

Test-Driven Development on Android

Testing

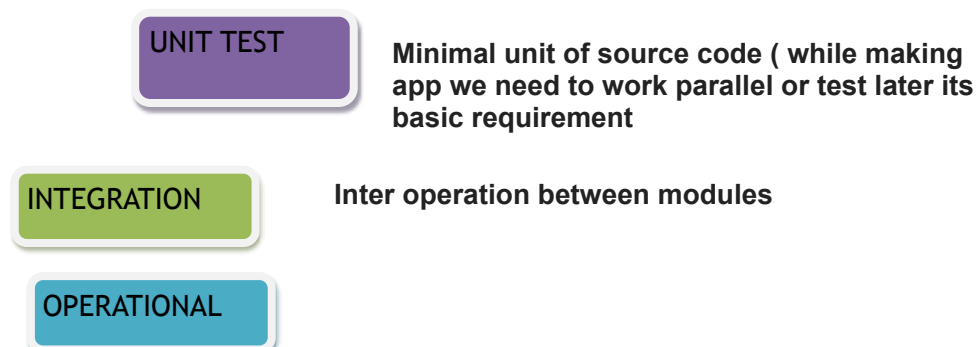
Testing is an important part of software development. By including tests with your code, you can ensure that your code additions work and that later changes don't break them.

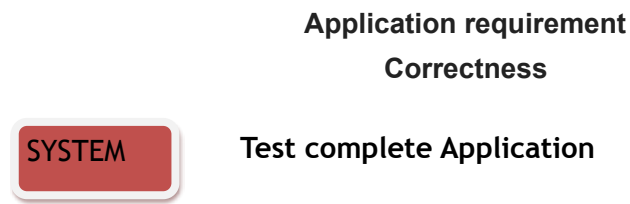
So its requirement whenever you implement the things need to be test .As a programmer we already know maximum chances of bugs where we have done so during work its good to be check those areas of bugs by unit testing .

If we go with testing there are some tests steps that we have to learn today.

When we go with the testing steps we follow some rules ...

1. **Unit Test**
2. **Integration Test**
3. **Operational Test**
4. **System Test**



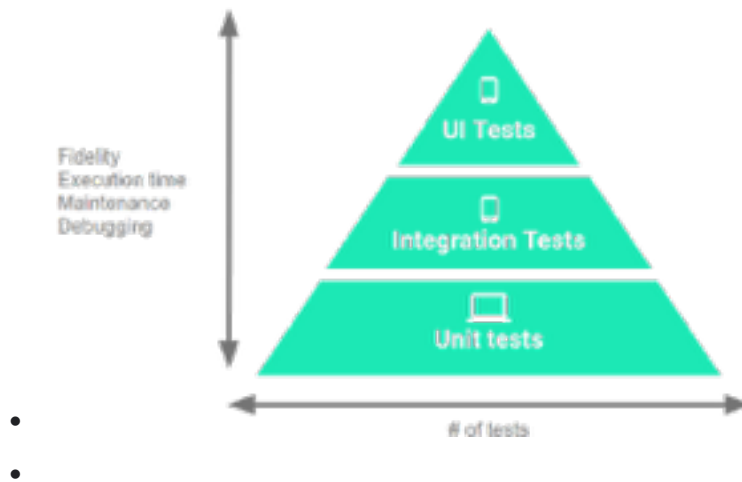


Test-Driven Development, also known as **TDD**, is one way of ensuring to include tests with any new code. When following this process, you write the tests for the thing you are adding before you write the code to implement it.

The Testing Pyramid

how your app should include the three categories of tests: small, medium, and large:

- **Small tests** are unit tests that you can run in isolation from production systems. They typically mock every major component and should run quickly on your machine.
- **Medium tests** are integration tests that sit in between small tests and large tests. They integrate several components, and they run on emulators or real devices.
- **Large tests** are integration and UI tests that run by completing a UI workflow. They ensure that key end-user tasks work as expected on emulators or real devices.
 - small tests are fast and focused, allowing you to address failures quickly, they're also low-fidelity and self-contained, making it difficult to have confidence that a passing test allows your app to work. You encounter the opposite set of tradeoffs when writing large tests.
 - 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 small, 20 percent medium, and 10 percent large.**



AndroidX Test makes use of the following threads:

- The *main thread*, also known as the "UI thread" or the "activity thread", where UI interactions and activity lifecycle events occur.
- The *instrumentation thread*, where most of your tests run. When your test suite begins, the `AndroidJUnitTest` class starts this thread.

If you need a test to execute on the main thread, annotate it using `@UiThreadTest`.

here we have robolectric , mockito , Junit4 ...

Unit tests mainly target smallest functionality (like method, class , component or small module) with isolation from other component.

If your app's testing environment requires unit tests to interact more extensively with the Android framework, you can use [Robolectric](#). This tool executes real Android framework code and fakes of native framework code on your local JVM.

- a simple framework to write repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks.
- [Robolectric](#) is popular Android unit test framework that allows faster test execution by running tests on the JVM (no device or emulator needed 😊).

Robolectric is a framework that brings fast and reliable unit tests to Android. Tests run inside the JVM on your workstation in seconds.

Some other day we shall discuss .

interact with the Android environment here we can use [Mockito](#).

Write medium tests

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. Medium 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.

Medium tests evaluate how your app coordinates multiple units, but they don't test the full app. Examples of medium tests include service tests, integration tests, and hermetic UI tests that simulate the behavior of external dependencies.

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.

The [AndroidJUnitRunner](#) class defines an instrumentation-based [JUnit](#) test runner that lets you run JUnit 3- or JUnit 4-style test classes on Android devices.

Espresso

Espresso synchronizes asynchronous tasks while automating the following in-app interactions, both on your development machine and on real devices:

- Performing actions on [View](#) objects.
- Completing workflows that cross your app's process boundaries. *Available only on Android 8.0 (API level 26) and higher.*
- Assessing how users with accessibility needs can use your app.

- Locating and activating items within [RecyclerView](#) and [AdapterView](#) objects.
- Validating the state of outgoing intents.
- Verifying the structure of a DOM within [WebView](#) objects.
- Tracking long-running background operations within your app.

Espresso is targeted at developers, who believe that automated testing is an integral part of the development lifecycle. While it can be used for black-box testing, Espresso's full power is unlocked by those who are familiar with the codebase under test.

- Some other tools are [UIAutomator](#)—Android UI Test Framework provided by Google for testing across multiple apps at the same time.
- [Robotium](#)—Third party Android UI Test Framework ([Robotium vs Espresso](#))
- [Selendroid](#)—Selenium for Android

Test-Driven Development, also known as **TDD**, is one way of ensuring to include tests with any new code. When following this process, you write the tests for the thing you are adding before you write the code to implement it.

- The import part of Tdd is how we can test on view models
-
- How to write tests for **ViewModel** instances that use **LiveData**, both part of the **Architecture Components** from Google.
 - Unit testing is a practice of writing narrowly scoped automated tests to ensure correctness of your production code. These automated tests allow you to write code that has fewer bugs and is safer to work with in the long term. If you do unit testing, you can write higher quality Android applications that are easier to maintain.
 -
 - **What's test driven development?**

- Test driven development (TDD) is a set of special unit testing techniques that mandate writing the tests before the production code. While TDD might look complicated on the first sight, I'll show you that it's actually the easiest way to do unit testing.
- TDD can be defined as a programming practice that instructs developers to write new code only if an automated test has failed. This avoids duplication of code. TDD means “Test Driven Development”. The primary goal of TDD is to make the code clearer, simple and bug-free.
 - Add a test.
 - Run all tests and see if any new test fails.
 - Write some code.
 - Run tests and Refactor code.
 - Repeat.

TDD or Test Driven Development is a style of programming where you start defining the requirements and writing tests to reflect them before doing any implementation.

After that, we write minimum code necessary in order to make the tests pass.

JUNIT TEST RULES

AndroidX Test includes a set of JUnit rules to be used with the [AndroidJUnitRunner](#). JUnit rules provide more flexibility and reduce the boilerplate code required in tests.

ActivityTestRule

This rule provides functional testing of a single activity. The activity under test is launched before each test annotated with `@Test` and before any method annotated with `@Before`. It's terminated after the test is completed and all methods annotated with `@After` are finished. To access the activity under test in your test logic, call **`ActivityTestRule.getActivity()`**.

```
@RunWith(AndroidJUnit4.class)
@LargeTest
public class MyClassTest {
    @Rule
    public ActivityTestRule<MyClass> activityRule =
```

```

        new ActivityTestRule(MyClass.class);

@Test
public void myClassMethod_ReturnsTrue() { ... }
}

```

ServiceTestRule

This rule provides a simplified mechanism to start up and shut down your service before and after your test. It also guarantees that the service is successfully connected when starting a service or binding to one. The service can be started or bound using one of the helper methods. It automatically stops or unbinds after the test completes and any methods annotated with `@After` are finished.

```

@RunWith(AndroidJUnit4.class)
@MediumTest
public class MyServiceTest {
    @Rule
    public final ServiceTestRule serviceRule = new ServiceTestRule();

    @Test
    public void testWithStartedService() {
        serviceRule.startService(
            new Intent(ApplicationProvider.getApplicationContext(),
                MyService.class));
        // Add your test code here.
    }

    @Test
    public void testWithBoundService() {
        IBinder binder = serviceRule.bindService(
            new Intent(ApplicationProvider.getApplicationContext(),
                MyService.class));
        MyService service = ((MyService.LocalBinder) binder).getService();
        assertTrue(service.doSomethingToReturnTrue());
    }
}

```

In unit testing we need UI test also
Every Widget , menu,button , spinner etc keyevent ,

Integration tests

In [Integration Testing](#), all unit tested modules, are combined and verified. In Android, integration tests often involve checking integration with Android components such as Service testing, Activity testing, Content Provider testing, etc.

Activity Testing ,Service Testing ,Content Provider Testing

Activity Testing :

Activities have complex lifecycle provided by Activity Manager , and UI testing with events

Service Testing :

Test states of service lifecycle and interaction between service and application

Content Provider Testing :

Store and retrieve data and make it accessible across application

Operational tests

Operational are also called Functional Tests or Acceptation Tests. They are high level tests designed to check the completeness and correctness of application.

In Android, **FitNesse** is open-source framework that makes it easy to conduct operational tests for target application.

System tests

In **System Testing** the system is tested as a whole and the interaction between the components, software and hardware is checked.

In Android, System Testing normally includes

GUI tests

Usability tests

Performance tests

Stress test

UNIT TEST :

Simple test techniques , Viewmodel , MVVM , clean architecture layers testing ...

Robolectric is a framework that brings fast and reliable unit tests to Android. Tests run inside the JVM on your workstation in seconds.

Robolectric provides a JVM compliant version of the android.jar file. Robolectric handles inflation of views, resource loading, and lots of other stuff that's implemented in native C code on Android devices.

This enables you to run your Android tests in your continuous integration environment without any additional setup. Robolectric supports resource handling, e.g., inflation of views. You can also use the `findViewById()` to search in a view.

Unit Tests

Why ActivityTestRule give error

These run directly on the JVM and do not have access to the Android framework classes.

They are kept in the test/java package

Dependencies need to be added in the build.gradle file with the command `testCompile`

You generally use Mockito, Robolectric & JUnit for these tests

Instrumentation Tests``

These run on an Android emulator and have full access to all the Android classes

They are kept in the **androidTest/java package**

Dependencies need to be added to build.gradle with androidTestCompile

You generally use Espresso and JUnit for these tests.

Three important implementation we need

androidTestImplementation

'com.android.support:support-annotations:28.0.0'

androidTestImplementation

'com.android.support.test:rules:1.0.2'

androidTestImplementation

'com.android.support.test:runner:1.0.2'

@Test is an annotation provided by JUnit Framework for marking a method as a test case. As you can see here, each method is a test case testing the input field for a possible input. This instructs the compiler to consider the method as a test case in the test suit.

assertEquals(): assertTrue is a method provided by Junit to check input values

assertTrue(): assertTrue is a method provided by Junit Framework to assert (force) the value inside it's parentheses as TRUE. If the value inside the parentheses evaluates to be false, the test case fails.

assertFalse(): Same as the assertTrue method except that it asserts the argument inside the parentheses to be false instead of true. If the passed parameter is true, the test case fails.

@Before: This annotation is used to mark any method to run before executing the test cases.

@After: This annotation is used to mark any method to run after executing the test cases.

```
public boolean isValidEmail(String
emailInput) {
    String EMAIL_PATTERN = "^[_A-Za-z0-9-
\\+]+(\\.[_A-Za-z0-9-]+)*@"
        + "[A-Za-z0-9-]+(\\. [A-Za-
z0-9]+)* (\\. [A-Za-z]{2,})$";

    Pattern pattern =
Pattern.compile(EMAIL_PATTERN);
    Matcher matcher =
pattern.matcher(emailInput);
    return matcher.matches();
}
```

```
public class ActivityTestRule  
extends Object implements TestRule
```

This rule provides functional testing of a single Activity. When `launchActivity` is set to `true` in the constructor, the Activity under test will be launched before each test annotated with `Test` and before methods annotated with `Before`, and it will be terminated after the test is completed and methods annotated with `After` are finished.

During the duration of the test you will be able to manipulate your Activity directly using the reference obtained from `getActivity()`. If the Activity is finished and relaunched, the reference returned by `getActivity()` will always point to the current instance of the Activity.

Mockito

WE shall discuss today what is Mock ..but before that we need to know about
InstantTaskExecutorRule

A JUnit Test Rule that swaps the background executor used by the Architecture Components with a different one which executes each task synchronously.

we need this for testing the new components like live data

androidTestImplementation

"android.arch.core:core-testing:1.1.1"

implementation 'org.mockito:mockito-core:2.8.+'

testImplementation

"android.arch.core:core-testing:1.1.1"

// required if you want to use Mockito for Android tests

androidTestImplementation

'org.mockito:mockito-android:2.8.+'

Mock objects do the mocking of the real service. A mock object returns a dummy data corresponding to some dummy input passed to it.

Mock means not real or authentic

Mockito Annotations

Before hitting the keyboard to write application and unit tests, let's quickly overview the useful mockito annotations.

We have two methods, `mock()` and `spy()`, that can be used to create mock methods or fields.

Using the `mock()` method, we create complete mock or fake objects. The default behaviour of mock methods is to do nothing.

Using the `spy()` method, we just spy or stub specific methods of a real object. As we use real objects in this case, if

we do not stub the method, then the actual method is called.

@Mock is used for mock creation. It makes the test class more readable.

@Spy is used to create a spy instance. We can use it instead `spy(Object)` method.

What is the difference between Mock and Spy

Mock is fake object , we can say a dummy class replacing a real one returning something null or 0 for each method call .

While spy is always wraps real object

@InjectMocks is used to instantiate the tested object automatically and inject all the @Mock or @Spy annotated field dependencies into it (if applicable).

@Captor is used to create an argument captor

APPLY

```
MockitoAnnotations.initMocks(testClass);
```

must be used at least once. To process annotations, we can use the built-in runner MockitoJUnitRunner or rule MockitoRule.

```
@RunWith(MockitoJUnitRunner.class)
```

```
testCompile 'org.mockito:mockito-core:2.8.+'
```

when() is used to configure simple return behaviour for a mock or spy object.

doReturn() is used when we want to return a specific value when calling a method on a mock object. The mocked method is called in case of both mock and spy objects. doReturn() can also be used with methods that don't return any value.

thenReturn() is used when we want to return a specific value when calling a method on a mock object. The mocked method is called in case of mock objects, and real method in case of spy objects. thenReturn() always expects a return type.

for void functions

```
doNothing().when(mock).method()
```

```
doNothing().when(mockObject).methodToStub  
(any());
```

or

```
doReturn().when(mockObject).methodToStub(  
any());
```

```
doAnswer().when(mockObject).methodToStub(  
any());
```

Advantages

We can Mock any class or interface as per our need.

It supports Test Spies, not just Mocks, thus it can be used for partial mocking as well.

Disadvantages

Mockito cannot test static classes. So, if you're using Mockito, it's recommended to change static classes to Singletons. Mockito cannot be used to test a private method.

PowerMockito

less than android 3.2

```
android {  
    ...  
    testOptions {  
        unitTests.returnDefaultValues = true  
    }  
}
```

but its good to update the version of android

ESPRESSO

Use Espresso to write concise , beautiful, and reliable Android UI tests Espresso is targeted at developers who believe that automated testing is an integral part of the development lifecycle . while it can be used for black box testing

Espresso is an instrumentation Testing framework made available by google for the ease of UI Testing

There are six types of annotations that can be applied to the methods used inside the test class which are @Test @Before @BeforeClass @After @AfterClass @Rule Mainly working with rule and test

The activity will be launched using the @Rule before test code begins

By default 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 every @Test method

The activity's launch can be postponed by setting the launch activity to false in

the constructor of `ActivityTestRule` in that case you will have to manually launch the activity before the tests

Espresso – Entry point to interactions with views (via `onView()` and `onData()`). Also exposes APIs that are not necessarily tied to any view, such as `pressBack()`.


ViewMatchers – A collection of objects that implement the `Matcher<? super View>` interface. You can pass one or more of these to the `onView()` method to locate a view within the current view hierarchy.

ViewActions – A collection of `ViewAction` objects that can be passed to the `ViewInteraction.perform()` method, such as `click()`.

ViewAssertions – A collection of `ViewAssertion` objects that can be passed to the `ViewInteraction.check()` method. Most of the time, you will use the `matches` assertion, which uses a `View` matcher to assert the state of the currently selected view.

```
onView(withId(R.id.my_view))
    .perform(click())
    .check(matches(isDisplayed()));
```

```
onView(withId(R.id.my_view));
```



onView(ViewMatcher)

```

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

View Matchers

USER PROPERTIES

- withId(...)
- withClassName(...)
- withPackage(...)
- withResourceName(...)
- withClassNameAndResourceName(...)
- withResource(...)
- withPackageName(...)
- withClassNameAndPackageName(...)

HIERARCHY

- withParent(...)
- withChild(...)
- withDescendant(...)
- withSibling(...)
- withRoot(...)

UI PROPERTIES

- withText(...)
- withTextContains(...)
- withTextNotContains(...)
- withTextMatches(...)
- withTextNotMatches(...)
- withTextMatchesRegex(...)
- withTextNotMatchesRegex(...)

CLASS

- withClassName(...)
- withPackageName(...)

OBJECT MATCHER

- allOf(...)
- anyOf(...)
- not(...)
- notAnyOf(...)
- withResourceName(...)
- withResourceNameAndPackageName(...)

ROOT MATCHERS

- withRoot(...)
- withRootNot(...)
- withRootMatches(...)
- withRootNotMatches(...)

OR ALSO

- Performance matchers
- Custom matchers
- Legacy matchers

onData(ObjectMatcher)

```

        .DataOptions
        .perform(ViewAction)
        .check(ViewAssertion);
    
```

Data Options

- withData(...)
- withDataNot(...)
- withDataMatches(...)

View Actions

CLICK/PRESS

- click(...)
- clickLong(...)
- clickLongTimes(...)
- press(...)
- pressAndRelease(...)
- pressAndReleaseTimes(...)
- pressAndReleaseLong(...)
- pressAndReleaseLongTimes(...)

GESTURES

- swipe(...)
- swipeUp(...)
- swipeDown(...)
- swipeLeft(...)
- swipeRight(...)

View Assertions

LAYOUT ASSERTIONS

- withLayout(...)
- withLayoutNot(...)
- withLayoutMatches(...)
- withLayoutNotMatches(...)

POSITION ASSERTIONS

- withPosition(...)
- withPositionNot(...)
- withPositionMatches(...)
- withPositionNotMatches(...)

intended(IntentMatcher);

Intent Matchers

INTENT

- withIntent(...)
- withIntentCategory(...)
- withIntentAction(...)
- withIntentData(...)
- withIntentDataScheme(...)
- withIntentDataAuthority(...)
- withIntentDataPort(...)
- withIntentDataPath(...)
- withIntentDataSchemeAndAuthority(...)
- withIntentDataSchemeAndPort(...)
- withIntentDataSchemeAndPath(...)
- withIntentDataSchemeAndPortAndPath(...)

URI

- withUri(...)
- withUriScheme(...)
- withUriSchemeAndAuthority(...)
- withUriSchemeAndPort(...)
- withUriSchemeAndPath(...)
- withUriSchemeAndPortAndPath(...)

BUNDLE

- withBundle(...)
- withBundleNot(...)
- withBundleMatches(...)
- withBundleNotMatches(...)

COMPONENT NAME

- withClassName(...)
- withPackageName(...)
- withClassNameAndPackageName(...)
- withClassNameAndPackageNameAndResourceName(...)

intending(IntentMatcher)

```

        .respondWith(ActivityResult);
    
```

v2.1.0, 4/21/2015

```
onView(...).perform(click());
onView(allOf(withId(...),
withText("Hello!"))).check(matches(isDisplayed()));
```

```
Espresso.onView(withId(R.id.login_button)
)
Espresso.onView(withId(R.id.login_button)
).perform(click())
Espresso.onView(withId(R.id.login_result)
).check(matches(withText(R.string.login_s
uccess))))
```

```
Espresso.onView((withId(R.id.textval1)))
    .perform(ViewActions.typeText("vishal"))
    .check(matches(withText("vishal")))
    ;
```

```
Espresso.onView((withId(R.id.textval2)))
    .perform(ViewActions.typeText("vishal"))
    .check(matches(withText("vishal")))
    ;
```

```
Espresso.onView(withId(R.id.clickbtn)).perform(click());
Espresso.onView(withId(R.id.textval))
```

```
        .check(matches(withText("success"
)))
```

androidTestImplementation

```
'com.android.support.test:rules:1.0.2'
```

to avoid flakiness, we highly recommend that you turn off system animations on the virtual or physical devices used for testing. On your device, under Settings > Developer options, disable the following 3 settings:

Window animation scale

Transition animation scale

Animator duration scale

Espresso offers mechanisms to scroll to or act on a particular item for two types of lists: adapter views and recycler views.

Espresso lists (Recyclerview Test)

```
dependencies {
    implementation fileTree(dir: 'libs',
include: ['*.jar'])
    implementation
'com.android.support:appcompat-v7:28.0.0'
    implementation
"com.android.support:design:28.0.0"
    implementation
'com.android.support.constraint:constraint-layout:1.1.3'
    implementation
'com.android.support:recyclerview-v7:28.0.0'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation
'com.android.support.test:runner:1.0.2'
    androidTestImplementation
'com.android.support.test.espresso:espresso-core:3.0.2'

    androidTestImplementation
'com.android.support.test:rules:1.0.2'

    androidTestImplementation("com.android.support.test.espresso:espresso-contrib:2.2.2") {
```



```
        exclude group:
'com.android.support', module:
'appcompat'
        exclude group:
'com.android.support', module: 'support-
v4'
        exclude group:
'com.android.support', module: 'support-
v7'
        exclude group:
'com.android.support', module: 'design'
        exclude module: 'support-
annotations'
        exclude module: 'recyclerview-v7'
    }
```

To interact with RecyclerViews using Espresso, you can use the espresso-contrib package, which has a collection of RecyclerViewActions that can be used to scroll to positions or to perform actions on items:

scrollTo() - Scrolls to the matched View.
scrollToHolder() - Scrolls to the matched View Holder.
scrollToPosition() - Scrolls to a specific position.

actionOnHolderItem() - Performs a View Action on a matched View Holder.
actionOnItem() - Performs a View Action on a matched View.
actionOnItemAtPosition() - Performs a ViewAction on a view at a specific position.

ViewActions to interact RecyclerView. RecyclerView works differently than AdapterView. In fact, RecyclerView is not an AdapterView anymore, hence it can't be used in combination with `onData(Matcher)`.

To use ViewActions in this class use `onView(Matcher)` with a Matcher that matches your RecyclerView, then perform a ViewAction from this class.

<https://developer.android.com/reference/android/support/test/espresso/contrib/RecyclerViewActions>

viewHolderMatcher **Matcher:** a Matcher that matches an item view holder in RecyclerView

viewAction **ViewAction:** the action that is performed on the view matched by **viewHolderMatcher**

```
@Test  
public void  
scrollToItemBelowFold_checkItsText() {  
    // First, scroll to the position that  
needs to be matched and click on it.
```

```
onView(ViewMatchers.withId(R.id.recyclerView))  
    .perform(RecyclerViewActions.  
actionOnItemAtPosition(ITEM_BELOW_THE_FOLD,  
        click())));
```

```
@Test  
public void  
itemInMiddleOfList_hasSpecialText() {  
    // First, scroll to the view holder  
using the isInTheMiddle() matcher.
```

```
Espresso.onView(withId(R.id.recyclerview))  
.perform(
```

```
RecyclerViewActions.actionOnItemAtPosition(0, click()));
```

```
Espresso.onView(withId(R.id.recyclerview)  
).perform(
```

```
RecyclerViewActions.actionOnItemAtPosition  
(1, click())));
```

```
@Test
```

```
    public void  
    itemInMiddleOfList_hasSpecialText() {  
        // First, scroll to the view  
holder using the isInTheMiddle() matcher.
```

```
Espresso.onView(withId(R.id.recyclerview)  
).perform(
```

```
RecyclerViewActions.actionOnItemAtPosition  
(0, click())));
```

```
Espresso.onView(ViewMatchers.withId(R.id.  
recyclerview))
```

```
        .perform(RecyclerViewActi  
ons.actionOnItemAtPosition(ITEM_BELOW_THE  
_FOLD, click())));
```

```
        // Match the text in an item
        below the fold and check that it's
        displayed.
        String itemElementText = ("Inside
        Out");

        Espresso.onView(withText(itemElementText)
        ).check(matches(isDisplayed()));
    }
}
```

When dealing with lists, especially those created with a `RecyclerView` or an `AdapterView` object, the view that you're interested in might not even be on the screen because only a small number of children are displayed and are recycled as you scroll. The `scrollTo()` method can't be used in this case because it requires an existing view.

instead of using the `onView()` method, start your search with `onData()` and provide a matcher against the data that is backing the view you'd like to match. Espresso will do all the work of finding the row in the `Adapter` object and making the item visible in the viewport.

```
onData(allOf(is(instanceOf(Map.class)),
hasEntry(equalTo("STR"), is("item:
50"))))
    .perform(click());
```

RecyclerView objects work differently than AdapterView objects, so onData() cannot be used to interact with them.

Intent Testing Espresso

Espresso-Intents is an extension to Espresso, which enables validation and stubbing of intents sent out by the application under test.

We can work on that using Mockito or Roboelectric and simple Espresso

Espresso

```
@Rule
```

```
public IntentsTestRule<MyActivity>
intentsTestRule =
    new
IntentsTestRule<>(MyActivity.class);
```

Espresso-Intents offers the intended() and intending() methods for intent validation and stubbing

Today we work on Explicit Intents

```
androidTestImplementation  
'com.android.support.test:runner:1.0.2'
```

```
androidTestImplementation  
'com.android.support.test:rules:1.0.2'
```

```
androidTestImplementation  
'com.android.support.test.espresso:espresso-intents:3.0.2'
```

```
androidTestImplementation  
'com.android.support.test.espresso:espresso-core:3.0.2'
```

Intended

Espresso-Intents records all intents that attempt to launch activities from the application under test. Using the `intended()` method, which is similar to `Mockito.verify()`

Sending without any value

```
onView(withId(R.id.submitbutton)).perform  
(click());
```

```
intended(hasComponent(SecondActivity.class.getName()));
```

```
@Rule  
public IntentsTestRule<MainActivity>  
intentsTestRule =  
    new  
IntentsTestRule<>(MainActivity.class);
```

```
@Test  
public void testIntent()  
{  
  
    onView(withId(R.id.senddata)).perform(click());  
  
    intended(hasComponent(SecondActivity.class.getName()));  
  
}
```

if we doing testing without rule it not recognise which activity we doing testing so rule is important.

Sending with values

Put Extra

Using activity test rule

Intending

Using the `intending()` method, which is similar to `Mockito.when()`, you can provide a stub response for activities that are launched with `startActivityForResult()`. This is particularly useful for external activities because you cannot manipulate the user interface of an external activity nor control the `ActivityResult` returned to the activity under test.

```
Intent i = new Intent();
    i.putExtra("name", "vishal");
    //
activityActivityTestRule.launchActivity(i
);
Instrumentation.ActivityResult result=
```

```
        new  
Instrumentation.ActivityResult(Activity.R  
RESULT_OK, i);
```

```
    intending(toPackage(SecondActivity.class.  
getName())) .respondWith(result);
```

```
//  
onView(withId(R.id.textval)).check(matches(isDisplayed()));
```

```
    Espresso.pressBack();  
    Espresso.closeSoftKeyboard();  
    Thread.sleep(2000);
```

```
onView(withId(R.id.senddata1)).perform(click());  
    Thread.sleep(2000);
```

using activity test rule

```
public ActivityTestRule<MainActivity>  
activityActivityTestRule=  
        new  
ActivityTestRule<MainActivity>(MainActivi  
ty.class, true, false);
```

@Test

```
public void shouldShowHelloEarth() throws  
InterruptedException {
```

```
    activityActivityTestRule.launchActivity(c  
        reateIntentWithName("Vishal"));;
```

```
    onView(withId(R.id.senddata)).perform(c  
        lick());  
        Thread.sleep(2000);
```

```
}
```

```
private static Intent  
createIntentWithName(String name) {
```

```
    onView(withId(R.id.senddata)).perform(c  
        lick());
```

```
        Context targetContext =  
        InstrumentationRegistry.getInstrumentatio  
        n().getTargetContext();
```

```
        Intent intent = new  
        Intent(targetContext,  
        MainActivity.class);  
        intent.putExtra("name", name);  
        return intent;
```

}