Testing

**Fundamentals of Testing**

Users interact with your app on a variety of levels, from pressing a **Submit** button to downloading information onto their device. Accordingly, you should test a variety of use cases and interactions as you iteratively develop your app.

**Testing Pyramid**

The Testing Pyramid, shown in Figure 2, illustrates how your app should include the three categories of tests: small, medium, and large:

A close up of a sign

Description automatically generated

Because of the different characteristics of each test category, you should include tests from each layer of the test pyramid. Although the proportion of tests for each category can vary based on your app's use cases, we generally recommend the following split among the categories: **70 percent unit tests, 20 percent integration tests, and 10 percent UI tests**.

**Unit Tests**

* Unit tests are small tests that you can run in isolation from production systems. They typically mock every major component and should run quickly on your machine.
* As you add and change your app's functionality, make sure that these features behave as intended by creating and running unit tests against them.
* Although it's possible to evaluate units on a device or emulator, it's usually quicker and easier to test the units in your development environment, adding stubbed or mocked methods as needed to interact with the Android system.

**Integration Tests**

* Integration tests are medium tests that sit in between unit tests and User Interface tests. They integrate several components, and they run on emulators or real devices.
* After you've tested each unit of your app within your development environment, you should verify that the components behave properly when run on an emulator or device. Integration tests allow you to complete this part of the development process.
* These tests are particularly important to create and run if some of your app's components depend on physical hardware.
* Integration tests evaluate how your app coordinates multiple units, but they don't test the full app.
* Examples of integration tests include service tests, integration tests, and hermetic UI tests that simulate the behavior of external dependencies.
* Typically, it's better to test your app on an emulated device or a cloud-based service like [Firebase Test Lab](https://firebase.google.com/docs/test-lab/), rather than on a physical device, as you can test multiple combinations of screen sizes and hardware configurations more easily and quickly.

**User Interface (UI) Tests**

* Integration and UI are large tests that run by completing a UI workflow.
* They ensure that key end-user tasks work as expected on emulators or real devices.
* Although it's important to test each layer and feature within your app in isolation, it's just as important to test common workflows and use cases that involve the complete stack, from the UI through business logic to the data layer.
* If your app is small enough, you might need only one suite of large tests to evaluate your app's functionality as a whole. Otherwise, you should divide your large test suites by team ownership, functional verticals, or user goals.

**Note:** For each large UI or integration test, you should also write medium integration tests that check the functionality of each UI component included in the workflow. That way, your test suite can continue to identify potential issues within each step of a critical user journey, even when the corresponding UI or integration test keeps failing during one of the first few steps.

This information was found here: <https://developer.android.com/training/testing/fundamentals#large-tests>

Robolectric - unit test framework

Robolectric is a framework that brings fast and reliable unit tests to Android. Tests run inside the JVM on your workstation in seconds. This is the framework that we will be using for unit testing.

**Test APIs & Isolation**

Unlike traditional emulator-based Android tests, Robolectric tests run inside a sandbox which allows the Android environment to be precisely configured to the desired conditions for each test, isolates each test from its neighbors, and extends the Android framework with test APIs which provide minute control over the Android framework’s behavior and visibility of state for assertions.

While much of the Android framework will work as expected inside a Robolectric test, some Android components’ regular behavior doesn’t translate well to unit tests: hardware sensors need to be simulated, system services need to be loaded with test fixture data. In those cases, Robolectric provides a [test double](https://en.wikipedia.org/wiki/Test_double) that’s suitable for most unit testing scenarios.

You can find documentation for Robolectric’s test APIs by installing the [Robolectric plugin for Chrome](https://chrome.google.com/webstore/detail/pjepcinimnfnaoopahdkpkefnefdkdgh) and visiting the [Android API Reference](https://developer.android.com/reference/packages).

**Run Tests Outside of the Emulator**

Robolectric lets you run your tests on your workstation, or on your continuous integration environment in a regular JVM, without an emulator. Because of this, the dexing, packaging, and installing-on-the emulator steps aren’t necessary, reducing test cycles from minutes to seconds so you can iterate quickly and refactor your code with confidence.

**SDK, Resources, & Native Method Simulation**

Robolectric handles inflation of views, resource loading, and lots of other stuff that’s implemented in native C code on Android devices. This allows tests to do most things you could do on a real device. It’s easy to provide your own implementation for specific SDK methods too, so you could simulate error conditions or real-world sensor behavior, for example.

**No Mocking Frameworks Required**

An alternate approach to Robolectric is to use mock frameworks such as [Mockito](http://code.google.com/p/mockito/) or to mock out the Android SDK. While this is a valid approach, it often yields tests that are essentially reverse implementations of the application code.

Robolectric allows a test style that is closer to black box testing, making the tests more effective for refactoring and allowing the tests to focus on the behavior of the application instead of the implementation of Android. You can still use a mocking framework along with Robolectric if you like.

This information was found here: <http://robolectric.org/>

Tutorials:

* <http://www.vogella.com/tutorials/Robolectric/article.html>
* <https://guides.codepath.com/android/unit-testing-with-robolectric>
* <http://robolectric.org/writing-a-test/>
* <https://medium.com/android-testing-daily/integration-testing-with-robolectric-2482b9f2c91f>
* <https://antonioleiva.com/android-unit-testing-using-robolectric-introduction/>

# Testing with Roboelectric

Robolectric works best with Android Studio and [Android Gradle Plugin 3.2.1](https://developer.android.com/studio/releases/gradle-plugin#updating-plugin) or newer.

## Annotate your test with the Robolectric test runner

               @RunWith(RobolectricTestRunner.class)

               public class SandwichTest {

                }

This information was found here: <http://robolectric.org/getting-started/>

Appium

We'll be using Appium with Kotlin for our dynamic code analysis.

Introduction

Appium is an open-source tool for automating native, mobile web, and hybrid applications on iOS mobile, Android mobile, and Windows desktop platforms. **Native apps** are those written using the iOS, Android, or Windows SDKs. **Mobile web apps** are web apps accessed using a mobile browser (Appium supports Safari on iOS and Chrome or the built-in 'Browser' app on Android). **Hybrid apps** have a wrapper around a "webview" -- a native control that enables interaction with web content. Projects like [Apache Cordova](https://cordova.apache.org/) or [Phonegap](http://phonegap.com/) make it easy to build apps using web technologies that are then bundled into a native wrapper, creating a hybrid app.

Importantly, Appium is "cross-platform": it allows you to write tests against multiple platforms (iOS, Android, Windows), using the same API. This enables code reuse between iOS, Android, and Windows testsuites.

For specific information about what it means for Appium to "support" its platforms, and automation modalities, please see the [platform support doc](http://appium.io/docs/en/about-appium/platform-support/index.html).

Appium Philosophy

Appium was designed to meet mobile automation needs according to a philosophy outlined by the following four tenets:

1. You shouldn't have to recompile your app or modify it in any way in order to automate it.
2. You shouldn't be locked into a specific language or framework to write and run your tests.
3. A mobile automation framework shouldn't reinvent the wheel when it comes to automation APIs.
4. A mobile automation framework should be open source, in spirit and practice as well as in name!

Appium Design

So how does the structure of the Appium project live out this philosophy? We meet requirement #1 by using vendor-provided automation frameworks under the hood. That way, we don't need to compile in any Appium-specific or third-party code or frameworks to your app. This means **you're testing the same app you're shipping**. The vendor-provided frameworks we use are:

* iOS 9.3 and above: Apple's [XCUITest](https://developer.apple.com/reference/xctest)
* iOS 9.3 and lower: Apple's [UIAutomation](https://web.archive.org/web/20160904214108/https:/developer.apple.com/library/ios/documentation/DeveloperTools/Reference/UIAutomationRef/)
* Android 4.2+: Google's [UiAutomator/UiAutomator2](https://developer.android.com/training/testing/ui-automator)
* Android 2.3+: Google's [Instrumentation](http://developer.android.com/reference/android/app/Instrumentation.html). (Instrumentation support is provided by bundling a separate project, [Selendroid](http://selendroid.io/))
* Windows: Microsoft's [WinAppDriver](http://github.com/microsoft/winappdriver)

We meet requirement #2 by wrapping the vendor-provided frameworks in one API, the [WebDriver](http://docs.seleniumhq.org/projects/webdriver/) API. WebDriver (aka "Selenium WebDriver") specifies a client-server protocol (known as the [JSON Wire Protocol](https://w3c.github.io/webdriver/webdriver-spec.html)). Given this client-server architecture, a client written in any language can be used to send the appropriate HTTP requests to the server. There are already [clients written in every popular programming language](http://appium.io/downloads). This also means that you're free to use whatever test runner and test framework you want; the client libraries are simply HTTP clients and can be mixed into your code any way you please. In other words, Appium & WebDriver clients are not technically "test frameworks" -- they are "automation libraries". You can manage your test environment any way you like!

We meet requirement #3 in the same way: WebDriver has become the de facto standard for automating web browsers, and is a [W3C Working Draft](https://dvcs.w3.org/hg/webdriver/raw-file/tip/webdriver-spec.html). Why do something totally different for mobile? Instead we have [extended the protocol](https://github.com/SeleniumHQ/mobile-spec/blob/master/spec-draft.md) with extra API methods useful for mobile automation.

It should be obvious that requirement #4 is a given -- you're reading this because [Appium is open source](https://github.com/appium/appium).

Appium Concepts

**Client/Server Architecture**  
Appium is at its heart a webserver that exposes a REST API. It receives connections from a client, listens for commands, executes those commands on a mobile device, and responds with an HTTP response representing the result of the command execution. The fact that we have a client/server architecture opens up a lot of possibilities: we can write our test code in any language that has a http client API, but it is easier to use one of the [Appium client libraries](http://appium.io/downloads). We can put the server on a different machine than our tests are running on. We can write test code and rely on a cloud service like [Sauce Labs](https://saucelabs.com/products/mobile-app-testing) to receive and interpret the commands.

**Session**  
Automation is always performed in the context of a session. Clients initiate a session with a server in ways specific to each library, but they all end up sending a POST /session request to the server, with a JSON object called the 'desired capabilities' object. At this point the server will start up the automation session and respond with a session ID which is used for sending further commands.

**Desired Capabilities**  
Desired capabilities are a set of keys and values (i.e., a map or hash) sent to the Appium server to tell the server what kind of automation session we're interested in starting up. There are also various capabilities which can modify the behavior of the server during automation. For example, we might set the platformNamecapability to iOS to tell Appium that we want an iOS session, rather than an Android or Windows one. Or we might set the safariAllowPopups capability to true in order to ensure that, during a Safari automation session, we're allowed to use JavaScript to open up new windows. See the [capabilities doc](http://appium.io/docs/en/writing-running-appium/caps/index.html) for the complete list of capabilities available for Appium.

**Appium Server**  
Appium is a server written in Node.js. It can be built and installed [from source](http://appium.io/docs/en/contributing-to-appium/appium-from-source/index.html) or installed directly from [NPM](https://www.npmjs.com/package/appium):

$ npm install -g appium

$ appium  
  
The beta of Appium is available via NPM with npm install -g appium@beta. It is the development version so it might have breaking changes. Please uninstall appium@beta (npm uninstall -g appium@beta) before installing new versions in order to have a clean set of dependencies.

**Appium Clients**  
There are client libraries (in Java, Ruby, Python, PHP, JavaScript, and C#) which support Appium's extensions to the WebDriver protocol. When using Appium, you want to use these client libraries instead of your regular WebDriver client. You can view the full list of libraries [here](http://appium.io/docs/en/about-appium/appium-clients/index.html).

[**Appium Desktop**](https://github.com/appium/appium-desktop)  
There is a GUI wrapper around the Appium server that can be downloaded for any platform. It comes bundled with everything required to run the Appium server, so you don't need to worry about Node. It also comes with an Inspector, which enables you to check out the hierarchy of your app. This can come in handy when writing tests.

This information can be found here: <http://appium.io/docs/en/about-appium/intro/?lang=en>

Detekt

Using Detekt for static code analysis. Detekt is the most popular static analysis tool for Kotlin, and one of the only. Checks different categories such as naming conventions and method complexity. Warns if there are at least 5 code smells. Fails to build if there 10 code smells.

<https://arturbosch.github.io/detekt/>

# Testing the App

Android Studio is designed to make testing simple. With just a few clicks, you can set up a JUnit test that runs on the local JVM or an instrumented test that runs on a device. Of course, you can also extend your test capabilities by integrating test frameworks such as [Mockito](https://github.com/mockito/mockito) to test Android API calls in your local unit tests, and [Espresso](https://developer.android.com/topic/libraries/testing-support-library/index.html#Espresso) or [UI Automator](https://developer.android.com/topic/libraries/testing-support-library/index.html#UIAutomator) to exercise user interaction in your instrumented tests. You can generate Espresso tests automatically using [Espresso Test Recorder](https://developer.android.com/studio/test/espresso-test-recorder.html).

This page provides the basic information about how to add new tests to your app and run them from Android Studio.

For a more complete how-to guide for writing tests, see [Testing apps on Android](https://developer.android.com/training/testing/index.html).

# **Test types and location**

The location of your test code depends on the type of test you are writing. Android Studio provides source code directories (source sets), for the following two types of tests:

## ****Local unit tests****

Located at module-name/src/test/java/.

These are tests that run on your machine's local Java Virtual Machine (JVM). Use these tests to minimize execution time when your tests have no Android framework dependencies or when you can mock the Android framework dependencies.

At runtime, these tests are executed against a modified version of android.jar where all final modifiers have been stripped off. This lets you use popular mocking libraries, like Mockito.

## ****Instrumented tests****

Located at module-name/src/androidTest/java/.

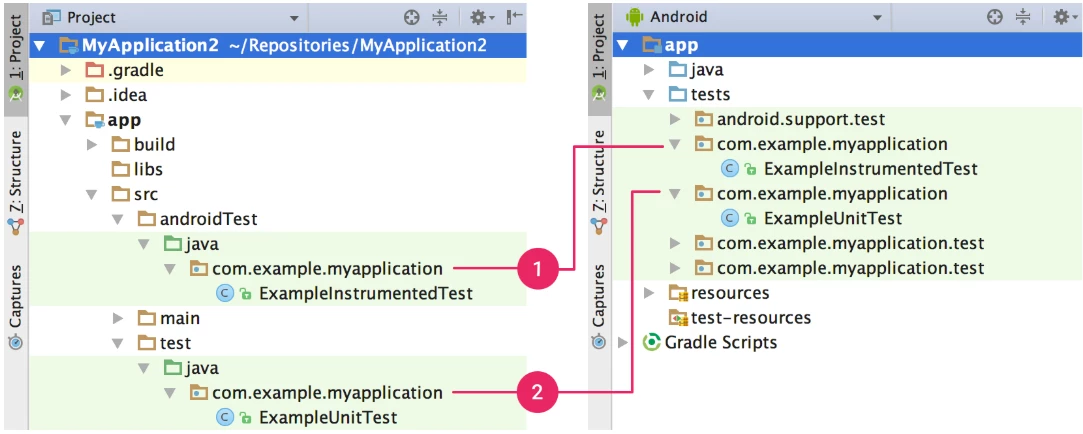
These are tests that run on a hardware device or emulator. These tests have access to [Instrumentation](https://developer.android.com/reference/android/app/Instrumentation.html) APIs, give you access to information such as the [Context](https://developer.android.com/reference/android/content/Context.html) of the app you are testing, and let you control the app under test from your test code. Use these tests when writing integration and functional UI tests to automate user interaction, or when your tests have Android dependencies that mock objects cannot satisfy.

Because instrumented tests are built into an APK (separate from your app APK), they must have their own [AndroidManifest.xml](https://developer.android.com/guide/topics/manifest/manifest-intro.html) file. However, Gradle automatically generates this file during the build so it is not visible in your project source set. You can add your own manifest file if necessary, such as to specify a different value for `minSdkVersion` or register run listeners just for your tests. When building your app, Gradle merges multiple manifest files into one manifest.

## ****File system****

The Gradle build interprets these test source sets in the same manner as it does for your project's [app source sets](https://developer.android.com/studio/build/index.html#sourcesets), which allows you to [create tests based on build variants](https://developer.android.com/studio/test/#create_instrumented_test_for_a_build_variant).

When you create a new project or add an app module, Android Studio creates the test source sets listed above and includes an example test file in each. You can see them in the **Project** window as shown in figure 1.



**Figure 1.** Your project's **(1)** instrumented tests and **(2)** local JVM tests are visible in either the **Project** view (left) or **Android** view (right).

# **Add a new test**

To create either a local unit test or an instrumented test, you can create a new test for a specific class or method by following these steps:

1. Open the Java file containing the code you want to test.
2. Click the class or method you want to test, then press Ctrl+Shift+T (⇧⌘T).
3. In the menu that appears, click **Create New Test**.
4. In the **Create Test** dialog, edit any fields and select any methods to generate, and then click **OK**.
5. In the **Choose Destination Directory** dialog, click the source set corresponding to the type of test you want to create: **androidTest** for an instrumented test or **test** for a local unit test. Then click **OK**.

Alternatively, you can create a generic Java file in the appropriate test source set as follows:

1. In the **Project** window on the left, click the drop-down menu and select the **Project** view.
2. Expand the appropriate module folder and the nested **src** folder. To add a local unit test, expand the **test** folder and the nested **java** folder; to add an instrumented test, expand the **androidTest**folder and the nested **java** folder.
3. Right-click on the Java package directory and select **New > Java Class**.
4. Name the file and then click **OK**.

Also be sure you specify the test library dependencies in your app module's build.gradle file:

dependencies {  
    // Required for local unit tests (JUnit 4 framework)  
    testImplementation 'junit:junit:4.12'  
  
    // Required for instrumented tests  
    androidTestImplementation 'com.android.support:support-annotations:24.0.0'  
    androidTestImplementation 'com.android.support.test:runner:0.5'  
}

For other optional library dependencies and more information about how to write your tests, see [Building local unit tests](https://developer.android.com/training/testing/unit-testing/local-unit-tests.html) and [Building instrumented unit tests](https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests.html).

#### Resolve conflicts between the app and test APK

Both your app APK and its test APK share the same classpaths. So, you can get a build error if both APKs depend on different versions of the same library. If Gradle isn't able to detect this version conflict, your app may behave unexpectedly or crash during runtime.

To learn more, read [Fix dependency resolution errors](https://developer.android.com/studio/build/dependencies#resolution_errors).

## Create instrumented test for a build variant

If your project includes [build variants](https://developer.android.com/studio/build/build-variants.html) with unique source sets, then you might want corresponding instrumented test source sets. Creating instrumented tests in source sets that correspond to your build variants helps keep your test code organized and allows you to run only the tests that apply to a given build variant.

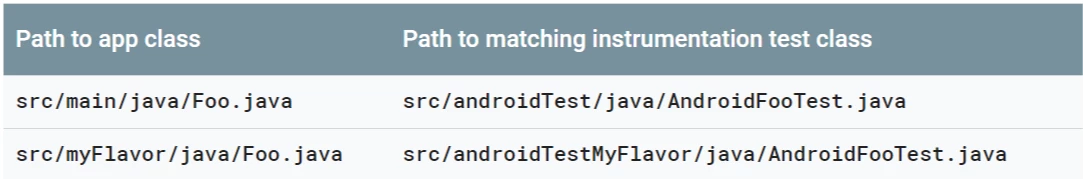
To add a testing source set for your build variant, follow these steps:

1. In the **Project** window on the left, click the drop-down menu and select the **Project** view.
2. Within the appropriate module folder, right-click the **src** folder and click **New > Directory**.
3. For the directory name, enter "androidTestVariantName." For example, if you have a build variant called "MyFlavor" then the directory name should be "androidTestMyFlavor." Then click **OK**.
4. Right-click on the new directory and click **New > Directory**.
5. Enter "java" as the directory name, and then click **OK**.

Now you can add tests to this new source set by following the steps above to add a new test. When you reach the **Choose Destination Directory** dialog, select the new variant test source set.

The instrumented tests in src/androidTest/ source set are shared by all build variants. When building a test APK for the "MyFlavor" variant of your app, Gradle combines both the src/androidTest/ and src/androidTestMyFlavor/ source sets.

For example, the following table shows how instrumentation test files should reside in source sets that correspond to the app's code source sets.



Just as it does for your app source sets, the Gradle build merges and overrides files from different test source sets. In this case, the AndroidFooTest.java file in the "androidTestMyFlavor" source set overrides the version in the "androidTest" source set. For more information about how source sets are merged, see [Configure your build](https://developer.android.com/studio/build/index.html#sourcesets).

Another reason you should use build variants for your app and test source sets is to create hermetic tests through mock dependencies. That is, you can create a product flavor for your app that contains fake implementations of dependencies (such as network requests or device sensor data that is ordinarily flaky), and then add a corresponding mock test source set. For more information, see the blog post about [leveraging product flavors for hermetic testing](http://android-developers.blogspot.com/2015/12/leveraging-product-%0Aflavors-in-android.html).

## Configure instrumentation manifest settings

When Gradle builds your test APK, it automatically generates the [AndroidManifest.xml](https://developer.android.com/guide/topics/manifest/manifest-intro.html) file and configures it with the [<instrumentation>](https://developer.android.com/guide/topics/manifest/instrumentation-element.html) node. One of the reasons Gradle configures this node for you is to make sure that the [targetPackage](https://developer.android.com/guide/topics/manifest/instrumentation-element.html" \l "trgt) property specifies the correct package name of the app under test. You can change some of the other settings for this node by either creating another manifest file in the test source set or configuring your module-level build.gradle file, as shown in the following code sample.

android {  
  ...  
  // Each product flavor you configure can override properties in the  
  // defaultConfig {} block. To learn more, go to [Configure product flavors](https://developer.android.com/studio/build/build-variants.html#product-flavors).  
  defaultConfig {  
    ...  
    // Specifies the [application ID](https://developer.android.com/studio/build/application-id.html) for the test APK.  
    [testApplicationId](http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.ProductFlavor.html" \l "com.android.build.gradle.internal.dsl.ProductFlavor:testApplicationId) "com.test.foo"  
    // Specifies the fully-qualified class name of the test instrumentation runner.  
    [testInstrumentationRunner](http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.ProductFlavor.html#com.android.build.gradle.internal.dsl.ProductFlavor:testInstrumentationRunner) "android.test.InstrumentationTestRunner"  
    // If set to 'true', enables the instrumentation class to start and stop profiling.  
    // If set to false (default), profiling occurs the entire time the instrumentation  
    // class is running.  
    [testHandleProfiling](http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.ProductFlavor.html" \l "com.android.build.gradle.internal.dsl.ProductFlavor:testHandleProfiling) true  
    // If set to 'true', indicates that the Android system should run the instrumentation  
    // class as a functional test. The default value is 'false'  
    [testFunctionalTest](http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.ProductFlavor.html" \l "com.android.build.gradle.internal.dsl.ProductFlavor:testFunctionalTest) true  
  }  
}  
...

## Change the test build type

By default, all tests run against the debug build type. You can change this to another build type by using the testBuildType property in your module-level build.gradle file. For example, if you want to run your tests against your "staging" build type, edit the file as shown in the following snippet.

android {  
    ...  
    testBuildType "staging"  
}

## Configure Gradle test options

The [Android plugin for Gradle](https://developer.android.com/studio/releases/gradle-plugin.html) allows you to specify certain options for all or just some of your unit tests. In the module-level build.gradle file, use the [testOptions {}](http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.TestOptions.html) block to specify options that change how Gradle runs all your tests.

android {  
  ...  
  // Encapsulates options for running tests.  
  testOptions {  
    // Changes the directory where Gradle saves test reports. By default, Gradle saves test reports  
    // in the path\_to\_your\_project/module\_name/build/outputs/reports/ directory.  
    // '$rootDir' sets the path relative to the root directory of the current project.  
    reportDir "$rootDir/test-reports"  
    // Changes the directory where Gradle saves test results. By default, Gradle saves test results  
    // in the path\_to\_your\_project/module\_name/build/outputs/test-results/ directory.  
    // '$rootDir' sets the path relative to the root directory of the current project.  
    resultsDir "$rootDir/test-results"  
  }  
}

To specify options for only local unit tests, configure the [unitTests {}](http://google.github.io/android-gradle-dsl/current/com.android.build.gradle.internal.dsl.TestOptions.UnitTestOptions.html) block inside testOptions {}.

android {  
  ...  
  testOptions {  
    ...  
    // Encapsulates options for local unit tests.  
    unitTests {  
      // By default, local unit tests throw an exception any time the code you are testing tries to access  
      // Android platform APIs (unless you [mock Android dependencies](https://developer.android.com/training/testing/unit-testing/local-unit-tests.html#mocking-dependencies) yourself or with a testing  
      // framework like Mockito). However, you can enable the following property so that the test  
      // returns either null or zero when accessing platform APIs, rather than throwing an exception.  
      returnDefaultValues true  
  
      // Encapsulates options for controlling how Gradle executes local unit tests. For a list  
      // of all the options you can specify, read [Gradle's reference documentation](https://docs.gradle.org/current/javadoc/org/gradle/api/tasks/testing/Test.html).  
      all {  
        // Sets JVM argument(s) for the test JVM(s).  
        jvmArgs '-XX:MaxPermSize=256m'  
  
        // You can also check the task name to apply options to only the tests you specify.  
        if (it.name == 'testDebugUnitTest') {  
          systemProperty 'debug', 'true'  
        }  
        ...  
      }  
    }  
  }  
}

## Use separate test modules for instrumented tests

If you want to have a module dedicated for instrumented tests and isolate the rest of your code from your tests, you can create a separate test module and configure its build similar to that of a library module. To create a test module, proceed as follows:

1. [Create a library module](https://developer.android.com/studio/projects/android-library#CreateLibrary).
2. In the [module level build file](https://developer.android.com/studio/build/#module-level), apply the com.android.test plugin instead of com.android.library.
3. [Sync your project](https://developer.android.com/studio/build/#sync-files).

After you create your test module, you can include your test code in the main or variant source set (for example, src/main/java or src/variant/java). If your app module defines multiple product flavors, you can recreate those flavors in your test module, and, using [variant aware dependency management](https://developer.android.com/studio/build/dependencies#variant_aware), the test module attempts to test the matching flavor in the target module.

By default, test modules contain and test only a debug variant. However, you can create new build types to match the tested app project. To make the test module test a different build type, and not the debug one, use VariantFilter to disable the debug variant in the test project, as shown below:

android {  
    variantFilter { variant ->  
        if (variant.buildType.name.equals('debug')) {  
            variant.setIgnore(true);  
        }  
    }  
}

If you want a test module to target only certain flavors or build types of an app, you can use the [matchingFallbacks](https://developer.android.com/studio/build/dependencies" \l "resolve_matching_errors) property to target only the variants you want to test. This also prevents the test module from having to configure those variants for itself.

## Run a test

To run a test, proceed as follows:

1. Be sure your project is synchronized with Gradle by clicking **Sync Project** https://developer.android.com/studio/images/buttons/toolbar-sync-gradle.png in the toolbar.
2. Run your test in one of the following ways:
   * In the **Project** window, right-click a test and click **Run** https://developer.android.com/studio/images/buttons/toolbar-run.png.
   * In the Code Editor, right-click a class or method in the test file and click **Run** https://developer.android.com/studio/images/buttons/toolbar-run.png to test all methods in the class.
   * To run all tests, right-click on the test directory and click **Run tests** https://developer.android.com/studio/images/buttons/toolbar-run.png.

By default, your test runs using Android Studio's default run configuration. If you'd like to change some run settings such as the instrumentation runner and deployment options, you can edit the run configuration in the **Run/Debug Configurations** dialog (click **Run > Edit Configurations**).

## View test coverage

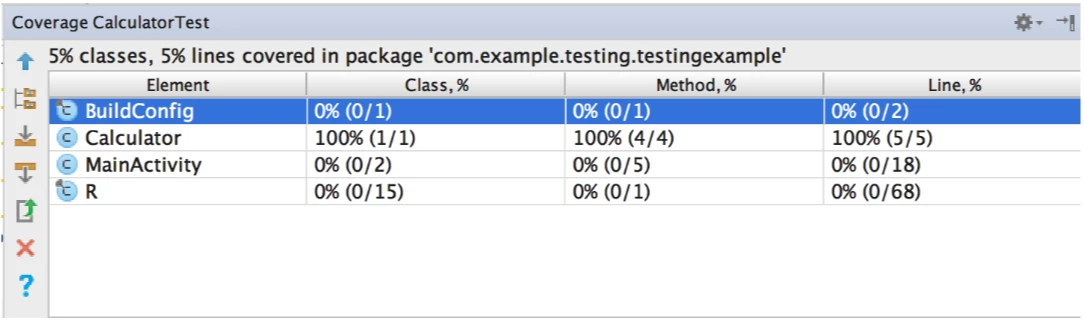
The test coverage tool is available for local unit tests to track the percentage and areas of your app code that your unit tests have covered. Use the test coverage tool to determine whether you have adequately tested the elements, classes, methods, and lines of code that make up your app.

There are a number of ways to run a unit test, and they are described on the IntelliJ [Running with Coverage](https://www.jetbrains.com/help/idea/2018.2/running-with-coverage.html) page. The following procedure shows how to run a unit test inline from the editor:

1. Double-click the unit test you want to run.
2. In the editor, place your cursor in the line you want to run with coverage.
   * If you place your cursor in the class declaration, all test methods in that class run.
   * If you place your cursor in a method declaration, all code in that method runs.
   * If you place your cursor on a particular line within a method, only that line runs.
3. Right-click the line where you placed your cursor.
4. In the context menu, choose **Run test-name with coverage**.

The [coverage tool window](https://www.jetbrains.com/help/idea/2018.2/coverage-tool-window.html) appears.

Figure 2 shows the coverage tool window for a calculator unit test that tests for addition, subtraction, multiplication, and division.

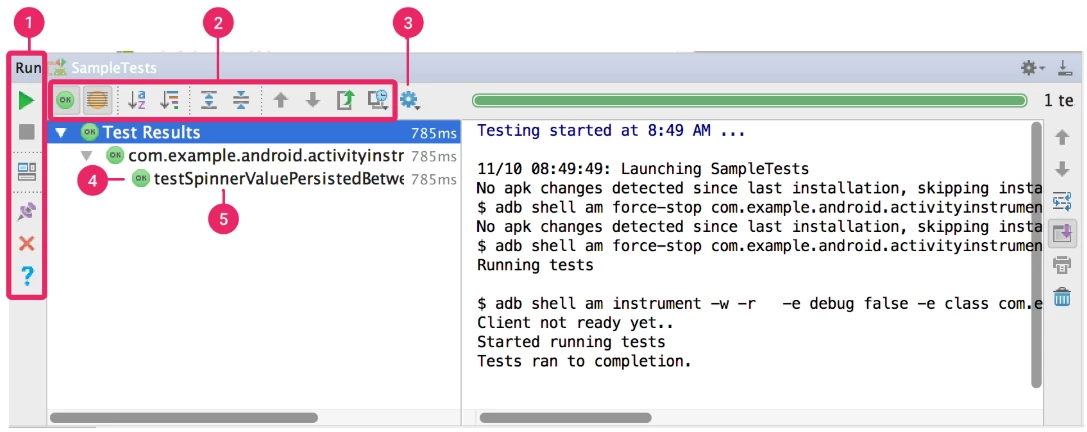


**Figure 2.** See the code coverage percentages for your application.

For more information about local unit tests, see [Building local unit tests](https://developer.android.com/training/testing/unit-testing/local-unit-tests.html).

## View the test results

When you run a JUnit or instrumented test, the results appear in the **Run** window. A green bar means all tests succeeded and a red bar means at least one test failed. Figure 3 shows a successful test run.



**Figure 3.** Test results appear in the Run window.

The **Run** window displays the tests in a tree view on the left, and the results and messages for the current test suite in the output pane on the right. Use the toolbars, context menus, and status icons to manage the test results, as follows:

1. Use the **run toolbar** to rerun the current test, stop the current test, rerun failed tests (not shown because it is available for unit tests only), pause output, and dump threads.
2. Use the **testing toolbar** to filter and sort test results. You can also expand or collapse nodes, show test coverage, and import or export test results.
3. Click the **context menu** https://developer.android.com/studio/images/buttons/runpane-cog-button.png to track the running test, show inline statistics, scroll to the stacktrace, open the source code at an exception, auto scroll to the source, and select the first failed test when the test run completes.
4. **Test status icons** indicate whether a test has an error, was ignored, failed, is in progress, has passed, is paused, was terminated, or was not run.
5. Right-click a line in the tree view to display a context menu that lets you run the tests in debug mode, open the test source code file, or jump to the line in the source code being tested.

For more information about the **Run** window, its toolbars, and context menus, see the IntelliJ page,[Test Runner Tab](https://www.jetbrains.com/help/idea/2018.2/test-runner-tab.html).

### View inline statistics

To find out how long your tests took to run, do the following:

1. Click the cog icon https://developer.android.com/studio/images/buttons/runpane-cog-button.png.
2. In the drop-down list, select **Show Inline Statistics**.

The elapsed time in milliseconds displays to the right of the tests.

### Compare strings

If a unit test contains assertEquals() failures from comparing two string objects, you can see what the differences are between the two string objects to find out what caused the failures, as follows:

1. In the output pane, click the **Click to see difference** link.
2. In the **Differences viewer**, explore the differences as described on the IntelliJ page, [Differences viewer for files](https://www.jetbrains.com/help/idea/2018.2/differences-viewer-for-files.html).

### Export test results

You can export your test results in XML or HTML format, as follows:

1. Click **Export Test Results** https://developer.android.com/studio/images/buttons/runpane-export-test-results.png.
2. In the **Export Test Results** dialog, provide format and output information, and click **OK**. The exported test results save to the specified folder.

### Import test results

You can import your exported test results, as follows:

1. Click **Import Test Results** https://developer.android.com/studio/images/buttons/runpane-import-test-results.png.
2. In the drop-down menu, select the file you want to import. The imported tests results appear in the **Run** window.

This information came from: <https://developer.android.com/studio/test/>