Android Developer Fundamentals V2

Testing your UI

Lesson 6

6.1 UI testing

Contents

- UI testing overview
- Test environment and Espresso setup
- Creating Espresso tests
- Espresso test examples
- Recording tests

UI testing overview

UI testing

- Perform all user UI actions with View elements
  - Tap a View, and enter data or make a choice
  - Examine the values of the properties of each View
- Provide input to all View elements
  - Try invalid values
- Check returned output
  - Correct or expected values?
  - Correct presentation?

Problems with testing manually

- Time consuming, tedious, error-prone
- UI may change and need frequent retesting
- Some paths fail over time
- As app gets more complex, possible sequences of actions may grow non-linearly

Benefits of testing automatically

- Free your time and resources for other work
- Faster than manual testing
- Repeatable
- Run tests for different device states and configurations

Espresso for single app testing

- Verify that the UI behaves as expected
- Check that the app returns the correct UI output in response to user interactions
- Navigation and controls behave correctly
- App responds correctly to mocked-out dependencies

UI Automator for multiple apps

- Verify that interactions between different user apps and system apps behave as expected
- Interact with visible elements on a device
- Monitor interactions between app and system
- Simulate user interactions
- Requires instrumentation

What is instrumentation?

- A set of hooks in the Android system
- Loads test package and app into same process, allowing tests to call methods and examine fields
- Control components independently of app's lifecycle
- Control how Android loads apps

Benefits of instrumentation

- Tests can monitor all interaction with Android system
- Tests can invoke methods in the app
- Tests can modify and examine fields in the app independent of the app's lifecycle

Test environment And Espresso setup

Install Android Support Library

1. In Android Studio choose T**ools > Android > SDK Manager**
2. Click **SDK Tools** and look for **Android Support Repository**
3. If necessary, update or install the library

Add dependencies to build.gradle

- Android Studio templates include dependencies
- If needed, add the following dependencies:

```
testImplementation 'junit:junit:4.12'
androidTestImplementation 'com.android.support.test:runner:1.0.1'
androidTestImplementation
          'com.android.support.test.espresso:espresso-core:3.0.1'
```

Add defaultConfig to build.gradle

- Android Studio templates include defaultConfig setting
- If needed, add the following to defaultConfig section:

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

Prepare your device

1. Turn on USB Debugging
2. Turn off all animations in **Developer Options > Drawing**
    - Window animation scale
    - Transition animation scale
    - Animator duration scale

Create tests

- Store in *module-name*/src/androidTests/java/
  - In Android Studio: app > java > *module-name* (androidTest)
- Create tests as JUnit classes

Creating Espresso tests

Test class definition

`@RunWith(AndroidJUnit4.class)` — Required annotation for tests

`@LargeTest` — Based on resources the test uses and time to run

```
public class ChangeTextBehaviorTest {}
```

`@SmallTest` — Runs in < 60s and uses no external resources

`@MediumTest` – Runs in < 300s, only local network

`@LargeTest` – Runs for a long time and uses many resources

@Rule specifies the context of testing

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

@ActivityTestRule – Testing support for a single specified activity

@ServiceTestRule – Testing support for starting, binding, shutting down a service

@Before and @After set up and tear down

```
@Before
public void initValidString() {
    mStringToBetyped = "Espresso";
}
```

@Before – Setup, initializations

@After – Teardown, freeing resources

@Test method structure

```
@Test
public void changeText_sameActivity() {
    // 1. Find a View
    // 2. Perform an action
    // 3. Verify action was taken, assert result
}
```

"Hamcrest" simplifies tests

- “Hamcrest” an anagram of “Matchers”
- Framework for creating custom matchers and assertions
- Match rules defined declaratively
- Enables precise testing
- The Hamcrest Tutorial

Hamcrest Matchers

- ViewMatcher — find Views by id, content, focus, hierarchy
- ViewAction — perform an action on a view
- ViewAssertion — assert state and verify the result

Basic example test

```
@Test
public void changeText_sameActivity() {
    // 1. Find view by Id
    onView(withId(R.id.editTextUserInput))

    // 2. Perform action—type string and click button
    .perform(typeText(mStringToBetyped), closeSoftKeyboard());
    onView(withId(R.id.changeTextBt)).perform(click());

    // 3. Check that the text was changed
    onView(withId(R.id.textToBeChanged))
       .check(matches(withText(mStringToBetyped)));
    }
```

Finding views with onView

- `withId()` — find a view with the specified Android id
  - `onView(withId(R.id.editTextUserInput))`
- `withText()` — find a view with specific text
- `allOf()` — find a view to that matches multiple conditions
- Example: Find a visible list item with the given text:

```
onView(allOf(withId(R.id.word),
             withText("Clicked! Word 15"),
             isDisplayed()))
```

onView returns ViewInteraction object

- If you need to reuse the View returned by onView
- Make code more readable or explicit
- `check()` and `perform()` methods

```
ViewInteraction textView = onView(
    allOf(withId(R.id.word), withText("Clicked! Word 15"),
    isDisplayed()));
textView.check(matches(withText("Clicked! Word 15")));
```

Perform actions

- Perform an action on the View found by a ViewMatcher
- Can be any action you can perform on the View

```
// 1. Find view by Id
onView(withId(R.id.editTextUserInput))

// 2. Perform action—type string and click button
.perform(typeText(mStringToBetyped), closeSoftKeyboard());
onView(withId(R.id.changeTextBt)).perform(click());
```

Check result

- Asserts or checks the state of the View

```
// 3. Check that the text was changed
onView(withId(R.id.textToBeChanged))
   .check(matches(withText(mStringToBetyped)));
```

When a test fails

Test

```
onView(withId(R.id.text_message))
    .check(matches(withText("This is a failing test.")));
```

Result snippet

```
android.support.test.espresso.base.DefaultFailureHandler$Assertion
FailedWithCauseError: 'with text: is "This is a failing test."'
doesn't match the selected view.
Expected: with text: is "This is a failing test."
Got: "AppCompatTextView{id=2131427417, res-name=text_message ...
```

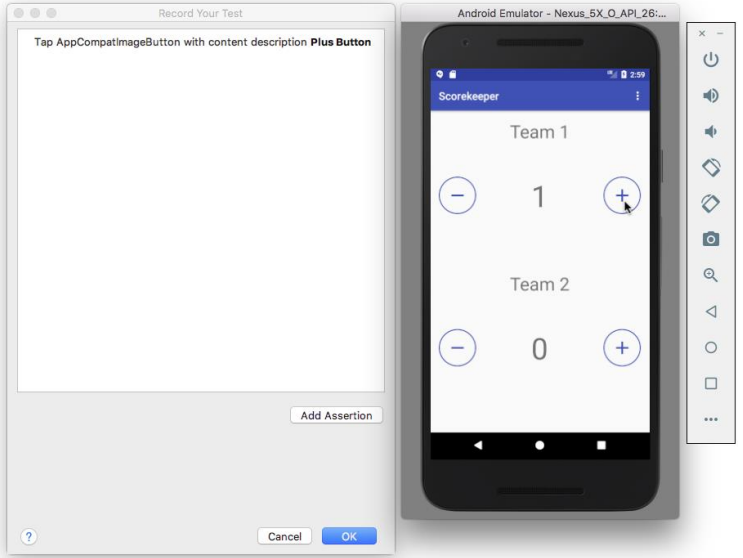
Recording tests

Recording an Espresso test

- Use app normally, clicking through the UI
- Editable test code generated automatically
- Add assertions to check if a view holds a certain value
- Record multiple interactions in one session, or record multiple sessions

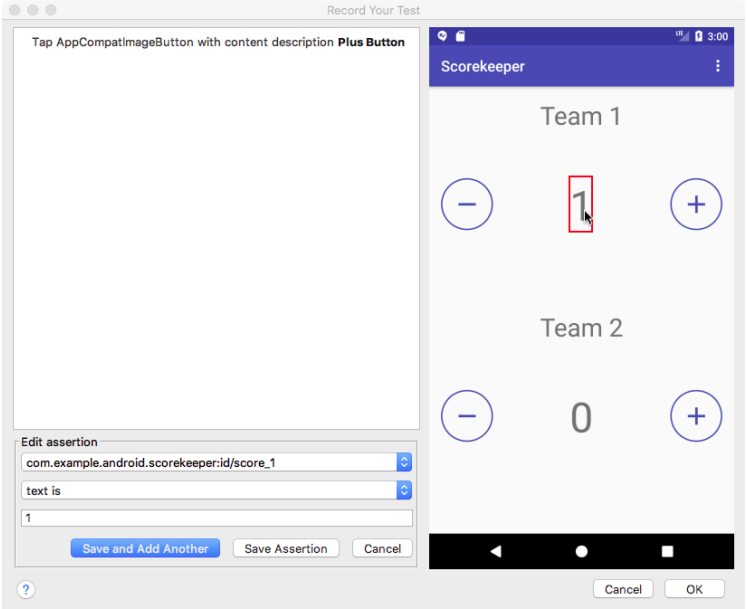
Start recording an Espresso test

1. **Run > Record Espresso Test**
2. Click **Restart app**, select target, and click **OK**
3. Interact with the app to do what you want to test



Add assertion to Espresso test recording

4. Click **Add Assertion** and select a UI element
5. Choose **text is** and enter the text you expect to see
6. Click **Save Assertion** and click **Complete Recording**



Learn more from developer docs

Android Studio Documentation

- Test Your App
- Espresso basics
- Espresso cheat sheet

Android Developer Documentation

- Best Practices for Testing
- Getting Started with Testing
- Testing UI for a Single App
- Building Instrumented Unit Tests
- Espresso Advanced Samples
- The Hamcrest Tutorial
- Hamcrest API and Utility Classes
- Test Support APIs

Learn even more

Android Testing Support Library

- Espresso documentation
- Espresso Samples

Videos

- Android Testing Support - Android Testing Patterns #1 (introduction)
- Android Testing Support - Android Testing Patterns #2 (onView view matching)
- Android Testing Support - Android Testing Patterns #3 (onData & adapter views)

Learn even more

- Google Testing Blog: Android UI Automated Testing
- Atomic Object: "Espresso – Testing RecyclerViews at Specific Positions"
- Stack Overflow: "How to assert inside a RecyclerView in Espresso?"
- GitHub: Android Testing Samples
- Google Codelabs: Android Testing Codelab

What's Next?

- Concept Chapter: 6.1 UI testing
- Practical: 6.1 Espresso for UI testing

END