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/)

# Using Gradle Plugins

Contents

[What plugins do](#4d34og8)

[Types of plugins](#2s8eyo1)

[Using plugins](#17dp8vu)

[Script plugins](#3rdcrjn)

[Binary plugins](#26in1rg)

[Finding community plugins](#lnxbz9)

[More on plugins](#35nkun2)

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](#4d34og8)

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](#2s8eyo1)

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](#17dp8vu)

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](#1ksv4uv) 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](#3rdcrjn)

[Example: Applying a script plugin](#44sinio)

**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](#26in1rg)

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](#1ksv4uv) or the [buildscript block.](#2jxsxqh)

[Locations of binary plugins](#z337ya)

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](#3j2qqm3) using the plugins DSL (see [Applying plugins using the plugins DSL](#1ksv4uv)).
* Including the plugin from an external jar defined as a buildscript dependency (see see [Applying plugins using the buildscript block](#1y810tw)).
* Defining the plugin as a source file under the buildSrc directory in the project (see [Using buildSrc to extract functional logic](http://docs.google.com/organizing_gradle_projects.html#sec:build_sources)).
* Defining the plugin as an inline class declaration inside a build script.

For more on defining your own plugins, see [Custom Plugins](http://docs.google.com/custom_plugins.html#custom_plugins).

[Applying plugins with the plugins DSL](#1ksv4uv)

| **✨** | The plugins DSL is currently [incubating](http://docs.google.com/feature_lifecycle.html#feature_lifecycle). 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](#4i7ojhp)

**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](#2xcytpi)

**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](#1ci93xb)

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](#3whwml4)

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](#2bn6wsx)

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](#1y810tw).

[Applying plugins to subprojects](#qsh70q)

If you have a [multi-project build](http://docs.google.com/multi_project_builds.html#multi_project_builds), 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.](#3as4poj)

**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](#1pxezwc)

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](#49x2ik5)

| **✨** | The pluginManagement {} DSL is currently [incubating](http://docs.google.com/feature_lifecycle.html#feature_lifecycle). 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](http://docs.google.com/init_scripts.html#init_scripts).

[Example: Configuring pluginManagement per-project and globally](#2p2csry)

**settings.gradle**

pluginManagement {  
 resolutionStrategy {  
 }  
 repositories {  
 }  
}

**init.gradle**

settingsEvaluated { settings ->  
 settings.pluginManagement {  
 resolutionStrategy {  
 }  
 repositories {  
 }  
 }  
}

[Custom Plugin Repositories](#3j2qqm3)

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.](#147n2zr)

**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](#3o7alnk)

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.](#23ckvvd)

**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](#ihv636) in addition to the artifacts which actually implement the plugin. For more information on publishing plugins to custom repositories read [Gradle Plugin Development Plugin](http://docs.google.com/java_gradle_plugin.html#java_gradle_plugin).

See [PluginManagementSpec](http://docs.google.com/javadoc/org/gradle/plugin/management/PluginManagementSpec.html) for complete documentation for using the pluginManagement {} block.

[Plugin Marker Artifacts](#ihv636)

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](http://docs.google.com/java_gradle_plugin.html#java_gradle_plugin).

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](http://docs.google.com/java_gradle_plugin.html#java_gradle_plugin), the [maven-publish](http://docs.google.com/publishing_maven.html#publishing_maven) plugin, and the [ivy-publish](http://docs.google.com/publishing_ivy.html#publishing_ivy) plugin.

[Example: Complete Plugin Publishing Sample](#32hioqz)

**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](#2jxsxqh)

With the introduction of the [plugins DSL](#1ksv4uv), 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](#1hmsyys)

[Example: Applying a binary plugin](#41mghml)

**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](#2grqrue)

**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](http://docs.google.com/writing_build_scripts.html#script-default-imports)). 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](#1y810tw)

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](http://docs.google.com/tutorial_using_tasks.html#sec:build_script_external_dependencies).

[Example: Applying a plugin with the buildscript block](#vx1227)

**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](#lnxbz9)

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](#35nkun2)

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](http://docs.google.com/custom_plugins.html#custom_plugins).

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/)