TDD(Espresso)

TDD(Test-Driven Development):

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. Using TDD when developing an Android app is what you will learn in this tutorial, and by the end, you'll understand:

- The features and uses of TDD.
- Why TDD is useful.
- The steps for practicing TDD.
- How to use TDD in your own projects.
- How to write tests for **ViewModel** instances that use **LiveData**, both part of the **Architecture Components** from Google.

What Is TDD?

TDD is a software-development process. You can apply TDD wherever you practice software development, whether it's iOS, Android, back-end, front-end, embedded, etc. You may have also heard it described as **Red-Green-Refactor**. It's a process in which you write the tests that specify the code you're going to write before you start writing any of the code.

Why Is TDD Important?

There are a number of motives for using TDD, and they all have a lasting impact:

- **Faster development time**: When you have well-written tests, they provide an excellent description of what your code should do. From the start, you have the end goal in mind. Writing these specifications as tests can help keep the result from drifting away from the initial idea.
- **Automatic, up-to-date documentation**: When you're coming into a piece of code, you can look at the tests to help you understand what the code does. Because these

are tests rather than a static document, you can be sure this form of documentation is likely up-to-date!

- **More maintainable code**: When practicing TDD, it encourages you to pay attention to the structure of your code. You will want to architect your app in a way that is easier to test, which is generally cleaner and easier to maintain and read. For example, decoupled classes are easier to set up test classes for, encouraging you to structure your classes this way.

  Refactoring is also built into this development process. By having this refactoring step built in, your code starts off squeaky clean!

- **Greater confidence in your code**: Tests help you to ensure that your code works the way it should. Because of this, you can have greater confidence that what you wrote is "complete." In addition, with any changes that you make in the future, you can know that you didn't break that functionality as long as the tests you wrote with the code are passing.

- **Higher test coverage**: This one is fairly obvious. If you're writing tests along side the code you're writing, you're going to have more test coverage over the code! This is important to many organizations.

The Five Steps of TDD

Through the TDD process, you'll write a number of tests. You usually want a test for the happy path and at least one sad path. If there is a method with a lot of branching, it's ideal to have a test for each of the branches.

You accomplish TDD by following five steps, which you'll walk through over the course of this tutorial:

1. **Add a test**: Anytime you start a new feature, fix or refactor, you write a test for it. This test will specify how this change or addition should behave. You only write the test at this step and just enough code to make it compile.

2. **Run it and watch it fail**: Here, you run the tests. If you did step one correctly, these tests should fail.

3. **Write the code to make the test pass**: This is when you write your feature, fix or refactor. It will follow the specifications you laid down in the tests.

4. **Run the tests and see them pass**: Now, you get to run the tests again! At this point, they should pass. If they don't, go back to step three until all your tests are green!

5. **Do any refactoring**: Now that you have a test that makes sure your implementation matches the specifications, you can adjust and refactor the implementation that you have to ensure that it's clean and structured the way you want without any worries that you'll break what you just wrote.

There are a couple of files that you should locate and become familiar with as they are the ones you'll work with in this tutorial.

- **MainActivity.kt**: This is where you'll put any changes that affect the view.
- **MainActivityTest.kt**: Here is where the tests go for `MainActivity`.
- **VictoryViewModel.kt**: The `ViewModel` will contain the logic you'll work with.
- **VictoryViewModelTest.kt**: Likewise, this is where you'll test `VictoryViewModel`.

You'll also interact with the `VictoryRepository` interface, but you won't need to change anything there. Similarly, you will use `VictoryUiModel` to represent state, but it won't require any changes.

Instrumentation and Unit Tests

Instrumentation tests are for the parts of your code that are dependent on the Android framework but that do not require the UI. These need an emulator or physical device to run because of this dependency. You are using the architecture component `ViewModel`, which requires mocking the `MainLooper` to test, so you will use an instrumentation test for this. These tests go in a `app/src/androidTest/` directory with the same package structure as your project.

A **unit test**, in contrast with an instrumentation test, focuses on the small building blocks of your code. It's generally concerned with one class at a time, testing one function at a time. Unit tests typically run the fastest out of the different kinds of tests, because they are small and independent of the Android framework and so do not need to run on a device or emulator. **JUnit** is usually used to run these tests.

In order to ensure that you're purely testing just the class of interest, you **mock** or **stub** dependencies when writing unit tests. Because unit tests are

independent of the Android framework, they generally go in the `app/src/test/` directory with the same package structure as your project. You can learn all about unit tests on Android here.

You will write your instrumentations tests in this tutorial much like a unit test, with the exception of the ViewModel dependency on the Android framework.

Writing a Failing Instrumentation Test

The first task you're going to complete is implementing the increment victory count. When you tap the **Floating Action Button** in the bottom corner, the behavior you want is that it will increment a count in the star in the center of the screen. If you take a look at the `VictoryViewModel`, you might get a hint of how you will implement it. There is already an `incrementVictoryCount()` method that you will fill in. But first — tests!

Running the Tests

Now that you have your tests written, you can run them and see them fail, following step two of the TDD process. You can run them right in Android Studio a couple of different ways. In the test class file itself, there should be icons in the gutter that you can click to run the full test class or a single test method.

Making the Tests Pass

In the failing tests, you have specified what the code should do, so now you can go on to step three to write the code to implement the feature and make the test pass. Locate the `incrementVictoryCount()` method in `VictoryViewModel`. Add the following code to that method:

```
val newCount = repository.getVictoryCount() + 1

repository.setVictoryCount(newCount)

viewState.value = VictoryUiModel.CountUpdated(newCount)
```

Refactoring

Now that you have written the tests and gotten them passing, this is where you would do any refactoring to make your code as nice as possible while keeping your tests green. This tutorial doesn't have a specific refactor for you for this method, as it is a simple example, but if you are feeling creative you're welcome to refactor this method the way you want as long as your tests are still passing!

Writing a UI Test

UI tests test what the user sees on the screen. They are dependent on the Android framework and need to run on a device or emulator. Like instrumentation tests, they also go in the **androidTest/** directory.

UI Tests are the slowest to run out of all three of these categories of tests, so you want to be selective about what you test with these. You want to test as much of your logic as you can in unit and instrumentation tests, as they run faster.

On Android, UI tests usually use the **Espresso** library to interface with and test the view. Mockito is also sometimes used here.

Something to note when you're adding test library dependencies to your project: In your app module **build.gradle** file, you specify whether the dependency is for an Android test or unit test. If you take a look at the **app/build.gradle** file in this project, you'll see that some dependencies such as JUnit use `testImplementation`, and others, such as Espresso, use `androidTestImplementation`. This matches up with whether the test file is in the **test/** or **androidTest/** folders.

```
testImplementation 'junit:junit:4.12'

// ...

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

1. The Espresso test framework

Espresso is a testing framework for Android to make it easy to write reliable user interface tests.

Google released the Espresso framework in Oct. 2013. Since its 2.0 release Espresso is part of the Android Support Repository.

Espresso automatically synchronizes your test actions with the user interface of your application. The framework also ensures that your activity is started before the tests run. It also let the test wait until all observed background activities have finished.

It is intended to test a single application but can also be used to test across applications. If used for testing outside your application, you can only perform black box testing, as you cannot access the classes outside of your application.

Espresso has basically three components:

- *ViewMatchers* - allows to find view in the current view hierarchy
- *ViewActions* - allows to perform actions on the views
- *ViewAssertions* - allows to assert state of a view

The case construct for Espresso tests is the following

Base Espresso Test

onView(ViewMatcher)

.perform(ViewAction)

.check(ViewAssertion);

- Finds the view
- Performs an action on the view
- Validates a assertioin

The following code demonstrates the usage of the Espresso test framework.

```
import static android.support.test.espresso.Espresso.onView;
import static android.support.test.espresso.action.ViewActions.click;
import static android.support.test.espresso.assertion.ViewAssertions.matches;
```

```
import static android.support.test.espresso.matcher.ViewMatchers.isDisplayed;
import static android.support.test.espresso.matcher.ViewMatchers.withId;

// image more code here...

// test statement
onView(withId(R.id.my_view))          // withId(R.id.my_view) is a ViewMatcher
      .perform(click())              // click() is a ViewAction
      .check(matches(isDisplayed())); // matches(isDisplayed()) is a ViewAssertion

// new test
onView(withId(R.id.greet_button))
.perform(click())
.check(matches(not(isEnabled())));
```

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.

Create project under test

Create a new Android project called *Espresso First* with the package name *com.vogella.android.espressofirst*. Use the *Blank Template* as basis for this project.

Change the generated *activity_main.xml* layout file to the following.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools"

android:layout_width="match_parent"

android:layout_height="match_parent"

>



<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>
```
Create a new file called *activity_second.xml*.
```
<?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>
```

Create a new activity with the following code.

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

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);
    }
}
```

Also adjust your MainActivity class.

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

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;
        }

    }
}
```

Create your Espresso test

```
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")));
}
```

}

More on writing Espresso unit tests

Espresso tests must be placed in the *app/src/androidTest* folder.

To simplify the usage of the Espresso API it is recommended to add the following static imports. This allows to access these methods without the class prefix.

```
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;
```

Using ViewMatcher:

To find a view, use the onView() method with a view matcher which selects the correct view. If you are using an AdapterView use the onData() method instead of the onView() method. The The onView() methods return an object of type ViewInteraction. The onData() method returns an object of type DataInteraction.

Performing Actions

ViewInteraction and DataInteraction allow to specify an action for test via an object of type ViewAction via the perform method. The ViewActions class provides helper methods for the most common actions, like:

- ViewActions.click
- ViewActions.typeText()
- ViewActions.pressKey()
- ViewActions.clearText()

The perform method returns again an object of type ViewInteraction on which you can perform more actions or validate the result. It also uses varags as argument, i.e, you can pass several actions at the same time to it.

Verifying test results

Call the ViewInteraction.check() method to assert a view state. This method expects a ViewAssertion object as input. The ViewAssertions class provides helper methods for creating these objects:

- matches - Hamcrest matcher
- doesNotExist - asserts that the select view does not exist

You can use the powerful Hamcrest matchers. The following gives a few examples:

```
onView(withText(startsWith("ABC"))).perform(click());


onView(withText(endsWith("YYZZ"))).perform(click());


onView(withId(R.id.viewId)).check(matches(withContentDescription(containsString("YYZZ"))));


onView(withText(equalToIgnoringCase("xxYY"))).perform(click());
```

-

```
onView(withText(equalToIgnoringWhiteSpace("XX YY ZZ"))).perform(click());


onView(withId(R.id.viewId)).check(matches(withText(not(containsString("YYZZ")))));
```

Access to the instrumentation API

Via the InstrumentationRegistry.getTargetContext() you have access to the target context of your application. For example, if you want to use the id without using R.id you can use the following helper method to determine it.

Adapter views

AdapterView is a special type of widget that loads its data dynamically from an adapter. Only a subset of the data has real views in the current view hierarchy. A onView() search would not find views for them. onData can be used to interactive with adapter views, like ListView. The following gives a few examples.

Espresso testing with permissions

Via instrumentation you can grand your tests the permission to execute.

```
@Before

public void grantPermission() {

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {

        getInstrumentation().getUiAutomation().executeShellCommand(

                "pm grant " + getTargetContext().getPackageName()

                        + " android.permission.CALL_PHONE");

    }

}
```

Checking for a toast

There is an example how you can click on an list item and check for a toast to be displayed.

```
package com.vogella.android.test.juntexamples;



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.onData;
import static android.support.test.espresso.Espresso.onView;
import static android.support.test.espresso.action.ViewActions.click;
import static android.support.test.espresso.assertion.ViewAssertions.matches;
import static android.support.test.espresso.matcher.RootMatchers.withDecorView;
import static android.support.test.espresso.matcher.ViewMatchers.isDisplayed;
import static android.support.test.espresso.matcher.ViewMatchers.withText;
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.CoreMatchers.is;
import static org.hamcrest.CoreMatchers.startsWith;
import static org.hamcrest.Matchers.greaterThan;
import static org.hamcrest.Matchers.hasToString;
import static org.hamcrest.Matchers.instanceOf;
import static org.hamcrest.Matchers.not;
import static org.hamcrest.Matchers.notNullValue;
import static org.junit.Assert.assertThat;

@RunWith(AndroidJUnit4.class)
public class MainActivityTestList {
```

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


    @Test
    public void ensureListViewIsPresent() throws Exception {
        onData(hasToString(containsString("Frodo"))).perform(click());
        onView(withText(startsWith("Clicked:"))).
        inRoot(withDecorView(
            not(is(rule.getActivity().
            getWindow().getDecorView())))).
            check(matches(isDisplayed()));
    }
}
```

Mocking intents with Espresso Intents

Espresso provides also the option to mock intents. This allows you to check if an activity has issued the correct intents and reacts correct if it receives the correct intent results.

Espressos intents is provided by the `com.android.support.test.espresso:espresso-intents` library. For the setup see Configuration of the Gradle build file for Espresso.

If you want to use Espresso intent in your Espresso tests, use the `IntentsTestRule` instead of `ActivityTestRule`.

```
package testing.android.vogella.com.simpleactivity;
```

```
import android.support.test.espresso.intent.rule.IntentsTestRule;
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.intent.Intents.intended;
import static android.support.test.espresso.intent.matcher.IntentMatchers.toPackage;
import static android.support.test.espresso.matcher.ViewMatchers.withId;



@RunWith(AndroidJUnit4.class)
public class TestIntent {

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

```
    @Test
    public void triggerIntentTest() {
        onView(withId(R.id.button)).perform(click());
        intended(allOf(
                hasAction(Intent.ACTION_CALL),
                hasData(INTENT_DATA_PHONE_NUMBER),
                toPackage(PACKAGE_ANDROID_DIALER)));
    }


}
```

Creating a custom Espresso matcher

Android provides the BoundedMatcher class which allows to create Espresso view matchers for specific view types.

```
public static Matcher<View> withItemHint(String itemHintText) {
  checkArgument(!(itemHintText.equals(null)));
  return withItemHint(is(itemHintText));
}


public static Matcher<View> withItemHint(final Matcher<String> matcherText) {
  // use preconditions to fail fast when a test is creating an invalid matcher.
  checkNotNull(matcherText);
  return new BoundedMatcher<View, EditText>(EditText.class) {
```

```
    @Override
    public void describeTo(Description description) {
      description.appendText("with item hint: " + matcherText);
    }

    @Override
    protected boolean matchesSafely(EditText editTextField) {
      return matcherText.matches(editTextField.getHint().toString());
    }
  };
}
```

4.8. Espresso testing with permissions