an *incubating* feature (see [more about the Gradle feature lifecycle](#2pcmsun)). |
| --- | --- |

Finalizer tasks are automatically added to the task graph when the finalized task is scheduled to run.

[Example: Adding a task finalizer](#445i6xd)

**build.gradle**

task taskX {  
 doLast {  
 println 'taskX'  
 }  
}  
task taskY {  
 doLast {  
 println 'taskY'  
 }  
}  
  
taskX.finalizedBy taskY

**Output of** gradle -q taskX

> gradle -q taskX  
taskX  
taskY

Finalizer tasks will be executed even if the finalized task fails.

[Example: Task finalizer for a failing task](#2jash56)

**build.gradle**

task taskX {  
 doLast {  
 println 'taskX'  
 **throw** **new** RuntimeException()  
 }  
}  
task taskY {  
 doLast {  
 println 'taskY'  
 }  
}  
  
taskX.finalizedBy taskY

**Output of** gradle -q taskX

> 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

On the other hand, finalizer tasks are not executed if the finalized task didn’t do any work, for example if it is considered up to date or if a dependent task fails.

Finalizer tasks are useful in situations where the build creates a resource that has to be cleaned up regardless of the build failing or succeeding. An example of such a resource is a web container that is started before an integration test task and which should be always shut down, even if some of the tests fail.

To specify a finalizer task you use the [Task.finalizedBy(java.lang.Object…​)](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:finalizedBy(java.lang.Object%5B%5D)) method. This method accepts a task instance, a task name, or any other input accepted by [Task.dependsOn(java.lang.Object…​)](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:dependsOn(java.lang.Object%5B%5D)).

[Lifecycle tasks](#34qadz2)

Lifecycle tasks are tasks that do not do work themselves. They typically do not have any task actions. Lifecycle tasks can represent several concepts:

* a work-flow step (e.g., run all checks with check)
* a buildable thing (e.g., create a debug 32-bit executable for native components with debug32MainExecutable)
* a convenience task to execute many of the same logical tasks (e.g., run all compilation tasks with compileAll)

Many Gradle plug-ins define their own lifecycle tasks to make it convenient to do specific things. When developing your own plugins, you should consider using your own lifecycle tasks or hooking into some of the tasks already provided by Gradle. See the Java plugin [tasks](#yg2rcz) for an example.

Unless a lifecycle task has actions, its outcome is determined by its dependencies. If any of the task’s dependencies are executed, the lifecycle task will be considered executed. If all of the task’s dependencies are up-to-date, skipped or from cache, the lifecycle task will be considered up-to-date.

[Summary](#3ifqa0s)

If you are coming from Ant, an enhanced Gradle task like *Copy* seems like a cross between an Ant target and an Ant task. Although Ant’s tasks and targets are really different entities, Gradle combines these notions into a single entity. Simple Gradle tasks are like Ant’s targets, but enhanced Gradle tasks also include aspects of Ant tasks. All of Gradle’s tasks share a common API and you can create dependencies between them. These tasks are much easier to configure than an Ant task. They make full use of the type system, and are more expressive and easier to maintain.

[Logging](#3o7alnk)

The log is the main 'UI' of a build tool. If it is too verbose, real warnings and problems are easily hidden by this. On the other hand you need relevant information for figuring out if things have gone wrong. Gradle defines 6 log levels, as shown in [Log levels](#1xl0k8l). There are two Gradle-specific log levels, in addition to the ones you might normally see. Those levels are *QUIET* and *LIFECYCLE*. The latter is the default, and is used to report build progress.

[Log levels](#1xl0k8l)

| **ERROR** | Error messages |
| --- | --- |
| **QUIET** | Important information messages |
| **WARNING** | Warning messages |
| **LIFECYCLE** | Progress information messages |
| **INFO** | Information messages |
| **DEBUG** | Debug messages |

| **Note** | The rich components of the console (build status and work in progress area) are displayed regardless of the log level used. Before Gradle 4.0 those rich components were only displayed at log level LIFECYCLE or below. |
| --- | --- |

[Choosing a log level](#1ulbmlt)

You can use the command line switches shown in [Log level command-line options](#4hko2we) to choose different log levels. You can also configure the log level using gradle.properties, see [Gradle properties](#1idq7dh). In [Stacktrace command-line options](#2wpyd47) you find the command line switches which affect stacktrace logging.

Table 2. Log level command-line options

| **Option** | **Outputs Log Levels** |
| --- | --- |
| no logging options | LIFECYCLE and higher |
| -q or --quiet | QUIET and higher |
| -w or --warn | WARN and higher |
| -i or --info | INFO and higher |
| -d or --debug | DEBUG and higher (that is, all log messages) |

[Stacktrace command-line options](#2wpyd47)

-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 for 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.

[Writing your own log messages](#1bv8nc0)

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.

[Example: Using stdout to write log messages](#3vuw5zt)

**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](http://docs.google.com/javadoc/org/gradle/api/logging/Logger.html). This interface extends the SLF4J Logger interface and adds a few Gradle specific methods to it. Below is an example of how this is used in the build script:

[Example: Writing your own log messages](#2b06g7m)

**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.')

Use the [typical SLF4J pattern](https://www.slf4j.org/manual.html#typical_usage) to replace a placeholder with an actual value as part of the log message.

[Example: Writing a log message with placeholder](#q5gqff)

**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). Simply use an SLF4J logger. You can use this logger the same way as you use the provided logger in the build script.

[Example: Using SLF4J to write log messages](#3a54938)

**build.gradle**

**import** org.slf4j.Logger  
**import** org.slf4j.LoggerFactory  
  
Logger slf4jLogger = LoggerFactory.getLogger('some-logger')  
slf4jLogger.info('An info log message logged using SLF4j')

[Logging from external tools and libraries](#1yib0wl)

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 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.

There are many tools out there which still use 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](http://docs.google.com/javadoc/org/gradle/api/logging/LoggingManager.html), which allows you to change the log levels that standard out or error are redirected to when your build script is evaluated.

[Example: Configuring standard output capture](#1paejb1)

**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, tasks also provide a [LoggingManager](http://docs.google.com/javadoc/org/gradle/api/logging/LoggingManager.html).

[Example: Configuring standard output capture for a task](#49a21yu)

**build.gradle**

task logInfo {  
 logging.captureStandardOutput LogLevel.INFO  
 doFirst {  
 println 'A task message which is logged at INFO level'  
 }  
}

Gradle also provides integration with the Java Util Logging, Jakarta Commons Logging and Log4j logging toolkits. Any log messages which your build classes write using these logging toolkits will be redirected to Gradle’s logging system.

[Changing what Gradle logs](#3jd0qos)

You can replace much of Gradle’s logging UI with your own. You might do this, for example, if you want to customize the UI in some way - to log more or less information, or to change the formatting. You replace the logging using the [Gradle.useLogger(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.invocation.Gradle.html#org.gradle.api.invocation.Gradle:useLogger(java.lang.Object)) method. This is accessible from a build script, or an init script, or via the embedding API. Note that this completely disables Gradle’s default output. Below is an example init script which changes how task execution and build completion is logged.

[Example: Customizing what Gradle logs](#2ofcc6n)

**init.gradle**

useLogger(**new** CustomEventLogger())  
  
**class** **CustomEventLogger** **extends** BuildAdapter **implements** TaskExecutionListener {  
  
 **public** **void** beforeExecute(Task task) {  
 println "[$task.name]"  
 }  
  
 **public** **void** afterExecute(Task task, TaskState state) {  
 println()  
 }  
  
 **public** **void** buildFinished(BuildResult result) {  
 println 'build completed'  
 **if** (result.failure != null) {  
 result.failure.printStackTrace()  
 }  
 }  
}

**Output of** gradle -I init.gradle build

> gradle -I 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 that 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](#1qym8dq).

* [BuildListener](http://docs.google.com/javadoc/org/gradle/BuildListener.html)
* [ProjectEvaluationListener](http://docs.google.com/javadoc/org/gradle/api/ProjectEvaluationListener.html)
* [TaskExecutionGraphListener](http://docs.google.com/javadoc/org/gradle/api/execution/TaskExecutionGraphListener.html)
* [TaskExecutionListener](http://docs.google.com/javadoc/org/gradle/api/execution/TaskExecutionListener.html)
* [TaskActionListener](http://docs.google.com/javadoc/org/gradle/api/execution/TaskActionListener.html)

[Standard Gradle plugins](#23ckvvd)

There are a number of plugins included in the Gradle distribution. These are listed below.

[Language plugins](#13kmmeg)

These plugins add support for various languages which can be compiled for and executed in the JVM.

| **Plugin Id** | **Automatically applies** | **Description** |
| --- | --- | --- |
| [java](#2w5ecyt) | java-base | Adds Java compilation, testing and bundling capabilities to a project. It serves as the basis for many of the other Gradle plugins. See also [this tutorial on Java projects](#3hv69ve). |
| [groovy](#43ky6rz) | java, groovy-base | Adds support for building Groovy projects. See also [this tutorial for Groovy projects](#1jlao46). |
| [scala](#3ep43zb) | java, scala-base | Adds support for building Scala projects. |
| [antlr](#39kk8xu) | java | Adds support for generating parsers using [Antlr](http://www.antlr.org/). |

[Incubating language plugins](#3nka529)

These plugins add support for various languages:

| **Plugin Id** | **Automatically applies** | **Description** |
| --- | --- | --- |
| [assembler](#2r0uhxc) | - | Adds native assembly language capabilities to a project. |
| [c](#2r0uhxc) | - | Adds C source compilation capabilities to a project. |
| [cpp](#2r0uhxc) | - | Adds C++ source compilation capabilities to a project. |
| [objective-c](#2r0uhxc) | - | Adds Objective-C source compilation capabilities to a project. |
| [objective-cpp](#2r0uhxc) | - | Adds Objective-C++ source compilation capabilities to a project. |
| [windows-resources](#2r0uhxc) | - | Adds support for including Windows resources in native binaries. |

[Integration plugins](#22pkfa2)

These plugins provide some integration with various runtime technologies.

| **Plugin Id** | **Automatically applies** | **Works with** | **Description** |
| --- | --- | --- | --- |
| [application](#1opuj5n) | java, distribution | - | Adds tasks for running and bundling a Java project as a command-line application. |
| [ear](#1gf8i83) | - | java | Adds support for building J2EE applications. |
| [ivy-publish](#3ygebqi) | - | application, distribution, java, war | Provides a new DSL to support publishing artifacts to Ivy repositories, which improves on the existing DSL. |
| [maven-publish](#1egqt2p) | - | application, distribution, java, war | Provides a new DSL to support publishing artifacts to Maven repositories, which improves on the existing DSL. |
| [maven](#sqyw64) | - | java, war | Adds support for publishing artifacts to Maven repositories using the *original* publishing mechanism available in Gradle 1.0. See also [Legacy Publishing](#2dlolyb). |
| [osgi](#2250f4o) | java-base | java | Adds support for building OSGi bundles. |
| [war](#2fk6b3p) | java | - | Adds support for assembling web application WAR files. |

[Incubating integration plugins](#huuphv)

These plugins provide some integration with various runtime technologies.

| **Plugin Id** | **Automatically applies** | **Description** |
| --- | --- | --- |
| [distribution](#1rvwp1q) | - | Adds support for building ZIP and TAR distributions. |
| [java-library-distribution](#3vac5uf) | java, distribution | Adds support for building ZIP and TAR distributions for a Java library. |

[Software development plugins](#31ui85o)

These plugins provide help with your software development process.

| **Plugin Id** | **Automatically applies** | **Works with** | **Description** |
| --- | --- | --- | --- |
| [announce](http://docs.google.com/announce_plugin.html#announce_plugin) | - | - | Publish messages to your favourite platforms, such as Twitter or Growl. |
| [build-announcements](http://docs.google.com/build_announcements_plugin.html#build_announcements_plugin) | announce | - | Sends local announcements to your desktop about interesting events in the build lifecycle. |
| [checkstyle](#48pi1tg) | java-base | - | Performs quality checks on your project’s Java source files using [Checkstyle](http://checkstyle.sourceforge.net/index.html) and generates reports from these checks. |
| [codenarc](#2iq8gzs) | groovy-base | - | Performs quality checks on your project’s Groovy source files using [CodeNarc](http://codenarc.sourceforge.net/index.html) and generates reports from these checks. |
| [eclipse](#4du1wux) | - | java,groovy, scala | Generates files that are used by [Eclipse IDE](http://eclipse.org), thus making it possible to import the project into Eclipse. See also [this tutorial for Java projects](#3hv69ve). |
| [eclipse-wtp](#4du1wux) | - | ear, war | Does the same as the eclipse plugin plus generates eclipse WTP (Web Tools Platform) configuration files. After importing to eclipse your war/ear projects should be configured to work with WTP. See also [this tutorial for Java projects](#3hv69ve). |
| [findbugs](#2nusc19) | java-base | - | Performs quality checks on your project’s Java source files using [FindBugs](http://findbugs.sourceforge.net) and generates reports from these checks. |
| [idea](#2szc72q) | - | java | Generates files that are used by [Intellij IDEA IDE](http://www.jetbrains.com/idea/index.html), thus making it possible to import the project into IDEA. |
| [jdepend](#3mzq4wv) | java-base | - | Performs quality checks on your project’s source files using [JDepend](http://clarkware.com/software/JDepend.html) and generates reports from these checks. |
| [pmd](#haapch) | java-base | - | Performs quality checks on your project’s Java source files using [PMD](http://pmd.sourceforge.net) and generates reports from these checks. |
| [project-report](http://docs.google.com/project_report_plugin.html#project_report_plugin) | reporting-base | - | Generates reports containing useful information about your Gradle build. |
| [signing](#3cqmetx) | base | - | Adds the ability to digitally sign built files and artifacts. |

[Incubating software development plugins](#1gzsidh)

These plugins provide help with your software development process.

Table 3. Software development plugins

| **Plugin Id** | **Automatically applies** | **Works with** | **Description** |
| --- | --- | --- | --- |
| [build-dashboard](http://docs.google.com/build_dashboard_plugin.html#build_dashboard_plugin) | reporting-base | - | Generates build dashboard report. |
| [cunit](#2r0uhxc) | - | - | Adds support for running [CUnit](http://cunit.sourceforge.net) tests. |
| [jacoco](#1302m92) | reporting-base | java | Provides integration with the [JaCoCo](http://www.eclemma.org/jacoco/) code coverage library for Java. |
| [visual-studio](#2r0uhxc) | - | native language plugins | Adds integration with Visual Studio. |
| [java-gradle-plugin](#meukdy) | java |  | Assists with development of Gradle plugins by providing standard plugin build configuration and validation. |

[Base plugins](#40zg11a)

These plugins form the basic building blocks which the other plugins are assembled from. They are available for you to use in your build files, and are listed here for completeness. However, be aware that they are not yet considered part of Gradle’s public API. As such, these plugins are not documented in the user guide. You might refer to their API documentation to learn more about them.

| **Plugin Id** | **Description** |
| --- | --- |
| base | Adds the standard lifecycle tasks and configures reasonable defaults for the archive tasks. See [Base Plugin](#vx1227). |
| java-base | Adds the source sets concept to the project. Does not add any particular source sets. |
| groovy-base | Adds the Groovy source sets concept to the project. |
| scala-base | Adds the Scala source sets concept to the project. |
| reporting-base | Adds some shared convention properties to the project, relating to report generation. |

[Third party plugins](#2g4qb93)

You can find a list of external plugins at the [Gradle Plugins site](http://plugins.gradle.org/).

[Testing Build Logic with TestKit](#ihv636)

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.

[Usage](#va0lgw)

To use the TestKit, include the following in your plugin’s build:

[Example: Declaring the TestKit dependency](#3f9o44p)

**build.gradle**

dependencies {  
 testCompile gradleTestKit()  
}

The gradleTestKit() encompasses the classes of the TestKit, as well as the [Gradle Tooling API client](#184mhaj). It does not include a version of [JUnit](http://junit.org), [TestNG](http://testng.org), or any other test execution framework. Such a dependency must be explicitly declared.

[Example: Declaring the JUnit dependency](#1ueyeci)

**build.gradle**

dependencies {  
 testCompile 'junit:junit:4.12'  
}

[Functional testing with the Gradle runner](#4eelx0b)

The [GradleRunner](http://docs.google.com/javadoc/org/gradle/testkit/runner/GradleRunner.html) 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()](http://docs.google.com/javadoc/org/gradle/testkit/runner/GradleRunner.html#build--) or [GradleRunner.buildAndFail()](http://docs.google.com/javadoc/org/gradle/testkit/runner/GradleRunner.html#buildAndFail--) methods depending on the anticipated outcome.

The following demonstrates the usage of Gradle runner in a Java JUnit test:

[Example: Using GradleRunner with JUnit](#2tjw784)

**BuildLogicFunctionalTest.java**

**import** org.gradle.testkit.runner.BuildResult;  
**import** org.gradle.testkit.runner.GradleRunner;  
**import** org.junit.Before;  
**import** org.junit.Rule;  
**import** org.junit.Test;  
**import** org.junit.rules.TemporaryFolder;  
  
**import** java.io.BufferedWriter;  
**import** java.io.File;  
**import** java.io.FileWriter;  
**import** java.io.IOException;  
**import** java.util.Collections;  
  
**import** static org.junit.Assert.assertEquals;  
**import** static org.junit.Assert.assertTrue;  
  
**import** static org.gradle.testkit.runner.TaskOutcome.\*;  
  
**public** **class** **BuildLogicFunctionalTest** {  
 @Rule **public** **final** TemporaryFolder testProjectDir = **new** TemporaryFolder();  
 **private** File buildFile;  
  
 @Before  
 **public** **void** setup() **throws** IOException {  
 buildFile = testProjectDir.newFile("build.gradle");  
 }  
  
 @Test  
 **public** **void** testHelloWorldTask() **throws** IOException {  
 String buildFileContent = "task helloWorld {" +  
 " doLast {" +  
 " println 'Hello world!'" +  
 " }" +  
 "}";  
 writeFile(buildFile, buildFileContent);  
  
 BuildResult result = GradleRunner.create()  
 .withProjectDir(testProjectDir.getRoot())  
 .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 are written in the Groovy programming language, and as many plugins are implemented in Groovy, 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](https://code.google.com/p/spock/) as it offers many compelling features over the use of JUnit.

The following demonstrates the usage of Gradle runner in a Groovy Spock test:

[Example: Using GradleRunner with Spock](#18p6hfx)

**BuildLogicFunctionalTest.groovy**

**import** org.gradle.testkit.runner.GradleRunner  
**import** static org.gradle.testkit.runner.TaskOutcome.\*  
**import** org.junit.Rule  
**import** org.junit.rules.TemporaryFolder  
**import** spock.lang.Specification  
  
**class** **BuildLogicFunctionalTest** **extends** Specification {  
 @Rule **final** TemporaryFolder testProjectDir = **new** TemporaryFolder()  
 File buildFile  
  
 **def** **setup**() {  
 buildFile = testProjectDir.newFile('build.gradle')  
 }  
  
 **def** "hello world task prints hello world"() {  
 given:  
 buildFile << """  
 task helloWorld {  
 doLast {  
 println 'Hello world!'  
 }  
 }  
 """  
  
 when:  
 **def** result = GradleRunner.create()  
 .withProjectDir(testProjectDir.root)  
 .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](#3sou03q)

The GradleRunner uses the [Tooling API](#184mhaj) 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.

Starting with version 2.13, Gradle provides a [conventional mechanism to inject the code under test into the test build](#27u4abj).

For earlier versions of Gradle (before 2.13), it is possible to manually make the code under test available via some extra configuration. The following example demonstrates having the build generate a file containing the implementation classpath of the code under test, and making it available at test runtime.

[Example: Making the code under test classpath available to the tests](#mzekjc)

**build.gradle**

*// Write the plugin's classpath to a file to share with the tests*  
task createClasspathManifest {  
 **def** outputDir = file("$buildDir/$name")  
  
 inputs.files sourceSets.main.runtimeClasspath  
 outputs.dir outputDir  
  
 doLast {  
 outputDir.mkdirs()  
 file("$outputDir/plugin-classpath.txt").text = sourceSets.main.runtimeClasspath.join("\n")  
 }  
}  
  
*// Add the classpath file to the test runtime classpath*  
dependencies {  
 testRuntime files(createClasspathManifest)  
}

| **Note** | The code for this example can be found at samples/testKit/gradleRunner/manualClasspathInjection in the ‘-all’ distribution of Gradle. |
| --- | --- |

The tests can then read this value, and inject the classpath into the test build by using the method [GradleRunner.withPluginClasspath(java.lang.Iterable)](http://docs.google.com/javadoc/org/gradle/testkit/runner/GradleRunner.html#withPluginClasspath-java.lang.Iterable-). This classpath is then available to use to locate plugins in a test build via the plugins DSL (see [Plugins](#32hioqz)). Applying plugins with the plugins DSL requires the definition of a plugin identifier. The following is an example (in Groovy) of doing this from within a Spock Framework setup() method, which is analogous to a JUnit @Before method.

[Example: Injecting the code under test classes into test builds](#36z2375)

**src/test/groovy/org/gradle/sample/BuildLogicFunctionalTest.groovy**

List<File> pluginClasspath  
  
 **def** **setup**() {  
 buildFile = testProjectDir.newFile('build.gradle')  
  
 **def** pluginClasspathResource = getClass().classLoader.findResource("plugin-classpath.txt")  
 **if** (pluginClasspathResource == null) {  
 **throw** **new** IllegalStateException("Did not find plugin classpath resource, run `testClasses` build task.")  
 }  
  
 pluginClasspath = pluginClasspathResource.readLines().collect { **new** File(it) }  
 }  
  
 **def** "hello world task prints hello world"() {  
 given:  
 buildFile << """  
 plugins {  
 id 'org.gradle.sample.helloworld'  
 }  
 """  
  
 when:  
 **def** result = GradleRunner.create()  
 .withProjectDir(testProjectDir.root)  
 .withArguments('helloWorld')  
 .withPluginClasspath(pluginClasspath)  
 .build()  
  
 then:  
 result.output.contains('Hello world!')  
 result.task(":helloWorld").outcome == SUCCESS  
 }

| **Note** | The code for this example can be found at samples/testKit/gradleRunner/manualClasspathInjection in the ‘-all’ distribution of Gradle. |
| --- | --- |

This approach works well when executing the functional tests as part of the Gradle build. When executing the functional tests from an IDE, there are extra considerations. Namely, the classpath manifest file points to the class files etc. generated by Gradle and not the IDE. This means that after making a change to the source of the code under test, the source must be recompiled by Gradle. Similarly, if the effective classpath of the code under test changes, the manifest must be regenerated. In either case, executing the testClasses task of the build will ensure that things are up to date.

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. Please consult the documentation of your IDE for more information.

[Working with Gradle versions prior to 2.8](#1m4cdey)

The [GradleRunner.withPluginClasspath(java.lang.Iterable)](http://docs.google.com/javadoc/org/gradle/testkit/runner/GradleRunner.html#withPluginClasspath-java.lang.Iterable-) method will not work when executing the build with a Gradle version earlier than 2.8 (see [The version used to test](#463zw2r)), as this feature is not supported on such Gradle versions.

Instead, the code must be injected via the build script itself. The following sample demonstrates how this can be done.

[Example: Injecting the code under test classes into test builds for Gradle versions prior to 2.8](#2l9a6ak)

**src/test/groovy/org/gradle/sample/BuildLogicFunctionalTest.groovy**

List<File> pluginClasspath  
  
 **def** **setup**() {  
 buildFile = testProjectDir.newFile('build.gradle')  
  
 **def** pluginClasspathResource = getClass().classLoader.findResource("plugin-classpath.txt")  
 **if** (pluginClasspathResource == null) {  
 **throw** **new** IllegalStateException("Did not find plugin classpath resource, run `testClasses` build task.")  
 }  
  
 pluginClasspath = pluginClasspathResource.readLines().collect { **new** File(it) }  
 }  
  
 **def** "hello world task prints hello world with pre Gradle 2.8"() {  
 given:  
 **def** classpathString = pluginClasspath  
 .collect { it.absolutePath.replace('\\', '\\\\') } *// escape backslashes in Windows paths*  
 .collect { "'$it'" }  
 .join(", ")  
  
 buildFile << """  
 buildscript {  
 dependencies {  
 classpath files($classpathString)  
 }  
 }  
 apply plugin: "org.gradle.sample.helloworld"  
 """  
  
 when:  
 **def** result = GradleRunner.create()  
 .withProjectDir(testProjectDir.root)  
 .withArguments('helloWorld')  
 .withGradleVersion("2.7")  
 .build()  
  
 then:  
 result.output.contains('Hello world!')  
 result.task(":helloWorld").outcome == SUCCESS  
 }

| **Note** | The code for this example can be found at samples/testKit/gradleRunner/manualClasspathInjection in the ‘-all’ distribution of Gradle. |
| --- | --- |

[Automatic injection with the Java Gradle Plugin Development plugin](#27u4abj)

The [Java Gradle Plugin development plugin](#meukdy) 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 test compile configuration. Furthermore, it automatically generates the classpath for the code under test and injects it via [GradleRunner.withPluginClasspath()](http://docs.google.com/javadoc/org/gradle/testkit/runner/GradleRunner.html#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](#10ekgid). If the [target Gradle version](#463zw2r) 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](http://docs.google.com/javadoc/org/gradle/plugin/devel/GradlePluginDevelopmentExtension.html).

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: Using the Java Gradle Development plugin for generating the plugin metadata](#3ke7z66)

**build.gradle**

apply plugin: 'groovy'  
apply plugin: 'java-gradle-plugin'  
  
dependencies {  
 testCompile('org.spockframework:spock-core:1.1-groovy-2.4') {  
 exclude module: 'groovy-all'  
 }  
}

| **Note** | The code for this example can be found at samples/testKit/gradleRunner/automaticClasspathInjectionQuickstart in the ‘-all’ distribution of Gradle. |
| --- | --- |

[Example: Automatically injecting the code under test classes into test builds](#1zji9dz)

**src/test/groovy/org/gradle/sample/BuildLogicFunctionalTest.groovy**

**def** "hello world task prints hello world"() {  
 given:  
 buildFile << """  
 plugins {  
 id 'org.gradle.sample.helloworld'  
 }  
 """  
  
 when:  
 **def** result = GradleRunner.create()  
 .withProjectDir(testProjectDir.root)  
 .withArguments('helloWorld')  
 .withPluginClasspath()  
 .build()  
  
 then:  
 result.output.contains('Hello world!')  
 result.task(":helloWorld").outcome == SUCCESS  
}

| **Note** | The code for this example can be found at samples/testKit/gradleRunner/automaticClasspathInjectionQuickstart in the ‘-all’ distribution of Gradle. |
| --- | --- |

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.

[Example: Reconfiguring the classpath generation conventions of the Java Gradle Development plugin](#4jj5s1s)

**build.gradle**

apply plugin: 'groovy'  
apply plugin: 'java-gradle-plugin'  
  
sourceSets {  
 functionalTest {  
 groovy {  
 srcDir file('src/functionalTest/groovy')  
 }  
 resources {  
 srcDir file('src/functionalTest/resources')  
 }  
 compileClasspath += sourceSets.main.output + configurations.testRuntime  
 runtimeClasspath += output + compileClasspath  
 }  
}  
  
task functionalTest(type: Test) {  
 testClassesDirs = sourceSets.functionalTest.output.classesDirs  
 classpath = sourceSets.functionalTest.runtimeClasspath  
}  
  
check.dependsOn functionalTest  
  
gradlePlugin {  
 testSourceSets sourceSets.functionalTest  
}  
  
dependencies {  
 functionalTestCompile('org.spockframework:spock-core:1.1-groovy-2.4') {  
 exclude module: 'groovy-all'  
 }  
}

| **Note** | The code for this example can be found at samples/testKit/gradleRunner/automaticClasspathInjectionCustomTestSourceSet in the ‘-all’ distribution of Gradle. |
| --- | --- |

[Controlling the build environment](#2yog29l)

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 directory (e.g. ~/.gradle/gradle.properties) is not used for test execution. The TestKit does not expose a mechanism for fine grained control of environment variables etc. 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 Gradle version used to test](#463zw2r)

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)](http://docs.google.com/javadoc/org/gradle/testkit/runner/GradleRunner.html#withGradleVersion-java.lang.String-)
* [GradleRunner.withGradleInstallation(java.io.File)](http://docs.google.com/javadoc/org/gradle/testkit/runner/GradleRunner.html#withGradleInstallation-java.io.File-)
* [GradleRunner.withGradleDistribution(java.net.URI)](http://docs.google.com/javadoc/org/gradle/testkit/runner/GradleRunner.html#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](#1dtqche)

**BuildLogicFunctionalTest.groovy**

**import** org.gradle.testkit.runner.GradleRunner  
**import** static org.gradle.testkit.runner.TaskOutcome.\*  
**import** org.junit.Rule  
**import** org.junit.rules.TemporaryFolder  
**import** spock.lang.Specification  
**import** spock.lang.Unroll  
  
**class** **BuildLogicFunctionalTest** **extends** Specification {  
 @Rule **final** TemporaryFolder testProjectDir = **new** TemporaryFolder()  
 File buildFile  
  
 **def** **setup**() {  
 buildFile = testProjectDir.newFile('build.gradle')  
 }  
  
 @Unroll  
 **def** "can execute hello world task with Gradle version #gradleVersion"() {  
 given:  
 buildFile << """  
 task helloWorld {  
 doLast {  
 logger.quiet 'Hello world!'  
 }  
 }  
 """  
  
 when:  
 **def** result = GradleRunner.create()  
 .withGradleVersion(gradleVersion)  
 .withProjectDir(testProjectDir.root)  
 .withArguments('helloWorld')  
 .build()  
  
 then:  
 result.output.contains('Hello world!')  
 result.task(":helloWorld").outcome == SUCCESS  
  
 where:  
 gradleVersion << ['2.6', '2.7']  
 }  
}

[Feature support when testing with different Gradle versions](#3xtdv57)

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 4. Gradle version compatibility

| **Feature** | **Minimum Version** | **Description** |
| --- | --- | --- |
| Inspecting executed tasks | 2.5 | Inspecting the executed tasks, using [BuildResult.getTasks()](http://docs.google.com/javadoc/org/gradle/testkit/runner/BuildResult.html#getTasks--) and similar methods. |
| [Plugin classpath injection](#3sou03q) | 2.8 | Injecting the code under test via[GradleRunner.withPluginClasspath(java.lang.Iterable)](http://docs.google.com/javadoc/org/gradle/testkit/runner/GradleRunner.html#withPluginClasspath-java.lang.Iterable-). |
| [Inspecting build output in debug mode](#2cyo5d0) | 2.9 | Inspecting the build’s text output when run in debug mode, using [BuildResult.getOutput()](http://docs.google.com/javadoc/org/gradle/testkit/runner/BuildResult.html#getOutput--). |
| [Automatic plugin classpath injection](#27u4abj) | 2.13 | Injecting the code under test automatically via [GradleRunner.withPluginClasspath()](http://docs.google.com/javadoc/org/gradle/testkit/runner/GradleRunner.html#withPluginClasspath--) by applying the Java Gradle Plugin Development plugin. |

[Debugging build logic](#2cyo5d0)

The runner uses the [Tooling API](#184mhaj) 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)](http://docs.google.com/javadoc/org/gradle/testkit/runner/GradleRunner.html#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](#s3yfkt)

To enable the [Build Cache](#2bn6wsx) in your tests, you can pass the --build-cache argument to [GradleRunner](http://docs.google.com/javadoc/org/gradle/testkit/runner/GradleRunner.html) or use one of the other methods described in [Enable the build cache](#3qg2avn). You can then check for the task outcome [TaskOutcome.FROM\_CACHE](http://docs.google.com/javadoc/org/gradle/testkit/runner/TaskOutcome.html#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](#3c3ly8m)

**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.root, '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)](http://docs.google.com/javadoc/org/gradle/testkit/runner/GradleRunner.html#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](#1r8w8gf)

**BuildLogicFunctionalTest.groovy**

@Rule **final** TemporaryFolder testProjectDir = **new** TemporaryFolder()  
File buildFile  
File localBuildCacheDirectory  
  
**def** **setup**() {  
 localBuildCacheDirectory = testProjectDir.newFolder('local-cache')  
 testProjectDir.newFile('settings.gradle') << """  
 buildCache {  
 local {  
 directory '${localBuildCacheDirectory.toURI()}'  
 }  
 }  
 """  
 buildFile = testProjectDir.newFile('build.gradle')  
}

[Using Gradle Plugins](#32hioqz)

Gradle at its core intentionally provides very little for real world automation. All of the useful features, like the ability to compile Java code, are added by *plugins*. Plugins add new tasks (e.g. [JavaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.JavaCompile.html)), domain objects (e.g. [SourceSet](http://docs.google.com/dsl/org.gradle.api.tasks.SourceSet.html)), conventions (e.g. Java source is located at src/main/java) as well as extending core objects and objects from other plugins.

In this chapter we discuss how to use plugins and the terminology and concepts surrounding plugins.

[What plugins do](#4b8jr48)

Applying a plugin to a project allows the plugin to extend the project’s capabilities. It can do things such as:

* Extend the Gradle model (e.g. add new DSL elements that can be configured)
* Configure the project according to conventions (e.g. add new tasks or configure sensible defaults)
* Apply specific configuration (e.g. add organizational repositories or enforce standards)

By applying plugins, rather than adding logic to the project build script, we can reap a number of benefits. Applying plugins:

* Promotes reuse and reduces the overhead of maintaining similar logic across multiple projects
* Allows a higher degree of modularization, enhancing comprehensibility and organization
* Encapsulates imperative logic and allows build scripts to be as declarative as possible

[Types of plugins](#2qdu1c1)

There are two general types of plugins in Gradle, *script* plugins and *binary* plugins. Script plugins are additional build scripts that further configure the build and usually implement a declarative approach to manipulating the build. They are typically used within a build although they can be externalized and accessed from a remote location. Binary plugins are classes that implement the [Plugin](http://docs.google.com/javadoc/org/gradle/api/Plugin.html) interface and adopt a programmatic approach to manipulating the build. Binary plugins can reside within a build script, within the project hierarchy or externally in a plugin jar.

A plugin often starts out as a script plugin (because they are easy to write) and 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](#31k882z)

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](http://docs.google.com/dsl/org.gradle.api.Project.html).

*Resolving* a plugin means finding the correct version of the jar which contains a given plugin and adding it 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.

*Applying* a plugin means actually executing the plugin’s [Plugin.apply(T)](http://docs.google.com/javadoc/org/gradle/api/Plugin.html#apply-T-) on the Project you want to enhance with the plugin. Applying plugins is *idempotent*. That is, you can safely apply any plugin multiple times without side effects.

The most common use case for using a plugin is to both resolve the plugin and apply it to the current project. Since this is such a common use case, it’s recommended that build authors use the [plugins DSL](#10ekgid) to both resolve and apply plugins in one step. The feature is technically still incubating, but it works well, and should be used by most users.

[Script plugins](#2dvym10)

[Example: Applying a script plugin](#15j4bju)

**build.gradle**

apply from: 'other.gradle'

Script plugins are automatically resolved and can be applied from a script on the local filesystem or at a remote location. Filesystem locations are relative to the project directory, while remote script locations are specified with an HTTP URL. Multiple script plugins (of either form) can be applied to a given target.

[Binary plugins](#3piru7n)

You apply plugins by their *plugin id*, which is a globally unique identifier, or name, for plugins. Core Gradle plugins are special in that they provide short names, such as 'java' for the core [JavaPlugin](http://docs.google.com/javadoc/org/gradle/api/plugins/JavaPlugin.html). All other binary plugins must use the fully qualified form of the plugin id (e.g. com.github.foo.bar), although some legacy plugins may still utilize a short, unqualified form. Where you put the plugin id depends on whether you are using the [plugins DSL](#10ekgid) or the [buildscript block.](#24o24fg)

[Locations of binary plugins](#jtcen9)

A plugin is simply any class that implements the [Plugin](http://docs.google.com/javadoc/org/gradle/api/Plugin.html) interface. Gradle provides the core plugins (e.g. JavaPlugin) as part of its distribution which means they are automatically resolved. However, non-core binary plugins need to be resolved before they can be applied. This can be achieved in a number of ways:

* Including the plugin from the plugin portal or a [custom repository](#33szxb2) using the plugins DSL (see [Applying plugins using the plugins DSL](#10ekgid)).
* Including the plugin from an external jar defined as a buildscript dependency (see see [Applying plugins using the buildscript block](#1iya7iv)).
* Defining the plugin as a source file under the buildSrc directory in the project (see [Using buildSrc to extract functional logic](#xevivl)).
* Defining the plugin as an inline class declaration inside a build script.

For more on defining your own plugins, see [Custom Plugins](#279ka65).

[Applying plugins with the plugins DSL](#10ekgid)

| **Note** | The plugins DSL is currently [incubating](#2xcytpi). Please be aware that the DSL and other configuration may change in later Gradle versions. |
| --- | --- |

The new plugins DSL provides a succinct and convenient way to declare plugin dependencies. It works with the [Gradle plugin portal](http://plugins.gradle.org) to provide easy access to both core and community plugins. The plugins DSL block configures an instance of [PluginDependenciesSpec](http://docs.google.com/javadoc/org/gradle/plugin/use/PluginDependenciesSpec.html).

To apply a core plugin, the short name can be used:

[Example: Applying a core plugin](#42xxq6o)

**build.gradle**

plugins {  
 id 'java'  
}

To apply a community plugin from the portal, the fully qualified plugin id must be used:

[Example: Applying a community plugin](#2i380eh)

**build.gradle**

plugins {  
 id 'com.jfrog.bintray' version '0.4.1'  
}

See [PluginDependenciesSpec](http://docs.google.com/javadoc/org/gradle/plugin/use/PluginDependenciesSpec.html) for more information on using the Plugin DSL.

[Limitations of the plugins DSL](#x8iama)

This way of adding plugins to a project is much more than a more convenient syntax. The plugins DSL is processed in a way which allows Gradle to determine the plugins in use very early and very quickly. This allows Gradle to do smart things such as:

* Optimize the loading and reuse of plugin classes.
* Allow different plugins to use different versions of dependencies.
* Provide editors detailed information about the potential properties and values in the buildscript for editing assistance.

This requires that plugins be specified in a way that Gradle can easily and quickly extract, before executing the rest of the build script. It also requires that the definition of plugins to use be somewhat static.

There are some key differences between the new plugin mechanism and the “traditional” apply() method mechanism. There are also some constraints, some of which are temporary limitations while the mechanism is still being developed and some are inherent to the new approach.

[Constrained Syntax](#3h85ta3)

The new plugins {} block does not support arbitrary Groovy code. It is constrained, in order 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» version «plugin version» [apply «false»]  
}

Where «plugin version» and «plugin id» must be constant, literal, strings and the apply statement with a boolean can be used to disable the default behavior of applying the plugin immediately (e.g. you want to apply it only in subprojects). No other statements are allowed; their presence will cause a compilation error.

The plugins {} block must also be a top level statement in the buildscript. It cannot be nested inside another construct (e.g. an if-statement or for-loop).

[Can only be used in build scripts](#1wdg3hw)

The plugins {} block can currently only be used in a project’s build script. It cannot be used in script plugins, the settings.gradle file or init scripts.

*Future versions of Gradle will remove this restriction.*

If the restrictions of the new syntax are prohibitive, the recommended approach is to apply plugins using the [buildscript {} block](#1iya7iv).

[Applying plugins to subprojects](#4gd3m5p)

If you have a [multi-project build](#2p2csry), you probably want to apply plugins to some or all of the subprojects in your build, but not to the root or master project. The default behavior of the plugins {} block is to immediately resolve *and* apply the plugins. But, you can use the apply false syntax to tell Gradle not to apply the plugin to the current project and then use apply plugin: «plugin id» in the subprojects block:

[Example: Applying plugins only on certain subprojects.](#2vidwdi)

**settings.gradle**

include 'helloA'  
include 'helloB'  
include 'goodbyeC'

**build.gradle**

plugins {  
 id "org.gradle.sample.hello" version "1.0.0" apply false  
 id "org.gradle.sample.goodbye" version "1.0.0" apply false  
}  
  
subprojects { subproject ->  
 **if** (subproject.name.startsWith("hello")) {  
 apply plugin: 'org.gradle.sample.hello'  
 }  
 **if** (subproject.name.startsWith("goodbye")) {  
 apply plugin: 'org.gradle.sample.goodbye'  
 }  
}

If you then run gradle hello you’ll see that only the helloA and helloB subprojects had the hello plugin applied.

gradle/subprojects/docs/src/samples/plugins/multiproject $> gradle hello  
Parallel execution is an incubating feature.  
:helloA:hello  
:helloB:hello  
Hello!  
Hello!  
  
BUILD SUCCEEDED

[Applying plugins from the *buildSrc* directory](#1ano6lb)

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 a plugin implementation class — my.MyPlugin — defined in *buildSrc* to the ID "my-plugin":

*Example 1. Defining a buildSrc plugin with an ID*

**buildSrc/build.gradle**

plugins {  
 id 'java'  
 id 'java-gradle-plugin'  
}  
  
gradlePlugin {  
 plugins {  
 myPlugins {  
 id = 'my-plugin'  
 implementationClass = 'my.MyPlugin'  
 }  
 }  
}  
  
dependencies {  
 compileOnly gradleApi()  
}

The plugin can then be applied by ID as normal:

*Example 2. Applying a plugin from buildSrc*

**build.gradle**

plugins {  
 id 'my-plugin'  
}

[Plugin Management](#3unbp94)

| **Note** | The pluginManagement {} DSL is currently [incubating](#2xcytpi). Please be aware that the DSL and other configuration may change in later Gradle versions. |
| --- | --- |

The pluginManagement {} block may only appear in either the settings.gradle file, where it must be the first block in the file, or in an [Initialization Script](#2jxsxqh).

[Example: Configuring pluginManagement per-project and globally](#29slzgx)

**settings.gradle**

pluginManagement {  
 resolutionStrategy {  
 }  
 repositories {  
 }  
}

**init.gradle**

settingsEvaluated { settings ->  
 settings.pluginManagement {  
 resolutionStrategy {  
 }  
 repositories {  
 }  
 }  
}

[Custom Plugin Repositories](#33szxb2)

By default, the plugins {} DSL resolves plugins from the public [Gradle Plugin Portal.](https://plugins.gradle.org) Many build authors would also like to resolve plugins from private Maven or Ivy repositories because the plugins contain proprietary implementation details, or just to have more control over what plugins are available to their builds.

To specify custom plugin repositories, use the repositories {} block inside pluginManagement {}:

[Example: Using plugins from custom plugin repositories.](#oxw9oq)

**settings.gradle**

pluginManagement {  
 repositories {  
 maven {  
 url 'maven-repo'  
 }  
 gradlePluginPortal()  
 ivy {  
 url '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 Resolution Rules](#38xjscj)

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:

[Example: Plugin resolution strategy.](#1o2u2kc)

**settings.gradle**

pluginManagement {  
 resolutionStrategy {  
 eachPlugin {  
 **if** (requested.id.namespace == 'org.gradle.sample') {  
 useModule('org.gradle.sample:sample-plugins:1.0.0')  
 }  
 }  
 }  
 repositories {  
 maven {  
 url 'maven-repo'  
 }  
 gradlePluginPortal()  
 ivy {  
 url 'ivy-repo'  
 }  
 }  
}

This tells Gradle to use the specified plugin implementation artifact instead of using its built-in default mapping from plugin ID to Maven/Ivy coordinates.

Custom Maven and Ivy plugin repositories must contain [plugin marker artifacts](#482hl85) in addition to the artifacts which actually implement the plugin. For more information on publishing plugins to custom repositories read [Gradle Plugin Development Plugin](#meukdy).

See [PluginManagementSpec](http://docs.google.com/javadoc/org/gradle/plugin/management/PluginManagementSpec.html) for complete documentation for using the pluginManagement {} block.

[Plugin Marker Artifacts](#482hl85)

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](#meukdy).

For example, the following complete sample from the sample-plugins project shows how to publish a org.gradle.sample.hello plugin and a org.gradle.sample.goodbye plugin to both an Ivy and Maven repository using the combination of the [java-gradle-plugin](#meukdy), the [maven-publish](#1egqt2p) plugin, and the [ivy-publish](#3ygebqi) plugin.

[Example: Complete Plugin Publishing Sample](#2n7rvfy)

**build.gradle**

plugins {  
 id 'java-gradle-plugin'  
 id 'maven-publish'  
 id 'ivy-publish'  
}  
  
group 'org.gradle.sample'  
version '1.0.0'  
  
gradlePlugin {  
 plugins {  
 hello {  
 id = "org.gradle.sample.hello"  
 implementationClass = "org.gradle.sample.hello.HelloPlugin"  
 }  
 goodbye {  
 id = "org.gradle.sample.goodbye"  
 implementationClass = "org.gradle.sample.goodbye.GoodbyePlugin"  
 }  
 }  
}  
  
publishing {  
 repositories {  
 maven {  
 url "../consuming/maven-repo"  
 }  
 ivy {  
 url "../consuming/ivy-repo"  
 }  
 }  
}

Running gradle publish in the sample directory causes the following repo layouts to exist:



[Legacy Plugin Application](#24o24fg)

With the introduction of the [plugins DSL](#10ekgid), 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 plugins DSL due to restrictions in how it currently works.

[Applying Binary Plugins](#12d25nr)

[Example: Applying a binary plugin](#3mcpobk)

**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](http://docs.google.com/javadoc/org/gradle/api/plugins/JavaPlugin.html).

Rather than using a plugin id, plugins can also be applied by simply specifying the class of the plugin:

[Example: Applying a binary plugin by type](#21hzyjd)

**build.gradle**

apply plugin: JavaPlugin

The JavaPlugin symbol in the above sample refers to the [JavaPlugin](http://docs.google.com/javadoc/org/gradle/api/plugins/JavaPlugin.html). 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](#gna8r6)). Furthermore, it is not necessary to append .class to identify a class literal in Groovy as it is in Java.

[Applying plugins with the buildscript block](#1iya7iv)

Binary plugins that have been published as external jar files can be added to a project by adding the plugin to the build script classpath and then applying the plugin. External jars can be added to the build script classpath using the buildscript {} block as described in [External dependencies for the build script](#3utoxif).

[Example: Applying a plugin with the buildscript block](#30mxrez)

**build.gradle**

buildscript {  
 repositories {  
 jcenter()  
 }  
 dependencies {  
 classpath "com.jfrog.bintray.gradle:gradle-bintray-plugin:0.4.1"  
 }  
}  
  
apply plugin: "com.jfrog.bintray"

[Finding community plugins](#1fs81ms)

Gradle has a vibrant community of plugin developers who contribute plugins for a wide variety of capabilities. The Gradle [plugin portal](https://plugins.gradle.org) provides an interface for searching and exploring community plugins.

[More on plugins](#3zrvkal)

This chapter aims to serve as an introduction to plugins and Gradle and the role they play. For more information on the inner workings of plugins, see [Custom Plugins](#279ka65).

[Working With Files](#1hmsyys)

Almost every Gradle build interacts with files in some way: think source files, file dependencies, reports and so on. That’s why Gradle comes with a comprehensive API that makes it simple to perform the file operations you need.

The API has two parts to it:

* Specifying which files and directories to process
* Specifying what to do with them

The [File paths in depth](#2ex5uie) section covers the first of these in detail, while subsequent sections, like [File copying in depth](#u2g4q7), cover the second. To begin with, we’ll show you examples of the most common scenarios that users encounter.

[Copying a single file](#3e23ne0)

You copy a file by creating an instance of Gradle’s builtin [Copy](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html) 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:

[Example: How to copy a single file](#1t7dxlt)

**build.gradle**

task copyReport(type: Copy) {  
 from file("${buildDir}/reports/my-report.pdf")  
 into file("${buildDir}/toArchive")  
}

The [Project.file(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:file(java.lang.Object)) method is used to create a file or directory path relative to the current project and is a common way to make build scripts work regardless of the project path. The file and directory paths are then used to specify what file to copy using [Copy.from(java.lang.Object…​)](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html#org.gradle.api.tasks.Copy:from(java.lang.Object%5B%5D)) and which directory to copy it to using [Copy.into(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html#org.gradle.api.tasks.Copy:into(java.lang.Object)).

You can even use the path directly without the file() method, as explained early in the section [File copying in depth](#u2g4q7):

[Example: Using implicit string paths](#4d71g9m)

**build.gradle**

task copyReport2(type: Copy) {  
 from "${buildDir}/reports/my-report.pdf"  
 into "${buildDir}/toArchive"  
}

Although hard-coded paths make for simple examples, they also make the build brittle. It’s better to use a reliable, single source of truth, such as a task or shared project property. In the following modified example, we use a report task defined elsewhere that has the report’s location stored in its outputFile property:

[Example: Prefer task/project properties over hard-coded paths](#2scbqhf)

**build.gradle**

task copyReport3(type: 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](#17hm0p8)

You can extend the previous examples to multiple files very easily by providing multiple arguments to from():

[Example: Using multiple arguments with from()](#3rh9jd1)

**build.gradle**

task copyReportsForArchiving(type: Copy) {  
 from "${buildDir}/reports/my-report.pdf", "src/docs/manual.pdf"  
 into "${buildDir}/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](#u2g4q7).

Now consider another example: what if you want to copy all the PDFs in a directory without having to specify 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:

[Example: Using a flat filter](#26mjtku)

**build.gradle**

task copyPdfReportsForArchiving(type: Copy) {  
 from "${buildDir}/reports"  
 include "\*.pdf"  
 into "${buildDir}/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:



*Figure 8. The effect of a flat filter on copying*

You can include files in subdirectories by using an Ant-style glob pattern (\*\*/\*), as done in this updated example:

[Example: Using a deep filter](#lru3sn)

**build.gradle**

task copyAllPdfReportsForArchiving(type: Copy) {  
 from "${buildDir}/reports"  
 include "\*\*/\*.pdf"  
 into "${buildDir}/toArchive"  
}

This task has the following effect:



*Figure 9. The effect of a deep filter on copying*

One thing to bear in mind is that a deep filter like this has the side effect of copying the directory structure below reports as well as the files. If you just want to copy the files without the directory structure, you need to use an explicit fileTree(*dir*) { *includes* }.files expression. We talk more about the difference between file trees and file collections in the [File trees](#35rhmgg) section.

This is just one of the variations in behavior you’re likely to come across when dealing with file operations in Gradle builds. Fortunately, Gradle provides elegant solutions to almost all those use cases. Read the *in-depth* sections later in the chapter for more detail on how the file operations work in Gradle and what options you have for configuring them.

[Copying directory hierarchies](#1kwrwo9)

You may have a need to copy not just files, but the directory structure they reside in as well. 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:

[Example: Copying an entire directory](#44wffc2)

**build.gradle**

task copyReportsDirForArchiving(type: Copy) {  
 from "${buildDir}/reports"  
 into "${buildDir}/toArchive"  
}

The key aspect that users struggle 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 $buildDir? The answer is to add it as an include pattern:

[Example: Copying an entire directory, including itself](#2k1ppjv)

**build.gradle**

task copyReportsDirForArchiving2(type: Copy) {  
 from("${buildDir}") {  
 include "reports/\*\*"  
 }  
 into "${buildDir}/toArchive"  
}

You’ll get the same behavior as before except with one extra level of directory 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 easily handle most requirements that you will come across. You can learn more about this in the section on [child specifications](#z6zzro).

[Creating archives (zip, tar, etc.)](#3j6nifh)

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. This means that creating archives looks a lot like copying, with all of the same features!

The simplest case involves archiving the entire contents of a directory, which this example demonstrates by creating a ZIP of the toArchive directory:

[Example: Archiving a directory as a ZIP](#1ybxsna)

**build.gradle**

task packageDistribution(type: Zip) {  
 archiveName = "my-distribution.zip"  
 destinationDir = file("${buildDir}/dist")  
  
 from "${buildDir}/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](#vx1227). It provides some conventional values for those properties. The next example demonstrates this and you can learn more about the conventions in the [archive naming](#4iblbb3) section.

Each type of archive has its own task type, the most common ones being [Zip](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Zip.html), [Tar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Tar.html) and [Jar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Jar.html). 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 subdirectories of the archive. For example, let’s say you want to package all PDFs into a docs directory in the root of the archive. This docs directory doesn’t exist in the source location, so you have to create it as part of the archive. You do this by adding an into() declaration for just the PDFs:

[Example: Using the Base Plugin for its archive name convention](#2xgvliw)

**build.gradle**

plugins {  
 id 'base'  
}  
  
version = "1.0.0"  
  
task packageDistribution(type: Zip) {  
 from("${buildDir}/toArchive") {  
 exclude "\*\*/\*.pdf"  
 }  
  
 from("${buildDir}/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](#z6zzro) for more information on this feature.

[Unpacking archives](#1cm5vqp)

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](#35rhmgg)).

The two functions of interest are [Project.zipTree(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:zipTree(java.lang.Object)) and [Project.tarTree(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:tarTree(java.lang.Object)), which produce a [FileTree](http://docs.google.com/javadoc/org/gradle/api/file/FileTree.html) from a corresponding archive file. That file tree can then be used in a from() specification, like so:

[Example: Unpacking a ZIP file](#3wlteei)

**build.gradle**

task unpackFiles(type: Copy) {  
 from zipTree("src/resources/thirdPartyResources.zip")  
 into "${buildDir}/resources"  
}

As with a normal copy, you can control which files are unpacked via filters and even rename files as they are unpacked.

If you’re a Java developer and are wondering why there is no jarTree() method, that’s because zipTree() works perfectly well for JARs, WARs and EARs.

[Creating "uber" or "fat" JARs](#2br3omb)

In the Java space, applications and their dependencies typically used to be packaged as separate JARs within a single distribution archive. That still happens, but there is another approach that is now common: placing the classes and resources of the dependencies directly into the application JAR, creating what is known as an uber or fat JAR.

Gradle makes this approach easy to accomplish. Consider the aim: to copy the contents of other JAR files into the application JAR. All you need for this is the [Project.zipTree(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:zipTree(java.lang.Object)) method and the [Jar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Jar.html) task, as demonstrated by the uberJar task in the following example:

[Example: Creating a Java uber or fat JAR](#qwdyu4)

**build.gradle**

plugins {  
 id 'java'  
}  
  
version = '1.0.0'  
  
repositories {  
 mavenCentral()  
}  
  
dependencies {  
 implementation 'commons-io:commons-io:2.6'  
}  
  
task uberJar(type: Jar) {  
 appendix = 'uber'  
  
 from sourceSets.main.output  
 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](#3aw1hhx)

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. You can learn about this feature in the [incremental build](#odc9jc) section of the user guide. All core Gradle tasks ensure that any output directories they need are created if necessary using this mechanism.

In cases where you need to create a directory manually, you can use the [Project.mkdir(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:mkdir(java.lang.Object)) method from within your build scripts or custom task implementations. Here’s a simple example that creates a single images directory in the project folder:

[Example: Manually creating a directory](#1q1brpq)

**build.gradle**

task ensureDirectory {  
 doLast {  
 mkdir "images"  
 }  
}

As described in the [Apache Ant manual](https://ant.apache.org/manual/Tasks/mkdir.html), the mkdir task will automatically create all necessary directories in the given path and will do nothing if the directory already exists.

[Moving files and directories](#4a0zadj)

Gradle has no API for moving files and directories around, but you can use the [Apache Ant integration](#pkwqa1) to easily do that, as shown in this example:

[Example: Moving a directory using the Ant task](#2p69klc)

**build.gradle**

task moveReports {  
 doLast {  
 ant.move file: "${buildDir}/reports",  
 todir: "${buildDir}/toArchive"  
 }  
}

This is not a common requirement and should be used sparingly as you lose information and can easily break a build. It’s generally preferable to copy directories and files instead.

[Renaming files on copy](#14bjut5)

The files used and generated by your builds sometimes don’t have names that suit, in which case you want to 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:

[Example: Renaming files as they are copied](#3ob7dgy)

**build.gradle**

task copyFromStaging(type: Copy) {  
 from "src/main/webapp"  
 into "${buildDir}/explodedWar"  
  
 rename '(.+)-staging(.+)', '$1$2'  
}

You can use regular expressions for this, as in the above example, or closures that use more complex logic to determine the target filename. For example, the following task truncates filenames:

[Example: Truncating filenames as they are copied](#23ghnor)

**build.gradle**

task copyWithTruncate(type: Copy) {  
 from "${buildDir}/reports"  
 rename { String filename ->  
 **if** (filename.size() > 10) {  
 **return** filename[0..7] + "~" + filename.size()  
 }  
 **else** **return** filename  
 }  
 into "${buildDir}/toArchive"  
}

As with filtering, you can also apply renaming to a subset of files by configuring it as part of a child specification on a from().

[Deleting files and directories](#ilrxwk)

You can easily delete files and directories using either the [Delete](http://docs.google.com/dsl/org.gradle.api.tasks.Delete.html) task or the [Project.delete(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:delete(org.gradle.api.Action)) method. In both cases, you specify which files and directories to delete in a way supported by the [Project.files(java.lang.Object…​)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:files(java.lang.Object%5B%5D)), [ProjectLayout.files(java.lang.Object…​)](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#files-java.lang.Object...-), and [ProjectLayout.configurableFiles(java.lang.Object…​)](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#configurableFiles-java.lang.Object...-) methods.

For example, the following task deletes the entire contents of a build’s output directory:

[Example: Deleting a directory](#32lfgkd)

**build.gradle**

task myClean(type: Delete) {  
 delete buildDir  
}

If you want more control over which files are deleted, you can’t use inclusions and exclusions in the same way as for copying files. Instead, you have to use the builtin filtering mechanisms of FileCollection and FileTree. The following example does just that to clear out temporary files from a source directory:

[Example: Deleting files matching a specific pattern](#1hqpqs6)

**build.gradle**

task cleanTempFiles(type: Delete) {  
 delete fileTree("src").matching {  
 include "\*\*/\*.tmp"  
 }  
}

You’ll learn more about file collections and file trees in the next section.

[File paths in depth](#2ex5uie)

In order 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](https://docs.oracle.com/javase/7/docs/api/java/io/File.html) class, which represents the location of a single file, and provides new APIs for dealing with collections of paths. This section shows you how to use the Gradle APIs to specify file paths for use in tasks and file operations.

But first, an important note on using hard-coded file paths in your builds.

[On hard-coded file paths](#41qd9fz)

Many examples in this chapter use hard-coded paths as string literals. This makes them easy to understand, but it’s not good practice for real builds. The problem is that paths often change and the more places you need to change them, the more likely you are to miss one and break the build.

Where possible, you should use tasks, task properties, and [project properties](#2emvufp) — in that order of preference — to configure file paths. For example, if you were to create a task that packages the compiled classes of a Java application, you should aim for something like this:

[Example: How to minimize the number of hard-coded paths in your build](#2gvnjns)

**build.gradle**

ext {  
 archivesDirPath = "${buildDir}/archives"  
}  
  
task packageClasses(type: Zip) {  
 appendix = "classes"  
 destinationDir = file(archivesDirPath)  
  
 from compileJava  
}

See how we’re using the compileJava task as the source of the files to package and we’ve created a project property archivesDirPath to store the location where we put archives, on the basis we’re likely to use it elsewhere in the build.

Using a task directly as an argument like this relies on it having [defined outputs](#4hae2tp), so it won’t always be possible. In addition, this example could be improved further by relying on the Java plugin’s convention for destinationDir rather than overriding it, but it does demonstrate the use of project properties.

[Single files and directories](#w0xtvl)

Gradle provides the [Project.file(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.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), as this creates a path relative to the current working directory, which could be anywhere. |
| --- | --- |

Here are some examples of using the file() method with different types of argument:

[Example: Locating files](#3g0lcje)

**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](https://docs.oracle.com/javase/7/docs/api/java/nio/file/Path.html) instances to the file() method, all of which result in an absolute File object. You can find other options for argument types in the reference guide, linked in the previous paragraph.

What happens in the case of multi-project builds? The file() method will always turn relative paths into paths that are relative to the current project directory, which may be a child project. If you want to use a path that’s relative to the *root project* directory, then you need to use the special [Project.getRootDir()](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:rootDir) property to construct an absolute path, like so:

[Example: Creating a path relative to a parent project](#1v5vmr7)

**build.gradle**

File configFile = file("${rootDir}/shared/config.xml")

Let’s say you’re working on a multi-project build in a dev/projects/AcmeHealth directory. You use the above example in the build of the library you’re fixing — at AcmeHealth/subprojects/AcmePatientRecordLib/build.gradle. The file path will resolve to the absolute version of dev/projects/AcmeHealth/shared/config.xml.

The file() method can be used to configure any task that has a property of type File. Many tasks, though, work on multiple files, so we look at how to specify sets of files next.

[File collections](#4f5j5f0)

A *file collection* is simply a set of file paths that’s represented by the [FileCollection](http://docs.google.com/javadoc/org/gradle/api/file/FileCollection.html) interface. *Any* file paths. It’s important to understand that the file paths don’t have to be related in any way, so they don’t have to be in the same directory or even have a shared parent directory. You will also find that many parts of the Gradle API use FileCollection, such as the copying API discussed later in this chapter and [dependency configurations](#37m2jsg).

The recommended way to specify a collection of files is to use the link:../javadoc/org/gradle/api/file/ProjectLayout.html#files-java.lang.Object...-[ProjectLayout.files(java.lang.Object...) method, which returns a FileCollection instance. This method is very flexible and allows you to pass multiple strings, File instances, collections of strings, collections of Files, and more. You can even pass in tasks as arguments if they have [defined outputs](#4hae2tp). Learn about all the supported argument types in the reference guide.

As with the [Project.file(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:file(java.lang.Object)) method covered in the [previous section](#w0xtvl), 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, a list and a [Path](https://docs.oracle.com/javase/7/docs/api/java/nio/file/Path.html):

[Example: Creating a file collection](#2uatfmt)

**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 some important attributes in Gradle. They can be:

* created lazily
* iterated over
* filtered
* combined

*Lazy creation* of a file collection is useful when you need to evaluate the files that make up a collection at the time 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:

[Example: Implementing a file collection](#19g3pum)

**build.gradle**

task list {  
 doLast {  
 File srcDir  
  
 *// Create a file collection using a closure*  
 collection = layout.files { srcDir.listFiles() }  
  
 srcDir = file('src')  
 println "Contents of $srcDir.name"  
 collection.collect { relativePath(it) }.sort().each { println it }  
  
 srcDir = file('src2')  
 println "Contents of $srcDir.name"  
 collection.collect { relativePath(it) }.sort().each { println it }  
 }  
}

**Output of** gradle -q list

> 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 to the files() method. Your closure simply needs to return a value of a type accepted by files(), such as List<File>, String, FileCollection, etc.

*Iterating over a file collection* can be done through the each() method 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 as well as how you can convert file collections to other types using the as operator or supported properties:

[Example: Using a file collection](#3tfr8if)

**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  
 File file2 = collection **as** File  
  
 *// Add and subtract collections*  
 **def** union = collection + layout.files('src/file2.txt')  
 **def** difference = collection - layout.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 in 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*. If you want to use a subset of a file collection, you can take advantage of the [FileCollection.filter(org.gradle.api.specs.Spec)](http://docs.google.com/javadoc/org/gradle/api/file/FileCollection.html#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:

[Example: Filtering a file collection](#28l1iq8)

**build.gradle**

FileCollection textFiles = collection.filter { File f ->  
 f.name.endsWith(".txt")  
}

**Output of** gradle -q filterTextFiles

> 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.

[File trees](#35rhmgg)

A *file tree* is a file collection that retains the directory structure of the files it contains and has the type [FileTree](http://docs.google.com/javadoc/org/gradle/api/file/FileTree.html). This means that 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 common case of copying files:



*Figure 10. The differences in how file trees and file collections behave when copying files*

| **Note** | Although FileTree extends FileCollection (an is-a relationship), their behaviors do differ. In other words, you can use a file tree wherever a file collection is required, but remember: 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()](http://docs.google.com/javadoc/org/gradle/api/file/FileTree.html#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)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.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 the basic method and, in addition, how to filter the files and directories using Ant-style patterns:

[Example: Creating a file tree](#nqbsy1)

**build.gradle**

*// Create a file tree with a base directory*  
FileTree tree = fileTree(dir: 'src/main')  
  
*// Add include and exclude patterns to the tree*  
tree.include '\*\*/\*.java'  
tree.exclude '\*\*/Abstract\*'  
  
*// Create a tree using path*  
tree = fileTree('src').include('\*\*/\*.java')  
  
*// 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](http://docs.google.com/javadoc/org/gradle/api/tasks/util/PatternFilterable.html). Also, see the API documentation for fileTree() to see what types you can pass as the base directory.

By default, fileTree() returns a FileTree instance that applies some default exclusion patterns for convenience — the same defaults as Ant in fact. For the complete default exclusion list, see [the Ant manual](http://ant.apache.org/manual/dirtasks.html#defaultexcludes).

If those default exclusions prove problematic, you can workaround the issue by using the [defaultexcludes Ant task](https://ant.apache.org/manual/Tasks/defaultexcludes.html), as demonstrated in this example:

[Example: Changing Ant default exclusions for a copy task](#37pzblu)

**build.gradle**

task forcedCopy(type: Copy) {  
 into "${buildDir}/inPlaceApp"  
 from 'src/main/webapp'  
  
 doFirst {  
 ant.defaultexcludes remove: "\*\*/.git"  
 ant.defaultexcludes remove: "\*\*/.git/\*\*"  
 ant.defaultexcludes remove: "\*\*/\*~"  
 }  
  
 doLast {  
 ant.defaultexcludes **default**: true  
 }  
}

In general, it’s best to ensure that the default exclusions are reset whenever you change them as modifications are visible to the entire build. The above example is performing such a reset in its doLast action.

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)](http://docs.google.com/javadoc/org/gradle/api/file/FileTree.html#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)](http://docs.google.com/javadoc/org/gradle/api/file/FileTree.html#visit-org.gradle.api.Action-) method. All of these techniques are demonstrated in the following example:

[Example: Using a file tree](#1mv9ltn)

**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"  
}

We’ve discussed how to create your own file trees and file collections, but it’s also worth bearing in mind that many Gradle plugins provide their own instances of file trees, such as [Java’s source sets](#3lbifu6). These can be used and manipulated in exactly the same way as the file trees you create yourself.

Another specific type of file tree that users commonly need is the archive, i.e. ZIP files, TAR files, etc. We look at those next.

[Using archives as file trees](#46ux4hg)

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)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:zipTree(java.lang.Object)) and [Project.tarTree(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.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:

[Example: Using an archive as a file tree](#2m07ep9)

**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 among the common scenarios](#1cm5vqp) we cover.

[Understanding implicit conversion to file collections](#115hox2)

Many objects in Gradle have properties which accept a set of input files. For example, the [JavaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.JavaCompile.html) 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()](#4f5j5f0) 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.

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 link:../javadoc/org/gradle/api/file/ProjectLayout.html#files-java.lang.Object...-[ProjectLayout.files(java.lang.Object...) in your build.

Here are some examples of the different types of arguments that the source property can take:

[Example: Specifying a set of files](#3l557kv)

**build.gradle**

task compile(type: JavaCompile)  
  
*// Use a File object to specify the source directory*  
compile {  
 source = file('src/main/java')  
}  
  
*// Use a String path to specify the source directory*  
compile {  
 source = 'src/main/java'  
}  
  
*// Use a collection to specify multiple source directories*  
compile {  
 source = ['src/main/java', '../shared/java']  
}  
  
*// Use a FileCollection (or FileTree in this case) to specify the source files*  
compile {  
 source = fileTree(dir: 'src/main/java').matching { include 'org/gradle/api/\*\*' }  
}  
  
*// Using a closure to specify the source files.*  
compile {  
 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()](#4f5j5f0) method, as shown here:

[Example: Appending a set of files](#20afhso)

**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.

[File copying in depth](#u2g4q7)

The basic process of copying files in Gradle is a simple one:

* Define a task of type [Copy](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html)
* 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 form what is known as a *copy specification*. This is formally based on the [CopySpec](http://docs.google.com/javadoc/org/gradle/api/file/CopySpec.html) interface, which the Copy task implements, and offers:

* A [CopySpec.from(java.lang.Object…​)](http://docs.google.com/javadoc/org/gradle/api/file/CopySpec.html#from-java.lang.Object...-) method to define what to copy
* An [CopySpec.into(java.lang.Object)](http://docs.google.com/javadoc/org/gradle/api/file/CopySpec.html#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)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.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](#4hae2tp) are included

In fact, from() accepts all the same arguments as [Project.files(java.lang.Object…​)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:files(java.lang.Object%5B%5D)), [ProjectLayout.files(java.lang.Object…​)](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#files-java.lang.Object...-), and [ProjectLayout.configurableFiles(java.lang.Object…​)](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#configurableFiles-java.lang.Object...-), so see those methods 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 — a technique that also works with from():

[Example: Specifying copy task source files and destination directory](#4ka30gh)

**build.gradle**

task anotherCopyTask(type: 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](#z6zzro), even though the syntax is similar. Keep an eye on the number of arguments to distinguish between them.

[Filtering files](#2zfdaoa)

You’ve already seen that you can filter file collections and file trees directly in a Copy task, but you can also apply filtering in any copy specification through the [CopySpec.include(java.lang.String…​)](http://docs.google.com/javadoc/org/gradle/api/file/CopySpec.html#include-java.lang.String...-) and [CopySpec.exclude(java.lang.String…​)](http://docs.google.com/javadoc/org/gradle/api/file/CopySpec.html#exclude-java.lang.String...-) methods.

Both of these methods are normally used with Ant-style include or exclude patterns, as described in [PatternFilterable](http://docs.google.com/javadoc/org/gradle/api/tasks/util/PatternFilterable.html). You can also perform more complex logic by using a closure that takes a [FileTreeElement](http://docs.google.com/javadoc/org/gradle/api/file/FileTreeElement.html) 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:

[Example: Selecting the files to copy](#1eknkw3)

**build.gradle**

task copyTaskWithPatterns(type: Copy) {  
 from 'src/main/webapp'  
 into "${buildDir}/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](#z6zzro).

[Renaming files](#3ykb3jw)

The [example of how to rename files on copy](#14bjut5) gives you most of the information you need to perform this operation. It demonstrates the two options for renaming:

* Using a regular expression
* Using a closure

Regular expressions are a flexible approach to renaming, particularly as Gradle supports regex groups that allow you to remove and replaces 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:

[Example: Renaming files as they are copied](#2dpldrp)

**build.gradle**

task rename(type: Copy) {  
 from 'src/main/webapp'  
 into "${buildDir}/explodedWar"  
 *// Use a closure to convert all file names to upper case*  
 rename { String fileName ->  
 fileName.toUpperCase()  
 }  
 *// Use a regular expression to map the file name*  
 rename '(.+)-staging-(.+)', '$1$2'  
 rename(/(.+)-staging-(.+)/, '$1$2')  
}

You can use any regular expression supported by the Java [Pattern](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html) class and the substitution string (the second argument of rename() works on the same principles as the [Matcher.appendReplacement()](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html#appendReplacement(java.lang.StringBuffer,%20java.lang.String)) method.

| **Note** | 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 the name of a file and you return a new name for that file, or null if you don’t want to change the name. Do be aware that the closure will be executed for every file that’s copied, so try to avoid expensive operations where possible.

[Filtering file content (token substitution, templating, etc.)](#suvnzi)

Not to be confused with filtering which files are copied, *file content filtering* allows you to transform the content of files while they are being copied. This can involve 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)](http://docs.google.com/javadoc/org/gradle/api/file/CopySpec.html#expand-java.util.Map-) method and another using [CopySpec.filter(java.lang.Class)](http://docs.google.com/javadoc/org/gradle/api/file/CopySpec.html#filter-java.lang.Class-) with an [Ant filter](https://ant.apache.org/manual/Types/filterchain.html):

[Example: Filtering files as they are copied](#3cuj6nb)

**build.gradle**

**import** org.apache.tools.ant.filters.FixCrLfFilter  
**import** org.apache.tools.ant.filters.ReplaceTokens  
  
task filter(type: Copy) {  
 from 'src/main/webapp'  
 into "${buildDir}/explodedWar"  
 *// Substitute property tokens in files*  
 expand(copyright: '2009', version: '2.3.1')  
 expand(project.properties)  
 *// Use some of the filters provided by Ant*  
 filter(FixCrLfFilter)  
 filter(ReplaceTokens, tokens: [copyright: '2009', version: '2.3.1'])  
 *// Use a closure to filter each line*  
 filter { String line ->  
 "[$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](https://docs.oracle.com/javase/7/docs/api/java/io/FilterReader.html) and is designed to work with Ant filters, such as ReplaceTokens
* one takes a closure or [Transformer](http://docs.google.com/javadoc/org/gradle/api/Transformer.html) 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(), the result is a template engine that replaces tokens of the form @tokenName@ (the Ant-style token) with values that you define.

The expand() method treats the source files as [Groovy templates](http://docs.groovy-lang.org/latest/html/api/groovy/text/SimpleTemplateEngine.html), which evaluate and expand 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** | It’s good practice to specify the character set when reading and writing the file, otherwise the transformations won’t work properly for non-ASCII text. You configure the character set with the [CopySpec.getFilteringCharset()](http://docs.google.com/javadoc/org/gradle/api/file/CopySpec.html#getFilteringCharset--) property. If it’s not specified, the JVM default character set is used, which is likely to be different from the one you want. |
| --- | --- |

[Using the CopySpec class](#1rztgv4)

A copy specification (or copy spec for short) determines what gets copied to where, and what happens to files during the copy. You’ve alread seen many examples in the form of configuration for Copy and archiving tasks. But copy specs have two attributes that are worth covering in more detail:

1. They can be independent of tasks
2. They are hierarchical

The first of these attributes allows you to *share copy specs within a build*. The second provides fine-grained control within the overall copy specification.

[Sharing copy specs](#4bzgzix)

Consider a build that has 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 Gradle provides is the [Project.copySpec(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:copySpec(org.gradle.api.Action)) method. This allows you to create a copy spec outside of a task, which can then be attached to an appropriate task using the [CopySpec.with(org.gradle.api.file.CopySpec…​)](http://docs.google.com/javadoc/org/gradle/api/file/CopySpec.html#with-org.gradle.api.file.CopySpec...-) method. The following example demonstrates how this is done:

[Example: Sharing copy specifications](#2r4r9qq)

**build.gradle**

CopySpec webAssetsSpec = copySpec {  
 from 'src/main/webapp'  
 include '\*\*/\*.html', '\*\*/\*.png', '\*\*/\*.jpg'  
 rename '(.+)-staging(.+)', '$1$2'  
}  
  
task copyAssets(type: Copy) {  
 into "${buildDir}/inPlaceApp"  
 with webAssetsSpec  
}  
  
task distApp(type: Zip) {  
 archiveName = 'my-app-dist.zip'  
 destinationDir = file("${buildDir}/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 to understand, 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. |
| --- | --- |

If you encounter a scenario in which you want to apply the same copy configuration to *different* sets of files, then 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:

[Example: Sharing copy patterns only](#16a1jyj)

**build.gradle**

**def** webAssetPatterns = {  
 include '\*\*/\*.html', '\*\*/\*.png', '\*\*/\*.jpg'  
}  
  
task copyAppAssets(type: Copy) {  
 into "${buildDir}/inPlaceApp"  
 from 'src/main/webapp', webAssetPatterns  
}  
  
task archiveDistAssets(type: Zip) {  
 archiveName = 'distribution-assets.zip'  
 destinationDir = file("${buildDir}/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.

[Using child specifications](#z6zzro)

If you only use a single copy spec, the file filtering and renaming will apply to *all* the files that are copied. Sometimes this is what you want, but not always. Consider the following example that copies files into a directory structure that can be used by a Java Servlet container to deliver a website:



*Figure 11. Creating an exploded WAR for a Servlet container*

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. So 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:

[Example: Nested copy specs](#3q9p2mc)

**build.gradle**

task nestedSpecs(type: Copy) {  
 into "${buildDir}/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: that’s 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 may want to 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 simply 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.

[Copying files in your own tasks](#25ezcu5)

There might be occasions when 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 then archived. You still want to take advantage of Gradle’s copy API, but without introducing an extra Copy task.

The solution is to use the [Project.copy(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:copy(org.gradle.api.Action)) method. It works the same way as the Copy task by configuring it with a copy spec. Here’s a trivial example:

[Example: Copying files using the copy() method without up-to-date check](#kk9n1y)

**build.gradle**

task copyMethod {  
 doLast {  
 copy {  
 from 'src/main/webapp'  
 into "${buildDir}/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](#odc9jc). 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 manually define the task inputs and outputs.
2. Using a task as a copy source, i.e. as an argument to from(), won’t set up an automatic task dependency between your task and that copy source. As such, if you are using the copy() method as part of a task action, you must explicitly declare all inputs and outputs in order to get the correct behavior.

The following example shows you how to workaround these limitations by using the [dynamic API for task inputs and outputs](#3gh8kve):

[Example: Copying files using the copy() method with up-to-date check](#34jx5pr)

**build.gradle**

task copyMethodWithExplicitDependencies{  
 *// up-to-date check for inputs, plus add copyTask as dependency*  
 inputs.files copyTask  
 outputs.dir 'some-dir' *// up-to-date check for outputs*  
 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 builtin support for incremental building and task dependency inference. That is why the copy() method is intended for use by [custom tasks](#2grqrue) 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.

[Mirroring directories and file collections with the Sync task](#1jp7fxk)

The [Sync](http://docs.google.com/dsl/org.gradle.api.tasks.Sync.html) 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. In other words, 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 which maintains a copy of the project’s runtime dependencies in the build/libs directory.

[Example: Using the Sync task to copy dependencies](#43ouyld)

**build.gradle**

task libs(type: Sync) {  
 from configurations.runtime  
 into "${buildDir}/libs"  
}

You can also perform the same function in your own tasks with the [Project.sync(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:sync(org.gradle.api.Action)) method.

[Archive creation in depth](#2iu58t6)

Archives are essentially self-contained file systems and Gradle treats them as such. This is why working with archives is very similar to working with files and directories, including such things as file permissions.

Out of the box, Gradle supports creation of both 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](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Zip.html), [Tar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Tar.html), [Jar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Jar.html), [War](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.War.html), and [Ear](http://docs.google.com/dsl/org.gradle.plugins.ear.Ear.html). 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’s a basic example that specifies the path and name of the target archive file:

[Example: Archiving a directory as a ZIP](#xzfj0z)

**build.gradle**

task packageDistribution(type: Zip) {  
 archiveName = "my-distribution.zip"  
 destinationDir = file("${buildDir}/dist")  
  
 from "${buildDir}/toArchive"  
}

In the next section you’ll learn about convention-based archive names, which can save you from always configuring the destination directory and archive name.

The full power of copy specifications are available to you when creating archives, which means you can do content filtering, file renaming or anything else that is covered in the previous section. A particularly 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](#z6zzro).

Gradle does of course allow you create as many archive tasks as you want, but it’s worth bearing in mind 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 as well as the copy specifications used. We recommend you use these tasks wherever you can, rather than overriding them with your own.

[Archive naming](#4iblbb3)

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](#vx1227), which defaults to creating archives in the $buildDir/distributions directory and typically uses archive names of the form *[projectName]-[version].[type]*.

The following example comes from a project named 'zipProject', hence the myZip task creates an archive named 'zipProject-1.0.zip':

[Example: Creation of ZIP archive](#3hz31os)

**build.gradle**

plugins {  
 id 'base'  
}  
  
version = 1.0  
  
task myZip(type: Zip) {  
 from 'somedir'  
  
 doLast {  
 println archiveName  
 println relativePath(destinationDir)  
 println relativePath(archivePath)  
 }  
}

**Output of** gradle -q myZip

> gradle -q myZip  
zipProject-1.0.zip  
build/distributions  
build/distributions/zipProject-1.0.zip

Note that the name of the archive does *not* derive from the name of the task that creates it.

If you want to change the name and location of a generated archive file, you can provide values for the archiveName and destinationDir properties of the corresponding task. These override any conventions that would otherwise apply.

Alternatively, you can make use of the default archive name pattern provided by [AbstractArchiveTask.getArchiveName()](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.AbstractArchiveTask.html#org.gradle.api.tasks.bundling.AbstractArchiveTask:archiveName): *[baseName]-[appendix]-[version]-[classifier].[extension]*. You can set each of these properties on the task separately if you wish. Note that the Base Plugin uses the convention of project name for *baseName*, project version for *version* and the archive type for *extension*. It does not provide values for the other properties.

This example — from the same project as the one above — configures just the baseName property, overriding the default value of the project name:

[Example: Configuration of archive task - custom archive name](#1x4dbwl)

**build.gradle**

task myCustomZip(type: Zip) {  
 baseName = 'customName'  
 from 'somedir'  
  
 doLast {  
 println archiveName  
 }  
}

**Output of** gradle -q myCustomZip

> gradle -q myCustomZip  
customName-1.0.zip

You can also override the default baseName value for *all* the archive tasks in your build by using the *project* property archivesBaseName, as demonstrated by the following example:

[Example: Configuration of archive task - appendix & classifier](#4h40uke)

**build.gradle**

plugins {  
 id 'base'  
}  
  
version = 1.0  
archivesBaseName = "gradle"  
  
task myZip(type: Zip) {  
 from 'somedir'  
}  
  
task myOtherZip(type: Zip) {  
 appendix = 'wrapper'  
 classifier = 'src'  
 from 'somedir'  
}  
  
task echoNames {  
 doLast {  
 println "Project name: ${project.name}"  
 println myZip.archiveName  
 println myOtherZip.archiveName  
 }  
}

**Output of** gradle -q echoNames

> gradle -q echoNames  
Project name: zipProject  
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](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.AbstractArchiveTask.html), but we have also summarized the main ones here:

archiveName — String, default: *baseName*-*appendix*-*version*-*classifier*.*extension*

The complete file name of the generated archive. If any of the properties in the default value are empty, their '-' separator is dropped.

archivePath — File, *read-only*, default: *destinationDir*/*archiveName*

The absolute file path of the generated archive.

destinationDir — File, default: depends on archive type

The target directory in which to put the generated archive. By default, JARs and WARs go into $buildDir/libs. ZIPs and TARs go into $buildDir/distributions.

baseName — 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.

appendix — 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.

version — String, default: *project.version*

The version portion of the archive file name, typically in the form of a normal project or product version.

classifier — String, default: null

The classifier portion of the archive file name. Often used to distinguish between archives that target different platforms.

extension — 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](#2w9b4s7)

[As described earlier](#4bzgzix), you can use the [Project.copySpec(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:copySpec(org.gradle.api.Action)) method to share content between archives.

[Reproducible archives](#1belf00)

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](https://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](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.AbstractArchiveTask.html) (e.g. Jar, Zip) tasks shipped with Gradle include [incubating](#2xcytpi) support producing reproducible archives.

For example, to make a Zip task reproducible you need to set [Zip.isReproducibleFileOrder()](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Zip.html#org.gradle.api.tasks.bundling.Zip:reproducibleFileOrder) to true and [Zip.isPreserveFileTimestamps()](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Zip.html#org.gradle.api.tasks.bundling.Zip:preserveFileTimestamps) to false. In order to make all archive tasks in your build reproducible, consider adding the following configuration to your build file:

[Example: Activating reproducible archives](#3ve8xnt)

**build.gradle**

tasks.withType(AbstractArchiveTask) {  
 preserveFileTimestamps = false  
 reproducibleFileOrder = true  
}

Often you will want to publish an archive, so that it is usable from another project. This process is described in [Legacy Publishing](#2dlolyb).

[Writing Build Scripts](#41mghml)

This chapter looks at some of the details of writing a build script.

[The Gradle build language](#2ajj7vm)

Gradle provides a *domain specific language*, or DSL, for describing builds. This build language is based on Groovy, with some additions to make it easier to describe a build.

A build script can contain any Groovy language element.[[7](#poti3f)] Gradle assumes that each build script is encoded using UTF-8.

[The Project API](#39oh0r8)

In [the tutorial](#3hv69ve) we used, for example, the apply() method. Where does this method come from? We said earlier that the build script defines a project in Gradle. For each project in the build, Gradle creates an object of type [Project](http://docs.google.com/dsl/org.gradle.api.Project.html) and associates this Project object with the build script. As the build script executes, it configures this Project object:

| **Tip** | Getting help writing build scripts  Don’t forget that your build script is simply Groovy code that drives the Gradle API. And the [Project](http://docs.google.com/dsl/org.gradle.api.Project.html) interface is your starting point for accessing everything in the Gradle API. So, if you’re wondering what 'tags' are available in your build script, you can start with the documentation for the Project interface. |
| --- | --- |

* Any method you call in your build script which *is not defined* in the build script, is delegated to the Project object.
* Any property you access in your build script, which *is not defined* in the build script, is delegated to the Project object.

Let’s try this out and try to access the name property of the Project object.

[Example: Accessing property of the Project object](#1otraz1)

**build.gradle**

println name  
println project.name

**Output of** gradle -q check

> gradle -q check  
projectApi  
projectApi

Both println statements print out the same property. The first uses auto-delegation to the Project object, for properties not defined in the build script. The other statement uses the project property available to any build script, which returns the associated Project object. Only if you define a property or a method which has the same name as a member of the Project object, would you need to use the project property.

[Standard project properties](#48tetmu)

The Project object provides some standard properties, which are available in your build script. The following table lists a few of the commonly used ones.

Table 5. Project Properties

| **Name** | **Type** | **Default Value** |
| --- | --- | --- |
| project | [Project](http://docs.google.com/dsl/org.gradle.api.Project.html) | The Project instance |
| name | String | The name of the project directory. |
| path | String | The absolute path of the project. |
| description | String | A description for the project. |
| projectDir | File | The directory containing the build script. |
| buildDir | File | *projectDir*/build |
| group | Object | unspecified |
| version | Object | unspecified |
| ant | [AntBuilder](http://docs.google.com/javadoc/org/gradle/api/AntBuilder.html) | An AntBuilder instance |

[The Script API](#2nyp3un)

When Gradle executes a script, it compiles the script into a class which implements [Script](http://docs.google.com/dsl/org.gradle.api.Script.html). This means that all of the properties and methods declared by the Script interface are available in your script.

[Declaring variables](#133ze2g)

There are two kinds of variables that can be declared in a build script: local variables and extra properties.

[Local variables](#3n3mwq9)

Local variables are declared with the def keyword. They are only visible in the scope where they have been declared. Local variables are a feature of the underlying Groovy language.

[Example: Using local variables](#228x6y2)

**build.gradle**

**def** dest = "dest"  
  
task copy(type: Copy) {  
 from "source"  
 into dest  
}

[Extra properties](#2emvufp)

All enhanced objects in Gradle’s domain model can hold extra user-defined properties. This includes, but is not limited to, projects, tasks, and source sets. Extra properties can be added, read and set via the owning object’s ext property. Alternatively, an ext block can be used to add multiple properties at once.

[Example: Using extra properties](#he7h5v)

**build.gradle**

apply plugin: "java"  
  
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"  
 }  
}  
  
task printProperties {  
 doLast {  
 println springVersion  
 println emailNotification  
 sourceSets.matching { it.purpose == "production" }.each { println it.name }  
 }  
}

**Output of** gradle -q printProperties

> gradle -q printProperties  
3.1.0.RELEASE  
build@master.org  
main  
plugin

In this example, an ext block adds two extra properties to the project object. Additionally, a property named purpose is added to each source set by setting ext.purpose to null (null is a permissible value). Once the properties have been added, they can be read and set like predefined properties.

By requiring special syntax for adding a property, Gradle can fail fast when an attempt is made to set a (predefined or extra) property but the property is misspelled or does not exist. Extra properties can be accessed from anywhere their owning object can be accessed, giving them a wider scope than local variables. Extra properties on a project are visible from its subprojects.

For further details on extra properties and their API, see the [ExtraPropertiesExtension](http://docs.google.com/dsl/org.gradle.api.plugins.ExtraPropertiesExtension.html) class in the API documentation.

[Configuring arbitrary objects](#31duzto)

You can configure arbitrary objects in the following very readable way.

[Example: Configuring arbitrary objects](#1gj5a1h)

**build.gradle**

task configure {  
 doLast {  
 **def** pos = configure(**new** java.text.FieldPosition(10)) {  
 beginIndex = 1  
 endIndex = 5  
 }  
 println pos.beginIndex  
 println pos.endIndex  
 }  
}

**Output of** gradle -q configure

> gradle -q configure  
1  
5

[Configuring arbitrary objects using an external script](#40isspa)

You can also configure arbitrary objects using an external script.

[Example: Configuring arbitrary objects using a script](#2fo32x3)

**build.gradle**

task configure {  
 doLast {  
 **def** pos = **new** java.text.FieldPosition(10)  
 *// Apply the script*  
 apply from: 'other.gradle', to: pos  
 println pos.beginIndex  
 println pos.endIndex  
 }  
}

**other.gradle**

*// Set properties.*  
beginIndex = 1  
endIndex = 5

**Output of gradle -q configure**

> gradle -q configure  
1  
5

[Some Groovy basics](#utdd4w)

The [Groovy language](http://docs.groovy-lang.org/latest/html/documentation/index.html) provides plenty of features for creating DSLs, and the Gradle build language takes advantage of these. Understanding how the build language works will help you when you write your build script, and in particular, when you start to write custom plugins and tasks.

[Groovy JDK](#3et0vsp)

Groovy adds lots of useful methods to the standard Java classes. For example, Iterable gets an each method, which iterates over the elements of the Iterable:

[Example: Groovy JDK methods](#1tyb60i)

**build.gradle**

*// Iterable gets an each() method*  
configurations.runtime.each { File f -> println f }

Have a look at <http://groovy-lang.org/gdk.html> for more details.

[Property accessors](#4dxyoob)

Groovy automatically converts a property reference into a call to the appropriate getter or setter method.

[Example: Property accessors](#2t38yw4)

**build.gradle**

*// Using a getter method*  
println project.buildDir  
println getProject().getBuildDir()  
  
*// Using a setter method*  
project.buildDir = 'target'  
getProject().setBuildDir('target')

[Optional parentheses on method calls](#188j93x)

Parentheses are optional for method calls.

[Example: Method call without parentheses](#3s86rrq)

**build.gradle**

test.systemProperty 'some.prop', 'value'  
test.systemProperty('some.prop', 'value')

[List and map literals](#27dh1zj)

Groovy provides some shortcuts for defining List and Map instances. Both kinds of literals are straightforward, but map literals have some interesting twists.

For instance, the “apply” method (where you typically apply plugins) actually takes a map parameter. However, when you have a line like “apply plugin:'java'”, you aren’t actually using a map literal, you’re actually using “named parameters”, which have almost exactly the same syntax as a map literal (without the wrapping brackets). That named parameter list gets converted to a map when the method is called, but it doesn’t start out as a map.

[Example: List and map literals](#mirc7c)

**build.gradle**

*// List literal*  
test.includes = ['org/gradle/api/\*\*', 'org/gradle/internal/\*\*']  
  
List<String> list = **new** ArrayList<String>()  
list.add('org/gradle/api/\*\*')  
list.add('org/gradle/internal/\*\*')  
test.includes = list  
  
*// Map literal.*  
Map<String, String> map = [key1:'value1', key2: 'value2']  
  
*// Groovy will coerce named arguments*  
*// into a single map argument*  
apply plugin: 'java'

[Closures as the last parameter in a method](#36ieuv5)

The Gradle DSL uses closures in many places. You can find out more about closures [here](http://docs.groovy-lang.org/latest/html/documentation/index.html#_closures). When the last parameter of a method is a closure, you can place the closure after the method call:

[Example: Closure as method parameter](#1lnp52y)

**build.gradle**

repositories {  
 println "in a closure"  
}  
repositories() { println "in a closure" }  
repositories({ println "in a closure" })

[Closure delegate](#45ncnqr)

Each closure has a delegate object, which Groovy uses to look up variable and method references which are not local variables or parameters of the closure. Gradle uses this for *configuration closures*, where the delegate object is set to the object to be configured.

[Example: Closure delegates](#2ksmxyk)

**build.gradle**

dependencies {  
 **assert** delegate == project.dependencies  
 testCompile('junit:junit:4.12')  
 delegate.testCompile('junit:junit:4.12')  
}

[Default imports](#gna8r6)

To make build scripts more concise, Gradle automatically adds a set of import statements to the Gradle scripts. This means that instead of using throw new org.gradle.api.tasks.StopExecutionException() you can just type throw new StopExecutionException() instead.

Listed below are the imports added to each script:

**Gradle default imports**

**import** org.gradle.\*  
**import** org.gradle.api.\*  
**import** org.gradle.api.artifacts.\*  
**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.attributes.\*  
**import** org.gradle.api.capabilities.\*  
**import** org.gradle.api.component.\*  
**import** org.gradle.api.credentials.\*  
**import** org.gradle.api.distribution.\*  
**import** org.gradle.api.distribution.plugins.\*  
**import** org.gradle.api.dsl.\*  
**import** org.gradle.api.execution.\*  
**import** org.gradle.api.file.\*  
**import** org.gradle.api.initialization.\*  
**import** org.gradle.api.initialization.definition.\*  
**import** org.gradle.api.initialization.dsl.\*  
**import** org.gradle.api.invocation.\*  
**import** org.gradle.api.java.archives.\*  
**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.announce.\*  
**import** org.gradle.api.plugins.antlr.\*  
**import** org.gradle.api.plugins.buildcomparison.gradle.\*  
**import** org.gradle.api.plugins.osgi.\*  
**import** org.gradle.api.plugins.quality.\*  
**import** org.gradle.api.plugins.scala.\*  
**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.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.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.authentication.\*  
**import** org.gradle.authentication.aws.\*  
**import** org.gradle.authentication.http.\*  
**import** org.gradle.buildinit.plugins.\*  
**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.platform.\*  
**import** org.gradle.jvm.plugins.\*  
**import** org.gradle.jvm.tasks.\*  
**import** org.gradle.jvm.tasks.api.\*  
**import** org.gradle.jvm.test.\*  
**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.coffeescript.\*  
**import** org.gradle.language.cpp.\*  
**import** org.gradle.language.cpp.plugins.\*  
**import** org.gradle.language.cpp.tasks.\*  
**import** org.gradle.language.java.\*  
**import** org.gradle.language.java.artifact.\*  
**import** org.gradle.language.java.plugins.\*  
**import** org.gradle.language.java.tasks.\*  
**import** org.gradle.language.javascript.\*  
**import** org.gradle.language.jvm.\*  
**import** org.gradle.language.jvm.plugins.\*  
**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.objectivecpp.\*  
**import** org.gradle.language.objectivecpp.plugins.\*  
**import** org.gradle.language.objectivecpp.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.routes.\*  
**import** org.gradle.language.scala.\*  
**import** org.gradle.language.scala.plugins.\*  
**import** org.gradle.language.scala.tasks.\*  
**import** org.gradle.language.scala.toolchain.\*  
**import** org.gradle.language.swift.\*  
**import** org.gradle.language.swift.plugins.\*  
**import** org.gradle.language.swift.tasks.\*  
**import** org.gradle.language.twirl.\*  
**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.base.\*  
**import** org.gradle.platform.base.binary.\*  
**import** org.gradle.platform.base.component.\*  
**import** org.gradle.platform.base.plugins.\*  
**import** org.gradle.play.\*  
**import** org.gradle.play.distribution.\*  
**import** org.gradle.play.platform.\*  
**import** org.gradle.play.plugins.\*  
**import** org.gradle.play.plugins.ide.\*  
**import** org.gradle.play.tasks.\*  
**import** org.gradle.play.toolchain.\*  
**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.javascript.base.\*  
**import** org.gradle.plugins.javascript.coffeescript.\*  
**import** org.gradle.plugins.javascript.envjs.\*  
**import** org.gradle.plugins.javascript.envjs.browser.\*  
**import** org.gradle.plugins.javascript.envjs.http.\*  
**import** org.gradle.plugins.javascript.envjs.http.simple.\*  
**import** org.gradle.plugins.javascript.jshint.\*  
**import** org.gradle.plugins.javascript.rhino.\*  
**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.vcs.\*  
**import** org.gradle.vcs.git.\*  
**import** org.gradle.workers.\*

[Writing Custom Task Classes](#2grqrue)

Gradle supports two types of task. One such type is the simple task, where you define the task with an action closure. We have seen these in [Build Script Basics](#1pxezwc). For this type of task, the action closure determines the behaviour of the task. This type of task is good for implementing one-off tasks in your build script.

The other type of task is the enhanced task, where the behaviour is built into the task, and the task provides some properties which you can use to configure the behaviour. We have seen these in [Authoring Tasks](#147n2zr). Most Gradle plugins use enhanced tasks. With enhanced tasks, you don’t need to implement the task behaviour as you do with simple tasks. You simply declare the task and configure the task using its properties. In this way, enhanced tasks let you reuse a piece of behaviour in many different places, possibly across different builds.

The behaviour and properties of an enhanced task is defined by the task’s class. When you declare an enhanced task, you specify the type, or class of the task.

Implementing your own custom task class in Gradle is easy. You can implement a custom task class in pretty much any language you like, provided it ends up compiled to bytecode. In our examples, we are going to use Groovy as the implementation language. Groovy, Java or Kotlin are all good choices as the language to use to implement a task class, as the Gradle API has been designed to work well with these languages. In general, a task implemented using Java or Kotlin, which are statically typed, will perform better than the same task implemented using Groovy.

[Packaging a task class](#zxx86d)

There are several places where you can put the source for the task class.

Build script

You can include the task class directly in the build script. This has the benefit that the task class is automatically compiled and included in the classpath of the build script without you having to do anything. However, the task class is not visible outside the build script, and so you cannot reuse the task class outside the build script it is defined in.

buildSrc project

You can put the source for the task class in the *rootProjectDir*/buildSrc/src/main/groovy directory. Gradle will take care of compiling and testing the task class and making it available on the classpath of the build script. The task class is visible to every build script used by the build. However, it is not visible outside the build, and so you cannot reuse the task class outside the build it is defined in. Using the buildSrc project approach separates the task declaration - that is, what the task should do - from the task implementation - that is, how the task does it.

See [Organizing Gradle Projects](#3whwml4) for more details about the buildSrc project.

Standalone project

You can create a separate project for your task class. This project produces and publishes a JAR which you can then use in multiple builds and share with others. Generally, this JAR might include some custom plugins, or bundle several related task classes into a single library. Or some combination of the two.

In our examples, we will start with the task class in the build script, to keep things simple. Then we will look at creating a standalone project.

[Writing a simple task class](#3jxkqu6)

To implement a custom task class, you extend [DefaultTask](http://docs.google.com/dsl/org.gradle.api.DefaultTask.html).

[Example: Defining a custom task](#1z2v11z)

**build.gradle**

**class** **GreetingTask** **extends** DefaultTask {  
}

This task doesn’t do anything useful, so let’s add some behaviour. To do so, we add a method to the task and mark it with the [TaskAction](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskAction.html) annotation. Gradle will call the method when the task executes. You don’t have to use a method to define the behaviour for the task. You could, for instance, call doFirst() or doLast() with a closure in the task constructor to add behaviour.

[Example: A hello world task](#4j2ijps)

**build.gradle**

**class** **GreetingTask** **extends** DefaultTask {  
 @TaskAction  
 **def** **greet**() {  
 println 'hello from GreetingTask'  
 }  
}  
  
*// Create a task using the task type*  
task hello(type: GreetingTask)

**Output of** gradle -q hello

> gradle -q hello  
hello from GreetingTask

Let’s add a property to the task, so we can customize it. Tasks are simply POGOs, and when you declare a task, you can set the properties or call methods on the task object. Here we add a greeting property, and set the value when we declare the greeting task.

[Example: A customizable hello world task](#2y7stxl)

**build.gradle**

**class** **GreetingTask** **extends** DefaultTask {  
 String greeting = 'hello from GreetingTask'  
  
 @TaskAction  
 **def** **greet**() {  
 println greeting  
 }  
}  
  
*// Use the default greeting*  
task hello(type: GreetingTask)  
  
*// Customize the greeting*  
task greeting(type: GreetingTask) {  
 greeting = 'greetings from GreetingTask'  
}

**Output of** gradle -q hello greeting

> gradle -q hello greeting  
hello from GreetingTask  
greetings from GreetingTask

[A standalone project](#1dd345e)

Now we will move our task to a standalone project, so we can publish it and share it with others. This project is simply a Groovy project that produces a JAR containing the task class. Here is a simple build script for the project. It applies the Groovy plugin, and adds the Gradle API as a compile-time dependency.

[Example: A build for a custom task](#3xcqmt7)

**build.gradle**

plugins {  
 id 'groovy'  
}  
  
dependencies {  
 compile gradleApi()  
 compile localGroovy()  
}

| **Note** | The code for this example can be found at samples/customPlugin/plugin in the ‘-all’ distribution of Gradle. |
| --- | --- |

We just follow the convention for where the source for the task class should go.

[Example: A custom task](#2ci0x10)

**src/main/groovy/org/gradle/GreetingTask.groovy**

**package** org.gradle  
  
**import** org.gradle.api.DefaultTask  
**import** org.gradle.api.tasks.TaskAction  
  
**class** **GreetingTask** **extends** DefaultTask {  
 String greeting = 'hello from GreetingTask'  
  
 @TaskAction  
 **def** **greet**() {  
 println greeting  
 }  
}

[Using your task class in another project](#rnb78t)

To use a task class in a build script, you need to add the class to the build script’s classpath. To do this, you use a buildscript { } block, as described in [External dependencies for the build script](#3utoxif). The following example shows how you might do this when the JAR containing the task class has been published to a local repository:

[Example: Using a custom task in another project](#3bmypwm)

**build.gradle**

buildscript {  
 repositories {  
 maven {  
*// END SNIPPET use-plugin*  
*// END SNIPPET use-task*  
 **def** producerName = findProperty('producerName') ?: 'plugin'  
 **def** repoLocation = "../$producerName/build/repo"  
*// START SNIPPET use-plugin*  
*// START SNIPPET use-task*  
 url = uri(repoLocation)  
 }  
 }  
 dependencies {  
 classpath group: 'org.gradle', name: 'customPlugin',  
 version: '1.0-SNAPSHOT'  
 }  
}  
  
task greeting(type: org.gradle.GreetingTask) {  
 greeting = 'howdy!'  
}

[Writing tests for your task class](#1qs904f)

You can use the [ProjectBuilder](http://docs.google.com/javadoc/org/gradle/testfixtures/ProjectBuilder.html) class to create [Project](http://docs.google.com/dsl/org.gradle.api.Project.html) instances to use when you test your task class.

[Example: Testing a custom task](#4arwis8)

**src/test/groovy/org/gradle/GreetingTaskTest.groovy**

**class** **GreetingTaskTest** {  
 @Test  
 **public** **void** canAddTaskToProject() {  
 Project project = ProjectBuilder.builder().build()  
 **def** task = project.task('greeting', type: GreetingTask)  
 assertTrue(task **instanceof** GreetingTask)  
 }  
}

[Incremental tasks](#j2f68k)

| **Note** | Incremental tasks are an [incubating](#2xcytpi) feature.  Since the introduction of the implementation described above (early in the Gradle 1.6 release cycle), discussions within the Gradle community have produced superior ideas for exposing the information about changes to task implementors to what is described below. As such, the API for this feature will almost certainly change in upcoming releases. However, please do experiment with the current implementation and share your experiences with the Gradle community.  The feature incubation process, which is part of the Gradle feature lifecycle (see [Feature Lifecycle](#2xcytpi)), exists for this purpose of ensuring high quality final implementations through incorporation of early user feedback. |
| --- | --- |

With Gradle, it’s very simple to implement a task that is skipped when all of its inputs and outputs are up to date (see [Incremental Builds](#odc9jc)). However, there are times when only a few input files have changed since the last execution, and you’d like to avoid reprocessing all of the unchanged inputs. This can be particularly useful for a transformer task, that converts input files to output files on a 1:1 basis.

If you’d like to optimise your build so that only out-of-date inputs are processed, you can do so with an *incremental task*.

[Implementing an incremental task](#2px6t01)

For a task to process inputs incrementally, that task must contain an *incremental task action*. This is a task action method that contains a single [IncrementalTaskInputs](http://docs.google.com/javadoc/org/gradle/api/tasks/incremental/IncrementalTaskInputs.html) parameter, which indicates to Gradle that the action will process the changed inputs only.

The incremental task action may supply an [IncrementalTaskInputs.outOfDate(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.tasks.incremental.IncrementalTaskInputs.html#org.gradle.api.tasks.incremental.IncrementalTaskInputs:outOfDate(org.gradle.api.Action)) action for processing any input file that is out-of-date, and a [IncrementalTaskInputs.removed(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.tasks.incremental.IncrementalTaskInputs.html#org.gradle.api.tasks.incremental.IncrementalTaskInputs:removed(org.gradle.api.Action)) action that executes for any input file that has been removed since the previous execution.

[Example: Defining an incremental task action](#152h37u)

**build.gradle**

**class** **IncrementalReverseTask** **extends** DefaultTask {  
 @InputDirectory  
 **def** File inputDir  
  
 @OutputDirectory  
 **def** File outputDir  
  
 @Input  
 **def** inputProperty  
  
 @TaskAction  
 **void** execute(IncrementalTaskInputs inputs) {  
 println inputs.incremental ? 'CHANGED inputs considered out of date'  
 : 'ALL inputs considered out of date'  
 **if** (!inputs.incremental)  
 project.delete(outputDir.listFiles())  
  
 inputs.outOfDate { change ->  
 println "out of date: ${change.file.name}"  
 **def** targetFile = **new** File(outputDir, change.file.name)  
 targetFile.text = change.file.text.reverse()  
 }  
  
 inputs.removed { change ->  
 println "removed: ${change.file.name}"  
 **def** targetFile = **new** File(outputDir, change.file.name)  
 targetFile.delete()  
 }  
 }  
}

| **Note** | The code for this example can be found at samples/userguide/tasks/incrementalTask in the ‘-all’ distribution of Gradle. |
| --- | --- |

If for some reason the task is not run incremental, e.g. by running with --rerun-tasks, only the outOfDate action is executed, even if there were deleted input files. You should consider handling this case at the beginning, as is done in the example above.

For a simple transformer task like this, the task action simply needs to generate output files for any out-of-date inputs, and delete output files for any removed inputs.

A task may only contain a single incremental task action.

[Which inputs are considered out of date?](#3p24lvn)

When Gradle has history of a previous task execution, and the only changes to the task execution context since that execution are to input files, then Gradle is able to determine which input files need to be reprocessed by the task. In this case, the [IncrementalTaskInputs.outOfDate(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.tasks.incremental.IncrementalTaskInputs.html#org.gradle.api.tasks.incremental.IncrementalTaskInputs:outOfDate(org.gradle.api.Action)) action will be executed for any input file that was *added* or *modified*, and the [IncrementalTaskInputs.removed(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.tasks.incremental.IncrementalTaskInputs.html#org.gradle.api.tasks.incremental.IncrementalTaskInputs:removed(org.gradle.api.Action)) action will be executed for any *removed* input file.

However, there are many cases where Gradle is unable to determine which input files need to be reprocessed. 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 criteria added to the task returns false.
* An input property has changed since the previous execution.
* One or more output files have changed since the previous execution.

In any of these cases, Gradle will consider all of the input files to be outOfDate. The [IncrementalTaskInputs.outOfDate(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.tasks.incremental.IncrementalTaskInputs.html#org.gradle.api.tasks.incremental.IncrementalTaskInputs:outOfDate(org.gradle.api.Action)) action will be executed for every input file, and the [IncrementalTaskInputs.removed(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.tasks.incremental.IncrementalTaskInputs.html#org.gradle.api.tasks.incremental.IncrementalTaskInputs:removed(org.gradle.api.Action)) action will not be executed at all.

You can check if Gradle was able to determine the incremental changes to input files with [IncrementalTaskInputs.isIncremental()](http://docs.google.com/dsl/org.gradle.api.tasks.incremental.IncrementalTaskInputs.html#org.gradle.api.tasks.incremental.IncrementalTaskInputs:incremental).

[An incremental task in action](#247ew3g)

Given the incremental task implementation [above](#152h37u), we can explore the various change scenarios by example. Note that the various mutation tasks ('updateInputs', 'removeInput', etc) are only present for demonstration purposes: these would not normally be part of your build script.

First, consider the IncrementalReverseTask executed against a set of inputs for the first time. In this case, all inputs will be considered “out of date”:

[Example: Running the incremental task for the first time](#jcp6b9)

**build.gradle**

task incrementalReverse(type: IncrementalReverseTask) {  
 inputDir = file('inputs')  
 outputDir = file("$buildDir/outputs")  
 inputProperty = project.properties['taskInputProperty'] ?: 'original'  
}

**Build layout**

.  
├── build.gradle  
└── inputs  
 ├── 1.txt  
 ├── 2.txt  
 └── 3.txt

**Output of gradle -q incrementalReverse**

> gradle -q incrementalReverse  
ALL inputs considered out of date  
out of date: 1.txt  
out of date: 2.txt  
out of date: 3.txt

Naturally when the task is executed again with no changes, then the entire task is up to date and no files are reported to the task action:

[Example: Running the incremental task with unchanged inputs](#33ccoz2)

**Output of** gradle -q incrementalReverse

> gradle -q incrementalReverse

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 reported to [IncrementalTaskInputs.outOfDate(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.tasks.incremental.IncrementalTaskInputs.html#org.gradle.api.tasks.incremental.IncrementalTaskInputs:outOfDate(org.gradle.api.Action)):

[Example: Running the incremental task with updated input files](#1ihmz6v)

**build.gradle**

task updateInputs() {  
 doLast {  
 file('inputs/1.txt').text = 'Changed content for existing file 1.'  
 file('inputs/4.txt').text = 'Content for new file 4.'  
 }  
}

**Output of** gradle -q updateInputs incrementalReverse

> gradle -q updateInputs incrementalReverse  
CHANGED inputs considered out of date  
out of date: 1.txt  
out of date: 4.txt

When an existing input file is removed, then re-executing the task results in that file being reported to [IncrementalTaskInputs.removed(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.tasks.incremental.IncrementalTaskInputs.html#org.gradle.api.tasks.incremental.IncrementalTaskInputs:removed(org.gradle.api.Action)):

[Example: Running the incremental task with an input file removed](#42hahuo)

**build.gradle**

task removeInput() {  
 doLast {  
 file('inputs/3.txt').delete()  
 }  
}

**Output of** gradle -q removeInput incrementalReverse

> gradle -q removeInput incrementalReverse  
CHANGED inputs considered out of date  
removed: 3.txt

When an output file is deleted (or modified), then Gradle is unable to determine which input files are out of date. In this case, *all* input files are reported to the [IncrementalTaskInputs.outOfDate(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.tasks.incremental.IncrementalTaskInputs.html#org.gradle.api.tasks.incremental.IncrementalTaskInputs:outOfDate(org.gradle.api.Action)) action, and no input files are reported to the [IncrementalTaskInputs.removed(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.tasks.incremental.IncrementalTaskInputs.html#org.gradle.api.tasks.incremental.IncrementalTaskInputs:removed(org.gradle.api.Action)) action:

[Example: Running the incremental task with an output file removed](#2hmks2h)

**build.gradle**

task removeOutput() {  
 doLast {  
 file("$buildDir/outputs/1.txt").delete()  
 }  
}

**Output of** gradle -q removeOutput incrementalReverse

> gradle -q removeOutput incrementalReverse  
ALL inputs considered out of date  
out of date: 1.txt  
out of date: 2.txt  
out of date: 3.txt

When a task input property is modified, Gradle is unable to determine how this property impacted the task outputs, so all input files are assumed to be out of date. So similar to the changed output file example, *all* input files are reported to the [IncrementalTaskInputs.outOfDate(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.tasks.incremental.IncrementalTaskInputs.html#org.gradle.api.tasks.incremental.IncrementalTaskInputs:outOfDate(org.gradle.api.Action)) action, and no input files are reported to the [IncrementalTaskInputs.removed(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.tasks.incremental.IncrementalTaskInputs.html#org.gradle.api.tasks.incremental.IncrementalTaskInputs:removed(org.gradle.api.Action)) action:

[Example: Running the incremental task with an input property changed](#wrv2aa)

**Output of** gradle -q -PtaskInputProperty=changed incrementalReverse

> gradle -q -PtaskInputProperty=changed incrementalReverse  
ALL inputs considered out of date  
out of date: 1.txt  
out of date: 2.txt  
out of date: 3.txt

[Storing incremental state for cached tasks](#1i7cz46)

Using Gradle’s IncrementalTaskInputs is not the only way to create tasks that only works on changes since the last execution. Tools like the Kotlin compiler provide incrementality as a built-in feature. The way this is typically implemented is that the tool stores some analysis data about the state of the previous execution in some file. If such state files are [relocatable](#1xaqk5w), then they can be declared as outputs of the task. This way when the task’s results are loaded from cache, the next execution can already use the analysis data loaded from cache, too.

However, if the state files are non-relocatable, then they can’t be shared via the build cache. Indeed, when the task is loaded from cache, any such state files must be cleaned up to prevent stale state to confuse the tool during the next execution. Gradle can ensure such stale files are removed if they are declared via task.localState.register() or a property is marked with the @LocalState annotation.

[Declaring and Using Command Line Options](#2hio093)

| **Note** | The API for exposing command line options is an [incubating](#2xcytpi) feature. |
| --- | --- |

Sometimes a user wants to declare the value of an exposed task property on the command line instead of the build script. Being able to pass in 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.

[Declaring a command-line option](#3griky3)

Exposing a new command line option for a task property is straightforward. You just have to annotate the corresponding setter method of a property with [Option](http://docs.google.com/javadoc/org/gradle/api/tasks/options/Option.html). An option requires a mandatory identifier. Additionally, you can provide an optional description. A task can expose as many command line options as properties available in the class.

Let’s have a look at an example to illustrate the functionality. The custom task UrlVerify verifies whether a given URL can be resolved by making a 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](http://docs.google.com/javadoc/org/gradle/api/tasks/options/Option.html).

[Example: Declaring a command line option](#1vwsv5w)

**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](#4fwgdtp) by running the help task and the --task option.

[Using an option on the command line](#2v1qo1i)

Using an option on the command line has to adhere to the following rules:

* The option uses a double-dash as 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 options of a task can be declared in any order on the command line following the task name.

Getting back to the previous example, the build script creates a task instance of type UrlVerify and provides a value from the command line through the exposed option.

[Example: Using a command line option](#1a70y9b)

**build.gradle**

task verifyUrl(type: UrlVerify)

**Output of** gradle -q verifyUrl --url=http://www.google.com/

> gradle -q verifyUrl --url=http://www.google.com/  
Verifying URL 'http://www.google.com/'

[Supported data types for options](#3u6ogx4)

Gradle limits the set of data types that can be used for declaring command line options. The use on the command line differ per type.

boolean, Boolean

Describes an option with the value true or false. Passing the option on the command line does not require assigning a value. For example --enabled equates to true. The absence of the option uses the default values assign to the property; that is false for boolean and null for the complex data type.

String

Describes an option with an arbitrary String value. Passing the option on the command line requires a key-value pair of option and value separated by an equals sign e.g. --containerId=2x94held.

enum

Describes an option as enum. The enum has to be passed on the command line as key-value pair similar to the String type e.g. --log-level=DEBUG. The provided value is not case sensitive.

List<String>, List<enum>

Describes an option that can takes multiple values of a given type. The values for the option have to be provided as distinct declarations e.g. --imageId=123 --imageId=456. Other notations like comma-separated lists or multiple values separated by a space character are currently not supported.

[Documenting available values for an option](#29byr4x)

In theory, an option for a property type String or List<String> can accept any arbitrary value. Expected values for such an option can be documented programmatically with the help of the annotation [OptionValues](http://docs.google.com/javadoc/org/gradle/api/tasks/options/OptionValues.html). This annotation may be assigned to any method that returns a List of one of the supported data types. In addition, you have to provide the option identifier to indicate the relationship between option and available values.

| **Note** | Passing a value on the command line that is not supported by the option does not fail the build or throw an exception. You’ll have to implement custom logic for such behavior in the task action. |
| --- | --- |

This example demonstrates the use of multiple options for a single task. The task implementation provides a list of available values for the option output-type.

[Example: Declaring available values for an option](#oh91cq)

**UrlProcess.java**

**import** org.gradle.api.tasks.options.Option;  
**import** org.gradle.api.tasks.options.OptionValues;  
  
**public** **class** **UrlProcess** **extends** DefaultTask {  
 **private** String url;  
 **private** OutputType outputType;  
  
 @Option(option = "url", description = "Configures the URL to be write to the output.")  
 **public** **void** setUrl(String url) {  
 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 reponse from '{}' to '{}'", url, outputType);  
  
 *// retrieve content from URL and write to output*  
 }  
  
 **private** **static** **enum** OutputType {  
 CONSOLE, FILE  
 }  
}

[Listing command line options](#4fwgdtp)

Command line options using the annotations [Option](http://docs.google.com/javadoc/org/gradle/api/tasks/options/Option.html) and [OptionValues](http://docs.google.com/javadoc/org/gradle/api/tasks/options/OptionValues.html) are self-documenting. You will see [declared options](#3griky3) and their [available values](#29byr4x) reflected in the console output of the help task. The output renders options in alphabetical order.

[Example: Listing available values for option](#38gwk0j)

**Output of** gradle -q help --task processUrl

> gradle -q help --task processUrl  
Detailed task information for processUrl  
  
Path  
 :processUrl  
  
Type  
 UrlProcess (UrlProcess)  
  
Options  
 --output-type Configures the output type.  
 Available values are:  
 CONSOLE  
 FILE  
  
 --url Configures the URL to be write to the output.  
  
Description  
 -  
  
Group  
 -

[Limitations](#1nm6u8c)

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 then 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.

[The Worker API](#47lucw5)

| **Note** | The Worker API is an [incubating](#2xcytpi) feature. |
| --- | --- |

As can be seen from the discussion of [incremental tasks](#j2f68k), the work that a task performs can be viewed as discrete units (i.e. a subset of inputs that are transformed to a certain subset of outputs). Many times, these units of work are highly independent of each other, meaning they can be performed in any order and simply aggregated together to form the overall action of the task. In a single threaded execution, these units of work would execute in sequence, however if we have multiple processors, it would be desirable to perform independent units of work concurrently. By doing so, we can fully utilize the available resources at build time and complete the activity of the task faster.

The Worker API provides a mechanism for doing exactly this. It allows for safe, concurrent execution of multiple items of work during a task action. But the benefits of the Worker API are not confined to parallelizing the work of a task. You can also configure a desired level of isolation such that work can be executed in an isolated classloader or even in an isolated process. Furthermore, the benefits extend beyond even the execution of a single task. Using the Worker API, Gradle can begin to execute tasks in parallel by default. In other words, once a task has submitted its work to be executed asynchronously, and has exited the task action, Gradle can then begin the execution of other independent tasks in parallel, even if those tasks are in the same project.

[Using the Worker API](#2mr4n3y)

In order to submit work to the Worker API, two things must be provided: an implementation of the unit of work, and a configuration for the unit of work. The implementation is simply a class that extends java.lang.Runnable. This class should have a constructor that is annotated with javax.inject.Inject and accepts parameters that configure the class for a single unit of work. When a unit of work is submitted to the [WorkerExecutor](http://docs.google.com/javadoc/org/gradle/workers/WorkerExecutor.html), an instance of this class will be created and the parameters configured for the unit of work will be passed to the constructor.

[Example: Creating a unit of work implementation](#11wexbr)

**build.gradle**

**import** org.gradle.workers.WorkerExecutor  
  
**import** javax.inject.Inject  
  
*// The implementation of a single unit of work*  
**class** **ReverseFile** **implements** Runnable {  
 File fileToReverse  
 File destinationFile  
  
 @Inject  
 **public** ReverseFile(File fileToReverse, File destinationFile) {  
 this.fileToReverse = fileToReverse  
 this.destinationFile = destinationFile  
 }  
  
 @Override  
 **public** **void** run() {  
 destinationFile.text = fileToReverse.text.reverse()  
 }  
}

The configuration of the worker is represented by a [WorkerConfiguration](http://docs.google.com/javadoc/org/gradle/workers/WorkerConfiguration.html) and is set by configuring an instance of this object at the time of submission. However, in order to submit the unit of work, it is necessary to first acquire the [WorkerExecutor](http://docs.google.com/javadoc/org/gradle/workers/WorkerExecutor.html). To do this, a constructor should be provided that is annotated with javax.inject.Inject and accepts a [WorkerExecutor](http://docs.google.com/javadoc/org/gradle/workers/WorkerExecutor.html) parameter. Gradle will inject the instance of [WorkerExecutor](http://docs.google.com/javadoc/org/gradle/workers/WorkerExecutor.html) at runtime when the task is created.

[Example: Submitting a unit of work for execution](#3lw2fzk)

**build.gradle**

**class** **ReverseFiles** **extends** SourceTask {  
 **final** WorkerExecutor workerExecutor  
  
 @OutputDirectory  
 File outputDir  
  
 *// The WorkerExecutor will be injected by Gradle at runtime*  
 @Inject  
 **public** ReverseFiles(WorkerExecutor workerExecutor) {  
 this.workerExecutor = workerExecutor  
 }  
  
 @TaskAction  
 **void** reverseFiles() {  
 *// Create and submit a unit of work for each file*  
 source.each { file ->  
 workerExecutor.submit(ReverseFile.class) { WorkerConfiguration config ->  
 *// Use the minimum level of isolation*  
 config.isolationMode = IsolationMode.NONE  
  
 *// Constructor parameters for the unit of work implementation*  
 config.params file, project.file("${outputDir}/${file.name}")  
 }  
 }  
 }  
}

Note that one element of the [WorkerConfiguration](http://docs.google.com/javadoc/org/gradle/workers/WorkerConfiguration.html) is the params property. These are the parameters passed to the constructor of the unit of work implementation for each item of work submitted. Any parameters provided to the unit of work *must* be java.io.Serializable.

Once all of the work for a task action has been submitted, it is safe to exit the task action. The work will be executed asynchronously and in parallel (up to the setting of max-workers). Of course, any tasks that are dependent on this task (and any subsequent task actions of this task) will not begin executing until all of the asynchronous work completes. However, other independent tasks that have no relationship to this task can begin executing immediately.

If any failures occur while executing the asynchronous work, the task will fail and a [WorkerExecutionException](http://docs.google.com/javadoc/org/gradle/workers/WorkerExecutionException.html) will be thrown detailing the failure for each failed work item. This will be treated like any failure during task execution and will prevent any dependent tasks from executing.

In some cases, however, it might be desirable to wait for work to complete before exiting the task action. This is possible using the [WorkerExecutor.await()](http://docs.google.com/javadoc/org/gradle/workers/WorkerExecutor.html#await--) method. As in the case of allowing the work to complete asynchronously, any failures that occur while executing an item of work will be surfaced as a [WorkerExecutionException](http://docs.google.com/javadoc/org/gradle/workers/WorkerExecutionException.html) thrown from the [WorkerExecutor.await()](http://docs.google.com/javadoc/org/gradle/workers/WorkerExecutor.html#await--) method.

| **Note** | Note that Gradle will only begin running other independent tasks in parallel when a task has exited a task action and returned control of execution to Gradle. When [WorkerExecutor.await()](http://docs.google.com/javadoc/org/gradle/workers/WorkerExecutor.html#await--) is used, execution does not leave the task action. This means that Gradle will not allow other tasks to begin executing and will wait for the task action to complete before doing so. |
| --- | --- |

[Example: Waiting for asynchronous work to complete](#211cq7d)

**build.gradle**

*// Create and submit a unit of work for each file*  
source.each { file ->  
 workerExecutor.submit(ReverseFile.class) { config ->  
 config.isolationMode = IsolationMode.NONE  
 *// Constructor parameters for the unit of work implementation*  
 config.params file, project.file("${outputDir}/${file.name}")  
 }  
}  
  
*// Wait for all asynchronous work to complete before continuing*  
workerExecutor.await()  
logger.lifecycle("Created ${outputDir.listFiles().size()} reversed files in ${project.relativePath(outputDir)}")

[Isolation Modes](#4l108v6)

Gradle provides three isolation modes that can be configured on a unit of work and are specified using the [IsolationMode](http://docs.google.com/javadoc/org/gradle/workers/IsolationMode.html) enum:

IsolationMode.NONE

This states that the work should be run in a thread with a minimum of isolation. For instance, it will share the same classloader that the task is loaded from. This is the fastest level of isolation.

IsolationMode.CLASSLOADER

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 [WorkerConfiguration.classpath(java.lang.Iterable)](http://docs.google.com/javadoc/org/gradle/workers/WorkerConfiguration.html#classpath-java.lang.Iterable-).

IsolationMode.PROCESS

This states that the work should be run with a maximum level of isolation 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 [WorkerConfiguration.classpath(java.lang.Iterable)](http://docs.google.com/javadoc/org/gradle/workers/WorkerConfiguration.html#classpath-java.lang.Iterable-). Furthermore, the process will be a *Worker Daemon* which will stay alive and can be reused for future work items that may have the same requirements. This process can be configured with different settings than the Gradle JVM using [WorkerConfiguration.forkOptions(org.gradle.api.Action)](http://docs.google.com/javadoc/org/gradle/workers/WorkerConfiguration.html#forkOptions-org.gradle.api.Action-).

[Worker Daemons](#306aj2z)

When using IsolationMode.PROCESS, gradle will start a long-lived *Worker Daemon* process that can be reused for future work items.

[Example: Submitting an item of work to run in a worker daemon](#1fbktas)

**build.gradle**

workerExecutor.submit(ReverseFile.class) { WorkerConfiguration config ->  
 *// Run this work in an isolated process*  
 config.isolationMode = IsolationMode.PROCESS  
  
 *// Configure the options for the forked process*  
 config.forkOptions { JavaForkOptions options ->  
 options.maxHeapSize = "512m"  
 options.systemProperty "org.gradle.sample.showFileSize", "true"  
 }  
  
 *// Constructor parameters for the unit of work implementation*  
 config.params file, project.file("${outputDir}/${file.name}")  
}

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 [WorkerConfiguration.forkOptions(org.gradle.api.Action)](http://docs.google.com/javadoc/org/gradle/workers/WorkerConfiguration.html#forkOptions-org.gradle.api.Action-).

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 of the classpath entries requested. Note that a daemon is considered compatible if it has more classpath entries in addition to those requested.

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 considered compatible if it has set all of the jvm arguments requested. Note that a daemon is considered compatible if it has additional jvm arguments beyond those requested (except for arguments treated specially such as heap settings, assertions, debug, etc).

system properties

A daemon is considered compatible if it has set all of the system properties requested with the same values. Note that a daemon is considered compatible if it has additional system properties beyond those requested.

environment variables

A daemon is considered compatible if it has set all of the environment variables requested with the same values. Note that a daemon is considered compatible if it has more environment variables in addition to those requested.

bootstrap classpath

A daemon is considered compatible if it contains all of the bootstrap classpath entries requested. Note that a daemon is considered compatible if it has more bootstrap classpath entries in addition to those 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 is 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 either the build daemon that started them is stopped, or system memory becomes scarce. When available system memory is low, Gradle will begin stopping worker daemons in an attempt to minimize memory consumption.

[Re-using logic between task classes](#3zb8byl)

There are different ways to re-use logic between task classes. The easiest case is when you can extract the logic you want to share in a separate method or class and then use the extracted piece of code in your tasks. For example, the [Copy](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html) task re-uses the logic of the [Project.copy(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:copy(org.gradle.api.Action)) method. Another option is to add a task dependency on the task which outputs you want to re-use. Other options include using [task rules](#25vml65) or the [worker API](#47lucw5).

[The Base Plugin](#vx1227)

The Base Plugin provides some tasks and conventions that are common to most builds and adds a structure to the build that promotes consistency in how they are run. Its most significant contribution is a set of [*lifecycle* tasks](#34qadz2) that act as an umbrella for the more specific tasks provided by other plugins and build authors.

[Usage](#2egim6e)

[Example: Applying the Base Plugin](#tlswe7)

**build.gradle**

plugins {  
 id 'base'  
}

[Task](#3dlgf20)

clean — Delete

Deletes the build directory and everything in it, i.e. the path specified by the [Project.getBuildDir()](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:buildDir) project property.

check — *lifecycle task*

Plugins and build authors should attach their verification tasks, such as ones that run tests, to this lifecycle task using check.dependsOn(*task*).

assemble — *lifecycle task*

Plugins and build authors should attach tasks that produce distributions and other consumable artifacts to this lifecycle task. For example, jar produces the consumable artifact for Java libraries. Attach tasks to this lifecycle task using assemble.dependsOn(*task*).

build — *lifecycle task*

*Depends on*: check, assemble

Intended to build everything, including running all tests, producing the production artifacts and generating documentation. You will probably rarely attach concrete tasks directly to build as assemble and check are typically more appropriate.

build*Configuration* — task rule

Assembles those artifacts attached to the named configuration. For example, buildArchives will execute any task that is required to create any artifact attached to the archives configuration.

upload*Configuration* — task rule

Does the same as build*Configuration*, but also uploads all the artifacts attached to the given configuration.

clean*Task* — task rule

Removes the [defined outputs](#4hae2tp) of a task, e.g. cleanJar will delete the JAR file produced by the jar task of the Java Plugin.

[Dependency management](#1sqqp9t)

The Base Plugin adds no [configurations for dependencies](#37m2jsg), but it does add the following configurations for [artifacts](#4cqe7xm):

default

A fallback configuration used by consumer projects. Let’s say you have project B with a [project dependency](#43ekyio) on project A. Gradle uses some internal logic to determine which of project A’s artifacts and dependencies are added to the specified configuration of project B. If no other factors apply — you don’t need to worry what these are — then Gradle falls back to using everything in project A’s default configuration.

New builds and plugins should not be using the default configuration! It remains for the reason of backwards compatibility.

archives

A standard configuration for the production artifacts of a project. This results in an uploadArchives task for publishing artifacts attached to the archives configuration.

Note that the assemble task generates all artifacts that are attached to the archives configuration.

[Conventions](#2rvoi5f)

The Base Plugin only adds conventions related to the creation of archives, such as ZIPs, TARs and JARs. Specifically, it provides the following project properties that you can set:

archivesBaseName — default: $project.name

Provides the default [AbstractArchiveTask.getBaseName()](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.AbstractArchiveTask.html#org.gradle.api.tasks.bundling.AbstractArchiveTask:baseName) for archive tasks.

distsDirName — default: *distributions*

Default name of the directory in which distribution archives, i.e. non-JARs, are created.

libsDirName — default: *libs*

Default name of the directory in which library archives, i.e. JARs, are created.

The plugin also provides default values for the following properties on any task that extends [AbstractArchiveTask](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.AbstractArchiveTask.html):

destinationDir

Defaults to *$buildDir/$distsDirName* for non-JAR archives and *$buildDir/$libsDirName* for JARs and derivatives of JAR, such as WARs.

version

Defaults to $project.version or 'unspecified' if the project has no version.

baseName

Defaults to $archivesBaseName.

[Dependency Management](#3fwokq0)

[Introduction to Dependency Management](#1v1yuxt)

[What is dependency management?](#170ysd8)

Software projects rarely work in isolation. In most cases, a project relies on reusable functionality in the form of libraries or is broken up into individual components to compose a modularized system. Dependency management is a technique for declaring, resolving and using dependencies required by the project in an automated fashion.

| **Note** | For a general overview on the terms used throughout the user guide, refer to [Dependency Management Terminology](#4f1mdlm). |
| --- | --- |

[Dependency management in Gradle](#3r0mb11)

Gradle has built-in support for dependency management and lives up the task of fulfilling typical scenarios encountered in modern software projects. We’ll explore the main concepts with the help of an example project. The illustration below should give you an rough overview on all the moving parts.



*Figure 12. Dependency management big picture*

The example project builds Java source code. Some of the Java source files import classes from [Google Guava](https://github.com/google/guava), a open-source library providing a wealth of utility functionality. In addition to Guava, the project needs the [JUnit](http://junit.org/junit5/) libraries for compiling and executing test code.

Guava and JUnit represent the *dependencies* of this project. A build script developer can [declare dependencies](#3tbugp1) for different scopes e.g. just for compilation of source code or for executing tests. In Gradle, the [scope of a dependency](#37m2jsg) is called a *configuration*. For a full overview, see the reference material on [dependency types](#2u6wntf).

Often times dependencies come in the form of [modules](#265wl8u). You’ll need to tell Gradle where to find those modules so they can be consumed by the build. The location for storing modules is called a *repository*. By [declaring repositories](#28h4qwu) for a build, Gradle will know how to find and retrieve modules. Repositories can come in different forms: as local directory or a remote repository. The reference on [repository types](#19c6y18) provides a broad coverage on this topic.

At runtime, Gradle will locate the declared dependencies if needed for operating a specific task. The dependencies might need to be downloaded from a remote repository, retrieved from a local directory or requires another project to be built in a multi-project setting. This process is called *dependency resolution*. You can find a detailed discussion in [How dependency resolution works](#lb6vgn).

Once resolved, the resolution mechanism [stores the underlying files of a dependency in a local cache](#3l18frh), also referred to as the *dependency cache*. Future builds reuse the files stored in the cache to avoid unnecessary network calls.

Modules can provide additional metadata. Metadata is the data that describes the module in more detail e.g. the coordinates for finding it in a repository, information about the project, or its authors. As part of the metadata, a module can define that other modules are needed for it to work properly. For example, the JUnit 5 platform module also requires the platform commons module. Gradle automatically resolves those additional modules, so called *transitive dependencies*. If needed, you can [customize the behavior the handling of transitive dependencies](#1mrcu09) to your project’s requirements.

Projects with tens or hundreds of declared dependencies can easily suffer from dependency hell. Gradle provides sufficient tooling to visualize, navigate and analyze the dependency graph of a project either with the help of a [build scan](https://scans.gradle.com/get-started) or built-in tasks. Learn more in [Inspecting Dependencies](#nmf14n).



*Figure 13. Build scan dependencies report*

[How dependency resolution works](#lb6vgn)

Gradle takes your dependency declarations and repository definitions and attempts to download all of your dependencies by a process called *dependency resolution*. Below is a brief outline of how this process works.

* Given a required dependency, Gradle attempts to resolve the dependency by searching for the module the dependency points at. Each repository is inspected in order. Depending on the type of repository, Gradle looks for metadata files describing the module (.module, .pom or ivy.xml file) or directly for artifact files.
  + If the dependency is declared as a dynamic version (like 1.+), Gradle will resolve this to the highest available concrete version (like 1.2) in the repository. For Maven repositories, this is done using the maven-metadata.xml file, while for Ivy repositories this is done by directory listing.
  + If the module metadata is a POM file that has a parent POM declared, Gradle will recursively attempt to resolve each of the parent modules for the POM.
* Once each repository has been inspected for the module, Gradle will choose the 'best' one to use. This is done using the following criteria:
  + For a dynamic version, a 'higher' concrete version is preferred over a 'lower' version.
  + Modules declared by a module metadata file (.module, .pom or ivy.xml file) are preferred over modules that have an artifact file only.
  + Modules from earlier repositories are preferred over modules in later repositories.
  + When the dependency is declared by a concrete version and a module metadata file is found in a repository, there is no need to continue searching later repositories and the remainder of the process is short-circuited.
* All of the artifacts for the module are then requested from the *same repository* that was chosen in the process above.

The dependency resolution process is highly customizable to meet enterprise requirements. For more information, see the chapter on [customizing dependency resolution](#111kx3o).

[Dependency Management Terminology](#4f1mdlm)

Dependency management comes with a wealth of terminology. Here you can find the most commonly-used terms including references to the user guide to learn about their practical application.

[Configuration](#35aue4g)

A configuration is a named set of [dependencies](#1kg4oc9) grouped together for a specific goal: For example the implementation configuration represents the set of dependencies required to compile a project. Configurations provide access to the underlying, resolved [modules](#265wl8u) and their artifacts. For more information, see [Managing Dependency Configurations](#37m2jsg).

| **Note** | The word "configuration" is an overloaded term and has a different meaning outside of the context of dependency management. |
| --- | --- |

[Dependency](#1kg4oc9)

A dependency is a pointer to another piece of software required to build, test or run a [module](#265wl8u). For more information, see [Declaring Dependencies](#3tbugp1).

[Dependency constraint](#44fs702)

A dependency constraint defines requirements that need to be met by a module to make it a valid resolution result for the dependency. For example, a dependency constraint can narrow down the set of supported module versions. Dependency constraints can be used to express such requirements for transitive dependencies. For more information, see [Dependency Constraints](#2jl2h7v).

[Module](#265wl8u)

A piece of software that evolves over time e.g. [Google Guava](https://github.com/google/guava). Every module has a name. Each release of a module is optimally represented by a [module version](#yqcrfo). For convenient consumption, modules can be hosted in a [repository](#3iq0a3h).

[Module metadata](#1xvakba)

Releases of a [module](#265wl8u) can provide metadata. Metadata is the data that describes the module in more detail e.g. the coordinates for locating it in a repository, information about the project or required [transitive dependencies](#4huy2z3). In Maven the metadata file is called .pom, in Ivy it is called ivy.xml.

[Module version](#yqcrfo)

A module version represents a distinct set of changes of a released [module](#265wl8u). For example 18.0 represents the version of the module with the coordinates com.google:guava:18.0. In practice there’s no limitation to the scheme of the module version. Timestamps, numbers, special suffixes like -GA are all allowed identifiers. The most widely-used versioning strategy is [semantic versioning](https://semver.org/).

[Repository](#3iq0a3h)

A repository hosts a set of [modules](#265wl8u), each of which may provide one or many releases indicated by a [module version](#yqcrfo). The repository can be based on a binary repository product (e.g. Artifactory or Nexus) or a directory structure in the filesystem. For more information, see [Declaring Repositories](#28h4qwu).

[Resolution rule](#2x08d6w)

A resolution rule influences the behavior of how a [dependency](#1kg4oc9) is resolved. Resolution rules are defined as part of the build logic. For more information, see [Customizing Dependency Resolution Behavior](#111kx3o).

[Transitive dependency](#4huy2z3)

A [module](#265wl8u) can have dependencies on other modules to work properly, so-called transitive dependencies. Releases of a module hosted on a [repository](#3iq0a3h) can provide [metadata](#1xvakba) to declare those transitive dependencies. By default, Gradle resolves transitive dependencies automatically. However, the behavior is highly customizable. For more information, see [Managing Transitive Dependencies](#1mrcu09).

[Dependency Types](#2u6wntf)

[Module dependencies](#1c5inep)

Module dependencies are the most common dependencies. They refer to a module in a repository.

[Example: Module dependencies](#3w5662i)

**build.gradle**

dependencies {  
 runtime group: 'org.springframework', name: 'spring-core', version: '2.5'  
 runtime 'org.springframework:spring-core:2.5',  
 'org.springframework:spring-aop:2.5'  
 runtime(  
 [group: 'org.springframework', name: 'spring-core', version: '2.5'],  
 [group: 'org.springframework', name: 'spring-aop', version: '2.5']  
 )  
 runtime('org.hibernate:hibernate:3.0.5') {  
 transitive = true  
 }  
 runtime group: 'org.hibernate', name: 'hibernate', version: '3.0.5', transitive: true  
 runtime(group: 'org.hibernate', name: 'hibernate', version: '3.0.5') {  
 transitive = true  
 }  
}

See the [DependencyHandler](http://docs.google.com/dsl/org.gradle.api.artifacts.dsl.DependencyHandler.html) class in the API documentation for more examples and a complete reference.

Gradle provides different notations for module dependencies. There is a string notation and a map notation. A module dependency has an API which allows further configuration. Have a look at [ExternalModuleDependency](http://docs.google.com/javadoc/org/gradle/api/artifacts/ExternalModuleDependency.html) to learn all about the API. This API provides properties and configuration methods. Via the string notation you can define a subset of the properties. With the map notation you can define all properties. To have access to the complete API, either with the map or with the string notation, you can assign a single dependency to a configuration together with a closure.

| **Note** | If you declare a module dependency, Gradle looks for a module metadata file (.module, .pom or ivy.xml) in the repositories. If such a module metadata file exists, it is parsed and the artifacts of this module (e.g. hibernate-3.0.5.jar) as well as its dependencies (e.g. cglib) are downloaded. If no such module metadata file exists, Gradle may look, depending on the [metadata sources definitions](#2baggab), for an artifact file called hibernate-3.0.5.jar directly. In Maven, a module can have one and only one artifact. In Gradle and Ivy, a module can have multiple artifacts. Each artifact can have a different set of dependencies. |
| --- | --- |

[File dependencies](#qfqqi4)

File dependencies allow you to directly add a set of files to a configuration, without first adding them to a repository. This can be useful if you cannot, or do not want to, place certain files in a repository. Or if you do not want to use any repositories at all for storing your dependencies.

To add some files as a dependency for a configuration, you simply pass a [file collection](#4f5j5f0) as a dependency:

[Example: File dependencies](#3afe95x)

**build.gradle**

dependencies {  
 runtime files('libs/a.jar', 'libs/b.jar')  
 runtime fileTree(dir: 'libs', include: '\*.jar')  
}

File dependencies are not included in the published dependency descriptor for your project. However, file dependencies are included in transitive project dependencies within the same build. This means they cannot be used outside the current build, but they can be used with the same build.

You can declare which tasks produce the files for a file dependency. You might do this when, for example, the files are generated by the build.

[Example: Generated file dependencies](#1pkojdq)

**build.gradle**

dependencies {  
 compile files("$buildDir/classes") {  
 builtBy 'compile'  
 }  
}  
  
task compile {  
 doLast {  
 println 'compiling classes'  
 }  
}  
  
task list(dependsOn: configurations.compile) {  
 doLast {  
 println "classpath = ${configurations.compile.collect { File file -> file.name }}"  
 }  
}

**Output of** gradle -q list

> gradle -q list  
compiling classes  
classpath = [classes]

[Project dependencies](#43ekyio)

Gradle distinguishes between external dependencies and dependencies on projects which are part of the same multi-project build. For the latter you can declare *project dependencies*.

[Example: Project dependencies](#49kc21j)

**build.gradle**

dependencies {  
 compile project(':shared')  
}

For more information see the API documentation for [ProjectDependency](http://docs.google.com/javadoc/org/gradle/api/artifacts/ProjectDependency.html).

Multi-project builds are discussed in [this chapter](#2p2csry).

[Gradle distribution-specific dependencies](#2opmc9c)

[Gradle API dependency](#13uwmh5)

You can declare a dependency on the API of the current version of Gradle by using the [DependencyHandler.gradleApi()](http://docs.google.com/dsl/org.gradle.api.artifacts.dsl.DependencyHandler.html#org.gradle.api.artifacts.dsl.DependencyHandler:gradleApi()) method. This is useful when you are developing custom Gradle tasks or plugins.

[Example: Gradle API dependencies](#3nuk54y)

**build.gradle**

dependencies {  
 compile gradleApi()  
}

[Gradle TestKit dependency](#22zufcr)

You can declare a dependency on the TestKit API of the current version of Gradle by using the [DependencyHandler.gradleTestKit()](http://docs.google.com/dsl/org.gradle.api.artifacts.dsl.DependencyHandler.html#org.gradle.api.artifacts.dsl.DependencyHandler:gradleTestKit()) method. This is useful for writing and executing functional tests for Gradle plugins and build scripts.

[Example: Gradle TestKit dependencies](#i54pkk)

**build.gradle**

dependencies {  
 testCompile gradleTestKit()  
}

[The TestKit chapter](#ihv636) explains the use of TestKit by example.

[Local Groovy dependency](#324s88d)

You can declare a dependency on the Groovy that is distributed with Gradle by using the [DependencyHandler.localGroovy()](http://docs.google.com/dsl/org.gradle.api.artifacts.dsl.DependencyHandler.html#org.gradle.api.artifacts.dsl.DependencyHandler:localGroovy()) method. This is useful when you are developing custom Gradle tasks or plugins in Groovy.

[Example: Gradle’s Groovy dependencies](#1ha2ig6)

**build.gradle**

dependencies {  
 compile localGroovy()  
}

[Repository Types](#19c6y18)

[Flat directory repository](#419q13z)

Some projects might prefer to store dependencies on a shared drive or as part of the project source code instead of a binary repository product. If you want to use a (flat) filesystem directory as a repository, simply type:

[Example: Flat repository resolver](#2gf0bbs)

**build.gradle**

repositories {  
 flatDir {  
 dirs 'lib'  
 }  
 flatDir {  
 dirs 'lib1', 'lib2'  
 }  
}

This adds repositories which look into one or more directories for finding dependencies. Note that this type of repository does not support any meta-data formats like Ivy XML or Maven POM files. Instead, Gradle will dynamically generate a module descriptor (without any dependency information) based on the presence of artifacts. However, as Gradle prefers to use modules whose descriptor has been created from real meta-data rather than being generated, flat directory repositories cannot be used to override artifacts with real meta-data from other repositories. For example, if Gradle finds only jmxri-1.2.1.jar in a flat directory repository, but jmxri-1.2.1.pom in another repository that supports meta-data, it will use the second repository to provide the module.

For the use case of overriding remote artifacts with local ones consider using an Ivy or Maven repository instead whose URL points to a local directory. If you only work with flat directory repositories you don’t need to set all attributes of a dependency.

[Maven Central repository](#vkaljl)

Maven Central is a popular repository hosting open source libraries for consumption by Java projects.

To declare the [central Maven repository](https://repo.maven.apache.org/maven2/) for your build add this to your script:

[Example: Adding central Maven repository](#3fjy47e)

**build.gradle**

repositories {  
 mavenCentral()  
}

[JCenter Maven repository](#1up8ef7)

[Bintray](http://bintray.com)'s JCenter is an up-to-date collection of all popular Maven OSS artifacts, including artifacts published directly to Bintray.

To declare the [JCenter Maven repository](https://jcenter.bintray.com) add this to your build script:

[Example: Adding Bintray’s JCenter Maven repository](#4eovx30)

**build.gradle**

repositories {  
 jcenter()  
}

[Google Maven repository](#2tu67at)

The Google repository hosts Android-specific artifacts including the Android SDK. For usage examples, see the [relevant documentation](https://developer.android.com/studio/build/dependencies.html#google-maven).

To declare the [Google Maven repository](https://maven.google.com/) add this to your build script:

[Example: Adding Google Maven repository](#18zghim)

**build.gradle**

repositories {  
 google()  
}

[Local Maven repository](#3sz406f)

Gradle can consume dependencies available in the [local Maven repository](https://maven.apache.org/guides/introduction/introduction-to-repositories.html). Declaring this repository is beneficial for teams that publish to the local Maven repository with one project and consume the artifacts by Gradle in another project.

| **Note** | Gradle stores resolved dependencies in [its own cache](#3l18frh). A build does not need to declare the local Maven repository even if you resolve dependencies from a Maven-based, remote repository. |
| --- | --- |

To declare the local Maven cache as a repository add this to your build script:

[Example: Adding the local Maven cache as a repository](#284eae8)

**build.gradle**

repositories {  
 mavenLocal()  
}

Gradle uses the same logic as Maven to identify the location of your local Maven cache. If a local repository location is defined in a settings.xml, this location will be used. The settings.xml in *USER\_HOME*/.m2 takes precedence over the settings.xml in *M2\_HOME*/conf. If no settings.xml is available, Gradle uses the default location *USER\_HOME*/.m2/repository.

[Custom Maven repositories](#n9okm1)

Many organizations host dependencies in an in-house Maven repository only accessible within the company’s network. Gradle can declare Maven repositories by URL.

For adding a custom Maven repository you can do:

[Example: Adding custom Maven repository](#379c39u)

**build.gradle**

repositories {  
 maven {  
 url "http://repo.mycompany.com/maven2"  
 }  
}

Sometimes a repository will have the POMs published to one location, and the JARs and other artifacts published at another location. To define such a repository, you can do:

[Example: Adding additional Maven repositories for JAR files](#1memdhn)

**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 look at the first URL for the POM and the JAR. If the JAR can’t be found there, the artifact URLs are used to look for JARs.

See [Configuring HTTP authentication schemes](#46e9w5g) for authentication options.

[Custom Ivy repositories](#2ljk6d9)

Organizations might decide to host dependencies in an in-house Ivy repository. Gradle can declare Ivy repositories by URL.

[Defining an Ivy repository with a standard layout](#10ougl2)

To declare an Ivy repository using the standard layout no additional customization is needed. You just declare the URL.

[Example: Ivy repository](#3kohz8v)

**build.gradle**

repositories {  
 ivy {  
 url "http://repo.mycompany.com/repo"  
 }  
}

[Defining a named layout for an Ivy repository](#1zts9go)

You can specify that your repository conforms to the Ivy or Maven default layout by using a named layout.

[Example: Ivy repository with named layout](#4jtfs4h)

**build.gradle**

repositories {  
 ivy {  
 url "http://repo.mycompany.com/repo"  
 layout "maven"  
 }  
}

Valid named layout values are 'gradle' (the default), 'maven', 'ivy' and 'pattern'. See [IvyArtifactRepository.layout(java.lang.String, groovy.lang.Closure)](http://docs.google.com/dsl/org.gradle.api.artifacts.repositories.IvyArtifactRepository.html#org.gradle.api.artifacts.repositories.IvyArtifactRepository:layout(java.lang.String,%20groovy.lang.Closure)) in the API documentation for details of these named layouts.

[Defining custom pattern layout for an Ivy repository](#2yyq2ca)

To define an Ivy repository with a non-standard layout, you can define a 'pattern' layout for the repository:

[Example: Ivy repository with pattern layout](#1e40ck3)

**build.gradle**

repositories {  
 ivy {  
 url "http://repo.mycompany.com/repo"  
 layout "pattern", {  
 artifact "[module]/[revision]/[type]/[artifact].[ext]"  
 }  
 }  
}

To define an Ivy repository which fetches Ivy files and artifacts from different locations, you can define separate patterns to use to locate the Ivy files and artifacts:

Each artifact or ivy specified for a repository adds an *additional* pattern to use. The patterns are used in the order that they are defined.

[Example: Ivy repository with multiple custom patterns](#3y3nv7w)

**build.gradle**

repositories {  
 ivy {  
 url "http://repo.mycompany.com/repo"  
 layout "pattern", {  
 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, a repository with pattern layout can have its 'organisation' part laid out in Maven style, with forward slashes replacing dots as separators. For example, the organisation my.company would then be represented as my/company.

[Example: Ivy repository with Maven compatible layout](#2d8y5fp)

**build.gradle**

repositories {  
 ivy {  
 url "http://repo.mycompany.com/repo"  
 layout "pattern", {  
 artifact "[organisation]/[module]/[revision]/[artifact]-[revision].[ext]"  
 m2compatible = true  
 }  
 }  
}

[Accessing password-protected Ivy repositories](#se8fni)

You can specify credentials for Ivy repositories secured by basic authentication.

[Example: Ivy repository with authentication](#3cdvybb)

**build.gradle**

repositories {  
 ivy {  
 url "http://repo.mycompany.com"  
 credentials {  
 username "user"  
 password "password"  
 }  
 }  
}

[Supported metadata sources](#2baggab)

When searching for a module in a repository, Gradle, by default, checks for supported metadata file formats in that repository. In a Maven repository, Gradle looks for a .pom file, in an ivy repository it looks for an ivy.xml file and in a flat directory repository it looks directly for .jar files as it does not expect any metadata. Starting with 5.0, Gradle also looks for .module (Gradle module metadata) files.

However, if you define a customized repository you might want to configure this behavior. For example, you can define a Maven repository without .pom files but only jars. To do so, you can configure *metadata sources* for any repository.

[Example: Maven repository that supports artifacts without metadata](#1rj68j4)

**build.gradle**

repositories {  
 maven {  
 url "http://repo.mycompany.com/repo"  
 metadataSources {  
 mavenPom()  
 artifact()  
 }  
 }  
}

You can specify multiple sources to tell Gradle to keep looking if a file was not found. In that case, the order of checking for sources is predefined.

The following metadata sources are supported:

Table 6. Repository transport protocols

| **Metadata source** | **Description** | **Order** | **Maven** | **Ivy / flat dir** |
| --- | --- | --- | --- | --- |
| gradleMetadata() | Look for Gradle .module files | 1st | yes | yes |
| mavenPom() | Look for Maven .pom files | 2nd | yes | yes |
| ivyDescriptor() | Look for ivy.xml files | 2nd | no | yes |
| artifact() | Look directly for artifact | 3rd | yes | yes |

| **Note** | The defaults for Ivy and Maven repositories change with Gradle 5.0. Before 5.0, artifact() was included in the defaults. Leading to some inefficiency when modules are missing completely. To restore this behavior, for example, for Maven central you can use mavenCentral { mavenPom(); artifact() }. In a similar way, you can opt into the new behavior in older Gradle verisions using mavenCentral { mavenPom() } |
| --- | --- |

[Supported repository transport protocols](#4bitr6x)

Maven and Ivy repositories support the use of various transport protocols. At the moment the following protocols are supported:

Table 7. Repository transport protocols

| **Type** | **Credential types** |
| --- | --- |
| file | none |
| http | username/password |
| https | username/password |
| sftp | username/password |
| s3 | access key/secret key/session token or Environment variables |
| gcs | [default application credentials](https://developers.google.com/identity/protocols/application-default-credentials) sourced from well known files, Environment variables etc. |

| **Note** | Username and password should never be checked in plain text into version control as part of your build file. You can store the credentials in a local gradle.properties file and use one of the open source Gradle plugins for encrypting and consuming credentials e.g. the [credentials plugin](https://plugins.gradle.org/plugin/nu.studer.credentials). |
| --- | --- |

The transport protocol is part of the URL definition for a repository. The following build script demonstrates how to create a HTTP-based Maven and Ivy repository:

[Example: Declaring a Maven and Ivy repository](#2qo41eq)

**build.gradle**

repositories {  
 maven {  
 url "http://repo.mycompany.com/maven2"  
 }  
  
 ivy {  
 url "http://repo.mycompany.com/repo"  
 }  
}

The following example shows how to declare SFTP repositories:

[Example: Using the SFTP protocol for a repository](#15tebmj)

**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"  
 }  
 }  
}

When using an AWS S3 backed repository you need to authenticate using [AwsCredentials](http://docs.google.com/dsl/org.gradle.api.credentials.AwsCredentials.html), providing access-key and a private-key. The following example shows how to declare a S3 backed repository and providing AWS credentials:

[Example: Declaring a S3 backed Maven and Ivy repository](#3pt1uac)

**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"  
 }  
 }  
}

You can also delegate all credentials to the AWS sdk by using the AwsImAuthentication. The following example shows how:

[Example: Declaring a S3 backed Maven and Ivy repository using IAM](#24yc4i5)

**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)  
 }  
 }  
}

When using a Google Cloud Storage backed repository default application credentials will be used with no further configuration required:

[Example: Declaring a Google Cloud Storage backed Maven and Ivy repository using default application credentials](#k3mepy)

**build.gradle**

repositories {  
 maven {  
 url "gcs://myCompanyBucket/maven2"  
 }  
  
 ivy {  
 url "gcs://myCompanyBucket/ivyrepo"  
 }  
}

[S3 configuration properties](#3439xdr)

The following system properties can be used to configure the interactions with s3 repositories:

org.gradle.s3.endpoint

Used to override the AWS S3 endpoint when using a non AWS, S3 API compatible, storage service.

org.gradle.s3.maxErrorRetry

Specifies the maximum number of times to retry a request in the event that the S3 server responds with a HTTP 5xx status code. When not specified a default value of 3 is used.

[S3 URL formats](#1j8k7lk)

S3 URL’s are 'virtual-hosted-style' and must be in the following format

s3://<bucketName>[.<regionSpecificEndpoint>]/<s3Key>

e.g. s3://myBucket.s3.eu-central-1.amazonaws.com/maven/release

* myBucket is the AWS S3 bucket name.
* s3.eu-central-1.amazonaws.com is the *optional* [region specific endpoint](http://docs.aws.amazon.com/general/latest/gr/rande.html#s3_region).
* /maven/release is the AWS S3 key (unique identifier for an object within a bucket)

[S3 proxy settings](#4387q9d)

A proxy for S3 can be configured using the following system properties:

* https.proxyHost
* https.proxyPort
* https.proxyUser
* https.proxyPassword
* http.nonProxyHosts

If the 'org.gradle.s3.endpoint' property has been specified with a http (not https) URI the following system proxy settings can be used:

* http.proxyHost
* http.proxyPort
* http.proxyUser
* http.proxyPassword
* http.nonProxyHosts

[AWS S3 V4 Signatures (AWS4-HMAC-SHA256)](#2idi0h6)

Some of the AWS S3 regions (eu-central-1 - Frankfurt) require that all HTTP requests are signed in accordance with AWS’s [signature version 4](http://docs.aws.amazon.com/general/latest/gr/signature-version-4.html). It is recommended to specify S3 URL’s containing the region specific endpoint when using buckets that require V4 signatures. e.g.

s3://somebucket.s3.eu-central-1.amazonaws.com/maven/release

| **Note** | When a region-specific endpoint is not specified for buckets requiring V4 Signatures, Gradle will use the default AWS region (us-east-1) and the following warning will appear on the console:  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 buckets requiring V4 signatures means:   * 3 round-trips to AWS, as opposed to one, for every file upload and download. * Depending on location - increased network latencies and slower builds. * Increased likelihood of transmission failures. |
| --- | --- |

[AWS S3 Cross Account Access](#xisaoz)

Some organizations may have multiple AWS accounts, e.g. one for each team. The AWS account of the bucket owner is often different from the artifact publisher and consumers. The bucket owner needs to be able to grant the consumers access otherwise the artifacts will only be usable by the publisher’s account. This is done by adding the bucket-owner-full-control [Canned ACL](https://docs.aws.amazon.com/AmazonS3/latest/dev/acl-overview.html#canned-acl) to the uploaded objects. Gradle will do this in every upload. Make sure the publisher has the required IAM permission, PutObjectAcl (and PutObjectVersionAcl if bucket versioning is enabled), either directly or via an assumed IAM Role (depending on your case). You can read more at [AWS S3 Access Permissions](https://docs.aws.amazon.com/AmazonS3/latest/dev/s3-access-control.html).

[Google Cloud Storage configuration properties](#3hiftcs)

The following system properties can be used to configure the interactions with [Google Cloud Storage](https://cloud.google.com/storage/) repositories:

org.gradle.gcs.endpoint

Used to override the Google Cloud Storage endpoint when using a non-Google Cloud Platform, Google Cloud Storage API compatible, storage service.

org.gradle.gcs.servicePath

Used to override the Google Cloud Storage root service path which the Google Cloud Storage client builds requests from, defaults to /.

[Google Cloud Storage URL formats](#1wnq3kl)

Google Cloud Storage URL’s are 'virtual-hosted-style' and must be in the following format gcs://<bucketName>/<objectKey>

e.g. gcs://myBucket/maven/release

* myBucket is the Google Cloud Storage bucket name.
* /maven/release is the Google Cloud Storage key (unique identifier for an object within a bucket)

[Configuring HTTP authentication schemes](#46e9w5g)

When configuring a repository using HTTP or HTTPS transport protocols, multiple authentication schemes are available. By default, Gradle will attempt to use all schemes that are supported by the Apache HttpClient library, [documented here](http://hc.apache.org/httpcomponents-client-ga/tutorial/html/authentication.html#d5e625). In some cases, it may be preferable to explicitly specify which authentication schemes should be used when exchanging credentials with a remote server. When explicitly declared, only those schemes are used when authenticating to a remote repository.

You can specify credentials for Maven repositories secured by basic authentication using api:org.gradle.api.credentials.PasswordCredentials[].

[Example: Accessing password-protected Maven repository](#4gndm8e)

**build.gradle**

repositories {  
 maven {  
 url "http://repo.mycompany.com/maven2"  
 credentials {  
 username "user"  
 password "password"  
 }  
 }  
}

| **Note** | The code for this example can be found at samples/userguide/artifacts/defineRepository in the ‘-all’ distribution of Gradle. |
| --- | --- |

The following example show how to configure a repository to use only api:org.gradle.authentication.http.DigestAuthentication[]:

[Example: Configure repository to use only digest authentication](#2vsnwg7)

**build.gradle**

repositories {  
 maven {  
 url 'https://repo.mycompany.com/maven2'  
 credentials {  
 username "user"  
 password "password"  
 }  
 authentication {  
 digest(DigestAuthentication)  
 }  
 }  
}

Currently supported authentication schemes are:

[BasicAuthentication](http://docs.google.com/javadoc/org/gradle/authentication/http/BasicAuthentication.html)

Basic access authentication over HTTP. When using this scheme, credentials are sent preemptively.

[DigestAuthentication](http://docs.google.com/javadoc/org/gradle/authentication/http/DigestAuthentication.html)

Digest access authentication over HTTP.

[HttpHeaderAuthentication](http://docs.google.com/javadoc/org/gradle/authentication/http/HttpHeaderAuthentication.html)

Authentication based on any custom HTTP header, e.g. private tokens, OAuth tokens, etc.

[Using preemptive authentication](#1axy6o0)

Gradle’s default behavior is to only submit credentials when a server responds with an authentication challenge in the form of a HTTP 401 response. In some cases, the server will respond with a different code (ex. for repositories hosted on GitHub a 404 is returned) causing dependency resolution to fail. To get around this behavior, credentials may be sent to the server preemptively. To enable preemptive authentication simply configure your repository to explicitly use the [BasicAuthentication](http://docs.google.com/javadoc/org/gradle/authentication/http/BasicAuthentication.html) scheme:

[Example: Configure repository to use preemptive authentication](#3uxlpbt)

**build.gradle**

repositories {  
 maven {  
 url 'https://repo.mycompany.com/maven2'  
 credentials {  
 username "user"  
 password "password"  
 }  
 authentication {  
 basic(BasicAuthentication)  
 }  
 }  
}

[Using HTTP header authentication](#2a2vzjm)

You can specify any HTTP header for secured Maven repositories requiring token, OAuth2 or other HTTP header based authentication using api:org.gradle.api.credentials.HttpHeaderCredentials[] with api:org.gradle.authentication.http.HttpHeaderAuthentication[].

[Example: Accessing header-protected Maven repository](#p869rf)

**build.gradle**

repositories {  
 maven {  
 url "http://repo.mycompany.com/maven2"  
 credentials(HttpHeaderCredentials) {  
 name = "Private-Token"  
 value = "TOKEN"  
 }  
 authentication {  
 header(HttpHeaderAuthentication)  
 }  
 }  
}

| **Note** | The code for this example can be found at samples/userguide/artifacts/defineRepository in the ‘-all’ distribution of Gradle. |
| --- | --- |

[Declaring Dependencies](#3tbugp1)

Gradle builds can declare dependencies on modules hosted in repositories, files and other Gradle projects. You can find examples for common scenarios in this section. For more information, see the [full reference on all types of dependencies](#2u6wntf).

Every dependency needs to be assigned to a configuration when declared in a build script. For more information on the purpose and syntax of configurations, see [Managing Dependency Configurations](#37m2jsg).

[Declaring a dependency to a module](#397tsf8)

Modern software projects rarely build code in isolation. Projects reference modules for the purpose of reusing existing and proven functionality. Upon resolution, selected versions of modules are downloaded from dedicated repositories and stored in the [dependency cache](#3l18frh) to avoid unnecessary network traffic.



*Figure 14. Resolving dependencies from remote repositories*

[Declaring a concrete version of a dependency](#1od42n1)

A typical example for such a library in a Java project is the [Spring framework](https://projects.spring.io/spring-framework/). The following code snippet declares a compile-time dependency on the Spring web module by its coordinates: org.springframework:spring-web:5.0.2.RELEASE. Gradle resolves the module including its transitive dependencies from the [Maven Central repository](https://search.maven.org/) and uses it to compile Java source code. The version attribute of the dependency coordinates points to a *concrete version* indicating that the underlying artifacts do not change over time. The use of concrete versions ensure reproducibility for the aspect of dependency resolution.

[Example: Declaring a dependency with a concrete version](#48crlau)

**build.gradle**

apply plugin: 'java-library'  
  
repositories {  
 mavenCentral()  
}  
  
dependencies {  
 implementation 'org.springframework:spring-web:5.0.2.RELEASE'  
}

A Gradle project can define other types of repositories hosting modules. You can learn more about the syntax and API in the section on [declaring repositories](#28h4qwu). Refer to [the chapter on the Java Plugin](#2w5ecyt) for a deep dive on declaring dependencies for a Java project. The resolution behavior for dependencies is [highly customizable](#111kx3o).

[Declaring a dependency without version](#2ni1vin)

A recommended practice for larger projects is to declare dependencies without versions and use [dependency constraints](#2jl2h7v) for version declaration. The advantage is that dependency constrains allow you to manage versions of all dependencies, including transitive ones, in one place.

[Example: Declaring a dependency without version](#12nc5qg)

**build.gradle**

dependencies {  
 implementation 'org.springframework:spring-web'  
}  
  
dependencies {  
 constraints {  
 implementation 'org.springframework:spring-web:5.0.2.RELEASE'  
 }  
}

[Declaring a dynamic version](#3mmzoe9)

Projects might adopt a more aggressive approach for consuming dependencies to modules. For example you might want to always integrate the latest version of a dependency to consume cutting edge features at any given time. A *dynamic version* allows for resolving the latest version or the latest version of a version range for a given module.

| **Note** | Using dynamic versions in a build bears the risk of potentially breaking it. As soon as a new version of the dependency is released that contains an incompatible API change your source code might stop compiling. |
| --- | --- |

[Example: Declaring a dependency with a dynamic version](#21s9ym2)

**build.gradle**

apply plugin: 'java-library'  
  
repositories {  
 mavenCentral()  
}  
  
dependencies {  
 implementation 'org.springframework:spring-web:5.+'  
}

A [build scan](https://scans.gradle.com/) can effectively visualize dynamic dependency versions and their respective, selected versions.



*Figure 15. Dynamic dependencies in build scan*

By default, Gradle caches dynamic versions of dependencies for 24 hours. Within this time frame, Gradle does not try to resolve newer versions from the declared repositories. The [threshold can be configured](#gxk8tv) as needed for example if you want to resolve new versions earlier.

[Declaring a changing version](#30x7rho)

A team might decide to 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 and often is to release a module with a so-called *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](https://maven.apache.org/guides/getting-started/index.html#What_is_a_SNAPSHOT_version). Snapshot versions contain the suffix -SNAPSHOT. The following example demonstrates how to declare a snapshot version on the Spring dependency.

[Example: Declaring a dependency with a changing version](#1g2i1ph)

**build.gradle**

apply plugin: 'java-library'  
  
repositories {  
 mavenCentral()  
 maven {  
 url 'https://repo.spring.io/snapshot/'  
 }  
}  
  
dependencies {  
 implementation 'org.springframework:spring-web:5.0.3.BUILD-SNAPSHOT'  
}

By default, Gradle caches changing versions of dependencies for 24 hours. Within this time frame, Gradle does not try to resolve newer versions from the declared repositories. The [threshold can be configured](#gxk8tv) as needed for example if you want to resolve new snapshot versions earlier.

Gradle is flexible enough to treat any version as changing version e.g. if you wanted to model snapshot behavior for an Ivy module. All you need to do is to set the property [ExternalModuleDependency.setChanging(boolean)](http://docs.google.com/javadoc/org/gradle/api/artifacts/ExternalModuleDependency.html#setChanging-boolean-) to true.

[Declaring a file dependency](#4025kda)

Projects sometimes do not rely on a binary repository product e.g. JFrog Artifactory or Sonatype Nexus for hosting and resolving external dependencies. It’s common practice to host those dependencies on a shared drive or check them into version control alongside the project source code. Those dependencies are referred to as *file dependencies*, the reason being that they represent a file without any [metadata](#1xvakba) (like information about transitive dependencies, the origin or its author) attached to them.



*Figure 16. Resolving file dependencies from the local file system and a shared drive*

The following example resolves file dependencies from the directories ant, libs and tools.

[Example: Declaring multiple file dependencies](#2f7ful3)

**build.gradle**

configurations {  
 antContrib  
 externalLibs  
 deploymentTools  
}  
  
dependencies {  
 antContrib files('ant/antcontrib.jar')  
 externalLibs files('libs/commons-lang.jar', 'libs/log4j.jar')  
 deploymentTools fileTree(dir: 'tools', include: '\*.exe')  
}

As you can see in the code example, every dependency has to define its exact location in the file system. The most prominent methods for creating a file reference are [Project.files(java.lang.Object…​)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:files(java.lang.Object%5B%5D)), [ProjectLayout.files(java.lang.Object…​)](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#files-java.lang.Object...-), [ProjectLayout.configurableFiles(java.lang.Object…​)](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#configurableFiles-java.lang.Object...-), and [Project.fileTree(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:fileTree(java.lang.Object)) Alternatively, you can also define the source directory of one or many file dependencies in the form of a [flat directory repository](#419q13z).

[Declaring a project dependency](#1n1mu2y)

Software projects often break up software components into modules to improve maintainability and prevent strong coupling. Modules can define dependencies between each other to reuse code within the same project.

Gradle can model dependencies between modules. Those dependencies are called *project dependencies* because each module is represented by a Gradle project. At runtime, the build automatically ensures that project dependencies are built in the correct order and added to the classpath for compilation. The chapter [Authoring Multi-Project Builds](#2p2csry) discusses how to set up and configure multi-project builds in more detail.



*Figure 17. Dependencies between projects*

The following example declares the dependencies on the utils and api project from the web-service project. The method [Project.project(java.lang.String)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:project(java.lang.String)) creates a reference to a specific subproject by path.

[Example: Declaring project dependencies](#ucq4sw)

**build.gradle**

project(':web-service') {  
 dependencies {  
 implementation project(':utils')  
 implementation project(':api')  
 }  
}

[Resolving specific artifacts from a module dependency](#3ecdngp)

Whenever Gradle tries to resolve a module from a Maven or Ivy repository, it looks for a metadata file and the default artifact file, a JAR. The build fails if none of these artifact files can be resolved. Under certain conditions, you might want to tweak the way Gradle resolves artifacts for a dependency.

* The dependency only provides a non-standard artifact without any metadata e.g. a ZIP file.
* The module metadata declares more than one artifact e.g. as part of an Ivy dependency descriptor.
* You only want to download a specific artifact without any of the transitive dependencies declared in the metadata.

Gradle is a polyglot build tool and not limited to just resolving Java libraries. Let’s assume you wanted to build a web application using JavaScript as the client technology. Most projects check in external JavaScript libraries into version control. An external JavaScript library is no different than a reusable Java library so why not download it from a repository instead?

[Google Hosted Libraries](https://developers.google.com/speed/libraries/) is a distribution platform for popular, open-source JavaScript libraries. With the help of the artifact-only notation you can download a JavaScript library file e.g. JQuery. The @ character separates the dependency’s coordinates from the artifact’s file extension.

[Example: Resolving a JavaScript artifact for a declared dependency](#1thnxoi)

**build.gradle**

repositories {  
 ivy {  
 url 'https://ajax.googleapis.com/ajax/libs'  
 layout 'pattern', {  
 artifact '[organization]/[revision]/[module].[ext]'  
 }  
 }  
}  
  
configurations {  
 js  
}  
  
dependencies {  
 js 'jquery:jquery:3.2.1@js'  
}

Some modules ship different "flavors" of the same artifact or they publish multiple artifacts that belong to a specific module version but have a different purpose. It’s common for a Java library to publish the artifact with the compiled class files, another one with just the source code in it and a third one containing the Javadocs.

In JavaScript, a library may exist as uncompressed or minified artifact. In Gradle, a specific artifact identifier is called *classifier*, a term generally used in Maven and Ivy dependency management.

Let’s say we wanted to download the minified artifact of the JQuery library instead of the uncompressed file. You can provide the classifier min as part of the dependency declaration.

[Example: Resolving a JavaScript artifact with classifier for a declared dependency](#4dhbgcb)

**build.gradle**

repositories {  
 ivy {  
 url 'https://ajax.googleapis.com/ajax/libs'  
 layout 'pattern', {  
 artifact '[organization]/[revision]/[module](.[classifier]).[ext]'  
 }  
 }  
}  
  
configurations {  
 js  
}  
  
dependencies {  
 js 'jquery:jquery:3.2.1:min@js'  
}

[Declaring Repositories](#28h4qwu)

Gradle can resolve dependencies from one or many repositories based on Maven, Ivy or flat directory formats. Check out the [full reference on all types of repositories](#19c6y18) for more information.

[Declaring a publicly-available repository](#2smlqk4)

Organizations building software may want to leverage public binary repositories to download and consume open source dependencies. Popular public repositories include Maven Central, Bintray JCenter and the Google Android repository. Gradle provides built-in shortcut methods for the most widely-used repositories.



*Figure 18. Declaring a repository with the help of shortcut methods*

To declare JCenter as repository, add this code to your build script:

[Example: Declaring JCenter repository as source for resolving dependencies](#17rw0rx)

**build.gradle**

repositories {  
 jcenter()  
}

Under the covers Gradle resolves dependencies from the respective URL of the public repository defined by the shortcut method. All shortcut methods are available via the [RepositoryHandler](http://docs.google.com/dsl/org.gradle.api.artifacts.dsl.RepositoryHandler.html) API. Alternatively, you can [spell out the URL of the repository](#3rrjjfq) for more fine-grained control.

[Declaring a custom repository by URL](#3rrjjfq)

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 measure and ensure uptime and availability. Specifying a custom URL is also helpful if you want to declare a less popular, but publicly-available repository.

Add the following code to declare an in-house repository for your build reachable through a custom URL.

[Example: Declaring a custom repository by URL](#26wttnj)

**build.gradle**

repositories {  
 maven {  
 url "http://repo.mycompany.com/maven2"  
 }  
}

Repositories with custom URLs can be specified as Maven or Ivy repositories by calling the corresponding methods available on the [RepositoryHandler](http://docs.google.com/dsl/org.gradle.api.artifacts.dsl.RepositoryHandler.html) API. Gradle supports other protocols than http or https as part of the custom URL e.g. file, sftp or s3. For a full coverage see the [reference manual on supported transport protocols](#4bitr6x).

You can also [define your own repository layout](#2yyq2ca) by using ivy { } repositories as they are very flexible in terms of how modules are organised in a repository.

[Declaring multiple repositories](#m243vc)

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](#19c6y18).

This example demonstrates how to declare various shortcut and custom URL repositories for a project:

[Example: Declaring multiple repositories](#361rmj5)

**build.gradle**

repositories {  
 jcenter()  
 maven {  
 url "https://maven.springframework.org/release"  
 }  
 maven {  
 url "https://maven.restlet.com"  
 }  
}

| **Note** | The order of declaration determines how Gradle will check for dependencies at runtime. If Gradle finds a module descriptor in a particular repository, it will attempt to download all of the artifacts for that module from *the same repository*. You can learn more about the inner workings of [Gradle’s resolution mechanism](#lb6vgn). |
| --- | --- |

[Inspecting Dependencies](#nmf14n)

Gradle provides sufficient tooling to navigate large dependency graphs and mitigate situations that can lead to [dependency hell](https://en.wikipedia.org/wiki/Dependency_hell). Users can chose to render the full graph of dependencies as well as identify the selection reason and origin for a dependency. The origin of a dependency can be a declared dependency in the build script or a transitive dependency in graph plus their corresponding configuration. Gradle offers both capabilities through visual representation via build scans and as command line tooling.

[Listing dependencies in a project](#1l71wqy)

A project can declare one or more dependencies. Gradle can visualize the whole dependency tree for every [configuration](#456pfer) available in the project.

Rendering the dependency tree is particularly useful if you’d like to identify which dependencies have been resolved at runtime. It also provides you with information about any dependency conflict resolution that occurred in the process and clearly indicates the selected version. The dependency report always contains declared and transitive dependencies.

Let’s say you’d want to create tasks for your project that use the [JGit library](https://www.eclipse.org/jgit/) to execute SCM operations e.g. to model a release process. You can declare dependencies for any external tooling with the help of a [custom configuration](#37m2jsg) so that it doesn’t doesn’t pollute other contexts like the compilation classpath for your production source code.

[Example: Declaring the JGit dependency with a custom configuration](#2kbzpmk)

**build.gradle**

repositories {  
 jcenter()  
}  
  
configurations {  
 scm  
}  
  
dependencies {  
 scm 'org.eclipse.jgit:org.eclipse.jgit:4.9.2.201712150930-r'  
}

A [build scan](https://scans.gradle.com/) can visualize dependencies as a navigable, searchable tree. Additional context information can be rendered by clicking on a specific dependency in the graph.



*Figure 19. Dependency tree in a build scan*

Every Gradle project provides the task dependencies to render the so-called *dependency report* from the command line. By default the dependency report renders dependencies for all configurations. To pair down on the information provide the optional parameter --configuration.

[Example: Rendering the dependency report for a custom configuration](#zh9zud)

**Output of** gradle -q dependencies --configuration scm

> gradle -q dependencies --configuration scm  
  
------------------------------------------------------------  
Root project  
------------------------------------------------------------  
  
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.

The dependencies report provides detailed information about the dependencies available in the graph. Any dependency that could not be resolved is marked with FAILED in red color. Dependencies with the same coordinates that can occur multiple times in the graph are omitted and indicated by an asterisk. Dependencies that had to undergo conflict resolution render the requested and selected version separated by a right arrow character.

[Identifying which dependency version was selected and why](#3jgxii6)

Large software projects inevitably deal with an increased number of dependencies either through direct or transitive dependencies. The [dependencies report](#1l71wqy) provides you with the raw list of dependencies but does not explain *why* they have been selected or *which* dependency is responsible for pulling them into the graph.

Let’s have a look at a concrete example. A project may request two different versions of the same dependency either as direct or transitive dependency. Gradle applies version conflict resolution to ensure that only one version of the dependency exists in the dependency graph. In this example the conflicting dependency is represented by commons-codec:commons-codec.

[Example: Declaring the JGit dependency and a conflicting dependency](#1ym7spz)

**build.gradle**

repositories {  
 jcenter()  
}  
  
configurations {  
 scm  
}  
  
dependencies {  
 scm 'org.eclipse.jgit:org.eclipse.jgit:4.9.2.201712150930-r'  
 scm 'commons-codec:commons-codec:1.7'  
}

The dependency tree in a [build scan](https://scans.gradle.com/) renders the selection reason (conflict resolution) as well as the origin of a dependency if you click on a dependency and select the "Required By" tab.



*Figure 20. Dependency insight capabilities in a build scan*

Every Gradle project provides the task dependencyInsight to render the so-called *dependency insight report* from the command line. Given a dependency in the dependency graph you can identify the selection reason and track down the origin of the dependency selection. You can think of the dependency insight report as the inverse representation of the dependency report for a given dependency. When executing the task you have to provide the mandatory parameter --dependency to specify the coordinates of the dependency under inspection. The parameters --configuration and --singlepath are optional but help with filtering the output.

[Example: Using the dependency insight report for a given dependency](#4ilvbds)

**Output of** gradle -q dependencyInsight --dependency commons-codec --configuration scm

> gradle -q dependencyInsight --dependency commons-codec --configuration scm  
commons-codec:commons-codec:1.7  
 variant "default+runtime" [  
 org.gradle.status = release (not requested)  
 ]  
 Selection reasons:  
 - Was requested  
 - 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.

[Justifying dependency declarations with custom reasons](#2xr5lll)

When you declare a [dependency](#3tbugp1) or a [dependency constraint](#2jl2h7v), you can provide a custom reason for the declaration. This makes the dependency declarations in your build script and the dependency insight report easier to interpret.

[Example: Giving a reason for choosing a certain module version in a dependency declaration](#1cwfvte)

**build.gradle**

apply plugin: 'java-library'  
  
repositories {  
 jcenter()  
}  
  
dependencies {  
 implementation('org.ow2.asm:asm:6.0') {  
 because 'we require a JDK 9 compatible bytecode generator'  
 }  
}

[Example: Using the dependency insight report with custom reasons](#3ww3eh7)

**Output of** gradle -q dependencyInsight --dependency asm

> gradle -q dependencyInsight --dependency asm  
org.ow2.asm:asm:6.0  
 variant "compile" [  
 org.gradle.status = release (not requested)  
 org.gradle.usage = java-api  
 ]  
 Selection reasons:  
 - Was requested : we require a JDK 9 compatible bytecode generator  
  
org.ow2.asm:asm:6.0  
\--- compileClasspath  
  
A web-based, searchable dependency report is available by adding the --scan option.

[Managing Dependency Configurations](#37m2jsg)

[What is a configuration?](#456pfer)

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. Gradle represents the scope of a dependency with the help of a [Configuration](http://docs.google.com/dsl/org.gradle.api.artifacts.Configuration.html). Every configuration can be identified by a unique name.

Many Gradle plugins add pre-defined configurations to your project. The Java plugin, for example, adds configurations to represent the various classpaths it needs for source code compilation, executing tests and the like. See [the Java plugin chapter](#2c1dop0) for an example. The sections above demonstrate how to [declare dependencies](#3tbugp1) for different use cases.



*Figure 21. Configurations use declared dependencies for specific purposes*

For more examples on the usage of configurations to navigate, inspect and post-process metadata and artifacts of assigned dependencies, see [Working with Dependencies](#206ipza).

[Defining custom configurations](#r6nywt)

You can define configurations yourself, so-called *custom configurations*. A custom configuration is useful for separating the scope of dependencies needed for a dedicated purpose.

Let’s say you wanted to declare a dependency on the [Jasper Ant task](https://tomcat.apache.org/tomcat-9.0-doc/jasper-howto.html) for the purpose of pre-compiling JSP files that should *not* end up in the classpath for compiling your source code. It’s fairly simple to achieve that goal by introducing a custom configuration and using it in a task.

[Example: Declaring and using a custom configuration](#3b6bhkm)

**build.gradle**

configurations {  
 jasper  
}  
  
repositories {  
 mavenCentral()  
}  
  
dependencies {  
 jasper 'org.apache.tomcat.embed:tomcat-embed-jasper:9.0.2'  
}  
  
task preCompileJsps {  
 doLast {  
 ant.taskdef(classname: 'org.apache.jasper.JspC',  
 name: 'jasper',  
 classpath: configurations.jasper.asPath)  
 ant.jasper(validateXml: false,  
 uriroot: file('src/main/webapp'),  
 outputDir: file("$buildDir/compiled-jsps"))  
 }  
}

A project’s configurations are managed by a configurations object. Configurations have a name and can extend each other. To learn more about this API have a look at [ConfigurationContainer](http://docs.google.com/dsl/org.gradle.api.artifacts.ConfigurationContainer.html).

[Inheriting dependencies from other configurations](#1qblrsf)

A configuration can extend other configurations to form an inheritance hierarchy. Child configurations inherit the whole set of dependencies declared for any of its superconfigurations.

Configuration inheritance is heavily used by Gradle core plugins like the [Java plugin](#2c1dop0). For example the testImplementation configuration extends the implementation configuration. The configuration hierarchy has a practical purpose: compiling tests requires the dependencies of the source code under test on top of the dependencies needed write the test class. A Java project that uses JUnit to write and execute test code also needs Guava if its classes are imported in the production source code.



*Figure 22. Configuration inheritance provided by the Java plugin*

Under the covers the testImplementation and implementation configurations form an inheritance hierarchy by calling the method [Configuration.extendsFrom(org.gradle.api.artifacts.Configuration[])](http://docs.google.com/dsl/org.gradle.api.artifacts.Configuration.html#org.gradle.api.artifacts.Configuration:extendsFrom(org.gradle.api.artifacts.Configuration%5B%5D)). A configuration can extend any other configuration irrespective of its definition in the build script or a plugin.

Let’s say you wanted to write a suite of smoke tests. Each smoke test makes a HTTP call to verify a web service endpoint. As the underlying test framework the project already uses JUnit. You can define a new configuration named smokeTest that extends from the testImplementation configuration to reuse the existing test framework dependency.

[Example: Extending a configuration from another configuration](#4ab9ag8)

**build.gradle**

configurations {  
 smokeTest.extendsFrom testImplementation  
}  
  
dependencies {  
 testImplementation 'junit:junit:4.12'  
 smokeTest 'org.apache.httpcomponents:httpclient:4.5.5'  
}

[Managing Transitive Dependencies](#1mrcu09)

Resolution behavior for transitive dependencies can be customized to a high degree to meet enterprise requirements.

[Managing versions of transitive dependencies with dependency constraints](#2jl2h7v)

Dependency constraints allow you to define the version or the version range of both dependencies declared in the build script and transitive dependencies. It is the preferred method to express constraints that should be applied to all dependencies of a configuration. When Gradle attempts to resolve a dependency to a module version, all [dependency declarations with version](#1od42n1), all transitive dependencies and all dependency constraints for that module are taken into consideration. The highest version that matches all conditions is selected. If no such version is found, Gradle fails with an error showing the conflicting declarations. If this happens you can adjust your dependencies or dependency constraints declarations, or [make other adjustments to the transitive dependencies](#1mrcu09) if needed. Similar to dependency declarations, dependency constraint declarations are [scoped by configurations](#37m2jsg) and can therefore be selectively defined for parts of a build. If a dependency constraint influenced the resolution result, any type of [dependency resolve rules](#2pgjko1) may still be applied afterwards.

[Example: Define dependency constraints](#14ltuvu)

**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'  
 }  
 }  
}

In the example, all versions are omitted from the dependency declaration. Instead, the versions are defined in the constraints block. The version definition for commons-codec:1.11 is only taken into account if commons-codec is brought in as transitive dependency, since commons-codec is not defined as dependency in the project. Otherwise, the constraint has no effect.

| **Note** | Dependency constraints are not yet published, but that will be added in a future release. This means that their use currently only targets builds that do not publish artifacts to maven or ivy repositories. |
| --- | --- |

Dependency constraints themselves can also be added transitively. If a modules’s metadata is defined in a .pom file that contains dependency entries with <optional>true</optional>, Gradle will create a dependency constraint for each of these so-called *optional dependencies*. This leads to a similar resolution behavior as provided by Maven: if the corresponding module is brought in by another, non-optional dependency declaration, then the constraint will apply when choosing the version for that module (e.g., if the optional dependency defines a higher version, that one is chosen).

| **Note** | Support for *optional dependencies* from pom files is active by default with Gradle 5.0+. For using it in Gradle 4.6+, you need to activate it by adding enableFeaturePreview('IMPROVED\_POM\_SUPPORT') in *settings.gradle*. |
| --- | --- |

[Excluding transitive module dependencies](#3olhdjn)

Declared dependencies in a build script can pull in a lot of transitive dependencies. You might decide that you do not want a particular transitive dependency as part of the dependency graph for a good reason.

* The dependency is undesired due to licensing constraints.
* The dependency is not available in any of the declared repositories.
* The metadata for the dependency exists but the artifact does not.
* The metadata provides incorrect coordinates for a transitive dependency.

Transitive dependencies can be excluded on the level of a declared dependency or a configuration. Let’s demonstrate both use cases. In the following two examples the build script declares a dependency on Log4J, a popular logging framework in the Java world. The metadata of the particular version of Log4J also defines transitive dependencies.

[Example: Unresolved artifacts for transitive dependencies](#23qrnrg)

**build.gradle**

apply plugin: 'java'  
  
repositories {  
 mavenCentral()  
}  
  
dependencies {  
 implementation 'log4j:log4j:1.2.15'  
}

If resolved from Maven Central some of the transitive dependencies provide metadata but not the corresponding binary artifact. As a result any task requiring the binary files will fail e.g. a compilation task.

> gradle -q compileJava  
  
\* What went wrong:  
Could not resolve all files for configuration ':compileClasspath'.  
> Could not find jms.jar (javax.jms:jms:1.1).  
 Searched in the following locations:  
 https://repo.maven.apache.org/maven2/javax/jms/jms/1.1/jms-1.1.jar  
> Could not find jmxtools.jar (com.sun.jdmk:jmxtools:1.2.1).  
 Searched in the following locations:  
 https://repo.maven.apache.org/maven2/com/sun/jdmk/jmxtools/1.2.1/jmxtools-1.2.1.jar  
> Could not find jmxri.jar (com.sun.jmx:jmxri:1.2.1).  
 Searched in the following locations:  
 https://repo.maven.apache.org/maven2/com/sun/jmx/jmxri/1.2.1/jmxri-1.2.1.jar

The situation can be fixed by adding a repository containing those dependencies. In the given example project, the source code does not actually use any of Log4J’s functionality that require the JMS (e.g. [JMSAppender](https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/net/JMSAppender.html)) or JMX libraries. It’s safe to exclude them from the dependency declaration.

Exclusions need to spelled out as a key/value pair via the attributes group and/or module. For more information, refer to [ModuleDependency.exclude(java.util.Map)](http://docs.google.com/javadoc/org/gradle/api/artifacts/ModuleDependency.html#exclude-java.util.Map-).

[Example: Excluding transitive dependency for a particular dependency declaration](#iw1xz9)

**build.gradle**

dependencies {  
 implementation('log4j:log4j:1.2.15') {  
 exclude group: 'javax.jms', module: 'jms'  
 exclude group: 'com.sun.jdmk', module: 'jmxtools'  
 exclude group: 'com.sun.jmx', module: 'jmxri'  
 }  
}

You may find that other dependencies will want to pull in the same transitive dependency that misses the artifacts. Alternatively, you can exclude the transitive dependencies for a particular configuration by calling the method [Configuration.exclude(java.util.Map)](http://docs.google.com/dsl/org.gradle.api.artifacts.Configuration.html#org.gradle.api.artifacts.Configuration:exclude(java.util.Map)).

[Example: Excluding transitive dependency for a particular configuration](#32vpgn2)

**build.gradle**

configurations {  
 implementation {  
 exclude group: 'javax.jms', module: 'jms'  
 exclude group: 'com.sun.jdmk', module: 'jmxtools'  
 exclude group: 'com.sun.jmx', module: 'jmxri'  
 }  
}  
  
dependencies {  
 implementation 'log4j:log4j:1.2.15'  
}

| **Note** | As a build script author you often times know that you want to exclude a dependency for all configurations available in the project. You can use the method [DomainObjectCollection.all(org.gradle.api.Action)](http://docs.google.com/javadoc/org/gradle/api/DomainObjectCollection.html#all-org.gradle.api.Action-) to define a global rule. |
| --- | --- |

You might encounter other use cases that don’t quite fit the bill of an exclude rule. For example you want to automatically select a version for a dependency with a specific requested version or you want to select a different group for a requested dependency to react to a relocation. Those use cases are better solved by the [ResolutionStrategy](http://docs.google.com/dsl/org.gradle.api.artifacts.ResolutionStrategy.html) API. Some of these use cases are covered in [Customizing Dependency Resolution Behavior](#111kx3o).

[Enforcing a particular dependency version](#1i0zquv)

Gradle resolves any dependency version conflicts by selecting the latest 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.

| **Note** | Enforcing a version of a dependency requires a conscious decision. Changing the version of a transitive dependency might lead to runtime errors if external libraries do not properly function without them. Consider upgrading your source code to use a newer version of the library as an alternative approach. |
| --- | --- |

Let’s say a project uses the [HttpClient library](https://hc.apache.org/httpcomponents-client-ga/) for performing HTTP calls. HttpClient pulls in [Commons Codec](https://commons.apache.org/proper/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 not available in 1.10 anymore. A dependency version can be enforced by declaring it in the build script and setting [ExternalDependency.setForce(boolean)](http://docs.google.com/javadoc/org/gradle/api/artifacts/ExternalDependency.html#setForce-boolean-) to true.

[Example: Enforcing a dependency version](#420n9io)

**build.gradle**

dependencies {  
 implementation 'org.apache.httpcomponents:httpclient:4.5.4'  
 implementation('commons-codec:commons-codec:1.9') {  
 force = true  
 }  
}

If the project requires a specific version of a dependency on a configuration-level then it can be achieved by calling the method [ResolutionStrategy.force(java.lang.Object[])](http://docs.google.com/dsl/org.gradle.api.artifacts.ResolutionStrategy.html#org.gradle.api.artifacts.ResolutionStrategy:force(java.lang.Object%5B%5D)).

[Example: Enforcing a dependency version on the configuration-level](#2h5xjqh)

**build.gradle**

configurations {  
 compileClasspath {  
 resolutionStrategy.force 'commons-codec:commons-codec:1.9'  
 }  
}  
  
dependencies {  
 implementation 'org.apache.httpcomponents:httpclient:4.5.4'  
}

[Disabling resolution of transitive dependencies](#wb7tya)

By default Gradle resolves all transitive dependencies specified by the dependency metadata. Sometimes this behavior may not be desirable e.g. if the metadata is incorrect or defines a large graph of transitive dependencies. You can tell Gradle to disable transitive dependency management for a dependency by setting [ModuleDependency.setTransitive(boolean)](http://docs.google.com/javadoc/org/gradle/api/artifacts/ModuleDependency.html#setTransitive-boolean-) to true. As a result only the main artifact will be resolved for the declared dependency.

[Example: Disabling transitive dependency resolution for a declared dependency](#3gavcm3)

**build.gradle**

dependencies {  
 implementation('com.google.guava:guava:23.0') {  
 transitive = false  
 }  
}

| **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. |
| --- | --- |

A project can decide to disable transitive dependency resolution completely. You either don’t want to rely on the metadata published to the consumed repositories or you want to gain full control over the dependencies in your graph. For more information, see [Configuration.setTransitive(boolean)](http://docs.google.com/javadoc/org/gradle/api/artifacts/Configuration.html#setTransitive-boolean-).

[Example: Disabling transitive dependency resolution on the configuration-level](#1vg5mtw)

**build.gradle**

configurations.all {  
 transitive = false  
}  
  
dependencies {  
 implementation 'com.google.guava:guava:23.0'  
}

[Importing version recommendations from a Maven BOM](#4fft5hp)

Gradle provides support for importing [bill of materials (BOM) files](https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Importing_Dependencies), which are effectively .pom files that use <dependencyManagement> to control the dependency versions of direct and transitive dependencies. The BOM support in Gradle works similar to using <scope>import</scope> when depending on a BOM in Maven. In Gradle however, it is done via a regular dependency declaration on the BOM:

[Example: Depending on a BOM to import its dependency constraints](#2ul3fpi)

**build.gradle**

dependencies {  
 *// import a BOM*  
 implementation 'org.springframework.boot:spring-boot-dependencies:1.5.8.RELEASE'  
  
 *// define dependencies without versions*  
 implementation 'com.google.code.gson:gson'  
 implementation 'dom4j:dom4j'  
}

In the example, the versions of gson and dom4j are provided by the Spring Boot BOM. This way, if you are developing for a platform like Spring Boot, you do not have to declare any versions yourself but can rely on the versions the platform provides.

Gradle treats all entries in the <dependencyManagement> block of a BOM similar to [Gradle’s dependency constraints](#2jl2h7v). This means that any version defined in the <dependencyManagement> block can impact the dependency resolution result. In order to qualify as a BOM, a .pom file needs to have <packaging>pom</packaging> set.

| **Note** | Importing dependency constraints from Maven BOMs is active by default with Gradle 5.0+. For using it in Gradle 4.6+, you need to activate it by adding enableFeaturePreview('IMPROVED\_POM\_SUPPORT') in *settings.gradle*. |
| --- | --- |

[Dependency Locking](#46r0co2)

Use of dynamic dependency versions (e.g. 1.+ or [1.0,2.0)) makes builds non-deterministic. This causes builds to break without any obvious change, and worse, can be caused by a transitive dependency that the build author has no control over.

To achieve [reproducible builds](https://reproducible-builds.org/), it is necessary to *lock* versions of dependencies and transitive dependencies such that a build with the same inputs will always resolve the same module versions. This is called *dependency locking*.

It enables, amongst others, the following scenarios:

* Companies dealing with multi repositories no longer need to rely on -SNAPSHOT or changing dependencies, which sometimes result in cascading failures when a dependency introduces a bug or incompatibility. Now dependencies can be declared against major or minor version range, enabling to test with the latest versions on CI while leveraging locking for stable developer builds.
* Teams that want to always use the latest of their dependencies can use dynamic versions, locking their dependencies only for releases. The release tag will contain the lock states, allowing that build to be fully reproducible when bug fixes need to be developed.

Locking is enabled per [dependency configuration](#37m2jsg). Once enabled, you must create an initial lock state. It will cause Gradle to verify that resolution results do not change, resulting in the same selected dependencies even if newer versions are produced. Modifications to your build that would impact the resolved set of dependencies will cause it to fail. This makes sure that changes, either in published dependencies or build definitions, do not alter resolution without adapting the lock state.

| **Note** | Dependency locking makes sense only with [dynamic versions](#3mmzoe9). It will have no impact on [changing versions](#30x7rho) (like -SNAPSHOT) whose coordinates remain the same, though the content may change. Gradle will even emit a warning when persisting lock state and changing dependencies are present in the resolution result. |
| --- | --- |

[Enabling locking on configurations](#19qdpxb)

Locking of a configuration happens through the [ResolutionStrategy](http://docs.google.com/dsl/org.gradle.api.artifacts.ResolutionStrategy.html):

[Example: Locking a specific configuration](#3tq18l4)

**build.gradle**

configurations {  
 compileClasspath {  
 resolutionStrategy.activateDependencyLocking()  
 }  
}

Or the following, as a way to lock all configurations:

[Example: Locking all configurations](#28vbisx)

**build.gradle**

dependencyLocking {  
 lockAllConfigurations()  
}

| **Note** | Only configurations that can be resolved will have lock state attached to them. Applying locking on non resolvable-configurations is simply a no-op. |
| --- | --- |

[Generating and updating dependency locks](#o0lt0q)

In order to generate or update lock state, you specify the --write-locks command line argument in addition to the normal tasks that would trigger configurations to be resolved. This will cause the creation of lock state for each resolved configuration in that build execution. Note that if lock state existed previously, it is overwritten.

[Lock all configurations in one build execution](#3809boj)

When locking multiple configurations, you may want to lock them all at once, during a single build execution.

For this, you have two options:

* Run gradle dependencies --write-locks. This will effectively lock all resolvable configurations that have locking enabled. Note that in a multi project setup, dependencies only is executed on *one* project, the root one in this case.
* Declare a custom task that will resolve all configurations

[Example: Resolving all configurations](#1n5jlwc)

**build.gradle**

task resolveAndLockAll {  
 doFirst {  
 **assert** gradle.startParameter.writeDependencyLocks  
 }  
 doLast {  
 configurations.findAll {  
 *// Add any custom filtering on the configurations to be resolved*  
 it.canBeResolved  
 }.each { it.resolve() }  
 }  
}

That second option, with proper choosing of configurations, can be the only option in the native world, where not all configurations can be resolved on a single platform.

[Lock state location and format](#47574k5)

Lock state will be preserved in a file located in the folder gradle/dependency-locks inside the project or subproject directory. Each file is named by the configuration it locks and has the lockfile extension.

The content of the file is a module notation per line, with a header giving some context. Module notations are ordered alphabetically, to ease diffs.

[Example: Lockfile content](#2mahery)

**gradle/dependency-locks/compileClasspath.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  
org.springframework:spring-core:5.0.5.RELEASE  
org.springframework:spring-jcl:5.0.5.RELEASE

which matches the following dependency declaration:

[Example: Dynamic dependency declaration](#11frozr)

**build.gradle**

dependencies {  
 implementation 'org.springframework:spring-beans:[5.0,6.0)'  
}

[Running a build with lock state present](#3lff7nk)

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 as stored in the lock state, regardless if new versions matching the dynamic selector have been produced.

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

[Selectively updating lock state entries](#20kphvd)

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.

❯ gradle classes --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](#4kkd0j6)

1. Make sure that the configuration for which you no longer want locking is not configured with locking.
2. Remove the file matching the configurations where you no longer want locking.

If you only perform the second step above, then locking will effectively no longer be applied. However, if that configuration happens to be resolved in the future at a time where lock state is persisted, it will once again be locked.

[Locking limitations](#2zpnaqz)

* It is currently not possible to lock the [classpath configuration](#1iya7iv) used for script plugins.
* Locking can not yet be applied to source dependencies.

[Nebula locking plugin](#1euxkys)

This feature is inspired by the [Nebula Gradle dependency lock plugin](https://github.com/nebula-plugins/gradle-dependency-lock-plugin).

[Troubleshooting Dependency Resolution](#2lwamvv)

Managing dependencies in a project can be challenging. This chapter describes techniques for troubleshooting issues you might encounter in your project as well as best practices for avoiding common problems.

[Resolving version conflicts](#3yul3ml)

Gradle resolves version conflicts by picking the highest version of a module. [Build scans](https://scans.gradle.com/get-started) and the [dependency insight report](#3jgxii6) are immensely helpful in identifying why a specific version was selected. If the resolution result is not satisfying (e.g. the selected version of a module is too high) or it fails (because you configured [ResolutionStrategy.failOnVersionConflict()](http://docs.google.com/dsl/org.gradle.api.artifacts.ResolutionStrategy.html#org.gradle.api.artifacts.ResolutionStrategy:failOnVersionConflict())) you have the following possibilities to fix it.

* Configuring any dependency (transitive or not) as *forced*. This approach is useful if the dependency in conflict is a transitive dependency. See [Enforcing a particular dependency version](#1i0zquv) for examples.
* Configuring dependency resolution to *prefer modules that are part of your build* (transitive or not). This approach is useful if your build contains custom forks of modules (as part of [multi-project builds](#2p2csry) or as include in [composite builds](#49x2ik5)). See [ResolutionStrategy.preferProjectModules()](http://docs.google.com/dsl/org.gradle.api.artifacts.ResolutionStrategy.html#org.gradle.api.artifacts.ResolutionStrategy:preferProjectModules()) for more information.
* Using [dependency resolve rules](#2pgjko1) for fine-grained control over the version selected for a particular dependency.

[Using dynamic versions and changing modules](#gxk8tv)

There are many situations when you want to use the latest version of a particular module dependency, or the latest in a range of versions. This can be a requirement during development, or you may be developing a library that is designed to work with a range of dependency versions. You can easily depend on these constantly changing dependencies by using a *dynamic version*. A [dynamic version](#3mmzoe9) can be either a version range (e.g. 2.+) or it can be a placeholder for the latest version available e.g. latest.integration.

Alternatively, the module you request can change over time even for the same version, a so-called [changing version](#30x7rho). An example of this type of *changing module* is a Maven SNAPSHOT module, which always points at the latest artifact published. In other words, a standard Maven snapshot is a module that is continually evolving, it is a "changing module".

| **Note** | Using dynamic versions and changing modules can lead to unreproducible builds. As new versions of a particular module are published, its API may become incompatible with your source code. Use this feature with caution! |
| --- | --- |

By default, Gradle caches dynamic versions and changing modules for 24 hours. During that time frame Gradle does not contact any of the declared, remote repositories for new versions. If you want Gradle to check the remote repository more frequently or with every execution of your build, then you will need to change the time to live (TTL) threshold.

| **Note** | Using a short TTL threshold for dynamic or changing versions may result in longer build times due to the increased number of HTTP(s) calls. |
| --- | --- |

You can override the default cache modes using [command line options](#1c1lvlb). You can also [change the cache expiry times in your build programmatically](#2dzvdue) using the resolution strategy.

[Controlling dependency caching programmatically](#2dzvdue)

You can fine-tune certain aspects of caching programmatically using the [ResolutionStrategy](http://docs.google.com/javadoc/org/gradle/api/artifacts/ResolutionStrategy.html) for a configuration. The programmatic approach is useful if you would like to change the settings permanently.

By default, Gradle caches dynamic versions for 24 hours. To change how long Gradle will cache the resolved version for a dynamic version, use:

[Example: Dynamic version cache control](#t55o27)

**build.gradle**

configurations.all {  
 resolutionStrategy.cacheDynamicVersionsFor 10, 'minutes'  
}

By default, Gradle caches changing modules for 24 hours. To change how long Gradle will cache the meta-data and artifacts for a changing module, use:

[Example: Changing module cache control](#3d4t6q0)

**build.gradle**

configurations.all {  
 resolutionStrategy.cacheChangingModulesFor 4, 'hours'  
}

[Controlling dependency caching from the command line](#1c1lvlb)

You can control the behavior of dependency caching for a distinct build invocation from the command line. Command line options are helpful for making a selective, ad-hoc choice for a single execution of the build.

[Avoiding network access with offline mode](#1sa3gxt)

The --offline command line switch tells Gradle to always use dependency modules from the cache, regardless if they are due to be checked again. When running with offline, Gradle will never attempt to access the network to perform dependency resolution. If required modules are not present in the dependency cache, build execution will fail.

[Forcing all dependencies to be re-resolved](#4c9qzlm)

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 perhaps 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 SHA1 values in the repository with the SHA1 values for existing downloaded artifacts.

[Locking dependency versions](#2rf19tf)

The use of [dynamic dependencies](#3mmzoe9) in a build is convenient. The user does not need to know the latest version of a dependency and Gradle automatically uses new versions once they are published. However, dynamic dependencies make builds non-reproducible, as they can resolve to a different version at a later point in time. This makes it hard to reproduce old builds when debugging a problem. It can also disrupt development if a new, but incompatible version is selected. In the best case the CI build catches the problem and someone needs to investigate. In the worst case, the problem makes it to production unnoticed.

Gradle offers [dependency locking](#46r0co2) to solve this problem. The user can run a build asking to persist the resolved versions for every module dependency. This file is then checked in and the versions in it are used on all subsequent runs until the lock is updated or removed again.

[Versioning of file dependencies](#16kbk18)

Legacy projects sometimes prefer to consume [file dependencies](#4025kda) instead of [module dependencies](#397tsf8). File dependencies can point to any file in the filesystem and do not need to adhere a specific naming convention. It is recommended to clearly express the intention and a concrete version for file dependencies. File dependencies are not considered by Gradle’s [version conflict resolution](#lb6vgn). Therefore, it is extremely important to assign a version to the file name to indicate the distinct set of changes shipped with it. For example commons-beanutils-1.3.jar lets you track the changes of the library by the release notes.

As a result, the dependencies of the project are easier to maintain and organize. It’s much easier to uncover potential API incompatibilities by the assigned version.

[Customizing Dependency Resolution Behavior](#111kx3o)

There are a number of ways that you can influence how Gradle resolves dependencies. All of these mechanisms offer an API to define a reason for why they are used. Providing reasons makes dependency resolution results more understandable. If any customization influenced the resolution result, the provided reason will show up in [dependency insight](#3jgxii6) report.

[Using dependency resolve rules](#2pgjko1)

A dependency resolve rule is executed for each resolved dependency, and offers a powerful api for manipulating a requested dependency prior to that dependency being resolved. The feature currently offers the ability to change the group, name and/or version of a requested dependency, allowing a dependency to be substituted with a completely different module during resolution.

Dependency resolve rules provide a very powerful way to control the dependency resolution process, and can be used to implement all sorts of advanced patterns in dependency management. Some of these patterns are outlined below. For more information and code samples see the [ResolutionStrategy](http://docs.google.com/dsl/org.gradle.api.artifacts.ResolutionStrategy.html) class in the API documentation.

[Modelling releasable units](#3qjz2p1)

Often an organisation publishes a set of libraries with a single version; where the libraries are built, tested and published together. These libraries form a "releasable unit", designed and intended to be used as a whole. It does not make sense to use libraries from different releasable units together.

But it is easy for transitive dependency resolution to violate this contract. For example:

* module-a depends on releasable-unit:part-one:1.0
* module-b depends on releasable-unit:part-two:1.1

A build depending on both module-a and module-b will obtain different versions of libraries within the releasable unit.

Dependency resolve rules give you the power to enforce releasable units in your build. Imagine a releasable unit defined by all libraries that have org.gradle group. We can force all of these libraries to use a consistent version:

[Example: Forcing a consistent version for a group of libraries](#25p9cwu)

**build.gradle**

configurations.all {  
 resolutionStrategy.eachDependency { DependencyResolveDetails details ->  
 **if** (details.requested.group == 'org.gradle') {  
 details.useVersion '1.4'  
 details.because 'API breakage in higher versions'  
 }  
 }  
}

[Implementing a custom versioning scheme](#kujn4n)

In some corporate environments, the list of module versions that can be declared in Gradle builds is maintained and audited externally. Dependency resolve rules provide a neat implementation of this pattern:

* In the build script, the developer declares dependencies with the module group and name, but uses a placeholder version, for example: default.
* The default version is resolved to a specific version via a dependency resolve rule, which looks up the version in a corporate catalog of approved modules.

This rule implementation can be neatly encapsulated in a corporate plugin, and shared across all builds within the organisation.

[Example: Using a custom versioning scheme](#34u75sg)

**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']  
}

[Blacklisting a particular version with a replacement](#1jzhg09)

Dependency resolve rules provide a mechanism for blacklisting a particular version of a dependency and providing a replacement version. This can be useful if a certain dependency version is broken and should not be used, where a dependency resolve rule causes this version to be replaced with a known good version. One example of a broken module is one that declares a dependency on a library that cannot be found in any of the public repositories, but there are many other reasons why a particular module version is unwanted and a different version is preferred.

In example below, imagine that version 1.2.1 contains important fixes and should always be used in preference to 1.2. The rule provided will enforce just this: any time version 1.2 is encountered it will be replaced with 1.2.1. Note that this is different from a forced version as described above, in that any other versions of this module would not be affected. This means that the 'newest' conflict resolution strategy would still select version 1.3 if this version was also pulled transitively.

[Example: Blacklisting a version with a replacement](#43z4yo2)

**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'  
 }  
 }  
}

[Substituting a dependency module with a compatible replacement](#2j4f8vv)

At times a completely different module can serve as a replacement for a requested module dependency. Examples include using groovy in place of groovy-all, or using log4j-over-slf4j instead of log4j. You can perform these substitutions using dependency resolve rules:

[Example: Changing dependency group and/or name during resolution](#y9pj3o)

**build.gradle**

configurations.all {  
 resolutionStrategy.eachDependency { DependencyResolveDetails details ->  
 **if** (details.requested.name == 'groovy-all') {  
 details.useTarget group: details.requested.group, name: 'groovy', version: details.requested.version  
 details.because "prefer 'groovy' over 'groovy-all'"  
 }  
 **if** (details.requested.name == 'log4j') {  
 details.useTarget "org.slf4j:log4j-over-slf4j:1.7.10"  
 details.because "prefer 'log4j-over-slf4j' 1.7.10 over any version of 'log4j'"  
 }  
 }  
}

[Using dependency substitution rules](#3i9d1rh)

Dependency substitution rules work similarly to dependency resolve rules. In fact, many capabilities of dependency resolve rules can be implemented with dependency substitution rules. They allow project and module dependencies to be transparently substituted with specified replacements. Unlike dependency resolve rules, dependency substitution rules allow project and module dependencies to be substituted interchangeably.

*Adding a dependency substitution rule to a configuration changes the timing of when that configuration is resolved.* Instead of being resolved on first use, the configuration is instead resolved when the task graph is being constructed. This can have unexpected consequences if the configuration is being further modified during task execution, or if the configuration relies on modules that are published during execution of another task.

To explain:

* A Configuration can be declared as an input to any Task, and that configuration can include project dependencies when it is resolved.
* If a project dependency is an input to a Task (via a configuration), then tasks to build the project artifacts must be added to the task dependencies.
* In order to determine the project dependencies that are inputs to a task, Gradle needs to 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.

In the absence of dependency substitution rules, Gradle knows that an external module dependency will never transitively reference a project dependency. This makes it easy to determine the full set of project dependencies for a configuration through simple graph traversal. With this functionality, Gradle can no longer make this assumption, and must perform a full resolve in order to determine the project dependencies.

[Substituting an external module dependency with a project dependency](#1xenbza)

One use case for dependency substitution is to use a locally developed version of a module in place of one that is downloaded from an external repository. This could be useful for testing a local, patched version of a dependency.

The module to be replaced can be declared with or without a version specified.

[Example: Substituting a module with a project](#4heaun3)

**build.gradle**

configurations.all {  
 resolutionStrategy.dependencySubstitution {  
 substitute module("org.utils:api") with project(":api") because "we work with the unreleased development version"  
 substitute module("org.utils:util:2.5") with project(":util")  
 }  
}

Note that a project that is substituted must be included in the multi-project build (via settings.gradle). Dependency substitution rules take care of replacing the module dependency with the project dependency and wiring up any task dependencies, but do not implicitly include the project in the build.

[Substituting a project dependency with a module replacement](#2wjl4uw)

Another way to use substitution rules is to replace a project dependency with a module in a multi-project build. This can be useful to speed up development with a large multi-project build, by allowing a subset of the project dependencies to be downloaded from a repository rather than being built.

The module to be used as a replacement must be declared with a version specified.

[Example: Substituting a project with a module](#1bovf2p)

**build.gradle**

configurations.all {  
 resolutionStrategy.dependencySubstitution {  
 substitute project(":api") with module("org.utils:api:1.3") because "we use a stable version of utils"  
 }  
}

When a project dependency has been replaced with a module dependency, that project is still included in the overall multi-project build. However, tasks to build the replaced dependency will not be executed in order to build the resolve the depending Configuration.

[Conditionally substituting a dependency](#3voixqi)

A common use case for dependency substitution is to allow more flexible assembly of sub-projects within a multi-project build. This can be useful for developing a local, patched version of an external dependency or for building a subset of the modules within a large multi-project build.

The following example uses a dependency substitution rule to replace any module dependency with the group org.example, but only if a local project matching the dependency name can be located.

[Example: Conditionally substituting a dependency](#2att7yb)

**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  
 }  
 }  
 }  
 }

Note that a project that is substituted must be included in the multi-project build (via settings.gradle). Dependency substitution rules take care of replacing the module dependency with the project dependency, but do not implicitly include the project in the build

[Using component metadata rules](#pz3i64)

Each module has metadata associated with it, such as its group, name, version, dependencies, and so on. This metadata typically originates in the module’s descriptor. Metadata rules allow certain parts of a module’s metadata to be manipulated from within the build script. They take effect after a module’s descriptor has been downloaded, but before it has been selected among all candidate versions. This makes metadata rules another instrument for customizing dependency resolution.

One piece of module metadata that Gradle understands is a module’s *status scheme*. This concept, also known from Ivy, models the different levels of maturity that a module transitions through over time. The default status scheme, ordered from least to most mature status, is integration, milestone, release. Apart from a status scheme, a module also has a (current) *status*, which must be one of the values in its status scheme. If not specified in the (Ivy) descriptor, the status defaults to integration for Ivy modules and Maven snapshot modules, and release for Maven modules that aren’t snapshots.

A module’s status and status scheme are 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, with the default status scheme in place, latest.integration will select the highest module version regardless of its status (because integration is the least mature status), whereas latest.release will select the highest module version with status release. Here is what this looks like in code:

[Example: 'Latest' version selector](#39yr0tx)

**build.gradle**

dependencies {  
 config1 "org.sample:client:latest.integration"  
 config2 "org.sample:client:latest.release"  
}  
  
task listConfigs {  
 doLast {  
 configurations.config1.each { println it.name }  
 println()  
 configurations.config2.each { println it.name }  
 }  
}

**Output of** gradle -q listConfigs

> gradle -q listConfigs  
client-1.5.jar  
  
client-1.4.jar

The next example demonstrates latest selectors based on a custom status scheme declared in a component metadata rule that applies to all modules:

[Example: Custom status scheme](#1p41b1q)

**build.gradle**

**class** **CustomStatusRule** **implements** ComponentMetadataRule {  
 @Override  
 **void** execute(ComponentMetadataContext context) {  
 **def** details = context.details  
 **if** (details.id.group == "org.sample" && details.id.name == "api") {  
 details.statusScheme = ["bronze", "silver", "gold", "platinum"]  
 }  
 }  
}  
  
dependencies {  
 config3 "org.sample:api:latest.silver"  
 components {  
 all(CustomStatusRule)  
 }  
}

Component metadata rules can be applied to a specified module. Modules must be specified in the form of group:module.

[Example: Custom status scheme by module](#493otpj)

**build.gradle**

**class** **ModuleStatusRule** **implements** ComponentMetadataRule {  
 @Override  
 **void** execute(ComponentMetadataContext context) {  
 context.details.statusScheme = ["int", "rc", "prod"]  
 }  
}  
  
dependencies {  
 config4 "org.sample:lib:latest.prod"  
 components {  
 withModule('org.sample:lib', ModuleStatusRule)  
 }  
}

Gradle can also provide to component metadata rules the Ivy-specific metadata for modules resolved from an Ivy repository. Values from the Ivy descriptor are made available via the [IvyModuleDescriptor](http://docs.google.com/javadoc/org/gradle/api/artifacts/ivy/IvyModuleDescriptor.html) interface.

[Example: Ivy component metadata rule](#2o8z3xc)

**build.gradle**

**class** **IvyComponentRule** **implements** ComponentMetadataRule {  
 @Override  
 **void** execute(ComponentMetadataContext context) {  
 **def** descriptor = context.getDescriptor(IvyModuleDescriptor)  
 **if** (descriptor != null && descriptor.branch == 'testing') {  
 context.details.status = "rc"  
 }  
 }  
}  
dependencies {  
 config5 "org.sample:lib:latest.rc"  
 components {  
 withModule("org.sample:lib", IvyComponentRule)  
 }  
}

Note that while any rule can request the [IvyModuleDescriptor](http://docs.google.com/javadoc/org/gradle/api/artifacts/ivy/IvyModuleDescriptor.html), only components sourced from an Ivy repository will have a non-null value for it.

As can be seen in the examples above, component metadata rules are defined by implementing [ComponentMetadataRule](http://docs.google.com/javadoc/org/gradle/api/artifacts/ComponentMetadataRule.html) which has a single execute method receiving an instance of [ComponentMetadataContext](http://docs.google.com/javadoc/org/gradle/api/artifacts/ComponentMetadataContext.html) as parameter.

The next example shows how you can configure the ComponentMetadataRule through an [ActionConfiguration](http://docs.google.com/javadoc/org/gradle/api/ActionConfiguration.html).

[Example: Configuration of ComponentMetadataRule](#13e9e55)

**build.gradle**

**class** **ConfiguredRule** **implements** ComponentMetadataRule {  
 String param  
  
 @javax.inject.Inject  
 ConfiguredRule(String param) {  
 this.param = param  
 }  
  
 @Override  
 **void** execute(ComponentMetadataContext context) {  
 **if** (param == 'sampleValue') {  
 context.details.statusScheme = ["bronze", "silver", "gold", "platinum"]  
 }  
 }  
}  
dependencies {  
 config6 "org.sample:api:latest.gold"  
 components {  
 withModule('org.sample:api', ConfiguredRule, {  
 params('sampleValue')  
 })  
 }  
}

This happens by having a constructor in your implementation of ComponentMetadataRule accepting the parameters that were configured and the services that need injecting.

Gradle enforces isolation of instances of ComponentMetadataRule. This means that all passed in parameters must be Serializable or known Gradle types that can be isolated.

In addition, Gradle services can be injected into your ComponentMetadataRule. This is for the moment limited to the [RepositoryResourceAccessor](http://docs.google.com/javadoc/org/gradle/api/artifacts/repositories/RepositoryResourceAccessor.html). Because of this, the moment you have a constructor, it must be annotated with @javax.inject.Inject.

[Using component selection rules](#3ndwwsy)

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 by rule. 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](http://docs.google.com/dsl/org.gradle.api.artifacts.ComponentSelectionRules.html) object. Each rule configured will be called with a [ComponentSelection](http://docs.google.com/dsl/org.gradle.api.artifacts.ComponentSelection.html) object as an argument which contains information about the candidate version being considered. Calling [ComponentSelection.reject(java.lang.String)](http://docs.google.com/dsl/org.gradle.api.artifacts.ComponentSelection.html#org.gradle.api.artifacts.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.

[Example: Component selection rule](#22j770r)

**build.gradle**

configurations {  
 rejectConfig {  
 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 {  
 rejectConfig "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. 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.

[Example: Component selection rule with module target](#hohh8k)

**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 metadata arguments that can be considered are [ComponentMetadata](http://docs.google.com/javadoc/org/gradle/api/artifacts/ComponentMetadata.html) and [IvyModuleDescriptor](http://docs.google.com/javadoc/org/gradle/api/artifacts/ivy/IvyModuleDescriptor.html).

[Example: Component selection rule with metadata](#31o4zwd)

**build.gradle**

configurations {  
 metadataRulesConfig {  
 resolutionStrategy {  
 componentSelection {  
 *// Reject any versions with a status of 'experimental'*  
 all { ComponentSelection selection, ComponentMetadata metadata ->  
 **if** (selection.candidate.group == 'org.sample' && 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, IvyModuleDescriptor descriptor, ComponentMetadata metadata ->  
 **if** (descriptor.branch != "release" && metadata.status != 'milestone') {  
 selection.reject("'org.sample:api' must have testing branch or milestone status")  
 }  
 }  
 }  
 }  
 }  
}

Note that a [ComponentSelection](http://docs.google.com/dsl/org.gradle.api.artifacts.ComponentSelection.html) argument is *always* required as the first parameter when declaring a component selection rule with additional Ivy metadata parameters, but the metadata parameters can be declared in any order.

Lastly, component selection rules can also be defined using a *rule source* object. A rule source object is any object that contains exactly one method that defines the rule action and is annotated with @Mutate.

This method:

* must return void.
* must have [ComponentSelection](http://docs.google.com/dsl/org.gradle.api.artifacts.ComponentSelection.html) as the first argument.
* may have additional parameters of type [ComponentMetadata](http://docs.google.com/javadoc/org/gradle/api/artifacts/ComponentMetadata.html) and/or [IvyModuleDescriptor](http://docs.google.com/javadoc/org/gradle/api/artifacts/ivy/IvyModuleDescriptor.html).

[Example: Component selection rule using a rule source object](#1gtfa46)

**build.gradle**

**class** **RejectTestBranch** {  
 @Mutate  
 **void** evaluateRule(ComponentSelection selection, IvyModuleDescriptor ivy) {  
 **if** (ivy.branch == "test") {  
 selection.reject("reject test branch")  
 }  
 }  
}  
  
configurations {  
 ruleSourceConfig {  
 resolutionStrategy {  
 componentSelection {  
 all **new** RejectTestBranch()  
 }  
 }  
 }  
}

[Using module replacement rules](#40t2srz)

Module replacement rules allow a build to declare that a legacy library has been replaced by a new one. A good example when a new library replaced a legacy one is the google-collections -> guava migration. The team that created google-collections decided to change the module name from com.google.collections:google-collections into com.google.guava:guava. This is a legal scenario in the industry: teams need to be able to change the names of products they maintain, including the module coordinates. Renaming of the module coordinates has impact on conflict resolution.

To explain the impact on conflict resolution, let’s consider the google-collections -> guava scenario. It may happen that both libraries are pulled into the same dependency graph. For example, *our project* depends on guava but some of *our dependencies* pull in a legacy version of google-collections. This can cause runtime errors, for example during test or application execution. Gradle does not automatically resolve the google-collections -> guava conflict because it is not considered as a *version conflict*. It’s because the module coordinates for both libraries are completely different and conflict resolution is activated when group and module coordinates are the same but there are different versions available in the dependency graph (for more info, refer to the section on conflict resolution). Traditional remedies to this problem are:

* Declare exclusion rule to avoid pulling in google-collections to graph. It is probably the most popular approach.
* Avoid dependencies that pull in legacy libraries.
* Upgrade the dependency version if the new version no longer pulls in a legacy library.
* Downgrade to google-collections. It’s not recommended, just mentioned for completeness.

Traditional approaches work but they are not general enough. For example, an organisation wants to resolve the google-collections -> guava conflict resolution problem in all projects. Starting from Gradle 2.2 it is possible to declare that certain module was replaced by other. This enables organisations to include the information about module replacement in the corporate plugin suite and resolve the problem holistically for all Gradle-powered projects in the enterprise.

[Example: Declaring a module replacement](#2fyd2zs)

**build.gradle**

dependencies {  
 modules {  
 module("com.google.collections:google-collections") {  
 replacedBy("com.google.guava:guava", "google-collections is now part of Guava")  
 }  
 }  
}

For more examples and detailed API, refer to the DSL reference for [ComponentMetadataHandler](http://docs.google.com/javadoc/org/gradle/api/artifacts/dsl/ComponentMetadataHandler.html).

What happens when we declare that google-collections is replaced by guava? Gradle can use this information for conflict resolution. Gradle will consider every version of guava newer/better than any version of google-collections. Also, Gradle will ensure that only guava jar is present in the classpath / resolved file list. Note that if only google-collections appears in the dependency graph (e.g. no guava) Gradle will not eagerly replace it with guava. Module replacement is an information that Gradle uses for resolving conflicts. If there is no conflict (e.g. only google-collections or only guava in the graph) the replacement information is not used.

Currently it is not possible to declare that a given module is replaced by a set of modules. However, it is possible to declare that multiple modules are replaced by a single module.

[Specifying default dependencies for a configuration](#v3nd7l)

A configuration can be configured with default dependencies to be used if no dependencies are explicitly set for the configuration. A primary use case of this functionality is for developing plugins that make use of versioned tools that the user might override. By specifying default dependencies, the plugin can use a default version of the tool only if the user has not specified a particular version to use.

[Example: Specifying default dependencies on a configuration](#3f3avve)

**build.gradle**

configurations {  
 pluginTool {  
 defaultDependencies { dependencies ->  
 dependencies.add(project.dependencies.create("org.gradle:my-util:1.0"))  
 }  
 }  
}

[Enabling Ivy dynamic resolve mode](#1u8l637)

Gradle’s Ivy repository implementations support the equivalent to Ivy’s dynamic resolve mode. Normally, Gradle will use the rev attribute for each dependency definition included in an ivy.xml file. In dynamic resolve mode, Gradle will instead prefer the revConstraint attribute over the rev attribute for a given dependency definition. If the revConstraint attribute is not present, the rev attribute is used instead.

To enable dynamic resolve mode, you need to set the appropriate option on the repository definition. A couple of examples are shown below. Note that dynamic resolve mode is only available for Gradle’s Ivy repositories. It is not available for Maven repositories, or custom Ivy DependencyResolver implementations.

[Example: Enabling dynamic resolve mode](#4e88or0)

**build.gradle**

*// Can enable dynamic resolve mode when you define the repository*  
repositories {  
 ivy {  
 url "http://repo.mycompany.com/repo"  
 resolve.dynamicMode = true  
 }  
}  
  
*// Can use a rule instead to enable (or disable) dynamic resolve mode for all repositories*  
repositories.withType(IvyArtifactRepository) {  
 resolve.dynamicMode = true  
}

[The Dependency Cache](#3l18frh)

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.

The Gradle dependency cache consists of two storage types located under GRADLE\_USER\_HOME/caches:

* A file-based store of downloaded artifacts, including binaries like jars as well as raw downloaded meta-data like POM files and Ivy files. The storage path for a downloaded artifact includes the SHA1 checksum, meaning that 2 artifacts with the same name but different content can easily be cached.
* A binary store of resolved module meta-data, including the results of resolving dynamic versions, module descriptors, and artifacts.

The Gradle cache does not allow the local cache to hide problems and create other mysterious and difficult to debug behavior. Gradle enables reliable and reproducible enterprise builds with a focus on bandwidth and storage efficiency.

[Separate metadata cache](#2tdiyyt)

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](#18it96m)

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](#3sigruf).

Dependency resolution will fail if the required artifacts are not available in any repository specified by the build, even if the local cache has a copy of this artifact which was retrieved from a different repository. Repository independence allows builds to be isolated from each other in an advanced way that no build tool has done before. This is a key feature to create builds that are reliable and reproducible in any environment.

[Artifact reuse](#3sigruf)

Before downloading an artifact, Gradle tries to determine the checksum of the required artifact by downloading the sha file associated with that artifact. If the checksum can be retrieved, an artifact is not downloaded if an artifact already exists with the same id and checksum. If the checksum cannot be retrieved from the remote server, the artifact will be downloaded (and ignored if it matches an existing artifact).

As well as considering artifacts downloaded from a different repository, Gradle will also attempt to reuse artifacts found in the local Maven Repository. If a candidate artifact has been downloaded by Maven, Gradle will use this artifact if it can be verified to match the checksum declared by the remote server.

[Checksum based storage](#27nr228)

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 SHA1 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](#mt1ca1)

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 meta-data store is being read or written, but is released for slow operations such as downloading remote artifacts.

[Cache Cleanup](#36souxu)

Gradle keeps track of which artifacts in the dependency cache are accessed. Using this information, the cache is periodically (at most every 24 hours) scanned for artifacts that have not been used for more than 30 days. Obsolete artifacts are then deleted to ensure the cache does not grow indefinitely.

[Working with Dependencies](#206ipza)

Gradle provides an extensive API for navigating, inspecting and post-processing metadata and artifacts of resolved dependencies.

The main entry point for this functionality is the [Configuration](http://docs.google.com/dsl/org.gradle.api.artifacts.Configuration.html) API. To learn more about the fundamentals of configurations, see [Managing Dependency Configurations](#37m2jsg).

[Iterating over dependencies assigned to a configuration](#1lxz55n)

Sometimes you’ll want to implement logic based on the dependencies declared in the build script of a project e.g. to inspect them in a Gradle plugin. You can iterate over the set of dependencies assigned to a configuration with the help of the method [Configuration.getDependencies()](http://docs.google.com/dsl/org.gradle.api.artifacts.Configuration.html#org.gradle.api.artifacts.Configuration:dependencies). Alternatively, you can also use [Configuration.getAllDependencies()](http://docs.google.com/dsl/org.gradle.api.artifacts.Configuration.html#org.gradle.api.artifacts.Configuration:allDependencies) to include the dependencies declared in [superconfigurations](#1qblrsf). These APIs only return the declared dependencies and do not trigger [dependency resolution](#lb6vgn). Therefore, the dependency sets do not include transitive dependencies. Calling the APIs during the [configuration phase of the build lifecycle](#27jua8u) does not result in a significant performance impact.

[Example: Iterating over the dependencies assigned to a configuration](#45xmntg)

**build.gradle**

task iterateDeclaredDependencies {  
 doLast {  
 DependencySet dependencySet = configurations.scm.dependencies  
  
 dependencySet.each {  
 logger.quiet "$it.group:$it.name:$it.version"  
 }  
 }  
}

[Iterating over artifacts resolved for a module](#2l2wy19)

None of the [dependency reporting](#nmf14n) helps you with inspecting or further processing the underlying, resolved artifacts of a module. A typical use case for accessing the artifacts is to copy them into a specific directory or filter out files of interest based on a specific file extension.

You can iterate over the complete set of artifacts resolved for a module with the help of the method [FileCollection.getFiles()](http://docs.google.com/javadoc/org/gradle/api/file/FileCollection.html#getFiles--). Every file instance returned from the method points to its location in the [dependency cache](#3l18frh). Using this method on a Configuration instance is possible as the interface extends FileCollection.

[Example: Iterating over the artifacts resolved for a module](#1087892)

**build.gradle**

task iterateResolvedArtifacts {  
 dependsOn configurations.scm  
  
 doLast {  
 configurations.scm.each {  
 logger.quiet it.absolutePath  
 }  
 }  
}

| **Note** | Iterating over the artifacts of a module automatically resolves the configuration. A resolved configuration becomes immutable and cannot add or remove dependencies. If needed you can copy a configuration for further modification via [Configuration.copy()](http://docs.google.com/dsl/org.gradle.api.artifacts.Configuration.html#org.gradle.api.artifacts.Configuration:copy()). |
| --- | --- |

[Navigating the dependency graph](#3k7uqwv)

As a plugin developer, you may want to navigate the full graph of dependencies assigned to a configuration e.g. for turning the dependency graph into a visualization. You can access the full graph of dependencies for a configuration with the help of the [ResolutionResult](http://docs.google.com/javadoc/org/gradle/api/artifacts/result/ResolutionResult.html).

The resolution result provides various methods for accessing the resolved and unresolved dependencies. For demonstration purposes the sample code uses [ResolutionResult.getRoot()](http://docs.google.com/javadoc/org/gradle/api/artifacts/result/ResolutionResult.html#getRoot--) to access the root node the resolved dependency graph. Each dependency of this component returns an instance of [ResolvedDependencyResult](http://docs.google.com/javadoc/org/gradle/api/artifacts/result/ResolvedDependencyResult.html) or [UnresolvedDependencyResult](http://docs.google.com/javadoc/org/gradle/api/artifacts/result/UnresolvedDependencyResult.html) providing detailed information about the node.

[Example: Walking the resolved and unresolved dependencies of a configuration](#1zd514o)

**build.gradle**

task walkDependencyGraph(type: DependencyGraphWalk) {  
 dependsOn configurations.scm  
}  
  
**class** **DependencyGraphWalk** **extends** DefaultTask {  
 @TaskAction  
 **void** walk() {  
 Configuration configuration = project.configurations.scm  
 ResolutionResult resolutionResult = configuration.incoming.resolutionResult  
 ResolvedComponentResult root = resolutionResult.root  
 logger.quiet configuration.name  
 traverseDependencies(0, root.dependencies)  
 }  
  
 **private** **void** traverseDependencies(**int** level, Set<? **extends** DependencyResult> results) {  
 **for** (DependencyResult result : results) {  
 **if** (result **instanceof** ResolvedDependencyResult) {  
 ResolvedComponentResult componentResult = result.selected  
 ComponentIdentifier componentIdentifier = componentResult.id  
 String node = calculateIndentation(level) + "- $componentIdentifier.displayName ($componentResult.selectionReason)"  
 logger.quiet node  
 traverseDependencies(level + 1, componentResult.dependencies)  
 } **else** **if** (result **instanceof** UnresolvedDependencyResult) {  
 ComponentSelector componentSelector = result.attempted  
 String node = calculateIndentation(level) + "- $componentSelector.displayName (failed)"  
 logger.quiet node  
 }  
 }  
 }  
  
 **private** String calculateIndentation(**int** level) {  
 ' ' \* level  
 }  
}

[Accessing a module’s metadata file](#4jcsjsh)

As part of the dependency resolution process, Gradle downloads the metadata file of a module and stores it in the dependency cache. Some organizations enforce strong restrictions on accessing repositories outside of internal network. Instead of downloading artifacts, those organizations prefer to provide an "installable" Gradle cache with all artifacts contained in it to fulfill the build’s dependency requirements.

The artifact query API provides access to the raw files of a module. Currently, it allows getting a handle to the metadata file and some selected, additional artifacts (e.g. a JVM-based module’s source and Javadoc files). The main API entry point is [ArtifactResolutionQuery](http://docs.google.com/dsl/org.gradle.api.artifacts.query.ArtifactResolutionQuery.html).

Let’s say you wanted to post-process the metadata file of a Maven module. The group, name and version of the module component serve as input to the artifact resolution query. After executing the query, you get a handle to all components that match the criteria and their underlying files. Additionally, it’s very easy to post-process the metadata file. The example code uses Groovy’s [XmlSlurper](http://docs.groovy-lang.org/latest/html/api/groovy/util/XmlSlurper.html) to ask for POM element values.

[Example: Accessing a Maven module’s metadata artifact](#2yi2u0a)

**build.gradle**

apply plugin: 'java-library'  
  
repositories {  
 mavenCentral()  
}  
  
dependencies {  
 implementation 'com.google.guava:guava:18.0'  
}  
  
task printGuavaMetadata {  
 dependsOn configurations.compileClasspath  
  
 doLast {  
 ArtifactResolutionQuery query = dependencies.createArtifactResolutionQuery()  
 .forModule('com.google.guava', 'guava', '18.0')  
 .withArtifacts(MavenModule, MavenPomArtifact)  
 ArtifactResolutionResult result = query.execute()  
  
 **for**(component **in** result.resolvedComponents) {  
 Set<ArtifactResult> mavenPomArtifacts = component.getArtifacts(MavenPomArtifact)  
 ArtifactResult guavaPomArtifact = mavenPomArtifacts.find { it.file.name == 'guava-18.0.pom' }  
 **def** xml = **new** XmlSlurper().parse(guavaPomArtifact.file)  
 println guavaPomArtifact.file  
 println xml.name  
 println xml.description  
 }  
 }  
}

[Publishing Artifacts](#4k668n3)

[Publishing](#2zbgiuw)

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](#1dnd483) to publish
2. Define [where](#3xn0mvw) to publish it to
3. [Do](#2csax3p) the publishing

Each of the 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.

| **Note** | Looking for information on upload tasks and the archives configuration? See the [Legacy Publishing](#2dlolyb) chapter. |
| --- | --- |

Gradle makes it easy to publish to these types of repository by providing some prepackaged infrastructure in the form of the [Maven Publish Plugin](#1egqt2p) and the [Ivy Publish Plugin](#3ygebqi). These plugins allow you to configure what to publish and perform the publishing with a minimum of effort.



*Figure 23. 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](#rxl7bi) and metadata that Gradle calls a [publication](#3bx8pzb). 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 — plus a POM file describing 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.

Where to publish

Gradle needs to know where to publish artifacts so that consumers can get hold of them. This is done via [repositories](#3iq0a3h), 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](http://docs.google.com/dsl/org.gradle.api.publish.maven.tasks.PublishToMavenRepository.html), while for Ivy repositories the tasks are of type [PublishToIvyRepository](http://docs.google.com/dsl/org.gradle.api.publish.ivy.tasks.PublishToIvyRepository.html).

What follows is a practical example that demonstrates the entire publishing process.

[Setting up basic publishing](#1r2j074)

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](#1egqt2p)
* [Ivy Publish Plugin](#3ygebqi)

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: Applying the necessary plugins](#4b26iux)

**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](http://docs.google.com/dsl/org.gradle.api.publish.PublishingExtension.html):

[Example: Configuring a Java library for publishing](#2q7gt2q)

**build.gradle**

group = 'org.example'  
version = '1.0'  
  
publishing {  
 publications {  
 myLibrary(MavenPublication) {  
 from components.java  
 }  
 }  
  
 repositories {  
 maven {  
 name = 'myRepo'  
 url = "file://${buildDir}/repo"  
 }  
 }  
}

This defines a publication called "myLibrary" that can be published to a Maven repository by virtue of its type: [MavenPublication](http://docs.google.com/dsl/org.gradle.api.publish.maven.MavenPublication.html). This publication consists of just the production JAR artifact and its metadata, which combined are represented by the java [component](#15cr3aj) 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](http://docs.google.com/javadoc/org/gradle/api/component/SoftwareComponent.html), 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 publish*PubName*PublicationTo*RepoName*Repository. 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](http://docs.google.com/dsl/org.gradle.api.publish.ivy.IvyPublication.html), 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.

[Adding custom artifacts to a publication](#3pcelyc)

Users often need to include additional artifacts with a publication, one of the most common examples being that of "-sources" and "-javadoc" JARs for JVM libraries. This is easy to do for both Maven- and Ivy-compatible repositories via the artifact configuration.

The following sample configures "-sources" and "-javadoc" JARs for a Java project and attaches them to the main (Maven) publication, i.e. the production JAR:

[Example: Adding an additional archive artifact to a MavenPublication](#24how65)

**build.gradle**

task sourcesJar(type: Jar) {  
 classifier = 'sources'  
 from sourceSets.main.allJava  
}  
  
task javadocJar(type: Jar) {  
 classifier = 'javadoc'  
 from javadoc.destinationDir  
}  
  
publishing {  
 publications {  
 mavenJava(MavenPublication) {  
 from components.java  
  
 artifact sourcesJar  
 artifact javadocJar  
 }  
 }  
}

There are several important things to note about the sample:

* The artifact() method accepts archive tasks as an argument — like sourcesJar in the sample — as well as any type of argument accepted by [Project.file(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:file(java.lang.Object)), such as a File instance or string file path.
* 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](#1egqt2p) and the [Ivy Publish Plugin](#3ygebqi).
* 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. That’s why the above sample does not specify those properties in the artifact configurations.

When you’re attaching extra artifacts to a publication, remember that they are *secondary* artifacts that support a *primary* artifact. The metadata that a publication defines — such as dependency information — is associated with that primary artifact only. Thinking about publications in this way should help you determine whether you should be adding custom artifacts to an existing publication, or defining a new publication.

[Publishing a custom primary artifact (no component)](#jmz6dy)

If your build produces a primary artifact that isn’t supported by a predefined component, then you will need to configure a custom artifact. This isn’t much different to adding a custom artifact to an existing publication. There are just a couple of extra considerations:

* You may want to make the artifact available to other projects in the build
* You will need to manually construct the necessary metadata for publishing

Inter-project dependencies have nothing to do with publishing, but both features typically apply to the same set of artifacts in a Gradle project. So how do you tie them together?

You start by defining a custom artifact and attaching it to a Gradle [configuration](#33mmp1r) 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 archives configuration:

[Example: Defining a custom artifact for a configuration](#1irwz9k)

**build.gradle**

**def** rpmFile = file("$buildDir/rpms/my-package.rpm")  
**def** rpmArtifact = artifacts.add('archives', rpmFile) {  
 type 'rpm'  
 builtBy 'rpm'  
}

The artifacts.add() method — from [ArtifactHandler](http://docs.google.com/dsl/org.gradle.api.artifacts.dsl.ArtifactHandler.html) — returns an artifact object of type [PublishArtifact](http://docs.google.com/javadoc/org/gradle/api/artifacts/PublishArtifact.html) that can then be used in defining a publication, as shown in the following sample:

[Example: Attaching a custom PublishArtifact to a publication](#42rkhxd)

**build.gradle**

publishing {  
 publications {  
 maven(MavenPublication) {  
 artifact rpmArtifact  
 }  
 }  
}

Now you can publish the RPM as well as depend on it from another project using the project(path: ':my-project', configuration: 'archives') syntax.

| **Note** | There is currently no easy way to define dependency information for a custom artifact. |
| --- | --- |

The groupId and artifactId properties are specific to Maven publications. See [IvyPublication](http://docs.google.com/dsl/org.gradle.api.publish.ivy.IvyPublication.html) for the relevant Ivy properties.

[Signing artifacts](#2hwus56)

The [Signing Plugin](#3cqmetx) can be used to sign all artifacts and metadata files that make up a publication, including Maven POM files and Ivy module desciptors. In order to use it:

1. Apply the Signing Plugin
2. Configure the [signatory credentials](#x252cz) — 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: Signing a publication](#3h1sl0s)

**build.gradle**

signing {  
 sign publishing.publications.mavenJava  
}

This will create a Sign task for each publication you specify and wire all publish*PubName*PublicationTo*RepoName*Repository 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](#1w72v8l)

**Output of gradle publish**

> gradle publish  
> Task :generatePomFileForMavenJavaPublication  
> Task :compileJava  
> Task :processResources  
> Task :classes  
> Task :jar  
> Task :javadoc  
> Task :javadocJar  
> Task :sourcesJar  
> Task :signMavenJavaPublication  
> Task :publishMavenJavaPublicationToMavenRepository  
> Task :publish  
  
BUILD SUCCESSFUL in 0s  
9 actionable tasks: 9 executed

[Restricting publications to specific repositories](#4g6qdwe)

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: Adding multiple publications and repositories](#2vc0o47)

**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 = "$buildDir/repos/external"  
 }  
 maven {  
 name = 'internal'  
 url = "$buildDir/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(org.gradle.api.specs.Spec)](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:onlyIf(org.gradle.api.specs.Spec)) method. The following sample demonstrates how to implement the constraints we just mentioned:

[Example: Configuring which artifacts should be published to which repositories](#1ahayc0)

**build.gradle**

tasks.withType(PublishToMavenRepository) {  
 onlyIf {  
 (repository == publishing.repositories.external &&  
 publication == publishing.publications.binary) ||  
 (repository == publishing.repositories.internal &&  
 publication == publishing.publications.binaryAndSources)  
 }  
}  
tasks.withType(PublishToMavenLocal) {  
 onlyIf {  
 publication == publishing.publications.binaryAndSources  
 }  
}

**Output of gradle publish**

> gradle publish  
> Task :generatePomFileForBinaryAndSourcesPublication  
> Task :compileJava  
> Task :processResources  
> Task :classes  
> Task :jar  
> Task :sourcesJar  
> Task :publishBinaryAndSourcesPublicationToExternalRepository SKIPPED  
> Task :publishBinaryAndSourcesPublicationToInternalRepository  
> Task :generatePomFileForBinaryPublication  
> Task :publishBinaryPublicationToExternalRepository  
> Task :publishBinaryPublicationToInternalRepository SKIPPED  
> Task :publish  
  
BUILD SUCCESSFUL in 0s  
8 actionable tasks: 8 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: Defining your own shorthand tasks for publishing](#3ugygzt)

**build.gradle**

task 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)](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskCollection.html#withType-java.lang.Class-) with the [PublishToMavenRepository](http://docs.google.com/dsl/org.gradle.api.publish.maven.tasks.PublishToMavenRepository.html) task type. You can do the same with [PublishToIvyRepository](http://docs.google.com/dsl/org.gradle.api.publish.ivy.tasks.PublishToIvyRepository.html) if you’re publishing to Ivy-compatible repositories.

[Configuring publishing tasks](#29m8r7m)

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 generatePomFileFor*PubName*Publication tasks write their POM files. You can do this by using the [TaskCollection.withType(java.lang.Class)](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskCollection.html#withType-java.lang.Class-) method, as demonstrated by this sample:

[Example: Configuring a dynamically named task created by the publishing plugins](#orj1ff)

**build.gradle**

tasks.withType(GenerateMavenPom).all {  
 **def** matcher = name =~ /generatePomFileFor(\w+)Publication/  
 **def** publicationName = matcher[0][1]  
 destination = "$buildDir/poms/${publicationName}-pom.xml"  
}

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.

[Terminology](#38r6k38)

Artifact

A file or directory produced by a build, such as a JAR, a ZIP distribution, or a native executable.

Artifacts are typically designed to be used or consumed by users or other projects, or deployed to hosting systems. In such cases, the artifact is a single file. Directories are common in the case of inter-project dependencies to avoid the cost of producing the publishable artifact.

Component

Any single version of a [module](#265wl8u).

Components are defined by plugins and provide a simple way to define a publication for publishing. They comprise one or more [artifacts](#rxl7bi) as well as the appropriate metadata. For example, the java component consists of the production JAR — produced by the jar task — and its dependency information.

Configuration

A named collection of [dependencies](#35aue4g) or [artifacts](#rxl7bi).

Gradle’s configurations can be somewhat confusing because they apply to both dependencies and artifacts. The main difference is that dependencies are consumed by the project, while artifacts are produced by it. Even then, the artifacts produced by a project are often consumed as dependencies by other projects.

Configurations allow different aspects of the build to work with known subsets of a project’s dependencies or artifacts, e.g. the dependencies required for compilation, or the artifacts related to a project’s API.

Publication

A description of the files and metadata that should be published to a repository as a single entity for use by consumers.

A publication has a name and consists of one or more artifacts plus information about those artifacts. The nature of that information depends on what type of repository you publish the publication to. In the case of Maven, the information takes the form of a POM.

One thing to bear in mind is that Maven repositories only allow a single *primary* artifact, i.e. one with metadata, but they do allow *secondary* artifacts such as packages of the associated source files and documentation ("-sources" and "-javadoc" JARs in the Java world).

[Maven Publish Plugin](#1egqt2p)

The Maven Publish Plugin provides the ability to publish build artifacts to an [Apache Maven](http://maven.apache.org/) repository. A module published to a Maven repository can be consumed by Maven, Gradle (see [Declaring Dependencies](#3tbugp1)) and other tools that understand the Maven repository format. You can learn about the fundamentals of publishing in [Publishing Overview](#2zbgiuw).

[Usage](#1nwgub1)

To use the Maven Publish Plugin, include the following in your build script:

[Example: Applying the Maven Publish Plugin](#47w4cyu)

**build.gradle**

plugins {  
 id 'maven-publish'  
}

The Maven Publish Plugin uses an extension on the project named publishing of type [PublishingExtension](http://docs.google.com/dsl/org.gradle.api.publish.PublishingExtension.html). This extension provides a container of named publications and a container of named repositories. The Maven Publish Plugin works with [MavenPublication](http://docs.google.com/dsl/org.gradle.api.publish.maven.MavenPublication.html) publications and [MavenArtifactRepository](http://docs.google.com/dsl/org.gradle.api.artifacts.repositories.MavenArtifactRepository.html) repositories.

[Tasks](#2n1en6n)

generatePomFileFor*PubName*Publication — [GenerateMavenPom](http://docs.google.com/dsl/org.gradle.api.publish.maven.tasks.GenerateMavenPom.html)

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*.

publish*PubName*PublicationTo*RepoName*Repository — [PublishToMavenRepository](http://docs.google.com/dsl/org.gradle.api.publish.maven.tasks.PublishToMavenRepository.html)

Publishes the *PubName* publication to the repository named *RepoName*. If you have a repository definition without an explicit name, *RepoName* will be "Maven".

publish*PubName*PublicationToMavenLocal — [PublishToMavenLocal](http://docs.google.com/javadoc/org/gradle/api/publish/maven/tasks/PublishToMavenLocal.html)

Copies the *PubName* publication to the local Maven cache — typically *$USER\_HOME/.m2/repository* — along with the publication’s POM file and other metadata.

publish

*Depends on*: All publish*PubName*PublicationTo*RepoName*Repository 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 publish*PubName*PublicationToMavenLocal tasks

Copies all defined publications to the local Maven cache, including their metadata (POM files, etc.).

[Publications](#126oxeg)

This plugin provides [publications](#3bx8pzb) of type [MavenPublication](http://docs.google.com/dsl/org.gradle.api.publish.maven.MavenPublication.html). To learn how to define and use publications, see the section on [basic publishing](#1r2j074).

There are four main things you can configure in a Maven publication:

* A [component](#15cr3aj) — via [MavenPublication.from(org.gradle.api.component.SoftwareComponent)](http://docs.google.com/dsl/org.gradle.api.publish.maven.MavenPublication.html#org.gradle.api.publish.maven.MavenPublication:from(org.gradle.api.component.SoftwareComponent)).
* [Custom artifacts](#3pcelyc) — via the [MavenPublication.artifact(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.publish.maven.MavenPublication.html#org.gradle.api.publish.maven.MavenPublication:artifact(java.lang.Object)) method. See [MavenArtifact](http://docs.google.com/dsl/org.gradle.api.publish.maven.MavenArtifact.html) 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)](http://docs.google.com/dsl/org.gradle.api.publish.maven.MavenPublication.html#org.gradle.api.publish.maven.MavenPublication:pom(org.gradle.api.Action)).

You can see all of these in action in the [complete publishing example](#3m6cg29). The API documentation for MavenPublication has additional code samples.

[Identity values in the generated POM](#21bmqa2)

The attributes of the generated POM file will contain identity values derived from the following project properties:

* groupId - [Project.getGroup()](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:group)
* artifactId - [Project.getName()](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:name)
* version - [Project.getVersion()](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:version)

Overriding the default identity values is easy: simply specify the groupId, artifactId or version attributes when configuring the [MavenPublication](http://docs.google.com/dsl/org.gradle.api.publish.maven.MavenPublication.html).

[Example: customizing the publication identity](#4lba8xv)

**build.gradle**

publishing {  
 publications {  
 maven(MavenPublication) {  
 groupId = 'org.gradle.sample'  
 artifactId = 'project1-sample'  
 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](#30gkj5o)

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](http://docs.google.com/dsl/org.gradle.api.publish.maven.MavenPom.html) 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: Customizing the POM file](#1flutdh)

**build.gradle**

publishing {  
 publications {  
 mavenJava(MavenPublication) {  
 pom {  
 name = 'My Library'  
 description = 'A concise description of my library'  
 url = 'http://www.example.com/library'  
 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](#3zlic1a)

This plugin provides [repositories](#3iq0a3h) of type [MavenArtifactRepository](http://docs.google.com/dslorg.gradle.api.artifacts.repositories.MavenArtifactRepository.html). To learn how to define and use repositories for publishing, see the section on [basic publishing](#1r2j074).

Here’s a simple example of defining a publishing repository:

[Example: Declaring repositories to publish to](#2eqsm93)

**build.gradle**

publishing {  
 repositories {  
 maven {  
 *// change to point to your repo, e.g. http://my.org/repo*  
 url = "$buildDir/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](http://docs.google.com/dslorg.gradle.api.artifacts.repositories.MavenArtifactRepository.html) for more details.

[Snapshot and release repositories](#tw2wgw)

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: Configuring repository URL based on project version](#3dvqf4p)

**build.gradle**

publishing {  
 repositories {  
 maven {  
 **def** releasesRepoUrl = "$buildDir/repos/releases"  
 **def** snapshotsRepoUrl = "$buildDir/repos/snapshots"  
 url = version.endsWith('SNAPSHOT') ? snapshotsRepoUrl : releasesRepoUrl  
 }  
 }  
}

Similarly, you can use a [project or system property](#35nkun2) 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: Configuring repository URL based on project property](#1t10pci)

**build.gradle**

publishing {  
 repositories {  
 maven {  
 **def** releasesRepoUrl = "$buildDir/repos/releases"  
 **def** snapshotsRepoUrl = "$buildDir/repos/snapshots"  
 url = project.hasProperty('release') ? releasesRepoUrl : snapshotsRepoUrl  
 }  
 }  
}

[Publishing to Maven Local](#4d0o80b)

For integration with a local Maven installation, it is sometimes useful to publish the module into the Maven local repository (typically at *$USER\_HOME/.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](http://docs.google.com/javadoc/org/gradle/api/publish/maven/tasks/PublishToMavenLocal.html) task for each [MavenPublication](http://docs.google.com/dsl/org.gradle.api.publish.maven.MavenPublication.html) in the publishing.publications container. The task name follows the pattern of publish*PubName*PublicationToMavenLocal. Each of these tasks is wired into the publishToMavenLocal aggregate task. You do not need to have mavenLocal() in your publishing.repositories section.

[Complete example](#3m6cg29)

The following example demonstrates how to sign and publish a Java library including sources, Javadoc, and a customized POM:

[Example: Publishing a Java library](#2s5yi84)

**build.gradle**

plugins {  
 id 'java-library'  
 id 'maven-publish'  
 id 'signing'  
}  
  
group = 'com.example'  
version = '1.0'  
  
task sourcesJar(type: Jar) {  
 from sourceSets.main.allJava  
 classifier = 'sources'  
}  
  
task javadocJar(type: Jar) {  
 from javadoc  
 classifier = 'javadoc'  
}  
  
publishing {  
 publications {  
 mavenJava(MavenPublication) {  
 artifactId = 'my-library'  
 from components.java  
 artifact sourcesJar  
 artifact javadocJar  
 pom {  
 name = 'My Library'  
 description = 'A concise description of my library'  
 url = 'http://www.example.com/library'  
 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 = "$buildDir/repos/releases"  
 **def** snapshotsRepoUrl = "$buildDir/repos/snapshots"  
 url = version.endsWith('SNAPSHOT') ? snapshotsRepoUrl : releasesRepoUrl  
 }  
 }  
}  
  
signing {  
 sign publishing.publications.mavenJava  
}  
  
  
javadoc {  
 **if**(JavaVersion.current().isJava9Compatible()) {  
 options.addBooleanOption('html4', 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](#3cqmetx) is used to generate a signature file for each artifact. In addition, checksum files will be generated for all artifacts and signature files.

[Removal of deferred configuration behavior](#17b8sfx)

| **Note** | Gradle 5.0 will change the behavior of the publishing {} block. Read on to find out how you can make your build compatible today. |
| --- | --- |

Prior to Gradle 4.8, the publishing {} block was implicitly treated as if all the logic inside it was executed after the project is evaluated. This caused quite a bit of confusion, because it was the only block that behaved that way. As part of the stabilization effort in Gradle 4.8, we are deprecating this behavior and asking all users to migrate their build.

The new, stable behavior can be switched on by adding the following to your settings file:

enableFeaturePreview('STABLE\_PUBLISHING')

We recommend doing a test run with a local repository to see whether all artifacts still have the expected coordinates. In most cases everything should work as before and you are done.

If the coordinates change unexpectedly, 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:

subprojects {  
 publishing {  
 publications {  
 mavenJava {  
 from components.java  
 artifactId = jar.baseName  
 }  
 }  
 }  
}

This kind of logic must be wrapped in an afterEvaluate {} block to make it work going forward.

subprojects {  
 publishing {  
 publications {  
 mavenJava {  
 from components.java  
 afterEvaluate {  
 artifactId = jar.baseName  
 }  
 }  
 }  
 }  
}

[Ivy Publish Plugin](#3ygebqi)

The Ivy Publish Plugin provides the ability to publish build artifacts in the [Apache Ivy](http://ant.apache.org/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](#3tbugp1)) and other tools that understand the Ivy format. You can learn about the fundamentals of publishing in [Publishing Overview](#2zbgiuw).

[Usage](#3rawb3q)

To use the Ivy Publish Plugin, include the following in your build script:

[Example: Applying the Ivy Publish Plugin](#26g6lbj)

**build.gradle**

plugins {  
 id 'ivy-publish'  
}

The Ivy Publish Plugin uses an extension on the project named publishing of type [PublishingExtension](http://docs.google.com/dsl/org.gradle.api.publish.PublishingExtension.html). This extension provides a container of named publications and a container of named repositories. The Ivy Publish Plugin works with [IvyPublication](http://docs.google.com/dsl/org.gradle.api.publish.ivy.IvyPublication.html) publications and [IvyArtifactRepository](http://docs.google.com/dsl/org.gradle.api.artifacts.repositories.IvyArtifactRepository.html) repositories.

[Tasks](#llgvjc)

generateDescriptorFileFor*PubName*Publication — [GenerateIvyDescriptor](http://docs.google.com/dsl/org.gradle.api.publish.ivy.tasks.GenerateIvyDescriptor.html)

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*.

publish*PubName*PublicationTo*RepoName*Repository — [PublishToIvyRepository](http://docs.google.com/dsl/org.gradle.api.publish.ivy.tasks.PublishToIvyRepository.html)

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 publish*PubName*PublicationTo*RepoName*Repository tasks

An aggregate task that publishes all defined publications to all defined repositories.

[Publications](#35l4e75)

This plugin provides [publications](#3bx8pzb) of type [IvyPublication](http://docs.google.com/dsl/org.gradle.api.publish.ivy.IvyPublication.html). To learn how to define and use publications, see the section on [basic publishing](#1r2j074).

There are four main things you can configure in an Ivy publication:

* A [component](#15cr3aj) — via [IvyPublication.from(org.gradle.api.component.SoftwareComponent)](http://docs.google.com/dsl/org.gradle.api.publish.ivy.IvyPublication.html#org.gradle.api.publish.ivy.IvyPublication:from(org.gradle.api.component.SoftwareComponent)).
* [Custom artifacts](#3pcelyc) — via the [IvyPublication.artifact(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.publish.ivy.IvyPublication.html#org.gradle.api.publish.ivy.IvyPublication:artifact(java.lang.Object)) method. See [IvyArtifact](http://docs.google.com/dsl/org.gradle.api.publish.ivy.IvyArtifact.html) 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)](http://docs.google.com/dsl/org.gradle.api.publish.ivy.IvyPublication.html#org.gradle.api.publish.ivy.IvyPublication:descriptor(org.gradle.api.Action)).

You can see all of these in action in the [complete publishing example](#1kqeoey). The API documentation for IvyPublication has additional code samples.

[Identity values for the published project](#44q272r)

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()](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:group)
* module - [Project.getName()](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:name)
* revision - [Project.getVersion()](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:version)
* status - [Project.getStatus()](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:status)
* branch - (not set)

Overriding the default identity values is easy: simply specify the organisation, module or revision properties when configuring the [IvyPublication](http://docs.google.com/dsl/org.gradle.api.publish.ivy.IvyPublication.html). status and branch can be set via the descriptor property — see [IvyModuleDescriptorSpec](http://docs.google.com/dsl/org.gradle.api.publish.ivy.IvyModuleDescriptorSpec.html).

The descriptor property can also be used to add additional custom elements as children of the <info> element, like so:

[Example: customizing the publication identity](#2jvchak)

**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](#z0mrid)

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](http://docs.google.com/dsl/org.gradle.api.publish.ivy.IvyModuleDescriptorSpec.html) 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: Customizing the module descriptor file](#3j0aa66)

**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'  
 }  
 }  
 }  
}

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)](http://docs.google.com/dsl/org.gradle.api.publish.ivy.IvyModuleDescriptorSpec.html#org.gradle.api.publish.ivy.IvyModuleDescriptorSpec:withXml(org.gradle.api.Action)), but you can not 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. |
| --- | --- |

[Repositories](#1y5kkdz)

This plugin provides [repositories](#3iq0a3h) of type [IvyArtifactRepository](http://docs.google.com/dsl/org.gradle.api.artifacts.repositories.IvyArtifactRepository.html). To learn how to define and use repositories for publishing, see the section on [basic publishing](#1r2j074).

Here’s a simple example of defining a publishing repository:

[Example: Declaring repositories to publish to](#4i5831s)

**build.gradle**

publishing {  
 repositories {  
 ivy {  
 *// change to point to your repo, e.g. http://my.org/repo*  
 url = "$buildDir/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](http://docs.google.com/dsl/org.gradle.api.artifacts.repositories.IvyArtifactRepository.html) for more details.

[Complete example](#1kqeoey)

The following example demonstrates publishing with a multi-project build. Each project publishes a Java component and a configured additional source artifact. The descriptor file is customized to include the project description for each project.

[Example: Publishing a Java module](#2xaid9l)

**build.gradle**

subprojects {  
 apply plugin: 'java'  
 apply plugin: 'ivy-publish'  
  
 version = '1.0'  
 group = 'org.gradle.sample'  
  
 repositories {  
 mavenCentral()  
 }  
 task sourcesJar(type: Jar) {  
 from sourceSets.main.java  
 classifier = 'sources'  
 }  
}  
  
project(':project1') {  
 description = 'The first project'  
  
 dependencies {  
 compile 'junit:junit:4.12', project(':project2')  
 }  
}  
  
project(':project2') {  
 description = 'The second project'  
  
 dependencies {  
 compile 'commons-collections:commons-collections:3.2.2'  
 }  
}  
  
subprojects {  
 publishing {  
 repositories {  
 ivy {  
 *// change to point to your repo, e.g. http://my.org/repo*  
 url = "${rootProject.buildDir}/repo"  
 }  
 }  
 publications {  
 ivy(IvyPublication) {  
 from components.java  
 artifact(sourcesJar) {  
 type = 'sources'  
 conf = 'compile'  
 }  
 descriptor.description {  
 text = description  
 }  
 }  
 }  
 }  
}

The result is that the following artifacts will be published for each project:

* The Ivy module descriptor file: ivy-1.0.xml.
* The primary JAR artifact for the Java component: project1-1.0.jar.
* The source JAR artifact that has been explicitly configured: project1-1.0-source.jar.

When project1 is published, the module descriptor (i.e. the ivy.xml file) that is produced will look like:

[Example: Generated ivy.xml](#1cfsnhe)

**output-ivy.xml**

*<!-- This file is an example of the Ivy module descriptor that this build will produce -->*  
<?xml version="1.0" encoding="UTF-8"?>  
<ivy-module version="2.0" xmlns:m="http://ant.apache.org/ivy/maven">  
 <info organisation="org.gradle.sample" module="project1" revision="1.0" status="integration" publication="«PUBLICATION-TIME-STAMP»">  
 <description>The first project</description>  
 </info>  
 <configurations>  
 <conf name="compile" visibility="public"/>  
 <conf name="default" visibility="public" extends="compile,runtime"/>  
 <conf name="runtime" visibility="public"/>  
 </configurations>  
 <publications>  
 <artifact name="project1" type="sources" ext="jar" conf="compile" m:classifier="sources"/>  
 <artifact name="project1" type="jar" ext="jar" conf="compile"/>  
 </publications>  
 <dependencies>  
 <dependency org="junit" name="junit" rev="4.12" conf="compile->default"/>  
 <dependency org="org.gradle.sample" name="project2" rev="1.0" conf="compile->default"/>  
 </dependencies>  
</ivy-module>

| **Tip** | Note that «PUBLICATION-TIME-STAMP» in this example Ivy module descriptor will be the timestamp of when the descriptor was generated. |
| --- | --- |

[Legacy publishing](#2dlolyb)

| **Note** | This chapter describes the *original* publishing mechanism available in Gradle 1.0, which has since been superseded by [an alternative model](#2zbgiuw). The approach detailed in this chapter — based on [Upload](http://docs.google.com/dsl/org.gradle.api.tasks.Upload.html) tasks — should not be used in new builds. We cover it in order to help users work with and update existing builds that use it. |
| --- | --- |

[Introduction](#3wfg657)

This chapter is about how you declare the outgoing artifacts of your project, and how to work with them (e.g. upload them). We define the artifacts of the projects as the files the project provides to the outside world. This might be a library or a ZIP distribution or any other file. A project can publish as many artifacts as it wants.

[Artifacts and configurations](#4cqe7xm)

Like dependencies, artifacts are grouped by configurations. In fact, a configuration can contain both artifacts and dependencies at the same time.

For each configuration in your project, Gradle provides the tasks upload*ConfigurationName* and build*ConfigurationName* when the [base plugin](#vx1227) is applied. Execution of these tasks will build or upload the artifacts belonging to the respective configuration.

[This listing](#2bkqgd0) shows the configurations added by the Java plugin. Two of the configurations are relevant for the usage with artifacts. The archives configuration is the standard configuration to assign your artifacts to. The Java plugin automatically assigns the default jar to this configuration. We will talk more about the runtime configuration [further on](#qq0qkt). As with dependencies, you can declare as many custom configurations as you like and assign artifacts to them.

[Declaring artifacts](#3apo98m)

[Archive task artifacts](#1puyjgf)

You can use an archive task to define an artifact:

[Example: Defining an artifact using an archive task](#49um248)

**build.gradle**

task myJar(type: Jar)  
  
artifacts {  
 archives myJar  
}

It is important to note that the custom archives you are creating as part of your build are not automatically assigned to any configuration. You have to explicitly do this assignment.

[File artifacts](#2ozwcc1)

You can also use a file to define an artifact:

[Example: Defining an artifact using a file](#1456mju)

**build.gradle**

**def** someFile = file('build/somefile.txt')  
  
artifacts {  
 archives someFile  
}

Gradle will figure out the properties of the artifact based on the name of the file. You can customize these properties:

[Example: Customizing an artifact](#3o4u57n)

**build.gradle**

task myTask(type: MyTaskType) {  
 destFile = file('build/somefile.txt')  
}  
  
artifacts {  
 archives(myTask.destFile) {  
 name 'my-artifact'  
 type 'text'  
 builtBy myTask  
 }  
}

There is a map-based syntax for defining an artifact using a file. The map must include a file entry that defines the file. The map may include other artifact properties:

[Example: Map syntax for defining an artifact using a file](#23a4ffg)

**build.gradle**

task generate(type: MyTaskType) {  
 destFile = file('build/somefile.txt')  
}  
  
artifacts {  
 archives file: generate.destFile, name: 'my-artifact', type: 'text', builtBy: generate  
}

[Publishing artifacts](#ifepn9)

We have said that there is a specific upload task for each configuration. Before you can do an upload, you have to configure the upload task and define where to publish the artifacts to. The repositories you have defined (as described in [Declaring Repositories](#28h4qwu)) are not automatically used for uploading. In fact, some of those repositories only allow downloading artifacts, not uploading. Here is an example of how you can configure the upload task of a configuration:

[Example: Configuration of the upload task](#32f28b2)

**build.gradle**

repositories {  
 flatDir {  
 name "fileRepo"  
 dirs "repo"  
 }  
}  
  
uploadArchives {  
 repositories {  
 add project.repositories.fileRepo  
 ivy {  
 credentials {  
 username "username"  
 password "pw"  
 }  
 url "http://repo.mycompany.com"  
 }  
 }  
}

As you can see, you can either use a reference to an existing repository or create a new repository.

If an upload repository is defined with multiple patterns, Gradle must choose a pattern to use for uploading each file. By default, Gradle will upload to the pattern defined by the url parameter, combined with the optional layout parameter. If no url parameter is supplied, then Gradle will use the first defined artifactPattern for uploading, or the first defined ivyPattern for uploading Ivy files, if this is set.

Uploading to a Maven repository is described in [this section](#1hkciiv).

[More about project libraries](#qq0qkt)

If your project is supposed to be used as a library, you need to define what are the artifacts of this library and what are the dependencies of these artifacts. The Java plugin adds a runtime configuration for this purpose, with the implicit assumption that the runtime dependencies are the dependencies of the artifact you want to publish. Of course this is fully customizable. You can add your own custom configuration or let the existing configurations extend from other configurations. You might have a different group of artifacts which have a different set of dependencies. This mechanism is very powerful and flexible.

If someone wants to use your project as a library, she simply needs to declare which configuration of the dependency to depend on. A Gradle dependency offers the configuration property to declare this. If this is not specified, the default configuration is used (see [Managing Dependency Configurations](#37m2jsg)). Using your project as a library can either happen from within a multi-project build or by retrieving your project from a repository. In the latter case, an ivy.xml descriptor in the repository is supposed to contain all the necessary information. If you work with Maven repositories you don’t have the flexibility as described above. For how to publish to a Maven repository, see the section [Uploading to Maven repositories](#1hkciiv).

[Maven Plugin](#sqyw64)

| **Note** | This chapter describes deploying artifacts to Maven repositories using the *original* publishing mechanism available in Gradle 1.0: in Gradle 1.3 a new mechanism for publishing was introduced. This new mechanism introduces some new concepts and features that make Gradle publishing even more powerful and is now the preferred option for publishing artifacts.  You can read about the new publishing plugins in [Publishing Ivy](#3ygebqi) and [Publishing Maven](#1egqt2p). |
| --- | --- |

The Maven plugin adds support for deploying artifacts to Maven repositories.

[Usage](#41k016o)

To use the Maven plugin, include the following in your build script:

[Example: Using the Maven plugin](#2gpabeh)

**build.gradle**

apply plugin: 'maven'

[Tasks](#vuklma)

The Maven plugin defines the following tasks:

install — [Upload](http://docs.google.com/dsl/org.gradle.api.tasks.Upload.html)

*Depends on*: All tasks that build the associated archives.

Installs the associated artifacts to the local Maven cache, including Maven metadata generation. By default the install task is associated with the archives configuration. This configuration has by default only the default jar as an element. To learn more about installing to the local repository, see [Installing to the local repository](#3fu84a3)

[Dependency management](#1uziehw)

The Maven plugin does not define any dependency configurations.

[Convention properties](#4ez5x5p)

The Maven plugin defines the following convention properties:

mavenPomDir — File

The directory where the generated POMs are written to. *Default value*: *${project.buildDir}*/poms

conf2ScopeMappings — [Conf2ScopeMappingContainer](http://docs.google.com/javadoc/org/gradle/api/artifacts/maven/Conf2ScopeMappingContainer.html)

Instructions for mapping Gradle configurations to Maven scopes. See [Dependency mapping](#2u4g7di).

These properties are provided by a [MavenPluginConvention](http://docs.google.com/dsl/org.gradle.api.plugins.MavenPluginConvention.html) convention object.

[Convention methods](#199qhlb)

The maven plugin provides a factory method for creating a POM. This is useful if you need a POM without the context of uploading to a Maven repo.

[Example: Creating a standalone pom.](#3t9e094)

**build.gradle**

task writeNewPom {  
 doLast {  
 pom {  
 project {  
 inceptionYear '2008'  
 licenses {  
 license {  
 name 'The Apache Software License, Version 2.0'  
 url 'http://www.apache.org/licenses/LICENSE-2.0.txt'  
 distribution 'repo'  
 }  
 }  
 }  
 }.writeTo("$buildDir/newpom.xml")  
 }  
}

Amongst other things, Gradle supports the same builder syntax as polyglot Maven. To learn more about the Gradle Maven POM object, see [MavenPom](http://docs.google.com/javadoc/org/gradle/api/artifacts/maven/MavenPom.html). See also: [MavenPluginConvention](http://docs.google.com/dsl/org.gradle.api.plugins.MavenPluginConvention.html)

[Interacting with Maven repositories](#1hkciiv)

[Introduction](#28eoagx)

With Gradle you can deploy to remote Maven repositories or install to your local Maven repository. This includes all Maven metadata manipulation and works also for Maven snapshots. In fact, Gradle’s deployment is 100 percent Maven compatible as we use the native Maven Ant tasks under the hood.

Deploying to a Maven repository is only half the fun if you don’t have a POM. Fortunately Gradle can generate this POM for you using the dependency information it has.

[Deploying to a Maven repository](#njykoq)

Let’s assume your project produces just the default jar file. Now you want to deploy this jar file to a remote Maven repository.

[Example: Upload of file to remote Maven repository](#37jm3cj)

**build.gradle**

apply plugin: 'maven'  
  
uploadArchives {  
 repositories {  
 mavenDeployer {  
 repository(url: "file://localhost/tmp/myRepo/")  
 }  
 }  
}

That is all. Calling the uploadArchives task will generate the POM and deploys the artifact and the POM to the specified repository.

There is more work to do if you need support for protocols other than file. In this case the native Maven code we delegate to needs additional libraries. Which libraries are needed depends on what protocol you plan to use. The available protocols and the corresponding libraries are listed in [Protocol JARs for Maven deployment](#1mowdkc) (those libraries have transitive dependencies which have transitive dependencies).[[8](#46ojw85)] For example, to use the ssh protocol you can do:

[Example: Upload of file via SSH](#2ltu6fy)

**build.gradle**

configurations {  
 deployerJars  
}  
  
repositories {  
 mavenCentral()  
}  
  
dependencies {  
 deployerJars "org.apache.maven.wagon:wagon-ssh:2.2"  
}  
  
uploadArchives {  
 repositories.mavenDeployer {  
 configuration = configurations.deployerJars  
 repository(url: "scp://repos.mycompany.com/releases") {  
 authentication(userName: "me", password: "myPassword")  
 }  
 }  
}

There are many configuration options for the Maven deployer. The configuration is done via a Groovy builder. All the elements of this tree are Java beans. To configure the simple attributes you pass a map to the bean elements. To add bean elements to its parent, you use a closure. In the example above *repository* and *authentication* are such bean elements. [Configuration elements of Maven deployer](#10z4gnr) lists the available bean elements and a link to the Javadoc of the corresponding class. In the Javadoc you can see the possible attributes you can set for a particular element.

In Maven you can define repositories and optionally snapshot repositories. If no snapshot repository is defined, releases and snapshots are both deployed to the repository element. Otherwise snapshots are deployed to the snapshotRepository element.

Table 8. Protocol jars for Maven deployment

| **Protocol** | **Library** |
| --- | --- |
| http | org.apache.maven.wagon:wagon-http:2.2 |
| ssh | org.apache.maven.wagon:wagon-ssh:2.2 |
| ssh-external | org.apache.maven.wagon:wagon-ssh-external:2.2 |
| ftp | org.apache.maven.wagon:wagon-ftp:2.2 |
| webdav | org.apache.maven.wagon:wagon-webdav:1.0-beta-2 |
| file | - |

Table 9. Configuration elements of the MavenDeployer

| **Element** | **Javadoc** |
| --- | --- |
| root | [MavenDeployer](http://docs.google.com/javadoc/org/gradle/api/artifacts/maven/MavenDeployer.html) |
| repository | [org.apache.maven.artifact.ant.RemoteRepository](http://www.docjar.com/docs/api/org/apache/maven/artifact/ant/RemoteRepository.html) |
| authentication | [org.apache.maven.artifact.ant.Authentication](http://www.docjar.com/docs/api/org/apache/maven/artifact/ant/Authentication.html) |
| releases | [org.apache.maven.artifact.ant.RepositoryPolicy](http://www.docjar.com/docs/api/org/apache/maven/artifact/ant/RepositoryPolicy.html) |
| snapshots | [org.apache.maven.artifact.ant.RepositoryPolicy](http://www.docjar.com/docs/api/org/apache/maven/artifact/ant/RepositoryPolicy.html) |
| proxy | [org.apache.maven.artifact.ant.Proxy](http://www.docjar.com/docs/api/org/apache/maven/artifact/ant/Proxy.html) |
| snapshotRepository | [org.apache.maven.artifact.ant.RemoteRepository](http://www.docjar.com/docs/api/org/apache/maven/artifact/ant/RemoteRepository.html) |

[Installing to the local repository](#3fu84a3)

The Maven plugin adds an install task to your project. This task depends on all the archives task of the archives configuration. It installs those archives to your local Maven repository. If the default location for the local repository is redefined in a Maven settings.xml, this is considered by this task.

[Maven POM generation](#3kyrzbk)

When deploying an artifact to a Maven repository, Gradle automatically generates a POM for it. The groupId, artifactId, version and packaging elements used for the POM default to the values shown in the table below. The dependency elements are created from the project’s dependency declarations.

Table 10. Default Values for Maven POM generation

| **Maven Element** | **Default Value** |
| --- | --- |
| groupId | project.group |
| artifactId | uploadTask.repositories.mavenDeployer.pom.artifactId (if set) or archiveTask.baseName. |
| version | project.version |
| packaging | archiveTask.extension |

Here, uploadTask and archiveTask refer to the tasks used for uploading and generating the archive, respectively (for example uploadArchives and jar). archiveTask.baseName defaults to project.archivesBaseName which in turn defaults to project.name.

| **Note** | When you set the “archiveTask.baseName” property to a value other than the default, you’ll also have to set uploadTask.repositories.mavenDeployer.pom.artifactId to the same value. Otherwise, the project at hand may be referenced with the wrong artifact ID from generated POMs for other projects in the same build. |
| --- | --- |

Generated POMs can be found in <buildDir>/poms. They can be further customized via the [MavenPom](http://docs.google.com/javadoc/org/gradle/api/artifacts/maven/MavenPom.html) API. For example, you might want the artifact deployed to the Maven repository to have a different version or name than the artifact generated by Gradle. To customize these you can do:

[Example: Customization of pom](#20429jd)

**build.gradle**

uploadArchives {  
 repositories {  
 mavenDeployer {  
 repository(url: "file://localhost/tmp/myRepo/")  
 pom.version = '1.0Maven'  
 pom.artifactId = 'myMavenName'  
 }  
 }  
}

To add additional content to the POM, the pom.project builder can be used. With this builder, any element listed in the [Maven POM reference](http://maven.apache.org/pom.html) can be added.

[Example: Builder style customization of pom](#4k3ps76)

**build.gradle**

uploadArchives {  
 repositories {  
 mavenDeployer {  
 repository(url: "file://localhost/tmp/myRepo/")  
 pom.project {  
 licenses {  
 license {  
 name 'The Apache Software License, Version 2.0'  
 url 'http://www.apache.org/licenses/LICENSE-2.0.txt'  
 distribution 'repo'  
 }  
 }  
 }  
 }  
 }  
}

Note: groupId, artifactId, version, and packaging should always be set directly on the pom object.

[Example: Modifying auto-generated content](#2z902ez)

**build.gradle**

**def** installer = install.repositories.mavenInstaller  
**def** deployer = uploadArchives.repositories.mavenDeployer  
  
[installer, deployer]\*.pom\*.whenConfigured {pom ->  
 pom.dependencies.find {dep -> dep.groupId == 'group3' && dep.artifactId == 'runtime' }.optional = true  
}

If you have more than one artifact to publish, things work a little bit differently. See [Multiple artifacts per project](#1eeacms).

To customize the settings for the Maven installer (see [Installing to the local repository](#3fu84a3)), you can do:

[Example: Customization of Maven installer](#3ydxval)

**build.gradle**

install {  
 repositories.mavenInstaller {  
 pom.version = '1.0Maven'  
 pom.artifactId = 'myName'  
 }  
}

[Multiple artifacts per project](#1eeacms)

Maven can only deal with one artifact per project. This is reflected in the structure of the Maven POM. We think there are many situations where it makes sense to have more than one artifact per project. In such a case you need to generate multiple POMs. In such a case you have to explicitly declare each artifact you want to publish to a Maven repository. The [MavenDeployer](http://docs.google.com/javadoc/org/gradle/api/artifacts/maven/MavenDeployer.html) and the MavenInstaller both provide an API for this:

[Example: Generation of multiple poms](#2dj85ie)

**build.gradle**

uploadArchives {  
 repositories {  
 mavenDeployer {  
 repository(url: "file://localhost/tmp/myRepo/")  
 addFilter('api') {artifact, file ->  
 artifact.name == 'api'  
 }  
 addFilter('service') {artifact, file ->  
 artifact.name == 'service'  
 }  
 pom('api').version = 'mySpecialMavenVersion'  
 }  
 }  
}

You need to declare a filter for each artifact you want to publish. This filter defines a boolean expression for which Gradle artifact it accepts. Each filter has a POM associated with it which you can configure. To learn more about this have a look at [PomFilterContainer](http://docs.google.com/javadoc/org/gradle/api/artifacts/maven/PomFilterContainer.html) and its associated classes.

[Dependency mapping](#2u4g7di)

The Maven plugin configures the default mapping between the Gradle configurations added by the Java and War plugin and the Maven scopes. Most of the time you don’t need to touch this and you can safely skip this section. The mapping works like the following. You can map a configuration to one and only one scope. Different configurations can be mapped to one or different scopes. You can also assign a priority to a particular configuration-to-scope mapping. Have a look at [Conf2ScopeMappingContainer](http://docs.google.com/javadoc/org/gradle/api/artifacts/maven/Conf2ScopeMappingContainer.html) to learn more. To access the mapping configuration you can say:

[Example: Accessing a mapping configuration](#soifq7)

**build.gradle**

task mappings {  
 doLast {  
 println conf2ScopeMappings.mappings  
 }  
}

Gradle exclude rules are converted to Maven excludes if possible. Such a conversion is possible if in the Gradle exclude rule the group as well as the module name is specified (as Maven needs both in contrast to Ivy). Per-configuration excludes are also included in the Maven POM, if they are convertible.

[The Signing Plugin](#3cqmetx)

The Signing Plugin adds the ability to digitally sign built files and artifacts. These digital signatures can then be used to prove who built the artifact the signature is attached to as well as other information such as when the signature was generated.

The Signing Plugin currently only provides support for generating [OpenPGP signatures](https://en.wikipedia.org/wiki/Pretty_Good_Privacy#OpenPGP) (which is the signature format [required for publication to the Maven Central Repository](http://central.sonatype.org/pages/requirements.html#sign-files-with-gpgpgp)).

[Usage](#3co5ye0)

To use the Signing Plugin, include the following in your build script:

[Example: Using the Signing Plugin](#1rtg8lt)

**build.gradle**

plugins {  
 id 'signing'  
}

[Signatory credentials](#x252cz)

In order to create OpenPGP signatures, you will need a key pair (instructions on creating a key pair using the [GnuPG tools](https://www.gnupg.org/) can be found in the [GnuPG HOWTOs](https://www.gnupg.org/documentation/howtos.html)). You need to provide the Signing Plugin with your key information, which means three things:

* The public key ID (The last 8 symbols of the keyId. You can use gpg -K to get it).
* The absolute path to the secret key ring file containing your private key. (Since gpg 2.1, you need to export the keys with command gpg --keyring secring.gpg --export-secret-keys > ~/.gnupg/secring.gpg).
* The passphrase used to protect your private key.

These items must be supplied as the values of the signing.keyId, signing.secretKeyRingFile, and signing.password properties, respectively.

| **Note** | Given the personal and private nature of these values, a good practice is to store them in the gradle.properties file in the user’s Gradle home directory (described in [System properties](#2b6jogx)) instead of in the project directory itself. |
| --- | --- |

signing.keyId=24875D73  
signing.password=secret  
signing.secretKeyRingFile=/Users/me/.gnupg/secring.gpg

If specifying this information (especially signing.password) in the user gradle.properties file is not feasible for your environment, you can source the information however you need to and set the project properties manually.

**import** org.gradle.plugins.signing.Sign  
  
gradle.taskGraph.whenReady { taskGraph ->  
 **if** (taskGraph.allTasks.any { it **instanceof** Sign }) {  
 *// Use Java 6's console to read from the console (no good for*  
 *// a CI environment)*  
 Console console = System.console()  
 console.printf "\n\nWe have to sign some things in this build." +  
 "\n\nPlease enter your signing details.\n\n"  
  
 **def** id = console.readLine("PGP Key Id: ")  
 **def** file = console.readLine("PGP Secret Key Ring File (absolute path): ")  
 **def** password = console.readPassword("PGP Private Key Password: ")  
  
 allprojects { ext."signing.keyId" = id }  
 allprojects { ext."signing.secretKeyRingFile" = file }  
 allprojects { ext."signing.password" = password }  
  
 console.printf "\nThanks.\n\n"  
 }  
}

Note that the presence of a null value for any these three properties will cause an exception.

[Using OpenPGP subkeys](#4bt3r9m)

OpenPGP supports subkeys, which are like the normal keys, except they’re bound to a master key pair. One feature of OpenPGP subkeys is that they can be revoked independently of the master keys which makes key management easier. A practical case study of how subkeys can be leveraged in software development can be read on the [Debian wiki](https://wiki.debian.org/Subkeys).

The Signing Plugin supports OpenPGP subkeys out of the box. Just specify a subkey ID as the value in the signing.keyId property.

[Using gpg-agent](#2qye1hf)

By default the Signing Plugin uses a Java-based implementation of PGP for signing. This implementation cannot use the gpg-agent program for managing private keys, though. If you want to use the gpg-agent, you can change the signatory implementation used by the Signing Plugin:

[Example: Sign with GnuPG](#163obp8)

**build.gradle**

signing {  
 useGpgCmd()  
 sign configurations.archives  
}

This tells the Signing Plugin to use the GnupgSignatory instead of the default [PgpSignatory](http://docs.google.com/javadoc/org/gradle/plugins/signing/signatory/pgp/PgpSignatory.html). The GnupgSignatory relies on the gpg2 program to sign the artifacts. Of course, this requires that GnuPG is installed.

Without any further configuration the gpg2 (on Windows: gpg2.exe) executable found on the PATH will be used. The password is supplied by the gpg-agent and the default key is used for signing.

[Gnupg signatory configuration](#3q3bud1)

The GnupgSignatory supports a number of configuration options for controlling how gpg is invoked. These are typically set in gradle.properties:

[Example: Configure the GnupgSignatory](#258m4ku)

**gradle.properties**

signing.gnupg.executable=gpg  
signing.gnupg.useLegacyGpg=true  
signing.gnupg.homeDir=gnupg-home  
signing.gnupg.optionsFile=gnupg-home/gpg.conf  
signing.gnupg.keyName=24875D73  
signing.gnupg.passphrase=gradle

signing.gnupg.executable

The gpg executable that is invoked for signing. The default value of this property depends on useLegacyGpg. If that is true then the default value of executable is "gpg" otherwise it is "gpg2".

signing.gnupg.useLegacyGpg

Must be true if GnuPG version 1 is used and false otherwise. The default value of the property is false.

signing.gnupg.homeDir

Sets the home directory for GnuPG. If not given the default home directory of GnuPG is used.

signing.gnupg.optionsFile

Sets a custom options file for GnuPG. If not given GnuPG’s default configuration file is used.

signing.gnupg.keyName

The id of the key that should be used for signing. If not given then the default key configured in GnuPG will be used.

signing.gnupg.passphrase

The passphrase for unlocking the secret key. If not given then the gpg-agent program is used for getting the passphrase.

All configuration properties are optional.

[Specifying what to sign](#kdwesn)

As well as configuring how things are to be signed (i.e. the signatory configuration), you must also specify what is to be signed. The Signing Plugin provides a DSL that allows you to specify the tasks and/or configurations that should be signed.

[Signing Publications](#34djxgg)

When publishing artifacts, you often want to sign them so the consumer of your artifacts can verify their signature. For example, the [Java plugin](#2w5ecyt) defines a component that you can use to define a publication to a Maven (or Ivy) repository using the [Maven Publish Plugin](#1egqt2p) (or the [Ivy Publish Plugin](#3ygebqi), respectively). Using the Signing DSL, you can specify that all of the artifacts of this publication should be signed.

[Example: Signing a publication](#1jiu7o9)

**build.gradle**

signing {  
 sign publishing.publications.mavenJava  
}

This will create a task (of type [Sign](http://docs.google.com/dsl/org.gradle.plugins.signing.Sign.html)) in your project named signMavenJavaPublication that will build all artifacts that are part of the publication (if needed) and then generate signatures for them. The signature files will be placed alongside the artifacts being signed.

[Example: Signing a publication output](#43ihqc2)

**Output of** gradle signMavenJavaPublication

> gradle signMavenJavaPublication  
> Task :generatePomFileForMavenJavaPublication  
> Task :compileJava  
> Task :processResources  
> Task :classes  
> Task :jar  
> Task :javadoc  
> Task :javadocJar  
> Task :sourcesJar  
> Task :signMavenJavaPublication  
  
BUILD SUCCESSFUL in 0s  
8 actionable tasks: 8 executed

In addition, the above DSL allows to sign multiple comma-separated publications. Alternatively, you may specify publishing.publications to sign all publications, or use publishing.publications.matching { … } to sign all publications that match the specified predicate.

[Signing Configurations](#2ins0jv)

It is common to want to sign the artifacts of a configuration. For example, the [Java plugin](#2w5ecyt) configures a jar to build and this jar artifact is added to the archives configuration. Using the Signing DSL, you can specify that all of the artifacts of this configuration should be signed.

[Example: Signing a configuration](#xt2aro)

**build.gradle**

signing {  
 sign configurations.archives  
}

This will create a task (of type [Sign](http://docs.google.com/dsl/org.gradle.plugins.signing.Sign.html)) in your project named signArchives, that will build any archives artifacts (if needed) and then generate signatures for them. The signature files will be placed alongside the artifacts being signed.

[Example: Signing a configuration output](#3hsptfh)

**Output of** gradle signArchives

> gradle signArchives  
> Task :compileJava  
> Task :processResources  
> Task :classes  
> Task :jar  
> Task :signArchives  
  
BUILD SUCCESSFUL in 0s  
4 actionable tasks: 4 executed

[Signing Tasks](#1wy03na)

In some cases the artifact that you need to sign may not be part of a configuration. In this case you can directly sign the task that produces the artifact to sign.

[Example: Signing a task](#4gxnmb3)

**build.gradle**

task stuffZip (type: Zip) {  
 baseName = "stuff"  
 from "src/stuff"  
}  
  
signing {  
 sign stuffZip  
}

This will create a task (of type [Sign](http://docs.google.com/dsl/org.gradle.plugins.signing.Sign.html)) in your project named signStuffZip, that will build the input task’s archive (if needed) and then sign it. The signature file will be placed alongside the artifact being signed.

[Example: Signing a task output](#2w2xwiw)

**Output of** gradle signStuffZip

> gradle signStuffZip  
> Task :stuffZip  
> Task :signStuffZip  
  
BUILD SUCCESSFUL in 0s  
2 actionable tasks: 2 executed

For a task to be *signable*, it must produce an archive of some type, i.e. it must extend [AbstractArchiveTask](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.AbstractArchiveTask.html). Tasks that do this are the [Tar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Tar.html), [Zip](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Zip.html), [Jar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Jar.html), [War](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.War.html) and [Ear](http://docs.google.com/dsl/org.gradle.plugins.ear.Ear.html) tasks.

[Conditional Signing](#1b886qp)

A common usage pattern is to require the signing of build artifacts only under certain conditions. For example, you may not need to sign artifacts for non-release versions. To achieve this, you can specify the condition as an argument of the required() method.

[Example: Specifying when signing is required](#3v7vpei)

**build.gradle**

version = '1.0-SNAPSHOT'  
ext.isReleaseVersion = !version.endsWith("SNAPSHOT")  
  
signing {  
 required { isReleaseVersion && gradle.taskGraph.hasTask("uploadArchives") }  
 sign configurations.archives  
}

In this example, we only want to require signing if we are building a release version and we are going to publish it. Because we are inspecting the task graph to determine if we are going to be publishing, we must set the signing.required property to a closure to defer the evaluation. See [SigningExtension.setRequired(java.lang.Object)](http://docs.google.com/javadoc/org/gradle/plugins/signing/SigningExtension.html#setRequired-java.lang.Object-) for more information.

If the required condition does not hold true, artifacts will only be signed if signatory credentials are configured. Alternatively, you may want to skip signing entirely whether or not signatory credentials are available. If so, you can configure the [Sign](http://docs.google.com/dsl/org.gradle.plugins.signing.Sign.html) tasks to be skipped, for example by attaching a predicate using the onlyIf() method shown in the following example:

[Example: Specifying when signing is skipped](#2ad5zmb)

**build.gradle**

tasks.withType(Sign) {  
 onlyIf { isReleaseVersion }  
}

[Publishing the signatures](#pig9u4)

When signing [publications](#34djxgg), the resultant signature artifacts are automatically added to the corresponding publication. Thus, when publishing to a repository, e.g. by executing the publish task, your signatures will be distributed along with the other artifacts without any additional configuration.

When signing [configurations](#2ins0jv) and [tasks](#1wy03na), the resultant signature artifacts are automatically added to the signatures and archives dependency configurations. This means that if you want to upload your signatures to your distribution repository along with the artifacts you simply execute the uploadArchives task.

[Signing POM files](#39i3shx)

| **Note** | This section covers signing POM files for the *original* publishing mechanism available in Gradle 1.0. The POM file generated by the *new* Maven publishing support provided by the [Maven Publishing plugin](#1egqt2p) is automatically signed if the corresponding publication is [specified to be signed](#34djxgg). |
| --- | --- |

When deploying signatures for your artifacts to a Maven repository, you will also want to sign the published POM file. The Signing Plugin adds a signing.signPom() (see [SigningExtension.signPom(org.gradle.api.artifacts.maven.MavenDeployment, groovy.lang.Closure)](http://docs.google.com/dsl/org.gradle.plugins.signing.SigningExtension.html#org.gradle.plugins.signing.SigningExtension:signPom(org.gradle.api.artifacts.maven.MavenDeployment,%20groovy.lang.Closure))) method that can be used in the beforeDeployment() block in your upload task configuration.

[Example: Signing a POM for deployment](#1one2pq)

**build.gradle**

uploadArchives {  
 repositories {  
 mavenDeployer {  
 beforeDeployment { MavenDeployment deployment -> signing.signPom(deployment) }  
 }  
 }  
}

When signing is not required and the POM cannot be signed due to insufficient configuration (i.e. no credentials for signing) then the signPom() method will silently do nothing.

[The Distribution Plugin](#1rvwp1q)

| **Note** | The Distribution Plugin is currently [incubating](#2xcytpi). Please be aware that the DSL and other configuration may change in later Gradle versions. |
| --- | --- |

The Distribution Plugin facilitates building archives that serve as distributions of the project. Distribution archives typically contain the executable application and other supporting files, such as documentation.

[Usage](#48n1ldj)

To use the Distribution Plugin, include the following in your build script:

[Example: Using the Distribution Plugin](#2nsbvlc)

**build.gradle**

apply plugin: 'distribution'

The plugin adds an extension named distributions of type [DistributionContainer](http://docs.google.com/dsl/org.gradle.api.distribution.DistributionContainer.html) to the project. It also creates a single distribution in the distributions container extension named main. If your build only produces one distribution you only need to configure this distribution (or use the defaults).

You can run gradle distZip to package the main distribution as a ZIP, or gradle distTar to create a TAR file. To build both types of archives just run gradle assembleDist. The files will be created at *$buildDir*/distributions/*${project.name}*-*${project.version}*.*«ext»*.

You can run gradle installDist to assemble the uncompressed distribution into *$buildDir*/install/*${project.name}*.

[Tasks](#12xm5t5)

The Distribution Plugin adds a number of tasks to your project, as shown below.

distZip — [Zip](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Zip.html)

Creates a ZIP archive of the distribution contents.

distTar — [Task](http://docs.google.com/dsl/org.gradle.api.Task.html)

Creates a TAR archive of the distribution contents.

assembleDist — [Task](http://docs.google.com/dsl/org.gradle.api.Task.html)

*Depends on*: distTar, distZip

Creates ZIP and TAR archives of the distribution contents.

installDist — [Sync](http://docs.google.com/dsl/org.gradle.api.tasks.Sync.html)

Assembles the distribution content and installs it on the current machine.

For each additional distribution you add to the project, the Distribution Plugin adds the following tasks, where *distributionName* comes from [Distribution.getName()](http://docs.google.com/javadoc/org/gradle/api/distribution/Distribution.html#getName--):

*distributionName*DistZip — [Zip](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Zip.html)

Creates a ZIP archive of the distribution contents.

*distributionName*DistTar — [Tar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Tar.html)

Creates a TAR archive of the distribution contents.

assemble*DistributionName*Dist — [Task](http://docs.google.com/dsl/org.gradle.api.Task.html)

*Depends on*: *distributionName*DistTar, *distributionName*DistZip

Creates ZIP and TAR archives of the distribution contents.

install*DistributionName*Dist — [Sync](http://docs.google.com/dsl/org.gradle.api.tasks.Sync.html)

Assembles the distribution content and installs it on the current machine.

The following sample creates a custom distribution that will cause four additional tasks to be added to the project: customDistZip, customDistTar, assembleCustomDist, and installCustomDist:

[Example: Adding extra distributions](#3mx9ogy)

**build.gradle**

distributions {  
 custom {}  
}

Given that the project name is myproject and version 1.2, running gradle customDistZip will produce a ZIP file named myproject-custom-1.2.zip.

Running gradle installCustomDist will install the distribution contents into *$buildDir*/install/custom.

[Distribution contents](#222jyor)

All of the files in the src/*$distribution.name*/dist directory will automatically be included in the distribution. You can add additional files by configuring the [Distribution](http://docs.google.com/javadoc/org/gradle/api/distribution/Distribution.html) object that is part of the container.

[Example: Configuring the main distribution](#h7u8wk)

**build.gradle**

distributions {  
 main {  
 baseName = 'someName'  
 contents {  
 from { 'src/readme' }  
 }  
 }  
}

In the example above, the content of the src/readme directory will be included in the distribution (along with the files in the src/main/dist directory which are added by default).

The baseName property has also been changed. This will cause the distribution archives to be created with a different name.

[Publishing](#317hrkd)

A distribution can be published using the [Ivy Publish Plugin](#3ygebqi) or [Maven Publish Plugin](#1egqt2p), or via the *original* publishing mechanism using the uploadArchives task.

[Using the Ivy/Maven Publish Plugins](#1gcs1s6)

To publish a distribution to an Ivy repository with the [Ivy Publish Plugin](#3ygebqi), simply add one or both of its archive tasks to an [IvyPublication](http://docs.google.com/dsl/org.gradle.api.publish.ivy.IvyPublication.html). The following sample demonstrates how to add the ZIP archive of the main distribution and the TAR archive of the custom distribution to the myDistribution publication:

[Example: Adding distribution archives to an Ivy publication](#40cfkfz)

**build.gradle**

apply plugin: 'ivy-publish'  
  
publishing {  
 publications {  
 myDistribution(IvyPublication) {  
 artifact distZip  
 artifact customDistTar  
 }  
 }  
}

Similarly, to publish a distribution to a Maven repository using the [Maven Publish Plugin](#1egqt2p), add one or both of its archive tasks to a [MavenPublication](http://docs.google.com/dsl/org.gradle.api.publish.maven.MavenPublication.html) as follows:

[Example: Adding distribution archives to a Maven publication](#2fhpuns)

**build.gradle**

apply plugin: 'maven-publish'  
  
publishing {  
 publications {  
 myDistribution(MavenPublication) {  
 artifact distZip  
 artifact customDistTar  
 }  
 }  
}

[Using the uploadArchives task](#un04vl)

The Distribution Plugin adds the distribution archives as default publishing artifact candidates. With the [Maven Plugin](#sqyw64) applied, the distribution ZIP file will be published when running uploadArchives if no other default artifact is configured.

[Example: Publishing the distribution ZIP with the Maven Plugin](#3emnnje)

**build.gradle**

apply plugin:'maven'  
  
uploadArchives {  
 repositories {  
 mavenDeployer {  
 repository(url: "file://some/repo")  
 }  
 }  
}

[Native Projects](#4bvk7pj)

[Building native software](#2r0uhxc)

| **Note** | Support for building native software is currently [incubating](#2xcytpi). Please be aware that the DSL, APIs and other configuration may change in later Gradle versions. |
| --- | --- |

The native software plugins add support for building native software components, such as executables or shared libraries, from code written in C++, C and other languages. While many excellent build tools exist for this space of software development, Gradle offers developers its trademark power and flexibility together with dependency management practices more traditionally found in the JVM development space.

The native software plugins make use of the Gradle [software model](#3q5sasy).

[Features](#1trxxr7)

The native software plugins provide:

* Support for building native libraries and applications on Windows, Linux, macOS and other platforms.
* Support for several source languages.
* Support for building different variants of the same software, for different architectures, operating systems, or for any purpose.
* Incremental parallel compilation, precompiled headers.
* Dependency management between native software components.
* Unit test execution.
* Generate Visual studio solution and project files.
* Deep integration with various tool chain, including discovery of installed tool chains.

[Supported languages](#4drlgf0)

The following source languages are currently supported:

* C
* C++
* Objective-C
* Objective-C++
* Assembly
* Windows resources

[Tool chain support](#2swvqmt)

Gradle offers the ability to execute the same build using different tool chains. When you build a native binary, Gradle will attempt to locate a tool chain installed on your machine that can build the binary. You can fine tune exactly how this works, see [Tool chain support](#18260um) for details.

The following tool chains are supported:

| **Operating System** | **Tool Chain** | **Notes** |
| --- | --- | --- |
| Linux | [GCC](http://gcc.gnu.org/) |  |
| Linux | [Clang](http://clang.llvm.org) |  |
| macOS | XCode | Uses the Clang tool chain bundled with XCode. |
| Windows | [Visual C++](https://visualstudio.microsoft.com/) | Windows XP and later, Visual C++ 2010/2012/2013/2015/2017. |
| Windows | [GCC](http://gcc.gnu.org/) with [Cygwin 32](http://cygwin.com) | Windows XP and later. |
| Windows | [GCC](http://gcc.gnu.org/) with [MinGW](http://www.mingw.org/) | Windows XP and later. [Mingw-w64](http://mingw-w64.sourceforge.net) is currently not supported. |

The following tool chains are unofficially supported. They generally work fine, but are not tested continuously:

| **Operating System** | **Tool Chain** | **Notes** |
| --- | --- | --- |
| macOS | [GCC](http://gcc.gnu.org/) from Macports |  |
| macOS | [Clang](http://clang.llvm.org) from Macports |  |
| Windows | [GCC](http://gcc.gnu.org/) with [Cygwin 64](http://cygwin.com) | Windows XP and later. |
| UNIX-like | [GCC](http://gcc.gnu.org/) |  |
| UNIX-like | [Clang](http://clang.llvm.org) |  |

[Tool chain installation](#3s1tjif)

| **Note** | Note that if you are using GCC then you currently need to install support for C++, even if you are not building from C++ source. This restriction will be removed in a future Gradle version. |
| --- | --- |

To build native software, you will need to have a compatible tool chain installed:

[Windows](#2773tq8)

To build on Windows, install a compatible version of Visual Studio. The native plugins will discover the Visual Studio installations and select the latest version. There is no need to mess around with environment variables or batch scripts. This works fine from a Cygwin shell or the Windows command-line.

Alternatively, you can install Cygwin with GCC or MinGW. Clang is currently not supported.

[macOS](#mce3y1)

To build on macOS, you should install XCode. The native plugins will discover the XCode installation using the system PATH.

The native plugins also work with GCC and Clang bundled with Macports. To use one of the Macports tool chains, you will need to make the tool chain the default using the port select command and add Macports to the system PATH.

[Linux](#36c1mlu)

To build on Linux, install a compatible version of GCC or Clang. The native plugins will discover GCC or Clang using the system PATH.

[Native software model](#1lhbwtn)

The native software model builds on the base Gradle [software model](#1664s55).

To build native software using Gradle, your project should define one or more *native components*. Each component represents either an executable or a library that Gradle should build. A project can define any number of components. Gradle does not define any components by default.

For each component, Gradle defines a *source set* for each language that the component can be built from. A source set is essentially just a set of source directories containing source files. For example, when you apply the c plugin and define a library called helloworld, Gradle will define, by default, a source set containing the C source files in the src/helloworld/c directory. It will use these source files to build the helloworld library. This is described in more detail below.

For each component, Gradle defines one or more *binaries* as output. To build a binary, Gradle will take the source files defined for the component, compile them as appropriate for the source language, and link the result into a binary file. For an executable component, Gradle can produce executable binary files. For a library component, Gradle can produce both static and shared library binary files. For example, when you define a library called helloworld and build on Linux, Gradle will, by default, produce libhelloworld.so and libhelloworld.a binaries.

In many cases, more than one binary can be produced for a component. These binaries may vary based on the tool chain used to build, the compiler/linker flags supplied, the dependencies provided, or additional source files provided. Each native binary produced for a component is referred to as a *variant*. Binary variants are discussed in detail below.

[Parallel Compilation](#45gzfhg)

Gradle uses the single build worker pool to concurrently compile and link native components, by default. No special configuration is required to enable concurrent building.

By default, the worker pool size is determined by the number of available processors on the build machine (as reported to the build JVM). To explicitly set the number of workers use the --max-workers command-line option or org.gradle.workers.max system property. There is generally no need to change this setting from its default.

The build worker pool is shared across all build tasks. This means that when using [parallel project execution](#44bvf6o), the maximum number of concurrent individual compilation operations does not increase. For example, if the build machine has 4 processing cores and 10 projects are compiling in parallel, Gradle will only use 4 total workers, not 40.

[Building a library](#2km9pp9)

To build either a static or shared native library, you define a library component in the components container. The following sample defines a library called hello:

[Example: Defining a library component](#zrjzx2)

**build.gradle**

model {  
 components {  
 hello(NativeLibrarySpec)  
 }  
}

A library component is represented using [NativeLibrarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeLibrarySpec.html). Each library component can produce at least one shared library binary ([SharedLibraryBinarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/SharedLibraryBinarySpec.html)) and at least one static library binary ([StaticLibraryBinarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/StaticLibraryBinarySpec.html)).

[Building an executable](#3jr7ikv)

To build a native executable, you define an executable component in the components container. The following sample defines an executable called main:

[Example: Defining executable components](#1ywhsso)

**build.gradle**

model {  
 components {  
 main(NativeExecutableSpec) {  
 sources {  
 c.lib library: "hello"  
 }  
 }  
 }  
}

An executable component is represented using [NativeExecutableSpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeExecutableSpec.html). Each executable component can produce at least one executable binary ([NativeExecutableBinarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeExecutableBinarySpec.html)).

For each component defined, Gradle adds a [FunctionalSourceSet](http://docs.google.com/javadoc/org/gradle/language/base/FunctionalSourceSet.html) with the same name. Each of these functional source sets will contain a language-specific source set for each of the languages supported by the project.

[Assembling or building dependents](#4iw5bgh)

Sometimes, you may need to *assemble* (compile and link) or *build* (compile, link and test) a component or binary and its *dependents* (things that depend upon the component or binary). The native software model provides tasks that enable this capability. First, the *dependent components* report gives insight about the relationships between each component. Second, the *build and assemble dependents* tasks allow you to assemble or build a component and its dependents in one step.

In the following example, the build file defines OpenSSL as a dependency of libUtil and libUtil as a dependency of LinuxApp and WindowsApp. Test suites are treated similarly. Dependents can be thought of as reverse dependencies.



*Figure 24. Dependent Components Example*

| **Note** | By following the dependencies backwards, you can see LinuxApp and WindowsApp are *dependents* of libUtil. When libUtil is changed, Gradle will need to recompile or relink LinuxApp and WindowsApp. |
| --- | --- |

When you *assemble* dependents of a component, the component and all of its dependents are compiled and linked, including any test suite binaries. Gradle’s up-to-date checks are used to only compile or link if something has changed. For instance, if you have changed source files in a way that do not affect the headers of your project, Gradle will be able to skip compilation for dependent components and only need to re-link with the new library. Tests are not run when assembling a component.

When you *build* dependents of a component, the component and all of its dependent binaries are compiled, linked *and checked*. Checking components means running any [check task](#2y1floa) including executing any test suites, so tests *are* run when building a component.

In the following sections, we will demonstrate the usage of the assembleDependents\*, buildDependents\* and dependentComponents tasks with a sample build that contains a CUnit test suite. The build script for the sample is the following:

[Example: Sample build](#1d6pvw3)

**build.gradle**

apply plugin: "c"  
apply plugin: 'cunit-test-suite'  
  
model {  
 flavors {  
 passing  
 failing  
 }  
 platforms {  
 x86 {  
 architecture "x86"  
 }  
 }  
 components {  
 operators(NativeLibrarySpec) {  
 targetPlatform "x86"  
 }  
 }  
 testSuites {  
 operatorsTest(CUnitTestSuiteSpec) {  
 testing $.components.operators  
 }  
 }  
}

| **Note** | The code for this example can be found at samples/native-binaries/cunit in the ‘-all’ distribution of Gradle. |
| --- | --- |

[Dependent components report](#3x6dejw)

Gradle provides a report that you can run from the command-line that shows a graph of components in your project and components that depend upon them. The following is an example of running gradle dependentComponents on the sample project:

[Example: Dependent components report](#2cbnorp)

**Output of** gradle dependentComponents

> gradle dependentComponents  
  
> Task :dependentComponents  
  
------------------------------------------------------------  
Root project  
------------------------------------------------------------  
  
operators - Components that depend on native library 'operators'  
+--- operators:failingSharedLibrary  
+--- operators:failingStaticLibrary  
+--- operators:passingSharedLibrary  
\--- operators:passingStaticLibrary  
  
Some test suites were not shown, use --test-suites or --all to show them.  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

| **Note** | See [DependentComponentsReport](http://docs.google.com/dsl/org.gradle.api.reporting.dependents.DependentComponentsReport.html) API documentation for more details. |
| --- | --- |

By default, non-buildable binaries and test suites are hidden from the report. The dependentComponents task provides options that allow you to see all dependents by using the --all option:

[Example: Dependent components report](#rgxyzi)

**Output of** gradle dependentComponents --all

> gradle dependentComponents --all  
  
> Task :dependentComponents  
  
------------------------------------------------------------  
Root project  
------------------------------------------------------------  
  
operators - Components that depend on native library 'operators'  
+--- operators:failingSharedLibrary  
+--- operators:failingStaticLibrary  
| \--- operatorsTest:failingCUnitExe (t)  
+--- operators:passingSharedLibrary  
\--- operators:passingStaticLibrary  
 \--- operatorsTest:passingCUnitExe (t)  
  
operatorsTest - Components that depend on Cunit test suite 'operatorsTest'  
+--- operatorsTest:failingCUnitExe (t)  
\--- operatorsTest:passingCUnitExe (t)  
  
(t) - Test suite binary  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

Here is the corresponding report for the operators component, showing dependents of all its binaries:

[Example: Report of components that depends on the operators component](#3bglhnb)

**Output of** gradle dependentComponents --component operators

> gradle dependentComponents --component operators  
  
> Task :dependentComponents  
  
------------------------------------------------------------  
Root project  
------------------------------------------------------------  
  
operators - Components that depend on native library 'operators'  
+--- operators:failingSharedLibrary  
+--- operators:failingStaticLibrary  
+--- operators:passingSharedLibrary  
\--- operators:passingStaticLibrary  
  
Some test suites were not shown, use --test-suites or --all to show them.  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

Here is the corresponding report for the operators component, showing dependents of all its binaries, including test suites:

[Example: Report of components that depends on the operators component, including test suites](#1qlvrv4)

**Output of** gradle dependentComponents --test-suites --component operators

> gradle dependentComponents --test-suites --component operators  
  
> Task :dependentComponents  
  
------------------------------------------------------------  
Root project  
------------------------------------------------------------  
  
operators - Components that depend on native library 'operators'  
+--- operators:failingSharedLibrary  
+--- operators:failingStaticLibrary  
| \--- operatorsTest:failingCUnitExe (t)  
+--- operators:passingSharedLibrary  
\--- operators:passingStaticLibrary  
 \--- operatorsTest:passingCUnitExe (t)  
  
(t) - Test suite binary  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

[Assembling dependents](#4aljaix)

For each [NativeBinarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeBinarySpec.html), Gradle will create a task named assembleDependents*${component.name}${binary.variant}* that *assembles* (compile and link) the binary and all of its dependent binaries.

For each [NativeComponentSpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeComponentSpec.html), Gradle will create a task named assembleDependents*${component.name}* that *assembles* all the binaries of the component and all of their dependent binaries.

For example, to assemble the dependents of the "passing" flavor of the "static" library binary of the "operators" component, you would run the assembleDependentsOperatorsPassingStaticLibrary task:

[Example: Assemble components that depends on the passing/static binary of the operators component](#2pqtkqq)

**Output of** gradle assembleDependentsOperatorsPassingStaticLibrary --max-workers=1

> gradle assembleDependentsOperatorsPassingStaticLibrary --max-workers=1  
> Task :compileOperatorsTestPassingCUnitExeOperatorsC  
> Task :operatorsTestCUnitLauncher  
> Task :compileOperatorsTestPassingCUnitExeOperatorsTestC  
> Task :compileOperatorsTestPassingCUnitExeOperatorsTestCunitLauncher  
> Task :linkOperatorsTestPassingCUnitExe  
> Task :operatorsTestPassingCUnitExe  
> Task :assembleDependentsOperatorsTestPassingCUnitExe  
> Task :compileOperatorsPassingStaticLibraryOperatorsC  
> Task :createOperatorsPassingStaticLibrary  
> Task :operatorsPassingStaticLibrary  
> Task :assembleDependentsOperatorsPassingStaticLibrary  
  
BUILD SUCCESSFUL in 0s  
7 actionable tasks: 7 executed

In the output above, the targeted binary gets assembled as well as the test suite binary that depends on it.

You can also assemble *all* of the dependents of a component (i.e. of all its binaries/variants) using the corresponding component task, e.g. assembleDependentsOperators. This is useful if you have many combinations of build types, flavors and platforms and want to assemble all of them.

[Building dependents](#14w3uyj)

For each [NativeBinarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeBinarySpec.html), Gradle will create a task named buildDependents*${component.name}${binary.variant}* that *builds* (compile, link and check) the binary and all of its dependent binaries.

For each [NativeComponentSpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeComponentSpec.html), Gradle will create a task named buildDependents*${component.name}* that *builds* all the binaries of the component and all of their dependent binaries.

For example, to build the dependents of the "passing" flavor of the "static" library binary of the "operators" component, you would run the buildDependentsOperatorsPassingStaticLibrary task:

[Example: Build components that depends on the passing/static binary of the operators component](#3ovrdmc)

**Output of** gradle buildDependentsOperatorsPassingStaticLibrary --max-workers=1

> gradle buildDependentsOperatorsPassingStaticLibrary --max-workers=1  
> Task :compileOperatorsTestPassingCUnitExeOperatorsC  
> Task :operatorsTestCUnitLauncher  
> Task :compileOperatorsTestPassingCUnitExeOperatorsTestC  
> Task :compileOperatorsTestPassingCUnitExeOperatorsTestCunitLauncher  
> Task :linkOperatorsTestPassingCUnitExe  
> Task :operatorsTestPassingCUnitExe  
> Task :installOperatorsTestPassingCUnitExe  
> Task :runOperatorsTestPassingCUnitExe  
> Task :checkOperatorsTestPassingCUnitExe  
> Task :buildDependentsOperatorsTestPassingCUnitExe  
> Task :compileOperatorsPassingStaticLibraryOperatorsC  
> Task :createOperatorsPassingStaticLibrary  
> Task :operatorsPassingStaticLibrary  
> Task :buildDependentsOperatorsPassingStaticLibrary  
  
BUILD SUCCESSFUL in 0s  
9 actionable tasks: 9 executed

In the output above, the targeted binary as well as the test suite binary that depends on it are built and the test suite has run.

You can also build *all* of the dependents of a component (i.e. of all its binaries/variants) using the corresponding component task, e.g. buildDependentsOperators.

[Tasks](#2411nu5)

For each [NativeBinarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeBinarySpec.html) that can be produced by a build, a single *lifecycle task* is constructed that can be used to create that binary, together with a set of other tasks that do the actual work of compiling, linking or assembling the binary.

*${component.name}*Executable

Component Type

[NativeExecutableSpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeExecutableSpec.html)

Native Binary Type

[NativeExecutableBinarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeExecutableBinarySpec.html)

Location of created binary

*${project.buildDir}*/exe/*${component.name}*/*${component.name}*

*${component.name}*SharedLibrary

Component Type

[NativeLibrarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeLibrarySpec.html)

Native Binary Type

[SharedLibraryBinarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/SharedLibraryBinarySpec.html)

Location of created binary

*${project.buildDir}*/libs/*${component.name}*/shared/lib*${component.name}*.so

*${component.name}*StaticLibrary

Component Type

[NativeLibrarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeLibrarySpec.html)

Native Binary Type

[StaticLibraryBinarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/StaticLibraryBinarySpec.html)

Location of created binary

*${project.buildDir}*/libs/*${component.name}*/static/*${component.name}*.a

[Check tasks](#2y1floa)

For each [NativeBinarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeBinarySpec.html) that can be produced by a build, a single *check task* is constructed that can be used to assemble and check that binary.

check*${component.name}*Executable

Component Type

[NativeExecutableSpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeExecutableSpec.html)

Native Binary Type

[NativeExecutableBinarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeExecutableBinarySpec.html)

check*${component.name}*SharedLibrary

Component Type

[NativeLibrarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeLibrarySpec.html)

Native Binary Type

[SharedLibraryBinarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/SharedLibraryBinarySpec.html)

check*${component.name}*StaticLibrary

Component Type

[NativeLibrarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeLibrarySpec.html)

Native Binary Type

[SharedLibraryBinarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/SharedLibraryBinarySpec.html)

The built-in check task depends on all the *check tasks* for binaries in the project. Without either [CUnit](#j6by1y) or [GoogleTest](#335zgpr) plugins, the binary check task only depends on the *lifecycle task* that assembles the binary, see [Native tasks](#2411nu5).

When the [CUnit](#j6by1y) or [GoogleTest](#335zgpr) plugins are applied, the task that executes the test suites for a component are automatically wired to the appropriate *check task*.

You can also add custom check tasks as follows:

[Example: Adding a custom check task](#1ib9qxk)

**build.gradle**

apply plugin: "cpp"  
*// You don't need to apply the plugin below if you're already using CUnit or GoogleTest support*  
apply plugin: TestingModelBasePlugin  
  
task myCustomCheck {  
 doLast {  
 println 'Executing my custom check'  
 }  
}  
  
model {  
 components {  
 hello(NativeLibrarySpec) {  
 binaries.all {  
 *// Register our custom check task to all binaries of this component*  
 checkedBy $.tasks.myCustomCheck  
 }  
 }  
 }  
}

| **Note** | The code for this example can be found at samples/native-binaries/custom-check in the ‘-all’ distribution of Gradle. |
| --- | --- |

Now, running check or any of the *check tasks* for the hello binaries will run the custom check task:

[Example: Running checks for a given binary](#42ax9ld)

**Output of** gradle checkHelloSharedLibrary

> gradle checkHelloSharedLibrary  
  
> Task :myCustomCheck  
Executing my custom check  
  
> Task :checkHelloSharedLibrary  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

[Working with shared libraries](#2hg7jt6)

For each executable binary produced, the cpp plugin provides an install${binary.name} task, which creates a development install of the executable, along with the shared libraries it requires. This allows you to run the executable without needing to install the shared libraries in their final locations.

[Finding out more about your project](#wlhu0z)

Gradle provides a report that you can run from the command-line that shows some details about the components and binaries that your project produces. To use this report, just run gradle components. Below is an example of running this report for one of the sample projects:

[Example: The components report](#3gl5cos)

**Output of** gradle components

> gradle components  
  
> Task :components  
  
------------------------------------------------------------  
Root project  
------------------------------------------------------------  
  
Native library 'hello'  
----------------------  
  
Source sets  
 C++ source 'hello:cpp'  
 srcDir: src/hello/cpp  
  
Binaries  
 Shared library 'hello:sharedLibrary'  
 build using task: :helloSharedLibrary  
 build type: build type 'debug'  
 flavor: flavor 'default'  
 target platform: platform 'current'  
 tool chain: Tool chain 'clang' (Clang)  
 shared library file: build/libs/hello/shared/libhello.dylib  
 Static library 'hello:staticLibrary'  
 build using task: :helloStaticLibrary  
 build type: build type 'debug'  
 flavor: flavor 'default'  
 target platform: platform 'current'  
 tool chain: Tool chain 'clang' (Clang)  
 static library file: build/libs/hello/static/libhello.a  
  
Native executable 'main'  
------------------------  
  
Source sets  
 C++ source 'main:cpp'  
 srcDir: src/main/cpp  
  
Binaries  
 Executable 'main:executable'  
 build using task: :mainExecutable  
 install using task: :installMainExecutable  
 build type: build type 'debug'  
 flavor: flavor 'default'  
 target platform: platform 'current'  
 tool chain: Tool chain 'clang' (Clang)  
 executable file: build/exe/main/main  
  
Note: currently not all plugins register their components, so some components may not be visible here.  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

[Language support](#1vqfmwl)

Presently, Gradle supports building native software from any combination of source languages listed below. A native binary project will contain one or more named FunctionalSourceSet instances (eg 'main', 'test', etc), each of which can contain LanguageSourceSets containing source files, one for each language.

* C
* C++
* Objective-C
* Objective-C++
* Assembly
* Windows resources

[C++ sources](#4fq35ke)

C++ language support is provided by means of the 'cpp' plugin.

[Example: The 'cpp' plugin](#2uvdfs7)

**build.gradle**

apply plugin: 'cpp'

C++ sources to be included in a native binary are provided via a [CppSourceSet](http://docs.google.com/dsl/org.gradle.language.cpp.CppSourceSet.html), which defines a set of C++ source files and optionally a set of exported header files (for a library). By default, for any named component the [CppSourceSet](http://docs.google.com/dsl/org.gradle.language.cpp.CppSourceSet.html) contains .cpp source files in src/${name}/cpp, and header files in src/${name}/headers.

While the cpp plugin defines these default locations for each [CppSourceSet](http://docs.google.com/dsl/org.gradle.language.cpp.CppSourceSet.html), it is possible to extend or override these defaults to allow for a different project layout.

[Example: C++ source set](#1a0nq00)

**build.gradle**

sources {  
 cpp {  
 source {  
 srcDir "src/source"  
 include "\*\*/\*.cpp"  
 }  
 }  
}

For a library named 'main', header files in src/main/headers are considered the "public" or "exported" headers. Header files that should not be exported should be placed inside the src/main/cpp directory (though be aware that such header files should always be referenced in a manner relative to the file including them).

[C sources](#3u0b8nt)

C language support is provided by means of the 'c' plugin.

[Example: The 'c' plugin](#295livm)

**build.gradle**

apply plugin: 'c'

C sources to be included in a native binary are provided via a [CSourceSet](http://docs.google.com/dsl/org.gradle.language.c.CSourceSet.html), which defines a set of C source files and optionally a set of exported header files (for a library). By default, for any named component the [CSourceSet](http://docs.google.com/dsl/org.gradle.language.c.CSourceSet.html) contains .c source files in src/${name}/c, and header files in src/${name}/headers.

While the c plugin defines these default locations for each [CSourceSet](http://docs.google.com/dsl/org.gradle.language.c.CSourceSet.html), it is possible to extend or override these defaults to allow for a different project layout.

[Example: C source set](#oavt3f)

**build.gradle**

sources {  
 c {  
 source {  
 srcDir "src/source"  
 include "\*\*/\*.c"  
 }  
 exportedHeaders {  
 srcDir "src/include"  
 }  
 }  
}

For a library named 'main', header files in src/main/headers are considered the "public" or "exported" headers. Header files that should not be exported should be placed inside the src/main/c directory (though be aware that such header files should always be referenced in a manner relative to the file including them).

[Assembler sources](#38ajbr8)

Assembly language support is provided by means of the 'assembler' plugin.

[Example: The 'assembler' plugin](#1nftlz1)

**build.gradle**

apply plugin: 'assembler'

Assembler sources to be included in a native binary are provided via a [AssemblerSourceSet](http://docs.google.com/dsl/org.gradle.language.assembler.AssemblerSourceSet.html), which defines a set of Assembler source files. By default, for any named component the [AssemblerSourceSet](http://docs.google.com/dsl/org.gradle.language.assembler.AssemblerSourceSet.html) contains .s source files under src/${name}/asm.

[Objective-C sources](#47fh4mu)

Objective-C language support is provided by means of the 'objective-c' plugin.

[Example: The 'objective-c' plugin](#2mkreun)

**build.gradle**

apply plugin: 'objective-c'

Objective-C sources to be included in a native binary are provided via a [ObjectiveCSourceSet](http://docs.google.com/dsl/org.gradle.language.objectivec.ObjectiveCSourceSet.html), which defines a set of Objective-C source files. By default, for any named component the [ObjectiveCSourceSet](http://docs.google.com/dsl/org.gradle.language.objectivec.ObjectiveCSourceSet.html) contains .m source files under src/${name}/objectiveC.

[Objective-C++ sources](#11q1p2g)

Objective-C++ language support is provided by means of the 'objective-cpp' plugin.

[Example: The 'objective-cpp' plugin](#3lpp7q9)

**build.gradle**

apply plugin: 'objective-cpp'

Objective-C++ sources to be included in a native binary are provided via a [ObjectiveCppSourceSet](http://docs.google.com/dsl/org.gradle.language.objectivecpp.ObjectiveCppSourceSet.html), which defines a set of Objective-C++ source files. By default, for any named component the [ObjectiveCppSourceSet](http://docs.google.com/dsl/org.gradle.language.objectivecpp.ObjectiveCppSourceSet.html) contains .mm source files under src/${name}/objectiveCpp.

[Configuring the compiler, assembler and linker](#20uzhy2)

Each binary to be produced is associated with a set of compiler and linker settings, which include command-line arguments as well as macro definitions. These settings can be applied to all binaries, an individual binary, or selectively to a group of binaries based on some criteria.

[Example: Settings that apply to all binaries](#4kun0lv)

**build.gradle**

model {  
 binaries {  
 all {  
 *// Define a preprocessor macro for every binary*  
 cppCompiler.define "NDEBUG"  
  
 *// Define toolchain-specific compiler and linker options*  
 **if** (toolChain **in** Gcc) {  
 cppCompiler.args "-O2", "-fno-access-control"  
 linker.args "-Xlinker", "-S"  
 }  
 **if** (toolChain **in** VisualCpp) {  
 cppCompiler.args "/Zi"  
 linker.args "/DEBUG"  
 }  
 }  
 }  
}

Each binary is associated with a particular [NativeToolChain](http://docs.google.com/javadoc/org/gradle/nativeplatform/toolchain/NativeToolChain.html), allowing settings to be targeted based on this value.

It is easy to apply settings to all binaries of a particular type:

[Example: Settings that apply to all shared libraries](#2zzxato)

**build.gradle**

*// For any shared library binaries built with Visual C++,*  
*// define the DLL\_EXPORT macro*  
model {  
 binaries {  
 withType(SharedLibraryBinarySpec) {  
 **if** (toolChain **in** VisualCpp) {  
 cCompiler.args "/Zi"  
 cCompiler.define "DLL\_EXPORT"  
 }  
 }  
 }  
}

Furthermore, it is possible to specify settings that apply to all binaries produced for a particular executable or library component:

[Example: Settings that apply to all binaries produced for the 'main' executable component](#1f57l1h)

**build.gradle**

model {  
 components {  
 main(NativeExecutableSpec) {  
 targetPlatform "x86"  
 binaries.all {  
 **if** (toolChain **in** VisualCpp) {  
 sources {  
 platformAsm(AssemblerSourceSet) {  
 source.srcDir "src/main/asm\_i386\_masm"  
 }  
 }  
 assembler.args "/Zi"  
 } **else** {  
 sources {  
 platformAsm(AssemblerSourceSet) {  
 source.srcDir "src/main/asm\_i386\_gcc"  
 }  
 }  
 assembler.args "-g"  
 }  
 }  
 }  
 }  
}

The example above will apply the supplied configuration to all executable binaries built.

Similarly, settings can be specified to target binaries for a component that are of a particular type: eg all shared libraries for the main library component.

[Example: Settings that apply only to shared libraries produced for the 'main' library component](#3z4v3pa)

**build.gradle**

model {  
 components {  
 main(NativeLibrarySpec) {  
 binaries.withType(SharedLibraryBinarySpec) {  
 *// Define a preprocessor macro that only applies to shared libraries*  
 cppCompiler.define "DLL\_EXPORT"  
 }  
 }  
 }  
}

[Windows Resources](#2ea5dx3)

When using the [VisualCpp](http://docs.google.com/dsl/org.gradle.nativeplatform.toolchain.VisualCpp.html) tool chain, Gradle is able to compile Window Resource (rc) files and link them into a native binary. This functionality is provided by the 'windows-resources' plugin.

[Example: The 'windows-resources' plugin](#tffo4w)

**build.gradle**

apply plugin: 'windows-resources'

Windows resources to be included in a native binary are provided via a [WindowsResourceSet](http://docs.google.com/dsl/org.gradle.language.rc.WindowsResourceSet.html), which defines a set of Windows Resource source files. By default, for any named component the [WindowsResourceSet](http://docs.google.com/dsl/org.gradle.language.rc.WindowsResourceSet.html) contains .rc source files under src/${name}/rc.

As with other source types, you can configure the location of the windows resources that should be included in the binary.

[Example: Configuring the location of Windows resource sources](#3df36sp)

**build-resource-only-dll.gradle**

sources {  
 rc {  
 source {  
 srcDirs "src/hello/rc"  
 }  
 exportedHeaders {  
 srcDirs "src/hello/headers"  
 }  
 }  
}

You are able to construct a resource-only library by providing Windows Resource sources with no other language sources, and configure the linker as appropriate:

[Example: Building a resource-only dll](#1skdh0i)

**build-resource-only-dll.gradle**

model {  
 components {  
 helloRes(NativeLibrarySpec) {  
 binaries.all {  
 rcCompiler.args "/v"  
 linker.args "/noentry", "/machine:x86"  
 }  
 sources {  
 rc {  
 source {  
 srcDirs "src/hello/rc"  
 }  
 exportedHeaders {  
 srcDirs "src/hello/headers"  
 }  
 }  
 }  
 }  
 }  
}

The example above also demonstrates the mechanism of passing extra command-line arguments to the resource compiler. The rcCompiler extension is of type [PreprocessingTool](http://docs.google.com/javadoc/org/gradle/nativeplatform/PreprocessingTool.html).

[Library Dependencies](#4ck0zob)

Dependencies for native components are binary libraries that export header files. The header files are used during compilation, with the compiled binary dependency being used during linking and execution. Header files should be organized into subdirectories to prevent clashes of commonly named headers. For instance, if your mylib project has a logging.h header, it will make it less likely the wrong header is used if you include it as "mylib/logging.h" instead of "logging.h".

[Dependencies within the same project](#2rpb9w4)

A set of sources may depend on header files provided by another binary component within the same project. A common example is a native executable component that uses functions provided by a separate native library component.

Such a library dependency can be added to a source set associated with the executable component:

[Example: Providing a library dependency to the source set](#16ulk3x)

**build.gradle**

sources {  
 cpp {  
 lib library: "hello"  
 }  
}

Alternatively, a library dependency can be provided directly to the NativeExecutableBinarySpec for the executable.

[Example: Providing a library dependency to the binary](#3qu92rq)

**build.gradle**

model {  
 components {  
 hello(NativeLibrarySpec) {  
 sources {  
 c {  
 source {  
 srcDir "src/source"  
 include "\*\*/\*.c"  
 }  
 exportedHeaders {  
 srcDir "src/include"  
 }  
 }  
 }  
 }  
 main(NativeExecutableSpec) {  
 sources {  
 cpp {  
 source {  
 srcDir "src/source"  
 include "\*\*/\*.cpp"  
 }  
 }  
 }  
 binaries.all {  
 *// Each executable binary produced uses the 'hello' static library binary*  
 lib library: 'hello', linkage: 'static'  
 }  
 }  
 }  
}

[Project Dependencies](#25zjczj)

For a component produced in a different Gradle project, the notation is similar.

[Example: Declaring project dependencies](#l4tn7c)

**build.gradle**

project(":lib") {  
 apply plugin: "cpp"  
 model {  
 components {  
 main(NativeLibrarySpec)  
 }  
  
 *// For any shared library binaries built with Visual C++,*  
 *// define the DLL\_EXPORT macro*  
 binaries {  
 withType(SharedLibraryBinarySpec) {  
 **if** (toolChain **in** VisualCpp) {  
 cppCompiler.define "DLL\_EXPORT"  
 }  
 }  
 }  
 }  
}  
  
project(":exe") {  
 apply plugin: "cpp"  
  
 model {  
 components {  
 main(NativeExecutableSpec) {  
 sources {  
 cpp {  
 lib project: ':lib', library: 'main'  
 }  
 }  
 }  
 }  
 }  
}

[Precompiled Headers](#354h5v5)

Precompiled headers are a performance optimization that reduces the cost of compiling widely used headers multiple times. This feature *precompiles* a header such that the compiled object file can be reused when compiling each source file rather than recompiling the header each time. This support is available for C, C++, Objective-C, and Objective-C++ builds.

To configure a precompiled header, first a header file needs to be defined that includes all of the headers that should be precompiled. It must be specified as the first included header in every source file where the precompiled header should be used. It is assumed that this header file, and any headers it contains, make use of header guards so that they can be included in an idempotent manner. If header guards are not used in a header file, it is possible the header could be compiled more than once and could potentially lead to a broken build.

[Example: Creating a precompiled header file](#1k9rg2y)

**src/hello/headers/pch.h**

#ifndef PCH\_H  
#define PCH\_H  
#include <iostream>  
#include "hello.h"  
#endif

[Example: Including a precompiled header file in a source file](#449eyqr)

**src/hello/cpp/hello.cpp**

#include "pch.h"  
  
**void** LIB\_FUNC Greeter::hello () {  
 std::cout << "Hello world!" << std::endl;  
}

Precompiled headers are specified on a source set. Only one precompiled header file can be specified on a given source set and will be applied to all source files that declare it as the first include. If a source files does not include this header file as the first header, the file will be compiled in the normal manner (without making use of the precompiled header object file). The string provided should be the same as that which is used in the "#include" directive in the source files.

[Example: Configuring a precompiled header](#2jep8yk)

**build.gradle**

model {  
 components {  
 hello(NativeLibrarySpec) {  
 sources {  
 cpp {  
 preCompiledHeader "pch.h"  
 }  
 }  
 }  
 }  
}

A precompiled header must be included in the same way for all files that use it. Usually, this means the header file should exist in the source set "headers" directory or in a directory included on the compiler include path.

[Native Binary Variants](#yjzj6d)

For each executable or library defined, Gradle is able to build a number of different native binary variants. Examples of different variants include debug vs release binaries, 32-bit vs 64-bit binaries, and binaries produced with different custom preprocessor flags.

Binaries produced by Gradle can be differentiated on [build type](#3ijn1u6), [platform](#1xoxc1z), and [flavor](#4hokups). For each of these 'variant dimensions', it is possible to specify a set of available values as well as target each component at one, some or all of these. For example, a plugin may define a range of support platforms, but you may choose to only target Windows-x86 for a particular component.

[Build types](#3ijn1u6)

A build type determines various non-functional aspects of a binary, such as whether debug information is included, or what optimisation level the binary is compiled with. Typical build types are 'debug' and 'release', but a project is free to define any set of build types.

[Example: Defining build types](#2wtv4xl)

**build.gradle**

model {  
 buildTypes {  
 debug  
 release  
 }  
}

If no build types are defined in a project, then a single, default build type called 'debug' is added.

For a build type, a Gradle project will typically define a set of compiler/linker flags per tool chain.

[Example: Configuring debug binaries](#1bz5f5e)

**build.gradle**

model {  
 binaries {  
 all {  
 **if** (toolChain **in** Gcc && buildType == buildTypes.debug) {  
 cppCompiler.args "-g"  
 }  
 **if** (toolChain **in** VisualCpp && buildType == buildTypes.debug) {  
 cppCompiler.args '/Zi'  
 cppCompiler.define 'DEBUG'  
 linker.args '/DEBUG'  
 }  
 }  
 }  
}

| **Note** | At this stage, it is completely up to the build script to configure the relevant compiler/linker flags for each build type. Future versions of Gradle will automatically include the appropriate debug flags for any 'debug' build type, and may be aware of various levels of optimisation as well. |
| --- | --- |

[Platform](#1xoxc1z)

An executable or library can be built to run on different operating systems and cpu architectures, with a variant being produced for each platform. Gradle defines each OS/architecture combination as a [NativePlatform](http://docs.google.com/dsl/org.gradle.nativeplatform.platform.NativePlatform.html), and a project may define any number of platforms. If no platforms are defined in a project, then a single, default platform 'current' is added.

| **Note** | Presently, a Platform consists of a defined operating system and architecture. As we continue to develop the native binary support in Gradle, the concept of Platform will be extended to include things like C-runtime version, Windows SDK, ABI, etc. Sophisticated builds may use the extensibility of Gradle to apply additional attributes to each platform, which can then be queried to specify particular includes, preprocessor macros or compiler arguments for a native binary. |
| --- | --- |

[Example: Defining platforms](#3vysxt7)

**build.gradle**

model {  
 platforms {  
 x86 {  
 architecture "x86"  
 }  
 x64 {  
 architecture "x86\_64"  
 }  
 itanium {  
 architecture "ia-64"  
 }  
 }  
}

For a given variant, Gradle will attempt to find a [NativeToolChain](http://docs.google.com/javadoc/org/gradle/nativeplatform/toolchain/NativeToolChain.html) that is able to build for the target platform. Available tool chains are searched in the order defined. See the [tool chains](#18260um) section below for more details.

[Flavor](#4hokups)

Each component can have a set of named flavors, and a separate binary variant can be produced for each flavor. While the build type and target platform variant dimensions have a defined meaning in Gradle, each project is free to define any number of flavors and apply meaning to them in any way.

An example of component flavors might differentiate between 'demo', 'paid' and 'enterprise' editions of the component, where the same set of sources is used to produce binaries with different functions.

[Example: Defining flavors](#2b43810)

**build.gradle**

model {  
 flavors {  
 english  
 french  
 }  
 components {  
 hello(NativeLibrarySpec) {  
 binaries.all {  
 **if** (flavor == flavors.french) {  
 cppCompiler.define "FRENCH"  
 }  
 }  
 }  
 }  
}

In the example above, a library is defined with a 'english' and 'french' flavor. When compiling the 'french' variant, a separate macro is defined which leads to a different binary being produced.

If no flavor is defined for a component, then a single default flavor named 'default' is used.

[Selecting the build types, platforms and flavors for a component](#q9di8t)

For a default component, Gradle will attempt to create a native binary variant for each and every combination of buildType and flavor defined for the project. It is possible to override this on a per-component basis, by specifying the set of targetBuildTypes and/or targetFlavors. By default, Gradle will build for the default platform, see [above](#1xoxc1z), unless specified explicitly on a per-component basis by specifying a set of targetPlatforms.

[Example: Targeting a component at particular platforms](#3a910wm)

**build.gradle**

model {  
 components {  
 hello(NativeLibrarySpec) {  
 targetPlatform "x86"  
 targetPlatform "x64"  
 }  
 main(NativeExecutableSpec) {  
 targetPlatform "x86"  
 targetPlatform "x64"  
 sources {  
 cpp.lib library: 'hello', linkage: 'static'  
 }  
 }  
 }  
}

Here you can see that the [TargetedNativeComponent.targetPlatform(java.lang.String)](http://docs.google.com/dsl/org.gradle.nativeplatform.TargetedNativeComponent.html#org.gradle.nativeplatform.TargetedNativeComponent:targetPlatform(java.lang.String)) method is used to specify a platform that the NativeExecutableSpec named main should be built for.

A similar mechanism exists for selecting [TargetedNativeComponent.targetBuildTypes(java.lang.String…​)](http://docs.google.com/dsl/org.gradle.nativeplatform.TargetedNativeComponent.html#org.gradle.nativeplatform.TargetedNativeComponent:targetBuildTypes(java.lang.String%5B%5D)) and [TargetedNativeComponent.targetFlavors(java.lang.String…​)](http://docs.google.com/dsl/org.gradle.nativeplatform.TargetedNativeComponent.html#org.gradle.nativeplatform.TargetedNativeComponent:targetFlavors(java.lang.String%5B%5D)).

[Building all possible variants](#1pebb4f)

When a set of build types, target platforms, and flavors is defined for a component, a [NativeBinarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeBinarySpec.html) model element is created for every possible combination of these. However, in many cases it is not possible to build a particular variant, perhaps because no tool chain is available to build for a particular platform.

If a binary variant cannot be built for any reason, then the [NativeBinarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeBinarySpec.html) associated with that variant will not be buildable. It is possible to use this property to create a task to generate all possible variants on a particular machine.

[Example: Building all possible variants](#49dyts8)

**build.gradle**

model {  
 tasks {  
 buildAllExecutables(Task) {  
 dependsOn $.binaries.findAll { it.buildable }  
 }  
 }  
}

[Tool chains](#18260um)

A single build may utilize different tool chains to build variants for different platforms. To this end, the core 'native-binary' plugins will attempt to locate and make available supported tool chains. However, the set of tool chains for a project may also be explicitly defined, allowing additional cross-compilers to be configured as well as allowing the install directories to be specified.

[Defining tool chains](#2oj9401)

The supported tool chain types are:

* [Gcc](http://docs.google.com/dsl/org.gradle.nativeplatform.toolchain.Gcc.html)
* [Clang](http://docs.google.com/dsl/org.gradle.nativeplatform.toolchain.Clang.html)
* [VisualCpp](http://docs.google.com/dsl/org.gradle.nativeplatform.toolchain.VisualCpp.html)

[Example: Defining tool chains](#13oje7u)

**build.gradle**

model {  
 toolChains {  
 visualCpp(VisualCpp) {  
 *// Specify the installDir if Visual Studio cannot be located*  
 *// installDir "C:/Apps/Microsoft Visual Studio 10.0"*  
 }  
 gcc(Gcc) {  
 *// Uncomment to use a GCC install that is not in the PATH*  
 *// path "/usr/bin/gcc"*  
 }  
 clang(Clang)  
 }  
}

Each tool chain implementation allows for a certain degree of configuration (see the API documentation for more details).

[Using tool chains](#3no6wvn)

It is not necessary or possible to specify the tool chain that should be used to build. For a given variant, Gradle will attempt to locate a [NativeToolChain](http://docs.google.com/javadoc/org/gradle/nativeplatform/toolchain/NativeToolChain.html) that is able to build for the target platform. Available tool chains are searched in the order defined.

| **Note** | When a platform does not define an architecture or operating system, the default target of the tool chain is assumed. So if a platform does not define a value for operatingSystem, Gradle will find the first available tool chain that can build for the specified architecture. |
| --- | --- |

The core Gradle tool chains are able to target the following architectures out of the box. In each case, the tool chain will target the current operating system. See the next section for information on cross-compiling for other operating systems.

| **Tool Chain** | **Architectures** |
| --- | --- |
| GCC | x86, x86\_64 |
| Clang | x86, x86\_64 |
| Visual C++ | x86, x86\_64, ia-64 |

So for GCC running on linux, the supported target platforms are 'linux/x86' and 'linux/x86\_64'. For GCC running on Windows via Cygwin, platforms 'windows/x86' and 'windows/x86\_64' are supported. (The Cygwin POSIX runtime is not yet modelled as part of the platform, but will be in the future.)

If no target platforms are defined for a project, then all binaries are built to target a default platform named 'current'. This default platform does not specify any architecture or operatingSystem value, hence using the default values of the first available tool chain.

Gradle provides a *hook* that allows the build author to control the exact set of arguments passed to a tool chain executable. This enables the build author to work around any limitations in Gradle, or assumptions that Gradle makes. The arguments hook should be seen as a 'last-resort' mechanism, with preference given to truly modelling the underlying domain.

[Example: Reconfigure tool arguments](#22th73g)

**build.gradle**

model {  
 toolChains {  
 visualCpp(VisualCpp) {  
 eachPlatform {  
 cppCompiler.withArguments { args ->  
 args << "-DFRENCH"  
 }  
 }  
 }  
 clang(Clang) {  
 eachPlatform {  
 cCompiler.withArguments { args ->  
 Collections.replaceAll(args, "CUSTOM", "-DFRENCH")  
 }  
 linker.withArguments { args ->  
 args.remove "CUSTOM"  
 }  
 staticLibArchiver.withArguments { args ->  
 args.remove "CUSTOM"  
 }  
 }  
 }  
 }  
}

[Cross-compiling with GCC](#hyrhb9)

Cross-compiling is possible with the [Gcc](http://docs.google.com/dsl/org.gradle.nativeplatform.toolchain.Gcc.html) and [Clang](http://docs.google.com/dsl/org.gradle.nativeplatform.toolchain.Clang.html) tool chains, by adding support for additional target platforms. This is done by specifying a target platform for a toolchain. For each target platform a custom configuration can be specified.

[Example: Defining target platforms](#31yezz2)

**build.gradle**

model {  
 toolChains {  
 gcc(Gcc) {  
 target("arm"){  
 cppCompiler.withArguments { args ->  
 args << "-m32"  
 }  
 linker.withArguments { args ->  
 args << "-m32"  
 }  
 }  
 target("sparc")  
 }  
 }  
 platforms {  
 arm {  
 architecture "arm"  
 }  
 sparc {  
 architecture "sparc"  
 }  
 }  
 components {  
 main(NativeExecutableSpec) {  
 targetPlatform "arm"  
 targetPlatform "sparc"  
 }  
 }  
}

[Visual Studio IDE integration](#1h3pa6v)

Gradle has the ability to generate Visual Studio project and solution files for the native components defined in your build. This ability is added by the visual-studio plugin. For a multi-project build, all projects with native components (and the root project) should have this plugin applied.

When the visual-studio plugin is applied to the root project, a task named visualStudio is created, which will generate a Visual Studio solution file containing all components in the build. This solution will include a Visual Studio project for each component, as well as configuring each component to build using Gradle.

A task named openVisualStudio is also created by the visual-studio plugin when the project is the root project. This task generates the Visual Studio solution and then opens the solution in Visual Studio. This means you can simply run gradlew openVisualStudio from the root project to generate and open the Visual Studio solution in one convenient step.

The content of the generated visual studio files can be modified via API hooks, provided by the visualStudio extension. Take a look at the 'visual-studio' sample, or see [VisualStudioExtension.getProjects()](http://docs.google.com/dsl/org.gradle.ide.visualstudio.VisualStudioExtension.html#org.gradle.ide.visualstudio.VisualStudioExtension:projects) and [VisualStudioRootExtension.getSolution()](http://docs.google.com/dsl/org.gradle.ide.visualstudio.VisualStudioRootExtension.html#org.gradle.ide.visualstudio.VisualStudioRootExtension:solution) in the API documentation for more details.

[CUnit support](#j6by1y)

The Gradle cunit plugin provides support for compiling and executing CUnit tests in your native-binary project. For each [NativeExecutableSpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeExecutableSpec.html) and [NativeLibrarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeLibrarySpec.html) defined in your project, Gradle will create a matching [CUnitTestSuiteSpec](http://docs.google.com/dsl/org.gradle.nativeplatform.test.cunit.CUnitTestSuiteSpec.html) component, named ${component.name}Test.

[CUnit sources](#413csuo)

Gradle will create a [CSourceSet](http://docs.google.com/dsl/org.gradle.language.c.CSourceSet.html) named 'cunit' for each [CUnitTestSuiteSpec](http://docs.google.com/dsl/org.gradle.nativeplatform.test.cunit.CUnitTestSuiteSpec.html) component in the project. This source set should contain the cunit test files for the component under test. Source files can be located in the conventional location (src/${component.name}Test/cunit) or can be configured like any other source set.

Gradle initialises the CUnit test registry and executes the tests, utilising some generated CUnit launcher sources. Gradle will expect and call a function with the signature void gradle\_cunit\_register() that you can use to configure the actual CUnit suites and tests to execute.

[Example: Registering CUnit tests](#2g8n32h)

**suite\_operators.c**

#include <CUnit/Basic.h>  
#include "gradle\_cunit\_register.h"  
#include "test\_operators.h"  
  
int suite\_init(**void**) {  
 **return** 0;  
}  
  
int suite\_clean(**void**) {  
 **return** 0;  
}  
  
**void** gradle\_cunit\_register() {  
 CU\_pSuite pSuiteMath = CU\_add\_suite("operator tests", suite\_init, suite\_clean);  
 CU\_add\_test(pSuiteMath, "test\_plus", test\_plus);  
 CU\_add\_test(pSuiteMath, "test\_minus", test\_minus);  
}

| **Note** | Due to this mechanism, your CUnit sources may not contain a main method since this will clash with the method provided by Gradle. |
| --- | --- |

[Building CUnit executables](#vdxdaa)

A [CUnitTestSuiteSpec](http://docs.google.com/dsl/org.gradle.nativeplatform.test.cunit.CUnitTestSuiteSpec.html) component has an associated [NativeExecutableSpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeExecutableSpec.html) or [NativeLibrarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeLibrarySpec.html) component. For each [NativeBinarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeBinarySpec.html) configured for the main component, a matching [CUnitTestSuiteBinarySpec](http://docs.google.com/dsl/org.gradle.nativeplatform.test.cunit.CUnitTestSuiteBinarySpec.html) will be configured on the test suite component. These test suite binaries can be configured in a similar way to any other binary instance:

[Example: Configuring CUnit tests](#3fdkvy3)

**build.gradle**

model {  
 binaries {  
 withType(CUnitTestSuiteBinarySpec) {  
 lib library: "cunit", linkage: "static"  
  
 **if** (flavor == flavors.failing) {  
 cCompiler.define "PLUS\_BROKEN"  
 }  
 }  
 }  
}

| **Note** | Both the CUnit sources provided by your project and the generated launcher require the core CUnit headers and libraries. Presently, this library dependency must be provided by your project for each [CUnitTestSuiteBinarySpec](http://docs.google.com/dsl/org.gradle.nativeplatform.test.cunit.CUnitTestSuiteBinarySpec.html). |
| --- | --- |

[Running CUnit tests](#1uiv65w)

For each [CUnitTestSuiteBinarySpec](http://docs.google.com/dsl/org.gradle.nativeplatform.test.cunit.CUnitTestSuiteBinarySpec.html), Gradle will create a task to execute this binary, which will run all of the registered CUnit tests. Test results will be found in the *${build.dir}*/test-results directory.

[Example: Running CUnit tests](#4eiiotp)

**build.gradle**

apply plugin: "c"  
apply plugin: 'cunit-test-suite'  
  
model {  
 flavors {  
 passing  
 failing  
 }  
 platforms {  
 x86 {  
 architecture "x86"  
 }  
 }  
 repositories {  
 libs(PrebuiltLibraries) {  
 cunit {  
 headers.srcDir "libs/cunit/2.1-2/include"  
 binaries.withType(StaticLibraryBinary) {  
 staticLibraryFile =  
 file("libs/cunit/2.1-2/lib/" +  
 findCUnitLibForPlatform(targetPlatform))  
 }  
 }  
 }  
 }  
 components {  
 operators(NativeLibrarySpec) {  
 targetPlatform "x86"  
 }  
 }  
 testSuites {  
 operatorsTest(CUnitTestSuiteSpec) {  
 testing $.components.operators  
 }  
 }  
}  
model {  
 binaries {  
 withType(CUnitTestSuiteBinarySpec) {  
 lib library: "cunit", linkage: "static"  
  
 **if** (flavor == flavors.failing) {  
 cCompiler.define "PLUS\_BROKEN"  
 }  
 }  
 }  
}

**Output of gradle -q runOperatorsTestFailingCUnitExe**

> gradle -q runOperatorsTestFailingCUnitExe  
  
There were test failures:  
 1. /home/user/gradle/samples/src/operatorsTest/c/test\_plus.c:6 - plus(0, -2) == -2  
 2. /home/user/gradle/samples/src/operatorsTest/c/test\_plus.c:7 - plus(2, 2) == 4  
  
  
FAILURE: Build failed with an exception.  
  
\* What went wrong:  
Execution failed for task ':runOperatorsTestFailingCUnitExe'.  
> There were failing tests. See the results at: file:///home/user/gradle/samples/build/test-results/operatorsTest/failing/  
  
\* 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

| **Note** | The code for this example can be found at samples/native-binaries/cunit in the ‘-all’ distribution of Gradle. |
| --- | --- |

| **Note** | The current support for CUnit is quite rudimentary. Plans for future integration include:   * Allow tests to be declared with Javadoc-style annotations. * Improved HTML reporting, similar to that available for JUnit. * Real-time feedback for test execution. * Support for additional test frameworks. |
| --- | --- |

[GoogleTest support](#335zgpr)

The Gradle google-test plugin provides support for compiling and executing GoogleTest tests in your native-binary project. For each [NativeExecutableSpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeExecutableSpec.html) and [NativeLibrarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeLibrarySpec.html) defined in your project, Gradle will create a matching [GoogleTestTestSuiteSpec](http://docs.google.com/dsl/org.gradle.nativeplatform.test.googletest.GoogleTestTestSuiteSpec.html) component, named ${component.name}Test.

[GoogleTest sources](#2tnsz1i)

Gradle will create a [CppSourceSet](http://docs.google.com/dsl/org.gradle.language.cpp.CppSourceSet.html) named 'cpp' for each [GoogleTestTestSuiteSpec](http://docs.google.com/dsl/org.gradle.nativeplatform.test.googletest.GoogleTestTestSuiteSpec.html) component in the project. This source set should contain the GoogleTest test files for the component under test. Source files can be located in the conventional location (src/${component.name}Test/cpp) or can be configured like any other source set.

[Building GoogleTest executables](#18t399b)

A [GoogleTestTestSuiteSpec](http://docs.google.com/dsl/org.gradle.nativeplatform.test.googletest.GoogleTestTestSuiteSpec.html) component has an associated [NativeExecutableSpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeExecutableSpec.html) or [NativeLibrarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeLibrarySpec.html) component. For each [NativeBinarySpec](http://docs.google.com/javadoc/org/gradle/nativeplatform/NativeBinarySpec.html) configured for the main component, a matching [GoogleTestTestSuiteBinarySpec](http://docs.google.com/dsl/org.gradle.nativeplatform.test.googletest.GoogleTestTestSuiteBinarySpec.html) will be configured on the test suite component. These test suite binaries can be configured in a similar way to any other binary instance:

[Example: Registering GoogleTest tests](#3ssqrx4)

**build.gradle**

model {  
 binaries {  
 withType(GoogleTestTestSuiteBinarySpec) {  
 lib library: "googleTest", linkage: "static"  
  
 **if** (flavor == flavors.failing) {  
 cppCompiler.define "PLUS\_BROKEN"  
 }  
  
 **if** (targetPlatform.operatingSystem.linux) {  
 cppCompiler.args '-pthread'  
 linker.args '-pthread'  
  
 **if** (toolChain **instanceof** Gcc || toolChain **instanceof** Clang) {  
 *// Use C++03 with the old ABIs, as this is what the googletest binaries were built with*  
 cppCompiler.args '-std=c++03', '-D\_GLIBCXX\_USE\_CXX11\_ABI=0'  
 linker.args '-std=c++03'  
 }  
 }  
 }  
 }  
}

| **Note** | The code for this example can be found at samples/native-binaries/google-test in the ‘-all’ distribution of Gradle. |
| --- | --- |

| **Note** | The GoogleTest sources provided by your project require the core GoogleTest headers and libraries. Presently, this library dependency must be provided by your project for each [GoogleTestTestSuiteBinarySpec](http://docs.google.com/dsl/org.gradle.nativeplatform.test.googletest.GoogleTestTestSuiteBinarySpec.html). |
| --- | --- |

[Running GoogleTest tests](#27y124x)

For each [GoogleTestTestSuiteBinarySpec](http://docs.google.com/dsl/org.gradle.nativeplatform.test.googletest.GoogleTestTestSuiteBinarySpec.html), Gradle will create a task to execute this binary, which will run all of the registered GoogleTest tests. Test results will be found in the *${build.dir}*/test-results directory.

| **Note** | The current support for GoogleTest is quite rudimentary. Plans for future integration include:   * Improved HTML reporting, similar to that available for JUnit. * Real-time feedback for test execution. * Support for additional test frameworks. |
| --- | --- |

[Software model concepts](#1664s55)

| **Caution** | Rule based configuration [will be deprecated](https://blog.gradle.org/state-and-future-of-the-gradle-software-model). New plugins should not use this concept. |
| --- | --- |

The software model describes how a piece of software is built and how the components of the software relate to each other. The software model is organized around some key concepts:

* A *component* is a general concept that represents some logical piece of software. Examples of components are a command-line application, a web application or a library. A component is often composed of other components. Most Gradle builds will produce at least one component.
* A *library* is a reusable component that is linked into or combined into some other component. In the Java ecosystem, a library is often built as a Jar file, and then later bundled into an application of some kind. In the native ecosystem, a library may be built as a shared library or static library, or both.
* A *source set* represents a logical group of source files. Most components are built from source sets of various languages. Some source sets contain source that is written by hand, and some source sets may contain source that is generated from something else.
* A *binary* represents some output that is built for a component. A component may produce multiple different output binaries. For example, for a C++ library, both a shared library and a static library binary may be produced. Each binary is initially configured to be built from the component sources, but additional source sets can be added to specific binary variants.
* A *variant* represents some mutually exclusive binary of a component. A library, for example, might target Java 7 and Java 8, effectively producing two distinct binaries: a Java 7 Jar and a Java 8 Jar. These are different variants of the library.
* The *API* of a library represents the artifacts and dependencies that are required to compile against that library. The API typically consists of a binary together with a set of dependencies.

[Rule based model configuration](#3q5sasy)

| **Caution** | Rule based configuration [will be deprecated](https://blog.gradle.org/state-and-future-of-the-gradle-software-model). New plugins should not use this concept. |
| --- | --- |

Rule based model configuration enables *configuration logic to itself have dependencies* on other elements of configuration, and to make use of the resolved states of those other elements of configuration while performing its own configuration.

[Background](#n3bccq)

In a nutshell, the Software Model is a very declarative way to describe how a piece of software is built and the other components it needs as dependencies in the process. It also provides a new, rule-based engine for configuring a Gradle build. When we started to implement the software model we set ourselves the following goals:

* Improve configuration and execution time performance.
* Make customizations of builds with complex tool chains easier.
* Provide a richer, more standardized way to model different software ecosystems.

Gradle drastically improved configuration performance through other measures. There is no longer any need for a drastic, incompatible change in how Gradle builds are configured. Gradle’s support for building [native software](#2r0uhxc) and [Play Framework applications](#40ew0vw) still use the configuration model.

[Basic Concepts](#372yv0j)

[The “model space”](#1m8958c)

The term “model space” is used to refer to the formal model, which can be read and modified by rules.

A counterpart to the model space is the “project space”, which should be familiar to readers. The “project space” is a graph of objects (e.g project.repositories, project.tasks etc.) having a [Project](http://docs.google.com/dsl/org.gradle.api.Project.html) as its root. A build script is effectively adding and configuring objects of this graph. For the most part, the “project space” is opaque to Gradle. It is an arbitrary graph of objects that Gradle only partially understands.

Each project also has its own model space, which is distinct from the project space. A key characteristic of the “model space” is that Gradle knows much more about it (which is knowledge that can be put to good use). The objects in the model space are “managed”, to a greater extent than objects in the project space. The origin, structure, state, collaborators and relationships of objects in the model space are first class constructs. This is effectively the characteristic that functionally distinguishes the model space from the project space: the objects of the model space are defined in ways that Gradle can understand them intimately, as opposed to an object that is the result of running relatively opaque code. A “rule” is effectively a building block of this definition.

The model space will eventually replace the project space, becoming the only “space”.

[Rules](#467wnw5)

The model space is defined by “rules”. A rule is just a function (in the abstract sense) that either produces a model element, or acts upon a model element. Every rule has a single subject and zero or more inputs. Only the subject can be changed by a rule, while the inputs are effectively immutable.

Gradle guarantees that all inputs are fully “realized“ before the rule executes. The process of “realizing” a model element is effectively executing all the rules for which it is the subject, transitioning it to its final state. There is a strong analogy here to Gradle’s task graph and task execution model. Just as tasks depend on each other and Gradle ensures that dependencies are satisfied before executing a task, rules effectively depend on each other (i.e. a rule depends on all rules whose subject is one of the inputs) and Gradle ensures that all dependencies are satisfied before executing the rule.

Model elements are very often defined in terms of other model elements. For example, a compile task’s configuration can be defined in terms of the configuration of the source set that it is compiling. In this scenario, the compile task would be the subject of a rule and the source set an input. Such a rule could configure the task subject based on the source set input without concern for how it was configured, who it was configured by or when the configuration was specified.

There are several ways to declare rules, and in several forms.

[Rule sources](#2ld6y3y)

One way to define rules is via a [RuleSource](http://docs.google.com/javadoc/org/gradle/model/RuleSource.html) subclass. If an object extends RuleSource and contains any methods annotated by '@Mutate', then each such method defines a rule. For each such method, the first argument is the subject, and zero or more subsequent arguments may follow and are inputs of the rule.

[Example: applying a rule source plugin](#10ih8br)

**build.gradle**

@Managed  
**interface** Person {  
 **void** setFirstName(String name)  
 String getFirstName()  
  
 **void** setLastName(String name)  
 String getLastName()  
}  
  
**class** **PersonRules** **extends** RuleSource {  
 @Model **void** person(Person p) {}  
  
 *//Create a rule that modifies a Person and takes no other inputs*  
 @Mutate **void** setFirstName(Person p) {  
 p.firstName = "John"  
 }  
  
 *//Create a rule that modifies a ModelMap<Task> and takes as input a Person*  
 @Mutate **void** createHelloTask(ModelMap<Task> tasks, Person p) {  
 tasks.create("hello") {  
 doLast {  
 println "Hello $p.firstName $p.lastName!"  
 }  
 }  
 }  
}  
  
apply plugin: PersonRules

**Output of** gradle hello

> gradle hello  
  
> Task :hello  
Hello John Smith!  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

Each of the different methods of the rule source are discrete, independent rules. Their order, or the fact that they belong to the same class, do not affect their behavior.

[Example: a model creation rule](#3ki4qzk)

**build.gradle**

@Model **void** person(Person p) {}

This rule declares that there is a model element at path "person" (defined by the method name), of type Person. This is the form of the [Model](http://docs.google.com/javadoc/org/gradle/model/Model.html) type rule for [Managed](http://docs.google.com/javadoc/org/gradle/model/Managed.html) types. Here, the person object is the rule subject. The method could potentially have a body, that mutated the person instance. It could also potentially have more parameters, which would be the rule inputs.

[Example: a model mutation rule](#1znf17d)

**build.gradle**

*//Create a rule that modifies a Person and takes no other inputs*  
@Mutate **void** setFirstName(Person p) {  
 p.firstName = "John"  
}

This [Mutate](http://docs.google.com/javadoc/org/gradle/model/Mutate.html) rule mutates the person object. The first parameter to the method is the subject. Here, a by-type reference is used as no [Path](http://docs.google.com/javadoc/org/gradle/model/Path.html) annotation is present on the parameter. It could also potentially have more parameters, that would be the rule inputs.

[Example: creating a task](#4jn2jv6)

**build.gradle**

*//Create a rule that modifies a ModelMap<Task> and takes as input a Person*  
@Mutate **void** createHelloTask(ModelMap<Task> tasks, Person p) {  
 tasks.create("hello") {  
 doLast {  
 println "Hello $p.firstName $p.lastName!"  
 }  
 }  
}

This [Mutate](http://docs.google.com/javadoc/org/gradle/model/Mutate.html) rule effectively adds a task, by mutating the tasks collection. The subject here is the "tasks" node, which is available as a [ModelMap](http://docs.google.com/javadoc/org/gradle/model/ModelMap.html) of [Task](http://docs.google.com/dsl/org.gradle.api.Task.html). The only input is our person element. As the person is being used as an input here, it will have been realised before executing this rule. That is, the task container effectively *depends on* the person element. If there are other configuration rules for the person element, potentially specified in a build script or other plugin, they will also be guaranteed to have been executed.

As Person is a [Managed](http://docs.google.com/javadoc/org/gradle/model/Managed.html) type in this example, any attempt to modify the person parameter in this method would result in an exception being thrown. Managed objects enforce immutability at the appropriate point in their lifecycle.

Rule source plugins can be packaged and distributed in the same manner as other types of plugins (see [Custom Plugins](#279ka65)). They also may be applied in the same manner (to project objects) as [Plugin](http://docs.google.com/javadoc/org/gradle/api/Plugin.html) implementations (i.e. via [Project.apply(java.util.Map)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:apply(java.util.Map))).

Please see the documentation for [RuleSource](http://docs.google.com/javadoc/org/gradle/model/RuleSource.html) for more information on constraints on how rule sources must be implemented and for more types of rules.

[Advanced Concepts](#2yscu2z)

[Model paths](#1dxn4as)

A model path identifies the location of an element relative to the root of its model space. A common representation is a period-delimited set of names. For example, the model path "tasks" is the path to the element that is the task container. Assuming a task whose name is hello, the path "tasks.hello" is the path to this task.

[Managed model elements](#3xxamyl)

Currently, any kind of Java object can be part of the model space. However, there is a difference between “managed” and “unmanaged” objects.

A “managed” object is transparent and enforces immutability once realized. Being transparent means that its structure is understood by the rule infrastructure and as such each of its properties are also individual elements in the model space.

An “unmanaged” object is opaque to the model space and does not enforce immutability. Over time, more mechanisms will be available for defining managed model elements culminating in all model elements being managed in some way.

Managed models can be defined by attaching the @Managed annotation to an interface:

[Example: a managed type](#2d2kx6e)

**build.gradle**

@Managed  
**interface** Person {  
 **void** setFirstName(String name)  
 String getFirstName()  
  
 **void** setLastName(String name)  
 String getLastName()  
}

By defining a getter/setter pair, you are effectively declaring a managed property. A managed property is a property for which Gradle will enforce semantics such as immutability when a node of the model is not the subject of a rule. Therefore, this example declares properties named *firstName* and *lastName* on the managed type *Person*. These properties will only be writable when the view is mutable, that is to say when the *Person* is the subject of a Rule (see below the explanation for rules).

Managed properties can be of any scalar type. In addition, properties can also be of any type which is itself managed:

| **Property type** | **Nullable** | **Example** |
| --- | --- | --- |
| String | Yes | **void** setFirstName(String name) String getFirstName() |
| File | Yes | **void** setHomeDirectory(File homeDir) File getHomeDirectory() |
| Integer, Boolean, Byte, Short, Float, Long, Double | Yes | **void** setId(Long id) Long getId() |
| int, boolean, byte, short, float, long, double | No | **void** setEmployed(**boolean** isEmployed) **boolean** isEmployed()  **void** setAge(**int** age) **int** getAge() |
| Another *managed* type. | Only if read/write | **void** setMother(Person mother) Person getMother() |
| An *enumeration* type. | Yes | **void** setMaritalStatus(MaritalStatus status) MaritalStatus getMaritalStatus() |
| A ManagedSet. A managed set supports the creation of new named model elements, but not their removal. | Only if read/write | ModelSet<Person> getChildren() |
| A Set or List of scalar types. All classic operations on collections are supported: add, remove, clear…​ | Only if read/write | **void** setUserGroups(List<String> groups) List<String> getUserGroups() |

If the type of a property is itself a managed type, it is possible to declare only a getter, in which case you are declaring a read-only property. A read-only property will be instantiated by Gradle, and cannot be replaced with another object of the same type (for example calling a setter). However, the properties of that property can potentially be changed, if, and only if, the property is the subject of a rule. If it’s not the case, the property is immutable, like any classic read/write managed property, and properties of the property cannot be changed at all.

Managed types can be defined out of interfaces or abstract classes and are usually defined in plugins, which are written either in Java or Groovy. Please see the [Managed](http://docs.google.com/javadoc/org/gradle/model/Managed.html) annotation for more information on creating managed model objects.

[Model element types](#s7v7e7)

There are particular types (language types) supported by the model space and can be generalised as follows:

Table 11. Type definitions

| **Type** | **Definition** |
| --- | --- |
| Scalar | A scalar type is one of the following:   * a primitive type (e.g. int) or its boxed type (e.g Integer) * a BigInteger or BigDecimal * a String * a File * an enumeration type |
| Scalar Collection | A java.util.List or java.util.Set containing one of the scalar types |
| Managed type | Any class which is a valid managed model (i.e.annotated with @[Managed](http://docs.google.com/javadoc/org/gradle/model/Managed.html)) |
| Managed collection | A [ModelMap](http://docs.google.com/javadoc/org/gradle/model/ModelMap.html) or [ModelSet](http://docs.google.com/javadoc/org/gradle/model/ModelSet.html) |

There are various contexts in which these types can be used:

Table 12. Model type support

| **Context** | **Supported types** |
| --- | --- |
| Creating top level model elements | * Any managed type * [FunctionalSourceSet](http://docs.google.com/javadoc/org/gradle/language/base/FunctionalSourceSet.html) (when the [LanguageBasePlugin](http://docs.google.com/javadoc/org/gradle/language/base/plugins/LanguageBasePlugin.html) plugin has been applied) * Subtypes of [LanguageSourceSet](http://docs.google.com/javadoc/org/gradle/language/base/LanguageSourceSet.html) which have been registered via [ComponentType](http://docs.google.com/javadoc/org/gradle/platform/base/ComponentType.html) |
| Properties of managed model elements | The properties (attributes) of a managed model elements may be one or more of the following:   * A managed type * A type which is annotated with @[Unmanaged](http://docs.google.com/javadoc/org/gradle/model/Unmanaged.html) * A Scalar Collection * A Managed collection containing managed types * A Managed collection containing [FunctionalSourceSet](http://docs.google.com/javadoc/org/gradle/language/base/FunctionalSourceSet.html)'s (when the [LanguageBasePlugin](http://docs.google.com/javadoc/org/gradle/language/base/plugins/LanguageBasePlugin.html) plugin has been applied) * Subtypes of [LanguageSourceSet](http://docs.google.com/javadoc/org/gradle/language/base/LanguageSourceSet.html) which have been registered via [ComponentType](http://docs.google.com/javadoc/org/gradle/platform/base/ComponentType.html) |

[Language source sets](#3c7iq20)

[FunctionalSourceSet](http://docs.google.com/javadoc/org/gradle/language/base/FunctionalSourceSet.html)s and subtypes of [LanguageSourceSet](http://docs.google.com/javadoc/org/gradle/language/base/LanguageSourceSet.html) (which have been registered via [ComponentType](http://docs.google.com/javadoc/org/gradle/platform/base/ComponentType.html)) can be added to the model space via rules or via the model DSL.

[Example: Strongly modelling sources sets](#1rct09t)

**build.gradle**

apply plugin: 'java-lang'  
  
*//Creating LanguageSourceSets via rules*  
**class** **LanguageSourceSetRules** **extends** RuleSource {  
 @Model  
 **void** mySourceSet(JavaSourceSet javaSource) {  
 javaSource.source.srcDir("src/main/my")  
 }  
}  
apply plugin: LanguageSourceSetRules  
  
*//Creating LanguageSourceSets via the model DSL*  
model {  
 another(JavaSourceSet) {  
 source {  
 srcDir "src/main/another"  
 }  
 }  
}  
  
*//Using FunctionalSourceSets*  
@Managed  
**interface** SourceBundle {  
 FunctionalSourceSet getFreeSources()  
 FunctionalSourceSet getPaidSources()  
}  
model {  
 sourceBundle(SourceBundle) {  
 freeSources.create("main", JavaSourceSet)  
 freeSources.create("resources", JvmResourceSet)  
 paidSources.create("main", JavaSourceSet)  
 paidSources.create("resources", JvmResourceSet)  
 }  
}

**Output of gradle help**

> gradle help  
  
> Task :help

| **Note** | The code for this example can be found at samples/modelRules/language-support in the ‘-all’ distribution of Gradle. |
| --- | --- |

[References, binding and scopes](#4bcgixm)

As previously mentioned, a rule has a subject and zero or more inputs. The rule’s subject and inputs are declared as “references” and are “bound” to model elements before execution by Gradle. Each rule must effectively forward declare the subject and inputs as references. Precisely how this is done depends on the form of the rule. For example, the rules provided by a [RuleSource](http://docs.google.com/javadoc/org/gradle/model/RuleSource.html) declare references as method parameters.

A reference is either “by-path” or “by-type”.

A “by-type” reference identifies a particular model element by its type. For example, a reference to the [TaskContainer](http://docs.google.com/javadoc/org/gradle/api/tasks/TaskContainer.html) effectively identifies the "tasks" element in the project model space. The model space is not exhaustively searched for candidates for by-type binding; rather, a rule is given a scope (discussed later) that determines the search space for a by-type binding.

A “by-path” reference identifies a particular model element by its path in model space. By-path references are always relative to the rule scope; there is currently no way to path “out” of the scope. All by-path references also have an associated type, but this does not influence what the reference binds to. The element identified by the path must however by type compatible with the reference, or a fatal “binding failure” will occur.

[Binding scope](#2qhqt5f)

Rules are bound within a “scope”, which determines how references bind. Most rules are bound at the project scope (i.e. the root of the model graph for the project). However, rules can be scoped to a node within the graph. The [ModelMap.named(java.lang.String, java.lang.Class)](http://docs.google.com/javadoc/org/gradle/model/ModelMap.html#named-java.lang.String,%20java.lang.Class-) method is an example of a mechanism for applying scoped rules. Rules declared in the build script using the model {} block, or via a RuleSource applied as a plugin use the root of the model space as the scope. This can be considered the default scope.

By-path references are always relative to the rule scope. When the scope is the root, this effectively allows binding to any element in the graph. When it is not, then only the children of the scope can be referenced using "by-path" notation.

When binding by-type references, the following elements are considered:

* The scope element itself.
* The immediate children of the scope element.
* The immediate children of the model space (i.e. project space) root.

For the common case, where the rule is effectively scoped to the root, only the immediate children of the root need to be considered.

[Binding to all elements in a scope matching type](#15n13d8)

Mutating or validating all elements of a given type in some scope is a common use-case. To accommodate this, rules can be applied via the @Each annotation.

In the example below, a @Defaults rule is applied to each FileItem in the model setting a default file size of "1024". Another rule applies a [RuleSource](http://docs.google.com/javadoc/org/gradle/model/RuleSource.html) to every DirectoryItem that makes sure all file sizes are positive and divisible by "16".

[Example: a DSL example applying a rule to every element in a scope](#3pmom11)

**build.gradle**

@Managed **interface** Item **extends** Named {}  
@Managed **interface** FileItem **extends** Item {  
 **void** setSize(**int** size)  
 **int** getSize()  
}  
@Managed **interface** DirectoryItem **extends** Item {  
 ModelMap<Item> getChildren()  
}  
  
**class** **PluginRules** **extends** RuleSource {  
 @Defaults **void** setDefaultFileSize(@Each FileItem file) {  
 file.size = 1024  
 }  
  
 @Rules **void** applyValidateRules(ValidateRules rules, @Each DirectoryItem directory) {}  
}  
apply plugin: PluginRules  
  
**abstract** **class** **ValidateRules** **extends** RuleSource {  
 @Validate  
 **void** validateSizeIsPositive(ModelMap<FileItem> files) {  
 files.each { file ->  
 **assert** file.size > 0  
 }  
 }  
  
 @Validate  
 **void** validateSizeDivisibleBySixteen(ModelMap<FileItem> files) {  
 files.each { file ->  
 **assert** file.size % 16 == 0  
 }  
 }  
}  
  
model {  
 root(DirectoryItem) {  
 children {  
 dir(DirectoryItem) {  
 children {  
 file1(FileItem)  
 file2(FileItem) { size = 2048 }  
 }  
 }  
 file3(FileItem)  
 }  
 }  
}

| **Note** | The code for this example can be found at samples/modelRules/ruleSourcePluginEach in the ‘-all’ distribution of Gradle. |
| --- | --- |

[The model DSL](#24ryw8u)

In addition to using a RuleSource, it is also possible to declare a model and rules directly in a build script using the “model DSL”.

| **Tip** | The model DSL makes heavy use of various Groovy DSL features. Please have a read of [Groovy DSL basics](#utdd4w) for an introduction to these Groovy features. |
| --- | --- |

The general form of the model DSL is:

model {  
 «rule-definitions»  
}

All rules are nested inside a model block. There may be any number of rule definitions inside each model block, and there may be any number of model blocks in a build script. You can also use a model block in build scripts that are applied using apply from: $uri.

There are currently 2 kinds of rule that you can define using the model DSL: configuration rules, and creation rules.

[Configuration rules](#jx96gn)

You can define a rule that configures a particular model element. A configuration rule has the following form:

model {  
 «model-path-to-subject» {  
 «configuration code»  
 }  
}

Continuing with the example so far of the model element "person" of type Person being present, the following DSL snippet adds a configuration rule for the person that sets its lastName property.

[Example: DSL configuration rule](#33wwp4g)

**build.gradle**

model {  
 person {  
 lastName = "Smith"  
 }  
}

A configuration rule specifies a path to the subject that should be configured and a closure containing the code to run when the subject is configured. The closure is executed with the subject passed as the closure delegate. Exactly what code you can provide in the closure depends on the type of the subject. This is discussed below.

You should note that the configuration code is not executed immediately but is instead executed only when the subject is required. This is an important behaviour of model rules and allows Gradle to configure only those elements that are required for the build, which helps reduce build time. For example, let’s run a task that uses the "person" object:

[Example: Configuration run when required](#1j26zc9)

**build.gradle**

model {  
 person {  
 println "configuring person"  
 lastName = "Smith"  
 }  
}

**Output of** gradle showPerson

> gradle showPerson  
configuring person  
  
> Task :showPerson  
Hello John Smith!  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

You can see that before the task is run, the "person" element is configured by running the rule closure. Now let’s run a task that does not require the "person" element:

[Example: Configuration not run when not required](#431ui02)

**Output of** gradle somethingElse

> gradle somethingElse  
  
> Task :somethingElse  
Not using person  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

In this instance, you can see that the "person" element is not configured at all.

[Creation rules](#2i74s7v)

It is also possible to create model elements at the root level. The general form of a creation rule is:

model {  
 «element-name»(«element-type») {  
 «initialization code»  
 }  
}

The following model rule creates the "person" element:

[Example: DSL creation rule](#xcf2fo)

**build.gradle**

model {  
 person(Person) {  
 firstName = "John"  
 }  
}

A creation rule definition specifies the path of the element to create, plus its public type, represented as a Java interface or class. Only certain types of model elements can be created.

A creation rule may also provide a closure containing the initialization code to run when the element is created. The closure is executed with the element passed as the closure delegate. Exactly what code you can provide in the closure depends on the type of the subject. This is discussed below.

The initialization closure is optional and can be omitted, for example:

[Example: DSL creation rule without initialization](#3hc2l3h)

**build.gradle**

model {  
 barry(Person)  
}

You should note that the initialization code is not executed immediately but is instead executed only when the element is required. The initialization code is executed before any configuration rules are run. For example:

[Example: Initialization before configuration](#1whcvba)

**build.gradle**

model {  
 person {  
 println "configuring person"  
 println "last name is $lastName, should be Smythe"  
 lastName = "Smythe"  
 }  
 person(Person) {  
 println "creating person"  
 firstName = "John"  
 lastName = "Smith"  
 }  
}

**Output of** gradle showPerson

> gradle showPerson  
creating person  
configuring person  
last name is Smith, should be Smythe  
  
> Task :showPerson  
Hello John Smythe!  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

Notice that the creation rule appears in the build script *after* the configuration rule, but its code runs before the code of the configuration rule. Gradle collects up all the rules for a particular subject before running any of them, then runs the rules in the appropriate order.

[Model rule closures](#4gh0dz3)

Most DSL rules take a closure containing some code to run to configure the subject. The code you can use in this closure depends on the type of the subject of the rule.

| **Tip** | You can use the [model report](#16x20ju) to determine the type of a particular model element. |
| --- | --- |

In general, a rule closure may contain arbitrary code, mixed with some type specific DSL syntax.

[ModelMap<T> subject](#2vmao6w)

A [ModelMap](http://docs.google.com/javadoc/org/gradle/model/ModelMap.html) is basically a map of model elements, indexed by some name. When a ModelMap is used as the subject of a DSL rule, the rule closure can use any of the methods defined on the [ModelMap](http://docs.google.com/javadoc/org/gradle/model/ModelMap.html) interface.

A rule closure with ModelMap as a subject can also include nested creation or configuration rules. These behave in a similar way to the creation and configuration rules that appear directly under the model block.

Here is an example of a nested creation rule:

[Example: Nested DSL creation rule](#1arkyep)

**build.gradle**

model {  
 people {  
 john(Person) {  
 firstName = "John"  
 }  
 }  
}

As before, a nested creation rule defines a name and public type for the element, and optionally, a closure containing code to use to initialize the element. The code is run only when the element is required in the build.

Here is an example of a nested configuration rule:

[Example: Nested DSL configuration rule](#3ur8h2i)

**build.gradle**

model {  
 people {  
 john {  
 lastName = "Smith"  
 }  
 }  
}

As before, a nested configuration rule defines the name of the element to configure and a closure containing code to use to configure the element. The code is run only when the element is required in the build.

ModelMap introduces several other kinds of rules. For example, you can define a rule that targets each of the elements in the map. The code in the rule closure is executed once for each element in the map, when that element is required. Let’s run a task that requires all of the children of the "people" element:

[Example: DSL configuration rule for each element in a map](#29wirab)

**build.gradle**

model {  
 people {  
 john(Person) {  
 println "creating $it"  
 firstName = "John"  
 lastName = "Smith"  
 }  
 all {  
 println "configuring $it"  
 }  
 barry(Person) {  
 println "creating $it"  
 firstName = "Barry"  
 lastName = "Barry"  
 }  
 }  
}

**Output of** gradle listPeople

> gradle listPeople  
creating Person 'people.barry'  
configuring Person 'people.barry'  
creating Person 'people.john'  
configuring Person 'people.john'  
  
> Task :listPeople  
Hello Barry Barry!  
Hello John Smith!  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

Any method on [ModelMap](http://docs.google.com/javadoc/org/gradle/model/ModelMap.html) that accepts an [Action](http://docs.google.com/javadoc/org/gradle/api/Action.html) as its last parameter can also be used to define a nested rule.

[@Managed type subject](#p1t1i4)

When a managed type is used as the subject of a DSL rule, the rule closure can use any of the methods defined on the managed type interface.

A rule closure can also configure the properties of the element using nested closures. For example:

[Example: Nested DSL property configuration](#391gk5x)

**build.gradle**

model {  
 person {  
 address {  
 city = "Melbourne"  
 }  
 }  
}

| **Note** | Currently, the nested closures do not define rules and are executed immediately. Please be aware that this behaviour will change in a future Gradle release. |
| --- | --- |

[All other subjects](#1o6qudq)

For all other types, the rule closure can use any of the methods defined by the type. There is no special DSL defined for these elements.

[Automatic type coercion](#486ed1j)

Scalar properties in managed types can be assigned CharSequence values (e.g. String, GString, etc.) and they will be converted to the actual property type for you. This works for all scalar types including `File`s, which will be resolved relative to the current project.

[Example: a DSL example showing type conversions](#2nbon9c)

**build.gradle**

**enum** Temperature {  
 TOO\_HOT,  
 TOO\_COLD,  
 JUST\_RIGHT  
}  
  
@Managed  
**interface** Item {  
 **void** setName(String n); String getName()  
  
 **void** setQuantity(**int** q); **int** getQuantity()  
  
 **void** setPrice(**float** p); **float** getPrice()  
  
 **void** setTemperature(Temperature t)  
 Temperature getTemperature()  
  
 **void** setDataFile(File f); File getDataFile()  
}  
  
**class** **ItemRules** **extends** RuleSource {  
 @Model  
 **void** item(Item item) {  
 **def** data = item.dataFile.text.trim()  
 **def** (name, quantity, price, temp) = data.split(',')  
 item.name = name  
 item.quantity = quantity  
 item.price = price  
 item.temperature = temp  
 }  
  
 @Defaults  
 **void** setDefaults(Item item) {  
 item.dataFile = 'data.csv'  
 }  
  
 @Mutate  
 **void** createDataTask(ModelMap<Task> tasks, Item item) {  
 tasks.create('showData') {  
 doLast {  
 println """  
Item '$item.name'  
 quantity: $item.quantity  
 price: $item.price  
 temperature: $item.temperature"""  
 }  
 }  
 }  
}  
  
apply plugin: ItemRules  
  
model {  
 item {  
 price = "${price \* (quantity < 10 ? 2 : 0.5)}"  
 }  
}

| **Note** | The code for this example can be found at samples/modelRules/modelDslCoercion in the ‘-all’ distribution of Gradle. |
| --- | --- |

In the above example, an Item is created and is initialized in setDefaults() by providing the path to the data file. In the item() method the resolved File is parsed to extract and set the data. In the DSL block at the end, the price is adjusted based on the quantity; if there are fewer than 10 remaining the price is doubled, otherwise it is reduced by 50%. The GString expression is a valid value since it resolves to a float value in string form.

Finally, in createDataTask() we add the showData task to display all of the configured values.

[Declaring input dependencies](#12gyxh5)

Rules declared in the DSL may *depend* on other model elements through the use of a special syntax, which is of the form:

$.«path-to-model-element»

Paths are a period separated list of identifiers. To directly depend on the firstName of the person, the following could be used:

$.person.firstName

[Example: a DSL rule using inputs](#3mgmg4y)

**build.gradle**

model {  
 tasks {  
 hello(Task) {  
 **def** p = $.person  
 doLast {  
 println "Hello $p.firstName $p.lastName!"  
 }  
 }  
 }  
}

| **Note** | The code for this example can be found at samples/modelRules/modelDsl in the ‘-all’ distribution of Gradle. |
| --- | --- |

In the above snippet, the $.person construct is an input reference. The construct returns the value of the model element at the specified path, as its default type (i.e. the type advertised by the [Model Report](#16x20ju)). It may appear anywhere in the rule that an expression may normally appear. It is not limited to the right hand side of variable assignments.

The input element is guaranteed to be fully configured before the rule executes. That is, all of the rules that mutate the element are guaranteed to have been previously executed, leaving the target element in its final, immutable, state.

Most model elements enforce immutability when being used as inputs. Any attempt to mutate such an element will result in a runtime error. However, some legacy type objects do not currently implement such checks. Regardless, it is always invalid to attempt to mutate an input to a rule.

[Using ModelMap<T> as an input](#21lwqcr)

When you use a [ModelMap](http://docs.google.com/javadoc/org/gradle/model/ModelMap.html) as input, each item in the map is made available as a property.

[The model report](#16x20ju)

The built-in [ModelReport](http://docs.google.com/dsl/org.gradle.api.reporting.model.ModelReport.html) task displays a hierarchical view of the elements in the model space. Each item prefixed with a + on the model report is a model element and the visual nesting of these elements correlates to the model path (e.g. tasks.help). The model report displays the following details about each model element:

Table 13. Model report - model element details

| **Detail** | **Description** |
| --- | --- |
| Type | This is the underlying type of the model element and is typically a fully qualified class name. |
| Value | Is conditionally displayed on the report when a model element can be represented as a string. |
| Creator | Every model element has a creator. A creator signifies the origin of the model element (i.e. what created the model element). |
| Rules | Is a listing of the rules, excluding the creator rule, which are executed for a given model element. The order in which the rules are displayed reflects the order in which they are executed. |

[Example: Model task output](#gr70kk)

**Output of gradle model**

> gradle model  
  
> Task :model  
  
------------------------------------------------------------  
Root project  
------------------------------------------------------------  
  
+ person  
 | Type: Person  
 | Creator: PersonRules#person(Person)  
 | Rules:  
 ⤷ person { ... } @ build.gradle line 97, column 3  
 ⤷ PersonRules#setFirstName(Person)  
 + age  
 | Type: int  
 | Value: 0  
 | Creator: PersonRules#person(Person)  
 + children  
 | Type: org.gradle.model.ModelSet<Person>  
 | Creator: PersonRules#person(Person)  
 + employed  
 | Type: boolean  
 | Value: false  
 | Creator: PersonRules#person(Person)  
 + father  
 | Type: Person  
 | Value: null  
 | Creator: PersonRules#person(Person)  
 + firstName  
 | Type: java.lang.String  
 | Value: John  
 | Creator: PersonRules#person(Person)  
 + homeDirectory  
 | Type: java.io.File  
 | Value: null  
 | Creator: PersonRules#person(Person)  
 + id  
 | Type: java.lang.Long  
 | Value: null  
 | Creator: PersonRules#person(Person)  
 + lastName  
 | Type: java.lang.String  
 | Value: Smith  
 | Creator: PersonRules#person(Person)  
 + maritalStatus  
 | Type: MaritalStatus  
 | Creator: PersonRules#person(Person)  
 + mother  
 | Type: Person  
 | Value: null  
 | Creator: PersonRules#person(Person)  
 + userGroups  
 | Type: java.util.List<java.lang.String>  
 | Value: null  
 | Creator: PersonRules#person(Person)  
+ tasks  
 | Type: org.gradle.model.ModelMap<org.gradle.api.Task>  
 | Creator: Project.<init>.tasks()  
 | Rules:  
 ⤷ PersonRules#createHelloTask(ModelMap<Task>, Person)  
 + buildEnvironment  
 | Type: org.gradle.api.tasks.diagnostics.BuildEnvironmentReportTask  
 | Value: task ':buildEnvironment'  
 | Creator: Project.<init>.tasks.buildEnvironment()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + components  
 | Type: org.gradle.api.reporting.components.ComponentReport  
 | Value: task ':components'  
 | Creator: Project.<init>.tasks.components()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + dependencies  
 | Type: org.gradle.api.tasks.diagnostics.DependencyReportTask  
 | Value: task ':dependencies'  
 | Creator: Project.<init>.tasks.dependencies()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + dependencyInsight  
 | Type: org.gradle.api.tasks.diagnostics.DependencyInsightReportTask  
 | Value: task ':dependencyInsight'  
 | Creator: Project.<init>.tasks.dependencyInsight()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + dependentComponents  
 | Type: org.gradle.api.reporting.dependents.DependentComponentsReport  
 | Value: task ':dependentComponents'  
 | Creator: Project.<init>.tasks.dependentComponents()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + hello  
 | Type: org.gradle.api.Task  
 | Value: task ':hello'  
 | Creator: PersonRules#createHelloTask(ModelMap<Task>, Person) > create(hello)  
 | Rules:  
 ⤷ copyToTaskContainer  
 + help  
 | Type: org.gradle.configuration.Help  
 | Value: task ':help'  
 | Creator: Project.<init>.tasks.help()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + init  
 | Type: org.gradle.buildinit.tasks.InitBuild  
 | Value: task ':init'  
 | Creator: Project.<init>.tasks.init()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + model  
 | Type: org.gradle.api.reporting.model.ModelReport  
 | Value: task ':model'  
 | Creator: Project.<init>.tasks.model()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + projects  
 | Type: org.gradle.api.tasks.diagnostics.ProjectReportTask  
 | Value: task ':projects'  
 | Creator: Project.<init>.tasks.projects()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + properties  
 | Type: org.gradle.api.tasks.diagnostics.PropertyReportTask  
 | Value: task ':properties'  
 | Creator: Project.<init>.tasks.properties()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + tasks  
 | Type: org.gradle.api.tasks.diagnostics.TaskReportTask  
 | Value: task ':tasks'  
 | Creator: Project.<init>.tasks.tasks()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + wrapper  
 | Type: org.gradle.api.tasks.wrapper.Wrapper  
 | Value: task ':wrapper'  
 | Creator: Project.<init>.tasks.wrapper()  
 | Rules:  
 ⤷ copyToTaskContainer

[Limitations and future direction](#30quj8d)

*The rule engine that was part of the Software Model will be deprecated.* Everything under the model block will be ported as extensions to the current model. Native users will no longer have a separate extension model compared to the rest of the Gradle community, and they will be able to make use of the new variant aware dependency management. For more information, see the [blog post](https://blog.gradle.org/state-and-future-of-the-gradle-software-model) on the state and future of the software model.

[Implementing model rules in a plugin](#25b2l0r)

A plugin can define rules by extending [RuleSource](http://docs.google.com/javadoc/org/gradle/model/RuleSource.html) and adding methods that define the rules. The plugin class can either extend [RuleSource](http://docs.google.com/javadoc/org/gradle/model/RuleSource.html) directly or can implement [Plugin](http://docs.google.com/javadoc/org/gradle/api/Plugin.html) and include a nested [RuleSource](http://docs.google.com/javadoc/org/gradle/model/RuleSource.html) subclass.

Refer to the API docs for [RuleSource](http://docs.google.com/javadoc/org/gradle/model/RuleSource.html) for more details.

[Applying additional rules](#1fw4tg6)

A rule method annotated with [Rules](http://docs.google.com/javadoc/org/gradle/model/Rules.html) can apply a [RuleSource](http://docs.google.com/javadoc/org/gradle/model/RuleSource.html) to a target model element.

[Extending the software model](#kgcv8k)

| **Caution** | Rule based configuration [will be deprecated](https://blog.gradle.org/state-and-future-of-the-gradle-software-model). New plugins should not use this concept. |
| --- | --- |

[Introduction](#3zvsc3z)

One of the strengths of Gradle has always been its extensibility, and its adaptability to new domains. The software model takes this extensibility to a new level, enabling the deep modeling of specific domains via richly typed DSLs. The following chapter describes how the model and the corresponding DSLs can be extended to support domains like the [Play Framework](#40ew0vw) or [native software development](#2r0uhxc). Before reading this you should be familiar with the Gradle software model [rule based configuration](#3q5sasy) and [concepts](#1664s55).

The following build script is an example of using a custom software model for building Markdown based documentation:

[Example: an example of using a custom software model](#2f12mbs)

**build.gradle**

**import** sample.documentation.DocumentationComponent  
**import** sample.documentation.TextSourceSet  
**import** sample.markdown.MarkdownSourceSet  
  
apply plugin:sample.documentation.DocumentationPlugin  
apply plugin:sample.markdown.MarkdownPlugin  
  
model {  
 components {  
 docs(DocumentationComponent) {  
 sources {  
 reference(TextSourceSet)  
 userguide(MarkdownSourceSet) {  
 generateIndex = true  
 smartQuotes = true  
 }  
 }  
 }  
 }  
}

| **Note** | The code for this example can be found at samples/customModel/languageType/ in the ‘-all’ distribution of Gradle. |
| --- | --- |

The rest of this chapter is dedicated to explaining what is going on behind this build script.

[Concepts](#u6cwjl)

A custom software model type has a public type, a base interface and internal views. Multiple such types then collaborate to define a custom software model.

[Public type and base interfaces](#3e60f7e)

Extended types declare a *public type* that extends a *base interface*:

* Components extend the [ComponentSpec](http://docs.google.com/javadoc/org/gradle/platform/base/ComponentSpec.html) base interface
* Binaries extend the [BinarySpec](http://docs.google.com/javadoc/org/gradle/platform/base/BinarySpec.html) base interface
* Source sets extend the [LanguageSourceSet](http://docs.google.com/javadoc/org/gradle/language/base/LanguageSourceSet.html) base interface

The *public type* is exposed to build logic.

[Internal views](#1tbapf7)

Adding internal views to your model type, you can make some data visible to build logic via a public type, while hiding the rest of the data behind the internal view types. This is covered in a [dedicated section](#4day830) below.

[Components all the way down](#2sg8iat)

Components are composed of other components. A source set is just a special kind of component representing sources. It might be that the sources are provided, or generated. Similarly, some components are composed of different binaries, which are built by tasks. All buildable components are built by tasks. In the software model, you will write rules to generate both binaries from components and tasks from binaries.

[Components](#17lisim)

To declare a custom component type one must extend [ComponentSpec](http://docs.google.com/javadoc/org/gradle/platform/base/ComponentSpec.html), or one of the following, depending on the use case:

* [SourceComponentSpec](http://docs.google.com/javadoc/org/gradle/platform/base/SourceComponentSpec.html) represents a component which has sources
* [VariantComponentSpec](http://docs.google.com/javadoc/org/gradle/platform/base/VariantComponentSpec.html) represents a component which generates different binaries based on context (target platforms, build flavors, …​). Such a component generally produces multiple binaries.
* [GeneralComponentSpec](http://docs.google.com/javadoc/org/gradle/platform/base/GeneralComponentSpec.html) is a convenient base interface for components that are built from sources and variant-aware. This is the typical case for a lot of software components, and therefore it should be in most of the cases the base type to be extended.

The core software model includes more types that can be used as base for extension. For example: [LibrarySpec](http://docs.google.com/javadoc/org/gradle/platform/base/LibrarySpec.html) and [ApplicationSpec](http://docs.google.com/javadoc/org/gradle/platform/base/ApplicationSpec.html) can also be extended in this manner. Theses are no-op extensions of GeneralComponentSpec used to describe a software model better by distinguishing libraries and applications components. [TestSuiteSpec](http://docs.google.com/javadoc/org/gradle/testing/base/TestSuiteSpec.html) should be used for all components that describe a test suite.

[Example: Declare a custom component](#3rl6b6f)

**DocumentationComponent.groovy**

@Managed  
**interface** DocumentationComponent **extends** GeneralComponentSpec {}

Types extending ComponentSpec are registered via a rule annotated with [ComponentType](http://docs.google.com/javadoc/org/gradle/platform/base/ComponentType.html):

[Example: Register a custom component](#26qgle8)

**DocumentationPlugin.groovy**

**class** **DocumentationPlugin** **extends** RuleSource {  
 @ComponentType  
 **void** registerComponent(TypeBuilder<DocumentationComponent> builder) {}  
}

[Binaries](#lvqvm1)

To declare a custom binary type one must extend [BinarySpec](http://docs.google.com/javadoc/org/gradle/platform/base/BinarySpec.html).

[Example: Declare a custom binary](#35vee9u)

**DocumentationBinary.groovy**

@Managed  
**interface** DocumentationBinary **extends** BinarySpec {  
 File getOutputDir()  
 **void** setOutputDir(File outputDir)  
}

Types extending BinarySpec are registered via a rule annotated with [ComponentType](http://docs.google.com/javadoc/org/gradle/platform/base/ComponentType.html):

[Example: Register a custom binary](#1l0oohn)

**DocumentationPlugin.groovy**

**class** **DocumentationPlugin** **extends** RuleSource {  
 @ComponentType  
 **void** registerBinary(TypeBuilder<DocumentationBinary> builder) {}  
}

[Source sets](#450c75g)

To declare a custom source set type one must extend [LanguageSourceSet](http://docs.google.com/javadoc/org/gradle/language/base/LanguageSourceSet.html).

[Example: Declare a custom source set](#2k5mhd9)

**MarkdownSourceSet.groovy**

@Managed  
**interface** MarkdownSourceSet **extends** LanguageSourceSet {  
 **boolean** isGenerateIndex()  
 **void** setGenerateIndex(**boolean** generateIndex)  
  
 **boolean** isSmartQuotes()  
 **void** setSmartQuotes(**boolean** smartQuotes)  
}

Types extending LanguageSourceSet are registered via a rule annotated with [ComponentType](http://docs.google.com/javadoc/org/gradle/platform/base/ComponentType.html):

[Example: Register a custom source set](#zawrl2)

**MarkdownPlugin.groovy**

**class** **MarkdownPlugin** **extends** RuleSource {  
 @ComponentType  
 **void** registerMarkdownLanguage(TypeBuilder<MarkdownSourceSet> builder) {}  
}

Setting the *language name* is mandatory.

[Putting it all together](#3jaka8v)

[Generating binaries from components](#1yfukgo)

Binaries generation from components is done via rules annotated with [ComponentBinaries](http://docs.google.com/javadoc/org/gradle/platform/base/ComponentBinaries.html). This rule generates a DocumentationBinary named exploded for each DocumentationComponent and sets its outputDir property:

[Example: Generates documentation binaries](#4ifi34h)

**DocumentationPlugin.groovy**

**class** **DocumentationPlugin** **extends** RuleSource {  
 @ComponentBinaries  
 **void** generateDocBinaries(ModelMap<DocumentationBinary> binaries, VariantComponentSpec component, @Path("buildDir") File buildDir) {  
 binaries.create("exploded") { binary ->  
 outputDir = **new** File(buildDir, "${component.name}/${binary.name}")  
 }  
 }  
}

[Generating tasks from binaries](#2xksdca)

Tasks generation from binaries is done via rules annotated with [BinaryTasks](http://docs.google.com/javadoc/org/gradle/platform/base/BinaryTasks.html). This rule generates a Copy task for each TextSourceSet of each DocumentationBinary:

[Example: Generates tasks for text source sets](#1cq2nk3)

**DocumentationPlugin.groovy**

**class** **DocumentationPlugin** **extends** RuleSource {  
 @BinaryTasks  
 **void** generateTextTasks(ModelMap<Task> tasks, **final** DocumentationBinary binary) {  
 binary.inputs.withType(TextSourceSet) { textSourceSet ->  
 **def** taskName = binary.tasks.taskName("compile", textSourceSet.name)  
 **def** outputDir = **new** File(binary.outputDir, textSourceSet.name)  
 tasks.create(taskName, Copy) {  
 from textSourceSet.source  
 destinationDir = outputDir  
 }  
 }  
 }  
}

This rule generates a MarkdownCompileTask task for each MarkdownSourceSet of each DocumentationBinary:

[Example: Register a custom source set](#3wpq67w)

**MarkdownPlugin.groovy**

**class** **MarkdownPlugin** **extends** RuleSource {  
 @BinaryTasks  
 **void** processMarkdownDocumentation(ModelMap<Task> tasks, **final** DocumentationBinary binary) {  
 binary.inputs.withType(MarkdownSourceSet) { markdownSourceSet ->  
 **def** taskName = binary.tasks.taskName("compile", markdownSourceSet.name)  
 **def** outputDir = **new** File(binary.outputDir, markdownSourceSet.name)  
 tasks.create(taskName, MarkdownHtmlCompile) { compileTask ->  
 compileTask.source = markdownSourceSet.source  
 compileTask.destinationDir = outputDir  
 compileTask.smartQuotes = markdownSourceSet.smartQuotes  
 compileTask.generateIndex = markdownSourceSet.generateIndex  
 }  
 }  
 }  
}

See the sample source for more on the MarkdownCompileTask task.

[Using your custom model](#2bv0gfp)

This build script demonstrate usage of the custom model defined in the sections above:

[Example: an example of using a custom software model](#r0aqni)

**build.gradle**

**import** sample.documentation.DocumentationComponent  
**import** sample.documentation.TextSourceSet  
**import** sample.markdown.MarkdownSourceSet  
  
apply plugin:sample.documentation.DocumentationPlugin  
apply plugin:sample.markdown.MarkdownPlugin  
  
model {  
 components {  
 docs(DocumentationComponent) {  
 sources {  
 reference(TextSourceSet)  
 userguide(MarkdownSourceSet) {  
 generateIndex = true  
 smartQuotes = true  
 }  
 }  
 }  
 }  
}

| **Note** | The code for this example can be found at samples/customModel/languageType/ in the ‘-all’ distribution of Gradle. |
| --- | --- |

And in the components reports for such a build script we can see our model types properly registered:

[Example: components report](#3azy9bb)

**Output of** gradle -q components

> gradle -q components  
  
------------------------------------------------------------  
Root project  
------------------------------------------------------------  
  
DocumentationComponent 'docs'  
-----------------------------  
  
Source sets  
 Markdown source 'docs:userguide'  
 srcDir: src/docs/userguide  
 Text source 'docs:reference'  
 srcDir: src/docs/reference  
  
Binaries  
 DocumentationBinary 'docs:exploded'  
 build using task: :docsExploded  
  
Note: currently not all plugins register their components, so some components may not be visible here.

[About internal views](#4day830)

Internal views can be added to an already registered type or to a new custom type. In other words, using internal views, you can attach extra properties to already registered components, binaries and source sets types like JvmLibrarySpec, JarBinarySpec or JavaSourceSet and to the custom types you write.

Let’s start with a simple component public type and its internal view declarations:

[Example: public type and internal view declaration](#1q58jj4)

**build.gradle**

@Managed **interface** MyComponent **extends** ComponentSpec {  
 String getPublicData()  
 **void** setPublicData(String data)  
}  
@Managed **interface** MyComponentInternal **extends** MyComponent {  
 String getInternalData()  
 **void** setInternalData(String internal)  
}

The type registration is as follows:

[Example: type registration](#4a4w26x)

**build.gradle**

**class** **MyPlugin** **extends** RuleSource {  
 @ComponentType  
 **void** registerMyComponent(TypeBuilder<MyComponent> builder) {  
 builder.internalView(MyComponentInternal)  
 }  
}

The internalView(type) method of the type builder can be called several times. This is how you would add several internal views to a type.

Now, let’s mutate both public and internal data using some rule:

[Example: public and internal data mutation](#2pa6ceq)

**build.gradle**

**class** **MyPlugin** **extends** RuleSource {  
 @Mutate  
 **void** mutateMyComponents(ModelMap<MyComponentInternal> components) {  
 components.all { component ->  
 component.publicData = "Some PUBLIC data"  
 component.internalData = "Some INTERNAL data"  
 }  
 }  
}

Our internalData property should not be exposed to build logic. Let’s check this using the model task on the following build file:

[Example: Build script and model report output](#14fgmmj)

**build.gradle**

apply plugin: MyPlugin  
model {  
 components {  
 my(MyComponent)  
 }  
}

**Output of gradle -q model**

> gradle -q model  
  
------------------------------------------------------------  
Root project  
------------------------------------------------------------  
  
+ components  
 | Type: org.gradle.platform.base.ComponentSpecContainer  
 | Creator: ComponentBasePlugin.PluginRules#components(ComponentSpecContainer)  
 | Rules:  
 ⤷ components { ... } @ build.gradle line 53, column 5  
 ⤷ MyPlugin#mutateMyComponents(ModelMap<MyComponentInternal>)  
 + my  
 | Type: MyComponent  
 | Creator: components { ... } @ build.gradle line 53, column 5 > create(my)  
 | Rules:  
 ⤷ MyPlugin#mutateMyComponents(ModelMap<MyComponentInternal>) > all()  
 + publicData  
 | Type: java.lang.String  
 | Value: Some PUBLIC data  
 | Creator: components { ... } @ build.gradle line 53, column 5 > create(my)  
+ tasks  
 | Type: org.gradle.model.ModelMap<org.gradle.api.Task>  
 | Creator: Project.<init>.tasks()  
 + assemble  
 | Type: org.gradle.api.DefaultTask  
 | Value: task ':assemble'  
 | Creator: Project.<init>.tasks.assemble()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + build  
 | Type: org.gradle.api.DefaultTask  
 | Value: task ':build'  
 | Creator: Project.<init>.tasks.build()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + buildEnvironment  
 | Type: org.gradle.api.tasks.diagnostics.BuildEnvironmentReportTask  
 | Value: task ':buildEnvironment'  
 | Creator: Project.<init>.tasks.buildEnvironment()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + check  
 | Type: org.gradle.api.DefaultTask  
 | Value: task ':check'  
 | Creator: Project.<init>.tasks.check()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + clean  
 | Type: org.gradle.api.tasks.Delete  
 | Value: task ':clean'  
 | Creator: Project.<init>.tasks.clean()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + components  
 | Type: org.gradle.api.reporting.components.ComponentReport  
 | Value: task ':components'  
 | Creator: Project.<init>.tasks.components()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + dependencies  
 | Type: org.gradle.api.tasks.diagnostics.DependencyReportTask  
 | Value: task ':dependencies'  
 | Creator: Project.<init>.tasks.dependencies()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + dependencyInsight  
 | Type: org.gradle.api.tasks.diagnostics.DependencyInsightReportTask  
 | Value: task ':dependencyInsight'  
 | Creator: Project.<init>.tasks.dependencyInsight()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + dependentComponents  
 | Type: org.gradle.api.reporting.dependents.DependentComponentsReport  
 | Value: task ':dependentComponents'  
 | Creator: Project.<init>.tasks.dependentComponents()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + help  
 | Type: org.gradle.configuration.Help  
 | Value: task ':help'  
 | Creator: Project.<init>.tasks.help()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + init  
 | Type: org.gradle.buildinit.tasks.InitBuild  
 | Value: task ':init'  
 | Creator: Project.<init>.tasks.init()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + model  
 | Type: org.gradle.api.reporting.model.ModelReport  
 | Value: task ':model'  
 | Creator: Project.<init>.tasks.model()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + projects  
 | Type: org.gradle.api.tasks.diagnostics.ProjectReportTask  
 | Value: task ':projects'  
 | Creator: Project.<init>.tasks.projects()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + properties  
 | Type: org.gradle.api.tasks.diagnostics.PropertyReportTask  
 | Value: task ':properties'  
 | Creator: Project.<init>.tasks.properties()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + tasks  
 | Type: org.gradle.api.tasks.diagnostics.TaskReportTask  
 | Value: task ':tasks'  
 | Creator: Project.<init>.tasks.tasks()  
 | Rules:  
 ⤷ copyToTaskContainer  
 + wrapper  
 | Type: org.gradle.api.tasks.wrapper.Wrapper  
 | Value: task ':wrapper'  
 | Creator: Project.<init>.tasks.wrapper()  
 | Rules:  
 ⤷ copyToTaskContainer

We can see in this report that publicData is present and that internalData is not.

[Groovy Projects](#34g0dwd)

[Groovy Quickstart](#1jlao46)

To build a Groovy project, you use the *Groovy plugin*. This plugin extends the Java plugin to add Groovy compilation capabilities to your project. Your project can contain Groovy source code, Java source code, or a mix of the two. In every other respect, a Groovy project is identical to a Java project, which we have already seen in [the Java projects tutorial](#3hv69ve).

[A basic Groovy project](#3of45ac)

Let’s look at an example. To use the Groovy plugin, add the following to your build file:

[Example: Groovy plugin](#23kefi5)

**build.gradle**

apply plugin: 'groovy'

| **Note** | The code for this example can be found at samples/groovy/quickstart in the ‘-all’ distribution of Gradle. |
| --- | --- |

This will also apply the Java plugin to the project, if it has not already been applied. The Groovy plugin extends the compile task to look for source files in directory src/main/groovy, and the compileTest task to look for test source files in directory src/test/groovy. The compile tasks use joint compilation for these directories, which means they can contain a mixture of Java and Groovy source files.

To use the Groovy compilation tasks, you must also declare the Groovy version to use and where to find the Groovy libraries. You do this by adding a dependency to the groovy configuration. The compile configuration inherits this dependency, so the Groovy libraries will be included in classpath when compiling Groovy and Java source. For our sample, we will use Groovy 2.2.0 from the public Maven repository:

[Example: Dependency on Groovy](#ipoppy)

**build.gradle**

repositories {  
 mavenCentral()  
}  
  
dependencies {  
 compile 'org.codehaus.groovy:groovy-all:2.4.15'  
}

Here is our complete build file:

[Example: Groovy example - complete build file](#32pc8dr)

**build.gradle**

apply plugin: 'groovy'  
repositories {  
 mavenCentral()  
}  
  
dependencies {  
 compile 'org.codehaus.groovy:groovy-all:2.4.15'  
}

Running gradle build will compile, test and JAR your project.

[Summary](#1humilk)

This chapter describes a very simple Groovy project. Usually, a real project will require more than this. Because a Groovy project *is* a Java project, whatever you can do with a Java project, you can also do with a Groovy project.

You can find out more about the [Groovy plugin](#43ky6rz), and you can find more sample Groovy projects in the samples/groovy directory in the Gradle distribution.

[The Groovy Plugin](#43ky6rz)

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.

[Usage](#41ua19d)

To use the Groovy plugin, include the following in your build script:

[Example: Using the Groovy plugin](#2gzkbh6)

**build.gradle**

apply plugin: 'groovy'

[Tasks](#w4uloz)

The Groovy plugin adds the following tasks to the project.

compileGroovy — [GroovyCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.GroovyCompile.html)

*Depends on*: compileJava

Compiles production Groovy source files.

compileTestGroovy — [GroovyCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.GroovyCompile.html)

*Depends on*: compileTestJava

Compiles test Groovy source files.

compile*SourceSet*Groovy — [GroovyCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.GroovyCompile.html)

*Depends on*: compile*SourceSet*Java

Compiles the given source set’s Groovy source files.

groovydoc — [Groovydoc](http://docs.google.com/dsl/org.gradle.api.tasks.javadoc.Groovydoc.html)

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** |
| --- | --- |
| classes | compileGroovy |
| testClasses | compileTestGroovy |
| *sourceSet*Classes | compile*SourceSet*Groovy |



*Figure 25. Groovy plugin - tasks*

[Project layout](#3g4i4cs)

The Groovy plugin assumes the project layout shown in [Groovy Layout](#1v9sekl). All the Groovy source directories can contain Groovy *and* Java code. The Java source directories may only contain Java source code.[[9](#4f9fx8e)] 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](#2ueq7g7)

Just like the Java plugin, the Groovy plugin allows you to configure custom locations for Groovy production and test source files.

[Example: Custom Groovy source layout](#19k0ho0)

**build.gradle**

sourceSets {  
 main {  
 groovy {  
 srcDirs = ['src/groovy']  
 }  
 }  
  
 test {  
 groovy {  
 srcDirs = ['test/groovy']  
 }  
 }  
}

[Dependency management](#3tjo0bt)

Because Gradle’s build language is based on Groovy, and parts of Gradle are implemented in 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 compile configuration:

[Example: Configuration of Groovy dependency](#28oyajm)

**build.gradle**

repositories {  
 mavenCentral()  
}  
  
dependencies {  
 compile 'org.codehaus.groovy:groovy-all:2.4.15'  
}

If Groovy is only used for test code, the Groovy dependency should be added to the testCompile configuration:

[Example: Configuration of Groovy test dependency](#nu8krf)

**build.gradle**

dependencies {  
 testCompile '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 then declaring a regular Groovy dependency.

[Example: Configuration of bundled Groovy dependency](#37tw3f8)

**build.gradle**

dependencies {  
 compile localGroovy()  
}

The Groovy library doesn’t necessarily have to come from a remote repository. It could also come from a local lib directory, perhaps checked in to source control:

[Example: Configuration of Groovy file dependency](#1mz6dn1)

**build.gradle**

repositories {  
 flatDir { dirs 'lib' }  
}  
  
dependencies {  
 compile module('org.codehaus.groovy:groovy:2.4.15') {  
 dependency('org.ow2.asm:asm-all:5.0.3')  
 dependency('antlr:antlr:2.7.7')  
 dependency('commons-cli:commons-cli:1.2')  
 module('org.apache.ant:ant:1.9.4') {  
 dependencies('org.apache.ant:ant-junit:1.9.4@jar',  
 'org.apache.ant:ant-launcher:1.9.4')  
 }  
 }  
}

[Automatic configuration of groovyClasspath](#46ytwau)

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](#2m446in)

The Groovy plugin does not add any convention properties to the project.

[Source set properties](#119egqg)

The Groovy plugin adds the following convention properties 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](#3l91ze9)

groovy — [SourceDirectorySet](http://docs.google.com/dsl/org.gradle.api.file.SourceDirectorySet.html) (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](#115hox2).

allGroovy — [FileTree](http://docs.google.com/javadoc/org/gradle/api/file/FileTree.html) (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](http://docs.google.com/dsl/org.gradle.api.tasks.GroovySourceSet.html).

The Groovy plugin also modifies some source set properties:

[Groovy Plugin - modified source set properties](#20ec9m2)

| **Property name** | **Change** |
| --- | --- |
| allJava | Adds all .java files found in the Groovy source directories. |
| allSource | Adds all source files found in the Groovy source directories. |

[GroovyCompile](#4kdzs9v)

The Groovy plugin adds a [GroovyCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.GroovyCompile.html) task for each source set in the project. The task type extends the JavaCompile task (see [the relevant Java Plugin section](#2zja2ho)). The GroovyCompile task supports most configuration options of the official Groovy compiler.

Table 15. Groovy plugin - GroovyCompile properties

| **Task Property** | **Type** | **Default Value** |
| --- | --- | --- |
| classpath | [FileCollection](http://docs.google.com/javadoc/org/gradle/api/file/FileCollection.html) | *sourceSet*.compileClasspath |
| source | [FileTree](http://docs.google.com/javadoc/org/gradle/api/file/FileTree.html). Can set using anything described in [Specifying Multiple Files](#115hox2). | *sourceSet*.groovy |
| destinationDir | File. | *sourceSet*.groovy.outputDir |
| groovyClasspath | [FileCollection](http://docs.google.com/javadoc/org/gradle/api/file/FileCollection.html) | groovy configuration if non-empty; Groovy library found on classpath otherwise |

[Compiling and testing for Java 6 or Java 7](#1eokcph)

The Groovy compiler will always be executed with the same version of Java that was used to start Gradle. You should set sourceCompatibility and targetCompatibility to 1.6 or 1.7. If you also have Java source files, you can follow the same steps as for the [Java plugin](#3yo7vda) to ensure the correct Java compiler is used.

[Example: Configure Java 6 build for Groovy](#2dti5l3)

**gradle.properties**

# in $HOME/.gradle/gradle.properties  
java6Home=/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home

**build.gradle**

sourceCompatibility = 1.6  
targetCompatibility = 1.6  
  
**assert** hasProperty('java6Home') : "Set the property 'java6Home' in your your gradle.properties pointing to a Java 6 installation"  
**def** javaExecutablesPath = **new** File(java6Home, 'bin')  
**def** javaExecutables = [:].withDefault { execName ->  
 **def** executable = **new** File(javaExecutablesPath, execName)  
 **assert** executable.exists() : "There is no ${execName} executable in ${javaExecutablesPath}"  
 executable  
}  
tasks.withType(AbstractCompile) {  
 options.with {  
 fork = true  
 forkOptions.javaHome = file(java6Home)  
 }  
}  
tasks.withType(Javadoc) {  
 executable = javaExecutables.javadoc  
}  
tasks.withType(Test) {  
 executable = javaExecutables.java  
}  
tasks.withType(JavaExec) {  
 executable = javaExecutables.java  
}

[The CodeNarc Plugin](#2iq8gzs)

The CodeNarc plugin performs quality checks on your project’s Groovy source files using [CodeNarc](http://codenarc.sourceforge.net/index.html) and generates reports from these checks.

[Usage](#sysfsw)

To use the CodeNarc plugin, include the following in your build script:

[Example: Using the CodeNarc plugin](#3cyfygp)

**build.gradle**

apply plugin: 'codenarc'

The plugin adds a number of tasks to the project that perform the quality checks when used with the [Groovy Plugin](#43ky6rz). You can execute the checks by running gradle check.

[Tasks](#1s3q8oi)

The CodeNarc plugin adds the following tasks to the project:

codenarcMain — [CodeNarc](http://docs.google.com/dsl/org.gradle.api.plugins.quality.CodeNarc.html)

Runs CodeNarc against the production Java source files.

codenarcTest — [CodeNarc](http://docs.google.com/dsl/org.gradle.api.plugins.quality.CodeNarc.html)

Runs CodeNarc against the test Java source files.

codenarc*SourceSet* — [CodeNarc](http://docs.google.com/dsl/org.gradle.api.plugins.quality.CodeNarc.html)

Runs CodeNarc against the given source set’s Java source files.

[Dependencies added to other tasks](#4c3drcb)

The CodeNarc plugin adds the following dependencies to tasks defined by the Groovy plugin.

check

*Depends on*: All CodeNarc tasks, including codenarcMain and codenarcTest.

[Project layout](#2r8o1k4)

The CodeNarc plugin expects the following project layout:

<root>  
└── config  
 └── codenarc (1)  
 └── codenarc.xml (2)

1. CodeNarc configuration files go here
2. Primary CodeNarc configuration file

[Dependency management](#16dybrx)

The CodeNarc plugin adds the following dependency configurations:

Table 16. CodeNarc plugin - dependency configurations

| **Name** | **Meaning** |
| --- | --- |
| codenarc | The CodeNarc libraries to use |

[Configuration](#3qdlufq)

See the [CodeNarcExtension](http://docs.google.com/dsl/org.gradle.api.plugins.quality.CodeNarcExtension.html) class in the API documentation.

[Java Projects](#xvir7l)

[Java Quickstart](#3hv69ve)

[The Java plugin](#25iw4nj)

As we have seen, Gradle is a general-purpose build tool. It can build pretty much anything you care to implement in your build script. Out-of-the-box, however, it doesn’t build anything unless you add code to your build script to do so.

Most Java projects are pretty similar as far as the basics go: you need to compile your Java source files, run some unit tests, and create a JAR file containing your classes. It would be nice if you didn’t have to code all this up for every project. Luckily, you don’t have to. Gradle solves this problem through the use of *plugins*. A plugin is an extension to Gradle which configures your project in some way, typically by adding some pre-configured tasks which together do something useful. Gradle ships with a number of plugins, and you can easily write your own and share them with others. One such plugin is the *Java plugin*. This plugin adds some tasks to your project which will compile and unit test your Java source code, and bundle it into a JAR file.

The Java plugin is convention based. This means that the plugin defines default values for many aspects of the project, such as where the Java source files are located. If you follow the convention in your project, you generally don’t need to do much in your build script to get a useful build. Gradle allows you to customize your project if you don’t want to or cannot follow the convention in some way. In fact, because support for Java projects is implemented as a plugin, you don’t have to use the plugin at all to build a Java project, if you don’t want to.

We have in-depth coverage with many examples about the Java plugin, dependency management and multi-project builds in later chapters. In this chapter we want to give you an initial idea of how to use the Java plugin to build a Java project.

[A basic Java project](#ko6evc)

Let’s look at a simple example. To use the Java plugin, add the following to your build file:

[Example: Using the Java plugin](#34ntxj5)

**build.gradle**

apply plugin: 'java'

| **Note** | The code for this example can be found at samples/java/quickstart in the ‘-all’ distribution of Gradle. |
| --- | --- |

This is all you need to define a Java project. This will apply the Java plugin to your project, which adds a number of tasks to your project.

| **Tip** | What tasks are available?  You can use gradle tasks to list the tasks of a project. This will let you see the tasks that the Java plugin has added to your project. |
| --- | --- |

Gradle expects to find your production source code under src/main/java and your test source code under src/test/java. In addition, any files under src/main/resources will be included in the JAR file as resources, and any files under src/test/resources will be included in the classpath used to run the tests. All output files are created under the build directory, with the JAR file ending up in the build/libs directory.

[Building the project](#1jt47qy)

The Java plugin adds quite a few tasks to your project. However, there are only a handful of tasks that you will need to use to build the project. The most commonly used task is the build task, which does a full build of the project. When you run gradle build, Gradle will compile and test your code, and create a JAR file containing your main classes and resources:

[Example: Building a Java project](#43srqer)

**Output of** gradle build

> gradle build  
> Task :compileJava  
> Task :processResources  
> Task :classes  
> Task :jar  
> Task :assemble  
> Task :compileTestJava  
> Task :processTestResources  
> Task :testClasses  
> Task :test  
> Task :check  
> Task :build  
  
BUILD SUCCESSFUL in 0s  
6 actionable tasks: 6 executed

Some other useful tasks are:

clean

Deletes the build directory, removing all built files.

assemble

Compiles and jars your code, but does not run the unit tests. Other plugins add more artifacts to this task. For example, if you use the War plugin, this task will also build the WAR file for your project.

check

Compiles and tests your code. Other plugins add more checks to this task. For example, if you use the checkstyle plugin, this task will also run Checkstyle against your source code.

[External dependencies](#2iy20mk)

Usually, a Java project will have some dependencies on external JAR files. To reference these JAR files in the project, you need to tell Gradle where to find them. In Gradle, artifacts such as JAR files, are located in a *repository*. A repository can be used for fetching the dependencies of a project, or for publishing the artifacts of a project, or both. For this example, we will use the public Maven repository:

[Example: Adding Maven repository](#y3caud)

**build.gradle**

repositories {  
 mavenCentral()  
}

Let’s add some dependencies. Here, we will declare that our production classes have a compile-time dependency on commons collections, and that our test classes have a compile-time dependency on junit:

[Example: Adding dependencies](#3i2zti6)

**build.gradle**

dependencies {  
 compile group: 'commons-collections', name: 'commons-collections', version: '3.2.2'  
 testCompile group: 'junit', name: 'junit', version: '4.+'  
}

You can find out more in [Dependency Management for Java Projects](#2afmg28).

[Customizing the project](#1x8a3pz)

The Java plugin adds a number of properties to your project. These properties have default values which are usually sufficient to get started. It’s easy to change these values if they don’t suit. Let’s look at this for our sample. Here we will specify the version number for our Java project, along with some attributes to the JAR manifest.

[Example: Customization of MANIFEST.MF](#4h7xmds)

**build.gradle**

version = '1.0'  
jar {  
 manifest {  
 attributes 'Implementation-Title': 'Gradle Quickstart',  
 'Implementation-Version': version  
 }  
}

| **Tip** | What properties are available?  You can use gradle properties to list the properties of a project. This will allow you to see the properties added by the Java plugin, and their default values. |
| --- | --- |

The tasks which the Java plugin adds are regular tasks, exactly the same as if they were declared in the build file. This means you can use any of the mechanisms shown in earlier chapters to customize these tasks. For example, you can set the properties of a task, add behaviour to a task, change the dependencies of a task, or replace a task entirely. In our sample, we will configure the test task, which is of type [Test](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html), to add a system property when the tests are executed:

[Example: Adding a test system property](#2wd7wll)

**build.gradle**

test {  
 systemProperties 'property': 'value'  
}

[Publishing the JAR file](#1bii6te)

Usually the JAR file needs to be published somewhere. To do this, you need to tell Gradle where to publish the JAR file. In Gradle, artifacts such as JAR files are published to repositories. In our sample, we will publish to a local directory. You can also publish to a remote location, or multiple locations.

[Example: Publishing the JAR file](#3vi5ph7)

**build.gradle**

uploadArchives {  
 repositories {  
 flatDir {  
 dirs 'repos'  
 }  
 }  
}

To publish the JAR file, run gradle uploadArchives.

[Creating an Eclipse project](#2anfzp0)

To create the Eclipse-specific descriptor files, like .project, you need to add another plugin to your build file:

[Example: Eclipse plugin](#psq9wt)

**build.gradle**

apply plugin: 'eclipse'

Now execute gradle eclipse command to generate Eclipse project files. More information about the eclipse task can be found in the [Eclipse Plugin](#4du1wux) chapter.

[Summary](#39sdskm)

Here’s the complete build file for our sample:

[Example: Java example - complete build file](#1oxo2sf)

**build.gradle**

apply plugin: 'java'  
apply plugin: 'eclipse'  
version = '1.0'  
jar {  
 manifest {  
 attributes 'Implementation-Title': 'Gradle Quickstart',  
 'Implementation-Version': version  
 }  
}  
repositories {  
 mavenCentral()  
}  
dependencies {  
 compile group: 'commons-collections', name: 'commons-collections', version: '3.2.2'  
 testCompile group: 'junit', name: 'junit', version: '4.+'  
}  
test {  
 systemProperties 'property': 'value'  
}  
uploadArchives {  
 repositories {  
 flatDir {  
 dirs 'repos'  
 }  
 }  
}

[Multi-project Java build](#48xblg8)

Now let’s look at a typical multi-project build. Below is the layout for the project:

**Multi-project build - hierarchical layout**

.  
├── api  
├── services  
│   ├── shared  
│   └── webservice  
└── shared

Here we have four projects. Project api produces a JAR file which is shipped to the client to provide them a Java client for your XML webservice. Project webservice is a webapp which returns XML. Project shared contains code used both by api and webservice. Project services/shared has code that depends on the shared project.

[Defining a multi-project build](#2o2lvo1)

To define a multi-project build, you need to create a *settings file*. The settings file lives in the root directory of the source tree, and specifies which projects to include in the build. It must be called settings.gradle. For this example, we are using a simple hierarchical layout. Here is the corresponding settings file:

[Example: Multi-project build - settings.gradle file](#137w5vu)

**settings.gradle**

include "shared", "api", "services:webservice", "services:shared"

You can find out more about the settings file in [Authoring Multi-Project Builds](#2p2csry).

[Common configuration](#3n7jojn)

For most multi-project builds, there is some configuration which is common to all projects. In our sample, we will define this common configuration in the root project, using a technique called *configuration injection*. Here, the root project is like a container and the subprojects method iterates over the elements of this container - the projects in this instance - and injects the specified configuration. This way we can easily define the manifest content for all archives, and some common dependencies:

[Example: Multi-project build - common configuration](#22ctyrg)

**build.gradle**

subprojects {  
 apply plugin: 'java'  
 apply plugin: 'eclipse-wtp'  
  
 repositories {  
 mavenCentral()  
 }  
  
 dependencies {  
 testCompile 'junit:junit:4.12'  
 }  
  
 version = '1.0'  
  
 jar {  
 manifest.attributes provider: 'gradle'  
 }  
}

Notice that our sample applies the Java plugin to each subproject. This means the tasks and configuration properties we have seen in the previous section are available in each subproject. So, you can compile, test, and JAR all the projects by running gradle build from the root project directory.

Also note that these plugins are only applied within the subprojects section, not at the root level, so the root build will not expect to find Java source files in the root project, only in the subprojects.

[Dependencies between projects](#hi48z9)

You can add dependencies between projects in the same build, so that, for example, the JAR file of one project is used to compile another project. In the api build file we will add a dependency on the shared project. Due to this dependency, Gradle will ensure that project shared always gets built before project api.

[Example: Multi-project build - dependencies between projects](#31hrrn2)

**api/build.gradle**

dependencies {  
 compile project(':shared')  
}

[Creating a distribution](#1gn21uv)

We also add a distribution, that gets shipped to the client:

[Example: Multi-project build - distribution file](#40mpkio)

**api/build.gradle**

task dist(type: Zip) {  
 dependsOn spiJar  
 from 'src/dist'  
 into('libs') {  
 from spiJar.archivePath  
 from configurations.runtime  
 }  
}  
  
artifacts {  
 archives dist  
}

[Where to next?](#2frzuqh)

In this chapter, you have seen how to do some of the things you commonly need to build a Java based project. This chapter is not exhaustive, and there are many other things you can do with Java projects in Gradle. You can find out more about the [Java plugin](#2w5ecyt), and you can find more sample Java projects in the samples/java directory in the Gradle distribution.

Otherwise, continue on to [Dependency Management for Java Projects](#2afmg28).

[Building Java & JVM projects](#1x0gk37)

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](https://guides.gradle.org/building-kotlin-jvm-libraries/), [Groovy](#43ky6rz) and [Scala](#3ep43zb). If you don’t have much experience with building JVM-based projects with Gradle, take a look at the [Java Quickstart](#3hv69ve) first as it will give you a good overview of the basics.

[Introduction](#uxa4ya)

The simplest build script for a Java project applies the [Java Plugin](#2w5ecyt) and optionally sets the project version and Java compatibility versions:

[Example: Applying the Java Plugin](#3ewxnm3)

**build.gradle**

plugins {  
 id 'java'  
}  
  
sourceCompatibility = '1.8'  
targetCompatibility = '1.8'  
version = '1.2.1'

By applying the Java 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. The compatibility options mitigate against problems with the project being built with different Java compiler versions, and 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 Plugin also integrates the above tasks into the standard [Base Plugin lifecycle tasks](#3dlgf20):

* jar is attached to assemble [[10](#1u27xtw)]
* 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](#3lbifu6)

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](#35aue4g))
* where the compiled class files are placed

You can see how these relate to one another in this diagram:



*Figure 26. Source sets and Java compilation*

The shaded boxes represent properties of the source set itself. On top of that, the Java Plugin automatically creates a compilation task for every source set you or a plugin defines — named compile*SourceSet*Java — and several [dependency configurations](#4e1vghp).

| **Note** | 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 Plugin handles this by automatically creating a dedicated task for each defined source set called process*SourceSet*Resources (or processResources for the main source set). The following diagram shows how the source set fits in with this task:



*Figure 27. 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 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](#4h042r0) 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](#2t75qpi) 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](#18cg0xb) in a project.

You’ll learn more about source sets and the features they provide in:

* [Customizing file and directory locations](#3sc3jl4)
* [Configuring Java integration tests](#18cg0xb)

[Managing your dependencies](#27hdtsx)

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](#1v1yuxt).

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](http://hibernate.org/) 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: Declaring dependencies](#mmo40q)

**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](#4f1mdlm).

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](#2c1dop0).

Be aware that the [Java Library Plugin](#1baon6m) creates an additional configuration — api — for dependencies that are required for compiling both the module and any modules that depend on it.

| **Note** | Why no compile configuration?  The Java 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 — although it won’t be going away any time soon — 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](#36mbmoj). |
| --- | --- |

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-](#n9okm1) or [Ivy-compatible](#2ljk6d9) repository
* Using dependencies from a [local filesystem directory](#419q13z)
* Declaring dependencies with [changing](#30x7rho) (e.g. SNAPSHOT) and [dynamic](#3mmzoe9) (range) versions
* Declaring a sibling [project as a dependency](#1n1mu2y)
* [Controlling transitive dependencies and their versions](#1mrcu09)
* Testing your fixes to a 3rd-party dependency via [composite builds](#49x2ik5) (a better alternative to publishing to and consuming from [Maven Local](#3sz406f))

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](#2zja2ho)

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](#43ky6rz), 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](#3sc3jl4)

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: Declaring custom source directories](#1lrlwwc)

**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: Declaring custom source directories additively](#45r9fk5)

**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](http://docs.google.com/dsl/org.gradle.api.tasks.SourceSet.html) and [SourceDirectorySet](http://docs.google.com/dsl/org.gradle.api.file.SourceDirectorySet.html). Note that srcDirs and srcDir() are both on SourceDirectorySet.

[Changing compiler options](#2kwjpry)

Most of the compiler options are accessible through the corresponding task, such as compileJava and compileTestJava. These tasks are of type [JavaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.JavaCompile.html), 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: Setting Java compiler options](#101tzzr)

**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.

Two common options for the Java compiler are defined at the project level:

sourceCompatibility

Defines which language version of Java your source files should be treated as.

targetCompatibility

Defines the minimum JVM version your code should run on, i.e. it determines the version of byte code the compiler generates.

If you need or want more than one compilation task for any reason, you can either [create a new source set](#2t75qpi) or simply define a new task of type [JavaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.JavaCompile.html). We look at setting up a new source set next.

[Compiling and testing Java 6/7](#3yo7vda)

Gradle can only run on Java version 7 or higher. However, support for running Gradle on Java 7 has been deprecated and is scheduled to be removed in Gradle 5.0. There are two reasons for deprecating support for Java 7:

* Java 7 reached [end of life](http://www.oracle.com/technetwork/java/javase/eol-135779.html). Therefore, Oracle ceased public availability of security fixes and upgrades for Java 7 as of April 2015.
* Once support for Java 7 has ceased (likely with Gradle 5.0), Gradle’s implementation can start to use Java 8 APIs optimized for performance and usability.

Gradle still supports compiling, testing, generating Javadoc and executing applications for Java 6 and Java 7. Java 5 is not supported.

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.

The following sample shows how the build.gradle needs to be adjusted. In order to be able to make the build machine-independent, the location of the old Java home and target version should be configured in GRADLE\_USER\_HOME/gradle.properties [[11](#3k1hink)] in the user’s home directory on each developer machine, as shown in the example.

[Example: Configure Java 6 build](#1z6rsvd)

**gradle.properties**

# in $HOME/.gradle/gradle.properties  
javaHome=/Library/Java/JavaVirtualMachines/1.7.0.jdk/Contents/Home  
targetJavaVersion=1.7

**build.gradle**

**assert** hasProperty('javaHome'): "Set the property 'javaHome' in your your gradle.properties pointing to a Java 6 or 7 installation"  
**assert** hasProperty('targetJavaVersion'): "Set the property 'targetJavaVersion' in your your gradle.properties to '1.6' or '1.7'"  
  
sourceCompatibility = targetJavaVersion  
  
**def** javaExecutablesPath = **new** File(javaHome, 'bin')  
**def** javaExecutables = [:].withDefault { execName ->  
 **def** executable = **new** File(javaExecutablesPath, execName)  
 **assert** executable.exists(): "There is no ${execName} executable in ${javaExecutablesPath}"  
 executable  
}  
tasks.withType(AbstractCompile) {  
 options.with {  
 fork = true  
 forkOptions.javaHome = file(javaHome)  
 }  
}  
tasks.withType(Javadoc) {  
 executable = javaExecutables.javadoc  
}  
tasks.withType(Test) {  
 executable = javaExecutables.java  
}  
tasks.withType(JavaExec) {  
 executable = javaExecutables.java  
}

[Compiling independent sources separately](#2t75qpi)

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](#18cg0xb). 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.

[Managing resources](#4j6fbj6)

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 Plugin adds a specific [Copy](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html) task for each source set that handles the processing of its associated resources.

The task’s name follows the convention of process*SourceSet*Resources — 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 Copy task, you can perform any of the processing described in the [Working With Files](#u2g4q7) chapter.

[Java properties files and reproducible builds](#2ybplqz)

You can easily create Java properties files via the [WriteProperties](http://docs.google.com/dsl/org.gradle.api.tasks.WriteProperties.html) task, which fixes a well-known problem with Properties.store() that can reduce the usefulness of [incremental builds](#odc9jc).

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](https://reproducible-builds.org). 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](#1dgzvys)

Alongside providing automatic compilation of unit tests in *src/test/java*, the Java Plugin has native support for running tests that use JUnit 3, 4 & 5 (JUnit 5 support [came in Gradle 4.6](https://docs.gradle.org/4.6/release-notes.html#junit-5-support)) and TestNG. You get:

* An automatic test task of type [Test](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html), 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](#2t75qpi) 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](#4h042r0) 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](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html).

[Packaging and publishing](#3xgneml)

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 Plugin.

The one and only packaging feature provided by the Java Plugin directly is a jar task that packages all the compiled production classes and resources into a single JAR. This JAR is then added as an artifact — as opposed to a dependency — in the archives configuration, hence why it is automatically built by the assemble task.

If you want any other JAR or alternative archive built, you either have to apply an appropriate plugin or create the task manually. For example, if you want a task that generates a 'sources' JAR, define your own Jar task like so:

[Example: Defining a custom task to create a 'sources' JAR](#2clxoue)

**build.gradle**

task sourcesJar(type: Jar) {  
 classifier = 'sources'  
 from sourceSets.main.allJava  
}

See [Jar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Jar.html) for more details on the configuration options available to you. And note that you need to use classifier rather than appendix here for correct publication of the JAR.

If you instead want to create an 'uber' (AKA 'fat') JAR, then you can use a task definition like this:

[Example: Creating a Java uber or fat JAR](#rr7z27)

**build.gradle**

plugins {  
 id 'java'  
}  
  
version = '1.0.0'  
  
repositories {  
 mavenCentral()  
}  
  
dependencies {  
 implementation 'commons-io:commons-io:2.6'  
}  
  
task uberJar(type: Jar) {  
 appendix = 'uber'  
  
 from sourceSets.main.output  
 from configurations.runtimeClasspath.  
 findAll { it.name.endsWith('jar') }.  
 collect { zipTree(it) }  
}

There are several options for publishing a JAR once it has been created:

* the [Maven Publish Plugin](#1egqt2p)
* the [Ivy Publish Plugin](#3ygebqi)
* the uploadArchives task — the [original publishing mechanism](#2dlolyb) — which works with both Ivy and (if you apply the [Maven Plugin](#sqyw64)) Maven

The former two "Publish" plugins are the preferred options.

[Modifying the JAR manifest](#3bqvhq0)

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:

[Example: Customization of MANIFEST.MF](#1qw5rxt)

**build.gradle**

jar {  
 manifest {  
 attributes("Implementation-Title": "Gradle",  
 "Implementation-Version": version)  
 }  
}

See [Manifest](http://docs.google.com/javadoc/org/gradle/api/java/archives/Manifest.html) 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:

[Example: Creating a manifest object.](#4avtalm)

**build.gradle**

ext.sharedManifest = manifest {  
 attributes("Implementation-Title": "Gradle",  
 "Implementation-Version": version)  
}  
task fooJar(type: Jar) {  
 manifest = project.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 for 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:

[Example: Separate MANIFEST.MF for a particular archive](#2q13ktf)

**build.gradle**

task barJar(type: 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](http://docs.google.com/javadoc/org/gradle/api/java/archives/ManifestMergeDetails.html) 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:

[Example: Saving a MANIFEST.MF to disk](#156dv18)

**build.gradle**

jar.manifest.writeTo("$buildDir/mymanifest.mf")

[Generating API documentation](#3p61dp1)

The Java Plugin provides a javadoc task of type [Javadoc](http://docs.google.com/dsl/org.gradle.api.tasks.javadoc.Javadoc.html), 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](http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html#options). See [CoreJavadocOptions](http://docs.google.com/javadoc/org/gradle/external/javadoc/CoreJavadocOptions.html) and [StandardJavadocDocletOptions](http://docs.google.com/javadoc/org/gradle/external/javadoc/StandardJavadocDocletOptions.html) 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:

[Example: Using a custom doclet with Javadoc](#24bbnwu)

**build.gradle**

configurations {  
 asciidoclet  
}  
  
dependencies {  
 asciidoclet 'org.asciidoctor:asciidoclet:1.+'  
}  
  
javadoc {  
 options.docletpath = configurations.asciidoclet.files.toList()  
 options.doclet = 'org.asciidoctor.Asciidoclet'  
}

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:

[Example: Defining a custom Javadoc task](#jgly4n)

**build.gradle**

task testJavadoc(type: Javadoc) {  
 source = sourceSets.test.allJava  
}

These are just two non-trivial but common customizations that you might come across.

[Cleaning the build](#33g9gsg)

The Java Plugin adds a clean task to your project by virtue of applying the [Base Plugin](#vx1227). This task simply deletes everything in the $buildDir directory, hence why you should always put files generated by the build in there. The task is an instance of [Delete](http://docs.google.com/dsl/org.gradle.api.tasks.Delete.html) and you can change what directory it deletes by setting its dir property.

[Building Java libraries](#36mbmoj)

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](#1baon6m), 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*.

| **Note** | The Java Library Plugin automatically applies the standard Java Plugin as well. |
| --- | --- |

If you’re unsure of the difference between an API and implementation dependency, the [Java Library Plugin chapter](#1iljr09) has a detailed explanation. In addition, you can see a basic, practical example of building a Java library in the corresponding [*guide*](https://guides.gradle.org/building-java-libraries/).

[Building Java applications](#42l79o2)

Java applications packaged as a JAR aren’t set up for easy launching from the command line or a desktop environment. The [Application Plugin](#1opuj5n) 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

Note that you will need to explicitly apply the Java Plugin in your build script.

You can see a basic example of building a Java application in the corresponding [*guide*](https://guides.gradle.org/building-java-applications/).

[Building Java web applications](#2hqhjvv)

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](https://projects.spring.io/spring-boot/) with a fat JAR or a [Reactive](https://www.reactivemanifesto.org/)-based system running on [Netty](https://netty.io/). 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](#2fk6b3p), 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](https://plugins.gradle.org/plugin/org.gretty) community plugin, which provides an embedded Servlet container.

[Building Java EE applications](#wvru3o)

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](#1gf8i83). This adds conventions and a task for building EAR files. The plugin’s chapter has more details.

[Testing in Java & JVM projects](#4h042r0)

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](https://junit.org/) and [TestNG](https://testng.org/).

It explains:

* Ways to control how the tests are run ([Test execution](#46ad4c2))
* How to select specific tests to run ([Test filtering](#42ddq1a))
* What test reports are generated and how to influence the process ([Test reporting](#3gvfcrh))
* How Gradle finds tests to run ([Test detection](#1w0pmza))
* How to make use of the major frameworks' mechanisms for grouping tests together ([Test grouping](#4g0d5n3))

But first, we look at the basics of JVM testing in Gradle.

[The basics](#2v5nfuw)

All JVM testing revolves around a single task type: [Test](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html). 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](http://docs.google.com/dsl/org.gradle.api.tasks.testing.TestReport.html) 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()](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html#org.gradle.api.tasks.testing.Test:testClassesDirs))
* The execution classpath, which should include the classes under test as well as the test library that you’re using (property: [Test.getClasspath()](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html#org.gradle.api.tasks.testing.Test:classpath))

When you’re using a JVM language plugin — such as the [Java Plugin](#2w5ecyt) — 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](#34qadz2).

It’s also worth bearing in mind that the test source set automatically creates [corresponding dependency configurations](#4e1vghp) — 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 4.x and changes the maximum heap size for the tests' JVM to 1 gigabyte:

[Example: A basic configuration for the 'test' task](#1aaxq2p)

**build.gradle**

dependencies {  
 testImplementation 'junit:junit:4.12'  
}  
  
test {  
 useJUnit()  
  
 maxHeapSize = '1G'  
}

The [Test](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html) task has many generic configuration options as well as several framework-specific ones that you can find described in [JUnitOptions](http://docs.google.com/javadoc/org/gradle/api/tasks/testing/junit/JUnitOptions.html), [JUnitPlatformOptions](http://docs.google.com/javadoc/org/gradle/api/tasks/testing/junitplatform/JUnitPlatformOptions.html) and [TestNGOptions](http://docs.google.com/javadoc/org/gradle/api/tasks/testing/testng/TestNGOptions.html). 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](#18cg0xb).

[Test execution](#46ad4c2)

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.

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](http://docs.google.com/javadoc/org/gradle/api/tasks/testing/logging/TestLoggingContainer.html) for more detail.

See [Test](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html) for details on all the available configuration options.

| **Note** | 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](https://docs.oracle.com/javase/7/docs/api/java/lang/SecurityManager.html) 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](#42ddq1a)

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: Filtering tests in the build script](#3ual8qi)

**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](http://docs.google.com/javadoc/org/gradle/api/tasks/testing/TestFilter.html) 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](#29fviyb)

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](#ol5t64)

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](#2olpkfy) 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.\*"

[Single test execution via System Properties](#38ktbtx)

| **Note** | This mechanism has been superseded by 'Test Filtering', described above. We only include it in case you encounter it in online forums and blogs. |
| --- | --- |

Test inclusions/exclusions are a file-based — as opposed to a class name-based — mechanism for selecting tests to run. It’s activated when you use the -D*taskName*.single=*<pattern>* option on the command line, e.g. -Dtest.single=MyTest.

[Test reporting](#3gvfcrh)

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.

Unlike with many of the testing configuration options, there are several project-level [convention properties that affect the test reports](#1nq3m1q). For example, you can change the destination of the test results and reports like so:

[Example: Changing the default test report and results directories](#47pr4pj)

**build.gradle**

reporting.baseDir = "my-reports"  
testResultsDirName = "$buildDir/my-test-results"  
  
task showDirs {  
 doLast {  
 logger.quiet(rootDir.toPath().relativize(project.reportsDir.toPath()).toString())  
 logger.quiet(rootDir.toPath().relativize(project.testResultsDir.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](http://docs.google.com/dsl/org.gradle.api.tasks.testing.TestReport.html) 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: Creating a unit test report for subprojects](#2mv1exc)

**build.gradle**

subprojects {  
 apply plugin: 'java'  
  
 *// Disable the test report for the individual test task*  
 test {  
 reports.html.enabled = false  
 }  
}  
  
task testReport(type: TestReport) {  
 destinationDir = file("$buildDir/reports/allTests")  
 *// Include the results from the `test` task in all subprojects*  
 reportOn subprojects\*.test  
}

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.

[Test detection](#1w0pmza)

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](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html) 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](http://junit.org/junit5/docs/current/user-guide), only includes and excludes are used to filter test classes — scanForTestClasses has no effect. |
| --- | --- |

[Test grouping](#4g0d5n3)

JUnit, JUnit Platform and TestNG allow sophisticated groupings of test methods.

JUnit 4.8 introduced the concept of categories for grouping JUnit 4 tests classes and methods.[[12](#120bp55)] [Test.useJUnit(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html#org.gradle.api.tasks.testing.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: JUnit Categories](#3lzz7sy)

**build.gradle**

test {  
 useJUnit {  
 includeCategories 'org.gradle.junit.CategoryA'  
 excludeCategories 'org.gradle.junit.CategoryB'  
 }  
}

[JUnit Platform](http://junit.org/junit5/docs/current/user-guide) introduced [tagging](http://junit.org/junit5/docs/current/user-guide/#writing-tests-tagging-and-filtering) to replace categories. You can specify the included/excluded tags via [Test.useJUnitPlatform(org.gradle.api.Action)](http://docs.google.com/javadoc/org/gradle/api/tasks/testing/Test.html#useJUnitPlatform-org.gradle.api.Action-), as follows:

[Example: JUnit Platform Tags](#2159i0r)

**build.gradle**

test {  
 useJUnitPlatform {  
 includeTags 'fast'  
 excludeTags 'slow'  
 }  
}

The TestNG framework uses the concept of test groups for a similar effect.[[13](#4l4x0ok)] You can configure which test groups to include or exclude during the test execution via the [Test.useTestNG(org.gradle.api.Action)](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html#org.gradle.api.tasks.testing.Test:useTestNG(org.gradle.api.Action)) setting, as seen here:

[Example: Grouping TestNG tests](#30a7awd)

**build.gradle**

test {  
 useTestNG {  
 excludeGroups 'integrationTests'  
 includeGroups 'unitTests'  
 }  
}

[Using JUnit 5](#1ffhl46)

[JUnit 5](http://junit.org/junit5) 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](http://junit.org/junit5/docs/current/user-guide/#writing-tests) and [extension model](http://junit.org/junit5/docs/current/user-guide/#extensions) 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: Enabling JUnit Platform to run your tests](#3zf53rz)

**build.gradle**

test {  
 useJUnitPlatform()  
}

See [Test.useJUnitPlatform()](http://docs.google.com/javadoc/org/gradle/api/tasks/testing/Test.html#useJUnitPlatform) for more details.

| **Note** | There are some known limitations of using JUnit 5 with Gradle, for example that tests in static nested classes won’t be discovered and classes are still displayed by their class name instead of @DisplayName. These will be fixed in future version of Gradle. If you find more, please tell us at <https://github.com/gradle/gradle/issues/new> |
| --- | --- |

[Compiling and executing JUnit Jupiter tests](#2ekfdzs)

To enable JUnit Jupiter support in Gradle, all you need to do is add the following dependencies:

[Example: JUnit Jupiter dependencies](#tppo7l)

**build.gradle**

dependencies {  
 testImplementation 'org.junit.jupiter:junit-jupiter-api:5.1.0'  
 testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.1.0'  
}

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](#3dpd6ve)

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: JUnit Vintage dependencies](#1sunh37)

**build.gradle**

dependencies {  
 testImplementation 'org.junit.jupiter:junit-jupiter-api:5.1.0'  
 testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.1.0'  
 testCompileOnly 'junit:junit:4.12'  
 testRuntimeOnly 'org.junit.vintage:junit-vintage-engine:5.1.0'  
}

In this way, you can use gradle test to test JUnit 3/4 tests on JUnit Platform, without the need to rewrite them.

A sample of mixed tests can be found at samples/testing/junitplatform/mix in the '-all' distribution of Gradle.

[Filtering test engine](#4cuazr0)

JUnit Platform allows you to use different test engines. JUnit currently provides two TestEngine implementations out of the box: [junit-jupiter-engine](https://junit.org/junit5/docs/current/api/org/junit/jupiter/engine/package-summary.html) and [junit-vintage-engine](https://junit.org/junit5/docs/current/api/org/junit/vintage/engine/package-summary.html). You can also write and plug in your own TestEngine implementation as documented [here](https://junit.org/junit5/docs/current/user-guide/#launcher-api-engines-custom).

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: Filter specific engines](#2rzl9yt)

**build.gradle**

test {  
 useJUnitPlatform {  
 includeEngines 'junit-vintage'  
 *// excludeEngines 'junit-jupiter'*  
 }  
}

A test engine filtering sample can be found at samples/testing/junitplatform/engine in the '-all' distribution of Gradle.

[Test execution order in TestNG](#174vk6m)

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()](http://docs.google.com/javadoc/org/gradle/api/tasks/testing/testng/TestNGOptions.html#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()](http://docs.google.com/javadoc/org/gradle/api/tasks/testing/testng/TestNGOptions.html#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](https://jitpack.io/com/github/cbeust/testng/master/javadoc/org/testng/TestNG.html) that is used by Gradle’s TestNG integration executes tests in unpredictable order by default.[[14](#3r4j2uf)] 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: Preserving order of TestNG tests](#269td28)

**build.gradle**

test {  
 useTestNG {  
 preserveOrder true  
 }  
}

The groupByInstance property controls whether tests should be grouped by instance rather than by class. The [TestNG documentation](http://testng.org/doc/documentation-main.html#dependencies-with-annotations) 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: Grouping TestNG tests by instances](#lf3na1)

**build.gradle**

test {  
 useTestNG {  
 groupByInstances = true  
 }  
}

[TestNG parameterized methods and reporting](#35er5xu)

TestNG supports [parameterizing test methods](http://testng.org/doc/documentation-main.html#parameters), 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](#18cg0xb)

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 taking these steps:

1. Create a new [source set](#3lbifu6) 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: Setting up working integration tests](#1kk1g5n)

**build.gradle**

sourceSets {  
 intTest {  
 compileClasspath += sourceSets.main.output  
 runtimeClasspath += sourceSets.main.output  
 }  
}  
  
configurations {  
 intTestImplementation.extendsFrom implementation  
 intTestRuntimeOnly.extendsFrom runtimeOnly  
}  
  
dependencies {  
 intTestImplementation 'junit:junit:4.12'  
}

This will set up a new source set called intTest that automatically creates:

* intTestImplementation, intTestCompileOnly, intTestRuntimeOnly configurations (and [a few others](#4e1vghp) that are less commonly needed)
* A compileIntTestJava task that will compile all the source files under *src/intTest/java*

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](#4f5j5f0) 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: Defining a working integration test task](#44joytg)

**build.gradle**

task integrationTest(type: Test) {  
 description = 'Runs integration tests.'  
 group = 'verification'  
  
 testClassesDirs = sourceSets.intTest.output.classesDirs  
 classpath = sourceSets.intTest.runtimeClasspath  
 shouldRunAfter test  
}  
  
check.dependsOn integrationTest

Again, we’re accessing a source set to get the relevant information, i.e. where the compiled test classes are — the testClassesDir 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.

[Skipping the tests](#2joz919)

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](#1a346fx) or [in the build script](#rd1764). 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(org.gradle.api.specs.Spec)](http://docs.google.com/dsl/org.gradle.api.Task.html#org.gradle.api.Task:onlyIf(org.gradle.api.specs.Spec)) method. The following sample skips the test task if the project has a property called mySkipTests:

[Example: Skipping the unit tests based on a project property](#yu9j92)

**build.gradle**

test.onlyIf { !project.hasProperty('mySkipTests') }

In this case, Gradle will mark the skipped tests as "SKIPPED" rather than exclude them from the build.

[Forcing tests to run](#3itx1wv)

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 force tests to run in this situation 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](#25vml65) provided by the [Base Plugin](#3dlgf20). You can use it for *any* task.

[Debugging when running tests](#1xz7c4o)

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()](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html#org.gradle.api.tasks.testing.Test:debug) property to true or use the --debug-jvm command line option.

When debugging for tests is enabled, Gradle will start the test process suspended and listening on port 5005.

[The Java Plugin](#2w5ecyt)

The Java plugin adds Java compilation along with testing and bundling capabilities to a project. It serves as the basis for many of the other JVM language Gradle plugins. You can find a comprehensive introduction and overview to the Java Plugin in the [Building Java Projects](#1x0gk37) chapter.

[Usage](#4hyuush)

To use the Java plugin, include the following in your build script:

[Example: Using the Java plugin](#2x4550a)

**build.gradle**

apply plugin: 'java'

[Project layout](#1c9ff83)

The Java plugin assumes the project layout shown below. None of these directories need to exist or have anything in them. The Java plugin will compile whatever it finds, and handles anything which is missing.

src/main/java

Production Java source.

src/main/resources

Production resources, such as XML and properties files.

src/test/java

Test Java source.

src/test/resources

Test resources.

src/*sourceSet*/java

Java source for the source set named *sourceSet*.

src/*sourceSet*/resources

Resources for the source set named *sourceSet*.

[Changing the project layout](#3w92xvw)

You configure the project layout by configuring the appropriate source set. This is discussed in more detail in the following sections. Here is a brief example which changes the main Java and resource source directories.

[Example: Custom Java source layout](#2bed83p)

**build.gradle**

sourceSets {  
 main {  
 java {  
 srcDirs = ['src/java']  
 }  
 resources {  
 srcDirs = ['src/resources']  
 }  
 }  
}

[Source sets](#qjnibi)

The plugin adds the following [source sets](#3lbifu6):

main

Contains the production source code of the project, which is compiled and assembled into a JAR.

test

Contains your test source code, which is compiled and executed using JUnit or TestNG. These are typically unit tests, but you can include any test in this source set as long as they all share the same compilation and runtime classpaths.

[Source set properties](#3ajb0zb)

The following table lists some of the important properties of a source set. You can find more details in the API documentation for [SourceSet](http://docs.google.com/dsl/org.gradle.api.tasks.SourceSet.html).

name — (read-only) String

The name of the source set, used to identify it.

output — (read-only) [SourceSetOutput](http://docs.google.com/dsl/org.gradle.api.tasks.SourceSetOutput.html)

The output files of the source set, containing its compiled classes and resources.

output.classesDirs — (read-only) [FileCollection](http://docs.google.com/javadoc/org/gradle/api/file/FileCollection.html)

*Default value*: $buildDir/classes/java/$name, e.g. *build/classes/java/main*

The directories to generate the classes of this source set into. May contain directories for other JVM languages, e.g. *build/classes/kotlin/main*.

output.resourcesDir — File

*Default value*: $buildDir/resources/$name, e.g. *build/resources/main*

The directory to generate the resources of this source set into.

compileClasspath — [FileCollection](http://docs.google.com/javadoc/org/gradle/api/file/FileCollection.html)

*Default value*: ${name}CompileClasspath configuration

The classpath to use when compiling the source files of this source set.

annotationProcessorPath — [FileCollection](http://docs.google.com/javadoc/org/gradle/api/file/FileCollection.html)

*Default value*: ${name}AnnotationProcessor configuration

The processor path to use when compiling the source files of this source set.

runtimeClasspath — [FileCollection](http://docs.google.com/javadoc/org/gradle/api/file/FileCollection.html)

*Default value*: $output, ${name}RuntimeClasspath configuration

The classpath to use when executing the classes of this source set.

java — (read-only) [SourceDirectorySet](http://docs.google.com/dsl/org.gradle.api.file.SourceDirectorySet.html)

The Java source files of this source set. Contains only .java files found in the Java source directories, and excludes all other files.

java.srcDirs — Set<File>

*Default value*: src/$name/java, e.g. *src/main/java*

The source directories containing the Java source files of this source set. You can set this to any value that is described in sec:specifying\_multiple\_files[this section](#115hox2).

java.outputDir — File

*Default value*: $buildDir/classes/java/$name, e.g. *build/classes/java/main*

The directory to generate compiled Java sources into. You can set this to any value that is described in [this section](#w0xtvl).

resources — (read-only) [SourceDirectorySet](http://docs.google.com/dsl/org.gradle.api.file.SourceDirectorySet.html)

The resources of this source set. Contains only resources, and excludes any .java files found in the resource directories. Other plugins, such as the [Groovy Plugin](#43ky6rz), exclude additional types of files from this collection.

resources.srcDirs — Set<File>

*Default value*: [src/$name/resources]

The directories containing the resources of this source set. You can set this to any type of value that is described in [this section](#115hox2).

allJava — (read-only) [SourceDirectorySet](http://docs.google.com/dsl/org.gradle.api.file.SourceDirectorySet.html)

*Default value*: Same as java property

All Java files of this source set. Some plugins, such as the Groovy Plugin, add additional Java source files to this collection.

allSource — (read-only) [SourceDirectorySet](http://docs.google.com/dsl/org.gradle.api.file.SourceDirectorySet.html)

*Default value*: Sum of everything in the resources and java properties

All source files of this source set of any language. This includes all resource files and all Java source files. Some plugins, such as the Groovy Plugin, add additional source files to this collection.

[Defining new source sets](#1polb74)

See the [integration test example](#18cg0xb) in the *Testing in Java & JVM projects* chapter.

[Some other simple source set examples](#49o8tux)

Adding a JAR containing the classes of a source set:

[Example: Assembling a JAR for a source set](#2otj42q)

**build.gradle**

task intTestJar(type: Jar) {  
 from sourceSets.intTest.output  
}

Generating Javadoc for a source set:

[Example: Generating the Javadoc for a source set](#13yteaj)

**build.gradle**

task intTestJavadoc(type: Javadoc) {  
 source sourceSets.intTest.allJava  
}

Adding a test suite to run the tests in a source set:

[Example: Running tests in a source set](#3nygwyc)

**build.gradle**

task intTest(type: Test) {  
 testClassesDirs = sourceSets.intTest.output.classesDirs  
 classpath = sourceSets.intTest.runtimeClasspath  
}

[Tasks](#yg2rcz)

The Java plugin adds a number of tasks to your project, as shown below.

compileJava — [JavaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.JavaCompile.html)

*Depends on*: All tasks which contribute to the compilation classpath, including jar tasks from projects that are on the classpath via project dependencies

Compiles production Java source files using the JDK compiler.

processResources — [Copy](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html)

Copies production resources into the production resources directory.

classes

*Depends on*: compileJava, processResources

This is an aggregate task that just depends on other tasks. Other plugins may attach additional compilation tasks to it.

compileTestJava — [JavaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.JavaCompile.html)

*Depends on*: classes, and all tasks that contribute to the test compilation classpath

Compiles test Java source files using the JDK compiler.

processTestResources — [Copy](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html)

Copies test resources into the test resources directory.

testClasses

*Depends on*: compileTestJava, processTestResources

This is an aggregate task that just depends on other tasks. Other plugins may attach additional test compilation tasks to it.

jar — [Jar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Jar.html)

*Depends on*: classes

Assembles the production JAR file, based on the classes and resources attached to the main source set.

javadoc — [Javadoc](http://docs.google.com/dsl/org.gradle.api.tasks.javadoc.Javadoc.html)

*Depends on*: classes

Generates API documentation for the production Java source using Javadoc.

test — [Test](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html)

*Depends on*: testClasses, and all tasks which produce the test runtime classpath

Runs the unit tests using JUnit or TestNG.

uploadArchives — [Upload](http://docs.google.com/dsl/org.gradle.api.tasks.Upload.html)

*Depends on*: jar, and any other task that produces an artifact attached to the archives configuration

Uploads artifacts in the archives configuration — including the production JAR file — to the configured repositories.

clean — [Delete](http://docs.google.com/dsl/org.gradle.api.tasks.Delete.html)

Deletes the project build directory.

clean*TaskName* — [Delete](http://docs.google.com/dsl/org.gradle.api.tasks.Delete.html)

Deletes files created by the specified task. For example, cleanJar will delete the JAR file created by the jar task and cleanTest will delete the test results created by the test task.

[SourceSet Tasks](#233r765)

For each source set you add to the project, the Java plugin adds the following tasks:

compile*SourceSet*Java — [JavaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.JavaCompile.html)

*Depends on*: All tasks which contribute to the source set’s compilation classpath

Compiles the given source set’s Java source files using the JDK compiler.

process*SourceSet*Resources — [Copy](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html)

Copies the given source set’s resources into the resources directory.

*sourceSet*Classes — [Task](http://docs.google.com/dsl/org.gradle.api.Task.html)

*Depends on*: compile*SourceSet*Java, process*SourceSet*Resources

Prepares the given source set’s classes and resources for packaging and execution. Some plugins may add additional compilation tasks for the source set.

[Lifecycle Tasks](#i91hdy)

The Java plugin attaches some of its tasks to the lifecycle tasks defined by the [Base Plugin](#3dlgf20) — which the Java Plugin applies automatically — and it also adds a few other lifecycle tasks:

assemble

*Depends on*: jar, and all other tasks that create artifacts attached to the archives configuration

Aggregate task that assembles all the archives in the project. This task is added by the Base Plugin.

check

*Depends on*: test

Aggregate task that performs verification tasks, such as running the tests. Some plugins add their own verification tasks to check. You should also attach any custom Test tasks to this lifecycle task if you want them to execute for a full build. This task is added by the Base Plugin.

build

*Depends on*: check, assemble

Aggregate tasks that performs a full build of the project. This task is added by the Base Plugin.

buildNeeded

*Depends on*: build, and buildNeeded tasks in all projects that are dependencies in the testRuntimeClasspath configuration.

Performs a full build of the project and all projects it depends on.

buildDependents

*Depends on*: build, and buildDependents tasks in all projects that have this project as a dependency in theeir testRuntimeClasspath configurations

Performs a full build of the project and all projects which depend upon it.

build*ConfigName* — *task rule*

*Depends on*: all tasks that generate the artifacts attached to the named — *ConfigName* — configuration

Assembles the artifacts for the specified configuration. This rule is added by the Base Plugin.

upload*ConfigName* — *task rule*, type: [Upload](http://docs.google.com/dsl/org.gradle.api.tasks.Upload.html)

*Depends on*: all tasks that generate the artifacts attached to the named — *ConfigName* — configuration

Assembles and uploads the artifacts in the specified configuration. This rule is added by the Base Plugin.

The following diagram shows the relationships between these tasks.



*Figure 28. Java plugin - tasks*

[Dependency management](#2c1dop0)

The Java plugin adds a number of [dependency configurations](#37m2jsg) to your project, as shown below. It assigns those configurations to tasks such as compileJava and test.

[Dependency configurations](#2bkqgd0)

| **Note** | To find information on the api configuration, please consult the [Java Library Plugin](#328p01r) reference documentation and the [Dependency Management Tutorial](#2afmg28). |
| --- | --- |

~~compile~~(Deprecated)

Compile time dependencies. Superseded by implementation.

implementation extends compile

Implementation only dependencies.

compileOnly

Compile time only dependencies, not used at runtime.

compileClasspath extends compile, compileOnly, implementation

Compile classpath, used when compiling source. Used by task compileJava.

annotationProcessor

Annotation processors used during compilation.

~~runtime~~(Deprecated) extends compile

Runtime dependencies. Superseded by runtimeOnly.

runtimeOnly

Runtime only dependencies.

runtimeClasspath extends runtimeOnly, runtime, implementation

Runtime classpath contains elements of the implementation, as well as runtime only elements.

~~testCompile~~(Deprecated) extends compile

Additional dependencies for compiling tests. Superseded by testImplementation.

testImplementation extends testCompile, implementation

Implementation only dependencies for tests.

testCompileOnly

Additional dependencies only for compiling tests, not used at runtime.

testCompileClasspath extends testCompile, testCompileOnly, testImplementation

Test compile classpath, used when compiling test sources. Used by task compileTestJava.

~~testRuntime~~(Deprecated) extends runtime, testCompile

Additional dependencies for running tests only. Used by task test. Superseded by testRuntimeOnly.

testRuntimeOnly extends runtimeOnly

Runtime only dependencies for running tests. Used by task test.

testRuntimeClasspath extends testRuntimeOnly, testRuntime, testImplementation

Runtime classpath for running tests.

archives

Artifacts (e.g. jars) produced by this project. Used by tasks uploadArchives.

default extends runtime

The default configuration used by a project dependency on this project. Contains the artifacts and dependencies required by this project at runtime.



*Figure 29. Java plugin - dependency configurations*

For each source set you add to the project, the Java plugins adds the following dependency configurations:

[SourceSet dependency configurations](#4e1vghp)

*~~sourceSet~~*~~Compile~~(Deprecated)

Compile time dependencies for the given source set. Superseded by *sourceSet*Implementation.

*sourceSet*Implementation extends *sourceSet*Compile

Compile time dependencies for the given source set. Used by *sourceSet*CompileClasspath, *sourceSet*RuntimeClasspath.

*sourceSet*CompileOnly

Compile time only dependencies for the given source set, not used at runtime.

*sourceSet*CompileClasspath extends compile*SourceSet*Java

Compile classpath, used when compiling source. Used by *sourceSet*Compile, *sourceSet*CompileOnly, *sourceSet*Implementation.

*sourceSet*AnnotationProcessor

Annotation processors used during compilation of this source set.

*~~sourceSet~~*~~Runtime~~(Deprecated)

Runtime dependencies for the given source set. Used by *sourceSet*Compile. Superseded by *sourceSet*RuntimeOnly.

*sourceSet*RuntimeOnly

Runtime only dependencies for the given source set.

*sourceSet*RuntimeClasspath extends *sourceSet*RuntimeOnly, *sourceSet*Runtime, *sourceSet*Implementation

Runtime classpath contains elements of the implementation, as well as runtime only elements.

[Publishing](#1hdza9k)

components.java

A [SoftwareComponent](http://docs.google.com/javadoc/org/gradle/api/component/SoftwareComponent.html) for [publishing](#2zbgiuw) the production JAR created by the jar task. This component includes the runtime dependency information for the JAR.

[Convention properties](#1nq3m1q)

The Java Plugin adds a number of convention properties to the project, shown below. You can use these properties in your build script as though they were properties of the project object.

[Directory properties](#41dmsxd)

String reporting.baseDir

The name of the directory to generate reports into, relative to the build directory. Default value: reports

(read-only) File reportsDir

The directory to generate reports into. Default value: *buildDir*/*reporting.baseDir*

String testResultsDirName

The name of the directory to generate test result .xml files into, relative to the build directory. Default value: test-results

(read-only) File testResultsDir

The directory to generate test result .xml files into. Default value: *buildDir*/*testResultsDirName*

String testReportDirName

The name of the directory to generate the test report into, relative to the reports directory. Default value: tests

(read-only) File testReportDir

The directory to generate the test report into. Default value: *reportsDir*/testReportDirName

String libsDirName

The name of the directory to generate libraries into, relative to the build directory. Default value: libs

(read-only) File libsDir

The directory to generate libraries into. Default value: *buildDir*/*libsDirName*

String distsDirName

The name of the directory to generate distributions into, relative to the build directory. Default value: distributions

(read-only) File distsDir

The directory to generate distributions into. Default value: *buildDir*/*distsDirName*

String docsDirName

The name of the directory to generate documentation into, relative to the build directory. Default value: docs

(read-only) File docsDir

The directory to generate documentation into. Default value: *buildDir*/*docsDirName*

String dependencyCacheDirName

The name of the directory to use to cache source dependency information, relative to the build directory. Default value: dependency-cache

[Other convention properties](#2gix356)

(read-only) [SourceSetContainer](http://docs.google.com/javadoc/org/gradle/api/tasks/SourceSetContainer.html) sourceSets

Contains the project’s source sets. Default value: Not null [SourceSetContainer](http://docs.google.com/javadoc/org/gradle/api/tasks/SourceSetContainer.html)

[JavaVersion](http://docs.google.com/javadoc/org/gradle/api/JavaVersion.html) sourceCompatibility

Java version compatibility to use when compiling Java source. Default value: version of the current JVM in use [JavaVersion](http://docs.google.com/javadoc/org/gradle/api/JavaVersion.html). Can also set using a String or a Number, e.g. '1.5' or 1.5.

[JavaVersion](http://docs.google.com/javadoc/org/gradle/api/JavaVersion.html) targetCompatibility

Java version to generate classes for. Default value: *sourceCompatibility*. Can also set using a String or Number, e.g. '1.5' or 1.5.

String archivesBaseName

The basename to use for archives, such as JAR or ZIP files. Default value: *projectName*

[Manifest](http://docs.google.com/javadoc/org/gradle/api/java/archives/Manifest.html) manifest

The manifest to include in all JAR files. Default value: an empty manifest.

These properties are provided by convention objects of type [JavaPluginConvention](http://docs.google.com/javadoc/org/gradle/api/plugins/JavaPluginConvention.html), and [BasePluginConvention](http://docs.google.com/javadoc/org/gradle/api/plugins/BasePluginConvention.html).

[Javadoc](#vo7dcz)

The javadoc task is an instance of [Javadoc](http://docs.google.com/dsl/org.gradle.api.tasks.javadoc.Javadoc.html). It supports the core Javadoc options and the options of the standard doclet described in the [reference documentation](http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html) of the Javadoc executable. For a complete list of supported Javadoc options consult the API documentation of the following classes: [CoreJavadocOptions](http://docs.google.com/javadoc/org/gradle/external/javadoc/CoreJavadocOptions.html) and [StandardJavadocDocletOptions](http://docs.google.com/javadoc/org/gradle/external/javadoc/StandardJavadocDocletOptions.html).

[Javadoc properties](#3fnuw0s)

[FileCollection](http://docs.google.com/javadoc/org/gradle/api/file/FileCollection.html) classpath

Default value: sourceSets.main.output + sourceSets.main.compileClasspath

[FileTree](http://docs.google.com/javadoc/org/gradle/api/file/FileTree.html) source

Default value: sourceSets.main.allJava. Can set using anything described in [Understanding implicit conversion to file collections](#115hox2).

File destinationDir

Default value: *docsDir*/javadoc

String title

Default value: The name and version of the project

[Clean](#1ut568l)

The clean task is an instance of [Delete](http://docs.google.com/dsl/org.gradle.api.tasks.Delete.html). It simply removes the directory denoted by its dir property.

[Clean properties](#4essowe)

File dir

Default value: *buildDir*

[Resources](#2ty2z47)

The Java plugin uses the [Copy](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html) task for resource handling. It adds an instance for each source set in the project. You can find out more about the copy task in [File copying in depth](#u2g4q7).

[ProcessResources properties](#193d9c0)

Object srcDirs

Default value: *sourceSet*.resources. Can set using anything described in [Understanding implicit conversion to file collections](#115hox2).

File destinationDir

Default value: *sourceSet*.output.resourcesDir. Can set using anything described in [file paths in depth](#2ex5uie).

[CompileJava](#3t30rzt)

The Java plugin adds a [JavaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.compile.JavaCompile.html) instance for each source set in the project. Some of the most common configuration options are shown below.

[Compile properties](#288b27m)

[FileCollection](http://docs.google.com/javadoc/org/gradle/api/file/FileCollection.html) classpath

Default value: *sourceSet*.compileClasspath

[FileTree](http://docs.google.com/javadoc/org/gradle/api/file/FileTree.html) source

Default value: *sourceSet*.java. Can set using anything described in [Understanding implicit conversion to file collections](#115hox2).

File destinationDir

Default value: *sourceSet*.java.outputDir

By default, the Java compiler runs in the Gradle process. Setting options.fork to true causes compilation to occur in a separate process. In the case of the Ant javac task, this means that a new process will be forked for each compile task, which can slow down compilation. Conversely, Gradle’s direct compiler integration (see above) will reuse the same compiler process as much as possible. In both cases, all fork options specified with options.forkOptions will be honored.

[Incremental Java compilation](#ndlcff)

Gradle comes with a sophisticated incremental Java compiler that is active by default.

This gives you the following benefits

* Incremental builds are much faster.
* The smallest possible number of class files are changed. Classes that don’t need to be recompiled remain unchanged in the output directory. An example scenario when this is really useful is using JRebel - the fewer output classes are changed the quicker the JVM can use refreshed classes.

To help you understand how incremental compilation works, the following provides a high-level overview:

* Gradle will recompile all classes *affected* by a change.
* A class is *affected* if it has been changed or if it depends on another affected class. This works no matter if the other class is defined in the same project, another project or even an external library.
* A class’s dependencies are determined from type references in its bytecode.
* Since constants can be inlined, any change to a constant will result in Gradle recompiling all source files. For that reason, you should try to minimize the use of constants in your source code and replace them with static methods where possible.
* Since source-retention annotations are not visible in bytecode, changes to a source-retention annotation will result in full recompilation.
* You can improve incremental compilation performance by applying good software desing principles like loose coupling. For instance, if you put an interface between a concrete class and its dependents, the dependent classes are only recompiled when the interface changes, but not when the implementation changes.
* The class analysis is cached in the project directory, so the first build after a clean checkout can be slower. Consider turning off the incremental compiler on your build server.

[Known issues](#37d8v38)

* If a compile task fails due to a compile error, it will do a full compilation again the next time it is invoked.

[Incremental annotation processing](#1mij5b1)

Starting with Gradle 4.7, the incremental compiler also supports incremental annotation processing. Annotation processors need to opt in to this feature, otherwise they will trigger a full recompilation.

As a user you can see which annotation processors are triggering full recompilations in the --info log. Incremental annotation processing will be deactivated if a custom executable or javaHome is configured on the compile task.

[Making an annotation processor incremental](#46i6nyu)

Please first have a look at [incremental Java compilation](#ndlcff), as incremental annotation processing builds on top of it.

Gradle supports incremental compilation for two common categories of annotation processors: "isolating" and "aggregating". Please consult the information below to decide which category fits your processor.

You can then register your processor for incremental compilation using a file in the processor’s META-INF directory. The format is one line per processor, with the fully qualified name of the processor class and its category separated by a comma.

[Example: Registering incremental annotation processors](#2lngy6n)

**processor/src/main/resources/META-INF/gradle/incremental.annotation.processors**

EntityProcessor,isolating  
ServiceRegistryProcessor,dynamic

If your processor can only decide at runtime whether it is incremental or not, you can declare it as "dynamic" in the META-INF descriptor and return its true type at runtime using the [Processor#getSupportedOptions()](https://docs.oracle.com/javase/10/docs/api/javax/annotation/processing/Processor.html#getSupportedOptions()) method.

[Example: Registering incremental annotation processors dynamically](#10sr8eg)

**processor/src/main/java/ServiceRegistryProcessor.java**

@Override  
**public** Set<String> getSupportedOptions() {  
 **return** Collections.singleton("org.gradle.annotation.processing.aggregating");  
}

Both categories have the following limitations:

* They must generate their files using the [Filer API](https://docs.oracle.com/javase/10/docs/api/javax/annotation/processing/Filer.html). Writing files any other way will result in silent failures later on, as these files won’t be cleaned up correctly. If your processor does this, it cannot be incremental.
* They must not depend on compiler-specific APIs like [com.sun.source.util.Trees](https://docs.oracle.com/javase/8/docs/jdk/api/javac/tree/com/sun/source/util/Trees.html). Gradle wraps the processing APIs, so attempts to cast to compiler-specific types will fail. If your processor does this, it cannot be incremental, unless you have some fallback mechanism.
* If they use [Filer#getResource](https://docs.oracle.com/javase/10/docs/api/javax/annotation/processing/Filer.html#getResource(javax.tools.JavaFileManager.Location,java.lang.CharSequence,java.lang.CharSequence)), Gradle will recompile all source files. See [gradle/issues/4701](https://github.com/gradle/gradle/issues/4701)
* If they use [Filer#createResource](https://docs.oracle.com/javase/10/docs/api/javax/annotation/processing/Filer.html#createResource(javax.tools.JavaFileManager.Location,java.lang.CharSequence,java.lang.CharSequence,javax.lang.model.element.Element...)), Gradle will recompile all source files. See [gradle/issues/4702](https://github.com/gradle/gradle/issues/4702)

["Isolating" annotation processors](#3kser29)

The fastest category, these look at each annotated element in isolation, creating generated files or validation messages for it. For instance an EntityProcessor could create a <TypeName>Repository for each type annotated with @Entity.

[Example: An isolated annotation processor](#1zxp1a2)

**processor/src/main/java/EntityProcessor.java**

Set<? **extends** Element> entities = roundEnv.getElementsAnnotatedWith(entityAnnotation);  
**for** (Element entity : entities) {  
 createRepository((TypeElement) entity);  
}

"Isolating" processors have the following limitations:

* They must make all decisions (code generation, validation messages) for an annotated type based on information reachable from its AST. This means you can analyze the types' super-class, method return types, annotations etc., even transitively. But you cannot make decisions based on unrelated elements in the RoundEnvironment. Doing so will result in silent failures because too few files will be recompiled later. If your processor needs to make decisions based on a combination of otherwise unrelated elements, mark it as "aggregating" instead.
* They must provide exactly one originating element for each file generated with the Filer API. If zero or many originating elements are provided, Gradle will recompile all source files.

When a source file is recompiled, Gradle will recompile all files generated from it. When a source file is deleted, the files generated from it are deleted.

["Aggregating" annotation processors](#4jxcjxv)

These can aggregate several source files into one ore more output files or validation messages. For instance, a ServiceRegistryProcessor could create a single ServiceRegistry with one method for each type annotated with @Service

[Example: An aggregating annotation processor](#2z2mu5o)

**processor/src/main/java/ServiceRegistryProcessor.java**

JavaFileObject serviceRegistry = filer.createSourceFile("ServiceRegistry");  
Writer writer = serviceRegistry.openWriter();  
writer.write("public class ServiceRegistry {");  
**for** (Element service : roundEnv.getElementsAnnotatedWith(serviceAnnotation)) {  
 addServiceCreationMethod(writer, (TypeElement) service);  
}  
writer.write("}");  
writer.close();

"Aggregating" processors have the following limitations:

* They can only read CLASS or RUNTIME retention annotations
* They can only read parameter names if the user passes the -parameters compiler argument.

Gradle will always reprocess (but not recompile) all annotated files that the processor was registered for. Gradle will always recompile any files the processor generates.

[Compile avoidance](#2hcarzs)

If a dependent project has changed in an [ABI](https://en.wikipedia.org/wiki/Application_binary_interface)-compatible way (only its private API has changed), then Java 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.

Some of the types of changes that do not affect the public API and are ignored:

* Changing a method body
* Changing a comment
* Adding, removing or changing private methods, fields, or inner classes
* Adding, removing or changing a resource
* Changing the name of jars or directories in the classpath
* Renaming a parameter

Compile-avoidance is deactivated if annotation processors are found on the compile classpath, because for annotation processors the implementation details matter. Annotation processors should be declared on the annotation processor path instead. Gradle 5.0 will ignore processors on the compile classpath.

[Example: Declaring annotation processors](#1e7x4dh)

**build.gradle**

dependencies {  
 *// The dagger compiler and its transitive dependencies will only be found on annotation processing classpath*  
 annotationProcessor 'com.google.dagger:dagger-compiler:2.8'  
  
 *// And we still need the Dagger library on the compile classpath itself*  
 implementation 'com.google.dagger:dagger:2.8'  
}

[Test](#3y7kn1a)

The test task is an instance of [Test](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html). It automatically detects and executes all unit tests in the test source set. It also generates a report once test execution is complete. JUnit and TestNG are both supported. Have a look at [Test](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html) for the complete API.

See the [Testing in Java & JVM projects](#4h042r0) chapter for more details.

[Jar](#2dcux93)

The jar task creates a JAR file containing the class files and resources of the project. The JAR file is declared as an artifact in the archives dependency configuration. This means that the JAR is available in the classpath of a dependent project. If you upload your project into a repository, this JAR is declared as part of the dependency descriptor. You can learn more about how to work with archives in [Archive creation in depth](#2iu58t6) and artifact configurations in [Legacy Publishing](#2dlolyb).

[Manifest](#si57gw)

Each jar or war object has a manifest property with a separate instance of [Manifest](http://docs.google.com/javadoc/org/gradle/api/java/archives/Manifest.html). When the archive is generated, a corresponding MANIFEST.MF file is written into the archive.

[Example: Customization of MANIFEST.MF](#3chsq4p)

**build.gradle**

jar {  
 manifest {  
 attributes("Implementation-Title": "Gradle",  
 "Implementation-Version": version)  
 }  
}

You can create stand-alone instances of a Manifest. You can use that for example, to share manifest information between jars.

[Example: Creating a manifest object.](#1rn30ci)

**build.gradle**

ext.sharedManifest = manifest {  
 attributes("Implementation-Title": "Gradle",  
 "Implementation-Version": version)  
}  
task fooJar(type: Jar) {  
 manifest = project.manifest {  
 from sharedManifest  
 }  
}

You can merge other manifests into any Manifest object. The other manifests might be either described by a file path or, like in the example above, by a reference to another Manifest object.

[Example: Separate MANIFEST.MF for a particular archive](#4bmqj0b)

**build.gradle**

task barJar(type: 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 by 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](http://docs.google.com/javadoc/org/gradle/api/java/archives/ManifestMergeDetails.html) instance for each entry of the resulting manifest. The merge is not immediately triggered by the from statement. It is done lazily, either when generating the jar, or by calling writeTo or effectiveManifest

You can easily write a manifest to disk.

[Example: Saving a MANIFEST.MF to disk](#2qs0t84)

**build.gradle**

jar.manifest.writeTo("$buildDir/mymanifest.mf")

[The Java Library Plugin](#1baon6m)

The Java Library plugin expands the capabilities of the Java plugin 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](#15xb3fx)

To use the Java Library plugin, include the following in your build script:

[Example: Using the Java Library plugin](#3pwym3q)

**build.gradle**

apply plugin: 'java-library'

[API and implementation separation](#328p01r)

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](#37m2jsg) 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: Declaring API and implementation dependencies](#2528wbj)

**build.gradle**

dependencies {  
 api 'commons-httpclient:commons-httpclient:3.1'  
 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 configuration still exists but should not be used as it will not offer the guarantees that the api and implementation configurations provide. |
| --- | --- |

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 compile scoped dependencies, while the runtime classpath adds the 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. 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.

| **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](#1iljr09)

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](#k7j6jc)

**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.httpclient.\*;  
**import** org.apache.commons.httpclient.methods.\*;  
**import** org.apache.commons.lang3.exception.ExceptionUtils;  
**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) {  
 GetMethod method = **new** GetMethod(url);  
 **try** {  
 **int** statusCode = doGet(method);  
 **return** method.getResponseBody();  
  
 } **catch** (Exception e) {  
 ExceptionUtils.rethrow(e); *// this dependency is internal only*  
 } **finally** {  
 method.releaseConnection();  
 }  
 **return** null;  
 }  
  
 *// GetMethod is used in a private method, so doesn't belong to the API*  
 **private** **int** doGet(GetMethod method) **throws** Exception {  
 **int** statusCode = client.executeMethod(method);  
 **if** (statusCode != HttpStatus.SC\_OK) {  
 System.err.println("Method failed: " + method.getStatusLine());  
 }  
 **return** statusCode;  
 }  
}

The *public* constructor of HttpClientWrapper uses HttpClient as a parameter, so it is exposed to consumers and therefore belongs to the API. Note that GetMethod is used in the signature of a *private* method, and so it doesn’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 commons-httpclient is an API dependency, whereas commons-lang is an implementation dependency. This conclusion translates into the following declaration in the build script:

[Example: Declaring API and implementation dependencies](#3476p75)

**build.gradle**

dependencies {  
 api 'commons-httpclient:commons-httpclient:3.1'  
 implementation 'org.apache.commons:commons-lang3:3.5'  
}

[The Java Library plugin configurations](#1jcgzey)

The following graph describes the main configurations 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
* The configurations in *white* are configurations inherited from the Java plugin

And the next graph describes the test configurations setup:



| **Note** | The *compile*, *testCompile*, *runtime* and *testRuntime* configurations inherited from the Java plugin are still available but are deprecated. You should avoid using them, as they are only kept for backwards compatibility. |
| --- | --- |

The role of each configuration is described in the following tables:

Table 17. Java Library plugin - configurations used to declare dependencies

| **Configuration name** | **Role** | **Consumable?** | **Resolvable?** | **Description** |
| --- | --- | --- | --- | --- |
| api | Declaring API dependencies | no | no | This is where you should declare dependencies which are transitively exported to consumers, for compile. |
| implementation | Declaring implementation dependencies | no | no | This is where you should declare dependencies which are purely internal and not meant to be exposed to consumers. |
| compileOnly | Declaring compile only dependencies | yes | yes | This is where you should declare dependencies which are only required at compile time, but should not leak into the runtime. This typically includes dependencies which are shaded when found at runtime. |
| runtimeOnly | Declaring runtime dependencies | no | no | This is where you should declare dependencies which are only required at runtime, and not at compile time. |
| testImplementation | Test dependencies | no | no | This is where you should declare dependencies which are used to compile tests. |
| testCompileOnly | Declaring test compile only dependencies | yes | yes | This is where you should 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. |
| testRuntimeOnly | Declaring test runtime dependencies | no | no | This is where you should declare dependencies which are only required at test runtime, and not at test compile time. |

Table 18. Java Library plugin — configurations used by consumers

| **Configuration name** | **Role** | **Consumable?** | **Resolvable?** | **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. Unlike the default configuration, this doesn’t leak implementation or runtime dependencies. |
| runtimeElements | 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 19. 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 |

[Known issues](#43c4i2r)

[Compatibility with other plugins](#2ihesak)

At the moment the Java Library plugin is only wired to behave correctly with the java plugin. Other plugins, such as the Groovy plugin, may not behave correctly. In particular, if the Groovy plugin is used in addition to the java-library plugin, then consumers may not get the Groovy classes when they consume the library. To workaround this, you need to explicitly wire the Groovy compile dependency, like this:

[Example: Configuring the Groovy plugin to work with Java Library](#xmp2id)

**a/build.gradle**

configurations {  
 apiElements {  
 outgoing.variants.getByName('classes').artifact(  
 file: compileGroovy.destinationDir,  
 type: ArtifactTypeDefinition.JVM\_CLASS\_DIRECTORY,  
 builtBy: compileGroovy)  
 }  
}

[Increased memory usage for consumers](#3hmcl66)

When a project uses the Java Library plugin, consumers will use the output classes directory of this project directly on their compile classpath, instead of the jar file if the project uses the Java plugin. 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.

[The Java Library Distribution Plugin](#3vac5uf)

| **Note** | The Java library distribution plugin is currently [incubating](#2xcytpi). Please be aware that the DSL and other configuration may change in later Gradle versions. |
| --- | --- |

The Java library distribution plugin adds support for building a distribution ZIP for a Java library. The distribution contains the JAR file for the library and its dependencies.

[Usage](#1wrmvdz)

To use the Java library distribution plugin, include the following in your build script:

[Example: Using the Java library distribution plugin](#4grae1s)

**build.gradle**

apply plugin: 'java-library-distribution'

To define the name for the distribution you have to set the baseName property as shown below:

[Example: Configure the distribution name](#2vwko9l)

**build.gradle**

distributions {  
 main{  
 baseName = 'my-name'  
 }  
}

The plugin builds a distribution for your library. The distribution will package up the runtime dependencies of the library. All files stored in src/main/dist will be added to the root of the archive distribution. You can run “gradle distZip” to create a ZIP file containing the distribution.

[Tasks](#1b1uyhe)

The Java library distribution plugin adds the following tasks to the project.

distZip — [Zip](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Zip.html)

*Depends on*: jar

Creates a full distribution ZIP archive including runtime libraries.

[Including other resources in the distribution](#3v1ih57)

All of the files from the src/dist directory are copied. To include any static files in the distribution, simply arrange them in the src/dist directory, or add them to the content of the distribution.

[Example: Include files in the distribution](#2a6srd0)

**build.gradle**

distributions {  
 main {  
 baseName = 'my-name'  
 contents {  
 from { 'src/dist' }  
 }  
 }  
}

[Dependency Management for Java Projects](#2afmg28)

This chapter explains how to apply basic dependency management concepts to Java-based projects. For a detailed introduction to dependency management, see [Introduction to Dependency Management](#1v1yuxt).

[Dissecting a typical build script](#pc31kt)

Let’s have a look at a very simple build script for a Java-based project. It applies the [Java Library plugin](#1baon6m) which automatically introduces a standard project layout, provides tasks for performing typical work and adequate support for dependency management.

[Example: Dependency declarations for a Java-based project](#39bqk8m)

**build.gradle**

apply plugin: 'java-library'  
  
repositories {  
 mavenCentral()  
}  
  
dependencies {  
 implementation 'org.hibernate:hibernate-core:3.6.7.Final'  
 api 'com.google.guava:guava:23.0'  
 testImplementation 'junit:junit:4.+'  
}

The [Project.dependencies{}](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:dependencies(groovy.lang.Closure)) 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{}](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:repositories(groovy.lang.Closure)). The following sections explain each aspect in more detail.

[Declaring module dependencies](#1oh0ugf)

There are various [types of dependencies](#2u6wntf) that you can declare. One such type is a *module dependency*. A [module dependency](#397tsf8) 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](#48god48):

[Example: Definition of a module dependency](#2nlync1)

**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](#3tbugp1).

[Using dependency configurations](#48god48)

A [Configuration](http://docs.google.com/dsl/org.gradle.api.artifacts.Configuration.html) 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](#1jcgzey).

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{}](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:configurations(groovy.lang.Closure)). See [Managing Dependency Configurations](#37m2jsg) for the details of defining and customizing dependency configurations.

[Declaring common Java repositories](#12r8xju)

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](#19c6y18), 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{}](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:repositories(groovy.lang.Closure)) before you can use module dependencies. One option is use the Maven Central repository:

[Example: Usage of Maven central repository](#3mqwg7n)

**build.gradle**

repositories {  
 mavenCentral()  
}

You can also have repositories on the local file system. This works for both Maven and Ivy repositories.

[Example: Usage of a local Ivy directory](#21w6qfg)

**build.gradle**

repositories {  
 ivy {  
 *// URL can refer to a local directory*  
 url "../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](#28h4qwu).

[Publishing artifacts](#h1h0n9)

Dependency configurations are also used to publish files. Gradle calls these files *publication artifacts*, or usually just *artifacts*. As a user you will need to tell Gradle where to publish the artifacts. You do this by declaring repositories for the uploadArchives task. Here’s an example of publishing to a Maven repository:

[Example: Publishing to a Maven repository](#3114jb2)

**build.gradle**

apply plugin: 'maven'  
  
uploadArchives {  
 repositories {  
 mavenDeployer {  
 repository(url: "file://localhost/tmp/myRepo/")  
 }  
 }  
}

Now, when you run gradle uploadArchives, Gradle will build the JAR file, generate a .pom file and upload the artifacts.

To learn more about publishing artifacts, have a look at [Legacy Publishing](#2dlolyb).

[Using Ant from Gradle](#pkwqa1)

Gradle provides excellent integration with Ant. You can use individual Ant tasks or entire Ant builds in your Gradle builds. In fact, you will find that it’s far easier and more powerful using Ant tasks in a Gradle build script, than it is to use Ant’s XML format. You could even use Gradle simply as a powerful Ant task scripting tool.

Ant can be divided into two layers. The first layer is the Ant language. It provides the syntax for the build.xml file, the handling of the targets, special constructs like macrodefs, and so on. In other words, everything except the Ant tasks and types. Gradle understands this language, and allows you to 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.

The second layer of Ant is its wealth of Ant tasks and types, like javac, copy or jar. For this layer Gradle provides integration simply by relying on Groovy, and the fantastic AntBuilder.

Finally, since build scripts are Groovy scripts, you can always execute an Ant build as an external process. Your build script may contain statements like: "ant clean compile".execute().[[15](#1g6etiv)]

You can use Gradle’s Ant integration as a path for migrating your build from Ant to Gradle. For example, you could start by importing your existing Ant build. Then you could move your dependency declarations from the Ant script to your build file. Finally, you could move your tasks across to your build file, or replace them with some of Gradle’s plugins. This process can be done in parts over time, and you can have a working Gradle build during the entire process.

[Using Ant tasks and types in your build](#4062c6o)

In your build script, a property called ant is provided by Gradle. This is a reference to an [AntBuilder](http://docs.google.com/javadoc/org/gradle/api/AntBuilder.html) instance. This AntBuilder is used to access Ant tasks, types and properties from your build script. There is a very simple mapping from Ant’s build.xml format to Groovy, which is explained below.

You execute an Ant task by calling a method on the AntBuilder instance. You use the task name as the method name. For example, you execute the Ant echo task by calling 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. Notice that we can also mix Groovy code and the Ant task markup. This can be extremely powerful.

[Example: Using an Ant task](#2fbcmeh)

**build.gradle**

task hello {  
 doLast {  
 String greeting = 'hello from Ant'  
 ant.echo(message: greeting)  
 }  
}

**Output of gradle hello**

> gradle hello  
  
> Task :hello  
[ant:echo] hello from Ant  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

You pass nested text to an Ant task by passing it as a parameter of the task method call. In this example, we pass the message for the echo task as nested text:

[Example: Passing nested text to an Ant task](#ugmwma)

**build.gradle**

task hello {  
 doLast {  
 ant.echo('hello from Ant')  
 }  
}

**Output of gradle hello**

> 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.

[Example: Passing nested elements to an Ant task](#3egafa3)

**build.gradle**

task zip {  
 doLast {  
 ant.zip(destfile: 'archive.zip') {  
 fileset(dir: 'src') {  
 include(name: '\*\*.xml')  
 exclude(name: '\*\*.java')  
 }  
 }  
 }  
}

You can access Ant types in the same way that you access tasks, using the name of the type as the method name. The method call returns the Ant data type, which you can then use directly in your build script. In the following example, we create an Ant path object, then iterate over the contents of it.

[Example: Using an Ant type](#1tlkphw)

**build.gradle**

task list {  
 doLast {  
 **def** path = ant.path {  
 fileset(dir: 'libs', includes: '\*.jar')  
 }  
 path.list().each {  
 println it  
 }  
 }  
}

More information about AntBuilder can be found in 'Groovy in Action' 8.4 or at the [Groovy Wiki](http://groovy-lang.org/scripting-ant.html).

[Using custom Ant tasks in your build](#4dl885p)

To make custom tasks available in your build, you can 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.

[Example: Using a custom Ant task](#2sqiidi)

**build.gradle**

task check {  
 doLast {  
 ant.taskdef(resource: 'checkstyletask.properties') {  
 classpath {  
 fileset(dir: 'libs', includes: '\*.jar')  
 }  
 }  
 ant.checkstyle(config: 'checkstyle.xml') {  
 fileset(dir: 'src')  
 }  
 }  
}

You can use Gradle’s dependency management to assemble the classpath to use for the custom tasks. To do this, you need to define a custom configuration for the classpath, then add some dependencies to the configuration. This is described in more detail in [Declaring Dependencies](http://docs.google.com/declaring_dependencies.html).

[Example: Declaring the classpath for a custom Ant task](#17vsslb)

**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.

[Example: Using a custom Ant task and dependency management together](#3rvgb94)

**build.gradle**

task 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](#270qlgx)

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 exactly the same way as Gradle tasks.

[Example: Importing an Ant build](#m60voq)

**build.gradle**

ant.importBuild 'build.xml'

**build.xml**

<project>  
 <target name="hello">  
 <echo>Hello, from Ant</echo>  
 </target>  
</project>

**Output of gradle hello**

> gradle hello  
  
> Task :hello  
[ant:echo] Hello, from Ant  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

You can add a task which depends on an Ant target:

[Example: Task that depends on Ant target](#365oecj)

**build.gradle**

ant.importBuild 'build.xml'  
  
task intro(dependsOn: hello) {  
 doLast {  
 println 'Hello, from Gradle'  
 }  
}

**Output of gradle intro**

> 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 behaviour to an Ant target:

[Example: Adding behaviour to an Ant target](#1layokc)

**build.gradle**

ant.importBuild 'build.xml'  
  
hello {  
 doLast {  
 println 'Hello, from Gradle'  
 }  
}

**Output of gradle hello**

> 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:

[Example: Ant target that depends on Gradle task](#45am785)

**build.gradle**

ant.importBuild 'build.xml'  
  
task intro {  
 doLast {  
 println 'Hello, from Gradle'  
 }  
}

**build.xml**

<project>  
 <target name="hello" depends="intro">  
 <echo>Hello, from Ant</echo>  
 </target>  
</project>

**Output of gradle hello**

> 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)](http://docs.google.com/javadoc/org/gradle/api/AntBuilder.html#importBuild-java.lang.Object-org.gradle.api.Transformer-) method.

[Example: Renaming imported Ant targets](#2kfwhfy)

**build.gradle**

ant.importBuild('build.xml') { antTargetName ->  
 'a-' + antTargetName  
}

**build.xml**

<project>  
 <target name="hello">  
 <echo>Hello, from Ant</echo>  
 </target>  
</project>

**Output of gradle a-hello**

> gradle a-hello  
  
> Task :a-hello  
[ant:echo] Hello, from Ant  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

Note that while the second argument to this method should be a [Transformer](http://docs.google.com/javadoc/org/gradle/api/Transformer.html), when programming in Groovy we can simply 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](http://mrhaki.blogspot.ie/2013/11/groovy-goodness-implicit-closure.html).

[Ant properties and references](#zl6rnr)

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. Below are some examples of how to do this.

[Example: Setting an Ant property](#3jkuabk)

**build.gradle**

ant.buildDir = buildDir  
ant.properties.buildDir = buildDir  
ant.properties['buildDir'] = buildDir  
ant.property(name: 'buildDir', location: buildDir)

**build.xml**

<echo>buildDir = ${buildDir}</echo>

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. Below are some examples.

[Example: Getting an Ant property](#1yq4kjd)

**build.xml**

<property name="antProp" value="a property defined in an Ant build"/>

**build.gradle**

println ant.antProp  
println ant.properties.antProp  
println ant.properties['antProp']

There are several ways to set an Ant reference:

[Example: Setting an Ant reference](#4ips376)

**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:

[Example: Getting an Ant reference](#2xv2dez)

**build.xml**

<path id="antPath" location="libs"/>

**build.gradle**

println ant.references.antPath  
println ant.references['antPath']

[Ant logging](#1d0cnms)

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:

Table 20. Ant message priority mapping

| **Ant Message Priority** | **Gradle Log Level** |
| --- | --- |
| *VERBOSE* | DEBUG |
| *DEBUG* | DEBUG |
| *INFO* | INFO |
| *WARN* | WARN |
| *ERROR* | ERROR |

[Fine tuning Ant logging](#3x006al)

The default mapping of Ant message priority to Gradle log level can sometimes be problematic. For example, there is no message priority that 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, to suppress those messages would require the build to be run at a higher log level, such as QUIET. However, this could result in other, desirable output being suppressed.

To help with this, Gradle allows the user to fine tune the Ant logging and control the mapping of message priority to 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)](http://docs.google.com/javadoc/org/gradle/api/AntBuilder.html#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 most at INFO.

For example, the following changes the mapping such that Ant *INFO* priority messages are exposed at the LIFECYCLE log level.

[Example: Fine tuning Ant logging](#2c5agie)

**build.gradle**

ant.lifecycleLogLevel = "INFO"  
  
task hello {  
 doLast {  
 ant.echo(level: "info", message: "hello from info priority!")  
 }  
}

**Output of gradle hello**

> 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 would be suppressed by default.

[API](#rakqq7)

The Ant integration is provided by [AntBuilder](http://docs.google.com/javadoc/org/gradle/api/AntBuilder.html).

[The ANTLR Plugin](#39kk8xu)

The ANTLR plugin extends the Java plugin to add support for generating parsers using [ANTLR](http://www.antlr.org/).

| **Note** | The ANTLR plugin supports ANTLR version 2, 3 and 4. |
| --- | --- |

[Usage](#3ba89e0)

To use the ANTLR plugin, include the following in your build script:

[Example: Using the ANTLR plugin](#1qfijlt)

**build.gradle**

apply plugin: 'antlr'

[Tasks](#4af629m)

The ANTLR plugin adds a number of tasks to your project, as shown below.

generateGrammarSource — [AntlrTask](http://docs.google.com/dsl/org.gradle.api.plugins.antlr.AntlrTask.html)

Generates the source files for all production ANTLR grammars.

generateTestGrammarSource — [AntlrTask](http://docs.google.com/dsl/org.gradle.api.plugins.antlr.AntlrTask.html)

Generates the source files for all test ANTLR grammars.

generate*SourceSet*GrammarSource — [AntlrTask](http://docs.google.com/dsl/org.gradle.api.plugins.antlr.AntlrTask.html)

Generates the source files for all ANTLR grammars for the given source set.

The ANTLR plugin adds the following dependencies to tasks added by the Java plugin.

Table 21. ANTLR plugin - additional task dependencies

| **Task name** | **Depends on** |
| --- | --- |
| compileJava | generateGrammarSource |
| compileTestJava | generateTestGrammarSource |
| compile*SourceSet*Java | generate*SourceSet*GrammarSource |

[Project layout](#2pkgchf)

src/main/antlr

Production ANTLR grammar files. If the ANTLR grammar is organized in packages, the structure in the antlr folder should reflect the package structure. This ensures that the generated sources end up in the correct target subfolder.

src/test/antlr

Test ANTLR grammar files.

src/*sourceSet*/antlr

ANTLR grammar files for the given source set.

[Dependency management](#14pqmp8)

The ANTLR plugin adds an antlr dependency configuration which provides the ANTLR implementation to use. The following example shows how to use ANTLR version 3.

[Example: Declare ANTLR version](#3ope5d1)

**build.gradle**

repositories {  
 mavenCentral()  
}  
  
dependencies {  
 antlr "org.antlr:antlr:3.5.2" *// use ANTLR version 3*  
 *// antlr "org.antlr:antlr4:4.5" // use ANTLR version 4*  
}

If no dependency is declared, antlr:antlr:2.7.7 will be used as the default. To use a different ANTLR version add the appropriate dependency to the antlr dependency configuration as above.

[Convention properties](#23uofku)

The ANTLR plugin does not add any convention properties.

[Source set properties](#izypsn)

The ANTLR plugin adds the following properties to each source set in the project.

antlr — [SourceDirectorySet](http://docs.google.com/dsl/org.gradle.api.file.SourceDirectorySet.html)

The ANTLR grammar files of this source set. Contains all .g or .g4 files found in the ANTLR source directories, and excludes all other types of files. *Default value is non-null.*

antlr.srcDirs — Set<File>

The source directories containing the ANTLR grammar files of this source set. Can set using anything [that implicitly converts to a file collection](#115hox2). Default value is [*projectDir*/src/*name*/antlr].

[Controlling the ANTLR generator process](#32zm8gg)

The ANTLR tool is executed in a forked process. This allows fine grained control over memory settings for the ANTLR process. To set the heap size of an ANTLR process, the maxHeapSize property of [AntlrTask](http://docs.google.com/dsl/org.gradle.api.plugins.antlr.AntlrTask.html) can be used. To pass additional command-line arguments, append to the arguments property of [AntlrTask](http://docs.google.com/dsl/org.gradle.api.plugins.antlr.AntlrTask.html).

[Example: Setting custom max heap size and extra arguments for ANTLR](#1i4wio9)

**build.gradle**

generateGrammarSource {  
 maxHeapSize = "64m"  
 arguments += ["-visitor", "-long-messages"]  
}

[The Application Plugin](#1opuj5n)

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](#2w5ecyt). The main source set is effectively the “application”.

Applying the Application plugin also implicitly applies the [Distribution plugin](#1rvwp1q). A main distribution is created that packages up the application, including code dependencies and generated start scripts.

[Usage](#424k1c2)

To use the application plugin, include the following in your build script:

[Example: Using the application plugin](#2h9ubjv)

**build.gradle**

apply plugin: 'application'

The only mandatory configuration for the plugin is the specification of the main class (i.e. entry point) of the application.

[Example: Configure the application main class](#wf4lro)

**build.gradle**

mainClassName = "org.gradle.sample.Main"

You can run the application by executing the run task (type: [JavaExec](http://docs.google.com/javadoc/org.gradle.api.tasks.JavaExec.html)). 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)](http://docs.google.com/javadoc/org/gradle/api/tasks/JavaExec.html#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)](http://docs.google.com/javadoc/org/gradle/api/tasks/JavaExec.html#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: Configure default JVM settings](#3ges4fh)

**build.gradle**

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: Configure custom directory for start scripts](#1vk2ena)

**build.gradle**

executableDir = "custom\_bin\_dir"

[The distribution](#4fjpxb3)

A distribution of the application can be created, by way of the [Distribution plugin](#1rvwp1q) (which is automatically applied). A main distribution is created with the following content:

Table 22. Distribution content

| **Location** | **Content** |
| --- | --- |
| (root dir) | src/dist |
| lib | All runtime dependencies and main source set class files. |
| bin | Start scripts (generated by createStartScripts 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](http://docs.google.com/javadoc/org.gradle.api.file.file.CopySpec.html) exposed by the main distribution.

[Example: Include output from other tasks in the application distribution](#2up07iw)

**build.gradle**

task createDocs {  
 **def** docs = file("$buildDir/docs")  
 outputs.dir docs  
 doLast {  
 docs.mkdirs()  
 **new** File(docs, "readme.txt").write("Read me!")  
 }  
}  
  
distributions {  
 main {  
 contents {  
 from(createDocs) {  
 into "docs"  
 }  
 }  
 }  
}

By specifying that the distribution should include the task’s output files (see [more about tasks](#4hae2tp)), 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.

[Example: Automatically creating files for distribution](#19uahqp)

**Output of gradle distZip**

> gradle distZip  
> Task :createDocs  
> Task :compileJava  
> Task :processResources NO-SOURCE  
> Task :classes  
> Task :jar  
> Task :startScripts  
> Task :distZip  
  
BUILD SUCCESSFUL in 0s  
5 actionable tasks: 5 executed

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](#3tty0ei)

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](http://docs.google.com/dsl/org.gradle.jvm.application.tasks.CreateStartScripts.html) for more details and customization examples.

[Tasks](#28z8amb)

The Application plugin adds the following tasks to the project.

run — [JavaExec](http://docs.google.com/dsl/org.gradle.api.tasks.JavaExec.html)

*Depends on*: classes

Starts the application.

startScripts — [CreateStartScripts](http://docs.google.com/dsl/org.gradle.api.tasks.application.tasks.application.CreateStartScripts.html)

*Depends on*: jar

Creates OS specific scripts to run the project as a JVM application.

installDist — [Sync](http://docs.google.com/dsl/org.gradle.api.tasks.Sync.html)

*Depends on*: jar, startScripts

Installs the application into a specified directory.

distZip — [Zip](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.tasks.bundling.Zip.html)

*Depends on*: jar, startScripts

Creates a full distribution ZIP archive including runtime libraries and OS specific scripts.

distTar — [Tar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.tasks.bundling.Tar.html)

*Depends on*: jar, startScripts

Creates a full distribution TAR archive including runtime libraries and OS specific scripts.

[Convention properties](#o4iku4)

The application plugin adds some properties to the project, which you can use to configure its behaviour. See the [Project](http://docs.google.com/dsl/org.gradle.api.Project.html) class in the API documentation.

[The Checkstyle Plugin](#48pi1tg)

The Checkstyle plugin performs quality checks on your project’s Java source files using [Checkstyle](http://checkstyle.sourceforge.net/index.html) and generates reports from these checks.

[Usage](#38463hx)

To use the Checkstyle plugin, include the following in your build script:

[Example: Using the Checkstyle plugin](#1n9gdpq)

**build.gradle**

apply plugin: 'checkstyle'

The plugin adds a number of tasks to the project that perform the quality checks. You can execute the checks by running gradle check.

Note that Checkstyle will run with the same Java version used to run Gradle.

[Tasks](#4793wdj)

The Checkstyle plugin adds the following tasks to the project:

checkstyleMain — [Checkstyle](http://docs.google.com/dsl/org.gradle.api.plugins.quality.Checkstyle.html)

*Depends on*: classes

Runs Checkstyle against the production Java source files.

checkstyleTest — [Checkstyle](http://docs.google.com/dsl/org.gradle.api.plugins.quality.Checkstyle.html)

*Depends on*: testClasses

Runs Checkstyle against the test Java source files.

checkstyle*SourceSet* — [Checkstyle](http://docs.google.com/dsl/org.gradle.api.plugins.quality.Checkstyle.html)

*Depends on*: *sourceSet*Classes

Runs Checkstyle against the given source set’s Java source files.

[Dependencies added to other tasks](#2mee6lc)

The Checkstyle plugin adds the following dependencies to tasks defined by the Java plugin.

check

*Depends on*: All Checkstyle tasks, including checkstyleMain and checkstyleTest.

[Project layout](#11jogt5)

By default, the Checkstyle plugin expects configuration files to be placed in the root project, but this can be changed.

<root>  
└── config  
 └── checkstyle (1)  
 └── checkstyle.xml (2)  
 └── suppressions.xml

1. Checkstyle configuration files go here
2. Primary Checkstyle configuration file

[Dependency management](#3ljbzgy)

The Checkstyle plugin adds the following dependency configurations:

Table 23. Checkstyle plugin - dependency configurations

| **Name** | **Meaning** |
| --- | --- |
| checkstyle | The Checkstyle libraries to use |

[Configuration](#20om9or)

See the [CheckstyleExtension](http://docs.google.com/dsl/org.gradle.api.plugins.quality.CheckstyleExtension.html) class in the API documentation.

[Built-in variables](#4ko9sck)

The Checkstyle plugin defines a config\_loc property that can be used in Checkstyle configuration files to define paths to other configuration files like suppressions.xml.

[Example: Using the config\_loc property](#2ztk2kd)

**config/checkstyle/checkstyle.xml**

<module name="SuppressionFilter">  
 <property name="file" value="${config\_loc}/suppressions.xml"/>  
</module>

[Customizing the HTML report](#1eyucs6)

The HTML report generated by the [Checkstyle](http://docs.google.com/dsl/org.gradle.api.plugins.quality.Checkstyle.html) task can be customized using a XSLT stylesheet, for example to highlight specific errors or change its appearance:

[Example: Customizing the HTML report](#3yyhvfz)

**build.gradle**

tasks.withType(Checkstyle) {  
 reports {  
 xml.enabled false  
 html.enabled true  
 html.stylesheet resources.text.fromFile('config/xsl/checkstyle-custom.xsl')  
 }  
}

[View a sample Checkstyle stylesheet.](https://github.com/checkstyle/contribution/tree/master/xsl)

[The FindBugs Plugin](#2nusc19)

The FindBugs plugin performs quality checks on your project’s Java source files using [FindBugs](http://findbugs.sourceforge.net) and generates reports from these checks.

[Usage](#2e3s5ns)

To use the FindBugs plugin, include the following in your build script:

[Example: Using the FindBugs plugin](#t92fvl)

**build.gradle**

apply plugin: 'findbugs'

The plugin adds a number of tasks to the project that perform the quality checks. You can execute the checks by running gradle check.

Note that Findbugs will run with the same Java version used to run Gradle.

[Tasks](#3d8pyje)

The FindBugs plugin adds the following tasks to the project:

findbugsMain — [FindBugs](http://docs.google.com/dsl/org.gradle.api.plugins.quality.FindBugs.html)

*Depends on*: classes

Runs FindBugs against the production Java source files.

findbugsTest — [FindBugs](http://docs.google.com/dsl/org.gradle.api.plugins.quality.FindBugs.html)

*Depends on*: testClasses

Runs FindBugs against the test Java source files.

findbugs*SourceSet* — [FindBugs](http://docs.google.com/dsl/org.gradle.api.plugins.quality.FindBugs.html)

*Depends on*: *sourceSet*Classes

Runs FindBugs against the given source set’s Java source files.

The FindBugs plugin adds the following dependencies to tasks defined by the Java plugin.

Table 24. FindBugs plugin - additional task dependencies

| **Task name** | **Depends on** |
| --- | --- |
| check | All FindBugs tasks, including findbugsMain and findbugsTest. |

[Dependency management](#1se08r7)

The FindBugs plugin adds the following dependency configurations:

Table 25. FindBugs plugin - dependency configurations

| **Name** | **Meaning** |
| --- | --- |
| findbugs | The FindBugs libraries to use |

[Configuration](#4cdnrf0)

See the [FindBugsExtension](http://docs.google.com/dsl/org.gradle.api.plugins.quality.FindBugsExtension.html) class in the API documentation.

[Customizing the HTML report](#2riy1mt)

The HTML report generated by the [FindBugs](http://docs.google.com/dsl/org.gradle.api.plugins.quality.FindBugs.html) task can be customized using a XSLT stylesheet, for example to highlight specific errors or change its appearance:

[Example: Customizing the HTML report](#16o8bum)

**build.gradle**

tasks.withType(FindBugs) {  
 reports {  
 xml.enabled false  
 html.enabled true  
 html.stylesheet resources.text.fromFile('config/xsl/findbugs-custom.xsl')  
 }  
}

[View a sample FindBugs stylesheet.](https://github.com/findbugsproject/findbugs/tree/master/findbugs/src/xsl)

[The JaCoCo Plugin](#1302m92)

| **Note** | The JaCoCo plugin is currently [incubating](#2xcytpi). Please be aware that the DSL and other configuration may change in later Gradle versions. |
| --- | --- |

The JaCoCo plugin provides code coverage metrics for Java code via integration with [JaCoCo](http://www.eclemma.org/jacoco/).

[Getting Started](#3qnvuif)

To get started, apply the JaCoCo plugin to the project you want to calculate code coverage for.

[Example: Applying the JaCoCo plugin](#25t64q8)

**build.gradle**

apply plugin: "jacoco"

If the Java plugin is also applied to your project, a new task named jacocoTestReport is created that depends on the test task. The report is available at *$buildDir*/reports/jacoco/test. By default, a HTML report is generated.

[Configuring the JaCoCo Plugin](#kygey1)

The JaCoCo plugin adds a project extension named jacoco of type [JacocoPluginExtension](http://docs.google.com/dsl/org.gradle.testing.jacoco.plugins.JacocoPluginExtension.html), which allows configuring defaults for JaCoCo usage in your build.

[Example: Configuring JaCoCo plugin settings](#34y3xlu)

**build.gradle**

jacoco {  
 toolVersion = "0.8.1"  
 reportsDir = file("$buildDir/customJacocoReportDir")  
}

Table 26. Gradle defaults for JaCoCo properties

| **Property** | **Gradle default** |
| --- | --- |
| reportsDir | *$buildDir*/reports/jacoco |

[JaCoCo Report configuration](#1k3e7tn)

The [JacocoReport](http://docs.google.com/dsl/org.gradle.testing.jacoco.tasks.JacocoReport.html) task can be used to generate code coverage reports in different formats. It implements the standard Gradle type [Reporting](http://docs.google.com/dsl/org.gradle.api.reporting.Reporting.html) and exposes a report container of type [JacocoReportsContainer](http://docs.google.com/javadoc/org/gradle/testing/jacoco/tasks/JacocoReportsContainer.html).

[Example: Configuring test task](#4431qhg)

**build.gradle**

jacocoTestReport {  
 reports {  
 xml.enabled false  
 csv.enabled false  
 html.destination file("${buildDir}/jacocoHtml")  
 }  
}



[Enforcing code coverage metrics](#2j8c0p9)

| **Note** | This feature requires the use of JaCoCo version 0.6.3 or higher. |
| --- | --- |

The [JacocoCoverageVerification](http://docs.google.com/dsl/org.gradle.testing.jacoco.tasks.JacocoCoverageVerification.html) task can be used to verify if code coverage metrics are met based on configured rules. Its API exposes the method [JacocoCoverageVerification.violationRules(org.gradle.api.Action)](http://docs.google.com/javadoc/org/gradle/testing/jacoco/tasks/JacocoCoverageVerification.html#violationRules-org.gradle.api.Action-) which is used as main entry point for configuring rules. Invoking any of those methods returns an instance of [JacocoViolationRulesContainer](http://docs.google.com/javadoc/org/gradle/testing/jacoco/tasks/rules/JacocoViolationRulesContainer.html) providing extensive configuration options. The build fails if any of the configured rules are not met. JaCoCo only reports the first violated rule.

Code coverage requirements can be specified for a project as a whole, for individual files, and for particular JaCoCo-specific types of coverage, e.g., lines covered or branches covered. The following example describes the syntax.

[Example: Configuring violation rules](#ydmax2)

**build.gradle**

jacocoTestCoverageVerification {  
 violationRules {  
 rule {  
 limit {  
 minimum = 0.5  
 }  
 }  
  
 rule {  
 enabled = false  
 element = 'CLASS'  
 includes = ['org.gradle.\*']  
  
 limit {  
 counter = 'LINE'  
 value = 'TOTALCOUNT'  
 maximum = 0.3  
 }  
 }  
 }  
}

| **Note** | The code for this example can be found at samples/testing/jacoco/quickstart in the ‘-all’ distribution of Gradle. |
| --- | --- |

The [JacocoCoverageVerification](http://docs.google.com/dsl/org.gradle.testing.jacoco.tasks.JacocoCoverageVerification.html) task is not a task dependency of the check task provided by the Java plugin. There is a good reason for it. The task is currently not incremental as it doesn’t declare any outputs. Any violation of the declared rules would automatically result in a failed build when executing the check task. This behavior might not be desirable for all users. Future versions of Gradle might change the behavior.

[JaCoCo specific task configuration](#3id9tkv)

The JaCoCo plugin adds a [JacocoTaskExtension](http://docs.google.com/dsl/org.gradle.testing.jacoco.plugins.JacocoTaskExtension.html) extension to all tasks of type [Test](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html). This extension allows the configuration of the JaCoCo specific properties of the test task.

[Example: Configuring test task](#1xik3so)

**build.gradle**

test {  
 jacoco {  
 append = false  
 destinationFile = file("$buildDir/jacoco/jacocoTest.exec")  
 classDumpDir = file("$buildDir/jacoco/classpathdumps")  
 }  
}

| **Note** | Using the configuration append = true (the default) causes the JaCoCo agent to append to a shared output file that may be left over from a different test execution. If append = true, Gradle disables caching for the Test task since it cannot guarantee the same results each time. |
| --- | --- |

**Default values of the JaCoCo Task extension**

test {  
 jacoco {  
 append = true  
 enabled = true  
 destPath = "$buildDir/jacoco"  
 includes = **[]**  
 excludes = **[]**  
 excludeClassLoaders = **[]**  
 includeNoLocationClasses = false  
 sessionId = "<auto-generated value>"  
 dumpOnExit = true  
 classDumpDir = null  
 output = Output.FILE  
 address = "localhost"  
 port = 6300  
 jmx = false  
 }  
}

While all tasks of type [Test](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html) are automatically enhanced to provide coverage information when the java plugin has been applied, any task that implements [JavaForkOptions](http://docs.google.com/javadoc/org/gradle/process/JavaForkOptions.html) can be enhanced by the JaCoCo plugin. That is, any task that forks Java processes can be used to generate coverage information.

For example you can configure your build to generate code coverage using the application plugin.

[Example: Using application plugin to generate code coverage data](#4hi7mgh)

**build.gradle**

apply plugin: "application"  
apply plugin: "jacoco"  
  
mainClassName = "org.gradle.MyMain"  
  
jacoco {  
 applyTo run  
}  
  
task applicationCodeCoverageReport(type:JacocoReport){  
 executionData run  
 sourceSets sourceSets.main  
}

| **Note** | The code for this example can be found at samples/testing/jacoco/application in the ‘-all’ distribution of Gradle. |
| --- | --- |

**Coverage reports generated by applicationCodeCoverageReport**

.  
└── build  
 ├── jacoco  
 │   └── run.exec  
 └── reports  
 └── jacoco  
 └── applicationCodeCoverageReport  
 └── html  
 └── index.html

[Tasks](#2wnhwoa)

For projects that also apply the Java Plugin, the JaCoCo plugin automatically adds the following tasks:

jacocoTestReport — [JacocoReport](http://docs.google.com/dsl/org.gradle.testing.jacoco.tasks.JacocoReport.html)

Generates code coverage report for the test task.

jacocoTestCoverageVerification — [JacocoCoverageVerification](http://docs.google.com/dsl/org.gradle.testing.jacoco.tasks.JacocoCoverageVerification.html)

Verifies code coverage metrics based on specified rules for the test task.

[Dependency management](#1bss6w3)

The JaCoCo plugin adds the following dependency configurations:

Table 27. JaCoCo plugin - dependency configurations

| **Name** | **Meaning** |
| --- | --- |
| jacocoAnt | The JaCoCo Ant library used for running the JacocoReport, JacocoMerge and JacocoCoverageVerification tasks. |
| jacocoAgent | The JaCoCo agent library used for instrumenting the code under test. |

[The JDepend Plugin](#3mzq4wv)

The JDepend plugin performs quality checks on your project’s source files using [JDepend](http://clarkware.com/software/JDepend.html) and generates reports from these checks.

[Usage](#3vsfpjw)

To use the JDepend plugin, include the following in your build script:

[Example: Using the JDepend plugin](#2axpzrp)

**build.gradle**

apply plugin: 'jdepend'

The plugin adds a number of tasks to the project that perform the quality checks. You can execute the checks by running gradle check.

Note that JDepend will run with the same Java version used to run Gradle.

[Tasks](#q309zi)

The JDepend plugin adds the following tasks to the project:

jdependMain — [JDepend](http://docs.google.com/dsl/org.gradle.api.plugins.quality.JDepend.html)

*Depends on*: classes

Runs JDepend against the production Java source files.

jdependTest — [JDepend](http://docs.google.com/dsl/org.gradle.api.plugins.quality.JDepend.html)

*Depends on*: testClasses

Runs JDepend against the test Java source files.

jdepend*SourceSet* — [JDepend](http://docs.google.com/dsl/org.gradle.api.plugins.quality.JDepend.html)

*Depends on*: *sourceSet*Classes

Runs JDepend against the given source set’s Java source files.

The JDepend plugin adds the following dependencies to tasks defined by the Java plugin.

[Additional task dependencies](#3a2nsnb)

check

All JDepend tasks, including jdependMain and jdependTest.

[Dependency management](#1p7y2v4)

The JDepend plugin adds the following dependency configurations:

[Dependency configurations](#497llix)

jdepend

The JDepend libraries to use

[Configuration](#2ocvvqq)

See the [JDependExtension](http://docs.google.com/dsl/org.gradle.api.plugins.quality.()JDependExtension.html) class in the API documentation.

[The OSGi Plugin](#2250f4o)

The OSGi plugin provides a factory method to create an [OsgiManifest](http://docs.google.com/javadoc/org/gradle/api/plugins/osgi/OsgiManifest.html) object. OsgiManifest extends [Manifest](http://docs.google.com/javadoc/org/gradle/api/java/archives/Manifest.html). To learn more about generic manifest handling, see [more about Java manifests](#si57gw). If the Java plugins is applied, the OSGi plugin replaces the manifest object of the default jar with an OsgiManifest object. The replaced manifest is merged into the new one.

| **Note** | The OSGi plugin makes heavy use of the [BND tool](http://bnd.bndtools.org/). A separate [plugin implementation](https://github.com/bndtools/bnd/blob/master/biz.aQute.bnd.gradle/README.md) is maintained by the BND authors that has more advanced features. |
| --- | --- |

[Usage](#13i65yj)

To use the OSGi plugin, include the following in your build script:

[Example: Using the OSGi plugin](#3nhtomc)

**build.gradle**

apply plugin: 'osgi'

[Implicitly applied plugins](#22n3yu5)

Applies the Java base plugin.

[Tasks](#hse91y)

The OSGi plugin adds the following tasks to the project:

osgiClasses — [Sync](http://docs.google.com/dsl/org.gradle.api.tasks.Sync.html)

*Depends on*: classes

Copies all classes from the main source set to a single directory that is processed by BND.

[Convention object](#31s1rpr)

The OSGi plugin adds the following convention object: [OsgiPluginConvention](http://docs.google.com/javadoc/org/gradle/api/plugins/osgi/OsgiPluginConvention.html)

[Convention properties](#1gxc1xk)

The OSGi plugin does not add any convention properties to the project.

[Convention methods](#40wzkld)

The OSGi plugin adds the following methods. For more details, see the API documentation of the convention object.

Table 28. OSGi methods

| **Method** | **Return Type** | **Description** |
| --- | --- | --- |
| osgiManifest() | [OsgiManifest](http://docs.google.com/javadoc/org/gradle/api/plugins/osgi/OsgiManifest.html) | Returns an OsgiManifest object. |
| osgiManifest(Closure cl) | [OsgiManifest](http://docs.google.com/javadoc/org/gradle/api/plugins/osgi/OsgiManifest.html) | Returns an OsgiManifest object configured by the closure. |

The classes in the classes dir are analyzed regarding their package dependencies and the packages they expose. Based on this the *Import-Package* and the *Export-Package* values of the OSGi Manifest are calculated. If the classpath contains jars with an OSGi bundle, the bundle information is used to specify version information for the *Import-Package* value. Beside the explicit properties of the OsgiManifest object you can add instructions.

[Example: Configuration of OSGi MANIFEST.MF file](#2g29ut6)

**build.gradle**

jar {  
 manifest { *// the manifest of the default jar is of type OsgiManifest*  
 name = 'overwrittenSpecialOsgiName'  
 instruction 'Private-Package',  
 'org.mycomp.package1',  
 'org.mycomp.package2'  
 instruction 'Bundle-Vendor', 'MyCompany'  
 instruction 'Bundle-Description', 'Platform2: Metrics 2 Measures Framework'  
 instruction 'Bundle-DocURL', 'http://www.mycompany.com'  
 }  
}  
task fooJar(type: Jar) {  
 manifest = osgiManifest {  
 instruction 'Bundle-Vendor', 'MyCompany'  
 }  
}

The first argument of the instruction call is the key of the property. The other arguments form the value. To learn more about the available instructions have a look at the [BND tool](http://bnd.bndtools.org/).

[The PMD Plugin](#haapch)

The PMD plugin performs quality checks on your project’s Java source files using [PMD](http://pmd.sourceforge.net) and generates reports from these checks.

[Usage](#v7k50z)

To use the PMD plugin, include the following in your build script:

[Example: Using the PMD plugin](#3f77nos)

**build.gradle**

apply plugin: 'pmd'

The plugin adds a number of tasks to the project that perform the quality checks. You can execute the checks by running gradle check.

Note that PMD will run with the same Java version used to run Gradle.

[Tasks](#1uchxwl)

The PMD plugin adds the following tasks to the project:

pmdMain — [Pmd](http://docs.google.com/dsl/org.gradle.api.plugins.quality.Pmd.html)

Runs PMD against the production Java source files.

pmdTest — [Pmd](http://docs.google.com/dsl/org.gradle.api.plugins.quality.Pmd.html)

Runs PMD against the test Java source files.

pmd*SourceSet* — [Pmd](http://docs.google.com/dsl/org.gradle.api.plugins.quality.Pmd.html)

Runs PMD against the given source set’s Java source files.

The PMD plugin adds the following dependencies to tasks defined by the Java plugin.

Table 29. PMD plugin - additional task dependencies

| **Task name** | **Depends on** |
| --- | --- |
| check | All PMD tasks, including pmdMain and pmdTest. |

[Dependency management](#4ec5gke)

The PMD plugin adds the following dependency configurations:

Table 30. PMD plugin - dependency configurations

| **Name** | **Meaning** |
| --- | --- |
| pmd | The PMD libraries to use |

[Configuration](#2thfqs7)

See the [PmdExtension](http://docs.google.com/dsl/org.gradle.api.plugins.quality.PmdExtension.html) class in the API documentation.

[Java Web Projects](#319y80a)

[The Ear Plugin](#1gf8i83)

The Ear plugin adds support for assembling web application EAR files. It adds a default EAR archive task. It doesn’t require the Java plugin, but for projects that also use the Java plugin it disables the default JAR archive generation.

[Usage](#18mq100)

To use the Ear plugin, include the following in your build script:

[Example: Using the Ear plugin](#3smdjnt)

**build.gradle**

apply plugin: 'ear'

[Tasks](#27rntvm)

The Ear plugin adds the following tasks to the project.

ear — [Ear](http://docs.google.com/dsl/org.gradle.plugins.ear.Ear.html)

*Depends on*: compile (only if the Java plugin is also applied)

Assembles the application EAR file.

[Dependencies added to other tasks](#mwy43f)

The Ear plugin adds the following dependencies to tasks added by the [Base Plugin](#vx1227).

assemble

*Depends on*: ear.

[Project layout](#36wlmr8)

.  
└── src  
 └── main  
 └── application (1)

1. Ear resources, such as a META-INF directory

[Dependency management](#1m1vwz1)

The Ear plugin adds two dependency configurations: deploy and earlib. All dependencies in the deploy configuration are placed in the root of the EAR archive, and are *not* transitive. All dependencies in the earlib configuration are placed in the 'lib' directory in the EAR archive and *are* transitive.

[Convention properties](#461jfmu)

appDirName — String

The name of the application source directory, relative to the project directory. *Default value: `src/main/application`*.

libDirName — String

The name of the lib directory inside the generated EAR. *Default value: `lib`*.

deploymentDescriptor — [DeploymentDescriptor](http://docs.google.com/javadoc/org/gradle/plugins/ear/descriptor/DeploymentDescriptor.html)

Metadata to generate a deployment descriptor file, e.g. application.xml. *Default value: A deployment descriptor with sensible defaults named application.xml`*. If this file already exists in the `appDirName/META-INF then the existing file contents will be used and the explicit configuration in the ear.deploymentDescriptor will be ignored.

These properties are provided by a [EarPluginConvention](http://docs.google.com/dsl/org.gradle.plugins.ear.EarPluginConvention.html) convention object.

[Ear](#2l6tpun)

The default behavior of the Ear task is to copy the content of src/main/application to the root of the archive. If your application directory doesn’t contain a META-INF/application.xml deployment descriptor then one will be generated for you.

The [Ear](http://docs.google.com/dsl/org.gradle.plugins.ear.Ear.html) class in the API documentation has additional useful information.

[Customizing](#10c402g)

Here is an example with the most important customization options:

[Example: Customization of ear plugin](#3kbriq9)

**build.gradle**

apply plugin: 'ear'  
apply plugin: 'java'  
  
repositories { mavenCentral() }  
  
dependencies {  
 *// The following dependencies will be the ear modules and*  
 *// will be placed in the ear root*  
 deploy project(path: ':war', configuration: 'archives')  
  
 *// The following dependencies will become ear libs and will*  
 *// be placed in a dir configured via the libDirName property*  
 earlib group: 'log4j', name: 'log4j', version: '1.2.15', ext: 'jar'  
}  
  
ear {  
 appDirName 'src/main/app' *// use application metadata found in this folder*  
 *// put dependent libraries into APP-INF/lib inside the generated EAR*  
 libDirName 'APP-INF/lib'  
 deploymentDescriptor { *// custom entries for application.xml:*  
*// fileName = "application.xml" // same as the default value*  
*// version = "6" // same as the default value*  
 applicationName = "customear"  
 initializeInOrder = true  
 displayName = "Custom Ear" *// defaults to project.name*  
 *// defaults to project.description if not set*  
 description = "My customized EAR for the Gradle documentation"  
*// libraryDirectory = "APP-INF/lib" // not needed, above libDirName setting does this*  
*// module("my.jar", "java") // won't deploy as my.jar isn't deploy dependency*  
*// webModule("my.war", "/") // won't deploy as my.war isn't deploy dependency*  
 securityRole "admin"  
 securityRole "superadmin"  
 withXml { provider -> *// add a custom node to the XML*  
 provider.asNode().appendNode("data-source", "my/data/source")  
 }  
 }  
}

You can also use customization options that the [Ear](http://docs.google.com/dsl/org.gradle.plugins.ear.Ear.html) task provides, such as from and metaInf.

[Using custom descriptor file](#1zh1sy2)

You may already have appropriate settings in a application.xml file and want to use that instead of configuring the ear.deploymentDescriptor section of the build script. To accommodate that goal, place the META-INF/application.xml in the right place inside your source folders (see the appDirName property). The file contents will be used and the explicit configuration in the ear.deploymentDescriptor will be ignored.

[Building Play applications](#40ew0vw)

| **Note** | Support for building Play applications is currently [incubating](#2xcytpi). Please be aware that the DSL, APIs and other configuration may change in later Gradle versions. |
| --- | --- |

[Play](https://www.playframework.com/) is a modern web application framework. The Play plugin adds support for building, testing and running Play applications with Gradle.

The Play plugin makes use of the Gradle [software model](#3q5sasy).

[Usage](#4jgpblv)

To use the Play plugin, include the following in your build script to apply the play plugin and add the Lightbend repositories:

[Example: Using the Play plugin](#2ylzlto)

**build.gradle**

plugins {  
 id 'play'  
}  
  
repositories {  
 jcenter()  
 maven {  
 name "lightbend-maven-release"  
 url "https://repo.lightbend.com/lightbend/maven-releases"  
 }  
 ivy {  
 name "lightbend-ivy-release"  
 url "https://repo.lightbend.com/lightbend/ivy-releases"  
 layout "ivy"  
 }  
}

Note that defining the Lightbend repositories is necessary. In future versions of Gradle, this will be replaced with a more convenient syntax.

[Limitations](#1dr9w1h)

The Play plugin currently has a few limitations.

* Gradle does not yet support aggregate reverse routes introduced in Play 2.4.x.
* A given project may only define a single Play application. This means that a single project cannot build more than one Play application. However, a multi-project build can have many projects that each define their own Play application.
* Play applications can only target a single “platform” (combination of Play, Scala and Java version) at a time. This means that it is currently not possible to define multiple variants of a Play application that, for example, produce jars for both Scala 2.10 and 2.11. This limitation may be lifted in future Gradle versions.
* Support for generating IDE configurations for Play applications is limited to [IDEA](#2szc72q).

[Software Model](#3xqxepa)

The Play plugin uses a *software model* to describe a Play application and how to build it. The Play software model extends the base Gradle [software model](#1664s55) to add support for building Play applications. A Play application is represented by a [PlayApplicationSpec](http://docs.google.com/javadoc/org/gradle/play/PlayApplicationSpec.html) component type. The plugin automatically creates a single [PlayApplicationBinarySpec](http://docs.google.com/javadoc/org/gradle/play/PlayApplicationBinarySpec.html) instance when it is applied. Additional Play components cannot be added to a project.



*Figure 30. Play plugin - software model*

[The Play application component](#2cw7ox3)

A Play application component describes the application to be built and consists of several configuration elements. One type of element that describes the application are the source sets that define where the application controller, route, template and model class source files should be found. These source sets are logical groupings of files of a particular type and a default source set for each type is created when the play plugin is applied.

Table 31. Default Play source sets

| **Source Set** | **Type** | **Directory** | **Filters** |
| --- | --- | --- | --- |
| java | [JavaSourceSet](http://docs.google.com/javadoc/org/gradle/language/java/JavaSourceSet.html) | app | \*\*/\*.java |
| scala | [ScalaLanguageSourceSet](http://docs.google.com/javadoc/org/gradle/language/scala/ScalaLanguageSourceSet.html) | app | \*\*/\*.scala |
| routes | [RoutesSourceSet](http://docs.google.com/javadoc/org/gradle/language/routes/RoutesSourceSet.html) | conf | routes, \*.routes |
| twirlTemplates | [TwirlSourceSet](http://docs.google.com/javadoc/org/gradle/language/twirl/TwirlSourceSet.html) | app | \*\*/\*.scala.\* |
| javaScript | [JavaScriptSourceSet](http://docs.google.com/javadoc/org/gradle/language/javascript/JavaScriptSourceSet.html) | app/assets | \*\*/\*.js |

These source sets can be configured or additional source sets can be added to the Play component. See [Configuring Play](#s1hz4w) for further information.

Another element of configuring a Play application is the *platform*. To build a Play application, Gradle needs to understand which versions of Play, Scala and Java to use. The Play component specifies this requirement as a [PlayPlatform](http://docs.google.com/javadoc/org/gradle/play/platform/PlayPlatform.html). If these values are not configured, a default version of Play, Scala and Java will be used. See [Targeting a certain version of Play](#3c15hsp) for information on configuring the Play platform.

Note that only a single platform can be specified for a given Play component. This means that only a single version of Play, Scala and Java can be used to build a Play component. In other words, a Play component can only produce one set of outputs, and those outputs will be built using the versions specified by the platform configured on the component.

[The Play application binary](#1r6fs0i)

A Play application component is compiled and packaged to produce a set of outputs which are represented by a [PlayApplicationBinarySpec](http://docs.google.com/javadoc/org/gradle/play/PlayApplicationBinarySpec.html). The Play binary specifies the jar files produced by building the component as well as providing elements by which additional content can be added to those jar files. It also exposes the tasks involved in building the component and creating the binary.

See [Configuring Play](#s1hz4w) for examples of configuring the Play binary.

[Project Layout](#4b63aob)

The Play plugin follows the typical Play application layout. You can [configure source sets](#2qbdkw4) to include additional directories or change the defaults.

├── app → Application source code.  
│   ├── assets → Assets that require compilation.  
│   │   └── javascripts → JavaScript source code to be minified.  
│   ├── controllers → Application controller source code.  
│   ├── models → Application business source code.  
│   └── views → Application UI templates.  
├── build.gradle → Your project's build script.  
├── conf → Main application configuration file and routes files.  
├── public → Public assets.  
│   ├── images → Application image files.  
│   ├── javascripts → Typically JavaScript source code.  
│   └── stylesheets → Typically CSS source code.  
└── test → Test source code.

[Tasks](#15gnv3x)

The Play plugin hooks into the normal Gradle lifecycle tasks such as assemble, check and build, but it also adds several additional tasks which form the lifecycle of a Play project:

[Play Plugin — lifecycle tasks](#3pgbdrq)

playBinary — [Task](http://docs.google.com/dsl/org.gradle.api.Task.html)

*Depends on*: All compile tasks for source sets added to the Play application.

Performs a build of just the Play application.

dist — [Task](http://docs.google.com/dsl/org.gradle.api.Task.html)

*Depends on*: createPlayBinaryZipDist, createPlayBinaryTarDist

Assembles the Play distribution.

stage — [Task](http://docs.google.com/dsl/org.gradle.api.Task.html)

*Depends on*: stagePlayBinaryDist

Stages the Play distribution.

The plugin also provides tasks for running, testing and packaging your Play application:

[Play Plugin — running and testing tasks](#24llnzj)

runPlayBinary — [PlayRun](http://docs.google.com/dsl/org.gradle.play.tasks.PlayRun.html)

*Depends on*: playBinary to build Play application.

Runs the Play application for local development. See [how this works with continuous build.](#jqvy7c)

testPlayBinary — [Test](http://docs.google.com/dsl/org.gradle.api.tasks.testing.Test.html)

*Depends on*: playBinary to build Play application and compilePlayBinaryTests.

Runs JUnit/TestNG tests for the Play application.

For the different types of sources in a Play application, the plugin adds the following compilation tasks:

[Play Plugin — source set tasks](#33qjgv5)

compilePlayBinaryScala — [PlatformScalaCompile](http://docs.google.com/javadoc/org/gradle/language/scala/tasks/PlatformScalaCompile.html)

*Depends on*: Scala and Java

Compiles all Scala and Java sources defined by the Play application.

compilePlayBinaryPlayTwirlTemplates — [TwirlCompile](http://docs.google.com/dsl/org.gradle.play.tasks.TwirlCompile.html)

*Depends on*: Twirl templates

Compiles Twirl templates with the Twirl compiler. Gradle supports all of the built-in Twirl template formats (HTML, XML, TXT and JavaScript). Twirl templates need to match the pattern \*.scala.\*.

compilePlayBinaryPlayRoutes — [RoutesCompile](http://docs.google.com/dsl/org.gradle.play.tasks.RoutesCompile.html)

*Depends on*: Play Route files

Compiles routes files into Scala sources.

minifyPlayBinaryJavaScript — [JavaScriptMinify](http://docs.google.com/dsl/org.gradle.play.tasks.JavaScriptMinify.html)

*Depends on*: JavaScript files

Minifies JavaScript files with the Google Closure compiler.

[Finding out more about your project](#1ivtr2y)

Gradle provides a report that you can run from the command-line that shows some details about the components and binaries that your project produces. To use this report, just run gradle components. Below is an example of running this report for one of the sample projects:

[Example: The components report](#42vh9qr)

**Output of** gradle components

> gradle components  
  
> Task :components  
  
------------------------------------------------------------  
Root project  
------------------------------------------------------------  
  
Play Application 'play'  
-----------------------  
  
Source sets  
 Java source 'play:java'  
 srcDir: app  
 includes: \*\*/\*.java  
 JavaScript source 'play:javaScript'  
 srcDir: app/assets  
 includes: \*\*/\*.js  
 JVM resources 'play:resources'  
 srcDir: conf  
 Routes source 'play:routes'  
 srcDir: conf  
 includes: routes, \*.routes  
 Scala source 'play:scala'  
 srcDir: app  
 includes: \*\*/\*.scala  
 Twirl template source 'play:twirlTemplates'  
 srcDir: app  
 includes: \*\*/\*.scala.\*  
  
Binaries  
 Play Application Jar 'play:binary'  
 build using task: :playBinary  
 target platform: Play Platform (Play 2.6.15, Scala: 2.12, Java: Java SE 8)  
 toolchain: Default Play Toolchain  
 classes dir: build/playBinary/classes  
 resources dir: build/playBinary/resources  
 JAR file: build/playBinary/lib/basic.jar  
  
Note: currently not all plugins register their components, so some components may not be visible here.  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

[Running a Play application](#jqvy7c)

The runPlayBinary task starts the Play application under development. During development it is beneficial to execute this task as a [continuous build](#2olpkfy). Continuous build is a generic feature that supports automatically re-running a build when inputs change. The runPlayBinary task is “continuous build aware” in that it behaves differently when run as part of a continuous build.

When not run as part of a continuous build, the runPlayBinary task will *block* the build. That is, the task will not complete as long as the application is running. When running as part of a continuous build, the task will start the application if not running and otherwise propagate any changes to the code of the application to the running instance. This is useful for quickly iterating on your Play application with an edit->rebuild->refresh cycle. Changes to your application will not take affect until the end of the overall build.

To enable continuous build, run Gradle with -t runPlayBinary or --continuous runPlayBinary.

Users of Play used to such a workflow with Play’s default build system should note that compile errors are handled differently. If a build failure occurs during a continuous build, the Play application will not be reloaded. Instead, you will be presented with an exception message. The exception message will only contain the overall cause of the build failure. More detailed information will only be available from the console.

[Configuring a Play application](#s1hz4w)

[Targeting a certain version of Play](#3c15hsp)

By default, Gradle uses Play 2.6.15, Scala 2.12 and the version of Java used to start the build. A Play application can select a different version by specifying a target [PlayApplicationSpec.platform(java.lang.Object)](http://docs.google.com/dsl/org.gradle.play.PlayApplicationSpec.html#org.gradle.play.PlayApplicationSpec:platform(java.lang.Object)) on the Play application component.

[Example: Selecting a version of the Play Framework](#2i0rjyk)

**build.gradle**

model {  
 components {  
 play {  
 platform play: '2.6.15', scala: '2.12', java: '1.8'  
 injectedRoutesGenerator = true  
 }  
 }  
}

The following versions of Play and Scala are supported:

Table 32. Play supported versions

| **Play** | **Scala** | **Java** |
| --- | --- | --- |
| 2.6.x | 2.11 and 2.12 | 1.8 |
| 2.5.x | 2.11 | 1.8 |
| 2.4.x | 2.10 and 2.11 | 1.8 |
| 2.3.x | 2.10 and 2.11 | 1.6, 1.7 and 1.8 |

[Adding dependencies](#x61u6d)

You can add compile, test and runtime dependencies to a Play application through [Configuration](http://docs.google.com/dsl/org.gradle.api.artifacts.Configuration.html) created by the Play plugin.

If you are coming from SBT, the Play SBT plugin provides short names for common dependencies. For instance, if your project has a dependency on ws, you will need to add a dependency to com.typesafe.play:play-ws\_2.11:2.3.9 where 2.11 is your Scala version and 2.3.9 is your Play framework version.

Other dependencies that have short names, such as jacksons may actually be multiple dependencies. For those dependencies, you will need to work out the dependency coordinates from a dependency report.

* play is used for compile time dependencies.
* playTest is used for test compile time dependencies.
* playRun is used for run time dependencies.

[Example: Adding dependencies to a Play application](#3h5pcu6)

**build.gradle**

dependencies {  
 play "commons-lang:commons-lang:2.6"  
 play "com.typesafe.play:play-guice\_2.12:2.6.15"  
 play "ch.qos.logback:logback-classic:1.2.3"  
}

Play 2.6 has a more modular architecture and, because of that, you may need to add some dependencies manually. For example, [Guice support was moved to a separated module](https://playframework.com/documentation/2.6.x/Migration26#Guice-DI-support-moved-to-separate-module). Considering the following definition for a Play 2.6 project:

[Example: A Play 2.6 project](#1wazn1z)

**build.gradle**

model {  
 components {  
 play {  
 platform play: '2.6.7', scala: '2.12', java: '1.8'  
 injectedRoutesGenerator = true  
 }  
 }  
}

You can add Guice dependency like:

[Example: Adding Guice dependency in Play 2.6 project](#4gan5ps)

**build.gradle**

dependencies {  
 play "com.typesafe.play:play-guice\_2.12:2.6.7"  
}

Of course, pay attention to keep the Play version and Scala version for the dependency consistent with the platform versions.

[Configuring the default source sets](#2qbdkw4)

You can further configure the default source sets to do things like add new directories, add filters, etc.

[Example: Configuring extra source sets to a Play application](#2vfxfxl)

**build.gradle**

model {  
 components {  
 play {  
 sources {  
 java {  
 source.srcDir "additional/java"  
 }  
 javaScript {  
 source {  
 srcDir "additional/javascript"  
 exclude "\*\*/old\_\*.js"  
 }  
 }  
 }  
 }  
 }  
}

[Adding extra source sets](#1al7q5e)

If your Play application has additional sources that exist in non-standard directories, you can add extra source sets that Gradle will automatically add to the appropriate compile tasks.

[Example: Adding extra source sets to a Play application](#3ukv8t7)

**build.gradle**

model {  
 components {  
 play {  
 sources {  
 extraJava(JavaSourceSet) {  
 source.srcDir "extra/java"  
 }  
 extraTwirl(TwirlSourceSet) {  
 source.srcDir "extra/twirl"  
 }  
 extraRoutes(RoutesSourceSet) {  
 source.srcDir "extra/routes"  
 }  
 }  
 }  
 }  
}

[Configuring compiler options](#29q5j10)

If your Play application requires additional Scala compiler flags, you can add these arguments directly to the Scala compiler task.

[Example: Configuring Scala compiler options](#ovft8t)

**build.gradle**

model {  
 components {  
 play {  
 binaries.all {  
 tasks.withType(PlatformScalaCompile) {  
 scalaCompileOptions.additionalParameters = ["-feature", "-language:implicitConversions"]  
 }  
 }  
 }  
 }  
}

[Configuring routes style](#38v3bwm)

| **Note** | The injected router is only supported in Play Framework 2.4 or better. |
| --- | --- |

If your Play application’s router uses dependency injection to access your controllers, you’ll need to configure your application to *not* use the default static router. Under the covers, the Play plugin is using the InjectedRoutesGenerator instead of the default StaticRoutesGenerator to generate the router classes.

[Example: Configuring routes style](#1o0dm4f)

**build.gradle**

model {  
 components {  
 play {  
 injectedRoutesGenerator = true  
 }  
 }  
}

[Configuring Twirl templates](#48014s8)

A custom Twirl template format can be configured independently for each Twirl source set. See the [TwirlSourceSet](http://docs.google.com/javadoc/org/gradle/language/twirl/TwirlSourceSet.html) for an example.

[Injecting a custom asset pipeline](#2n5bf01)

Gradle Play support comes with a simplistic asset processing pipeline that minifies JavaScript assets. However, many organizations have their own custom pipeline for processing assets. You can easily hook the results of your pipeline into the Play binary by utilizing the [PublicAssets](http://docs.google.com/javadoc/org/gradle/play/PublicAssets.html) property on the binary.

[Example: Configuring a custom asset pipeline](#12alp7u)

**build.gradle**

model {  
 components {  
 play {  
 binaries.all { binary ->  
 tasks.create("addCopyrightToPlay${binary.name.capitalize()}Assets", AddCopyrights) { copyrightTask ->  
 source "raw-assets"  
 copyrightFile = project.file('copyright.txt')  
 destinationDir = project.file("${buildDir}/play${binary.name.capitalize()}/addCopyRights")  
  
 *// Hook this task into the binary*  
 binary.assets.addAssetDir destinationDir  
 binary.assets.builtBy copyrightTask  
 }  
 }  
 }  
 }  
}  
  
**class** **AddCopyrights** **extends** SourceTask {  
 @InputFile  
 File copyrightFile  
  
 @OutputDirectory  
 File destinationDir  
  
 @TaskAction  
 **void** generateAssets() {  
 String copyright = copyrightFile.text  
 getSource().each { File file ->  
 File outputFile = **new** File(destinationDir, file.name)  
 outputFile.text = "${copyright}\n${file.text}"  
 }  
 }  
}

[Multi-project Play applications](#3ma97vn)

Play applications can be built in multi-project builds as well. Simply apply the play plugin in the appropriate subprojects and create any project dependencies on the play configuration.

[Example: Configuring dependencies on Play subprojects](#21fji3g)

**build.gradle**

dependencies {  
 play project(":admin")  
 play project(":user")  
 play project(":util")  
}

See the play/multiproject sample provided in the Gradle distribution for a working example.

[Packaging a Play application for distribution](#gktsb9)

Gradle provides the capability to package your Play application so that it can easily be distributed and run in a target environment. The distribution package (zip file) contains the Play binary jars, all dependencies, and generated scripts that set up the classpath and run the application in a Play-specific [Netty](http://netty.io) container.

The distribution can be created by running the dist lifecycle task and places the distribution in the $buildDir/distributions directory. Alternatively, one can validate the contents by running the stage lifecycle task which copies the files to the $buildDir/stage directory using the layout of the distribution package.

[Play Plugin — distribution tasks](#30khaz2)

createPlayBinaryStartScripts — [CreateStartScripts](http://docs.google.com/javadoc/org/gradle/api/tasks/application/CreateStartScripts.html)

Generates scripts to run the Play application distribution.

stagePlayBinaryDist — [Copy](http://docs.google.com/dsl/org.gradle.api.tasks.Copy.html)

*Depends on*: playBinary, createPlayBinaryStartScripts

Copies all jar files, dependencies and scripts into a staging directory.

createPlayBinaryZipDist — [Zip](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Zip.html)

Bundles the Play application as a standalone distribution packaged as a zip.

createPlayBinaryTarDist — [Tar](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.Tar.html)

Bundles the Play application as a standalone distribution packaged as a tar.

stage — [Task](http://docs.google.com/dsl/org.gradle.api.Task.html)

*Depends on*: stagePlayBinaryDist

Lifecycle task for staging a Play distribution.

dist — [Task](http://docs.google.com/dsl/org.gradle.api.Task.html)

*Depends on*: createPlayBinaryZipDist, createPlayBinaryTarDist

Lifecycle task for creating a Play distribution.

[Adding additional files to your Play application distribution](#1fprl6v)

You can add additional files to the distribution package using the [Distribution](http://docs.google.com/javadoc/org/gradle/api/distribution/Distribution.html) API.

[Example: Add extra files to a Play application distribution](#3zpf3uo)

**build.gradle**

model {  
 distributions {  
 playBinary {  
 contents {  
 from("README.md")  
 from("scripts") {  
 into "bin"  
 }  
 }  
 }  
 }  
}

[Building a Play application with an IDE](#2eupe2h)

If you want to generate IDE metadata configuration for your Play project, you need to apply the appropriate IDE plugin. Gradle supports generating IDE metadata for IDEA only for Play projects at this time.

To generate IDEA’s metadata, apply the idea plugin along with the play plugin.

[Example: Applying both the Play and IDEA plugins](#tzzoaa)

**build.gradle**

plugins {  
 id 'play'  
 id 'idea'  
}

Source code generated by routes and Twirl templates cannot be generated by IDEA directly, so changes made to those files will not affect compilation until the next Gradle build. You can run the Play application with Gradle in [continuous build](#2olpkfy) to automatically rebuild and reload the application whenever something changes.

[Resources](#3dzn6y3)

For additional information about developing Play applications:

* Play types in the Gradle DSL Guide:
  + [PlayApplicationBinarySpec](http://docs.google.com/javadoc/org/gradle/play/PlayApplicationBinarySpec.html)
  + [PlayApplicationSpec](http://docs.google.com/javadoc/org/gradle/play/PlayApplicationSpec.html)
  + [PlayPlatform](http://docs.google.com/javadoc/org/gradle/play/platform/PlayPlatform.html)
  + [JvmClasses](http://docs.google.com/javadoc/org/gradle/play/JvmClasses.html)
  + [PublicAssets](http://docs.google.com/javadoc/org/gradle/play/PublicAssets.html)
  + [PlayDistributionContainer](http://docs.google.com/javadoc/org/gradle/play/distribution/PlayDistributionContainer.html)
  + [JavaScriptMinify](http://docs.google.com/dsl/org.gradle.play.tasks.JavaScriptMinify.html)
  + [PlayRun](http://docs.google.com/dsl/org.gradle.play.tasks.PlayRun.html)
  + [RoutesCompile](http://docs.google.com/dsl/org.gradle.play.tasks.RoutesCompile.html)
  + [TwirlCompile](http://docs.google.com/dsl/org.gradle.play.tasks.TwirlCompile.html)
* [Play Framework Documentation](https://www.playframework.com/documentation).

[The War Plugin](#2fk6b3p)

The War plugin extends the Java plugin to add support for assembling web application WAR files. It disables the default JAR archive generation of the Java plugin and adds a default WAR archive task.

[Usage](#1t4xh5w)

To use the War plugin, include the following in your build script:

[Example: Using the War plugin](#4d4kztp)

**build.gradle**

plugins {  
 id 'war'  
}

[Project layout](#2s9va1i)

In addition to the [standard Java project layout](#1c9ff83), the War Plugin adds:

src/main/webapp

Web application sources

[Tasks](#17f5k9b)

The War plugin adds and modifies the following tasks:

war — [War](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.War.html)

*Depends on*: compile

Assembles the application WAR file.

assemble - *lifecycle task*

*Depends on*: war

The War plugin adds the following dependencies to tasks added by the Java plugin;



*Figure 31. War plugin - tasks*

[Dependency management](#3ret2x4)

The War plugin adds two dependency configurations:

* providedCompile
* providedRuntime

These two configurations have the same scope as the respective compile and runtime configurations, except that they are not added to the WAR archive.

It is important to note that these provided configurations work transitively. Let’s say you add commons-httpclient:commons-httpclient:3.0 to any of the provided configurations. This dependency has a dependency on commons-codec. Because this is a “provided” configuration, this means that neither of these dependencies will be added to your WAR, even if the commons-codec library is an explicit dependency of your compile configuration. If you don’t want this transitive behavior, simply declare your provided dependencies like commons-httpclient:commons-httpclient:3.0@jar.

[Publishing](#26k3d4x)

components.web

A [SoftwareComponent](http://docs.google.com/javadoc/org/gradle/api/component/SoftwareComponent.html) for [publishing](#2zbgiuw) the production WAR created by the war task.

[Convention properties](#lpdncq)

webAppDirName — String

*Default value*: src/main/webapp

The name of the web application source directory, relative to the project directory.

webAppDir — (read-only) File

*Default value*: $webAppDirName, e.g. *src/main/webapp*

The path to the web application source directory.

These properties are provided by a [WarPluginConvention](http://docs.google.com/dsl/org.gradle.api.plugins.WarPluginConvention.html) object.

[War](#35p160j)

The default behavior of the War task is to copy the content of src/main/webapp to the root of the archive. Your webapp directory may of course contain a WEB-INF sub-directory, which may contain a web.xml file. Your compiled classes are compiled to WEB-INF/classes. All the dependencies of the runtime [[16](#1kubg8c)] configuration are copied to WEB-INF/lib.

The [War](http://docs.google.com/dsl/org.gradle.api.tasks.bundling.War.html) class in the API documentation has additional useful information.

[Customizing](#44tyyw5)

Here is an example with the most important customization options:

[Example: Customization of war plugin](#2jz993y)

**build.gradle**

configurations {  
 moreLibs  
}  
  
repositories {  
 flatDir { dirs "lib" }  
 jcenter()  
}  
  
dependencies {  
 compile module(":compile:1.0") {  
 dependency ":compile-transitive-1.0@jar"  
 dependency ":providedCompile-transitive:1.0@jar"  
 }  
 providedCompile "javax.servlet:servlet-api:2.5"  
 providedCompile module(":providedCompile:1.0") {  
 dependency ":providedCompile-transitive:1.0@jar"  
 }  
 runtime ":runtime:1.0"  
 providedRuntime ":providedRuntime:1.0@jar"  
 testCompile "junit:junit:4.12"  
 moreLibs ":otherLib:1.0"  
}  
  
war {  
 from 'src/rootContent' *// adds a file-set to the root of the archive*  
 webInf { from 'src/additionalWebInf' } *// adds a file-set to the WEB-INF dir.*  
 classpath fileTree('additionalLibs') *// adds a file-set to the WEB-INF/lib dir.*  
 classpath configurations.moreLibs *// adds a configuration to the WEB-INF/lib dir.*  
 webXml = file('src/someWeb.xml') *// copies a file to WEB-INF/web.xml*  
}

Of course one can configure the different file-sets with a closure to define excludes and includes.

[Scala Projects](#upglbi)

[The Scala Plugin](#3ep43zb)

The Scala plugin extends the Java plugin to add support for Scala projects. It can deal with Scala code, mixed Scala 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 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.

[Usage](#z4jjbr)

To use the Scala plugin, include the following in your build script:

[Example: Using the Scala plugin](#3j471zk)

**build.gradle**

apply plugin: 'scala'

[Tasks](#1y9hc7d)

The Scala plugin adds the following tasks to the project.

compileScala — [ScalaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.scala.ScalaCompile.html)

*Depends on*: compileJava

Compiles production Scala source files.

compileTestScala — [ScalaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.scala.ScalaCompile.html)

*Depends on*: compileTestJava

Compiles test Scala source files.

compile*SourceSet*Scala — [ScalaCompile](http://docs.google.com/dsl/org.gradle.api.tasks.scala.ScalaCompile.html)

*Depends on*: compile*SourceSet*Java

Compiles the given source set’s Scala source files.

scaladoc — [ScalaDoc](http://docs.google.com/dsl/org.gradle.api.tasks.scala.ScalaDoc.html)

Generates API documentation for the production Scala source files.

The Scala plugin adds the following dependencies to tasks added by the Java plugin.

Table 33. Scala plugin - additional task dependencies

| **Task name** | **Depends on** |
| --- | --- |
| classes | compileScala |
| testClasses | compileTestScala |
| *sourceSet*Classes | compile*SourceSet*Scala |



*Figure 32. Scala plugin - tasks*

[Project layout](#4i94uv6)

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](#2xef52z)

Just like the Java plugin, the Scala plugin allows you to configure custom locations for Scala production and test source files.

[Example: Custom Scala source layout](#1cjpfas)

**build.gradle**

sourceSets {  
 main {  
 scala {  
 srcDirs = ['src/scala']  
 }  
 }  
 test {  
 scala {  
 srcDirs = ['test/scala']  
 }  
 }  
}

[Dependency management](#3wjcxyl)

Scala projects need to declare a scala-library dependency. This dependency will then be used on compile and runtime class paths. It will also be used to get hold of the Scala compiler and Scaladoc tool, respectively.[[17](#2bon86e)]

If Scala is used for production code, the scala-library dependency should be added to the compile configuration:

[Example: Declaring a Scala dependency for production code](#qtxie7)

**build.gradle**

repositories {  
 mavenCentral()  
}  
  
dependencies {  
 compile 'org.scala-lang:scala-library:2.11.12'  
 testCompile 'org.scalatest:scalatest\_2.11:3.0.0'  
 testCompile 'junit:junit:4.12'  
}

If Scala is only used for test code, the scala-library dependency should be added to the testCompile configuration:

[Example: Declaring a Scala dependency for test code](#3atl120)

**build.gradle**

dependencies {  
 testCompile "org.scala-lang:scala-library:2.11.1"  
}

[Automatic configuration of scalaClasspath](#1pyvb9t)

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.

Unless a task’s scalaClasspath is configured explicitly, the Scala (base) plugin will try to infer it 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](#49yitxm)

The Scala plugin uses a configuration named zinc to resolve the [Zinc compiler](https://github.com/typesafehub/zinc) and its dependencies. Gradle will provide a default version of Zinc, but if you need to use a particular Zinc version, you can add an explicit dependency like “com.typesafe.zinc:zinc:0.3.6” to the zinc configuration. Gradle supports version 0.3.0 of Zinc and above; however, due to a regression in the Zinc compiler, versions 0.3.2 through 0.3.5.2 cannot be used.

[Example: Declaring a version of the Zinc compiler to use](#2p3t45f)

**build.gradle**

dependencies {  
 zinc 'com.typesafe.zinc:zinc:0.3.9'  
}

It is important to take care when declaring your scala-library dependency. 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 adding a compatible version of scala-library for you, but over-broad dependency resolution rules could force an incompatible version to be used instead.

For example, using configurations.all to force a particular version of scala-library would also override the version used by the Zinc compiler:

[Example: Forcing a scala-library dependency for all configurations](#1493ed8)

**build.gradle**

configurations.all {  
 resolutionStrategy.force "org.scala-lang:scala-library:2.11.12"  
}

The best way to avoid this problem is to be more selective when configuring the scala-library dependency (such as not using a configuration.all rule or using a conditional to prevent the rule from being applied to the zinc configuration). Sometimes this rule may come from a plugin or other code that you do not have control over. In such a case, you can force a correct version of the library on the zinc configuration only:

[Example: Forcing a scala-library dependency for the zinc configuration](#3o8qx11)

**build.gradle**

configurations.zinc {  
 resolutionStrategy.force "org.scala-lang:scala-library:2.10.5"  
}

You can diagnose problems with the version of the Zinc compiler selected by running [dependencyInsight](#nmf14n) for the zinc configuration.

[Convention properties](#23e178u)

The Scala plugin does not add any convention properties to the project.

[Source set properties](#ijbhgn)

The Scala plugin adds the following convention properties 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.

scala — [SourceDirectorySet](http://docs.google.com/dsl/org.gradle.api.file.SourceDirectorySet.html) (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](#115hox2). *Default value:* [*projectDir*/src/*name*/scala].

allScala — [FileTree](http://docs.google.com/javadoc/org/gradle/api/file/FileTree.html) (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 convention properties are provided by a convention object of type [ScalaSourceSet](http://docs.google.com/dsl/org.gradle.api.tasks.ScalaSourceSet.html).

The Scala plugin also modifies some source set properties:

Table 34. 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. |

[Compiling in external process](#32iz04g)

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: Adjusting memory settings](#1ho9ac9)

**build.gradle**

tasks.withType(ScalaCompile) {  
 configure(scalaCompileOptions.forkOptions) {  
 memoryMaximumSize = '1g'  
 jvmArgs = ['-XX:MaxPermSize=512m']  
 }  
}

[Incremental compilation](#41nwt02)

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](https://github.com/typesafehub/zinc), a standalone version of [sbt](https://github.com/harrah/xsbt)'s incremental Scala compiler. If you want to disable the incremental compilation, set force = true in your build file:

[Example: Forcing all code to be compiled](#2gt737v)

**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.

[Compiling and testing for Java 6 or Java 7](#vyhdfo)

The Scala compiler ignores Gradle’s targetCompatibility and sourceCompatibility settings. In Scala 2.11, the Scala compiler always compiles to Java 6 compatible bytecode. In Scala 2.12, the Scala compiler always compiles to Java 8 compatible bytecode. If you also have Java source, you can follow the same steps as for the [Java plugin](#3yo7vda) to ensure the correct Java compiler is used.

[Example: Configure Java 6 build for Scala](#3fy4w3h)

**gradle.properties**

# in $HOME/.gradle/gradle.properties  
java6Home=/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home

**build.gradle**

sourceCompatibility = 1.6  
  
**assert** hasProperty('java6Home') : "Set the property 'java6Home' in your your gradle.properties pointing to a Java 6 installation"  
**def** javaExecutablesPath = **new** File(java6Home, 'bin')  
**def** javaExecutables = [:].withDefault { execName ->  
 **def** executable = **new** File(javaExecutablesPath, execName)  
 **assert** executable.exists() : "There is no ${execName} executable in ${javaExecutablesPath}"  
 executable  
}  
  
tasks.withType(AbstractCompile) {  
 options.with {  
 fork = true  
 forkOptions.javaHome = file(java6Home)  
 }  
}  
tasks.withType(Test) {  
 executable = javaExecutables.java  
}  
tasks.withType(JavaExec) {  
 executable = javaExecutables.java  
}  
tasks.withType(Javadoc) {  
 executable = javaExecutables.javadoc  
}

[Eclipse Integration](#1v3f6ba)

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](#4f32oz3)

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](http://docs.google.com/dsl/org.gradle.plugins.ide.idea.model.IdeaModel.html).

[Example: Explicitly specify a target IntelliJ IDEA version](#2u8cz6w)

**build.gradle**

idea {  
 targetVersion = "13"  
}

[Integrating Gradle](#1tuee74)

[The Eclipse Plugins](#4du1wux)

The Eclipse plugins generate files that are used by the [Eclipse IDE](http://eclipse.org), thus making it possible to import the project into Eclipse (File - Import…​ - Existing Projects into Workspace).

The eclipse-wtp is automatically applied whenever the eclipse plugin is applied to a [War](#2fk6b3p) or [Ear](#1gf8i83) project. For utility projects (i.e. [Java](#2w5ecyt) projects used by other web projects), you need to apply the eclipse-wtp plugin explicitly.

What exactly the eclipse plugin generates depends on which other plugins are used:

Table 35. Eclipse plugin behavior

| **Plugin** | **Description** |
| --- | --- |
| None | Generates minimal .project file. |
| [Java](#2w5ecyt) | Adds Java configuration to .project. Generates .classpath and JDT settings file. |
| [Groovy](#43ky6rz) | Adds Groovy configuration to .project file. |
| [Scala](#3ep43zb) | Adds Scala support to .project and .classpath files. |
| [War](#2fk6b3p) | Adds web application support to .project file. |
| [Ear](#1gf8i83) | Adds ear application support to .project file. |

The eclipse-wtp plugin generates all WTP settings files and enhances the .project file. If a [Java](#2w5ecyt) or [War](#2fk6b3p) is applied, .classpath will be extended to get a proper packaging structure for this utility library or web application project.

Both Eclipse plugins are open to customization and provide a standardized set of hooks for adding and removing content from the generated files.

[Usage](#19dn9ep)

To use either the Eclipse or the Eclipse WTP plugin, include one of the lines in your build script:

[Example: Using the Eclipse plugin](#3tdas2i)

**build.gradle**

apply plugin: 'eclipse'

[Example: Using the Eclipse WTP plugin](#28il2ab)

**build.gradle**

apply plugin: 'eclipse-wtp'

*Note:* Internally, the eclipse-wtp plugin also applies the eclipse plugin so you don’t need to apply both.

Both Eclipse plugins add a number of tasks to your projects. The main tasks that you will use are the eclipse and cleanEclipse tasks.

[Tasks](#nnvci4)

The Eclipse plugins add the tasks shown below to a project.

[Eclipse Plugin tasks](#37niv5x)

eclipse — [Task](http://docs.google.com/dsl/org.gradle.api.Task.html)

*Depends on*: all Eclipse configuration file generation tasks

Generates all Eclipse configuration files

cleanEclipse — [Delete](http://docs.google.com/dsl/org.gradle.api.tasks.Delete.html)

*Depends on*: all Eclipse configuration file clean tasks

Removes all Eclipse configuration files

cleanEclipseProject — [Delete](http://docs.google.com/dsl/org.gradle.api.tasks.Delete.html)

Removes the .project file.

cleanEclipseClasspath — [Delete](http://docs.google.com/dsl/org.gradle.api.tasks.Delete.html)

Removes the .classpath file.

cleanEclipseJdt — [Delete](http://docs.google.com/dsl/org.gradle.api.tasks.Delete.html)

Removes the .settings/org.eclipse.jdt.core.prefs file.

eclipseProject — [GenerateEclipseProject](http://docs.google.com/dsl/org.gradle.plugins.ide.eclipse.GenerateEclipseProject.html)

Generates the .project file.

eclipseClasspath — [GenerateEclipseClasspath](http://docs.google.com/dsl/org.gradle.plugins.ide.eclipse.GenerateEclipseClasspath.html)

Generates the .classpath file.

eclipseJdt — [GenerateEclipseJdt](http://docs.google.com/dsl/org.gradle.plugins.ide.eclipse.GenerateEclipseJdt.html)

Generates the .settings/org.eclipse.jdt.core.prefs file.

[Eclipse WTP Plugin — additional tasks](#1mst5dq)

cleanEclipseWtpComponent — [Delete](http://docs.google.com/dsl/org.gradle.api.tasks.Delete.html)

Removes the .settings/org.eclipse.wst.common.component file.

cleanEclipseWtpFacet — [Delete](http://docs.google.com/dsl/org.gradle.api.tasks.Delete.html)

Removes the .settings/org.eclipse.wst.common.project.facet.core.xml file.

eclipseWtpComponent — [GenerateEclipseWtpComponent](http://docs.google.com/dsl/org.gradle.plugins.ide.eclipse.GenerateEclipseWtpComponent.html)

Generates the .settings/org.eclipse.wst.common.component file.

eclipseWtpFacet — [GenerateEclipseWtpFacet](http://docs.google.com/dsl/org.gradle.plugins.ide.eclipse.GenerateEclipseWtpFacet.html)

Generates the .settings/org.eclipse.wst.common.project.facet.core.xml file.

[Configuration](#46sgo1j)

Table 36. Configuration of the Eclipse plugins

| **Model** | **Reference name** | **Description** |
| --- | --- | --- |
| [EclipseModel](http://docs.google.com/dsl/org.gradle.plugins.ide.eclipse.model.EclipseModel.html) | eclipse | Top level element that enables configuration of the Eclipse plugin in a DSL-friendly fashion. |
| [EclipseProject](http://docs.google.com/dsl/org.gradle.plugins.ide.eclipse.model.EclipseProject.html) | eclipse.project | Allows configuring project information |
| [EclipseClasspath](http://docs.google.com/dsl/org.gradle.plugins.ide.eclipse.model.EclipseClasspath.html) | eclipse.classpath | Allows configuring classpath information. |
| [EclipseJdt](http://docs.google.com/dsl/org.gradle.plugins.ide.eclipse.model.EclipseJdt.html) | eclipse.jdt | Allows configuring jdt information (source/target Java compatibility). |
| [EclipseWtpComponent](http://docs.google.com/dsl/org.gradle.plugins.ide.eclipse.model.EclipseWtpComponent.html) | eclipse.wtp.component | Allows configuring wtp component information only if eclipse-wtp plugin was applied. |
| [EclipseWtpFacet](http://docs.google.com/dsl/org.gradle.plugins.ide.eclipse.model.EclipseWtpFacet.html) | eclipse.wtp.facet | Allows configuring wtp facet information only if eclipse-wtp plugin was applied. |

[Customizing the generated files](#2lxqy9c)

The Eclipse plugins allow you to customize the generated metadata files. The plugins provide a DSL for configuring model objects that model the Eclipse view of the project. These model objects are then merged with the existing Eclipse XML metadata to ultimately generate new metadata. The model objects provide lower level hooks for working with domain objects representing the file content before and after merging with the model configuration. They also provide a very low level hook for working directly with the raw XML for adjustment before it is persisted, for fine tuning and configuration that the Eclipse and Eclipse WTP plugins do not model.

[Merging](#11318h5)

Sections of existing Eclipse files that are also the target of generated content will be amended or overwritten, depending on the particular section. The remaining sections will be left as-is.

[Disabling merging with a complete rewrite](#3l2or4y)

To completely rewrite existing Eclipse files, execute a clean task together with its corresponding generation task, like “gradle cleanEclipse eclipse” (in that order). If you want to make this the default behavior, add “tasks.eclipse.dependsOn(cleanEclipse)” to your build script. This makes it unnecessary to execute the clean task explicitly.

This strategy can also be used for individual files that the plugins would generate. For instance, this can be done for the “.classpath” file with “gradle cleanEclipseClasspath eclipseClasspath”.

[Hooking into the generation lifecycle](#207z1cr)

The Eclipse plugins provide objects modeling the sections of the Eclipse files that are generated by Gradle. The generation lifecycle is as follows:

1. The file is read; or a default version provided by Gradle is used if it does not exist
2. The beforeMerged hook is executed with a domain object representing the existing file
3. The existing content is merged with the configuration inferred from the Gradle build or defined explicitly in the eclipse DSL
4. The whenMerged hook is executed with a domain object representing contents of the file to be persisted
5. The withXml hook is executed with a raw representation of the XML that will be persisted
6. The final XML is persisted

[Advanced configuration hooks](#4k7mk0k)

The following list covers the domain object used for each of the Eclipse model types:

[EclipseProject](http://docs.google.com/dsl/org.gradle.plugins.ide.eclipse.model.EclipseProject.html)

* beforeMerged { [Project](http://docs.google.com/javadoc/org/gradle/plugins/ide/eclipse/model/Project.html) arg -> …​ }
* whenMerged { [Project](http://docs.google.com/javadoc/org/gradle/plugins/ide/eclipse/model/Project.html) arg -> …​ }
* withXml { [XmlProvider](http://docs.google.com/javadoc/org/gradle/api/XmlProvider.html) arg -> …​ }

[EclipseClasspath](http://docs.google.com/dsl/org.gradle.plugins.ide.eclipse.model.EclipseClasspath.html)

* beforeMerged { [Classpath](http://docs.google.com/javadoc/org/gradle/plugins/ide/eclipse/model/Classpath.html) arg -> …​ }
* whenMerged { [Classpath](http://docs.google.com/javadoc/org/gradle/plugins/ide/eclipse/model/Classpath.html) arg -> …​ }
* withXml { [XmlProvider](http://docs.google.com/javadoc/org/gradle/api/XmlProvider.html) arg -> …​ }

[EclipseWtpComponent](http://docs.google.com/dsl/org.gradle.plugins.ide.eclipse.model.EclipseWtpComponent.html)

* beforeMerged { [WtpComponent](http://docs.google.com/javadoc/org/gradle/plugins/ide/eclipse/model/WtpComponent.html) arg -> …​ }
* whenMerged { [WtpComponent](http://docs.google.com/javadoc/org/gradle/plugins/ide/eclipse/model/WtpComponent.html) arg -> …​ }
* withXml { [XmlProvider](http://docs.google.com/javadoc/org/gradle/api/XmlProvider.html) arg -> …​ }

[EclipseWtpFacet](http://docs.google.com/dsl/org.gradle.plugins.ide.eclipse.model.EclipseWtpFacet.html)

* beforeMerged { [WtpFacet](http://docs.google.com/javadoc/org/gradle/plugins/ide/eclipse/model/WtpFacet.html) arg -> …​ }
* whenMerged { [WtpFacet](http://docs.google.com/javadoc/org/gradle/plugins/ide/eclipse/model/WtpFacet.html) arg -> …​ }
* withXml { [XmlProvider](http://docs.google.com/javadoc/org/gradle/api/XmlProvider.html) arg -> …​ }

[EclipseJdt](http://docs.google.com/dsl/org.gradle.plugins.ide.eclipse.model.EclipseJdt.html)

* beforeMerged { [Jdt](http://docs.google.com/javadoc/org/gradle/plugins/ide/eclipse/model/Jdt.html) arg -> …​ }
* whenMerged { [Jdt](http://docs.google.com/javadoc/org/gradle/plugins/ide/eclipse/model/Jdt.html) arg -> …​ }
* withProperties { arg -> } argument type ⇒ [java.util.Properties](http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html)

[Partial overwrite of existing content](#2zcwu8d)

A complete overwrite causes all existing content to be discarded, thereby losing any changes made directly in the IDE. Alternatively, the beforeMerged hook makes it possible to overwrite just certain parts of the existing content. The following example removes all existing dependencies from the Classpath domain object:

[Example: Partial Overwrite for Classpath](#1ei74g6)

**build.gradle**

eclipse.classpath.file {  
 beforeMerged { classpath ->  
 classpath.entries.removeAll { entry -> entry.kind == 'lib' || entry.kind == 'var' }  
 }  
}

The resulting .classpath file will only contain Gradle-generated dependency entries, but not any other dependency entries that may have been present in the original file. (In the case of dependency entries, this is also the default behavior.) Other sections of the .classpath file will be either left as-is or merged. The same could be done for the natures in the .project file:

[Example: Partial Overwrite for Project](#3yhun3z)

**build.gradle**

eclipse.project.file.beforeMerged { project ->  
 project.natures.clear()  
}

[Modifying the fully populated domain objects](#2dn4xbs)

The whenMerged hook allows to manipulate the fully populated domain objects. Often this is the preferred way to customize Eclipse files. Here is how you would export all the dependencies of an Eclipse project:

[Example: Export Classpath Entries](#ssf7jl)

**build.gradle**

eclipse.classpath.file {  
 whenMerged { classpath ->  
 classpath.entries.findAll { entry -> entry.kind == 'lib' }\*.exported = false  
 }  
}

[Modifying the XML representation](#3cs2q7e)

The withXml hook allows to manipulate the in-memory XML representation just before the file gets written to disk. Although Groovy’s XML support makes up for a lot, this approach is less convenient than manipulating the domain objects. In return, you get total control over the generated file, including sections not modeled by the domain objects.

[Example: Customizing the XML](#1rxd0f7)

**build.gradle**

apply plugin: 'eclipse-wtp'  
  
eclipse.wtp.facet.file.withXml { provider ->  
 provider.asNode().fixed.find { it.@facet == 'jst.java' }.@facet = 'jst2.java'  
}

[The IDEA Plugin](#2szc72q)

The IDEA plugin generates files that are used by [IntelliJ IDEA](http://www.jetbrains.com/idea/), thus making it possible to open the project from IDEA (File - Open Project). Both external dependencies (including associated source and Javadoc files) and project dependencies are considered.

| **Note** | If you simply want to load a Gradle project into IntelliJ IDEA, then use the IDE’s [import facility](https://www.jetbrains.com/help/idea/gradle.html#gradle_import). You do not need to apply this plugin to import your project into IDEA, although if you do, the import will take account of any extra IDEA configuration you have that doesn’t directly modify the generated files — see the [Configuration](#4bx0j30) section for more details. |
| --- | --- |

What exactly the IDEA plugin generates depends on which other plugins are used:

Always

Generates an IDEA module file. Also generates an IDEA project and workspace file if the project is the root project.

[Java Plugin](#2w5ecyt)

Additionally adds Java configuration to the IDEA module and project files.

One focus of the IDEA plugin is to be open to customization. The plugin provides a standardized set of hooks for adding and removing content from the generated files.

[Usage](#2r2atat)

To use the IDEA plugin, include this in your build script:

[Example: Using the IDEA plugin](#167l3im)

**build.gradle**

apply plugin: 'idea'

The IDEA plugin adds a number of tasks to your project. The idea task generates an IDEA module file for the project. When the project is the root project, the idea task also generates an IDEA project and workspace. The IDEA project includes modules for each of the projects in the Gradle build.

The IDEA plugin also adds an openIdea task when the project is the root project. This task generates the IDEA configuration files and opens the result in IDEA. This means you can simply run ./gradlew openIdea from the root project to generate and open the IDEA project in one convenient step.

The IDEA plugin also adds a cleanIdea task to the project. This task deletes the generated files, if present.

[Tasks](#3q78m6f)

The IDEA plugin adds the tasks shown below to a project. Notice that the clean task does not depend on the cleanIdeaWorkspace task. This is because the workspace typically contains a lot of user specific temporary data and it is not desirable to manipulate it outside IDEA.

idea

*Depends on*: ideaProject, ideaModule, ideaWorkspace

Generates all IDEA configuration files

openIdea

*Depends on*: idea

Generates all IDEA configuration files and opens the project in IDEA

cleanIdea — [Delete](http://docs.google.com/dsl/org.gradle.api.tasks.Delete.html)

*Depends on*: cleanIdeaProject, cleanIdeaModule

Removes all IDEA configuration files

cleanIdeaProject — [Delete](http://docs.google.com/dsl/org.gradle.api.tasks.Delete.html)

Removes the IDEA project file

cleanIdeaModule — [Delete](http://docs.google.com/dsl/org.gradle.api.tasks.Delete.html)

Removes the IDEA module file

cleanIdeaWorkspace — [Delete](http://docs.google.com/dsl/org.gradle.api.tasks.Delete.html)

Removes the IDEA workspace file

ideaProject — [GenerateIdeaProject](http://docs.google.com/dsl/org.gradle.plugins.ide.idea.GenerateIdeaProject.html)

Generates the .ipr file. This task is only added to the root project.

ideaModule — [GenerateIdeaModule](http://docs.google.com/dsl/org.gradle.plugins.ide.idea.GenerateIdeaModule.html)

Generates the .iml file

ideaWorkspace — [GenerateIdeaWorkspace](http://docs.google.com/dsl/org.gradle.plugins.ide.idea.GenerateIdeaWorkspace.html)

Generates the .iws file. This task is only added to the root project.

[Configuration](#4bx0j30)

The plugin adds some configuration options that allow to customize the IDEA project and module files that it generates. These take the form of both model properties and lower-level mechanisms that modify the generated files directly. For example, you can add source and resource directories, as well as inject your own fragments of XML. The former type of configuration is honored by IDEA’s import facility, whereas the latter is not.

Here are the configuration properties you can use:

idea — [IdeaModel](http://docs.google.com/dsl/org.gradle.plugins.ide.idea.model.IdeaModel.html)

Top level element that enables configuration of the idea plugin in a DSL-friendly fashion

idea.project [IdeaProject](http://docs.google.com/dsl/org.gradle.plugins.ide.idea.model.IdeaProject.html)

Allows configuring project information

idea.module [IdeaModule](http://docs.google.com/dsl/org.gradle.plugins.ide.idea.model.IdeaModule.html)

Allows configuring module information

idea.workspace [IdeaWorkspace](http://docs.google.com/dsl/org.gradle.plugins.ide.idea.model.IdeaWorkspace.html)

Allows configuring the workspace XML

Follow the links to the types for examples of using these configuration properties.

[Customizing the generated files](#25ciwe8)

The IDEA plugin provides hooks and behavior for customizing the generated content in a more controlled and detailed way. In addition, the withXml hook is the only practical way to modify the workspace file because its corresponding domain object is essentially empty.

| **Note** | The techniques we discuss in this section don’t work with IDEA’s import facility |
| --- | --- |

The tasks recognize existing IDEA files and merge them with the generated content.

[Merging](#kht6m1)

Sections of existing IDEA files that are also the target of generated content will be amended or overwritten, depending on the particular section. The remaining sections will be left as-is.

[Disabling merging with a complete overwrite](#34hgp9u)

To completely rewrite existing IDEA files, execute a clean task together with its corresponding generation task, like “gradle cleanIdea idea” (in that order). If you want to make this the default behavior, add “tasks.idea.dependsOn(cleanIdea)” to your build script. This makes it unnecessary to execute the clean task explicitly.

This strategy can also be used for individual files that the plugin would generate. For instance, this can be done for the “.iml” file with “gradle cleanIdeaModule ideaModule”.

[Hooking into the generation lifecycle](#1jmqzhn)

The plugin provides objects modeling the sections of the metadata files that are generated by Gradle. The generation lifecycle is as follows:

1. The file is read; or a default version provided by Gradle is used if it does not exist
2. The beforeMerged hook is executed with a domain object representing the existing file
3. The existing content is merged with the configuration inferred from the Gradle build or defined explicitly in the eclipse DSL
4. The whenMerged hook is executed with a domain object representing contents of the file to be persisted
5. The withXml hook is executed with a raw representation of the XML that will be persisted
6. The final XML is persisted

The following are the domain objects used for each of the model types:

[IdeaProject](http://docs.google.com/dsl/org.gradle.plugins.ide.idea.model.IdeaProject.html)

* beforeMerged { [Project](http://docs.google.com/javadoc/org/gradle/plugins/ide/idea/model/Project.html) arg -> …​ }
* whenMerged { [Project](http://docs.google.com/javadoc/org/gradle/plugins/ide/idea/model/Project.html) arg -> …​ }
* withXml { [XmlProvider](http://docs.google.com/javadoc/org/gradle/api/XmlProvider.html) arg -> …​ }

[IdeaModule](http://docs.google.com/javadoc/org/gradle/plugins/ide/idea/model/IdeaModule.html)

* beforeMerged { [Module](http://docs.google.com/javadoc/org/gradle/plugins/ide/idea/model/Module.html) arg -> …​ }
* whenMerged { [Module](http://docs.google.com/javadoc/org/gradle/plugins/ide/idea/model/Module.html) arg -> …​ }
* withXml { [XmlProvider](http://docs.google.com/javadoc/org/gradle/api/XmlProvider.html) arg -> …​ }

[IdeaWorkspace](http://docs.google.com/javadoc/org/gradle/plugins/ide/idea/model/IdeaWorkspace.html)

* beforeMerged { [Workspace](http://docs.google.com/javadoc/org/gradle/plugins/ide/idea/model/Workspace.html) arg -> …​ }
* whenMerged { [Workspace](http://docs.google.com/javadoc/org/gradle/plugins/ide/idea/model/Workspace.html) arg -> …​ }
* withXml { [XmlProvider](http://docs.google.com/javadoc/org/gradle/api/XmlProvider.html) arg -> …​ }

[Partial rewrite of existing content](#43mei5g)

A "complete rewrite" causes all existing content to be discarded, thereby losing any changes made directly in the IDE. The beforeMerged hook makes it possible to overwrite just certain parts of the existing content. The following example removes all existing dependencies from the Module domain object:

[Example: Partial Rewrite for Module](#2irosd9)

**build.gradle**

idea.module.iml {  
 beforeMerged { module ->  
 module.dependencies.clear()  
 }  
}

The resulting module file will only contain Gradle-generated dependency entries, but not any other dependency entries that may have been present in the original file. (In the case of dependency entries, this is also the default behavior.) Other sections of the module file will be either left as-is or merged. The same could be done for the module paths in the project file:

[Example: Partial Rewrite for Project](#xwz2l2)

**build.gradle**

idea.project.ipr {  
 beforeMerged { project ->  
 project.modulePaths.clear()  
 }  
}

[Modifying the fully populated domain objects](#3hwml8v)

The whenMerged hook allows you to manipulate the fully populated domain objects. Often this is the preferred way to customize IDEA files. Here is how you would export all the dependencies of an IDEA module:

[Example: Export Dependencies](#1x1wvgo)

**build.gradle**

idea.module.iml {  
 whenMerged { module ->  
 module.dependencies\*.exported = true  
 }  
}

[Modifying the XML representation](#4h1ke4h)

The withXml hook allows you to manipulate the in-memory XML representation just before the file gets written to disk. Although Groovy’s XML support makes up for a lot, this approach is less convenient than manipulating the domain objects. In return, you get total control over the generated file, including sections not modeled by the domain objects.

[Example: Customizing the XML](#2w6uoca)

**build.gradle**

idea.project.ipr {  
 withXml { provider ->  
 provider.node.component  
 .find { it.@name == 'VcsDirectoryMappings' }  
 .mapping.@vcs = 'Git'  
 }  
}

[Further things to consider](#1bc4yk3)

The paths of dependencies in the generated IDEA files are absolute. If you manually define a path variable pointing to the Gradle dependency cache, IDEA will automatically replace the absolute dependency paths with this path variable. you can configure this path variable via the “idea.pathVariables” property, so that it can do a proper merge without creating duplicates.

[Embedding Gradle using the Tooling API](#184mhaj)

[Introduction to the Tooling API](#3vbsh7w)

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](#ihv636) uses the Tooling API for functional testing of your Gradle plugins.
* [Eclipse Buildship](http://projects.eclipse.org/projects/tools.buildship) uses the Tooling API for importing your Gradle project and running tasks.
* [IntelliJ IDEA](https://www.jetbrains.com/idea/) uses the Tooling API for importing your Gradle project and running tasks.

[Tooling API Features](#2ah2rfp)

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](#pmd1ni)

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](#44sinio) contains more details about the daemon, specifically information on situations when new daemons are forked.

[Quickstart](#39m0kbb)

As the Tooling API is an interface for developers, the Javadoc is the main documentation for it. We provide several *samples* that live in samples/toolingApi in your Gradle distribution. These samples specify all of the required dependencies for the Tooling API with examples for querying information from Gradle builds and executing tasks from the Tooling API.

To use the Tooling API, add the following repository and dependency declarations to your build script:

[Example: Using the tooling API](#1orauj4)

**build.gradle**

repositories {  
 maven { url 'https://repo.gradle.org/gradle/libs-releases' }  
}  
  
dependencies {  
 compile "org.gradle:gradle-tooling-api:${toolingApiVersion}"  
 *// The tooling API need an SLF4J implementation available at runtime, replace this with any other implementation*  
 runtime 'org.slf4j:slf4j-simple:1.7.10'  
}

The main entry point to the Tooling API is the [GradleConnector](http://docs.google.com/javadoc/org/gradle/tooling/GradleConnector.html). You can navigate from there to find code samples and explore the available Tooling API models. You can use [GradleConnector.connect()](https://docs.gradle.org/nightly/javadoc/org/gradle/tooling/GradleConnector.html#connect--) to create a [ProjectConnection](http://docs.google.com/javadoc/org/gradle/tooling/ProjectConnection.html). A ProjectConnection connects to a single Gradle project. Using the connection you can execute tasks, tests and retrieve models relative to this project.

[Gradle version and Java version compatibility](#48qyd6x)

[Provider side](#2nw8neq)

The current version of Tooling API supports running builds using Gradle versions 1.2 and later. However, support for running builds with Gradle versions older than 2.6 is deprecated and will be removed in Tooling API version 5.0.

[Consumer side](#131ixmj)

The current version of Gradle supports running builds via Tooling API versions 2.0 and later. However, support for running builds via Tooling API versions older than 3.0 is deprecated and will be removed in Gradle 5.0.

You should note that not all features of the Tooling API are available for all versions of Gradle. For example, build cancellation is only available when a build uses Gradle 2.1 and later. Refer to the documentation for each class and method for more details.

[Java version](#3n16gac)

The Tooling API requires Java 8 or later. Java 7 is currently still supported but will be removed in Gradle 5.0. The Gradle version used by builds may have additional Java version requirements.

[Extending Gradle](#3s49zyc)

[Writing Custom Plugins](#279ka65)

A Gradle plugin packages up reusable pieces of build logic, which can be used across many different projects and builds. Gradle allows you to implement your own plugins, so you can reuse your build logic, and share it with others.

You can implement a Gradle plugin in any language you like, provided the implementation ends up compiled as bytecode. In our examples, we are going to use Groovy as the implementation language. Groovy, Java or Kotlin are all good choices as the language to use to implement a plugin, as the Gradle API has been designed to work well with these languages. In general, a plugin implemented using Java or Kotlin, which are statically typed, will perform better than the same plugin implemented using Groovy.

[Packaging a plugin](#2k82xt6)

There are several places where you can put the source for the plugin.

Build script

You can include the source for the plugin directly in the build script. This has the benefit that the plugin is automatically compiled and included in the classpath of the build script without you having to do anything. However, the plugin is not visible outside the build script, and so you cannot reuse the plugin outside the build script it is defined in.

buildSrc project

You can put the source for the plugin in the *rootProjectDir*/buildSrc/src/main/groovy directory. Gradle will take care of compiling and testing the plugin and making it available on the classpath of the build script. The plugin is visible to every build script used by the build. However, it is not visible outside the build, and so you cannot reuse the plugin outside the build it is defined in.

See [Organizing Gradle Projects](#3whwml4) for more details about the buildSrc project.

Standalone project

You can create a separate project for your plugin. This project produces and publishes a JAR which you can then use in multiple builds and share with others. Generally, this JAR might include some plugins, or bundle several related task classes into a single library. Or some combination of the two.

In our examples, we will start with the plugin in the build script, to keep things simple. Then we will look at creating a standalone project.

[Writing a simple plugin](#226gqi5)

To create a Gradle plugin, you need to write a class that implements the [Plugin](http://docs.google.com/javadoc/org/gradle/api/Plugin.html) interface. When the plugin is applied to a project, Gradle creates an instance of the plugin class and calls the instance’s [Plugin.apply()](http://docs.google.com/javadoc/org/gradle/api/Plugin.html#apply-T-) method. The project object is passed as a parameter, which the plugin can use to configure the project however it needs to. The following sample contains a greeting plugin, which adds a hello task to the project.

[Example: A custom plugin](#hbr0py)

**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

**Output of** gradle -q hello

> gradle -q hello  
Hello from the GreetingPlugin

One thing to note is that a new instance of a plugin is created for each project it is applied to. Also note that the [Plugin](http://docs.google.com/javadoc/org/gradle/api/Plugin.html) class is a generic type. This example has it receiving the [Project](http://docs.google.com/dsl/org.gradle.api.Project.html) type as a type parameter. A plugin can instead receive a parameter of type [Settings](http://docs.google.com/dsl/org.gradle.api.initialization.Settings.html), in which case the plugin can be applied in a settings script, or a parameter of type [Gradle](http://docs.google.com/dsl/org.gradle.api.invocation.Gradle.html), in which case the plugin can be applied in an initialization script.

[Making the plugin configurable](#31bejdr)

Most plugins need to obtain some configuration from the build script. One method for doing this is to use *extension objects*. The Gradle [Project](http://docs.google.com/dsl/org.gradle.api.Project.html) has an associated [ExtensionContainer](http://docs.google.com/javadoc/org/gradle/api/plugins/ExtensionContainer.html) 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 a Java Bean compliant class. Groovy is a good language choice to implement an extension object because plain old Groovy objects contain all the getter and setter methods that a Java Bean requires. Java and Kotlin are other good choices.

Let’s add a simple extension object to the project. Here we add a greeting extension object to the project, which allows you to configure the greeting.

[Example: A custom plugin extension](#1ggotlk)

**build.gradle**

**class** **GreetingPluginExtension** {  
 String message = 'Hello from GreetingPlugin'  
}  
  
**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  
 }  
 }  
 }  
}  
  
apply plugin: GreetingPlugin  
  
*// Configure the extension*  
greeting.message = 'Hi from Gradle'

**Output of** gradle -q hello

> gradle -q hello  
Hi from Gradle

In this example, GreetingPluginExtension is a plain old Groovy object with a property called message. The extension object is added to the plugin list with the name greeting. This object then becomes available as a project property with the same name as the extension object.

Oftentimes, you have several related properties you need to specify on a single plugin. Gradle adds a configuration closure block for each extension object, so you can group settings together. The following example shows you how this works.

[Example: A custom plugin with configuration closure](#40gcc9d)

**build.gradle**

**class** **GreetingPluginExtension** {  
 String message  
 String greeter  
}  
  
**class** **GreetingPlugin** **implements** Plugin<Project> {  
 **void** apply(Project project) {  
 **def** extension = project.extensions.create('greeting', GreetingPluginExtension)  
 project.task('hello') {  
 doLast {  
 println "${extension.message} from ${extension.greeter}"  
 }  
 }  
 }  
}  
  
apply plugin: GreetingPlugin  
  
*// Configure the extension using a DSL block*  
greeting {  
 message = 'Hi'  
 greeter = 'Gradle'  
}

**Output of** gradle -q hello

> gradle -q hello  
Hi from Gradle

In this example, several settings can be grouped together within the greeting closure. The name of the closure block in the build script (greeting) needs to 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.

[Working with files in custom tasks and plugins](#2flmmh6)

When developing custom tasks and plugins, it’s a good idea to be very flexible when accepting input configuration for file locations. To do this, you can leverage the [Project.file(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:file(java.lang.Object)) method to resolve values to files as late as possible.

[Example: Evaluating file properties lazily](#uqwwoz)

**build.gradle**

**class** **GreetingToFileTask** **extends** DefaultTask {  
  
 **def** destination  
  
 File getDestination() {  
 project.file(destination)  
 }  
  
 @TaskAction  
 **def** **greet**() {  
 **def** file = getDestination()  
 file.parentFile.mkdirs()  
 file.write 'Hello!'  
 }  
}  
task greet(type: GreetingToFileTask) {  
 destination = { project.greetingFile }  
}  
  
task sayGreeting(dependsOn: greet) {  
 doLast {  
 println file(greetingFile).text  
 }  
}  
  
ext.greetingFile = "$buildDir/hello.txt"

**Output of** gradle -q sayGreeting

> gradle -q sayGreeting  
Hello!

In this example, we configure the greet task destination property as a closure, which is evaluated with the [Project.file(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:file(java.lang.Object)) method to turn the return value of the closure into a File object at the last minute. You will notice that in the example above we specify the greetingFile property value after we have configured to use it for the task. This kind of lazy evaluation is a key benefit of accepting any value when setting a file property, then resolving that value when reading the property.

[Mapping extension properties to task properties](#3eqkfcs)

Capturing user input from the build script through an extension and mapping it to input/output properties of a custom task is considered a best practice. The end user only interacts with the exposed DSL defined by the extension. The imperative logic is hidden in the plugin implementation.

The extension declaration in the build script as well as the mapping between extension properties and custom task properties occurs during Gradle’s configuration phase of the build lifecycle. To avoid evaluation order issues, the actual value of a mapped property has to be resolved during the execution phase. For more information please see [the section on build phases](#27jua8u). Gradle’s API offers types for representing a property that should be lazily evaluated e.g. during execution time. Refer to [Lazy Configuration](#36ei31r) for more information.

The following demonstrates the usage of the type for mapping an extension property to a task property:

[Example: Mapping extension properties to task properties](#1tvupkl)

**build.gradle**

**class** **GreetingPlugin** **implements** Plugin<Project> {  
 **void** apply(Project project) {  
 **def** extension = project.extensions.create('greeting', GreetingPluginExtension, project)  
 project.tasks.create('hello', Greeting) {  
 message = extension.message  
 outputFiles = extension.outputFiles  
 }  
 }  
}  
  
**class** **GreetingPluginExtension** {  
 **final** Property<String> message  
 **final** ConfigurableFileCollection outputFiles  
  
 GreetingPluginExtension(Project project) {  
 message = project.objects.property(String)  
 message.set('Hello from GreetingPlugin')  
 outputFiles = project.layout.configurableFiles()  
 }  
  
 **void** setOutputFiles(FileCollection outputFiles) {  
 this.outputFiles.setFrom(outputFiles)  
 }  
}  
  
**class** **Greeting** **extends** DefaultTask {  
 **final** Property<String> message = project.objects.property(String)  
 **final** ConfigurableFileCollection outputFiles = project.layout.configurableFiles()  
  
 **void** setOutputFiles(FileCollection outputFiles) {  
 this.outputFiles.setFrom(outputFiles)  
 }  
  
 @TaskAction  
 **void** printMessage() {  
 outputFiles.each {  
 logger.quiet "Writing message 'Hi from Gradle' to file"  
 it.text = message.get()  
 }  
 }  
}  
  
apply plugin: GreetingPlugin  
  
greeting {  
 message = 'Hi from Gradle'  
 outputFiles = layout.files('a.txt', 'b.txt')  
}

| **Note** | The code for this example can be found at samples/userguide/tasks/mapExtensionPropertiesToTaskProperties in the ‘-all’ distribution of Gradle. |
| --- | --- |

**Output of gradle -q hello**

> gradle -q hello  
Writing message 'Hi from Gradle' to file  
Writing message 'Hi from Gradle' to file

[A standalone project](#4dvi88e)

Now we will move our plugin to a standalone project, so we can publish it and share it with others. This project is simply a Groovy project that produces a JAR containing the plugin classes. Here is a simple build script for the project. It applies the Groovy plugin, and adds the Gradle API as a compile-time dependency.

[Example: A build for a custom plugin](#2t0sig7)

**build.gradle**

plugins {  
 id 'groovy'  
}  
  
dependencies {  
 compile gradleApi()  
 compile localGroovy()  
}

| **Note** | The code for this example can be found at samples/customPlugin/plugin in the ‘-all’ distribution of Gradle. |
| --- | --- |

So how does Gradle find the [Plugin](http://docs.google.com/javadoc/org/gradle/api/Plugin.html) implementation? The answer is you need to provide a properties file in the jar’s META-INF/gradle-plugins directory that matches the id of your plugin.

[Example: Wiring for a custom plugin](#1862so0)

**src/main/resources/META-INF/gradle-plugins/org.samples.greeting.properties**

implementation-class=org.gradle.GreetingPlugin

Notice that the properties filename matches the plugin id and is placed in the resources folder, and that the implementation-class property identifies the [Plugin](http://docs.google.com/javadoc/org/gradle/api/Plugin.html) implementation class.

[Creating a plugin id](#3s5qbbt)

Plugin ids are fully qualified in a manner similar to Java packages (i.e. a reverse domain name). This helps to avoid collisions and provides a way to group plugins with similar ownership.

Your plugin id should be a combination of components that reflect namespace (a reasonable pointer to you or your organization) and the name of the plugin it provides. For example if you had a Github account named "foo" and your plugin was 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 conform to the following:

* May contain any alphanumeric character, '.', and '-'.
* Must contain at least one '.' character separating the namespace from the name of the plugin.
* Conventionally use a lowercase reverse domain name convention for the namespace.
* Conventionally use only lowercase characters in the name.
* org.gradle and com.gradleware namespaces may not be used.
* Cannot start or end with a '.' character.
* Cannot contain consecutive '.' characters (i.e. '..').

Although there are conventional similarities between plugin ids and package names, package names are generally more detailed than is necessary for a plugin id. For instance, it might seem reasonable to add "gradle" as a component of your plugin id, but since plugin ids are only used for Gradle plugins, this would be superfluous. Generally, a namespace that identifies ownership and a name are all that are needed for a good plugin id.

[Publishing your plugin](#27b0ljm)

If you are publishing your plugin internally for use within your organization, you can publish it like any other code artifact. See the [Ivy](#3ygebqi) and [Maven](#1egqt2p) 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 the [Gradle Plugin Portal](http://plugins.gradle.org). This site provides the ability to search for and gather information about plugins contributed by the Gradle community. Please refer to the corresponding [guide](https://guides.gradle.org/publishing-plugins-to-gradle-plugin-portal/) on how to make your plugin available on this site.

[Using your plugin in another project](#mgavrf)

To use a plugin in a build script, you need to add the plugin classes to the build script’s classpath. To do this, you use a “buildscript { }” block, as described in see [Applying plugins using the buildscript block](#1iya7iv). The following example shows how you might do this when the JAR containing the plugin has been published to a local repository:

[Example: Using a custom plugin in another project](#36fyef8)

**build.gradle**

buildscript {  
 repositories {  
 maven {  
*// END SNIPPET use-plugin*  
*// END SNIPPET use-task*  
 **def** producerName = findProperty('producerName') ?: 'plugin'  
 **def** repoLocation = "../$producerName/build/repo"  
*// START SNIPPET use-plugin*  
*// START SNIPPET use-task*  
 url = uri(repoLocation)  
 }  
 }  
 dependencies {  
 classpath group: 'org.gradle', name: 'customPlugin',  
 version: '1.0-SNAPSHOT'  
 }  
}  
apply plugin: 'org.samples.greeting'

Alternatively, if your plugin is published to the plugin portal, you can use the incubating plugins DSL (see [Applying plugins using the plugins DSL](#10ekgid)) to apply the plugin:

[Example: Applying a community plugin with the plugins DSL](#1ll8on1)

**build.gradle**

plugins {  
 id 'com.jfrog.bintray' version '0.4.1'  
}

[Writing tests for your plugin](#45kw7au)

You can use the [ProjectBuilder](http://docs.google.com/javadoc/org/gradle/testfixtures/ProjectBuilder.html) class to create [Project](http://docs.google.com/dsl/org.gradle.api.Project.html) instances to use when you test your plugin implementation.

[Example: Testing a custom plugin](#2kq6hin)

**src/test/groovy/org/gradle/GreetingPluginTest.groovy**

**class** **GreetingPluginTest** {  
 @Test  
 **public** **void** greeterPluginAddsGreetingTaskToProject() {  
 Project project = ProjectBuilder.builder().build()  
 project.pluginManager.apply 'org.samples.greeting'  
  
 assertTrue(project.tasks.hello **instanceof** GreetingTask)  
 }  
}

[Using the Java Gradle Plugin Development Plugin](#zvgrqg)

You can use the incubating [Java Gradle Plugin Development Plugin](#meukdy) to eliminate some of the boilerplate declarations in your build script and provide some basic validations of plugin metadata. This plugin will automatically apply the [Java Plugin](#2w5ecyt), add the gradleApi() dependency to the compile configuration, and perform plugin metadata validations as part of the jar task execution, and generate plugin descriptors in the resulting JAR’s META-INF directory.

[Example: Using the Java Gradle Plugin Development plugin](#3jv4ae9)

**build.gradle**

plugins {  
 id 'java-gradle-plugin'  
 id 'groovy'  
}  
  
gradlePlugin {  
 plugins {  
 simplePlugin {  
 id = 'org.samples.greeting'  
 implementationClass = 'org.gradle.GreetingPlugin'  
 }  
 }  
}

When publishing plugins to custom plugin repositories using the [Ivy](#3ygebqi) or [Maven](#1egqt2p) publish plugins, the [Java Gradle Plugin Development Plugin](#meukdy) will also generate plugin marker artifacts named based on the plugin id which depend on the plugin’s implementation artifact.

[Providing a configuration DSL for the plugin](#1z0ekm2)

As we saw above, you can use an extension object to provide configuration for your plugin. Using an extension object also extends the Gradle DSL to add a project property and DSL block for the plugin. An extension object is simply a regular object, and so you can provide DSL elements nested inside this block by adding properties and methods to the extension object.

Gradle provides several conveniences to help create a well-behaved DSL for your plugin.

[Nested DSL elements](#4j0239v)

When Gradle creates a task or extension object, Gradle *decorates* the implementation class to mix in DSL support. To create a nested DSL element you can use the [ObjectFactory](http://docs.google.com/javadoc/org/gradle/api/model/ObjectFactory.html) type to create objects that are similarly decorated. These decorated objects can then be made visible to the DSL through properties and methods of the plugin’s extension:

[Example: Nested DSL elements](#2y5cdho)

**build.gradle**

**class** **Person** {  
 String name  
}  
  
**class** **GreetingPluginExtension** {  
 String message  
 **final** Person greeter  
  
 @javax.inject.Inject  
 GreetingPluginExtension(ObjectFactory objectFactory) {  
 *// Create a Person instance*  
 greeter = objectFactory.newInstance(Person)  
 }  
  
 **void** greeter(Action<? super Person> action) {  
 action.execute(greeter)  
 }  
}  
  
**class** **GreetingPlugin** **implements** Plugin<Project> {  
 **void** apply(Project project) {  
 *// Create the extension, passing in an ObjectFactory for it to use*  
 **def** extension = project.extensions.create('greeting', GreetingPluginExtension, project.objects)  
 project.task('hello') {  
 doLast {  
 println "${extension.message} from ${extension.greeter.name}"  
 }  
 }  
 }  
}  
  
apply plugin: GreetingPlugin  
  
greeting {  
 message = 'Hi'  
 greeter {  
 name = 'Gradle'  
 }  
}

**Output of** gradle -q hello

> gradle -q hello  
Hi from Gradle

In this example, the plugin passes the project’s ObjectFactory to the extension object through its constructor. The constructor uses this to create a nested object and makes this object available to the DSL through the greeter property.

[Configuring a collection of objects](#1damnph)

Gradle provides some utility classes for maintaining collections of objects, intended to work well with the Gradle DSL.

[Example: Managing a collection of objects](#3xaa6da)

**build.gradle**

**class** **Book** {  
 **final** String name  
 File sourceFile  
  
 Book(String name) {  
 this.name = name  
 }  
}  
  
**class** **DocumentationPlugin** **implements** Plugin<Project> {  
 **void** apply(Project project) {  
 *// Create a container of Book instances*  
 **def** books = project.container(Book)  
 books.all {  
 sourceFile = project.file("src/docs/$name")  
 }  
 *// Add the container as an extension object*  
 project.extensions.books = books  
 }  
}  
  
apply plugin: DocumentationPlugin  
  
*// Configure the container*  
books {  
 quickStart {  
 sourceFile = file('src/docs/quick-start')  
 }  
 userGuide {  
  
 }  
 developerGuide {  
  
 }  
}  
  
task books {  
 doLast {  
 books.each { book ->  
 println "$book.name -> $book.sourceFile"  
 }  
 }  
}

**Output of** gradle -q books

> gradle -q books  
developerGuide -> /home/user/gradle/samples/src/docs/developerGuide  
quickStart -> /home/user/gradle/samples/src/docs/quick-start  
userGuide -> /home/user/gradle/samples/src/docs/userGuide

The [Project.container(java.lang.Class)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:container(java.lang.Class)) methods create instances of [NamedDomainObjectContainer](http://docs.google.com/dsl/org.gradle.api.NamedDomainObjectContainer.html), that have many useful methods for managing and configuring the objects. In order to use a type with any of the project.container methods, it MUST expose a property named “name” as the unique, and constant, name for the object. The project.container(Class) variant of the container method creates new instances by attempting to invoke the constructor of the class that takes a single string argument, which is the desired name of the object. See the above link for project.container method variants that allow custom instantiation strategies.

[Gradle Plugin Development Plugin](#meukdy)

| **Note** | The Java Gradle plugin development plugin is currently [incubating](#2xcytpi). Please be aware that the DSL and other configuration may change in later Gradle versions. |
| --- | --- |

The Java Gradle Plugin development plugin can be used to assist in the development of Gradle plugins. It automatically applies the [Java](#2w5ecyt) plugin, adds the gradleApi() dependency to the compile configuration and performs validation of plugin metadata during jar task execution.

The plugin also integrates with [TestKit](#ihv636), a library that aids in writing and executing functional tests for plugin code. It automatically adds the gradleTestKit() dependency to the test compile configuration and generates a plugin classpath manifest file consumed by a GradleRunner instance if found. Please refer to [Automatic classpath injection with the Plugin Development Plugin](#27u4abj) for more on its usage, configuration options and samples.

[Usage](#2cfkgl3)

To use the Java Gradle Plugin Development plugin, include the following in your build script:

[Example: Using the Java Gradle Plugin Development plugin](#rkuqsw)

**build.gradle**

plugins {  
 id 'java-gradle-plugin'  
}

Applying the plugin automatically applies the [Java](#2w5ecyt) plugin and adds the gradleApi() dependency to the compile configuration. It also adds some validations to the build.

The following validations are performed:

* There is a plugin descriptor defined for the plugin.
* The plugin descriptor contains an implementation-class property.
* The implementation-class property references a valid class file in the jar.
* Each property getter or the corresponding field must be annotated with a property annotation like @InputFile and @OutputDirectory. Properties that don’t participate in up-to-date checks should be annotated with @Internal.

Any failed validations will result in a warning message.

For each plugin you are developing, add an entry to the gradlePlugin {} script block:

[Example: Using the gradlePlugin {} block.](#3bki9gp)

**build.gradle**

gradlePlugin {  
 plugins {  
 simplePlugin {  
 id = 'org.gradle.sample.simple-plugin'  
 implementationClass = 'org.gradle.sample.SimplePlugin'  
 }  
 }  
}

The gradlePlugin {} block defines the plugins being built by the project including the id and implementationClass of the plugin. From this data about the plugins being developed, Gradle can automatically:

* Generate the plugin descriptor in the jar file’s META-INF directory.
* Configure the [Maven](#1egqt2p) or [Ivy Publish Plugins](#3ygebqi) publishing plugins to publish a [Plugin Marker Artifact](#482hl85) for each plugin.
* Moreover, if the [Plugin Publishing Plugin](https://plugins.gradle.org/docs/publish-plugin) is applied, it will publish each plugin using the same name, plugin id, display name, and description to the Gradle Plugin Portal (see [Publishing Plugins to Gradle Plugin Portal](https://guides.gradle.org/publishing-plugins-to-gradle-plugin-portal/) for details).

[Lazy Configuration](#36ei31r)

As a build grows in complexity, knowing when and where a particular value is configured can become difficult to reason about. Gradle provides several ways to manage this complexity using *lazy configuration*.

[Lazy properties](#1qpsjoi)

| **Note** | The Provider API is currently [incubating](#2xcytpi). Please be aware that the DSL and other configuration may change in later Gradle versions. |
| --- | --- |

Gradle provides lazy properties, which delay the calculation of a property’s value until it’s absolutely required. Lazy types are faster, more understandable and better instrumented than the internal convention mapping mechanisms. This provides two main benefits to build script and plugin authors:

1. Build authors can wire together Gradle models without worrying when a particular property’s value will be known. For example, when you want to map properties in an extension to task properties but the values aren’t known until the build script configures them.
2. Build authors can avoid resource intensive work during the configuration phase, which can have a direct impact on maximum build performance. For example, when a property value comes from parsing a file.

Gradle represents lazy properties with two interfaces:

* [Provider](http://docs.google.com/javadoc/org/gradle/api/provider/Provider.html) are properties that can only be queried and cannot be changed.
  + Properties with these types are read-only.
  + The method [Provider.get()](http://docs.google.com/javadoc/org/gradle/api/provider/Provider.html#get--) returns the current value of the property.
  + A Provider can be created by the factory method [ProviderFactory.provider(java.util.concurrent.Callable)](http://docs.google.com/javadoc/org/gradle/api/provider/ProviderFactory.html#provider-java.util.concurrent.Callable-).
* [Property](http://docs.google.com/javadoc/org/gradle/api/provider/Property.html) are properties that can be queried and overwritten.
  + Properties with these types are configurable.
  + Property implements the Provider interface.
  + The method [Property.set(T)](http://docs.google.com/javadoc/org/gradle/api/provider/Property.html#set-T-) specifies a value for the property, overwriting whatever value may have been present.
  + The method [Property.set(org.gradle.api.provider.Provider)](http://docs.google.com/javadoc/org/gradle/api/provider/Property.html#set-org.gradle.api.provider.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(java.lang.Class)](http://docs.google.com/javadoc/org/gradle/api/model/ObjectFactory.html#property-java.lang.Class-).

Neither of these types nor their subtypes are intended to be implemented by a build script or plugin author. Gradle provides several factory methods to create instances of these types. See the [Quick Reference](#4apg2cb) for all of the types and factories available.

Lazy properties are intended to be passed around and only evaluated when required (usually, during the execution phase). For more information about the Gradle build phases, please see [Build Lifecycle](#27jua8u).

The following demonstrates a task with a read-only property and a configurable property:

[Example: Using a read-only and configurable property](#2puqck4)

**build.gradle**

**class** **Greeting** **extends** DefaultTask {  
 *// Configurable by the user*  
 @Input  
 **final** Property<String> message = project.objects.property(String)  
  
 *// Read-only property calculated from the message*  
 @Internal  
 **final** Provider<String> fullMessage = message.map { it + " from Gradle" }  
  
 @TaskAction  
 **void** printMessage() {  
 logger.quiet(fullMessage.get())  
 }  
}  
  
task greeting(type: Greeting) {  
 *// Note that this is effectively calling Property.set()*  
 message = 'Hi'  
}

**Output of** gradle greeting

> gradle greeting  
  
> Task :greeting  
Hi from Gradle  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

The Greeting task has a Property<String> for the mutable part of the message and a Provider<String> for the calculated, read-only, message.

| **Note** | Note that Groovy Gradle DSL will generate setter methods for each Property-typed property in a task implementation. These setter methods allow you to configure the property using the assignment (=) operator as a convenience. |
| --- | --- |

[Creating a Property or Provider](#1500mrx)

If provider types are not intended to be implemented directly by build script or plugin authors, how do you create a new one? Gradle provides various factory APIs to create new instances of both [Provider](http://docs.google.com/javadoc/org/gradle/api/provider/Provider.html) and [Property](http://docs.google.com/javadoc/org/gradle/api/provider/Property.html):

* [ProviderFactory.provider(java.util.concurrent.Callable)](http://docs.google.com/javadoc/org/gradle/api/provider/ProviderFactory.html#provider-java.util.concurrent.Callable-) instantiates a new Provider. An instance of the [ProviderFactory](http://docs.google.com/javadoc/org/gradle/api/provider/ProviderFactory.html) can be referenced from [Project.getProviders()](http://docs.google.com/javadoc/org/gradle/api/Project.html#getProviders--) or by injecting ProviderFactory through a constructor or method.
* [ObjectFactory.property(java.lang.Class)](http://docs.google.com/javadoc/org/gradle/api/model/ObjectFactory.html#property-java.lang.Class-) instantiates a new Property. An instance of the [ObjectFactory](http://docs.google.com/javadoc/org/gradle/api/model/ObjectFactory.html) can be referenced from [Project.getObjects()](http://docs.google.com/javadoc/org/gradle/api/Project.html#getObjects--) or by injecting ObjectFactory through a constructor or method.

| **Note** | [Project](http://docs.google.com/dsl/org.gradle.api.Project.html) does not provide a specific method signature for creating a provider from a groovy.lang.Closure. When writing a plugin with Groovy, you can use the method signature accepting a java.util.concurrent.Callable parameter. Groovy’s [Closure to type coercion](http://docs.groovy-lang.org/next/html/documentation/core-semantics.html#_assigning_a_closure_to_a_sam_type) will take care of the rest. |
| --- | --- |

[Working with files and Providers](#3ozo5fq)

In [Working with Files](#1hmsyys), we introduced four collection types for File-like objects:

Table 37. Collection of files recap

| **Read-only Type** | **Configurable Type** |
| --- | --- |
| [FileCollection](http://docs.google.com/javadoc/org/gradle/api/file/FileCollection.html) | [ConfigurableFileCollection](http://docs.google.com/javadoc/org/gradle/api/file/ConfigurableFileCollection.html) |
| [FileTree](http://docs.google.com/javadoc/org/gradle/api/file/FileTree.html) | [ConfigurableFileTree](http://docs.google.com/javadoc/org/gradle/api/file/ConfigurableFileTree.html) |

All of these types are also considered Provider types.

In this section, we are going to introduce more strongly typed models for a [FileSystemLocation](http://docs.google.com/javadoc/org/gradle/api/file/FileSystemLocation.html): [Directory](http://docs.google.com/javadoc/org/gradle/api/file/Directory.html) and [RegularFile](http://docs.google.com/javadoc/org/gradle/api/file/RegularFile.html). These types shouldn’t be confused with the standard Java [java.io.File](https://docs.oracle.com/javase/7/docs/api/java/io/File.html) type as they tell Gradle to expect more specific values (a directory or a non-directory, regular file).

Gradle provides two specialized Property subtypes for dealing with these types: [RegularFileProperty](http://docs.google.com/javadoc/org/gradle/api/file/RegularFileProperty.html) and [DirectoryProperty](http://docs.google.com/javadoc/org/gradle/api/file/DirectoryProperty.html). [ProjectLayout](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html) has methods to create these: [ProjectLayout.fileProperty()](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#fileProperty--) and [ProjectLayout.directoryProperty()](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#directoryProperty--).

A DirectoryProperty can also be used to create a lazily evaluated Provider for a Directory and RegularFile via [DirectoryProperty.dir(java.lang.String)](http://docs.google.com/javadoc/org/gradle/api/file/DirectoryProperty.html#dir-java.lang.String-) and [DirectoryProperty.file(java.lang.String)](http://docs.google.com/javadoc/org/gradle/api/file/DirectoryProperty.html#file-java.lang.String-) respectively. These methods create paths that are relative to the location set for the original DirectoryProperty.

[Example: Using file and directory property](#244yfnj)

**build.gradle**

**class** **FooExtension** {  
 **final** DirectoryProperty someDirectory  
 **final** RegularFileProperty someFile  
 **final** ConfigurableFileCollection someFiles  
  
 FooExtension(Project project) {  
 someDirectory = project.layout.directoryProperty()  
 someFile = project.layout.fileProperty()  
 someFiles = project.layout.configurableFiles()  
 }  
}  
  
project.extensions.create('foo', FooExtension, project)  
  
foo {  
 someDirectory = project.layout.projectDirectory.dir('some-directory')  
 someFile = project.layout.buildDirectory.file('some-file')  
 someFiles.from project.layout.configurableFiles(someDirectory, someFile)  
}  
  
task print {  
 doLast {  
 **def** someDirectory = project.foo.someDirectory.get().asFile  
 logger.quiet("foo.someDirectory = " + someDirectory)  
 logger.quiet("foo.someFiles contains someDirectory? " + project.foo.someFiles.contains(someDirectory))  
  
 **def** someFile = project.foo.someFile.get().asFile  
 logger.quiet("foo.someFile = " + someFile)  
 logger.quiet("foo.someFiles contains someFile? " + project.foo.someFiles.contains(someFile))  
 }  
}

**Output of** gradle print

> gradle print  
  
> Task :print  
foo.someDirectory = /home/user/gradle/samples/some-directory  
foo.someFiles contains someDirectory? true  
foo.someFile = /home/user/gradle/samples/build/some-file  
foo.someFiles contains someFile? true  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

This example shows how Provider types can be used inside an extension. Lazy values for [Project.getBuildDir()](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:buildDir) and [Project.getProjectDir()](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:projectDir) can be accessed through [Project.getLayout()](http://docs.google.com/javadoc/org/gradle/api/Project.html#getLayout--) with [ProjectLayout.getBuildDirectory()](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#getBuildDirectory--) and [ProjectLayout.getProjectDirectory()](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#getProjectDirectory--).

[Working with task dependencies and Providers](#ja8pvc)

Many builds have several tasks that depend on each other. This usually means that one task processes the outputs of another task as an input. For these outputs and inputs, we need to know their locations on the file system and appropriately configure each task to know where to look. This can be cumbersome if any of these values are configurable by a user or configured by multiple plugins.

To make this easier, Gradle offers convenient APIs for defining files or directories as task inputs and outputs in a descriptive way. As an example consider the following plugin with a producer and consumer task, which are wired together via inputs and outputs:

[Example: Implicit task dependency](#339w8j5)

**build.gradle**

**class** **Producer** **extends** DefaultTask {  
 @OutputFile  
 **final** RegularFileProperty outputFile = newOutputFile()  
  
 @TaskAction  
 **void** produce() {  
 String message = 'Hello, World!'  
 **def** output = outputFile.get().asFile  
 output.text = message  
 logger.quiet("Wrote '${message}' to ${output}")  
 }  
}  
  
**class** **Consumer** **extends** DefaultTask {  
 @InputFile  
 **final** RegularFileProperty inputFile = newInputFile()  
  
 @TaskAction  
 **void** consume() {  
 **def** input = inputFile.get().asFile  
 **def** message = input.text  
 logger.quiet("Read '${message}' from ${input}")  
 }  
}  
  
task producer(type: Producer)  
task consumer(type: Consumer)  
  
*// Wire property from producer to consumer task*  
consumer.inputFile = producer.outputFile  
  
*// Set values for the producer lazily*  
*// Note that the consumer does not need to be changed again.*  
producer.outputFile = layout.buildDirectory.file('file.txt')  
  
*// Change the base output directory.*  
*// Note that this automatically changes producer.outputFile and consumer.inputFile*  
buildDir = 'output'

**Output of** gradle consumer

> 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

In the example above, the task outputs and inputs are connected before any location is defined. This is possible because the input and output properties use the Provider API. The output property is created with [DefaultTask.newOutputFile()](http://docs.google.com/javadoc/org/gradle/api/DefaultTask.html#newOutputFile--) and the input property is created with [DefaultTask.newInputFile()](http://docs.google.com/javadoc/org/gradle/api/DefaultTask.html#newInputFile--). Values are only resolved when they are needed during execution. 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 thing to note is the absence of any explicit task dependency. Properties created via newOutputFile() and newOutputDirectory() bring knowledge about which task is generating them, so using them as task input will implicitly link tasks together.

[Working with collection Providers](#1if6iqy)

In this section, we are going to explore lazy collections. They work exactly like any other Provider and, just like FileSystemLocation providers, they have additional modeling around them. There are two provider interfaces available, one for List values and another for Set values:

* For List values the interface is called [ListProperty](http://docs.google.com/javadoc/org/gradle/api/provider/ListProperty.html). You can create a new ListProperty using [ObjectFactory.listProperty(java.lang.Class)](http://docs.google.com/javadoc/org/gradle/api/model/ObjectFactory.html#listProperty-java.lang.Class-) and specifying the element’s type.
* For Set values the interface is called [SetProperty](http://docs.google.com/javadoc/org/gradle/api/provider/SetProperty.html). You can create a new SetProperty using [ObjectFactory.setProperty(java.lang.Class)](http://docs.google.com/javadoc/org/gradle/api/model/ObjectFactory.html#setProperty-java.lang.Class-) and specifying the element’s type.

This type of property allows you to overwrite the entire collection value with [HasMultipleValues.set(java.lang.Iterable)](http://docs.google.com/javadoc/org/gradle/api/provider/HasMultipleValues.html#set-java.lang.Iterable-) and [HasMultipleValues.set(org.gradle.api.provider.Provider)](http://docs.google.com/javadoc/org/gradle/api/provider/HasMultipleValues.html#set-org.gradle.api.provider.Provider-) or add new elements through the various add methods:

* [HasMultipleValues.add(T)](http://docs.google.com/javadoc/org/gradle/api/provider/HasMultipleValues.html#add-T-): Add a single concrete element to the collection
* [HasMultipleValues.add(org.gradle.api.provider.Provider)](http://docs.google.com/javadoc/org/gradle/api/provider/HasMultipleValues.html#add-org.gradle.api.provider.Provider-): Add a lazily evaluated element to the collection
* [HasMultipleValues.addAll(org.gradle.api.provider.Provider)](http://docs.google.com/javadoc/org/gradle/api/provider/HasMultipleValues.html#addAll-org.gradle.api.provider.Provider-): Add a lazily evaluated collection of elements to the list

Just like every Provider, the collection is calculated when [Provider.get()](http://docs.google.com/javadoc/org/gradle/api/provider/Provider.html#get--) is called. The following example show the [ListProperty](http://docs.google.com/javadoc/org/gradle/api/provider/ListProperty.html) in action:

[Example: List property](#42eu1er)

**build.gradle**

task print {  
 doLast {  
 ListProperty<String> list = project.objects.listProperty(String)  
  
 *// Resolve the list*  
 logger.quiet('The list contains: ' + list.get())  
  
 *// Add elements to the empty list*  
 list.add(project.provider { 'element-1' }) *// Add a provider element*  
 list.add('element-2') *// Add a concrete element*  
  
 *// Resolve the list*  
 logger.quiet('The list contains: ' + list.get())  
  
 *// Overwrite the entire list with a new list*  
 list.set(['element-3', 'element-4'])  
  
 *// Resolve the list*  
 logger.quiet('The list contains: ' + list.get())  
  
 *// Add more elements through a list provider*  
 list.addAll(project.provider { ['element-5', 'element-6'] })  
  
 *// Resolve the list*  
 logger.quiet('The list contains: ' + list.get())  
 }  
}

**Output of** gradle print

> gradle print  
  
> Task :print  
The list contains: []  
The list contains: [element-1, element-2]  
The list contains: [element-3, element-4]  
The list contains: [element-3, element-4, element-5, element-6]  
  
BUILD SUCCESSFUL in 0s  
1 actionable task: 1 executed

[Guidelines](#2hk4bmk)

This section will introduce guidelines to be successful with the Provider API. To see those guidelines in action, have a look at [gradle-site-plugin](https://github.com/gradle-guides/gradle-site-plugin), a Gradle plugin demonstrating established techniques and practices for plugin development.

* The [Property](http://docs.google.com/javadoc/org/gradle/api/provider/Property.html) and [Provider](http://docs.google.com/javadoc/org/gradle/api/provider/Provider.html) 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](http://docs.google.com/javadoc/org/gradle/api/provider/Property.html) directly through a single getter.
  + For non-configurable properties, expose an [Provider](http://docs.google.com/javadoc/org/gradle/api/provider/Provider.html) directly through a single getter.
* Avoid simplifying calls like obj.getProperty().get() and obj.getProperty().set(T) in your code by introducing additional getters and setters.
* When migrating your plugin to use providers, follow these guidelines:
  + If it’s a new property, expose it as a [Property](http://docs.google.com/javadoc/org/gradle/api/provider/Property.html) or [Provider](http://docs.google.com/javadoc/org/gradle/api/provider/Provider.html) using a single getter.
  + If it’s incubating, change it to use a [Property](http://docs.google.com/javadoc/org/gradle/api/provider/Property.html) or [Provider](http://docs.google.com/javadoc/org/gradle/api/provider/Provider.html) using a single getter.
  + If it’s a stable property, add a new [Property](http://docs.google.com/javadoc/org/gradle/api/provider/Property.html) or [Provider](http://docs.google.com/javadoc/org/gradle/api/provider/Provider.html) and deprecate the old one. You should wire the old getter/setters into the new property as appropriate.

[Future development](#wpelud)

Going forward, new properties will use the Provider API. The Groovy Gradle DSL adds convenience methods to make the use of Providers mostly transparent in build scripts. Existing tasks will have their existing "raw" properties replaced by Providers as needed and in a backwards compatible way. New tasks will be designed with the Provider API.

The Provider API is [incubating](#2xcytpi). Please create new issues at [gradle/gradle](https://github.com/gradle/gradle/issues/new) to report bugs or to submit use cases for new features.

[Provider Files API Reference](#4apg2cb)

Use these types for *read-only* values:

[Provider](http://docs.google.com/javadoc/org/gradle/api/provider/Provider.html)<[RegularFile](http://docs.google.com/javadoc/org/gradle/api/file/RegularFile.html)>

File on disk

Factories

* [ProjectLayout.fileProperty()](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#fileProperty--)
* [DirectoryProperty.file(java.lang.String)](http://docs.google.com/javadoc/org/gradle/api/file/DirectoryProperty.html#file-java.lang.String-)

[Provider](http://docs.google.com/javadoc/org/gradle/api/provider/Provider.html)<[Directory](http://docs.google.com/javadoc/org/gradle/api/file/Directory.html)>

Directory on disk

Factories

* [ProjectLayout.directoryProperty()](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#directoryProperty--)
* [DirectoryProperty.dir(java.lang.String)](http://docs.google.com/javadoc/org/gradle/api/file/DirectoryProperty.html#dir-java.lang.String-)

[FileCollection](http://docs.google.com/javadoc/org/gradle/api/file/FileCollection.html)

Unstructured collection of files

Factories

* [Project.files(java.lang.Object[])](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:files(java.lang.Object%5B%5D))
* [ProjectLayout.files(java.lang.Object...)](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#files-java.lang.Object...-)

[FileTree](http://docs.google.com/javadoc/org/gradle/api/file/FileTree.html)

Hierarchy of files

Factories

* [Project.fileTree(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:fileTree(java.lang.Object)) will produce a [ConfigurableFileTree](http://docs.google.com/javadoc/org/gradle/api/file/ConfigurableFileTree.html), or you can use [Project.zipTree(Object)](http://docs.google.com/javadoc/org/gradle/api/Project.html#zipTree-java.lang.Object-) and [Project.tarTree(Object)](http://docs.google.com/javadoc/org/gradle/api/Project.html#tarTree-java.lang.Object-)

[Property Files API Reference](#3gp24i6)

Use these types for *mutable* values:

[RegularFileProperty](http://docs.google.com/javadoc/org/gradle/api/file/RegularFileProperty.html)

File on disk

Factories

* [DefaultTask.newInputFile()](http://docs.google.com/javadoc/org/gradle/api/DefaultTask.html#newInputFile--) and [DefaultTask.newOutputFile()](http://docs.google.com/javadoc/org/gradle/api/DefaultTask.html#newOutputFile--) if used as task input/output
* [Directory.file(java.lang.String)](http://docs.google.com/javadoc/org/gradle/api/file/Directory.html#file-java.lang.String-) otherwise

[DirectoryProperty](http://docs.google.com/javadoc/org/gradle/api/file/DirectoryProperty.html)

Directory on disk

Factories

* [DefaultTask.newInputDirectory()](http://docs.google.com/javadoc/org/gradle/api/DefaultTask.html#newInputDirectory--) and [DefaultTask.newOutputDirectory()](http://docs.google.com/javadoc/org/gradle/api/DefaultTask.html#newOutputDirectory--) if used as task input/output
* [Directory.dir(java.lang.String)](http://docs.google.com/javadoc/org/gradle/api/file/Directory.html#dir-java.lang.String-) otherwise

[ConfigurableFileCollection](http://docs.google.com/javadoc/org/gradle/api/file/ConfigurableFileCollection.html)

Unstructured collection of files

Factories

* [ProjectLayout.configurableFiles(java.lang.Object...)](http://docs.google.com/javadoc/org/gradle/api/file/ProjectLayout.html#configurableFiles-java.lang.Object...-)

[ConfigurableFileTree](http://docs.google.com/javadoc/org/gradle/api/file/ConfigurableFileTree.html)

Hierarchy of files

Factories

* [Project.fileTree(java.lang.Object)](http://docs.google.com/dsl/org.gradle.api.Project.html#org.gradle.api.Project:fileTree(java.lang.Object))

[Lazy Collections API Reference](#1vucepz)

* For lists, use [ObjectFactory.listProperty(java.lang.Class)](http://docs.google.com/javadoc/org/gradle/api/model/ObjectFactory.html#listProperty-java.lang.Class-) to get a [ListProperty](http://docs.google.com/javadoc/org/gradle/api/provider/ListProperty.html) which is also a Provider<List<T>>
* For sets, use [ObjectFactory.setProperty(java.lang.Class)](http://docs.google.com/javadoc/org/gradle/api/model/ObjectFactory.html#setProperty-java.lang.Class-) to get a [SetProperty](http://docs.google.com/javadoc/org/gradle/api/provider/SetProperty.html) which is a Provider<Set<T>>

[Lazy Objects API Reference](#4ftzxds)

Use [ObjectFactory.property(java.lang.Class)](http://docs.google.com/javadoc/org/gradle/api/model/ObjectFactory.html#property-java.lang.Class-) to construct a Property<T> which is a Provider<T>.

[Licenses](#1ljsd9k)

[Documentation licenses](#45jfvxd)

[Gradle Documentation](#2koq656)

*Copyright © 2007-2018 Gradle, Inc.*

Gradle build tool source code is open and licensed under the [Apache License 2.0](https://github.com/gradle/gradle/blob/master/LICENSE). Gradle user manual and DSL references are licensed under [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](http://creativecommons.org/licenses/by-nc-sa/4.0/).

[1](#2uza7ll). At <http://www.defmacro.org/ramblings/lisp.html> you find an interesting article comparing Ant, XML, Java and Lisp. It’s funny that the 'if Java had that syntax' syntax in this article is actually the Groovy syntax.

[2](#1a4khte). Gradle supports partial multi-project builds (see [Authoring Multi-Project Builds](#2p2csry)).

[3](#3u480h7). There are command line switches to change this behavior. See [Command-Line Interface](#lnxbz9))

[4](#299iap0). The real use case we had, was using <http://lucene.apache.org/solr>, where you need a separate war for each index you are accessing. That was one reason why we have created a distribution of webapps. The Resin servlet container allows us, to let such a distribution point to a base installation of the servlet container.

[5](#oeskwt). We do this here, as it makes the layout a bit easier. We usually put the project specific stuff into the build script of the respective projects.

[6](#38eg3km). 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](#gna8r6)).

[7](#1njqdsf). Any language element except for statement labels.

[8](#47jdwg8). It is planned for a future release to provide out-of-the-box support for this

[9](#2moo6o1). Gradle uses the same conventions as introduced by Russel Winder’s [Gant tool](https://gant.github.io/).

[10](#11tygvu). In fact, any artifact added to the archives configuration will be built by assemble

[11](#3ltlzjn). For more details on gradle.properties see [Gradle configuration properties](#1idq7dh)

[12](#20yw9rg). The JUnit wiki contains a detailed description on how to work with JUnit categories: <https://github.com/junit-team/junit/wiki/Categories>.

[13](#4kyjsf9). The TestNG documentation contains more details about test groups: <http://testng.org/doc/documentation-main.html#test-groups>.

[14](#303u2n2). The TestNG documentation contains more details about test ordering when working with testng.xml files: <http://testng.org/doc/documentation-main.html#testng-xml>.

[15](#1f94cuv). In Groovy you can execute Strings. To learn more about executing external processes with Groovy have a look in 'Groovy in Action' 9.3.2 or at the Groovy wiki

[16](#3z8rvio). The runtime configuration extends the compile configuration.

[17](#2ee25qh). See [Automatic configuration of Scala classpath](#1pyvb9t).

Docs

* [User Manual](http://docs.google.com/userguide/userguide.html)
* [DSL Reference](http://docs.google.com/dsl/)
* [Release Notes](http://docs.google.com/release-notes.html)
* [Javadoc](http://docs.google.com/javadoc/)

News

* [Blog](https://blog.gradle.org/)
* [Newsletter](https://newsletter.gradle.com/)
* [Twitter](https://twitter.com/gradle)

Products

* [Build Scans](https://gradle.com/build-scans)
* [Build Cache](https://gradle.com/build-cache)
* [Enterprise Docs](https://gradle.com/enterprise/resources)

Get Help

* [Forums](https://discuss.gradle.org/c/help-discuss)
* [GitHub](https://github.com/gradle/)
* [Training](https://gradle.org/training/)
* [Services](https://gradle.org/services/)

Subscribe for important Gradle updates and news

Subscribe

By entering your email, you agree to our [Terms](https://gradle.org/terms/) and [Privacy Policy](https://gradle.org/privacy/), including receipt of emails. You can unsubscribe at any time.

© [Gradle Inc.](https://gradle.com) 2018 All rights reserved.

[Careers](https://gradle.com/careers) | [Privacy](https://gradle.org/privacy) | [Terms of Service](https://gradle.org/terms) | [Contact](https://gradle.org/contact/)