Android Instrumented Test (Espresso)

This documentation is about testing the UI of Android Applications

Android has mainly two types of tests:

Unit Tests
Unit tests means testing every method/function (or unit) of your code for e.g: given a function, calling it with param x should return y. These tests are run on JVM locally without the need of an emulator or Device.

Instrumentation Testing
Instrumentation tests are used for testing Android Frameworks such as UI,SharedPreferences and So on.Since they are for Android Framework they are run on a device or an emulator. Instrumentation tests use a separate apk for the purpose of testing. Thus, every time a test case is run, Android Studio will first install the target apk on which the tests are conducted. After that, Android Studio will install the test apk which contains only test related code.

What is Espresso?

Espresso is an instrumentation Testing framework made available by Google for the ease of **UI Testing**. Compared to other Android user interface (UI) testing tools, Espresso provides synchronization of test actions with the UI. This means that Espresso knows when the UI thread of an application is idle, and will run test code at the appropriate times. Previously, other workarounds were needed (e.g. sleeping for a certain amount of time before executing the next instruction). Since the release of Espresso v2.0, the testing framework is available as part of the Android Support Repository and now supports Junit4.

Pre-requisites:

- Android Studio
- Java
- Set ANDROID_HOME and JAVA_HOME as environmental variable

After installation of Android studio, open the project in studio, click on ***app***, click on ***src*** folder, you will see ***androidTest*** folder. In androidTest folder, we write our test cases

Setting Up Espresso in Android

Go to your **app/build.gradle**

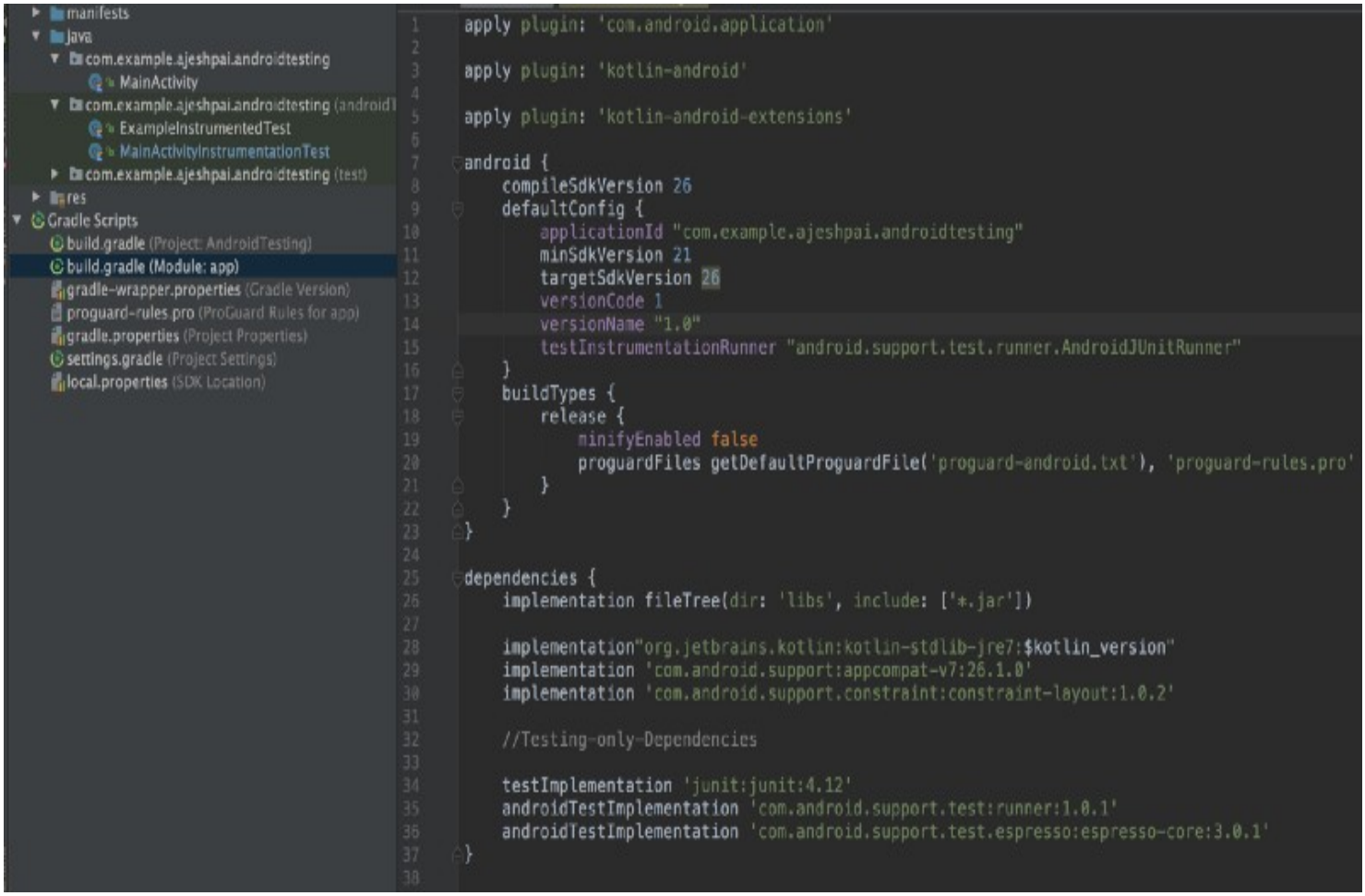
1. Add following dependencies

```
androidTestCompile 'com.android.support.test.espresso:espresso-core:2.2.2', {
    exclude group: 'com.android.support', module: 'support-annotations'
}
androidTestCompile 'com.android.support.test:runner:0.5',{
    exclude group: 'com.android.support', module: 'support-annotations'
}
```

2. Add to the **same build.gradle** file the following line in

```
android{
        ...
        defaultConfig{
                ...
                testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"

        }
}
```

As demonstrated in the image below:



AndroidJUnitRunner is the instrumentation runner. This is essentially the entry point into running your entire suite of tests. It controls the test environment, the test apk, and launches all of the tests defined in your test package.

Note: After adding dependencies in app/build.gradle file, **sync** the project by clicking sync button in order to take its effect.

Getting Started

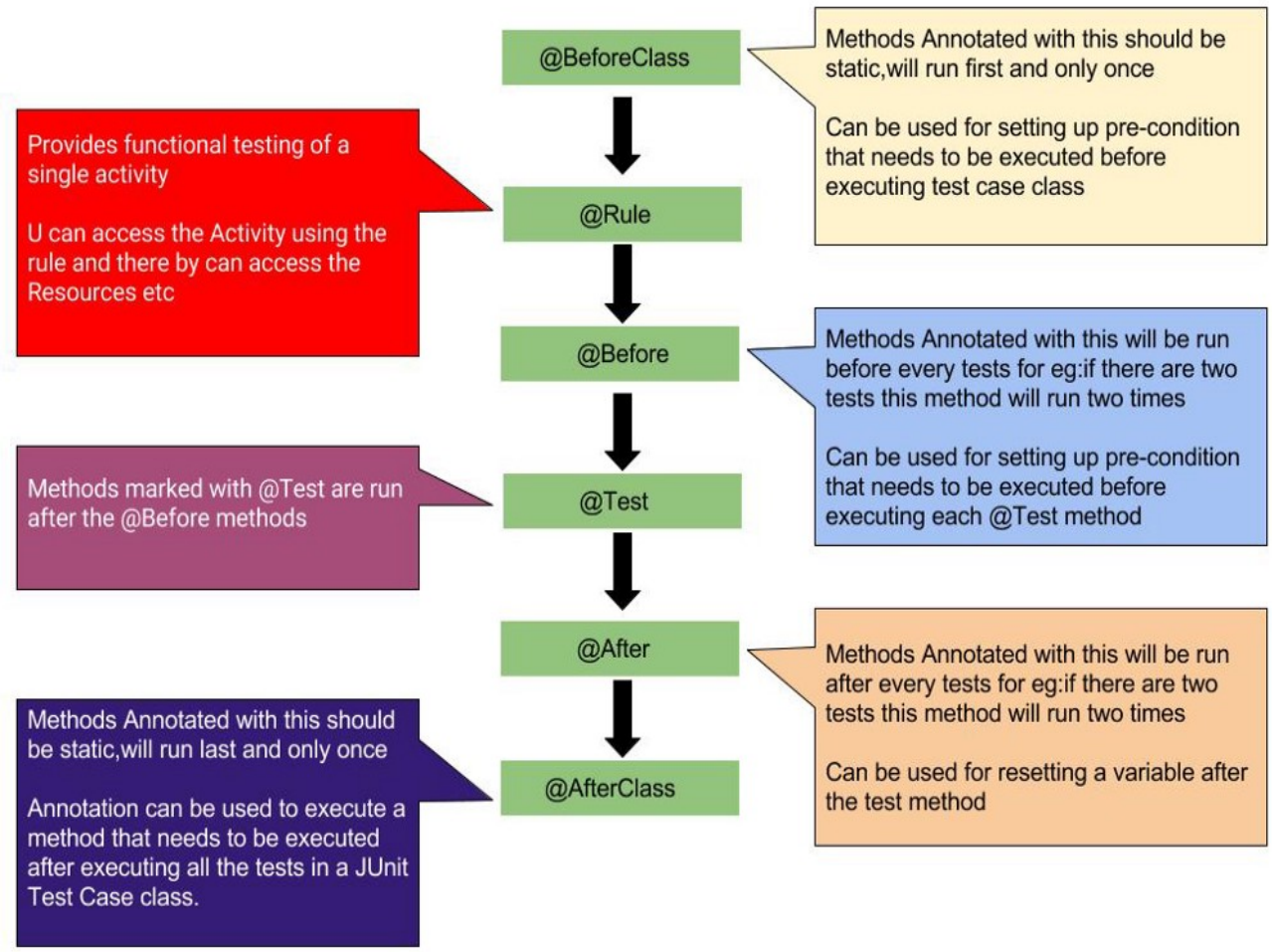
In order to test a UI create a new test class in the Location.

module-name/src/androidTest/java/

And annotate it with **@RunWith(AndroidJUnit4::class)**

The instrumentation runner will process each test class and inspect its annotations. It will determine which *class* runner is set with **@RunWith**, initialize it, and use it to run the tests in that class. In Android's case, the AndroidJUnitRunner explicitly checks if the AndroidJUnit4 class runner is set to allow passing configuration parameters to it.

*There are 6 types of annotations that can be applied to the methods used inside the test class,which are **@Test**, **@Before**, **@BeforeClass**, **@After**, **@AfterClass**, **@Rule***



Important things to note is that

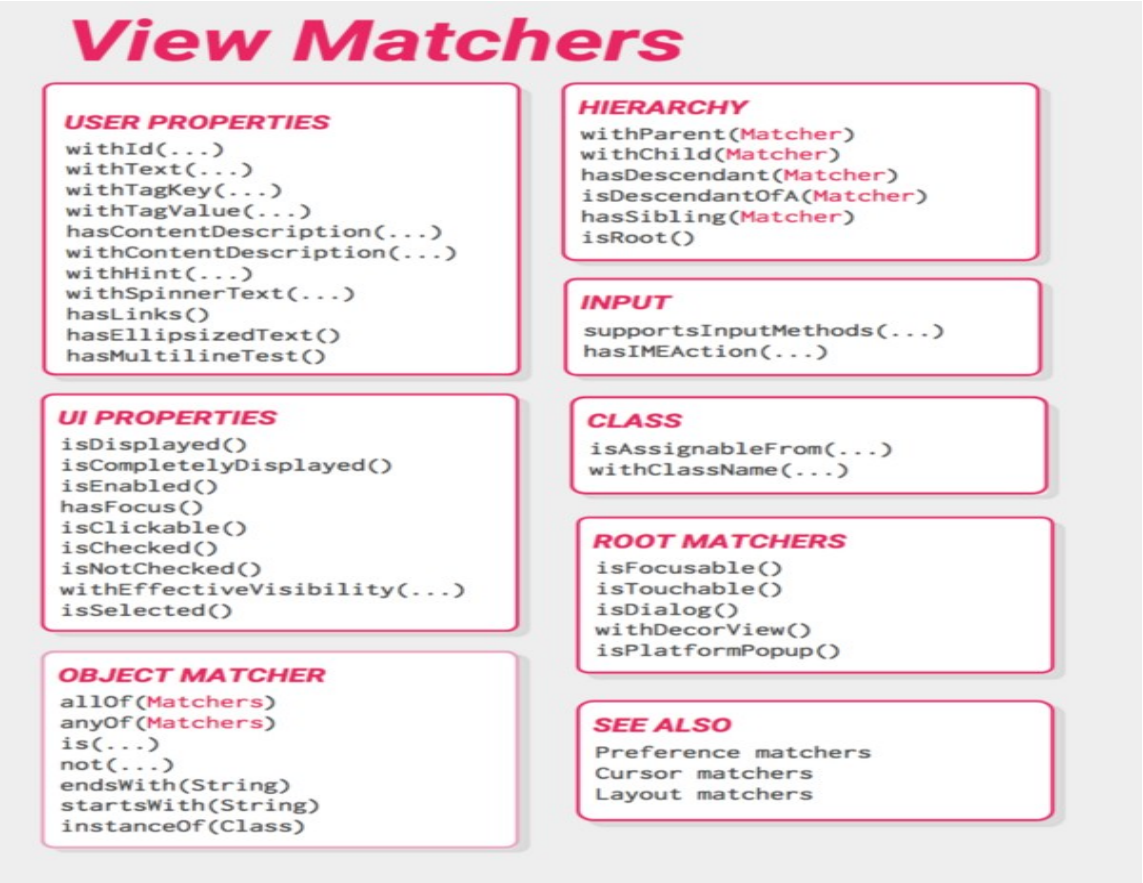
- The activity will be launched using the **@Rule** before test code begins
- By default the rule will be initialised and the activity will be launched(**onCreate, onStart, onResume**) before running every **@Before** method
- Activity will be Destroyed(**onPause, onStop, onDestroy**) after running the **@After** method which in turn is called after every **@Test** Method
- The activity's launch can be postponed by setting the launchActivity to false in the constructor of **ActivityTestRule**, in that case you will have to manually launch the activity before the tests

Note:You can have more than one method annotated with any of the annotations like @Before,@Beforeclass etc. But the order in which JUnit finds methods is not guaranteed.Like if there is two methods annotated with @Before, There is no guarantee which one will be executed.

Writing Tests

The espresso test of a view contains

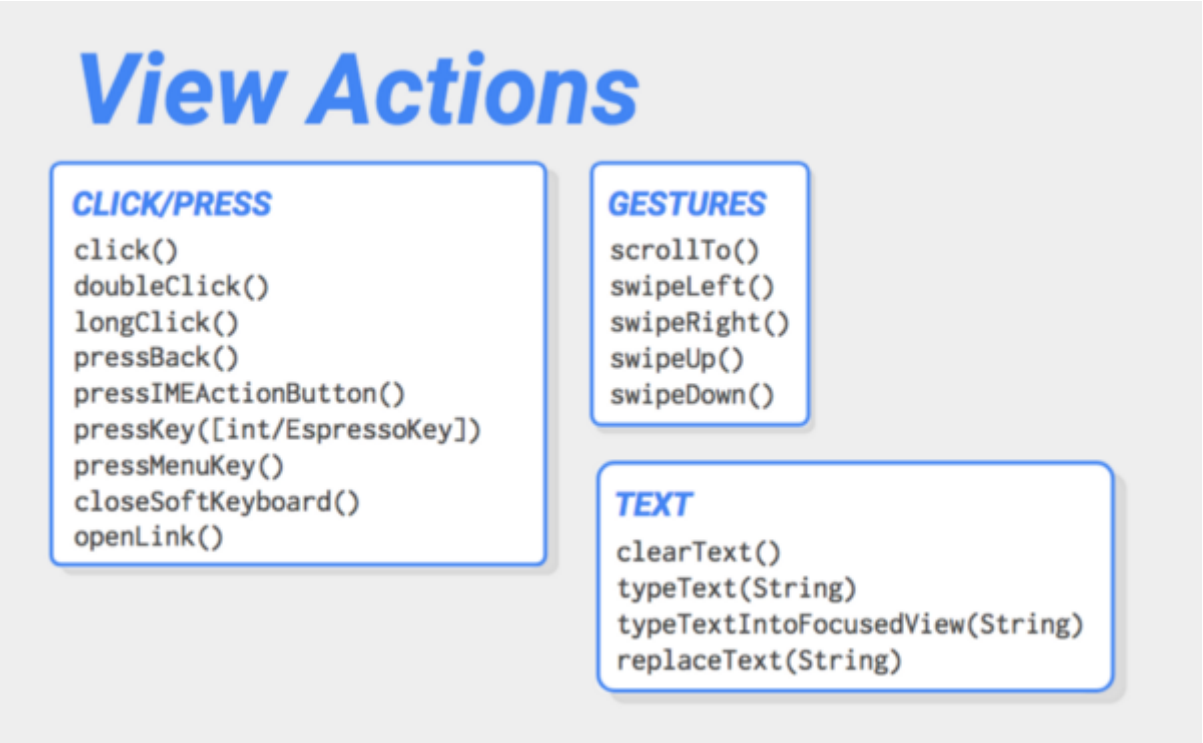
1. Finding a View using a ViewMatcher



Espresso uses **onView()** method to find a particular view among the View hierarchy. **onView()** method takes a **Matcher** as argument. Espresso provides number of **ViewMatchers** as showed in above diagram.
Every UI element contains properties or attributes which can be used to find the element with

android:id="+id/**login_button**"

2. Performing actions on the view



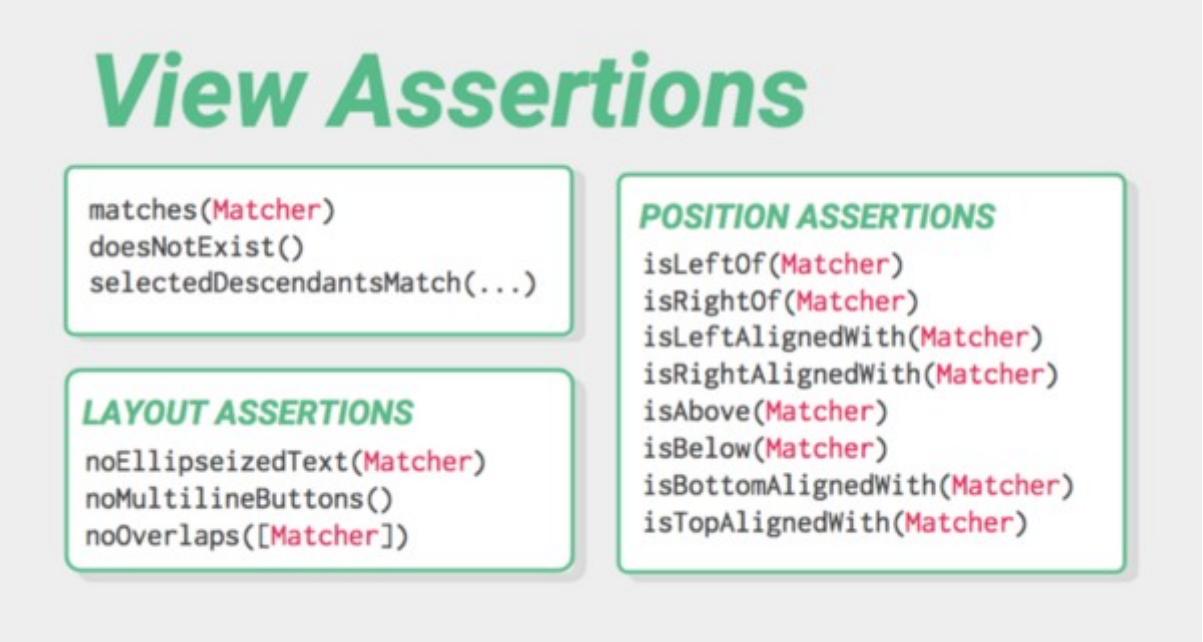
After finding the View you can perform actions on the View or on its Descendant using ViewActions of the Espresso. Some of the Most common actions are click(), clearText() etc.

Note: In cases where the View isn't directly visible such as in a case where the view is in a scroll view and isn't you will have to perform the **scrollTo()** function first and then perform the action.

E.g.

Espresso.onView(withId(R.id.login_button)).**perform(click())**

3. Checking the ViewAssertions



After performing an action on the View we would want to see if the view behaves as we want, this can be done using check*(ViewAssertion viewAssert)*

E.g.

Espresso.onView(withId(R.id.login_result)).**check(matches(withText(R.string.login_success)))**

The following code demonstrates the usage of the Espresso test framework

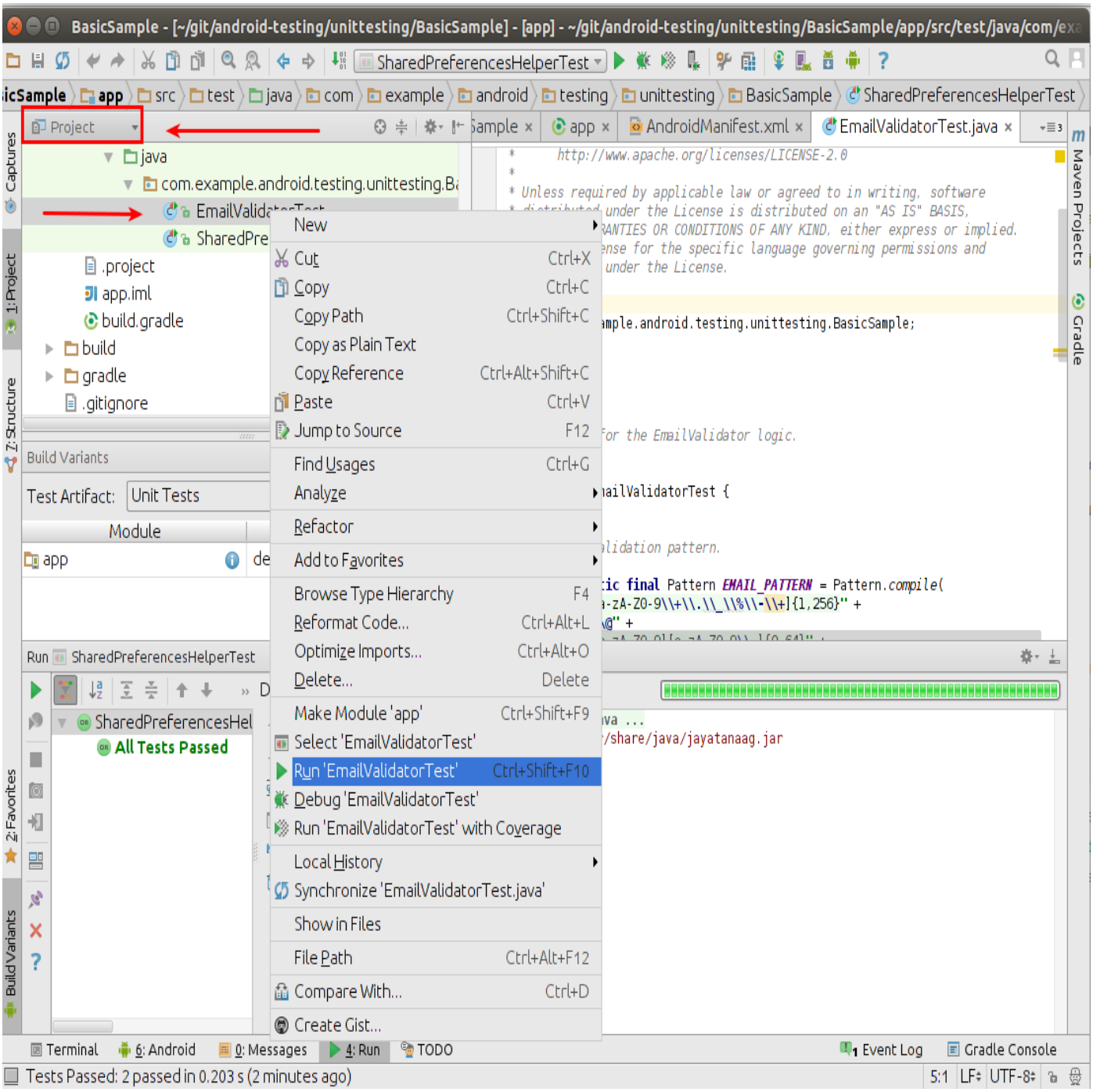
```
package com.vogella.android.espressofirst;

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")));
    }
}
```

If Espresso does not find a view via the ViewMatcher, it includes the whole view hierarchy into the error message. That is useful for analyzing the problem.

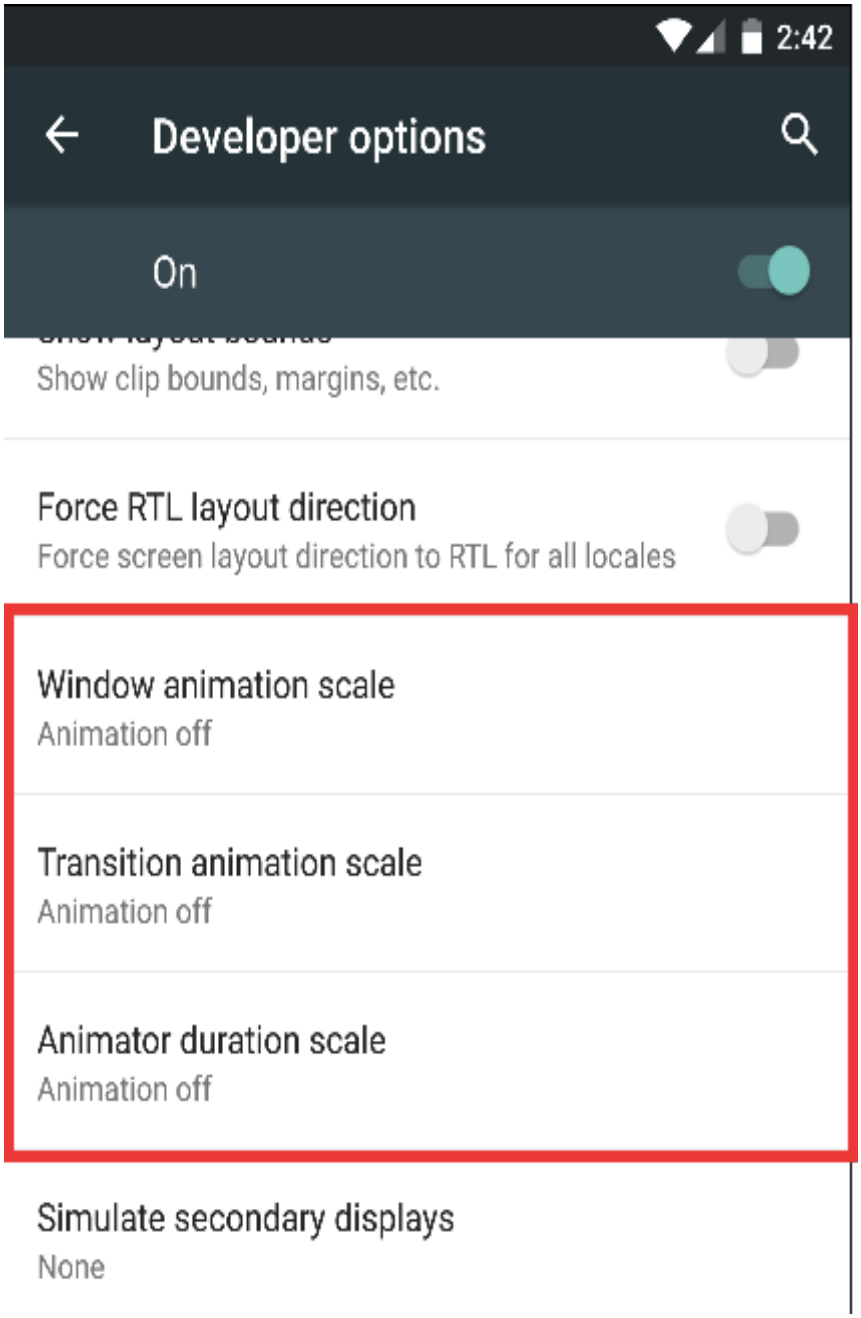
Run your tests

Right click on your test and select Run as below



Device Settings

It is recommended to turn off the animation on the Android device which is used for testing. Animations might confusing Espresso's check for ideling resources.



Note: Your tests may not run properly or result in failure if these options are not off.

ViewMatchers List (Frequently used Matchers)

1. **Using Resource id**
   - **ViewInteraction done = onView(withId(R.id.id_name));**
2. **Using Visible text**
   - **ViewInteraction done = onView(withText(“text”));**
3. **Using Content description**
   - **ViewInteraction done = onView(withContentDescription(“description”));**
4. **Using Hint text**
   - **ViewInteraction done = onView(withHint(“hint text”));**
5. **Using Spinner text**
   - **ViewInteraction done = onView(withSpinnerText(“spinner text”));**
6. **Return TextView with links**
   - **ViewInteraction done = onView(hasLinks());**

Use ViewMatchers with combination

If we are not able to uniquely identify an element with a id or a text etc., then we can use combination of two things and we can locate it. So how we can use that combination?

Using **allOf**

1. **onView(allOf(withId(R.id.id_name), isDisplayed()));**
2. **onView(allOf(withId(R.id.id_name), isCompletelyDisplayed()));**
3. **onView(allOf(withId(R.id.id_name), isClickable()));**
4. **onView(allOf(withId(R.id.id_name), isChecked()));**
5. **onView(allOf(withId(R.id.id_name), isNotChecked()));**
6. **onView(allOf(withId(R.id.id_name), isEnabled()));**
7. **onView(allOf(withId(R.id.id_name), hasFocus()));**
8. **onView(allOf(withId(R.id.id_name), hasLinks()));**
9. **onView(allOf(withId(R.id.id_name), isSelected()));**
10. **onView(allOf(withId(R.id.id_name), hasContentDescription()));**

Object Matchers

1. **onView(withClassName(endsWith(“text”)));**
2. **onView(withText(startsWith(“text”)));**

onData()

onData is used in case of ListView, GridView and AdapterView

onData(withId(“listview_id”)).atPosition(2).check(matches(isDisplayed()));

Root Matchers

1. It matches with text on dialog
    - **onView(withText(R.string.string_name)).inRoot(isDialog());**
2. Matches with Root that takes windows focus
    - **onView(withText(R.string.string_name)).inRoot(isFocusable());**
3. Matches with Root which is autocomplete or action bar spinner
    - **onView(withText(R.string.string_name)).inRoot(isPlatformPopup());**
4. Matches with Root that can handle touch events
    - **onView(withText(R.string.string_name)).inRoot(isTouchable());**
5. Matches with Decor view
    - **onView(withText(R.string.string_name)).inRoot(withDecorView(isDisplayed()));**

Toast Messages

Now a days may app uses Toast Messages and if you are testing android app which displays Toast Messages and you want to write Espresso test case to assert the Toast Message then use this approach.

Check below is the ToastMatcher which identifies the Toast is not part of activity window-

```
public class ToastMatcher extends TypeSafeMatcher<Root> {

    @Override    public void describeTo(Description description) {
        description.appendText("is toast");
    }

    @Override    public boolean matchesSafely(Root root) {
        int type = root.getWindowLayoutParams().get().type;
        if ((type == WindowManager.LayoutParams.TYPE_TOAST)) {
            IBinder windowToken = root.getDecorView().getWindowToken();
            IBinder appToken = root.getDecorView().getApplicationWindowToken();
            if (windowToken == appToken) {
              return true;
            }
        }
        return false;
    }
```

This ToastMatcher you can use it in your test case like this -

1. Test if the Toast Message is Displayed

```
onView(withText(R.string.mssage)).inRoot(new ToastMatcher())
.check(matches(isDisplayed()));
```

2. Test if the Toast Message is not Displayed

```
onView(withText(R.string.mssage)).inRoot(new ToastMatcher())
.check(matches(not(isDisplayed())));
```

3. Test id the Toast contains specific Text Message

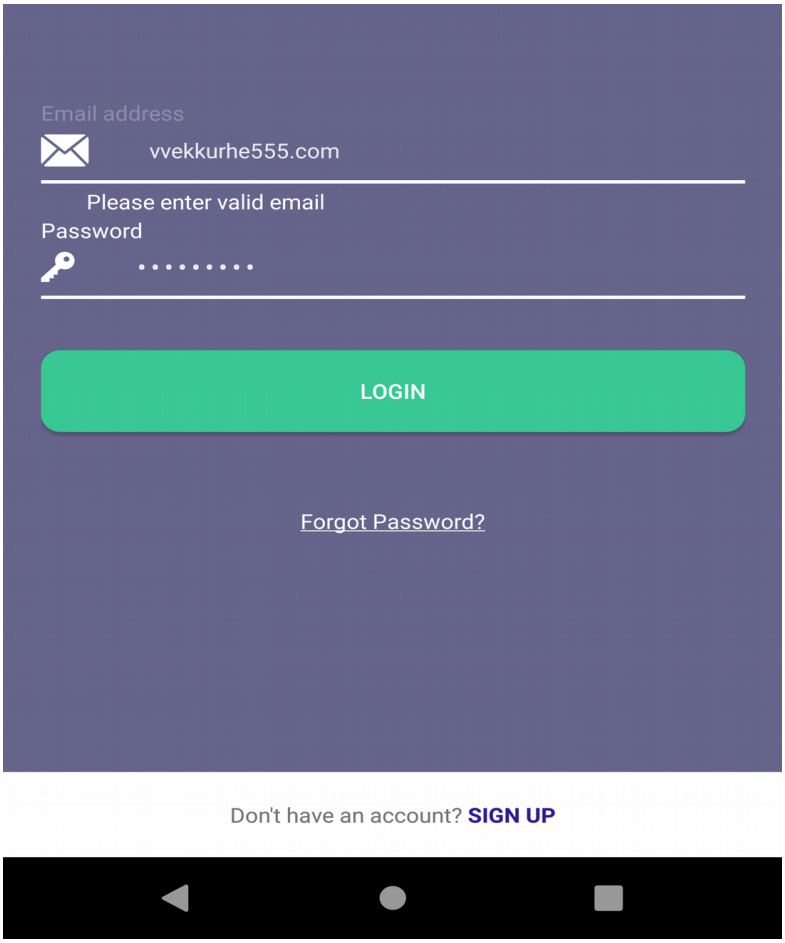
```
onView(withText(R.string.mssage)).inRoot(new ToastMatcher())
.check(matches(withText("Invalid Name"));
```

Selecting a String:

You may encounter a problem where you need to select a particular string for verification so to achieve this you can use **InstrumentationRegistry.**

e.g.

Suppose there are two fields on login screen Emal address and Password. Upon wrong email address it gives error messge "**Please enter valid email**" as shown in below image.



So in this case you can use following approach

String str = InstrumentationRegistry.getTargetContext().getString(R.string.string_name);

Using this you will be able to get a string and assert in following way

assertEquals(expected, str);

Idling Resource

We use Espresso to write effective and reliable UI tests. This testing framework provides a very well maintained set of APIs that ensures a solid and flexible testing environment.

In order to maintain this flexibility, at some point we ought to integrate an implementation of the IdlingResources.

What is an Idling Resource? As Android Developers Documentation states:

An idling resource represents an asynchronous operation whose results affect subsequent operations in a UI test.

Why do you need it?

By registering idling resources with Espresso, you can validate these asynchronous operations more reliably when testing your app.

In plain English, Idling Resources synchronize Espresso tests with your app's background tasks.

If you haven't encountered an Espresso PerformException yet, you might be wondering what's all this fuss about synchronizing Espresso, right?

Let's assume a simple scenario where you want to perform checks on some items within your `ListView`, but only after data has been loaded from your API. Let's try executing some right away like this:

```
@Test
public void contentTest() {
        // quick check if the ListView is visible
        onView(withId(R.id.list)).check(matches(isDisplayed()));

        // click on a certain child view from the loaded list
        onData(allOf()).inAdapterView(withId(R.id.list_news)).atPosition(i)
                  .perform(click());
    }
```

In most of the cases in this scenario, you will end up with an **exception: android.support.test.espresso.PerformException: Error performing 'load adapter data' on view 'with id:'.**

Why is this happening?

In short, while your app was busy trading data in its background threads, your impatient Espresso test wanted to execute some checks immediately. As your request takes some time to process, the test had pretty much no content to interact with, therefore throwing that nasty exception.In this case, trying to perform some checks on affected views when data is being retrieved remotely might ruin your testing experience immediately.

How to fix this?

Obviously, we could solve this by adding an artificial delay (solution illustrated in some of SO answers to this issue) either by adding code like ***SystemClock.sleep(period)*** or ***Thread.sleep(period)***. Yet with this approach, we might end up having inflexible and slow tests—not what we wanted in the first place.

In order to solve this in a clean and effective manner, we can integrate in our application an implementation of the ***IdlingResources***, and for the sake of simplicity, let's make it ***CountingIdlingResource***—one of the easiest to understand resource that comes bundled within the framework.

How does *CountingIdlingResource* work?

This resource acts on the simple concept of incrementing and decrementing. It maintains a counter of active tasks. When the counter is zero, the resource is considered idle, and when the counter is non-zero, the app is busy and therefore the resource is active.

! If you want to have a better understanding on how*CountingIdlingResource*internals work, check the below Extensive and explanatory custom implementation.

! Don't forget to add to your* build.gradle *file the following dependency:

Quick implementation:

A **CountingIdlingResource** can be easily used throughout the app by simply declaring a global variable in the Activity, Presenter or in any other component you are conducting the background tasks:

CountingIdlingResource mIdlingRes = new CountingIdlingResource("name");

We then increment it when Espresso should wait: **mIdlingRes.increment();**and as soon as the operation are finished, tests should resume and we then add **mIdlingRes.decrement();**

Also, we need a method in this specific component to get its resource:

```
public CountingIdlingResource getIdlingResourceInTest() {

    return mIdlingRes;

}
```

Now we're all set to go to use the IdlingResource in the Test:

```
@Test
public void customComponentTest() throws Exception {
    CountingIdlingResource componentIdlingResource =
testedComponent.getIdlingResourceInTest();

    Espresso.registerIdlingResources(componentIdlingResource);

    //perform checks for the specific component below
}
```

Note: *If you still didn't get how to use this resource, you should read the complementary below custom implementation of Espresso*IdlingResource.

Extensive and explanatory custom implementation:

For a better understanding on how CountingIdlingResource mechanism acts, we'll now display a simple custom implementation of EspressoIdlingResourcethat has been extracted right from Google's Android Architecture Blueprints, more precisely: **MVP + Dagger2 + Dagger-Android**.

Let's create a **SimpleCountingIdlingResource** class that implements IdlingResource:

```
import java.util.concurrent.atomic.AtomicInteger;
import static com.google.common.base.Preconditions.checkNotNull;

/**
 * An simple counter implementation of that determines idleness by
 * maintaining an internal counter. When the counter is 0 - it is considered to
be idle, when it is
 * non-zero it is not idle. This is very similar to the way a Semaphore
 * behaves.
 * This class can then be used to wrap up operations that while in progress
should block tests from
 * accessing the UI.
 */
public final class SimpleCountingIdlingResource implements IdlingResource {

    private final String mResourceName;

    private final AtomicInteger counter = new AtomicInteger(0);

    // written from main thread, read from any thread.
    private volatile ResourceCallback resourceCallback;
```

```
    /**
     * Creates a SimpleCountingIdlingResource
     *
     * @param resourceName the resource name this resource should report to
Espresso.
     */
    public SimpleCountingIdlingResource(String resourceName) {
        mResourceName = checkNotNull(resourceName);
    }

    @Override
    public String getName() {
        return mResourceName;
    }

    @Override
    public boolean isIdleNow() {
        return counter.get() == 0;
    }

    @Override
    public void registerIdleTransitionCallback(ResourceCallback
resourceCallback) {
        this.resourceCallback = resourceCallback;
    }

    /**
     * Increments the count of in-flight transactions to the resource being
monitored.
     */
    public void increment() {
        counter.getAndIncrement();
    }

    /**
     * Decrements the count of in-flight transactions to the resource being
monitored.
     *
     * If this operation results in the counter falling below 0 - an exception
is raised.
     *
     * @throws IllegalStateException if the counter is below 0.
     */
```

```
    public void decrement() {
        int counterVal = counter.decrementAndGet();
        if (counterVal == 0) {
            // we've gone from non-zero to zero. That means we're idle now! Tell
espresso.
            if (null != resourceCallback) {
                resourceCallback.onTransitionToIdle();
            }
        }

        if (counterVal < 0) {
            throw new IllegalArgumentException("Counter has been corrupted!");
        }
    }
}
```

Now let’s create the actual custom EspressoIdlingResource:

```
/**
 * Contains a static reference IdlingResource, and should be available only in a
mock build type.
 */
public class EspressoIdlingResource {

       private static final String RESOURCE = "GLOBAL";

       private static SimpleCountingIdlingResource mCountingIdlingResource =
        new SimpleCountingIdlingResource(RESOURCE);

       public static void increment() {
           mCountingIdlingResource.increment();
       }

       public static void decrement() {
           mCountingIdlingResource.decrement();
       }

       public static IdlingResource getIdlingResource() {
           return mCountingIdlingResource;
       }
}
```

How to use it? Assuming the same basic case scenario as above:

You are trying to test and assess some visual interactions (or results) only after some data has been asynchronously loaded from a data source. As mentioned above, trying to do some tests with Espresso right away might exactly get you a **PerformException**.

To avoid this, we can now programatically tell **Espresso** when the app is busy loading that data. In order to achieve this, in your component that is responsible of managing data transactions (in this example: the Presenter—from MVP presentation pattern), you could do something like this:

```
public void loadData() {
   // The network request might be handled in a different thread so make sure Espresso knows
   // that the app is busy until the response is handled.
   EspressoIdlingResource.increment(); // App is busy until further notice

   // let's get the data
   mRepository.getData(new LoadDataCallback() {
      @Override
      public void onDataLoaded(Data data) {
         // now that the data has been loaded, we can mark the app as idle
         // first, make sure the app is still marked as busy then decrement, there might be cases
         // when other components finished their asynchronous tasks and marked the app as
idle
         if (!EspressoIdlingResource.getIdlingResource().isIdleNow()) {
            EspressoIdlingResource.decrement(); // Set app as idle.
         }
         processData(data);
      }
   });
}
```

Now, it is easy to expand this example to other scenarios. As shown in both implementations, we simply add this line before an action (involved in a test) where **Espresso** should wait:

EspressoIdlingResource.increment(); // App is busy until further notice

At the completion of the action, notify **Espresso** that it can resume its tests:

EspressoIdlingResource.decrement(); // Set app as idle.

Let's not forget to set up our implementation of **EspressoIdlingResource** in every test that targets components where **Espresso** has been notified.

You can do this by simply adding these lines when preparing test fixtures:

```
// Register your Idling Resource before any tests regarding this component
@Before
 public void registerIdlingResource() {

IdlingRegistry.getInstance().register(EspressoIdlingResource.getIdlingResource());
}

// Unregister your Idling Resource so it can be garbage collected and does not leak any memory
@After
public void unregisterIdlingResource() {

IdlingRegistry.getInstance().unregister(EspressoIdlingResource.getIdlingResource());
}
```

References:

- https://medium.com/mindorks/android-testing-part-1-espresso-basics-7219b86c862b
- http://www.vogella.com/tutorials/AndroidTestingEspresso/article.html
- http://www.qaautomated.com/2016/01/how-to-test-toast-message-using-espresso.html
- https://android.jlelse.eu/integrate-espresso-idling-resources-in-your-app-to-build-flexible-ui-tests-c779e24f5057