**BASIC OF TESTING:**

Testing your app is an integral part of the app development process. By running tests against your app consistently, you can verify your app's correctness, functional behavior, and usability before you release it publicly.

Testing also provides you with the following advantages:

* **Rapid feedback** on failures.
* **Early failure detection** in the development cycle.
* **Safer code refactoring**, letting you optimize code without worrying about regressions.
* **Stable development velocity**, helping you minimize technical debt.

**Create and test code iteratively**

When developing a feature iteratively, you start by either writing a new test or by adding cases and assertions to an existing unit test. The test fails at first because the feature isn't implemented yet.

It's important to consider the units of responsibility that emerge as you design the new feature. For each unit, you write a corresponding unit test. Your unit tests should nearly exhaust all possible interactions with the unit, including standard interactions, invalid inputs, and cases where resources aren't available. Take advantage of [Jetpack libraries](https://developer.android.com/jetpack) whenever possible; when you use these well-tested libraries, you can focus on validating behavior that's specific to your app.

The testing development cycle consists of writing a failing unit
           test, writing code to make it pass, and then refactoring. The entire
           feature development cycle exists inside one step of a larger,
           UI-based cycle.

**Figure 1.** The two cycles associated with iterative, test-driven development

The full workflow, as shown in Figure 1, contains a series of nested, iterative cycles where a long, slow, UI-driven cycle tests the integration of code units. You test the units themselves using shorter, faster development cycles. This set of cycles continues until your app satisfies every use case.

**View your app as a series of modules**

To make your code easier to test, develop your code in terms of *modules*, where each module represents a specific task that users complete within your app. This perspective contrasts the stack-based view of an app that typically contains layers representing the UI, business logic, and data.

For example, a "task list" app might have modules for creating tasks, viewing statistics about completed tasks, and taking photographs to associate with a particular task. Such a modular architecture also helps you keep unrelated classes decoupled and provides a natural structure for assigning ownership within your development team.

It's important to set well-defined boundaries around each module, and to create new modules as your app grows in scale and complexity. Each module should have only one area of focus, and the APIs that allow for inter-module communication should be consistent. To make it easier and quicker to test these inter-module interactions, consider creating fake implementations of your modules. In your tests, the real implementation of one module can call the fake implementation of the other module.

As you create a new module, however, don't be too dogmatic about making it full-featured right away. It's OK for a particular module to not have one or more layers of the app's stack.

**Configure your test environment**

When setting up your environment and dependencies for creating tests in your app, follow the best practices described in this section.

Organize test directories based on execution environment

A typical project in Android Studio contains two directories in which you place tests. Organize your tests as follows:

* The androidTest directory should contain the tests that run on real or virtual devices. Such tests include integration tests, end-to-end tests, and other tests where the JVM alone cannot validate your app's functionality.
* The test directory should contain the tests that run on your local machine, such as unit tests.

**Consider tradeoffs of running tests on different types of devices**

When running your tests on a device, you can choose among the following types:

* Real device
* Virtual device (such as the [emulator](https://developer.android.com/studio/run/emulator) in Android Studio)
* Simulated device (such as Robolectric)

Real devices offer the highest fidelity but also take the most time to run your tests. Simulated devices, on the other hand, provide improved test speed at the cost of lower fidelity. The platform's improvements in binary resources and realistic loopers, however, allow simulated devices to produce more realistic results.

Virtual devices offer a balance of fidelity and speed. When you use virtual devices to test, use [snapshots](https://developer.android.com/studio/run/emulator#snapshots) to minimize setup time in between tests.

**Testing Android application:**

Android applications run on a variety of devices. Also the Android framework and the surrounding open source frameworks evolve at a high speed. To ensure that you application works well, it is import to write software tests. This helps you to enhance and maintain the Android application.

Unit testing for Android can be classified into:

* Local unit tests - tests which can run on the JVM.
* Instrumented unit tests - tests which require the Android system.

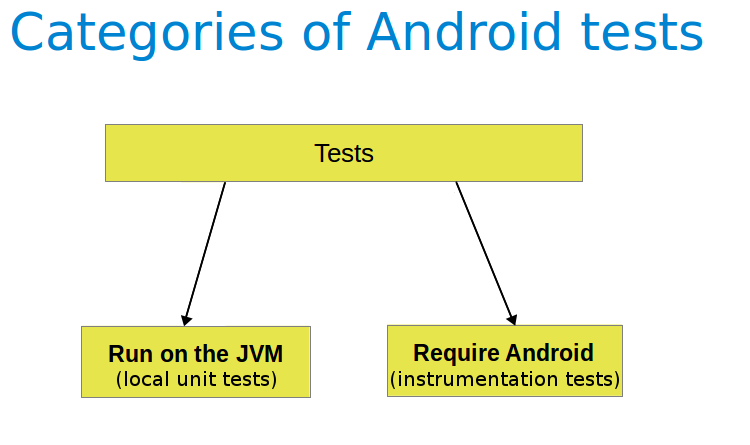
Local unit tests run much faster compared to the time required to deploy and run tests on an Android device. Prefer writing local unit tests and only run tests on Android, if you require a real Android system.

If you write local unit test and have dependencies to Android API, you need to replace them, e.g., via a mocking framework like Mockito.

### [What to test on Android applications](https://www.vogella.com/tutorials/AndroidTesting/article.html#what-to-test-on-android-applications)

Your test should focus on the business logic of your application. A good rule of thumb is to have the following distribution of tests:

* 70-80 % unit tests to ensure stability of your code basis
* 20-30 % functional tests to ensure that the application really works
* some cross functional tests if your application integrates intensively with other Application components



You should test your application at least on one device with the lowest possible configuration. In addition you should test on one device with the highest available configuration, e.g., pixel density, screen resolution to ensure that it works fine on these devices.

### [Tooling support for Android testing](https://www.vogella.com/tutorials/AndroidTesting/article.html#tooling-support-for-android-testing)

The [Android Testing Support library](https://google.github.io/android-testing-support-library/) (ATSL) project from Google provides tooling for Android testing. For example, it (AndroidJUnitRunner).

The library provides a JUnit 4-compatible test runner (AndroidJUnitRunner), the Espresso test framework and the UI Automator test framework. Espresso test framework can be used to test the User Interface of your application. UI Automator allows to write cross application functional tests.

AndroidJunitRunner provides access to the instrumentation API, via the InstrumentationRegistery.

* InstrumentationRegistry.getInstrumentation(), returns the Instrumentation currently running.
* InstrumentationRegistry.getContext(), returns the Context of this Instrumentation’s package.
* InstrumentationRegistry.getTargetContext(), returns the application Context of the target application.
* InstrumentationRegistry.getArguments(), returns a copy of arguments Bundle that was passed to this Instrumentation. This is useful when you want to access the command line arguments passed to the instrumentation for your test.

### [Android project organization for tests](https://www.vogella.com/tutorials/AndroidTesting/article.html#androidtesting_projectstructure)

The following is the default directory structure for your application and test code:

* app/src/main/java- for your source code of your main application build
* app/src/test/java - for any unit test which can run on the JVM
* app/src/androidTest/java - for any test which should run on an Android device

**Unit test:**

 unit test verifies in isolation the functionality of a certain component. For example, assume a button in an Android activity is used to start another activity. A unit test would determine, if the corresponding intent was issued, not if the second activity was started

A unit tests for an Android application can be:

* local unit test - which runs on the JVM
* Android unit test - which runs on the Android runtime

If they run on the JVM, they are executed against a modified version of the android.jar Android library. In this version all final modifiers have been stripped off. This makes it easier to use mocking libraries, like Mockito.

The local unit tests of an Android project should be located in the app/src/test folder.

### [Required dependencies in the Gradle build file](https://www.vogella.com/tutorials/AndroidTesting/article.html#androidtesting_unittests_gradledependencies)

To use JUnit tests for your Android application, you need to add it as dependency to your Gradle build file.

**dependencies** **{**

*// Unit testing dependencies*

testCompile **'junit:junit:4.12'**

*// Set this dependency if you want to use the Hamcrest matcher library*

testCompile **'org.hamcrest:hamcrest-library:1.3'**

*// more stuff, e.g., Mockito*

**}**

**Instrumented test**

The Android testing API provides hooks into the Android component and application life cycle. These hooks are called the instrumentation API and allow your tests to control the life cycle and user interaction events.

Under normal circumstances your application cannot control the life cycle events and the user drives the application. For example, if Android creates your activity the onCreate() method is called. Or if the user presses a button your corresponding code is called. Via instrumentation you can control these events via your test code. For example, your instrumentation test can start the activity. Afterwards, it can call the finish() and restart the activity to test if the instance state of the activity is correctly restored.

Instrumented unit tests are unit tests that run on Android devices and emulators instead of running on the Java virtual machine. These tests have access to the real device and its resources and are useful to unit test functionality which cannot be easily mocked by mocking frameworks. An example is a test which validates a Parcelable implementation.

An instrumentation-based test class allows you to send key events (or touch events) to the application under test.

With user interface testing framework like Espresso, the developer rarely has to use the instrumentation API directly.

### [How the Android system executes tests](https://www.vogella.com/tutorials/AndroidTesting/article.html#androidtesting_process)

The InstrumentationTestRunner is the base test runner for Android tests. This test runner starts and loads the test methods. Via the instrumentation API it communicates with the Android system. If you start a test for an Android application, the Android system kills any process of the application under test and then loads a new instance. It does not start the application, this is the responsibility of the test methods. The test method controls the life cycle of the components of the application.

The test runner also calls the onCreate() method of the application and activity under test during its initialization.

**ACTIVITY TESTING:**

To test an activity, you use the ActivityTestRule class provided by the Android Testing Support Library.

This rule provides functional testing of a single activity. The activity under test will be launched before each test annotated with @Test and before any method annotated with @Before. It will be terminated after the test is completed and all methods annotated with @After are finished. The Activity under Test can be accessed during your test by calling ActivityTestRule#getActivity().

**package** com**.**vogella**.**android**.**test**.**examples**;**

**import** **android.support.test.filters.MediumTest;**

**import** **android.support.test.rule.ActivityTestRule;**

**import** **android.support.test.runner.AndroidJUnit4;**

**import** **android.view.View;**

**import** **android.widget.ArrayAdapter;**

**import** **android.widget.ListAdapter;**

**import** **android.widget.ListView;**

**import** **org.junit.Rule;**

**import** **org.junit.Test;**

**import** **org.junit.runner.RunWith;**

**import** **static** org**.**hamcrest**.**Matchers**.**greaterThan**;**

**import** **static** org**.**hamcrest**.**Matchers**.**instanceOf**;**

**import** **static** org**.**hamcrest**.**Matchers**.**notNullValue**;**

**import** **static** org**.**junit**.**Assert**.**assertThat**;**

**@MediumTest**

**@RunWith(**AndroidJUnit4**.**class**)**

**public** **class** **MainActivityTest** **{**

**@Rule**

**public** ActivityTestRule**<**MainActivity**>** rule **=** **new** ActivityTestRule**<>(**MainActivity**.**class**);**

**@Test**

**public** **void** ensureListViewIsPresent**()** **throws** Exception **{**

MainActivity activity **=** rule**.**getActivity**();**

View viewById **=** activity**.**findViewById**(**R**.**id**.**listview**);**

assertThat**(**viewById**,**notNullValue**());**

assertThat**(**viewById**,** instanceOf**(**ListView**.**class**));**

ListView listView **=** **(**ListView**)** viewById**;**

ListAdapter adapter **=** listView**.**getAdapter**();**

assertThat**(**adapter**,** instanceOf**(**ArrayAdapter**.**class**));**

assertThat**(**adapter**.**getCount**(),** greaterThan**(5));**

**}**

**}**

To configure the intent which is used to start the activity, override ActivityTestRule#getActivityIntent.

**package** com**.**vogella**.**android**.**test**.**examples**;**

**import** **android.content.Intent;**

**import** **android.support.test.InstrumentationRegistry;**

**import** **android.support.test.filters.MediumTest;**

**import** **android.support.test.rule.ActivityTestRule;**

**import** **android.support.test.runner.AndroidJUnit4;**

**import** **android.view.View;**

**import** **android.widget.ListView;**

**import** **android.widget.TextView;**

**import** **org.junit.Rule;**

**import** **org.junit.Test;**

**import** **org.junit.runner.RunWith;**

**import** **static** org**.**hamcrest**.**Matchers**.**instanceOf**;**

**import** **static** org**.**hamcrest**.**Matchers**.**is**;**

**import** **static** org**.**hamcrest**.**Matchers**.**notNullValue**;**

**import** **static** org**.**junit**.**Assert**.**assertThat**;**

**@MediumTest**

**@RunWith(**AndroidJUnit4**.**class**)**

**public** **class** **SecondActivityTest** **{**

**@Rule**

**public** ActivityTestRule**<**SecondActivity**>** rule**=** **new** ActivityTestRule**<**SecondActivity**>(**SecondActivity**.**class**)**

**{**

**@Override**

**protected** Intent getActivityIntent**()** **{**

InstrumentationRegistry**.**getTargetContext**();**

Intent intent **=** **new** Intent**(**Intent**.**ACTION\_MAIN**);**

intent**.**putExtra**("MYKEY",** **"Hello");**

**return** intent**;**

**}**

**};**

**@Test**

**public** **void** ensureIntentDataIsDisplayed**()** **throws** Exception **{**

SecondActivity activity **=** rule**.**getActivity**();**

View viewById **=** activity**.**findViewById**(**R**.**id**.**target**);**

assertThat**(**viewById**,**notNullValue**());**

assertThat**(**viewById**,** instanceOf**(**TextView**.**class**));**

TextView textView **=** **(**TextView**)** viewById**;**

assertThat**(**textView**.**getText**().**toString**(),**is**("Hello"));**

**}**

**}**

# SERVICE TESTING

Android provides a testing framework for Service objects that can run them in isolation and provides mock objects. The test case class for Service objects is [ServiceTestCase](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/ServiceTestCase.html). Since the Service class assumes that it is separate from its clients, you can test a Service object without using instrumentation.

This document describes techniques for testing Service objects. If you aren't familiar with the Service class, please read the [Services](https://stuff.mit.edu/afs/sipb/project/android/docs/guide/components/services.html) document. If you aren't familiar with Android testing, please read [Testing Fundamentals](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/testing/testing_android.html), the introduction to the Android testing and instrumentation framework.

## Service Design and Testing

When you design a Service, you should consider how your tests can examine the various states of the Service lifecycle. If the lifecycle methods that start up your Service, such as [onCreate()](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/app/Service.html" \l "onCreate()) or [onStartCommand()](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/app/Service.html" \l "onStartCommand(android.content.Intent,%20int,%20int)) do not normally set a global variable to indicate that they were successful, you may want to provide such a variable for testing purposes.

Most other testing is facilitated by the methods in the [ServiceTestCase](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/ServiceTestCase.html) test case class. For example, the [getService()](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/ServiceTestCase.html" \l "getService()) method returns a handle to the Service under test, which you can test to confirm that the Service is running even at the end of your tests.

## ServiceTestCase

[ServiceTestCase](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/ServiceTestCase.html) extends the JUnit [TestCase](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/junit/framework/TestCase.html) class with with methods for testing application permissions and for controlling the application and Service under test. It also provides mock application and Context objects that isolate your test from the rest of the system.

[ServiceTestCase](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/ServiceTestCase.html) defers initialization of the test environment until you call [ServiceTestCase.startService()](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/ServiceTestCase.html#startService(android.content.Intent)) or [ServiceTestCase.bindService()](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/ServiceTestCase.html#bindService(android.content.Intent)). This allows you to set up your test environment, particularly your mock objects, before the Service is started.

Notice that the parameters to ServiceTestCase.bindService()are different from those for Service.bindService(). For the ServiceTestCase version, you only provide an Intent. Instead of returning a boolean, ServiceTestCase.bindService() returns an object that subclasses [IBinder](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/os/IBinder.html).

The [setUp()](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/ServiceTestCase.html" \l "setUp()) method for [ServiceTestCase](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/ServiceTestCase.html) is called before each test. It sets up the test fixture by making a copy of the current system Context before any test methods touch it. You can retrieve this Context by calling [getSystemContext()](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/ServiceTestCase.html" \l "getSystemContext()). If you override this method, you must call super.setUp() as the first statement in the override.

The methods [setApplication()](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/ServiceTestCase.html" \l "setApplication(android.app.Application)) and [setContext(Context)](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/AndroidTestCase.html" \l "setContext(android.content.Context)) setContext()} allow you to set a mock Context or mock Application (or both) for the Service, before you start it. These mock objects are described in [Mock object classes](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/testing/service_testing.html#MockObjects).

By default, [ServiceTestCase](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/ServiceTestCase.html) runs the test method [testAndroidTestCaseSetupProperly()](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/AndroidTestCase.html" \l "testAndroidTestCaseSetupProperly()), which asserts that the base test case class successfully set up a Context before running.

## Mock object classes

ServiceTestCase assumes that you will use a mock Context or mock Application (or both) for the test environment. These objects isolate the test environment from the rest of the system. If you don't provide your own instances of these objects before you start the Service, then [ServiceTestCase](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/ServiceTestCase.html) will create its own internal instances and inject them into the Service. You can override this behavior by creating and injecting your own instances before starting the Service

To inject a mock Application object into the Service under test, first create a subclass of [MockApplication](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/mock/MockApplication.html). MockApplication is a subclass of [Application](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/app/Application.html) in which all the methods throw an Exception, so to use it effectively you subclass it and override the methods you need. You then inject it into the Service with the [setApplication()](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/ServiceTestCase.html" \l "setApplication(android.app.Application)) method. This mock object allows you to control the application values that the Service sees, and isolates it from the real system. In addition, any hidden dependencies your Service has on its application reveal themselves as exceptions when you run the test.

# CONTENT PROVIDER TESTING

Content providers, which store and retrieve data and make it accessible across applications, are a key part of the Android API. As an application developer you're allowed to provide your own public providers for use by other applications. If you do, then you should test them using the API you publish.

This document describes how to test public content providers, although the information is also applicable to providers that you keep private to your own application. If you aren't familiar with content providers or the Android testing framework, please read [Content Providers](https://stuff.mit.edu/afs/sipb/project/android/docs/guide/topics/providers/content-providers.html), the guide to developing content providers, and [Testing Fundamentals](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/testing/testing_android.html), the introduction to the Android testing and instrumentation framework.

## Content Provider Design and Testing

In Android, content providers are viewed externally as data APIs that provide tables of data, with their internals hidden from view. A content provider may have many public constants, but it usually has few if any public methods and no public variables. This suggests that you should write your tests based only on the provider's public members. A content provider that is designed like this is offering a contract between itself and its users.

The base test case class for content providers, [ProviderTestCase2](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/ProviderTestCase2.html), allows you to test your content provider in an isolated environment. Android mock objects such as [IsolatedContext](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/IsolatedContext.html) and [MockContentResolver](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/mock/MockContentResolver.html) also help provide an isolated test environment.

As with other Android tests, provider test packages are run under the control of the test runner [InstrumentationTestRunner](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/InstrumentationTestRunner.html). The section [Running Tests With InstrumentationTestRunner](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/testing/testing_android.html#InstrumentationTestRunner) describes the test runner in more detail. The topic [Testing From Eclipse with ADT](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/testing/testing_eclipse.html) shows you how to run a test package in Eclipse, and the topic [Testing From Other IDEs](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/testing/testing_otheride.html) shows you how to run a test package from the command line.

## Content Provider Testing API

The main focus of the provider testing API is to provide an isolated testing environment. This ensures that tests always run against data dependencies set explicitly in the test case. It also prevents tests from modifying actual user data. For example, you want to avoid writing a test that fails because there was data left over from a previous test, and you want to avoid adding or deleting contact information in a actual provider.

The test case class and mock object classes for provider testing set up this isolated testing environment for you.

### ProviderTestCase2

You test a provider with a subclass of [ProviderTestCase2](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/ProviderTestCase2.html). This base class extends [AndroidTestCase](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/AndroidTestCase.html), so it provides the JUnit testing framework as well as Android-specific methods for testing application permissions. The most important feature of this class is its initialization, which creates the isolated test environment.

The initialization is done in the constructor for [ProviderTestCase2](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/ProviderTestCase2.html), which subclasses call in their own constructors. The [ProviderTestCase2](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/ProviderTestCase2.html) constructor creates an [IsolatedContext](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/IsolatedContext.html) object that allows file and database operations but stubs out other interactions with the Android system. The file and database operations themselves take place in a directory that is local to the device or emulator and has a special prefix.

The constructor then creates a [MockContentResolver](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/mock/MockContentResolver.html) to use as the resolver for the test. The [MockContentResolver](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/mock/MockContentResolver.html) class is described in detail in the section [Mock object classes](https://stuff.mit.edu/afs/sipb/project/android/docs/tools/testing/testing_android.html#MockObjectClasses).

Lastly, the constructor creates an instance of the provider under test. This is a normal [ContentProvider](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/content/ContentProvider.html) object, but it takes all of its environment information from the [IsolatedContext](https://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/test/IsolatedContext.html), so it is restricted to working in the isolated test environment. All of the tests done in the test case class run against this isolated object.

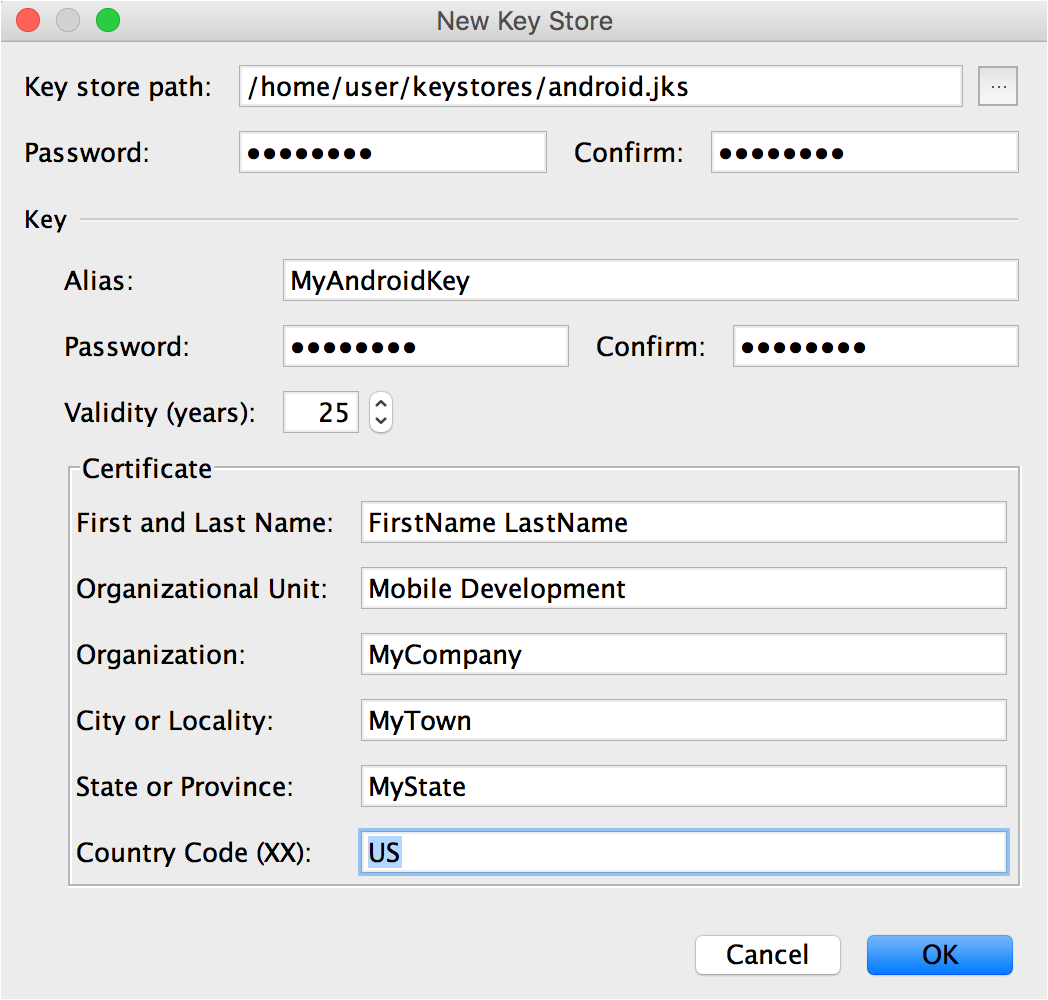
# HOW TO GET YOUR APP ON THE APP STORE

# Create Your Android App



* First thing for uploading your App on Play Store is to Develop your App and make it Store-Ready.
* Use Proper theme, colors, graphics, strings and icons.

# 2. Make its Signed Apk(Using Release Keystore)



This is a graphical way to generate a release keystore

To generate keystores for signing Android apps at the command line, use:

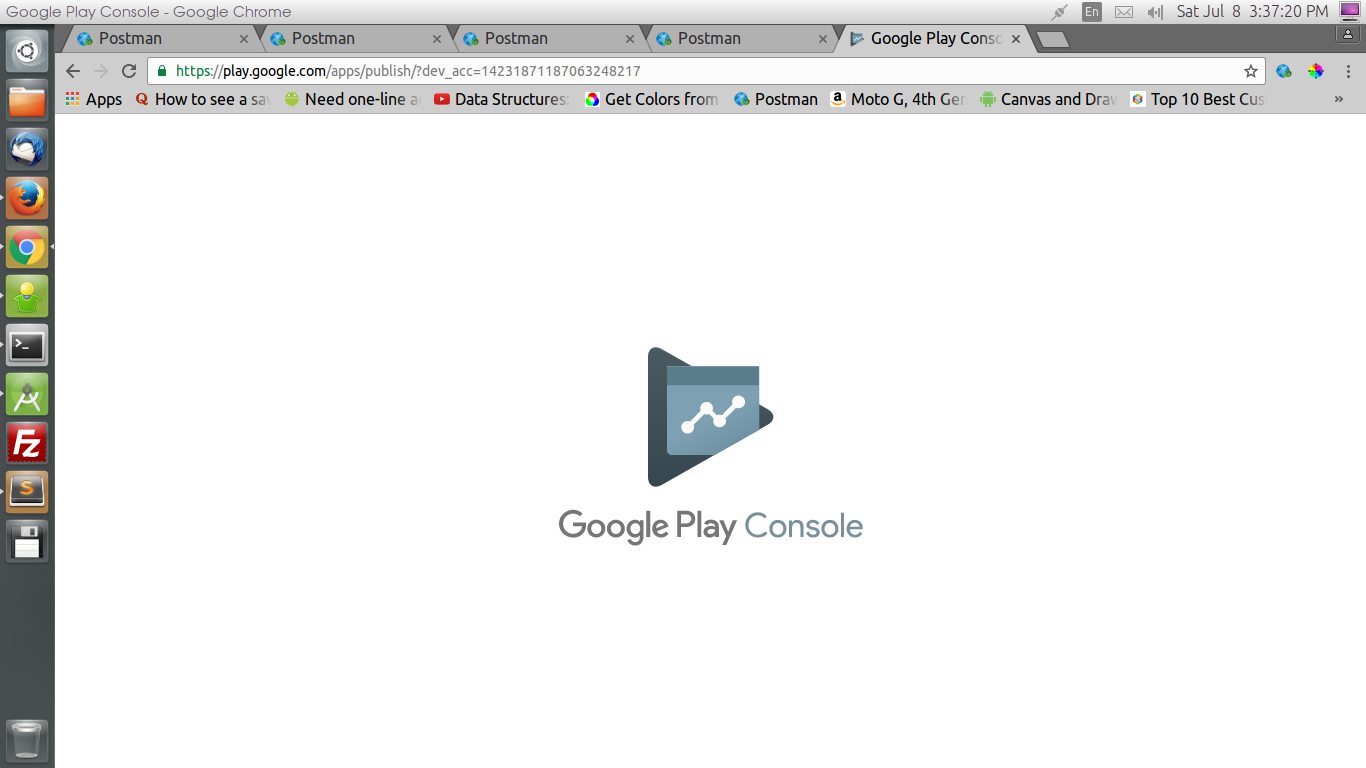
*$ keytool -genkey -v -keystore my-key.keystore -alias alias\_name -keyalg RSA -keysize 2048 -validity 10000*

A debug keystore which is used to sign an Android app during development needs a specific alias and password combination as dictated by Google. To create a debug keystore, use:

*$ keytool -genkey -v -keystore debug.keystore -storepass android -alias androiddebugkey -keypass android -keyalg RSA -keysize 2048 -validity 10000*

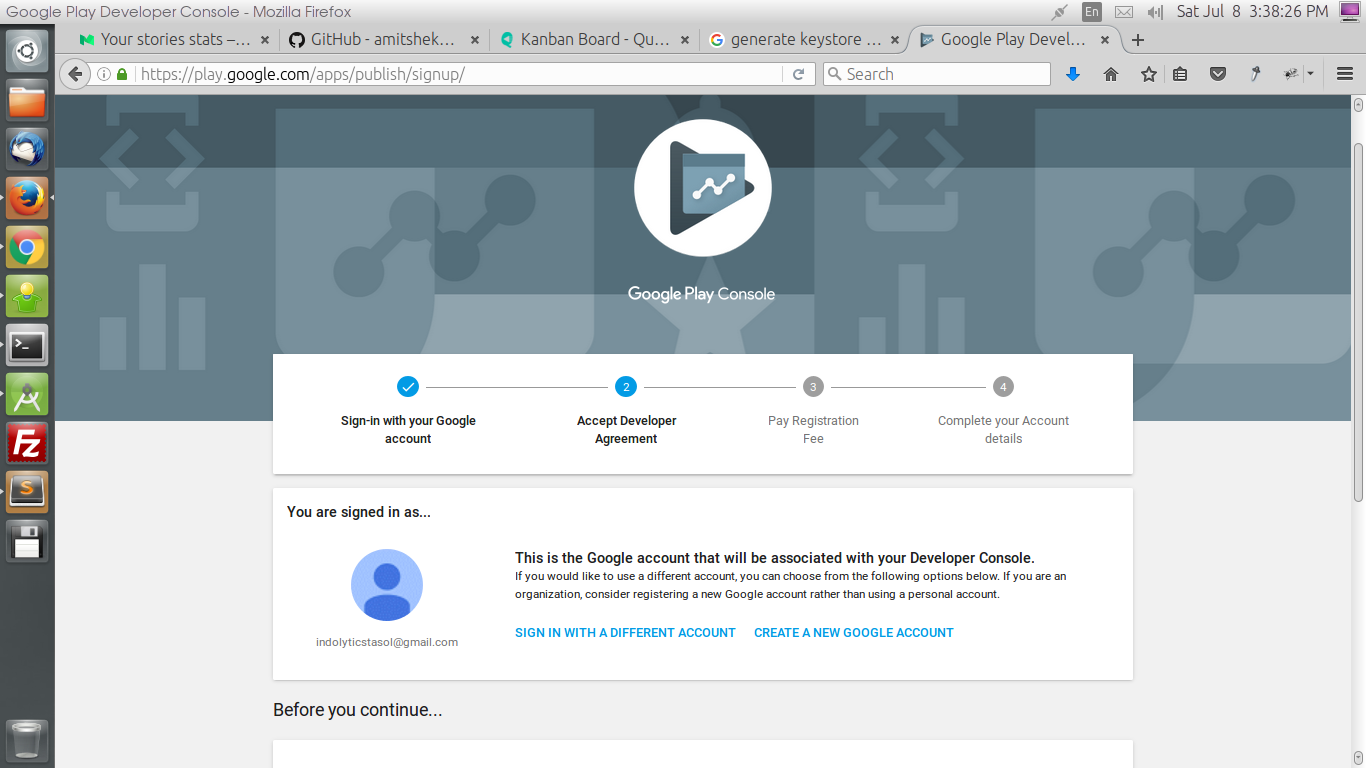
Keystore name: “debug.keystore”  
Keystore password: “android”  
Key alias: “androiddebugkey”  
Key password: “android”

# 3. Login to your Gmail Account and visit this link :



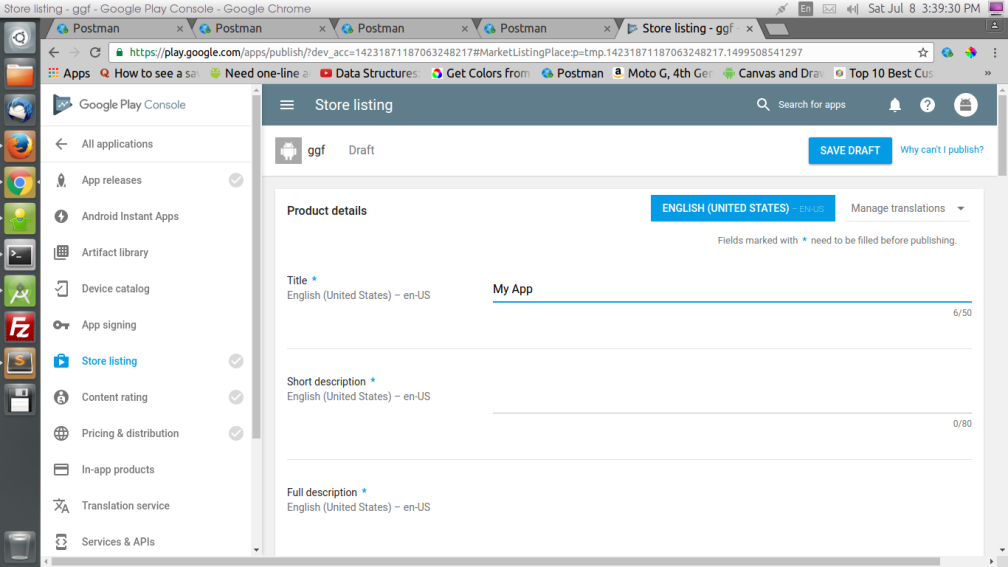
And Go to this Link — <https://play.google.com/apps/publish/>

# 4. Create a Merchant Account



Just Follow the steps kindly…

# 5. Mandatory Fields needed in App Listing:



— Title  
— Short Desc  
— Full Desc  
— App Screenshots(JPEG or 24-bit PNG (no alpha))(Min-2,Max-8)(Min-320px,Max-3840px)  
— Hi-res icon(512 x 512)(32-bit PNG (with alpha))  
— Feature Graphic(1024 w x 500 h)(JPG or 24-bit PNG (no alpha))  
— App Type  
— Category  
— Content Rating  
— Developer/Company Email  
— Privacy Policy Url  
— And some other details