Android app testing

with Android Studio

Last updated: 31 October 2017

Contents

Android app testing with Android Studio

Google’s free Android Integrated Development Environment (IDE) 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 Java Virtual Machine (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 to test Android API calls in your local unit tests, and Espresso or UI Automator to exercise user interaction in your instrumented tests. You can generate Espresso tests automatically using Espresso Test Recorder.

Android Studio also has a host of very smart, performance monitoring tools that allow developers to fine-tune projects and maximize their efficiency.

Logcat

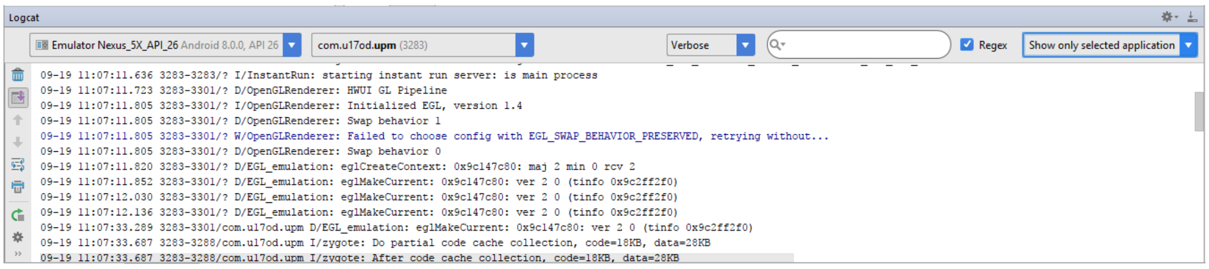
Logcat is an Android logging tool that allows access to messages from applications and services running on an Android device. Android developers can log messages for debugging with Logcat.

To use Logcat, first import “`android.util.Log`” into your Android project. Now you can call the static class “`Log`” from your project to start logging. Logcat has different levels of logging. When debugging, you might want to use Debug (d) to log the progress. Of course, when you want to log an actual error, you will use Error (e).

A good Logcat convention is to declare a “`TAG`” constant in your class to use in the first parameter. For example, you might create a debug log message as follows:

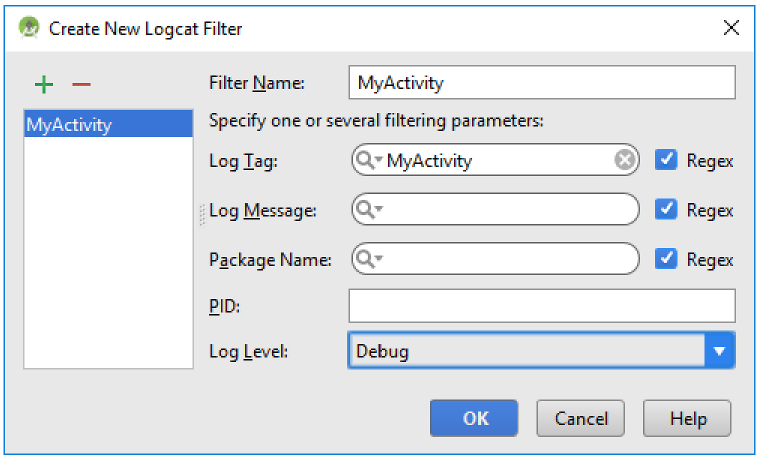
```
private static final String TAG = "MyActivity";
...
Log.d(TAG, "Here goes your text");
```

In Android Studio, the keyboard shortcut “<Alt><6>” opens the Logcat window.


Logcat
Emulator Nexus_5X_API_26 Android 8.0.0, API 26
com.u17od.upm (3283)
Verbose
Regex
Show only selected application

```
09-19 11:07:11.636 3283-3283/? I/InstantRun: starting instant run server: is main process
09-19 11:07:11.723 3283-3301/? D/OpenGLRenderer: HWUI GL Pipeline
09-19 11:07:11.805 3283-3301/? I/OpenGLRenderer: Initialized EGL, version 1.4
09-19 11:07:11.805 3283-3301/? D/OpenGLRenderer: Swap behavior 1
09-19 11:07:11.805 3283-3301/? W/OpenGLRenderer: Failed to choose config with EGL_SWAP_BEHAVIOR_PRESERVED, retrying without...
09-19 11:07:11.805 3283-3301/? D/OpenGLRenderer: Swap behavior 0
09-19 11:07:11.820 3283-3301/? D/EGL_emulation: eglCreateContext: 0x9c147c80: maj 2 min 0 rcv 2
09-19 11:07:11.852 3283-3301/? D/EGL_emulation: eglMakeCurrent: 0x9c147c80: ver 2 0 (tinfo 0x9c2ff2f0)
09-19 11:07:12.030 3283-3301/? D/EGL_emulation: eglMakeCurrent: 0x9c147c80: ver 2 0 (tinfo 0x9c2ff2f0)
09-19 11:07:12.136 3283-3301/? D/EGL_emulation: eglMakeCurrent: 0x9c147c80: ver 2 0 (tinfo 0x9c2ff2f0)
09-19 11:07:33.289 3283-3301/com.u17od.upm D/EGL_emulation: eglMakeCurrent: 0x9c147c80: ver 2 0 (tinfo 0x9c2ff2f0)
09-19 11:07:33.687 3283-3288/com.u17od.upm I/zygote: Do partial code cache collection, code=18KB, data=28KB
09-19 11:07:33.687 3283-3288/com.u17od.upm I/zygote: After code cache collection, code=18KB, data=28KB
```

When looking for a particular message, it is often required to set a filter before running the app to reduce the amount of displayed log information. This can be done by selecting “Select Edit Filter Configuration” from the dropdown in the top-right corner of the Logcat window, and complete the dialog as follows:


Create New Logcat Filter

Filter Name: MyActivity

Specify one or several filtering parameters:

Log Tag: MyActivity [✓ Regex]

Log Message: [✓ Regex]

Package Name: [✓ Regex]

PID:

Log Level: Debug

Detailed information about Logcat can be found here:

https://developer.android.com/studio/debug/am-logcat.html

JUnit testing

Android Studio incorporates JUnit testing directly into the workspace.

The Android Studio JUnit test framework provides for two distinct types of tests. They are as follows:

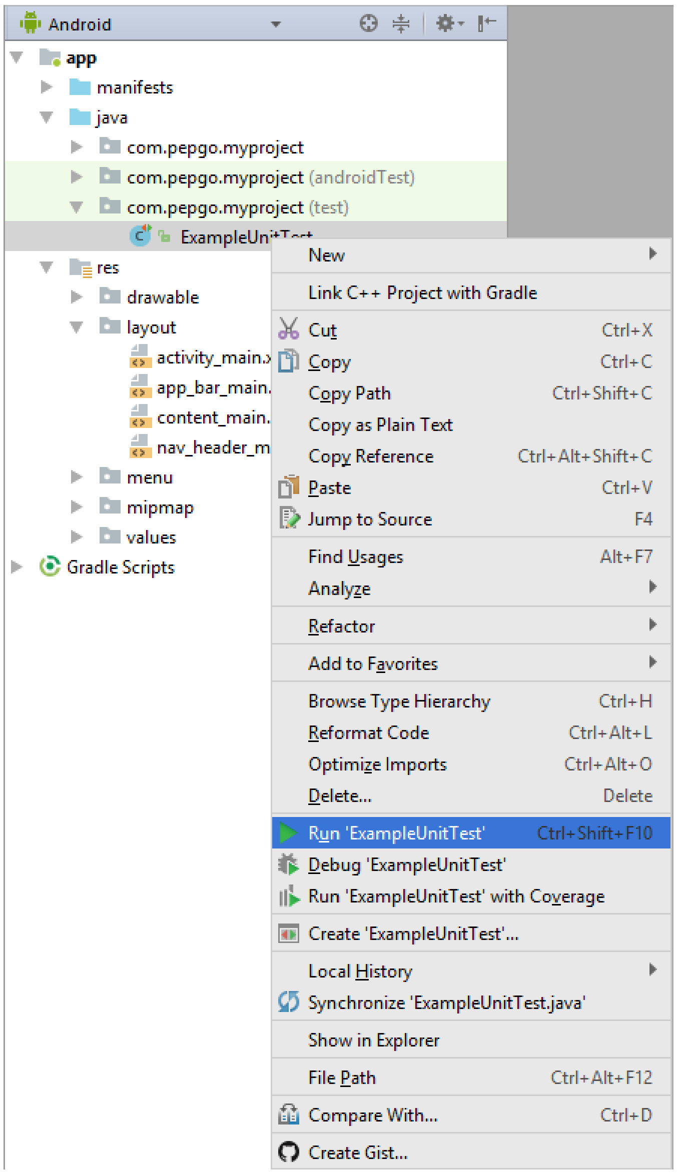
- Local unit tests are used to test business logic in an isolated environment that is not dependent on Android components or other code, although it is possible to mock some dependencies. These tests run on the local Java Virtual Machine (JVM) and are consequently considerably faster than testing on a hardware device or emulator.
- Instrumented tests are used when we want to test elements of the Android framework itself, such as how our User Interfaces behave. These tests generate an APK file and are, therefore, slower to build.

Local unit tests

If you have created an Android Studio project using the project wizard, then the basic test case for both test types will have been created automatically. The wizard will also include the necessary Gradle dependencies. It will have created an example test class "`ExampleUnitTest.Java`". This class contains a single method for testing arithmetic:

```
public class ExampleUnitTest {
     @Test
     public void addition_isCorrect() throws Exception {
          assertEquals(4, 2 + 2);
     }
}
```

Despite their actual location on disk, test modules can be found alongside your regular Java modules in the "Android" view of Android Studio's project explorer.



The following example demonstrates how to unit test application code:

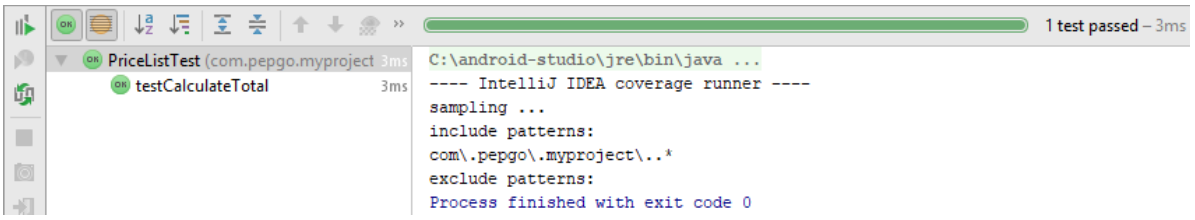
1. Create a Java class “`PriceList`” in the default package, with a single function, like the one here:

```
public class PriceList {
     public int CalculateTotal(int item1, int item2) {
          int total;
          total = (item1 + item2);
          return total;
     }
}
```

2. Create a new class “`PriceListTest`” in the test package:

```
import org.junit.Test;
import static org.junit.Assert.*;

public class PriceListTest {
     @Test
     public void testCalculateTotal(){
          PriceList priceList = new PriceList();
          int result = priceList.CalculateTotal(199, 250);
          assertEquals(449,result);
     }
}
```

3. Run the test:



PriceListTest (com.pepgo.myproject 3ms
testCalculateTotal 3ms
1 test passed – 3ms

```
C:\android-studio\jre\bin\java ...
---- IntelliJ IDEA coverage runner ----
sampling ...
include patterns:
com\.pepgo\.myproject\..*
exclude patterns:
Process finished with exit code 0
```

For all JUnit functionality, from assertions and structures up to building test suites, please refer to the JUnit documentation at:

http://junit.org/

User Interface tests

At the heart of instrumented User Interface testing lies the Android Testing Support Library. This includes the JUnit API's, a UI Automator and the Espresso testing framework.

Put simply, Espresso allows us to perform three essential tasks:

1. Identify and access views and other User Interface elements.
2. Perform an activity, such as clicking and swiping.
3. Validate assertions to test code.

The case construct for Espresso tests is the following:

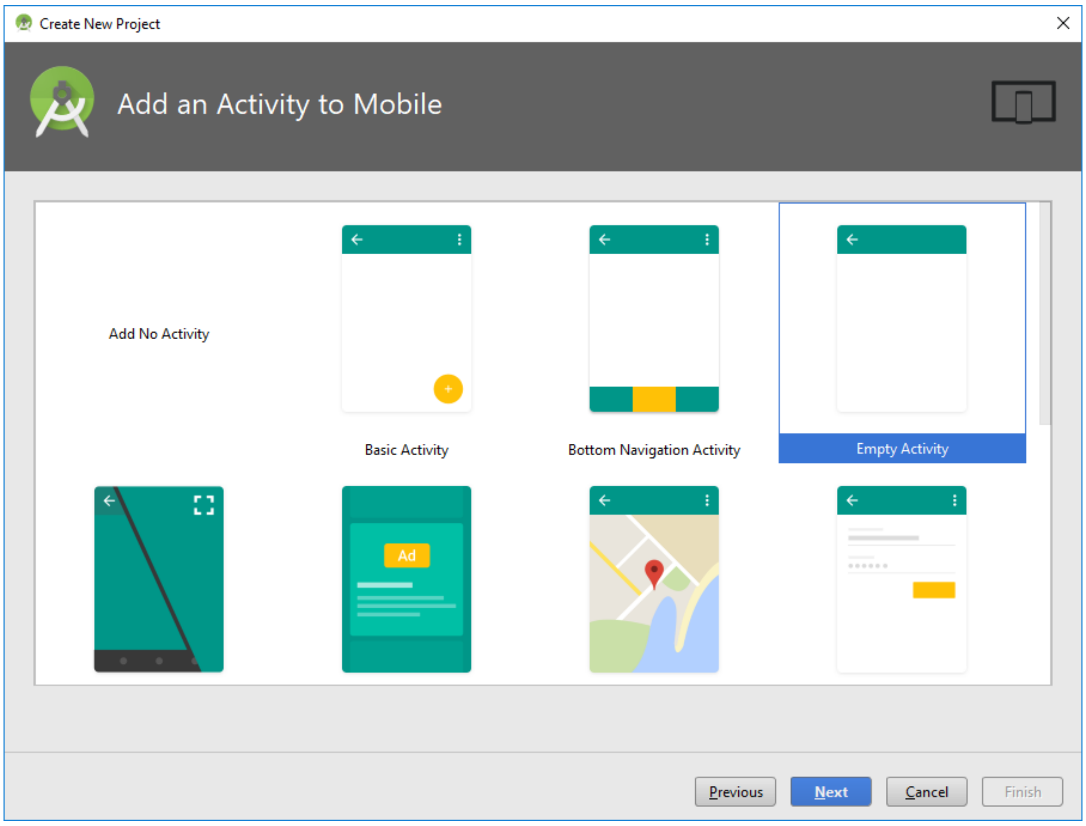
```
onView(ViewMatcher)
     .perform(ViewAction)
     .check(ViewAssertion);
```

"`onView`" finds the view, "`perform`" performs an action on the view, "`check`" validates an assertion.

The following walkthrough shows a simple Espresso test.

1. Start Android Studio and select "Start a new Android Studio project".
2. Name the project "myproject" and click on "Next".
3. Make it a "Phone or Tablet" application with a device and API number of your choice and click on "Next".

4. Use the “Empty Activity” template and click on “Next”:

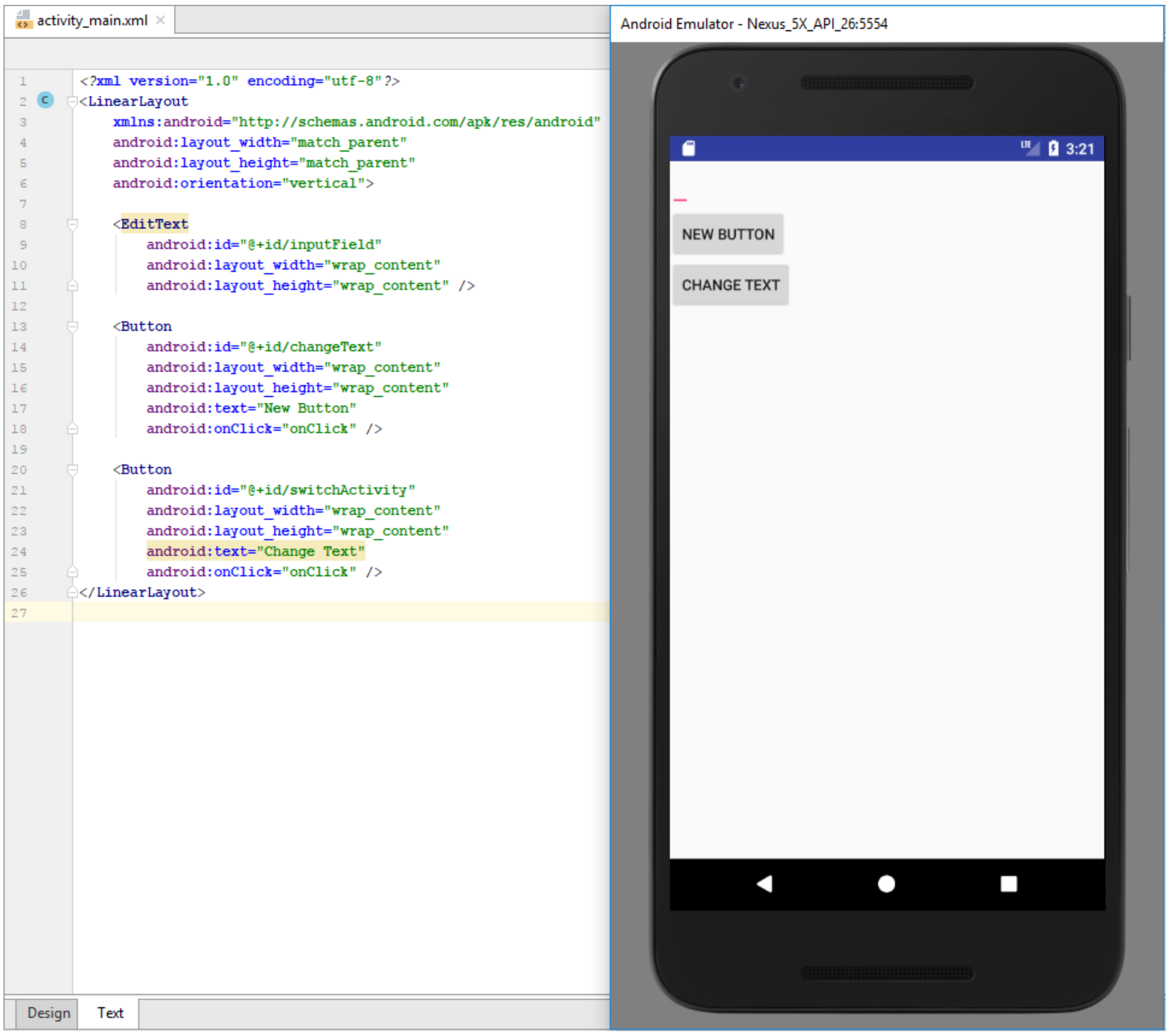


5. Accept the proposed names and click “Finish”.
6. Add the following to your “`build.gradle (Module:app)`” file:

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])

    testCompile 'junit:junit:4.12'

    // Android runner and rules support
    androidTestCompile 'com.android.support.test:runner:1.0.1'
    androidTestCompile 'com.android.support.test:rules:1.0.1'

    // Espresso support
    androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })

    // add this for intent mocking support
    androidTestCompile 'com.android.support.test.espresso:espresso-intents:3.0.1'

    // add this for webview testing support
    androidTestCompile 'com.android.support.test.espresso:espresso-web:3.0.1'

}
```

7. Change the existing “`activity_main.xml`” layout file to the following:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"

    <EditText
        android:id="@+id/inputField"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <Button
        android:id="@+id/changeText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="New Button"
        android:onClick="onClick" />

    <Button
        android:id="@+id/switchActivity"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Change Text"
        android:onClick="onClick" />
</LinearLayout>
```



```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <EditText
        android:id="@+id/inputField"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <Button
        android:id="@+id/changeText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="New Button"
        android:onClick="onClick" />

    <Button
        android:id="@+id/switchActivity"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Change Text"
        android:onClick="onClick" />
</LinearLayout>
```

8. Create a new layout resource file called "`activity_second.xml`" with the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:text="Large Text"
        android:id="@+id/resultView" />
</LinearLayout>
```

9. Create a new java class called “`SecondActivity`” (in the same location as the existing “`MainActivity`” class) with the following content:

```
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

public class SecondActivity extends Activity{
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);
        TextView viewById = (TextView) findViewById(R.id.resultView);
        Bundle inputData = getIntent().getExtras();
        String input = inputData.getString("input");
        viewById.setText(input);
    }
}
```

10. The just added “`SecondActivity`” class will only be accessible after it has been declared in the “`AndroidManifest.xml`”. It is therefore required to add the following line to “`AndroidManifest.xml`”:

```
<activity android:name=".SecondActivity" />
```

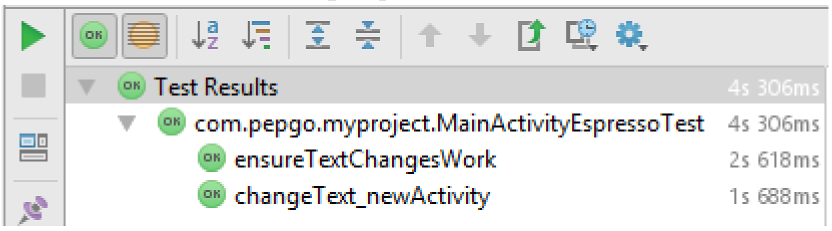
11. Adjust the existing “`MainActivity`” class to the following content:

```
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;

public class MainActivity extends Activity {

    EditText editText;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        editText = (EditText) findViewById(R.id.inputField);
    }

    public void onClick(View view) {
        switch (view.getId()) {
            case R.id.changeText:
                editText.setText("Lalala");
                break;
            case R.id.switchActivity:
                Intent intent = new Intent(this, SecondActivity.class);
                intent.putExtra("input", editText.getText().toString());
                startActivity(intent);
                break;
        }

    }
}
```

12. Create a new java class called "`MainActivityEspressoTest`" (in the "androidTest" section, which is the same location as the existing "`ExampleInstrumentedTest`" class) with the following content:

```
import android.support.test.rule.ActivityTestRule;
import android.support.test.runner.AndroidJUnit4;

import org.junit.Rule;
import org.junit.Test;
import org.junit.runner.RunWith;

import static android.support.test.espresso.Espresso.onView;
import static android.support.test.espresso.action.ViewActions.click;
import static android.support.test.espresso.action.ViewActions.closeSoftKeyboard;
import static android.support.test.espresso.action.ViewActions.typeText;
import static android.support.test.espresso.assertion.ViewAssertions.matches;
import static android.support.test.espresso.matcher.ViewMatchers.withId;
import static android.support.test.espresso.matcher.ViewMatchers.withText;

@RunWith(AndroidJUnit4.class)
public class MainActivityEspressoTest {

    @Rule
    public ActivityTestRule<MainActivity> mActivityRule =
        new ActivityTestRule<>(MainActivity.class);

    @Test
    public void ensureTextChangesWork() {
        // Type text and then press the button.
        onView(withId(R.id.inputField))
                .perform(typeText("HELLO"), closeSoftKeyboard());
        onView(withId(R.id.changeText)).perform(click());

        // Check that the text was changed.
        onView(withId(R.id.inputField)).check(matches(withText("Lalala")));
    }

    @Test
    public void changeText_newActivity() {
        // Type text and then press the button.
        onView(withId(R.id.inputField)).perform(typeText("NewText"),
                closeSoftKeyboard());
        onView(withId(R.id.switchActivity)).perform(click());

        // This view is in a different Activity, no need to tell Espresso.
        onView(withId(R.id.resultView)).check(matches(withText("NewText")));
    }
}
```

13. Build the project.
14. Run the "`MainActivitiyEspressoTest`". It should pass like this:

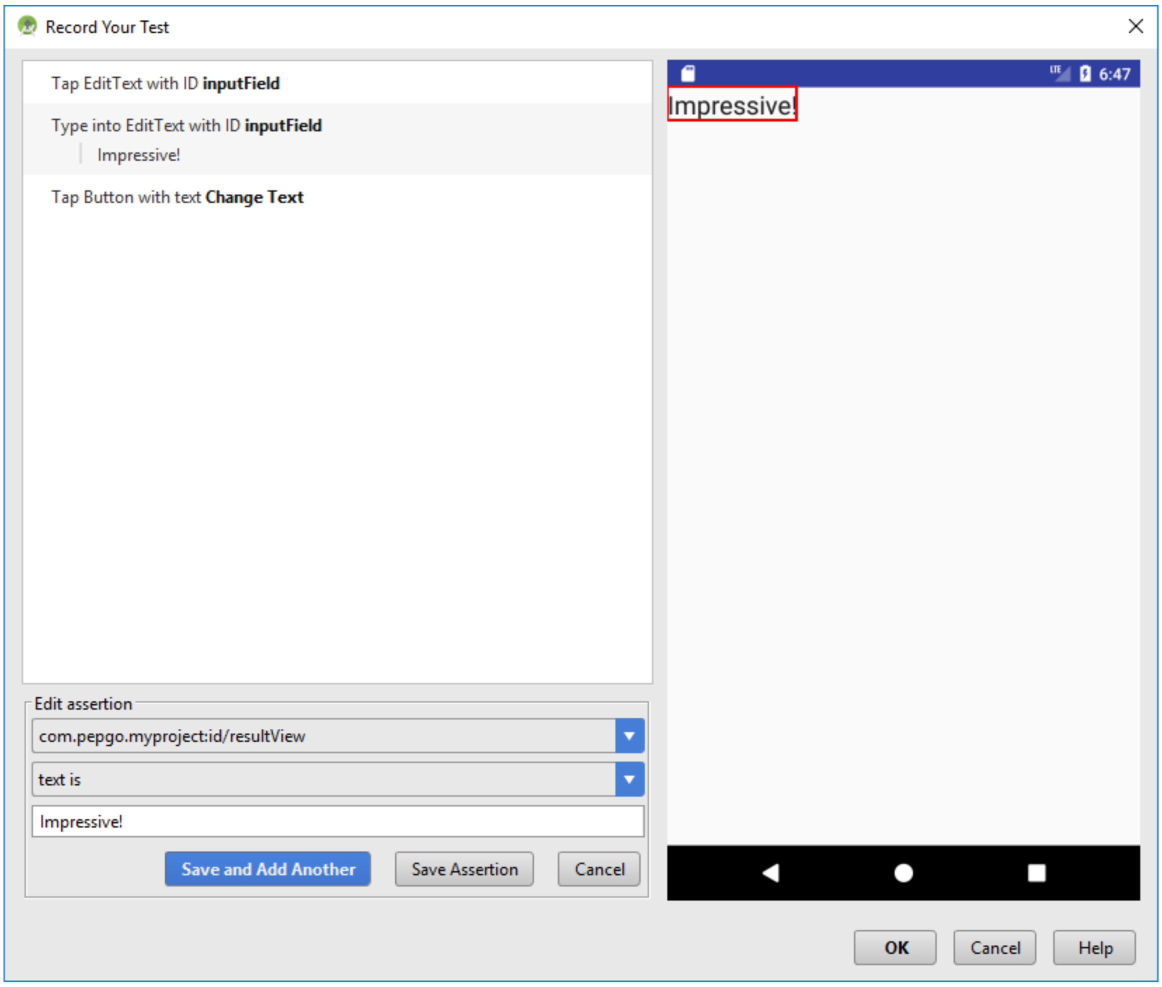


```
Test Results                                      4s 306ms
  com.pepgo.myproject.MainActivityEspressoTest    4s 306ms
    ensureTextChangesWork                         2s 618ms
    changeText_newActivity                        1s 688ms
```

Espresso Test Recorder

The Espresso Test Recorder tool lets you create User Interface tests for your app without writing any test code. By recording a test scenario, you can record your interactions with a device and add assertions to verify User Interface elements in particular snapshots of your app. Espresso Test Recorder then takes the saved recording and automatically generates a corresponding User Interface test that you can run to test your app.

1. Open the previously created “myproject” app, if is not already open.
2. Select “Record Espresso Test” from the “Run” menu and run the app on any device of your choice.
3. Select the connected device and click “OK”.
4. Type “Impressive!” in the EditBox.
5. Click the “Change Text” button.
6. In the “Record Your Test” window, click on the “Add Assertion” button.
7. Click on the “Impressive!” text, as shown here:



Record Your Test

Tap EditText with ID **inputField**

Type into EditText with ID **inputField**
Impressive!

Tap Button with text **Change Text**

Edit assertion
com.pepgo.myproject:id/resultView
text is
Impressive!

8. Click on the “Save Assertion” button.
9. Click on the “OK” button.
10. Confirm the proposed class name “`MainActivityTest`”, or give the class a different name of your choice, and click on “OK”.
11. Run “`MainActivityTest`”. It should pass.

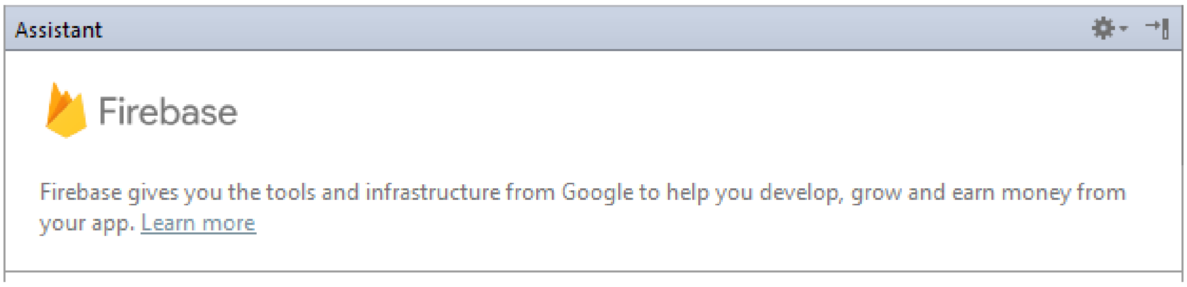
Firebase Test Lab

When running tests from the Select Deployment Target dialog (where you select the target device to execute your tests on), there is also another tab, called “Cloud Testing”. This feature gives you access to the Firebase Test Lab directly from Android Studio.

When developing an Android application for a general release, it is desirable to test it on as many different device configurations and platform versions as possible. Testing on a large number of real devices is impractical and expensive, and it would seem that virtual devices offer the only other option. Fortunately, Firebase provides a cloud-based test lab that allows us to test our apps on a wide range of real devices and emulators across all platform versions.

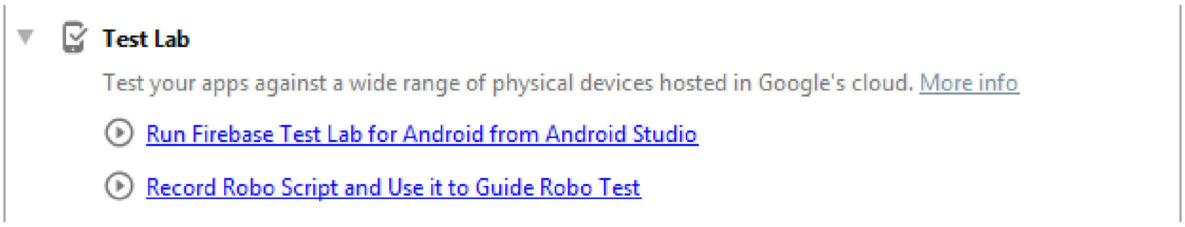
Firebase is a cloud-based application development suite with many useful features, such as file hosting and real-time crash reporting.

The simplest way to get started is the Firebase Assistant, which can be found in the “Tools” menu:



Firebase gives you the tools and infrastructure from Google to help you develop, grow and earn money from your app. Learn more

… the Assistant includes a “Test Lab” selection …

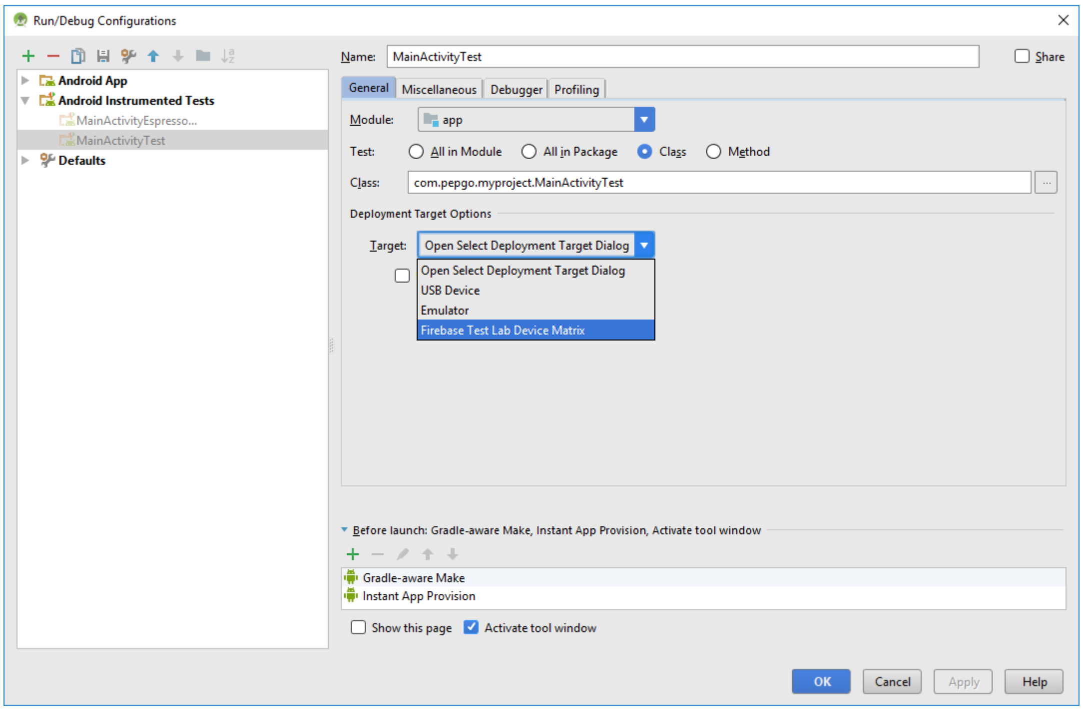


Test your apps against a wide range of physical devices hosted in Google's cloud. More info

Run Firebase Test Lab for Android from Android Studio

Record Robo Script and Use it to Guide Robo Test

To run tests in the Firebase cloud, Firebase will have to be set as the target platform in the “Run” menu with “Edit Configurations…”:



Run/Debug Configurations

- Android App
- Android Instrumented Tests
  - MainActivityEspresso...
  - MainActivityTest
- Defaults

Name: MainActivityTest

Share

General | Miscellaneous | Debugger | Profiling

Module: app

Test: All in Module, All in Package, [✓ Class], Method

Class: com.pepgo.myproject.MainActivityTest

Deployment Target Options

Target: Open Select Deployment Target Dialog

- Open Select Deployment Target Dialog
- USB Device
- Emulator
- Firebase Test Lab Device Matrix

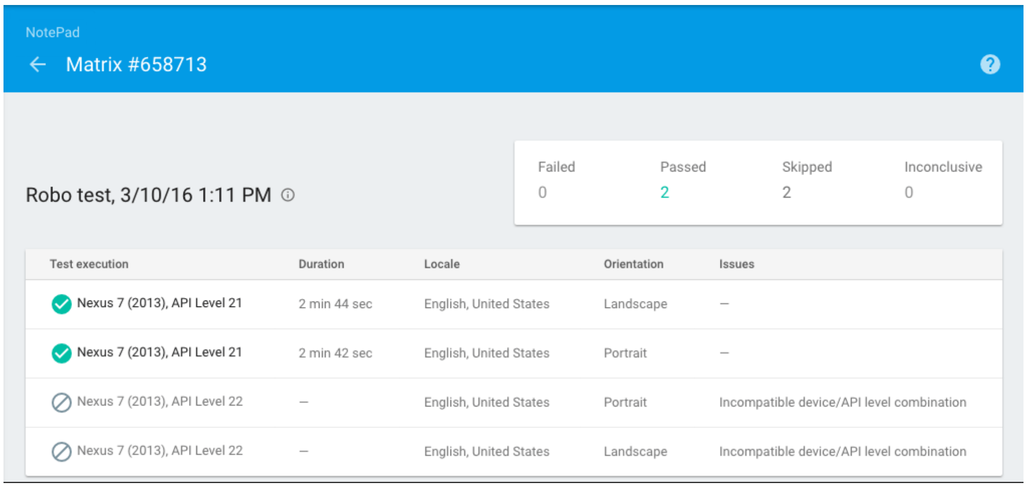
Before launch: Gradle-aware Make, Instant App Provision, Activate tool window

- Gradle-aware Make
- Instant App Provision

Show this page [✓ Activate tool window]

OK | Cancel | Apply | Help

The tests can then be started in the same way as any other tests, with the Run icon or menu item, and you will see from the test output a link to view an HTML version of the results, which might look as follows:



NotePad

Matrix #658713

Robo test, 3/10/16 1:11 PM

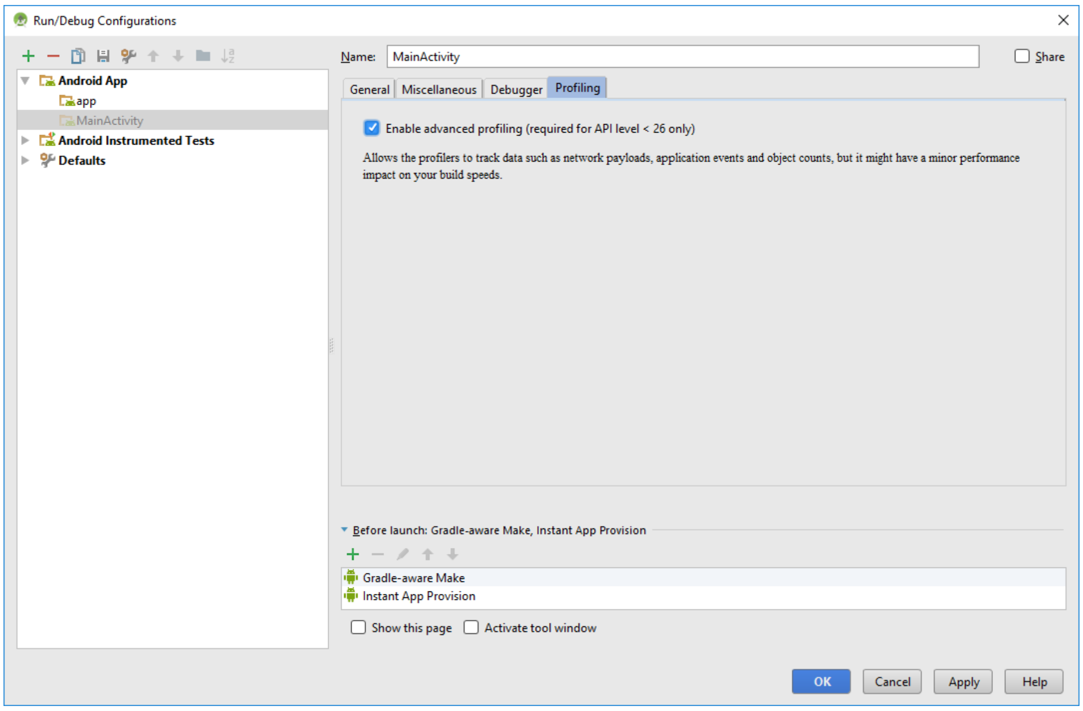
Failed	Passed	Skipped	Inconclusive
0	2	2	0

Test execution	Duration	Locale	Orientation	Issues
✓ Nexus 7 (2013), API Level 21	2 min 44 sec	English, United States	Landscape	–
✓ Nexus 7 (2013), API Level 21	2 min 42 sec	English, United States	Portrait	–
⊘ Nexus 7 (2013), API Level 22	–	English, United States	Portrait	Incompatible device/API level combination
⊘ Nexus 7 (2013), API Level 22	–	English, United States	Landscape	Incompatible device/API level combination

Performance monitoring

At the most basic level, Android Profiler monitors live CPU, Memory, and Network usage. This allows us to test our app under different conditions and configurations and improve its performance. The Android Profiler can be accessed from via "View" menu with "Tool Window", followed by "Android Profiler" or the tool window bar.

Advanced profiling can easily be enabled via the "Run" menu and "Edit Configurations…"on the "Profiling" tab:


Run/Debug Configurations

Name: MainActivity ☐ Share

General | Miscellaneous | Debugger | Profiling

☑ Enable advanced profiling (required for API level < 26 only)

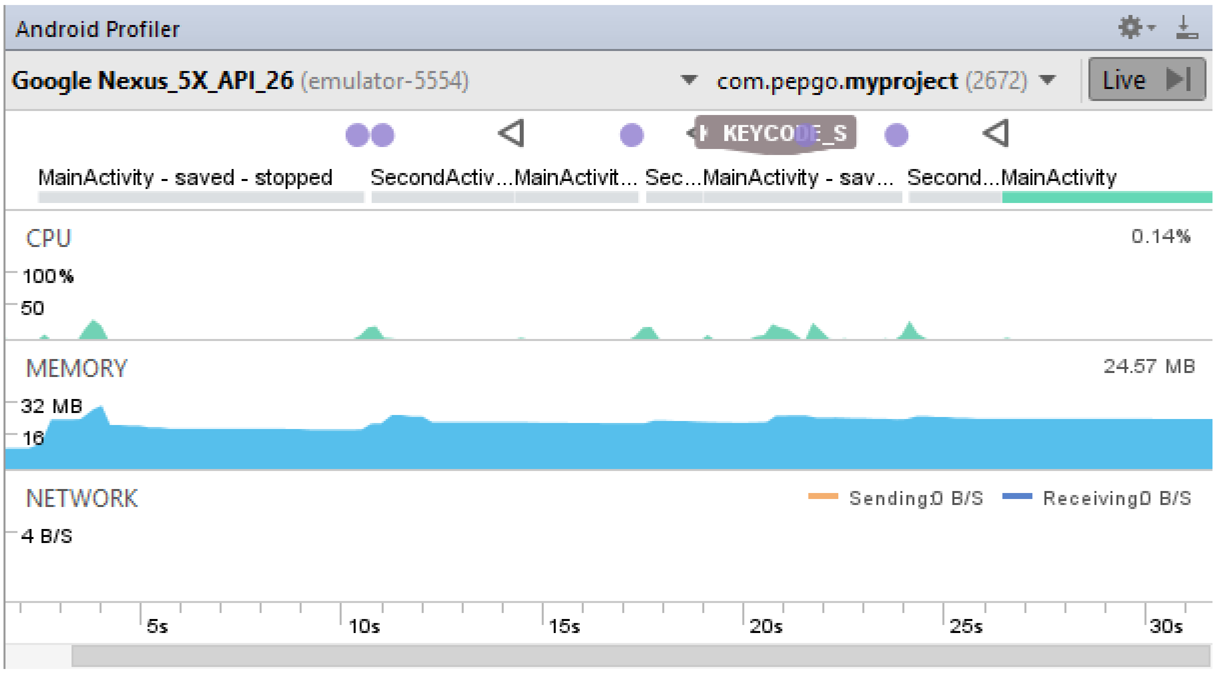
Allows the profilers to track data such as network payloads, application events and object counts, but it might have a minor performance impact on your build speeds.

Before launch: Gradle-aware Make, Instant App Provision

Gradle-aware Make
Instant App Provision

☐ Show this page ☐ Activate tool window

This results in the Android Profiler looking like this:


Android Profiler
Google Nexus_5X_API_26 (emulator-5554)
com.pepgo.myproject (2672)
Live
KEYCODE_S
MainActivity - saved - stopped
SecondActiv...MainActivit...
Sec...MainActivity - sav...
Second...MainActivity
CPU
0.14%
100%
50
MEMORY
24.57 MB
32 MB
16
NETWORK
Sending 0 B/S
Receiving 0 B/S
4 B/S
5s
10s
15s
20s
25s
30s

By clicking on the CPU, Memory, or Network counters, it is possible to drill down into the details.

It is also possible to record periods of activity for detailed analysis. There are two kinds of recording, instrumented and sampled. Their difference is as follows:

- Instrumented recordings take exact timings from when methods are called.
- Sampled recordings take samples of usage at regular intervals.