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/)

# Testing in Java & JVM projects

Contents

[The basics](#4d34og8)

[Test execution](#2s8eyo1)

[Test filtering](#17dp8vu)

[Test reporting](#3rdcrjn)

[Test detection](#26in1rg)

[Test grouping](#lnxbz9)

[Using JUnit 5](#35nkun2)

[Test execution order in TestNG](#1ksv4uv)

[Configuring integration tests](#44sinio)

[Skipping the tests](#2jxsxqh)

[Forcing tests to run](#z337ya)

[Debugging when running tests](#3j2qqm3)

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](#2s8eyo1))
* How to select specific tests to run ([Test filtering](#17dp8vu))
* What test reports are generated and how to influence the process ([Test reporting](#3rdcrjn))
* How Gradle finds tests to run ([Test detection](#26in1rg))
* How to make use of the major frameworks' mechanisms for grouping tests together ([Test grouping](#lnxbz9))

But first, we look at the basics of JVM testing in Gradle.

[The basics](#4d34og8)

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](http://docs.google.com/java_plugin.html#java_plugin) — you will automatically get the following:

* A dedicated test source set for unit tests
* A test task of type Test that runs those unit tests

The JVM language plugins use the source set to configure the task with the appropriate execution classpath and the directory containing the compiled test classes. In addition, they attach the test task to the check [lifecycle task](http://docs.google.com/more_about_tasks.html#sec:lifecycle_tasks).

It’s also worth bearing in mind that the test source set automatically creates [corresponding dependency configurations](http://docs.google.com/java_plugin.html#java_source_set_configurations) — of which the most useful are testImplementation and testRuntimeOnly — that the plugins tie into the test task’s classpath.

All you need to do in most cases is configure the appropriate compilation and runtime dependencies and add any necessary configuration to the test task. The following example shows a simple setup that uses JUnit 4.x and changes the maximum heap size for the tests' JVM to 1 gigabyte:

[Example: A basic configuration for the 'test' task](#1y810tw)

**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](#44sinio).

[Test execution](#2s8eyo1)

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.

| **✨** | 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](#17dp8vu)

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](#4i7ojhp)

**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.

| **✨** | 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](#2xcytpi)

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](#1ci93xb)

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](http://docs.google.com/command_line_interface.html#sec:continuous_build) to re-execute a subset of tests immediately after every change to a production or test source file. The following executes all tests in the 'com.mypackage.foo' package or subpackages whenever a change triggers the tests to run:

gradle test --continuous --tests "com.mypackage.foo.\*"

[Single test execution via System Properties](#3whwml4)

| **✨** | 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](#3rdcrjn)

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](http://docs.google.com/java_plugin.html#sec:java_convention_properties). For example, you can change the destination of the test results and reports like so:

[Example: Changing the default test report and results directories](#2bn6wsx)

**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](#qsh70q)

**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](#26in1rg)

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.

| **✨** | 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](#lnxbz9)

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.[[1](#3as4poj)] [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](#1pxezwc)

**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](#49x2ik5)

**build.gradle**

test {  
 useJUnitPlatform {  
 includeTags 'fast'  
 excludeTags 'slow'  
 }  
}

The TestNG framework uses the concept of test groups for a similar effect.[[2](#2p2csry)] 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](#147n2zr)

**build.gradle**

test {  
 useTestNG {  
 excludeGroups 'integrationTests'  
 includeGroups 'unitTests'  
 }  
}

[Using JUnit 5](#35nkun2)

[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](#3o7alnk)

**build.gradle**

test {  
 useJUnitPlatform()  
}

See [Test.useJUnitPlatform()](http://docs.google.com/javadoc/org/gradle/api/tasks/testing/Test.html#useJUnitPlatform) for more details.

| **✨** | 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](#23ckvvd)

To enable JUnit Jupiter support in Gradle, all you need to do is add the following dependencies:

[Example: JUnit Jupiter dependencies](#ihv636)

**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](#32hioqz)

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](#1hmsyys)

**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](#41mghml)

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](#2grqrue)

**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](#1ksv4uv)

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.[[3](#vx1227)] 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](#3fwokq0)

**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](#1v1yuxt)

**build.gradle**

test {  
 useTestNG {  
 groupByInstances = true  
 }  
}

[TestNG parameterized methods and reporting](#4f1mdlm)

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](#44sinio)

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](http://docs.google.com/building_java_projects.html#sec:java_source_sets) 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](#2u6wntf)

**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](http://docs.google.com/java_plugin.html#java_source_set_configurations) 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](http://docs.google.com/working_with_files.html#sec:file_collections) 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](#19c6y18)

**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](#2jxsxqh)

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](http://docs.google.com/command_line_interface.html#sec:excluding_tasks_from_the_command_line) or [in the build script](http://docs.google.com/more_about_tasks.html#sec:skipping_tasks). 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](#3tbugp1)

**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](#z337ya)

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](http://docs.google.com/more_about_tasks.html#sec:task_rules) provided by the [Base Plugin](http://docs.google.com/base_plugin.html#sec:base_tasks). You can use it for *any* task.

[Debugging when running tests](#3j2qqm3)

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.

[1](#28h4qwu). The JUnit wiki contains a detailed description on how to work with JUnit categories: <https://github.com/junit-team/junit/wiki/Categories>.

[2](#nmf14n). The TestNG documentation contains more details about test groups: <http://testng.org/doc/documentation-main.html#test-groups>.

[3](#37m2jsg). The TestNG documentation contains more details about test ordering when working with testng.xml files: <http://testng.org/doc/documentation-main.html#testng-xml>.

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/)