6.1: Testing the User Interface

Contents:

- Properly testing a user interface
- Setting up your test environment
- Using Espresso for tests that span a single app Using
- UI Automator for tests that span multiple apps

Properly testing a user interface

Writing and executing tests are integral components of the Android app development process. Thoughtfully crafted tests serve as a critical tool for identifying and rectifying issues early in the development cycle, enhancing your confidence in the code's reliability.

User interface (UI) testing t concentrates on assessing elements of the user interface and user interactions. Prioritizing the recognition and response to user input is paramount in UI testing and validation. It's imperative to ensure that your app not only identifies the type of input but also responds appropriately. As a developer, it's essential to establish a routine of testing user interactions to prevent users from encountering unexpected outcomes or having subpar experiences when interacting with your app. UI testing can aid in identifying input controls that require graceful handling of unexpected input or necessitate input validation mechanisms.

Note: We highly recommend using Android Studio for creating your test apps, as it offers streamlined project setup, library integration, and packaging capabilities. Android Studio enables you to execute UI tests on a range of physical or virtual Android devices, facilitating result analysis and code adjustments within the development environment itself.
The user interface (UI) comprises various views housing graphical elements like buttons, menus, and text fields, each possessing a distinct set of properties. To effectively evaluate the UI, it's essential to:

- Exercise all UI events with views:
  - Tap a UI view, and enter data or make a choice.
  - Examine the *values of the properties* of each view—referred to as the *state* of the UI—at different times during execution.
- Provide inputs to all UI views. Use this opportunity to test improper inputs, such as text when numbers are expected.
- Check the outputs and UI representations of data—such as strings and integers—to see if they are consistent with what you are expecting.

In addition to functionality, UI testing evaluates design elements such as layout, colors, fonts, font sizes, labels, text boxes, text formatting, captions, buttons, lists, icons, links, and content.

Manual Testing

As an app developer, you likely conduct manual testing for each UI component as you integrate it into the app's user interface. As the development process advances, one conventional approach to UI testing involves employing human testers to execute a series of user operations within the target app, subsequently validating its correct functionality.

Nonetheless, this manual method can be labor-intensive, monotonous, and prone to errors. When manually testing a complex app's UI, it becomes impractical to encompass all potential user interaction scenarios comprehensively. Moreover, the need for repetitive testing on numerous device configurations within an emulator and across various physical devices further compounds these challenges. In summary, the issues associated with manual testing can be categorized into two primary areas:

Domain size: A user interface (UI) involves a multitude of operations that necessitate testing. Even a seemingly modest app can encompass hundreds of potential UI operations. Across a development cycle, the UI may undergo substantial changes, even when the underlying app remains relatively stable. Manual tests that rely on specific instructions to navigate through the UI may become increasingly unreliable over time. This can occur due to shifts in the UI structure or functionality.

Sequences: Certain app functionalities may require a sequence of user interface (UI) events to achieve. For instance, adding an image to a message about to be sent may involve a series of actions like tapping a camera button, capturing a picture with the camera, or selecting an existing photo by tapping a photo button. Afterward, the user typically associates that picture with the message, often through the interaction of a share or send button. Expanding the array of potential operations further complicates the challenge of sequencing these actions correctly.

Automated testing

Automating tests for user interactions liberates your time and resources for more productive tasks. Test designers aim to create a comprehensive suite of test cases that encompasses the system's entire functionality and rigorously exercises the user interface (UI). Automating these UI interactions simplifies the process of running tests across diverse device states, including various orientations, and under different configurations.

For testing Android apps, you typically create these types of automated UI tests:

- *UI tests that work within a single app:* Verifies that the app behaves as expected when a user performs a specific action or enters a specific input. It allows you to check that the app returns the correct UI output in response to user interactions in the app's activities. UI testing frameworks like Espresso allow you to programmatically simulate user actions and test complex intra-app user interactions.
- *UI tests that span multiple apps:* Verifies the correct behavior of interactions between different user apps or between user apps and system apps. For example, you can test an app that launches the Maps app to show directions, or that launches the Android contact picker to select recipients for a message. UI testing frameworks that support cross-app interactions, such as UI Automator, allow you to create tests for such user-driven scenarios.

Using Espresso for tests that span a single app

The Espresso testing framework in the Android Testing Support Library provides APIs for writing UI tests to simulate user interactions within a single app. Espresso tests run on actual device or emulator and behave as if an actual user is using the app.

You can use Espresso to create UI tests to automatically verify the following:

- The app returns the correct UI output in response to a sequence of user actions on a device.
- The app's navigation and input controls bring up the correct activities, views, and fields.
- The app responds correctly with mocked-up dependencies, such as data from an outside server, or can work with stubbed out backend methods to simulate real interactions with backend components which can be programmed to reply with a set of defined responses.

A key benefit of using Espresso is that it has access to instrumentation information, such as the context for the app, so that you can monitor all of the interaction the Android system has with the app. Another key benefit is that it automatically synchronizes test actions with the app's UI. Espresso detects when the main thread is idle, so it is able to run your test at the appropriate time, improving the reliability of your tests. This capability also relieves you from having to add any timing workarounds, such as a sleep period, in your test code.

The Espresso testing framework works with the AndroidJUnitRunner test runner and requires instrumentation, which is described later in this section. Espresso tests can run on devices running Android 2.2 (API level 8) and higher.

Using UI Automator for tests that span multiple apps

The UI Automator testing framework within the Android Testing Support Library serves as a valuable tool for validating the accurate functioning of interactions between various user apps or between user apps and system apps. It not only offers insights into events occurring on the device both before and after an app launch but also facilitates interaction with visible elements on the device using UI Automator APIs. These APIs allow tests to identify UI components using descriptors like the displayed text or content description. Furthermore, a viewer tool provides a user-friendly interface for scrutinizing the layout hierarchy and inspecting the properties of visible UI components within the device's foreground.

The following are important functions of UI Automator:

- Like Espresso, UI Automator has access to system interaction information so that you can monitor all of the interaction the Android system has with the app.
- Your test can send an Intent or launch an Activity (without using shell commands) by getting a Context object through getContext().
- You can simulate user interactions on a collection of items, such as songs in a music album or a list of emails in an inbox.
- You can simulate vertical or horizontal scrolling across a display.
- You can use standard JUnit Assert methods to test that UI components in the app return the expected results.

The UI Automator testing framework works with the AndroidJUnitRunner test runner and requires instrumentation, which is described in the next section. UI Automator tests can run on devices running Android 4.3 (API level 18) or higher.

What is instrumentation?

Android instrumentation comprises a collection of control methods, often referred to as hooks, embedded within the Android system. These hooks govern the behavior of Android components and dictate how the Android system loads apps.

By default, the system runs all components of an app within a single process. While it is possible to allow certain components, such as content providers, to operate in a separate process, it is generally not feasible to compel an app to share a process with another concurrently running app.

However, in the context of instrumentation tests, both a test package and the app can be loaded into the same process. This coexistence of app components and their corresponding tests within a shared process enables your tests to call methods within these components and interact with their fields, facilitating the ability to make modifications and perform examinations as necessary.

Instrumentation provides a means to observe and oversee all interactions between the Android system and the application. It empowers tests to call methods within the app, manipulate its components, and inspect its fields autonomously, regardless of the app's typical lifecycle.

Normally, an Android component runs in a lifecycle that the system determines. For example, an Activity object's lifecycle starts when an Intent activates the Activity. The system calls the object's onCreate() method, and then the onResume() method. When the user starts another app, the system calls the onPause() method. If the Activity code calls the finish() method, the system calls the onDestroy() method.

The Android framework API does not provide a way for your app's code to invoke these callback methods directly, but you can do so using an Espresso or UI Automator test with instrumentation.

Setting up your test environment

To use the Espresso and UI Automator frameworks, you need to store the source files for instrumented tests at ***module-name*/src/androidTests/java/**. This directory already exists when you create a new Android Studio project. In the Project view of Android Studio, this directory is shown in **app > java** as ***module-name*** **(androidTest)**.

You also need to do the following:

- Install the Android Support Repository and the Android Testing Support Library. Add
- dependencies to the project's build.gradle file.
- Create test files in the **androidTest** directory.

Installing the Android Support Repository and Testing Support Library

You may already have the Android Support Repository and its Android Testing Support Library installed with Android Studio. To check for the Android Support Repository, follow these steps:

1. In Android Studio choose **Tools > Android > SDK Manager**.
2. Click the **SDK Tools** tab, and look for the Support Repository.

3. If necessary, update or install the library.

Adding the dependencies

When you start a project for the Phone and Tablet form factor using **API 15: Android 4.0.3 (Ice Cream Sandwich)** as the minimum SDK, Android Studio version 2.2 and newer automatically includes the dependencies you need to use Espresso. To ensure that you have these dependencies, follow these steps:

1. Open the **build.gradle (Module: app)** file in your project to make sure the following is included (along with other dependencies) in the `dependencies` section of your **build.gradle (Module: app)** file:

```
androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
testCompile 'junit:junit:4.12'
```

2. Android Studio also adds the following instrumentation statement to the end of the `defaultConfig` section:

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

Note: If you created your project in a previous version of Android Studio, you may have to add the above dependencies and instrumentation statement yourself.

1. When finished, click the **Sync Now** link in the notification about Gradle files in top right corner of the window.

Setting up the test rules and annotations

To write tests, Espresso and UI Automator use JUnit as their testing framework. JUnit is the most popular and widely-used unit testing framework for Java. Your test class using Espresso or UI Automator should be written as a JUnit 4 test class. If you do not have JUnit, you can get it at http://junit.org/junit4/.

Note: The most current JUnit revision is JUnit 5. However for the purposes of using Espresso or UI Automator, version 4.12 is recommended.
To create a basic JUnit 4 test class, create a Java class for testing in the directory specified at the beginning of this section. It should contain one or more methods and behavior rules defined by JUnit annotations.

For example, the following snippet shows a test class definition with annotations:

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

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

   @Test
   public void recyclerViewTest() {
      ...
   }
}
```

The following annotations are useful for testing:

@RunWith

To create an instrumented JUnit 4 test class, add the `@RunWith(AndroidJUnit4.class)` annotation at the beginning of your test class definition, which indicates the runner that will be used to run the tests in this class. A test runner is a library or set of tools that enables testing to occur and the results to be printed to a log.

@SmallTest, @MediumTest, and @LargeTest

The , , and Android annotations provide some clarity about what resources and `@SmallTest` `@MediumTest` `@LargeTest` features the test uses. For example, the o `@SmallTest` annotation tells you that the test doesn't interact with any file system network.

The following summarizes what they mean:

Feature	Small	Medium	Large
Network access	No	localhost only	Yes
Database	No	Yes	Yes
File system access	No	Yes	Yes
Use external systems	No	Discouraged	Yes
Multiple threads	No	Yes	Yes
Sleep statements	No	Yes	Yes
System properties	No	Yes	Yes
Time limit (seconds)	60	300	900+

For a description of the Android `@SmallTest` `@MediumTest`, and `@LargeTest` annotations, see "Test Sizes" in the Google Testing Blog. For a summary of JUnit annotations, see Package org.junit.

@Rule

Before declaring the test methods, use the `@Rule` annotation, such as ActivityTestRule or ServiceTestRule. The `@Rule` establishes the context for the testing code. For example:

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

This rule uses an `ActivityTestRule` object, which provides functional testing of a single Activity—in this case, `MainActivity.class`.

`ServiceTestRule` is a JUnit rule that provides a simplified mechanism to start and shutdown your service before and after the duration of your test. It also guarantees that the service is successfully connected when starting (or binding to) a service.

@Test

A test method begins with the `@Test` annotation and contains the code to exercise and verify a single function in the component that you want to test:

```
@Test
public void testActivityLaunch() { … }
```

The activity under test will be launched before each test annotated with `@Test`. During the duration of the test you will be able to manipulate your Activity directly.

@Before and @After

In certain rare cases you may need to set up variables or execute a sequence of steps *before* or *after* performing the user interface test. You can specify methods to run before running a `@Test` method, using the `@Before` annotation, and methods to run after, using the `@After` annotation.

- `@Before` : The `@Test` method will execute after the methods designated by the `@Before` annotation. The `@Before` methods terminate prior to the execution of the `@Test` method.
- `@After` : The `@Test` method will execute before the methods designated by the `@After` annotation.

Using Espresso for tests that span a single app

When writing tests it is sometimes difficult to get the balance right between over-specifying the test or not specifying enough. Over-specifying the test can make it brittle to changes, while under-specifying may make the test less valuable, since it continues to pass even when the UI element and its code under test is broken.

To make tests that are balanced, it helps to have a tool that allows you to pick out precisely the aspect under test and describe the values it should have. Such tests fail when the behavior of the aspect under test deviates from the expected behavior, yet continue to pass when minor, unrelated changes to the behavior are made. The tool available for Espresso is the Hamcrest framework.

Hamcrest (an anagram of "matchers") is a framework that assists writing software tests in Java. The framework lets you create custom assertion matchers, allowing match rules to be defined declaratively.

Tip: To learn more about Hamcrest, see The Hamcrest Tutorial.

Writing Espresso tests with Hamcrest Matchers

You write Espresso tests based on what a user might do while interacting with your app. The key concepts are *locating* and then *interacting* with UI elements. These are the basic steps:

1. **Match a view:** Find a view.
2. **Perform an action:** Perform a click or other action that triggers an event with the view.
3. **Assert and verify the result:** Check the view's state to see if it reflects the expected state or behavior defined by the assertion.

To create a test method, you use the following types of Hamcrest expressions to help find views and interact with them:

- *ViewMatchers:* A ViewMatcher expression to use with `onView()` to match a view in the current view hierarchy. Use `onView()` with a ViewMatcher so that you can examine something or perform some action. The most common ones are:
  - `withId()` : Find a view with a specific `android:id` (which is typically defined in the layout XML file). For example:

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

  - `withText()` : Find a view with specific text, typically used with `allOf()` and `withId()`. For example, the following uses `allOf` to cause a match if the examined object matches *all* of the specified conditions—the view uses the id (`word` `withId`), and the view has the text `"Clicked! Word 15"` (`withText`):

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

  - Others including matchers for state (`selected` , `focused` , `enabled`), and content description and hierarchy (`root` and `children`).
- *ViewActions:* A ViewAction expression lets you perform an action on the view already found by a ViewMatcher. The action can be any action you can perform on the view, such as a click. For example:

  ```
  .perform(click())
  ```

- *ViewAssertions:* A ViewAssertion expression lets you assert or checks the state of a view found by a ViewMatcher. For example:

```
.check(matches(isDisplayed()))
```

You would typically combine a ViewMatcher and a ViewAction in a single statement, followed by a ViewAssertion expression in a separate statement or included in the same statement.

You can see how all three expressions work in the following statement, which combines a ViewMatcher to find a view, a ViewAction to perform an action, and a ViewAssertion to check if the result of the action matches an assertion:

```
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
```

Why is the Hamcrest framework useful for tests? A simple assertion, such as `assert (x == y)`, lets you assert during a test that a particular condition must be true. If the condition is false, the test fails. But the simple assertion provides no useful error message. With a family of assertions, you can produce more useful error messages, but this leads to an explosion in the number of assertions.

With the Hamcrest framework, it is possible to define operations that take matchers as arguments and return them as results, leading to a grammar that can generate a huge number of possible matcher expressions from a small number of primitive matchers.

For a Hamcrest tutorial, see The Hamcrest Tutorial. For a quick summary of Hamcrest matcher expressions, see the Espresso cheat sheet.

Testing an AdapterView

In an AdapterView such as a spinner, the view is dynamically populated with child views at runtime. If the target view you want to test is inside a spinner, the `onView()` method might not work because only a subset of the views may be loaded in the current view hierarchy.

Espresso handles this by providing a separate `onData()` method, which is able to first load the adapter item and bring it into focus prior to operating on it or any of its children views. The `onData()` method uses a DataInteraction object and its methods, such as `atPosition()`, `check()`, and `perform()` to access the target view. Espresso handles loading the target view element into the current view hierarchy, scrolling to the target child view, and putting that view into focus.

For example, the following `onView()` and `onData()` statements test a spinner item click:

1. Find and click the spinner itself (the test must first click the spinner itself in order click any item in the spinner):

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

2. Find and then click the item in the spinner that matches *all* of the following conditions:
   - An item that is a `String`
   - An item that is equal to the String "Americano"

```
onData(allOf(is(instanceOf(String.class)),
                        is("Americano"))).perform(click());
```

As you can see in the above statement, matcher expressions can be combined to create flexible expressions of intent:

- `allOf`: Causes a match if the examined object matches *all* of the specified matchers. You can use `allOf()` to combine multiple matchers, such as `containsString()` and `instanceOf()`.
- `is`: Hamcrest strives to make your tests as readable as possible. The `is` matcher is a wrapper that doesn't add any extra behavior to the underlying matcher, but makes your test code more readable.
- `instanceOf`: Causes a match when the examined object is an instance of the specified type; in this case, a string. This match is determined by calling the Class.isInstance(Object) method, passing the object to examine.

The following example illustrates how you would test a spinner using a combination of `onView()` and `onData()` methods:

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

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

   @Test
   public void iterateSpinnerItems() {
       String[] myArray = mActivityRule.getActivity().getResources()
                               .getStringArray(R.array.labels_array);

      // Iterate through the spinner array of items.
      int size = myArray.length;
      for (int i=0; i<size; i++) {
         // Find the spinner and click on it.
         onView(withId(R.id.label_spinner)).perform(click());
         // Find the spinner item and click on it.
         onData(is(myArray[i])).perform(click());
         // Find the button and click on it.
         onView(withId(R.id.button_main)).perform(click());
         // Find the text view and check that the spinner item
         // is part of the string.
         onView(withId(R.id.text_phonelabel))
                    .check(matches(withText(containsString(myArray[i]))));
      }
   }
}
```

The test clicks each spinner item from top to bottom, checking to see if the item appears in the text field. It doesn't matter how many spinner items are defined in the array, or what language is used for the spinner's items—the test performs all of them and checks their output against the array.

The following is a step-by-step description of the above test:

1. The `iterateSpinnerItems()` method begins by getting the array used for the spinner items:

   ```
   public void iterateSpinnerItems() {
   String[] myArray =
        mActivityRule.getActivity().getResources()
        .getStringArray(R.array.labels_array);
   ...
   ```

   In the statement above, the test accesses the application's array (with the id `labels_array`) by establishing the context with the `getActivity()` method of the ActivityTestRule class, and getting a resources instance in the application's package using `getResources()`.

2. Using the length (`size`) of the array, the `for` loop iterates through each spinner item.

   ```
   ...
   int size = myArray.length;
   for (int i=0; i<size; i++) {
   // Find the spinner and click on it.
   ...
   ```

3. The `onView()` statement within the `for` loop finds the spinner and clicks on it. The test must click the spinner itself in order click any item in the spinner:

   ```
   ...
   // Find the spinner and click on it.
   onView(withId(R.id.label_spinner)).perform(click());
   ...
   ```

4. The `onData()` statement finds and clicks a spinner item:

```
...
// Find the spinner item and click on it.
onData(is(myArray[i])).perform(click());
...
```

   The spinner is populated from the `myArray` array, so `myArray[i]` represents a spinner element from the array. As the `for` loop iterates `for (int i=0; i<size; i++)`, it performs a click on each spinner element (`myArray[i]`) it finds.

5. The last `onView()` statement finds the text view (`text_phonelabel`) and checks that the spinner item is part of the string:

```
...
onView(withId(R.id.text_phonelabel))
     .check(matches(withText(containsString(myArray[i]))));
...
```

Using RecyclerViewActions for a Recycler View

A RecyclerView is useful when you have data collections with elements that change at runtime based on user action or network events. RecyclerView is a UI component designed to render a collection of data, but is not a subclass of AdapterView but of ViewGroup. This means that you can't use `onData()`, which is specific to AdapterView, to interact with list items.

However, there is a class called RecyclerViewActions that exposes a small API to operate on a RecyclerView. For example, the following test clicks on an item from the list by position:

```
onView(withId(R.id.recyclerView))
      .perform(RecyclerViewActions.actionOnItemAtPosition(0, click()));
```

The following test class demonstrates how to use RecyclerViewActions to test a RecyclerView. The app lets you scroll a list of words. When you click on a word such as **Word 15** the word in the list changes to "Clicked! Word 15":

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

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

   @Test
   public void recyclerViewTest() {
      ViewInteraction recyclerView = onView(
            allOf(withId(R.id.recyclerview), isDisplayed()));
      recyclerView.perform(actionOnItemAtPosition(15, click()));

      ViewInteraction textView = onView(
            allOf(withId(R.id.word), withText("Clicked! Word 15"),
                  childAtPosition(
                        childAtPosition(
                              withId(R.id.recyclerview),
                              11),
                        0),
                  isDisplayed()));
      textView.check(matches(withText("Clicked! Word 15")));

   }

   private static Matcher<View> childAtPosition(
            final Matcher<View> parentMatcher, final int position) {

      return new TypeSafeMatcher<View>() {
         @Override
         public void describeTo(Description description) {
            description.appendText("Child at position "
                                          + position + " in parent ");
            parentMatcher.describeTo(description);
         }

         @Override
         public boolean matchesSafely(View view) {
            ViewParent parent = view.getParent();
            return parent instanceof ViewGroup &&
                                    parentMatcher.matches(parent)
                                    && view.equals(((ViewGroup)
                                    parent).getChildAt(position));
         }
      };
   }
}
```

The test uses a recyclerView object of the ViewInteraction class, which is the primary interface for performing actions or assertions on views, providing both `check()` and `perform()` methods. Each interaction is associated with a view identified by a view matcher:

- The code below uses the `perform()` method and the actionOnItemAtPosition() method of the RecyclerViewActions class to scroll to the position (15) and click the item:

  ```
  ViewInteraction recyclerView = onView(
        allOf(withId(R.id.recyclerview), isDisplayed()));
  recyclerView.perform(actionOnItemAtPosition(15, click()));
  ```

- The code below checks to see if the clicked item matches the assertion that it should be `"Clicked! Word 15"`:

```
    ViewInteraction textView = onView(
          allOf(withId(R.id.word), withText("Clicked! Word 15"),
               childAtPosition(
                       childAtPosition(
                               withId(R.id.recyclerview),
                               11),
                       0),
               isDisplayed()));
    textView.check(matches(withText("Clicked! Word 15")));
```

- The code above uses a method called `childAtPosition()`, which is defined as a custom `Matcher`:

```
private static Matcher<View> childAtPosition(
      final Matcher<View> parentMatcher, final int position) {
   // TypeSafeMatcher() returned
   ...
   }
```

The custom matcher extends the abstract TypeSaveMatcher class and requires that you implement the following:

- The `matchesSafely()` method to define how to check for a view in a RecyclerView.
- The `describeTo()` method to define how Espresso describes the output's matcher in the Run pane at the bottom of Android Studio if a failure occurs.

```
    ...
    // TypeSafeMatcher() returned
    return new TypeSafeMatcher<View>() {
       @Override
       public void describeTo(Description description) {
          description.appendText("Child at position "
                                          + position + " in parent ");
          parentMatcher.describeTo(description);
       }

       @Override
       public boolean matchesSafely(View view) {
          ViewParent parent = view.getParent();
          return parent instanceof ViewGroup &&
                                    parentMatcher.matches(parent)
                                    && view.equals(((ViewGroup)
                                    parent).getChildAt(position));
       }
    };
  }
}
```

Recording a test

An Android Studio feature (in version 2.2 and newer) lets you *record* an Espresso test, creating the test automatically.

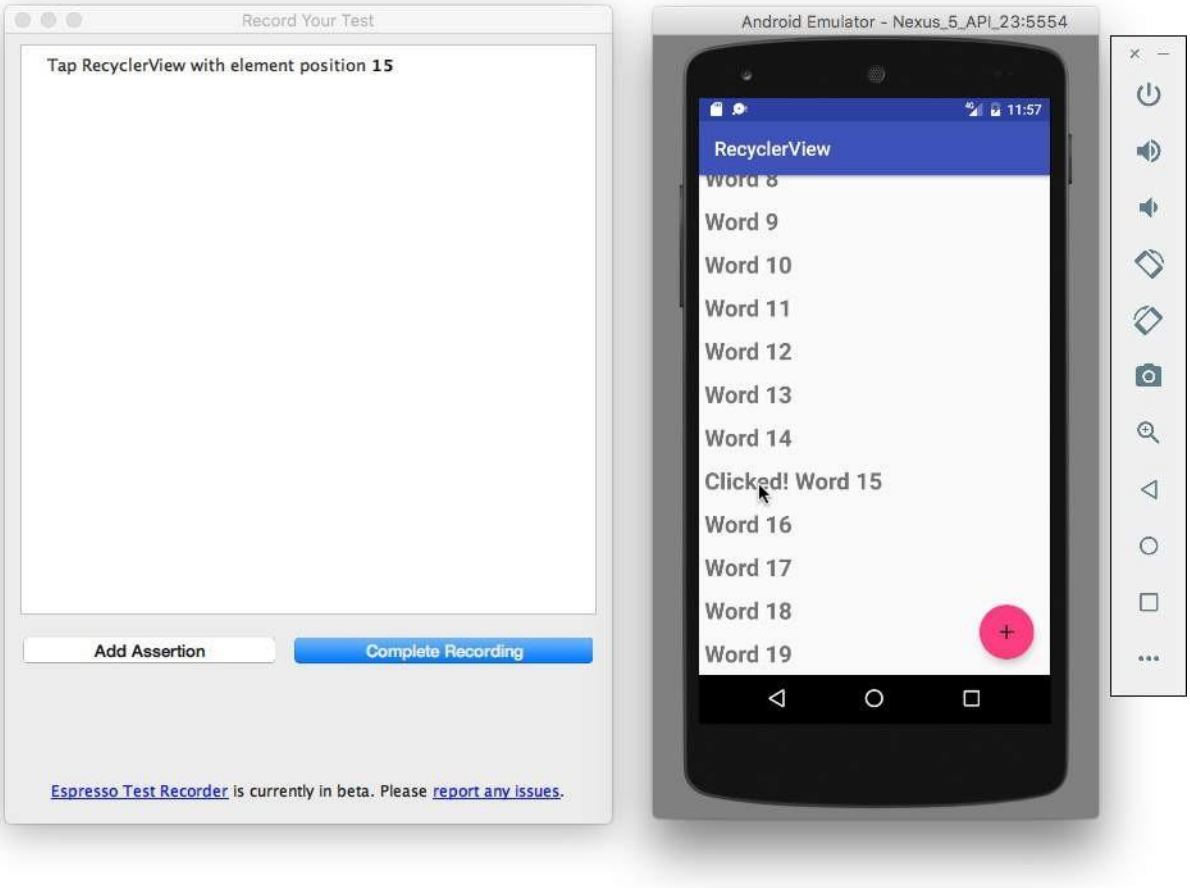
After choosing to record a test, use your app as a normal user would. As you click through the app UI, editable test code is generated for you. Add assertions to check if a view holds a certain value.

You can record multiple interactions with the UI in one recording session. You can also record multiple tests, and edit the tests to perform more actions, using the recorded code as a snippet to copy, paste, and edit.

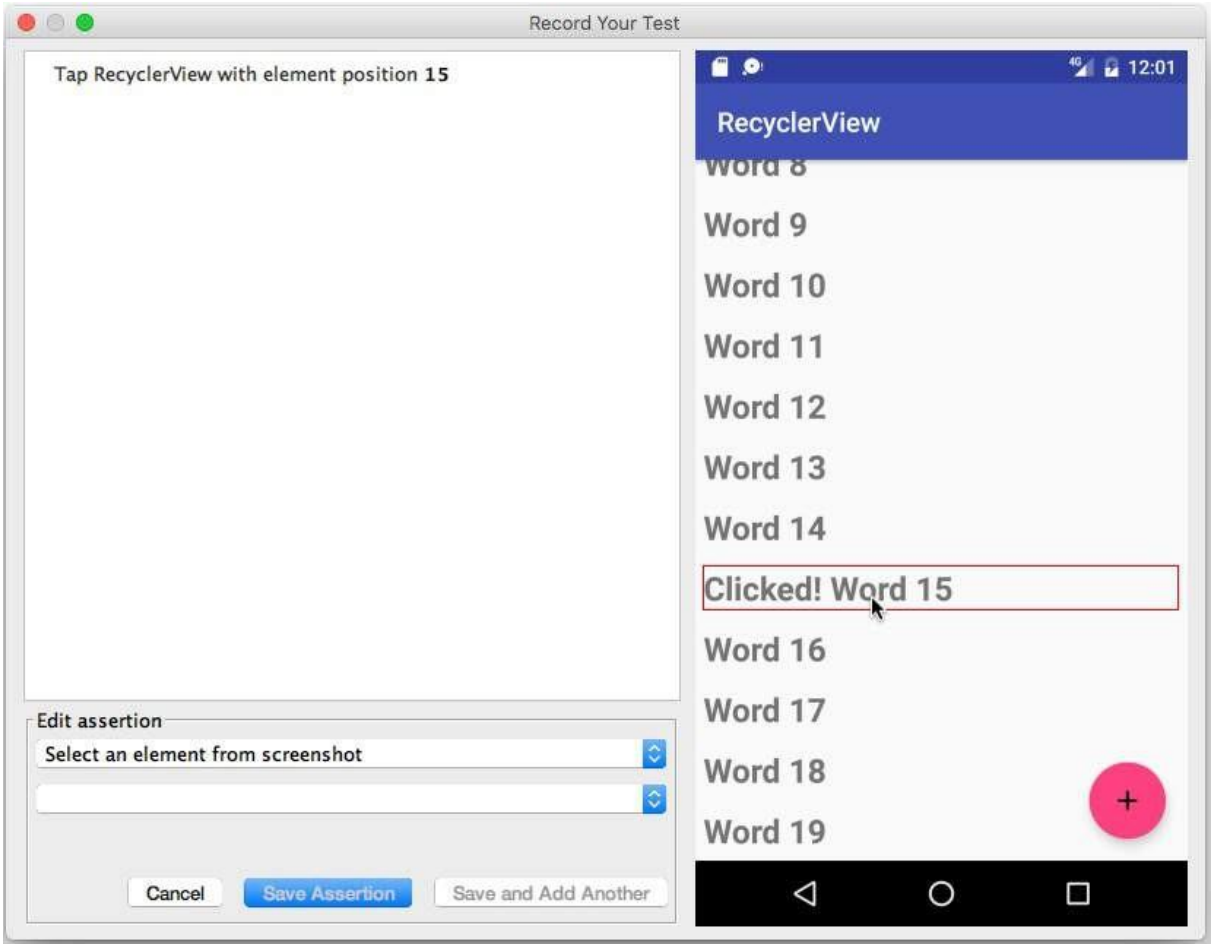
Follow these steps to record a test, using the RecyclerView app as an example:

Android Studio Project: RecyclerView

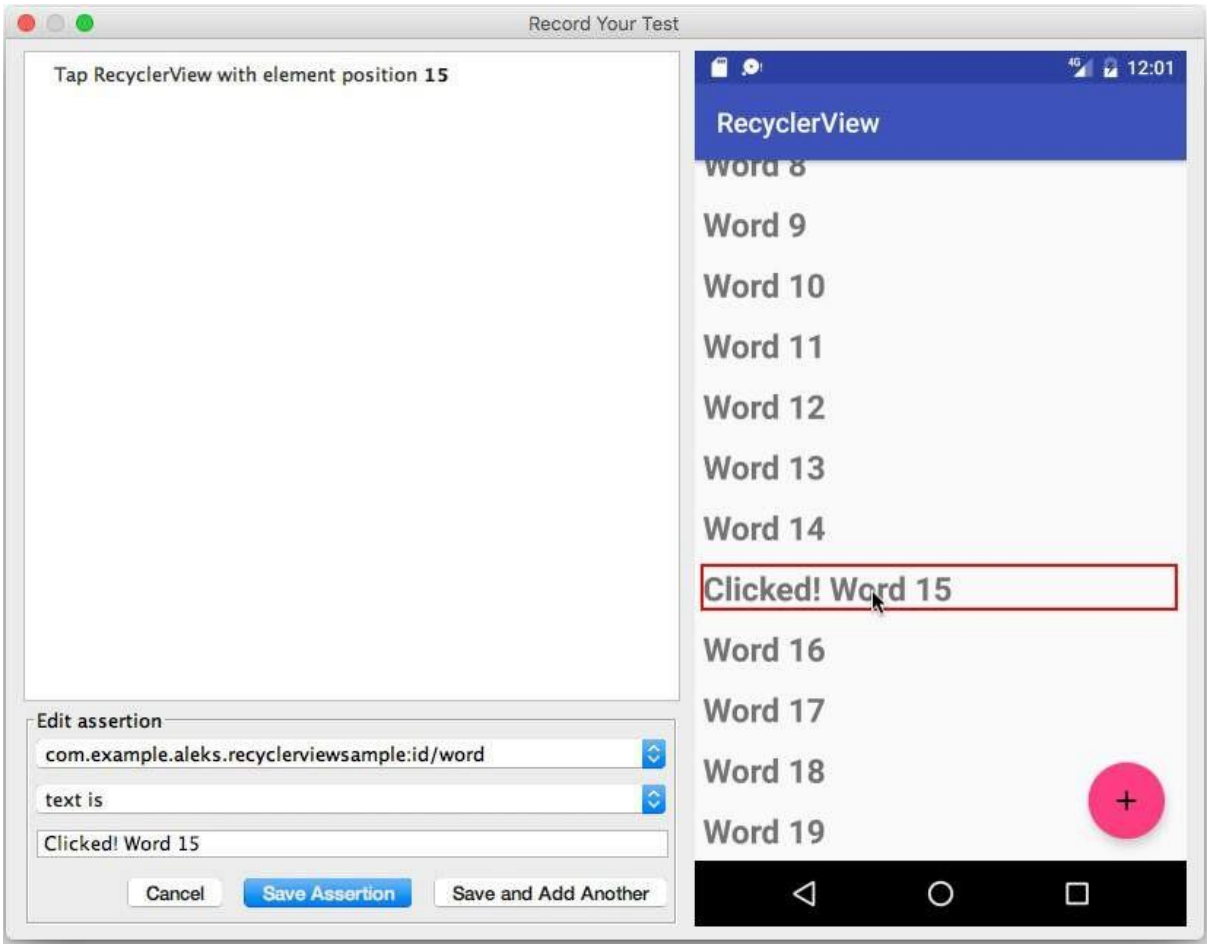
1. Choose **Run > Record Espresso Test**, select your deployment target (an emulator or a device), and click **OK**.
2. Interact with the UI to do what you want to test. In this case, scroll the word list in the app on the emulator or a device, and tap **Word 15**. The Record Your Test window shows the action that was recorded ("Tap RecyclerView with element position 15").



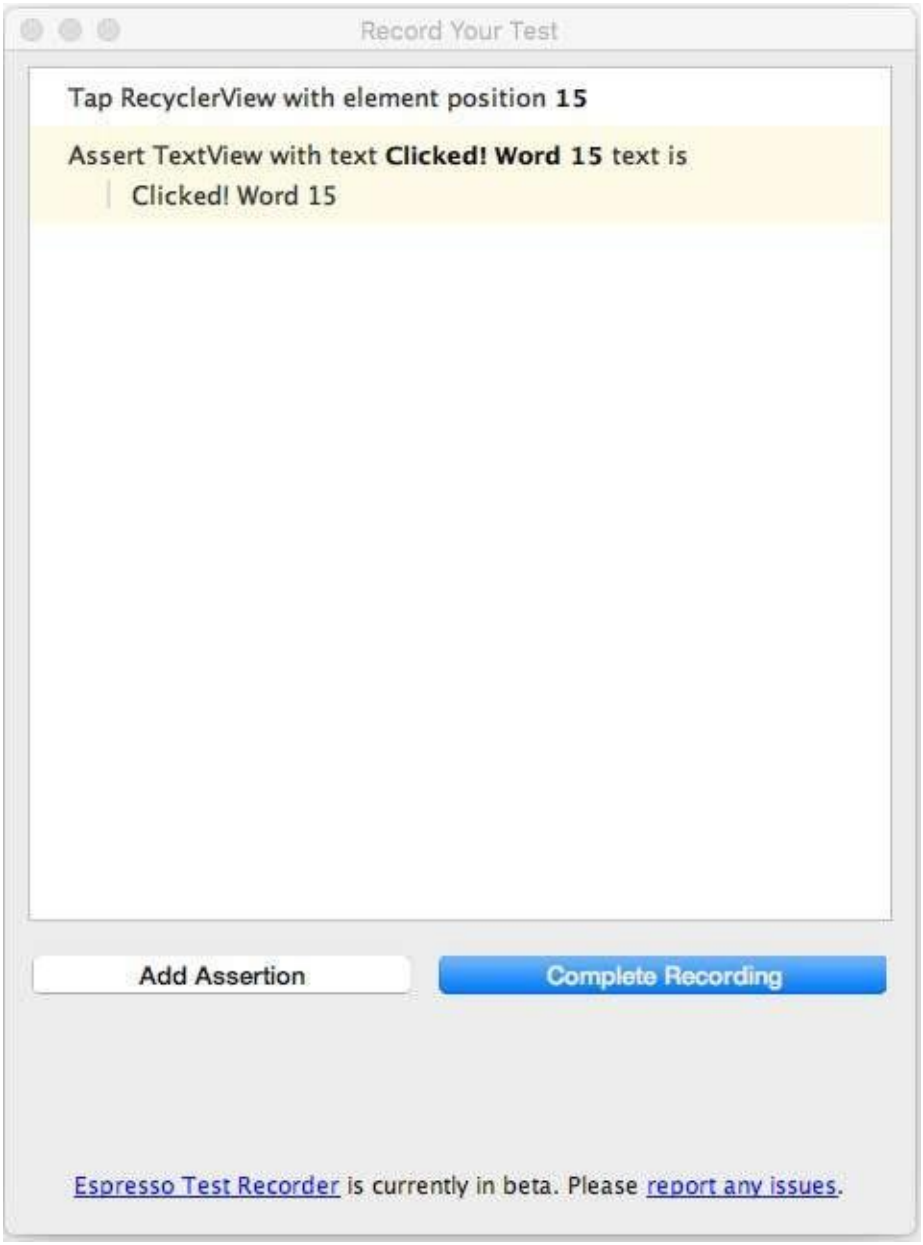
3. Click **Add Assertion** in the Record Your Test window. A screenshot of the app's UI appears in a pane on the right side of the window. Select **Clicked! Word 15** in the screenshot as the UI element you want to check. This creates an assertion for the selected element's view.



4. Choose **text is** from the second drop-down menu, and enter the text you expect to see in that UI element.



5. Click **Save Assertion**, and then click **Complete Recording**.



6. In the dialog that appears, you can edit the name of the test, or accept the name suggested (such as **MainActivityTest**).
7. Android Studio may display a request to add more dependencies to your Gradle Build file. Click **Yes** to add the dependencies.

Using UI Automator for tests that span multiple apps

UI Automator is a set of APIs that can help you verify the correct behavior of interactions between different user apps, or between user apps and system apps. It lets you interact with visible elements on a device. A viewer tool provides a visual interface to inspect the layout hierarchy and view the properties of UI components that are visible on the foreground of the device. Like Espresso, UI Automator has access to system interaction information so that you can monitor all of the interaction the Android system has with the app.

To use UI Automator, you must already have set up your test environment in the same way as for Espresso:

- Install the Android Support Repository and the Android Testing Support Library. Add
- the following dependency to the project's build.gradle file:

```
androidTestCompile
     'com.android.support.test.uiautomator:uiautomator-v18:2.1.2'
```

Use UI Automator Viewer to Inspect the UI on a device

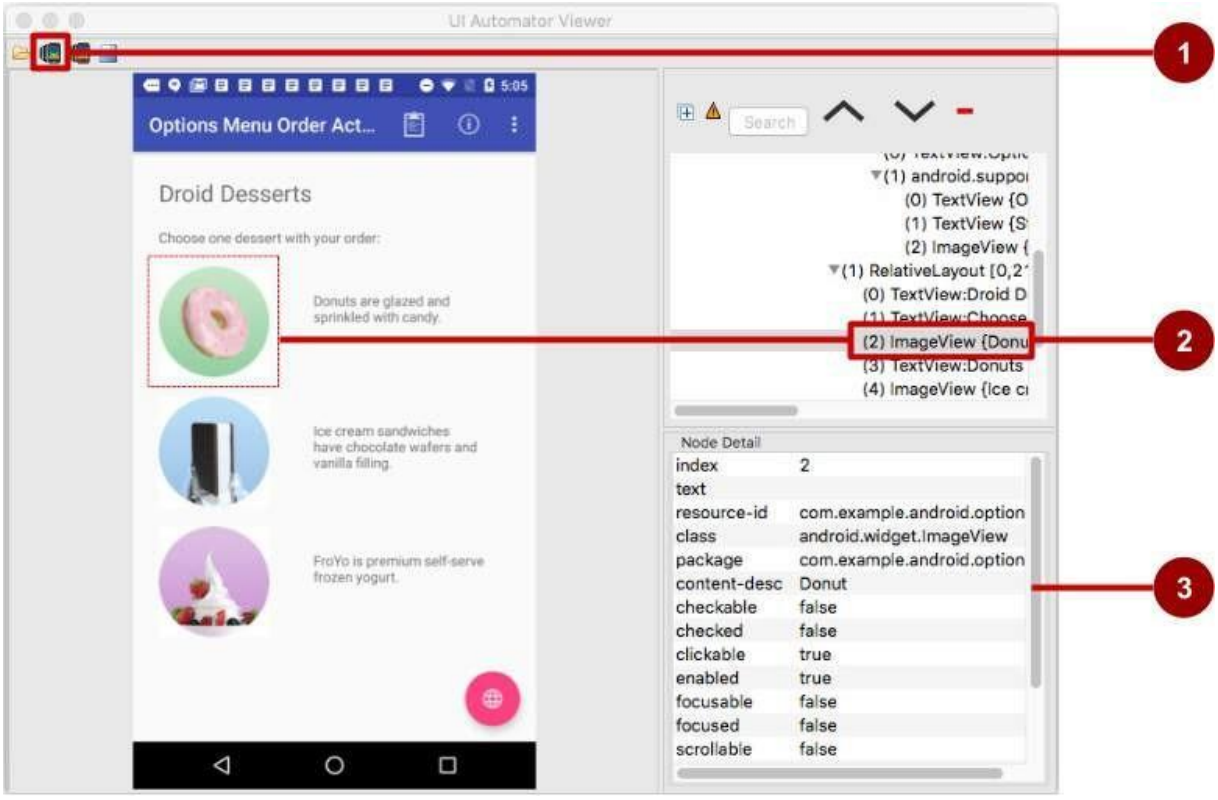
UI Automator Viewer (uiautomatorviewer) provides a convenient visual interface to inspect the layout hierarchy and view the properties of UI components that are visible on the foreground of the device.

To launch the uiautomatorviewer tool, follow these steps:

1. Install and then launch the app on a physical device such as a smartphone.
2. Connect the device to your development computer.
3. Open a terminal window and navigate to the **/tools/** directory. To find the specific path, choose **Preferences** in Android Studio, and click **Appearance & Behavior > System Settings > Android SDK**. The full path for appears in the Android SDK Location box at the top of the screen.
4. Run the tool with this command: **uiautomatorviewer**

To ensure that your UI Automator tests can access the app's UI elements, check that the elements have visible text labels, android:contentDescription values, or both. You can view the properties of a UI element by following these steps (refer to the figure below):

1. After launching uiautomatorviewer, the viewer is empty. Click the **Device Screenshot** button.
2. Hover over a UI element in the snapshot in the left-hand panel to see the element in the layout hierarchy in the upper right panel.
3. The layout attributes and other properties for the UI element appear in the lower right panel. Use this information to create tests that select a UI element that matches a specific visible attribute or property.

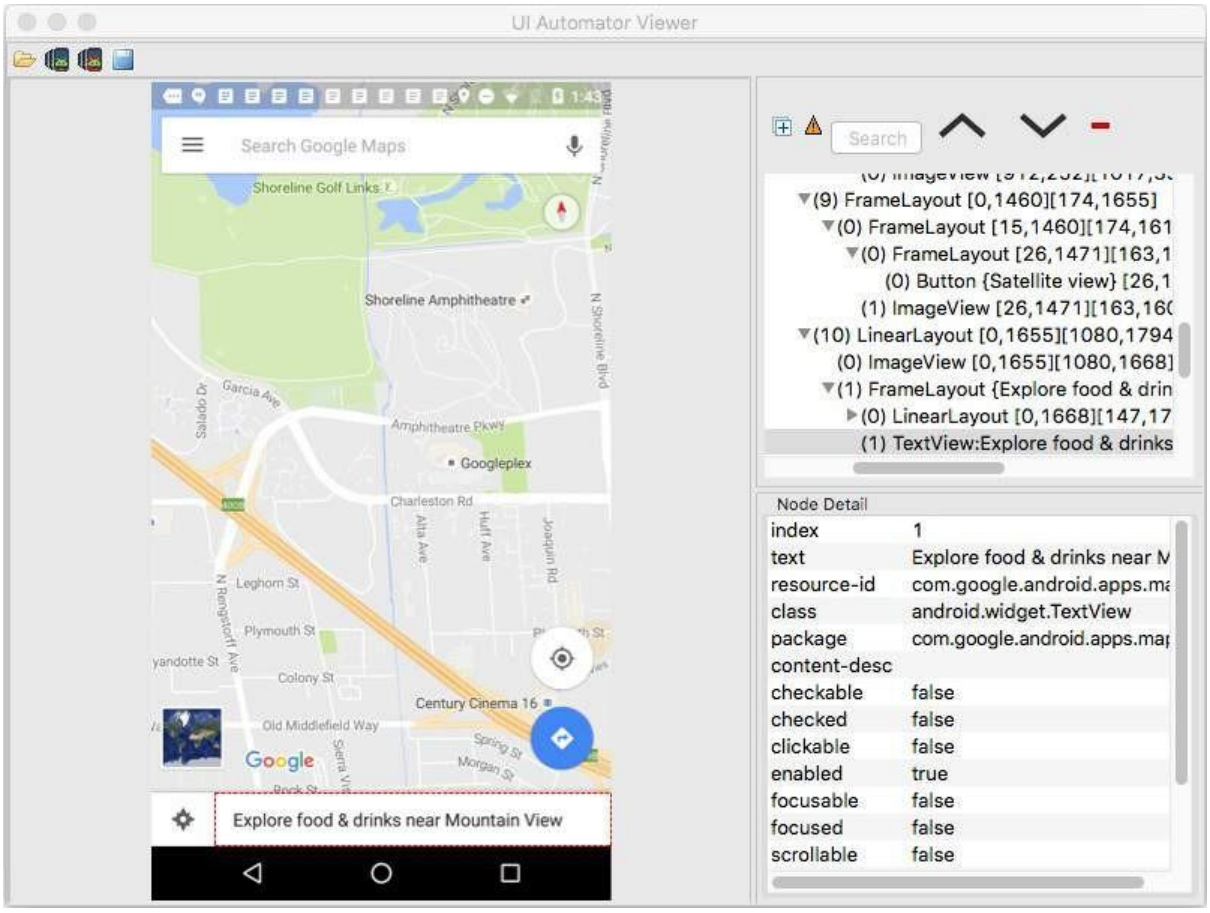


In the above figure:

1. **Device Screenshot** button
2. Selected component in the snapshot and in the layout hierarchy
3. Properties for the selected component

In the app shown above, the red floating action button launches the Maps app. Follow these steps to test the action performed by the floating action button:

1. Tap the floating action button.
2. The code for the button makes an implicit intent to launch the Maps app.
3. Click the **Device Screenshot** button to see the result of the implicit intent (the Maps screen).



Using the viewer, you can determine which UI elements are accessible to the UI Automator framework.

Setup: Ensuring that UI elements are accessible

The UI Automator framework depends on the accessibility features of the Android framework to look up individual UI elements. Implement view properties as follows:

- Include the android:contentDescription attribute in your XML layout to label the ImageButton, ImageView, CheckBox, and other user input controls. The following shows the `android:contentDescription` attribute added to a `RadioButton` using the same string resource tused for the `android:text` attribute:

```
<RadioButton
  android:id="@+id/sameday"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="@string/same_day_messenger_service"
  android:contentDescription="@string/same_day_messenger_service"
  android:onClick="onRadioButtonClicked"/>
```

  Tip: You can make input controls more accessible for the sight-impaired by using the `android:contentDescription` XML layout attribute. The text in this attribute does not appear on screen, but if the user enables accessibility services that provide audible prompts, then when the user navigates to that control, the text is spoken.

- Provide an android:hint attribute for EditText fields (in addition to `android:contentDescription`, which is useful for

accessibility services). With EditText fields, UI Automator looks for the `android:hint` attribute.

- Associate an `android:hint` attribute with any graphical icons used by controls that provide feedback to the user (for example, status or state information).

As a developer, you should implement the above minimum optimizations to support your users as well as UI Automator.

Creating a test class

A UI Automator test class generally follows this programming model:

1. **Access the device to test:** An instance of the InstrumentRegistry class holds a reference to the instrumentation running in the process along with the instrumentation arguments. It also provides an easy way for callers to get instrumentation, application context, and an instrumentation arguments Bundle. You can get a UiDevice object by calling the getInstance() method and passing it an Instrumentation object — `InstrumentationRegistry.getInstrumentation()` —as the argument. For example:

   ```
   mDevice = UiDevice.getInstance(InstrumentationRegistry.getInstrumentation());
   ```

2. **Access a UI element displayed on the device:** Get a UiObject by calling the findObject() method. For example:

   ```
   UiObject okButton = mDevice.findObject(new UiSelector()
       .text("OK"))
       .className("android.widget.Button"));
   ```

3. **Perform an action:** Simulate a specific user interaction to perform on that UI element by calling a UiObject method. For example:

   ```
   if(okButton.exists() && okButton.isEnabled()) {
       okButton.click();
   }
   ```

   You may want to call setText() to edit a text field, or performMultiPointerGesture() to simulate a multi-touch gesture.

   You can repeat steps 2 and 3 as needed to test more complex user interactions that involve multiple UI components or sequences of user actions.

4. **Verify results:** Check that the UI reflects the expected state or behavior after these user interactions are performed. You can use standard JUnit Assert methods to test that UI components in the app return the expected results. For example:

   ```
   UiObject result = mDevice.findObject(By.res(CALC_PACKAGE, "result"));
   assertEquals("5", result.getText());
   ```

Accessing the device

The UiDevice class provides the methods for accessing and manipulating the state of the device. Unlike Espresso, UI Automator can verify the correct behavior of interactions between different user apps, or between user apps and system apps. It lets you interact with visible elements on a device. In your tests, you can call UiDevice methods to check for the state of various properties, such as current orientation or display size. Your test can use the UiDevice object to perform device-level actions, such as forcing the device into a specific rotation, pressing D-pad hardware buttons, and pressing the Home button.

It's a good practice to start your test from the Home screen of the device. From the Home screen you can call the methods provided by the UI Automator API to select and interact with specific UI elements. The following code snippet shows how your test can get an instance of UiDevice, simulate a Home button press, and launch the app:

```
import org.junit.Before;
import android.support.test.runner.AndroidJUnit4;
import android.support.test.uiautomator.UiDevice;
import android.support.test.uiautomator.By;
import android.support.test.uiautomator.Until;
...

@RunWith(AndroidJUnit4.class)
@SdkSuppress(minSdkVersion = 18)
public class ChangeTextBehaviorTest {

    private static final String BASIC_SAMPLE_PACKAGE
            = "com.example.android.testing.uiautomator.BasicSample";
    private static final int LAUNCH_TIMEOUT = 5000;
    private static final String STRING_TO_BE_TYPED = "UiAutomator";
    private UiDevice mDevice;

    @Before
    public void startMainActivityFromHomeScreen() {
        // Initialize UiDevice instance
        mDevice =
          UiDevice.getInstance(InstrumentationRegistry.getInstrumentation());

        // Start from the home screen
        mDevice.pressHome();

        // Wait for launcher
        final String launcherPackage = mDevice.getLauncherPackageName();
        assertThat(launcherPackage, notNullValue());
        mDevice.wait(Until.hasObject(By.pkg(launcherPackage).depth(0)),
                LAUNCH_TIMEOUT);

        // Launch the app
        Context context = InstrumentationRegistry.getContext();
        final Intent intent = context.getPackageManager()
                .getLaunchIntentForPackage(BASIC_SAMPLE_PACKAGE);
        // Clear out any previous instances
        intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
        context.startActivity(intent);

        // Wait for the app to appear
        mDevice.wait(Until.hasObject(By.pkg(BASIC_SAMPLE_PACKAGE).depth(0)),
                LAUNCH_TIMEOUT);
    }
}
```

The `@SdkSuppress(minSdkVersion = 18)` annotation ensures that tests will only run on devices with Android 4.3 (API level 18) or higher, as required by the UI Automator framework.

Accessing a UI element

Use the findObject() method of the UiObject class to retrieve a UiObject instance that represents a UI element matching a given selector criteria. To access a specific UI element, use the UiSelector class, which represents a query for specific elements in the currently displayed UI.

You can reuse the UiObject instances that you created in other parts of your app testing. The UI Automator test framework searches the current display for a match every time your test uses a UiObject instance to click on a UI element or query an attribute.

The following shows how your test might construct UiObject instances using `findObject()` with a UiSelector that represent a **Cancel** button, and one that represents an **OK** button:

```
UiObject cancelButton = mDevice.findObject(new UiSelector()
        .text("Cancel"))
        .className("android.widget.Button"));
UiObject okButton = mDevice.findObject(new UiSelector()
        .text("OK"))
        .className("android.widget.Button"));
```

If more that one element matches, the first matching element in the layout hierarchy (found by moving from top to bottom, left to right) is returned as the target UiObject. When constructing a UiSelector, you can chain together multiple attributes and properties to refine your search. If no matching UI element is found, an exception (UiAutomatorObjectNotFoundException) is thrown.

To nest multiple UiSelector instances, use the childSelector() method of the UiSelector class. For example, the following shows how your test might specify a search to find the first ListView in the currently displayed UI, then search within that ListView to find a UI component with the `android:text` attribute `"List Item 14"`:

```
UiObject appItem = new UiObject(new UiSelector()
        .className("android.widget.ListView")
        .instance(1)
        .childSelector(new UiSelector()
        .text("List Item 14")));
```

While it may be useful to refer to the `android:text` attribute of an element of a ListView or RecycleView because there is no resource id (`android:id` attribute) for such an element, it is best to use a resource id when specifying a selector rather than the `android:text` or `android:contentDescription` attributes. Not all elements have a text attribute (for example, icons in a toolbar). Tests might fail if there are minor changes to the text of a UI component, and the tests would not be usable for apps translated into other languages because your text selectors would not match the translated string resources.

Performing actions

Once your test has retrieved a UiObject object, you can call the methods in the UiObject class to perform user interactions on the UI component represented by that object. For example, the constructed UiObject instances in the previous section for the OK and Cancel buttons can be used to perform a click:

```
// Simulate a user-click on the OK button, if found.
if(okButton.exists() && okButton.isEnabled()) {
    okButton.click();
}
```

You can use UiObject methods to perform actions such as:

- click(): Click the center of the visible bounds of the UI element.
- dragTo(): Drag the object to arbitrary coordinates.
- setText(): Set the text in an editable field, after clearing the field's content. Conversely, you use the clearTextField() method to clear the existing text in an editable field.
- swipeUp(): Perform the swipe up action on the UiObject. Similarly, the swipeDown(), swipeLeft(), and swipeRight() methods perform corresponding actions.

Sending an intent or launching an activity

The UI Automator testing framework enables you to send an Intent or launch an Activity without using shell commands, by getting a Context object through the getContext() method. For example, the following shows how your test can use an Intent to launch the app under test:

```
public void setUp() {
    ...
    // Launch a simple calculator app
    Context context = getInstrumentation().getContext();
    Intent intent = context.getPackageManager()
            .getLaunchIntentForPackage(CALC_PACKAGE);
    intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
            // Clear out any previous instances
    context.startActivity(intent);
    mDevice.wait(Until.hasObject(By.pkg(CALC_PACKAGE).depth(0)), TIMEOUT);
}
```

Performing actions on collections

Use the UiCollection class if you want to simulate user interactions on a collection of UI elements (for example, song titles or emails in a list). To create a UiCollection object, specify a UiSelector that searches for a UI container or a wrapper of other child UI elements, such as a layout group that contains child UI elements.

The following shows how a test can use a UiCollection to represent a video album that is displayed within a FrameLayout:

```
UiCollection videos = new UiCollection(new UiSelector()
        .className("android.widget.FrameLayout"));

// Retrieve the number of videos in this collection:
int count = videos.getChildCount(new UiSelector()
        .className("android.widget.LinearLayout"));

// Find a specific video and simulate a user-click on it
UiObject video = videos.getChildByText(new UiSelector()
        .className("android.widget.LinearLayout"), "Cute Baby Laughing");
video.click();

// Simulate selecting a checkbox that is associated with the video
UiObject checkBox = video.getChild(new UiSelector()
        .className("android.widget.Checkbox"));
if(!checkBox.isSelected()) checkbox.click();
```

Performing actions on scrollable views

Use the UiScrollable class to simulate vertical or horizontal scrolling across a display. This technique is helpful when a UI element is positioned off-screen and you need to scroll to bring it into view. For example, the following code snippet shows how to simulate scrolling down the **Settings** menu and clicking on the **About phone** option:

```
UiScrollable settingsItem = new UiScrollable(new UiSelector()
        .className("android.widget.ListView"));
UiObject about = settingsItem.getChildByText(new UiSelector()
        .className("android.widget.LinearLayout"), "About phone");
about.click();
```

Verifying results

You can use standard JUnit Assert methods to test that UI components in the app return the expected results. For example, you can use assertFalse() to assert that a condition is false in order to test if the condition truly is false as a result. Use assertEquals() to test if a floating point number result is equal to the assertion:

```
assertEquals("5", result.getText());
```

The following shows how your test can locate several buttons in a calculator app, click on them in order, then verify that the correct result is displayed:

```
private static final String CALC_PACKAGE = "com.myexample.calc";
public void testTwoPlusThreeEqualsFive() {
    // Enter an equation: 2 + 3 = ?
    mDevice.findObject(new UiSelector()
            .packageName(CALC_PACKAGE).resourceId("two")).click();
    mDevice.findObject(new UiSelector()
            .packageName(CALC_PACKAGE).resourceId("plus")).click();
    mDevice.findObject(new UiSelector()
            .packageName(CALC_PACKAGE).resourceId("three")).click();
    mDevice.findObject(new UiSelector()
            .packageName(CALC_PACKAGE).resourceId("equals")).click();

    // Verify the result = 5
    UiObject result = mDevice.findObject(By.res(CALC_PACKAGE, "result"));
    assertEquals("5", result.getText());
}
```

Running instrumented tests

To run a single test, right-click (or Control-click) the test in Android Studio, and choose **Run** from the pop-up menu.

To test a method in a test class, right-click the method in the test file and click **Run**.

To run all tests in a directory, right-click on the directory and select **Run tests**.

Android Studio displays the results of the test in the Run window.