WHITE PAPER

BEYOND APPIUM: IMPLEMENTING COMPREHENSIVE TESTING USING ESPRESSO AND XCUITEST

Today the degree of mobile app testing required to meet the market's relentless demand for new applications and new application features is beyond the capacity of human testers. Using an automated mobile testing tool is essential for the modern enterprise. In this white paper you will learn about the importance of having a well-defined mobile testing process and industry standard test automation framework, as well as the capabilities and benefits of implementing automated testing using Espresso and XCUITest.

SAUCELABS

TABLE OF CONTENTS

EXECUTIVE SUMMARY

Today more than half of the world's web traffic comes from mobile devices. Of that number, cell phones have the largest share of users. Yet, having millions of cellphones on line does not translate into millions of different applications being used. Surprisingly, the average consumer gives the most attention to five of the apps he or she has installed. Once an app is downloaded, it doesn't have a lot of time to make a good impression. If things go wrong, the app may not be given a second chance. One report cites that only 79% of users will try an app a second time. The number of users who will give an app a third try after a bad experience drops to 16%! Comprehensive testing is not a nice to have, it's critical. Mobile applications need to be world class because the world will be using them.

Still many companies do not have well defined testing processes, nor do they use an industry standard test automation framework. In a recent survey conducted by Sauce Labs, nearly one third of those surveyed report using no automation at all. (See Figure 1.)

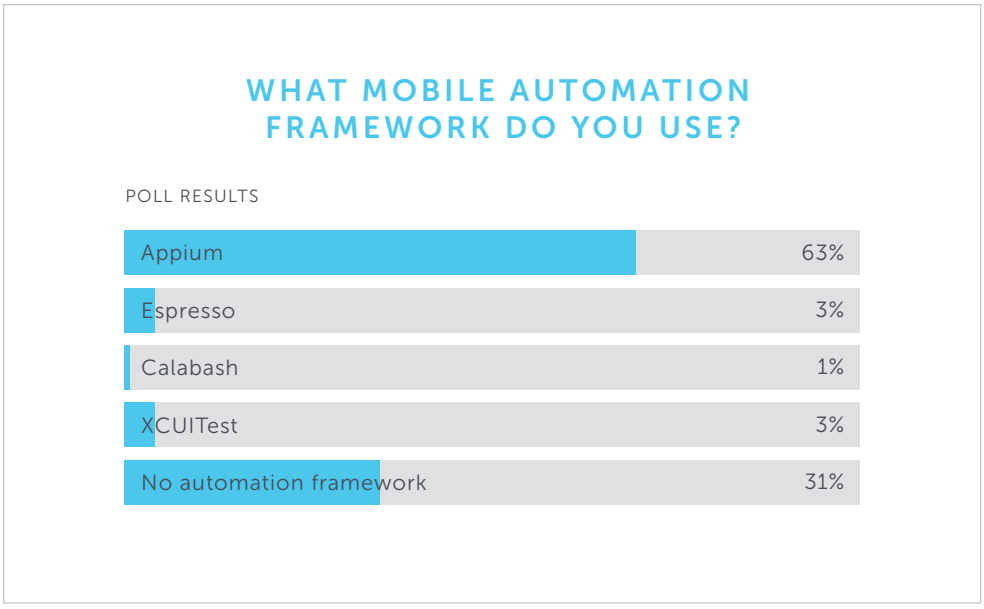

Figure 1: Many companies use no automation framework when testing

The risks incurred by not using an automation framework are significant. Automated testing provides the consistency and predictability that manual testing cannot. Today, the degree of mobile app testing required to meet the market's relentless demand for new applications and new application features is beyond the capacity of human testers. Using an automated mobile testing tool is essential for the modern enterprise. There is little choice otherwise. No one person can tap and swipe a cell phone fast enough to satisfy the volume

of testing required in typical testing scenarios. In order to survive, enterprises must automate mobile testing.

Mobile automation frameworks have matured to keep pace with the proliferation of cell phones and tablets accessing the Internet. In fact, automated testing is embedded in the two most common integrated development environments (IDEs) used for making mobile applications. Android Studio, published by Google, provides automated testing capability by way of Espresso. XCode, the standard development environment for creating Apple iOS applications, uses XCUITest.

The scope of this paper is to describe the capabilities and benefits of implementing automated testing using Espresso and XCUITest. Espresso is an automated testing framework for testing native Android apps. XCUITest is a framework dedicated to native iOS development. For purposes of illustration, we created a demonstration application named Wise Sayings. Wise Saying has two implementations, one in Android and the other in iOS. (Please see Figure 2, below).

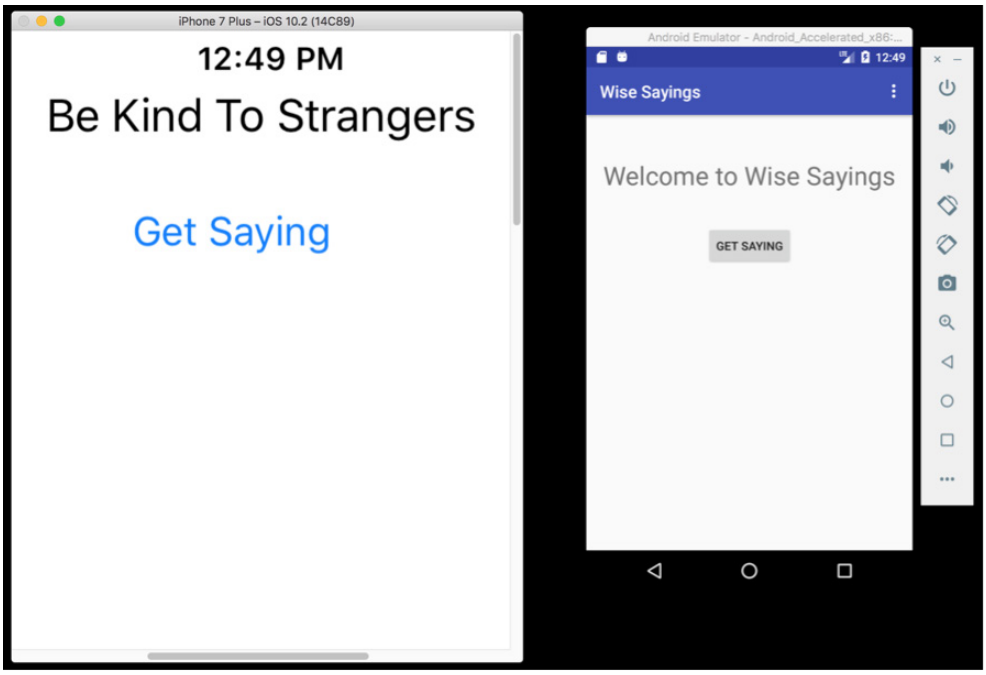

Figure 2: The demonstration application Wise Sayings is implemented in both iOS and Android

Wise Sayings implements a single feature-- to show a wise saying when the user clicks the Get Saying button. There are four wise sayings that a user cycles through in continuous rotation. Each time the user clicks the button in the application's GUI, the next wise saying appears. You can download the source code for both the Android and iOS versions on GitHub.

Admittedly, Wise Sayings is a trivial application. But its limited scope provides enough application behavior to demonstrate how to implement automated mobile testing under both Espresso and XCUITest.

CREATING TESTABLE CODE

Before delving into the details of automated mobile testing using Espresso or XCUITest, it's important to understand an essential concept in software testing – creating testable code.

One of the common mistakes made by developers new to automated testing is failing to create code that is testable. Ensuring that an application is testable is a practice that is best done when the code is being designed, not later in the software development lifecycle.

A simple analogy is being able to test a tire's air pressure on an automobile. All those involved in making a car understand that checking tire pressure is an essential feature of the vehicle. Thus, those designing the car will do best to make sure that the air valve is on the outside of the tire, easily accessible to the driver for testing. Putting the air valve on the inside of the tire forces the driver to crawl underneath the chassis to find the air valve in order to apply the pressure gauge. The bad design hinders the ability of the driver to check a tire's pressure easily and quickly.

Of course, no automobile designer in his or her right mind would put the air valve on the inside of a tire. However, there are more than a few instances out there where developers made code that's hard to test or completely untestable. To use our analogy, they put the air valve on the inside of the tire. Usually the developer writing such code was not responsible for testing the code written, or had no knowledge of the testing that needed to be supported. Thus, the code ended up downstream in the software development lifecycle (SDLC), in the hands of a QA engineer given the task to figure the testing out. The rule of thumb in software development is that the cost of work increases as the code goes further down the SDLC. Determining test requirements and testing implementation at the design time saves money and increases the velocity of release overall.

One of the best ways to ensure that a given codebase is testable is to make sure that the code is well encapsulated and that it follows the design

principle of separation of concerns. At the least the UI code should be distinctly separate from the code for data access. The architecture for the demonstration app, Wise Sayings, illustrates how to make such a separation. (Please see Figure 3.)

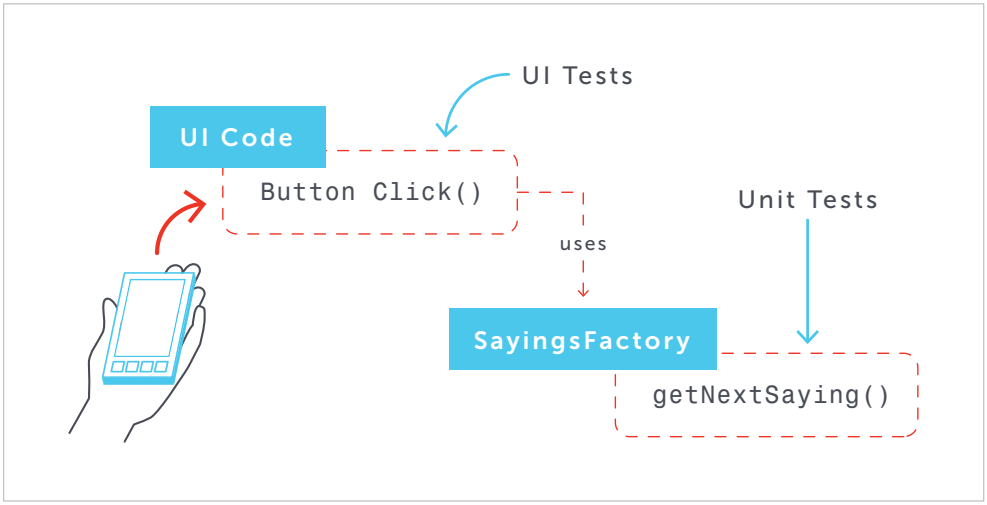

Figure 3: Well encapsulated code is easier to test than code that is monolithic

Notice that at the architectural level, Wise Sayings is made up of two logical components. One component is the UI code that contains the logic for rendering the UI and handling UI actions such as button clicks. The other component is the data access code encapsulated in `SayingsFactory`. `SayingsFactory` publishes one method, `getNextSaying()`, which returns a string that contains a saying. (Remember, a feature of Wise Sayings is to show a saying from a collection of four sayings in continuous cycle.)

Not only is segmenting UI code from data access code a best practice in terms of software design, supporting such separation of concerns makes the application easier to test.

When it comes to testing the Wise Sayings application, there are two types of testing that need to be done: UI Testing and Unit Testing. Each test type has a different scope of testing and will use a different testing tool.

Unit testing is concerned with testing logical functions. UI testing is concerned with testing the graphical behavior of an application. For example in the Wise Sayings illustration, a unit test will call the function, `SayingFactory.getNextSaying()` to verify that a different "saying" string is returned upon each invocation of the function. A UI test will use a device simulator to emulate the UI of a cell phone or tablet and then test the behavior of the application's graphical components. Both Android Studio/Espresso and XCode/XCUITest make a distinction between unit tests and UI tests. The distinction will become apparent when we examine each development environment in the sections to come.

WORKING WITH ESPRESSO

As mentioned above, Espresso is the testing framework that comes built into Android Studio. Typically creating a set of tests that run under Espresso is divided into two parts, as mentioned earlier. The first part is to write the unit tests that exercise the business logic and data access code of your application. The second part is to create the tests that interact with the various features exposed by way of the application's graphical user interface. We'll create the unit test manually. Testing the GUI will be done using the Espresso test recorder.

SEPARATING LOGIC FROM VIEW

The Android implementation of the Wise Sayings demonstration application follows the advice provided earlier with regard to creating testable code. As such, data access intelligence is encapsulated in a Java class, `SayingsFactory`. Listing 1, below, shows the code for `SayingsFactory`.

```
package com.example.reselbob.myappimapp;

import java.util.ArrayList;
import java.util.List;

public class SayingsFactory {
   private List<String> list = new ArrayList<String>();
   private static SayingFactory ref;
   private int currentIndex = 0;
   private SayingFactory()
   {
       list.add("Be Kind To Strangers");
       list.add("Always Be Honest");
       list.add("The Truth is the Best");
       list.add("Tip Well Always");
   }

   public static SayingsFactory getInstance()
   {
       if (ref == null) ref = new SayingsFactory();
       return ref;
   }

   public String getNextSaying(){
       if(currentIndex == list.size()) currentIndex = 0;
       String rtn = list.get(currentIndex);
       currentIndex++;
       return rtn;
   }
}
```

Listing 1: SayingsFactory encapsulates data retrieval behavior

Notice please that `SayingsFactory` is a singleton. Thus, the class's sole method, `getNextSaying()` will be called directly, like so:

```
SayingsFactory.getInstance().getNextSaying()
```

You can use `SayingsFactory` without having to create an instance of the class.

Providing all data access through a single factory class is useful in this case. `SayingsFactory` will be running on a single mobile device. There is no data sharing happening between mobile devices. Each device will carry its own list of data. Thus, using the singleton pattern is a safe approach in this scenario. However, should application requirements be such that list information is to be shared among multiple devices, then the complexity of coding increases. At the least, lists will need to be updated as more sayings are added.

ENCAPSULATION MAKES TESTING EASIER

Encapsulating data access behavior into a distinct class makes the testing process easier. As mentioned above, most modern enterprises require that the developer writing the code is responsible for testing the code. Thus, when data access is well encapsulated, the developer who is working on `SayingsFactory` can create tests independently from the developers working on other features of the application. In fact, developers can work in parallel, thus accelerating the velocity of the software development lifecycle. Developers who have code that depends on `SayingsFactory` will use a testing technique called mocking to emulate the behavior of `SayingsFactory` until such time that a release version of the data access code is ready.

ACCESSING DATA ACCESS LOGIC IN THE UI

Separating data access activity from the graphical user interface allows front end developers to focus their efforts on making GUIs that are visually effective and perform well. Whereas back end developers might be concerned with issues such as data synchronicity and server side processing, front end developers focus on visual accuracy and the overall responsiveness of the mobile application that is on the device in the hands of the end user. The testing needs are different too. The point of access for a backend test is typically a function or an URL in an API. Front tests are initiated by gestural input such as a button click, keyboard entry of swipe of the UI.

Listing 2, below shows the button click behavior that gets a saying from the `SayingsFactory` and assigns it to a text element for display in the UI.

```
public void onSayingButtonTap(View v) {
   TextView tv = (TextView) findViewById(R.id.textView);
   tv.setText(SayingFactory.getInstance().getNextSaying());
}
```

Listing 2: Android UI code uses the SayingFactory to get data to display

In terms of UI testing, coding the various gestures required just to make data available to a given test assertion can be time-consuming when done manually. Fortunately, Espresso has a recording tool that allows a tester to capture all UI gestures on the simulator.

USING THE ESPRESSO RECORDER

The Espresso Recorder is a tool built into the Android Studio IDE that allows a developer to record interaction gestures such as tapping and swiping. Developers make the gestures on one of the device simulators that ship with Android Studio. The Espresso Recorder records a gesture and translates the interaction into lines of code that it stores in a test file. Listing 3 below show a snippet of UI test code created using Espresso Recorder.

```
@Test
public void simpleButtonClickTest() {
   ViewInteraction appCompatButton = onView(
           allOf(withId(R.id.button), withText("Get Saying"), isDisplayed()));
   appCompatButton.perform(click());

   ViewInteraction viewGroup = onView(
           allOf(childAtPosition(childAtPosition(withId(android.R.id.content),
                           0),1), isDisplayed()));
   viewGroup.check(matches(isDisplayed()));

}
```

Listing 3: The Espresso Recorder records interactions with the UI into code in a test file.

Listing 3 above is a trivial test that checks to make sure that a text element is displayed when a button is clicked.

As mentioned earlier, the Espresso Recorder is built into the Android Studio IDE. You access the Recorder from the Run menu item in the Android Studio menu bar as shown below in Figure 4.

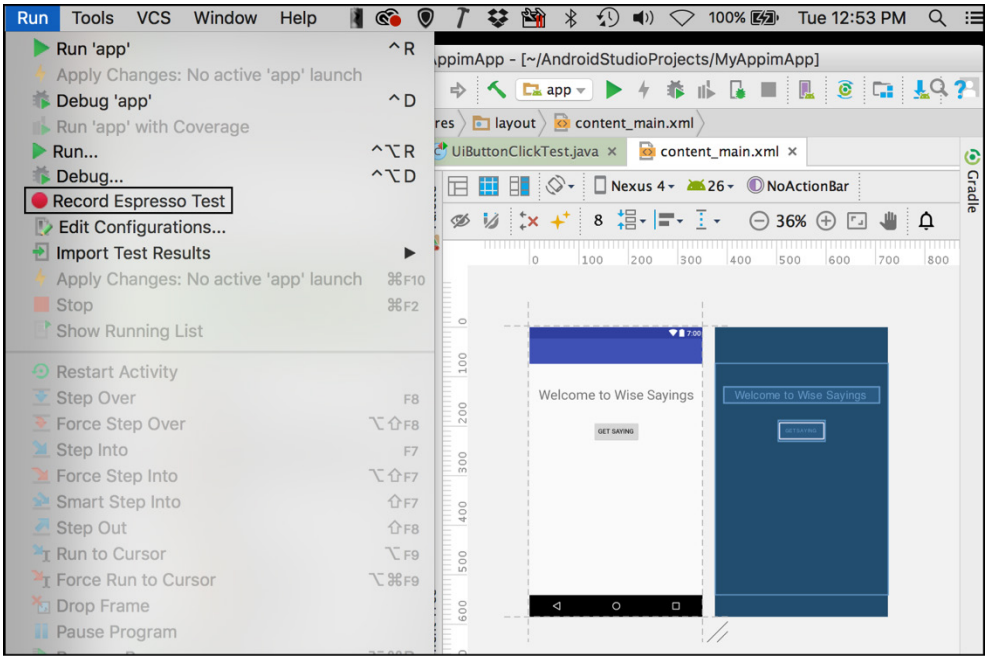

Figure 4: You can record tests to run under Espresso from within Android Studio

The Recorder will fire up a dialog that allows you to choose a simulator to use. Then the developer goes about making the gestures and entering data that is relevant to the scope of testing underway. Also, one of the nice features of the Recorder is that you can add an assertion after each data entry event. For example, in the case of the Wise Sayings application, the developer can use the Recorder to add an assertion that checks the value of the text element after a button is clicked. Being able to add assertions as part of the recording sessions saves significant time test creation process. Listing 4 below shows a test, `simpleResponseTest` that is the result of a test recording session in which an assertion is declared while recording.

```
@Test
public void simpleResponseTest() {
   ViewInteraction appCompatButton = onView(
           allOf(withId(R.id.button), withText("Get Saying"), isDisplayed()));
   appCompatButton.perform(click());

   ViewInteraction textView = onView(
           allOf(withId(R.id.textView), withText("Be Kind To Strangers"),
                   childAtPosition(
                           childAtPosition(
                                   IsInstanceOf.<View>instanceOf(android.view.
ViewGroup.class),
                                   1),
                           0),
                   isDisplayed()));
   textView.check(matches(withText("Be Kind To Strangers")));

}
```

Listing 4: The Espresso Recorder allows the developer to add assertions during a recording session

Once you finish a recording session, the Recorder will ask you to declare the name of the file in which to save the recorded code. The Recorder has the intelligence to store the test file in a category name androidTest, as shown below in Figure 5.

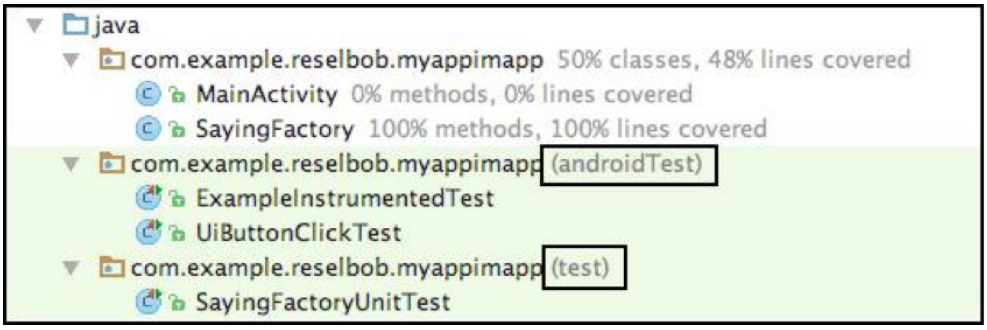

Figure 5: Android Studio distinguishes between device UI tests and business logic unit tests

Separating unit test code from UI test code according to category makes things easier when it's time to run the tests later on. Some developers will need to do unit testing only. Front end developers are more interested in UI tests. The same is true in terms of automated testing in a CI/CD process. At the least, unit tests tend to run faster than UI tests. UI tests take longer in part because of the time required to load in the device simulator. Thus, allowing tests to be executed according to category is useful in terms both of developer focus and general deployment needs.

RUNNING UNIT TESTS

Espresso allows a developer to run one or many test directly within Android Studio. All the developer needs to do is right click on the test or test category to run. A run dialog as shown in Figure 6 below appears.

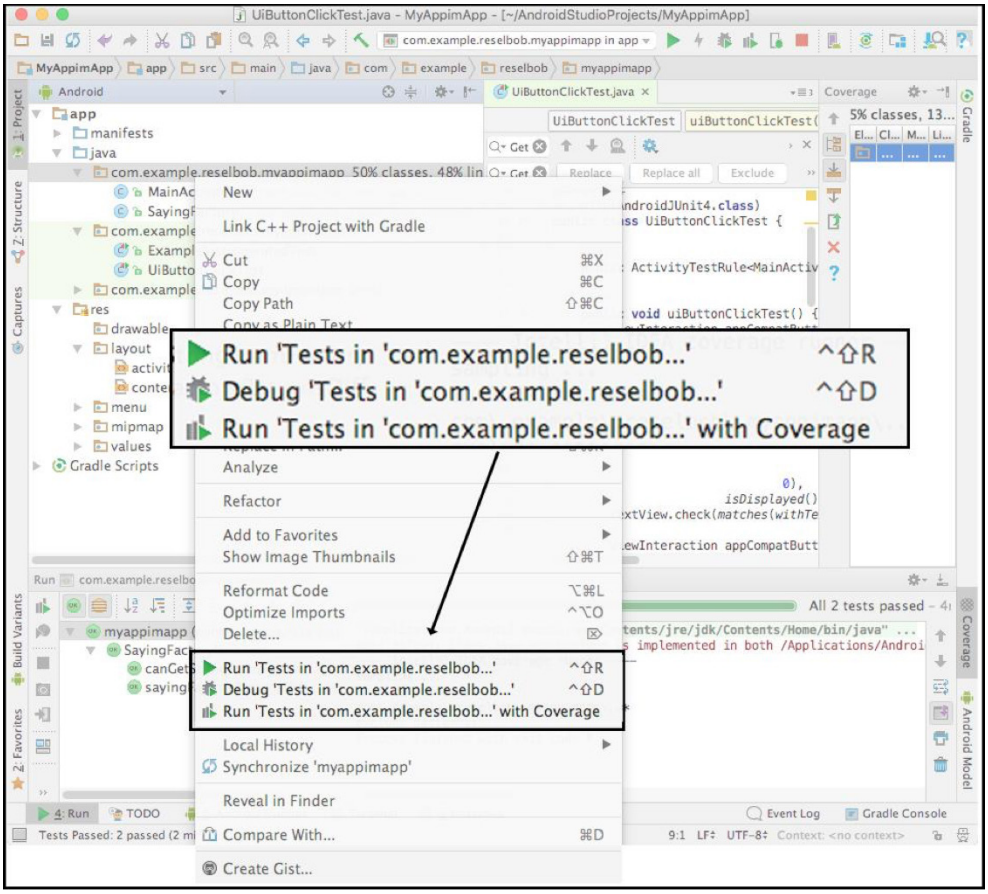

Figure 6: You can run Espresso tests in Android Studio

Then, either the developer can run the test or debug the test.

Being able to debug code from within a test session is a powerful feature for developers. Going to the location in an application's code that is causing a test to fail and then being able to inspect the values of variables and step through code at the failure point makes coding more efficient. It's better to catch test failure at development time than to have errors discovered later in the software development lifecycle. Writing tests during development and having a developer be able to debug failing tests during a testing session saves time and money.

Running Espresso Tests in the Cloud

Automated Continuous Integration and Delivery (CI/CD) is fast becoming a standard practice in the software development lifecycle of the modern enterprise. Thus, not only does testing need to be done by the developer at

design time, but also throughout the software development lifecycle, well after the code leaves human hands.

Standing up a CI/CD environment that supports machine-based, automated UI testing can be a laborious undertaking even for the most experienced engineers. Many enterprises reduce the costs of automated testing in the CI/CD process by using cloud based testing services. Testing services provided by companies such as Sauce Labs allow companies to run Espresso tests against their Android code in the cloud automatically. Using a cloud-based testing service allows a company to free up valuable engineering talent for other mission critical tasks.

TESTING AND CODE COVERAGE

Enterprises want to ensure that all code in force has been tested and performs to expectation. A test suite that exercises only 10% of the code written is of limited value. The quality of the remaining 90% is unknown. Large amounts of untested code create risks that companies cannot afford to take. Thus, a code coverage report becomes indispensable. Fortunately code coverage reports are easy to generate in Appium Studio when running Espresso tests.

Android Studio allows a developer to run tests and then analyze Espresso Tests in terms of code coverage. The developer can select the option to run a test along with Code Coverage. (Please see Figure 6 above). Code coverage reports are a powerful metric for determining the overall quality of the code and the degree to which the code has been tested.

RUNNING TESTS ACCORDING TO CONFIGURATION

Running tests from within Android Studio by right-clicking on the test and then selecting the run mode is useful in most cases. However, more complex situations require a developer to create special run configurations in which to run the test. Android Studio provides this capability. (Please see Figure 7.)

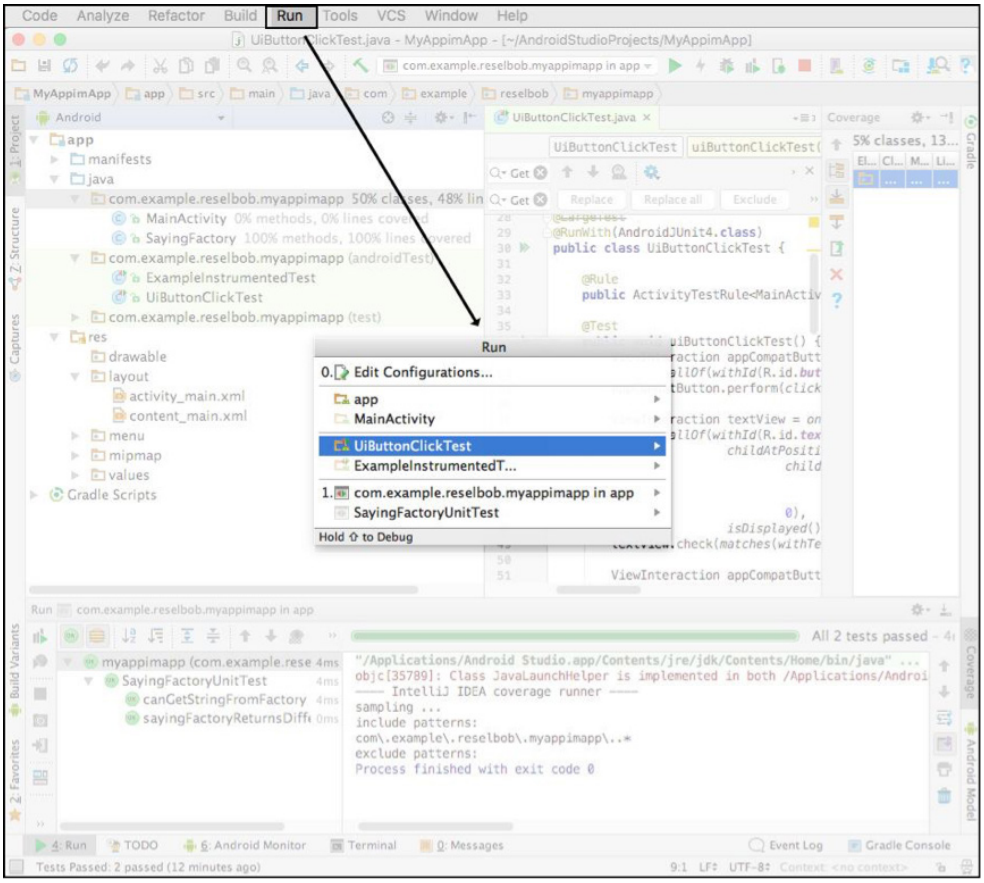

Figure 7: Android Studio allows you to configure test executions according to scope

For example, you can create a run configuration that defines the environment variables that the test needs to be functional. Or, a run configuration can be dedicated for a specific version of the Java JRE. Also, developers can set a run configuration to invoke another application that a test needs in order to operate properly. The important thing to understand is that Android Studio has the versatility to allow a developer to create one of many run configurations to meet the particular needs of a given testing scenario. Being able to create run configurations allows for a broad scope of testing without the developer having to do a lot of repetitive configuration work.

WORKING WITH XCUITEST

As mentioned earlier, XCUITest is the testing tool that ships with XCode. XCUITest provides developers with capabilities similar to those found in Espresso. However, while Espresso is dedicated to code written for the Android operating system, XCUITest is dedicated to Objective-C and Swift code that runs under iOS.

Writing tests in XCUITest is similar to the way you work in Espresso. First the developer or QA engineer needs to make ensure the application code is testable. This means that the code is well encapsulated, that business and data access logic is separated from UI code. The separation of concerns in the iOS version the Wise Saying application is similar to that of Android version. Data access logic encapsulated in a Swift class, `SayingsFactory`. Then, a device's UI code uses `SayingsFactory` to get data for display.

Listing 4 below shows the Swift 3 version of the `SayingsFactory`.

```
import Foundation

class SayingsFactory {

    static let sharedInstance = SayingsFactory();
    private var currentIndex = 0;

    var sayings = [“Be Kind To Strangers”,”Always Be Honest”,”The Truth is the
Best”,”Tip Well Always”];

    func getNextSaying()->String {
        if(currentIndex == sayings.count){
            currentIndex = 0;
        }
        let saying = sayings[currentIndex];
        currentIndex += 1;
        return saying;
    }
}
```

Listing 4: Implementing the SayingsFactory for iOS in Swift under XCode

Similar to the Android version, the `iOS` version of `SayingsFactory` is a singleton class that exposes the method, `getNextSay()`. `getNextSay()` returns the next saying from a predefined list.

The iOS application uses the `SayingsFactory` to get the text that is displayed when a user clicks the button on the device's UI. Listing 5 below shows the Swift-3 code that gets a saying from the `SayingsFactory` and displays the text in the UI.

```
import UIKit
class ViewController: UIViewController {

    @IBOutlet weak var lblSaying: UILabel!

    @IBAction func btnSaying(_ sender: Any) {
        lblSaying.text = SayingsFactory.sharedInstance.getNextSaying();
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

}
```

Listing 5: Applying data returned from the SayingsFactory from within an iOS button click action

Conceptually, the code for both the iOS and Android versions of the Wise Sayings is identical. The object models and syntax differ of course.

CREATING THE TEST FILES

Developers create unit and UI tests by declaring targets from within XCode. A developer will click Target from the File -> New menu. A dialog appears when the Target... menu item is selected as shown below in Figure 8.

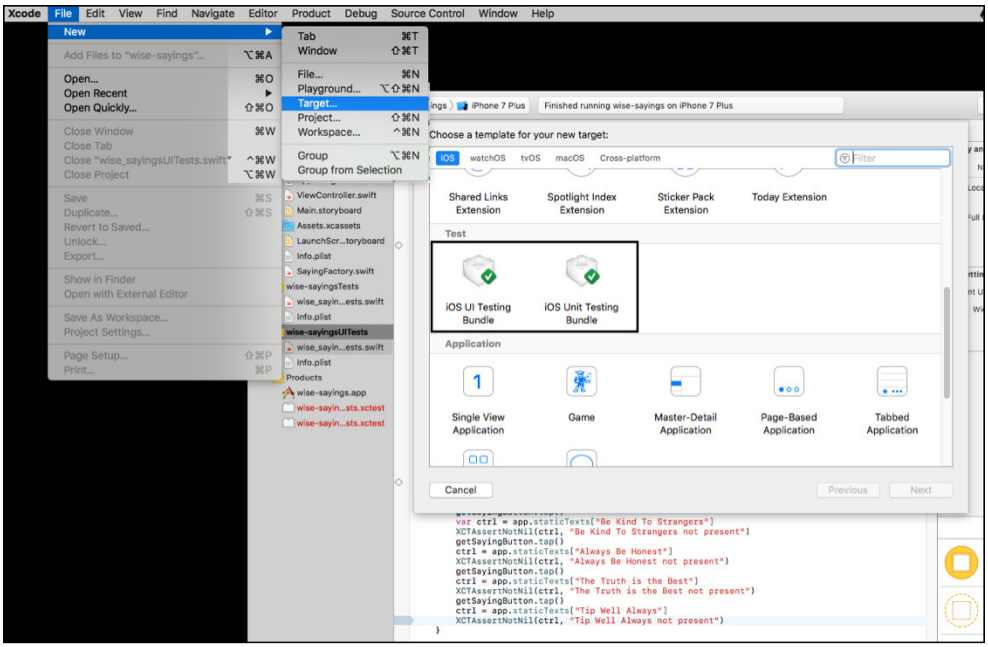

Figure 8: You declare XCUITest as targets from within XCode

The developer or QA Engineer selects either iOS UI Testing Bundle or iOS Unit Testing Bundle from among the target templates available in the dialog. XCode asks the developer to name the test. Then the IDE generates a folder according to the test name and adds files relevant to the test. One of these files will be a `info.plist` file. Another is a skeleton test class file. Figure 9 below shows the result of creating Unit Test and UI Test targets using XCode.

Writing a Unit Test is similar to the process that a developer does in Espresso. The developer will add Unit Test behavior manually to test methods in the skeleton test file. In terms of UI Testing, the developer will follow the process similar to that done in Espresso and use a UI test recorder to add code to the UI test file. Listing 6 below shows the UI test file skeleton that XCode creates.

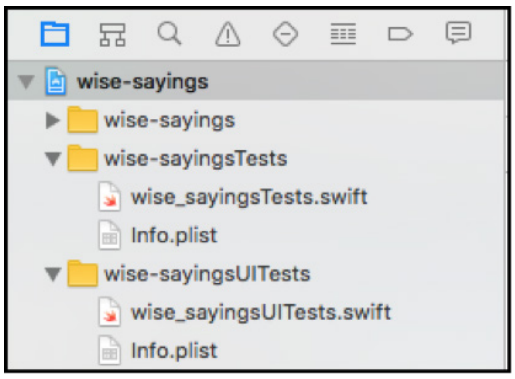

Figure 9: XCode supports the pattern of separating UI Tests from Unit Tests

```
import XCTest

class simple_unit_testing: XCTestCase {

    override func setUp() {
        super.setUp()
        // Put setup code here. This method is called before the invocation of each
test method in the class.
    }

    override func tearDown() {
        // Put teardown code here. This method is called after the invocation of
each test method in the class.
        super.tearDown()
    }

    func testExample() {
        // This is an example of a functional test case.
        // Use XCTAssert and related functions to verify your tests produce the
correct results.
    }

    func testPerformanceExample() {
        // This is an example of a performance test case.
        self.measure {
            // Put the code you want to measure the time of here.
        }
    }
}
```

Listing 6: A test class generated as part of the Unit Test Bundle workflow process

Figure 10 below shows the location of the test button that a developer clicks to start the XCUITest Recorder. The results of a recording session are added as code to the UI test file.

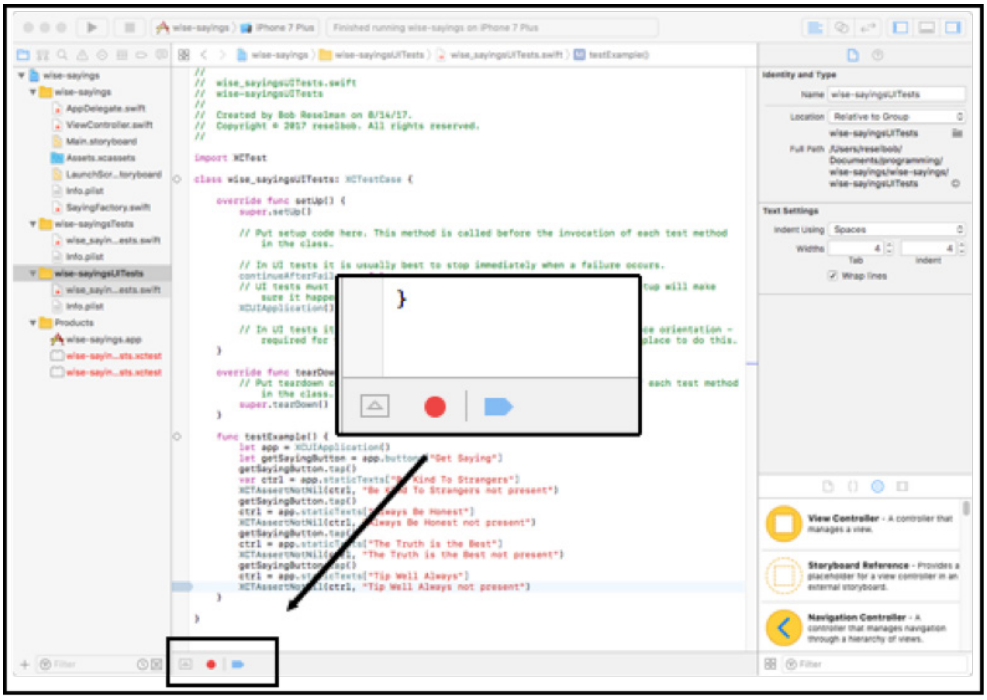

Figure 10: There is a XCUITest Recorder is embedded in the XCode IDE

Once test files are created, a developer uses the XCode IDE to compile the source and runs Unit and UI Tests.

RUNNING THE TEST

A developer runs the tests in an XCode development project by clicking Product -> Test in the XCode menu bar, as shown below in Figure 11.

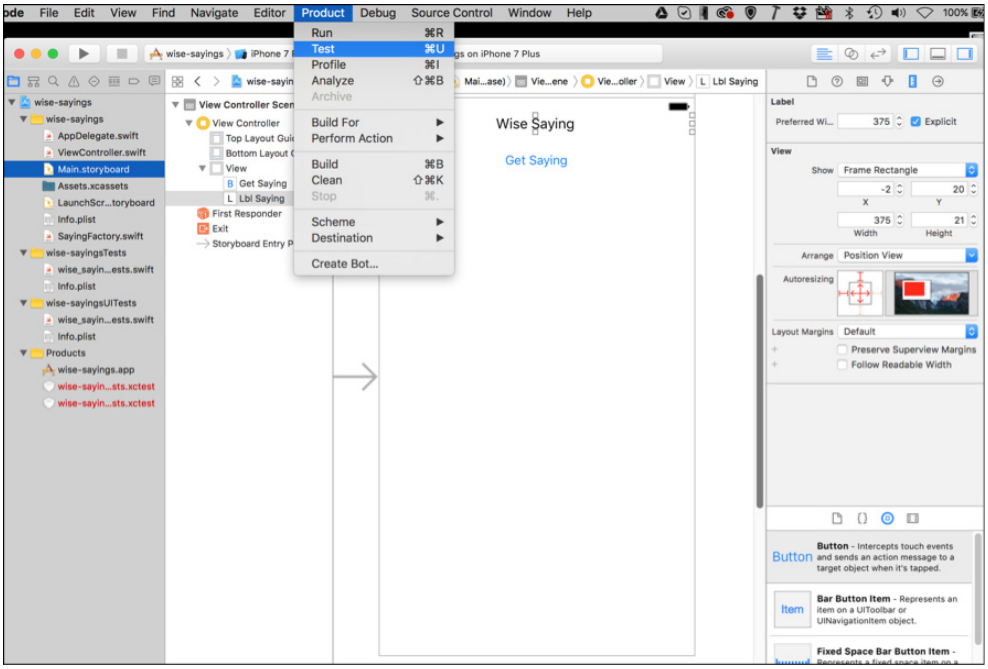

Figure 11: Developer and QA Engineers run XCUITest directly from within XCode

Also, XCode has a Test Navigator feature for those developers who want a higher degree of control over testing activity and more detail when viewing test results. Figure 12 below shows a view of the Unit Tests and UI Tests from within the Text Navigator. Developers access the Test Navigator by clicking the diamond-shaped icon on the left panel of the XCode IDE. (Notice that the Test Navigator icon is identified by a rectangle in the illustration.)

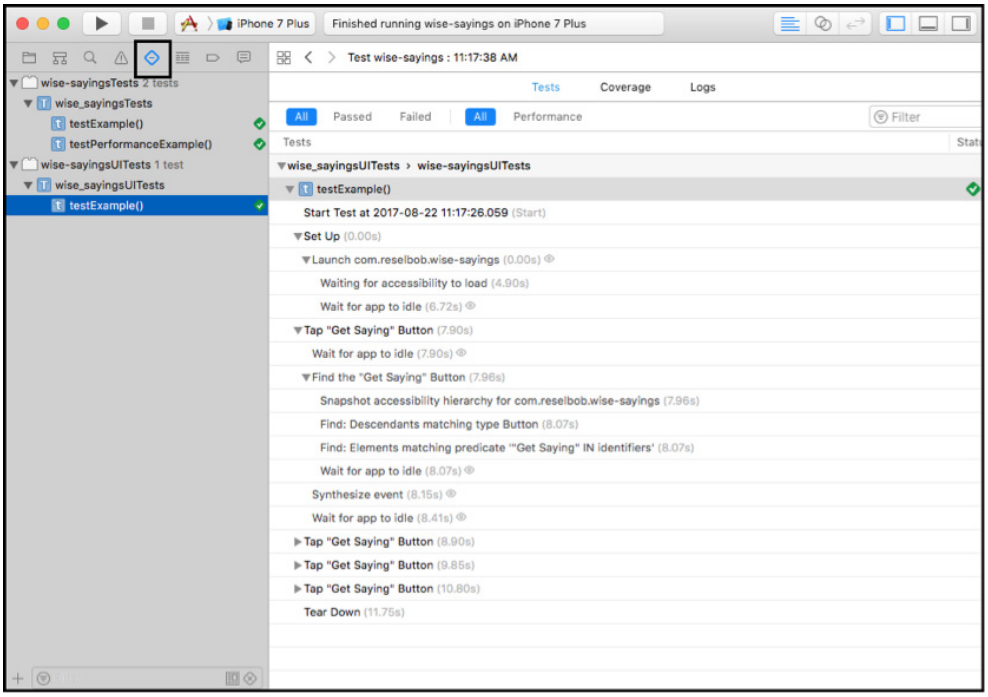

Figure 12: The Test Navigator provides more control and a detailed view of test activity

Clearly Apple has put a lot of work into integrating Unit and UI Testing capabilities directly into the XCode IDE. This reflects the industry trend that the most efficient way to ensure software quality at the code level is to have those closest to writing the code also be the ones writing the test. Thus, making it easy to write tests and view the results of that those tests is an important part of both the software development experience not only in XCode using XCUITest but also in Android Studio using Espresso.

Sauce Labs Provides Cloud Services for XCUITest for iOS Devices

Being able to implement automated testing in a cloud server is not only limited to Espresso testing. You can run XCUITest using cloud services as well. TestObject provides cloud based testing services that support automated XCUITest. Click here to view an introductory video about testing iOS code on Sauce Labs Real Device Cloud.

WORKING WITH APPIUM TO DO BLACK BOX TESTING

Although Espresso and XCUITest are powerful UI testing frameworks, they both share a common shortcoming. In order to use them you need to have direct access to the source code. For example, the following lines of Espresso test code invokes clicking the Get Sayings button on the UI:

```
ViewInteraction appCompatButton = onView(
       allOf(withId(R.id.button), withText("Get Saying"), isDisplayed()));
appCompatButton.perform(click());
```

And this line of XCUITest code invokes the same button click behavior:

```
let app = XCUIApplication()
let getSayingButton = app.buttons["Get Saying"]
getSayingButton.tap()
```

Conceptually the button click code above is similar, but each implementation is dramatically different. First, The way iOS organizes and names elements in the UI hierarchy is not at all like the way Android does it. And, the underlying programming languages differ. Android uses Java. iOS applications are written in Objective-C or Swift.

The type of testing in which the testing code needs direct access to the source code is called White Box Testing. White Box Testing is good as long the developer and tester have access to application before it is compiled and deployed. But, how do you write effective UI testing when all you have is the deployed executable such as an Android APK or an iOS APP file? In such cases you can use Appium.

Appium is an open source project that's well suited to writing test in situations in which the source code is not available. Appium was designed from the beginning to be useful when only an application's deployment artifact is available. Appium is divided into 2 components, the Appium Desktop and the Appium Server.

Appium Desktop is the test creation tool that's intended to be run on a developer's machine. Appium Desktop sits on top of the Appium Testing Server. (Please see Figure 12.)

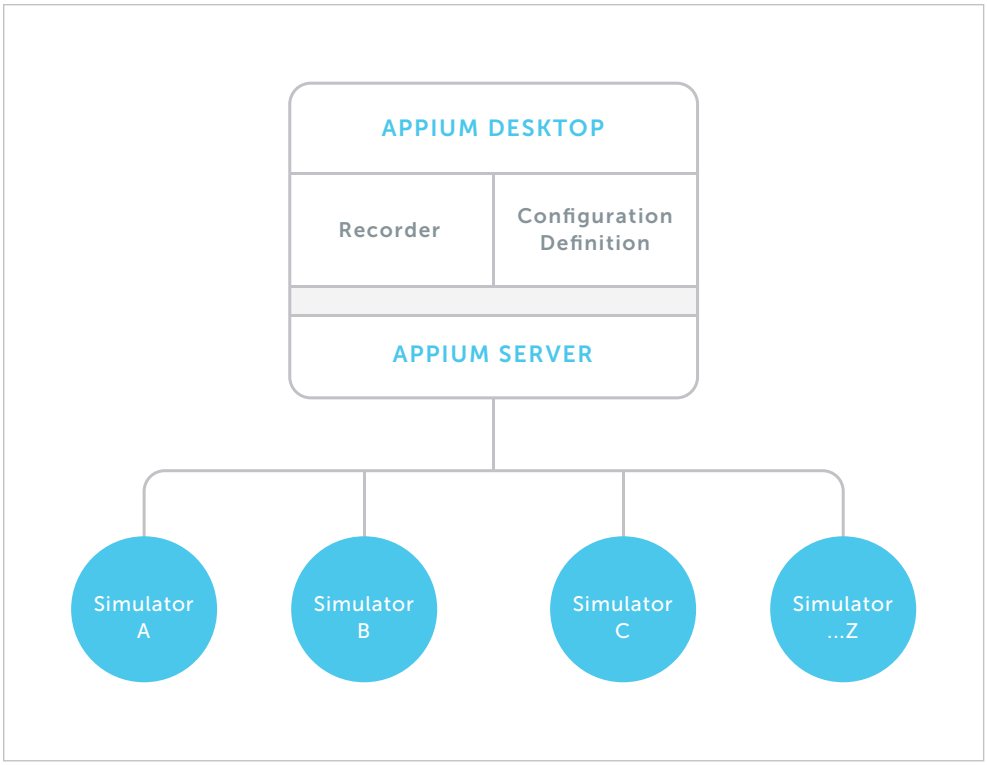

Figure 12: Appium Desktop sits on top of Appium Server

The Appium Desktop is an IDE similar to Espresso and XCUITest in that it allows a developer to set test configuration and record tests against a mobile application's UI. Figure 12 above shows a screenshot of Appium Desktop running the Android version of the Wise Saying demonstration application in an Android Simulator.

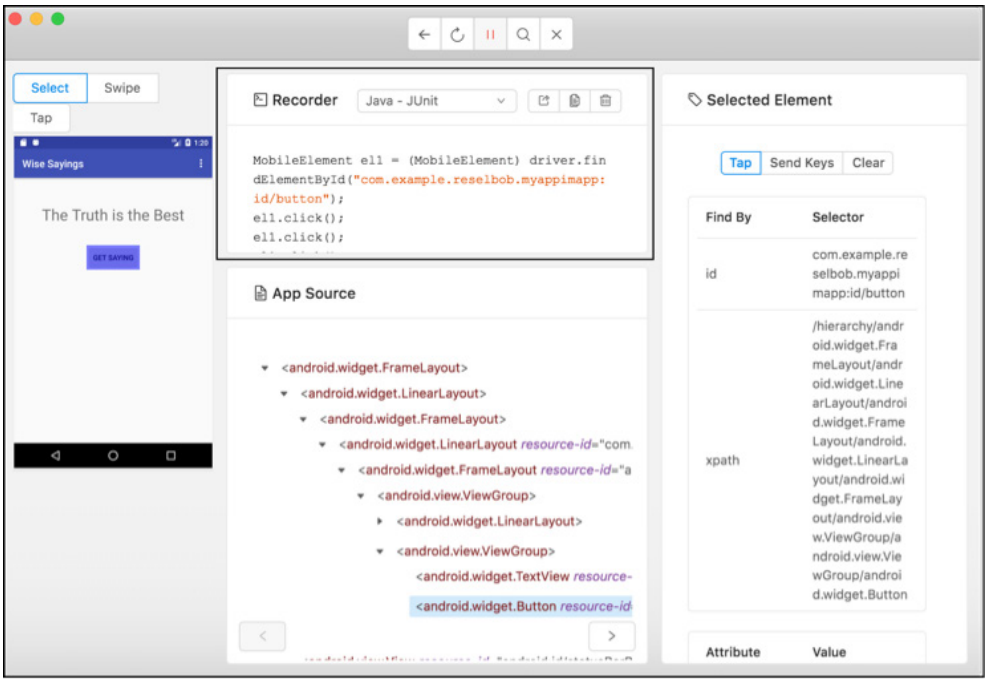

Figure 13: Appium Desktop has a test recorder that produces Black Box test scripts in a variety of languages

Appium Server contains the device simulators in which an application will be run. Once an application is up and running in a simulator, the Appium Server runs tests against the simulated application. Which simulator to use depends on the type of application being tested.

Appium does not need to have access to the source code. It has the intelligence to inspect a deployment executable and determine the UI element structure of an application. Then a developer uses the Appium Recorder to keep track of actions as the user taps through the application's UI and enters data using the simulator keyboard when necessary. The Appium Recorder translates the recording into code to which the developer adds assertion statement to verify expected behavior.

The test code is independent of the source code and is run as a separate project. Think of the testing project as a test runner that exercises the application's UI. The testing project can be written in any one of the variety of languages that Appium supports, for example, Java, .NET, Ruby or PHP. Remember, Appium has no concern with the language in which the application is written. It exposes the application's underlying UI structure in a format that is readable by a variety of languages.

Getting the familiar working Appium involves mastering a three step process. First the developer creates the UI tests using the Appium Desktop Recorder. Second, the developer submits to the Appium Server a JSON document that describes testing configuration. Third, the developer commands the server to run tests.

The Appium Server does the work of executing tests. It's transparent in terms of the Appium Desktop. However, when it comes time to use Appium as the testing framework in a CI/CD setting, you can set up separate Appium servers on a network to run many testing sessions. Or, if you want to save time and avoid the work of setting up a Appium Server and attaching the various simulators your testing environment requires, you can use a cloud-based Appium Service Provider to execute UI testing in a CI/CD workflow. Sauce Labs is one such platform.

FINDING THE RIGHT FRAMEWORK FOR THE RIGHT JOB

Espresso, XCUITest and Appium are all useful frameworks for writing tests for mobile applications. However, no one framework is applicable to all situations. Espresso and XCUITest are great frameworks for mobile testing when source

code is available and when the developer is working in an IDE. Appium is the mobile application testing framework to use when all that is available is the APK or APP deployment artifacts. All come with tradeoffs. Finding the right automation framework for the right job requires detailed consideration.

PUTTING IT ALL TOGETHER

Automated testing is an essential aspect of modern mobile application development. Gone are the days of rooms full of QA testers typing away on a cell phone according to list of instructions and saving the results in a spreadsheet shared by all. The volume and complexity of code that needs to be tested simply won't allow it. Today developers and QA Engineers need to write and execute faster. And the tests that are created need to be runnable in the CI/CD environment. The demands can be herculean.

Fortunately the frameworks have kept up with the times. Espresso, XCUITest and Appium make automated testing available to all, from the single developer to the large scale enterprise. These frameworks are free to use. Espresso and XCUITest are built right into their respective IDEs. Appium is open source. The only cost is the time necessary to learn how to use them.

This is the easy part.

The real trick is to get more enterprises to adopt the frameworks in their software development lifecycle. When it comes to mobile application development, in the not too distant future there will be two types of companies: Those that have embraced using automation tools and frameworks in mobile app testing and those that haven't. Those that have will enjoy a bright future of growth and prosperity meeting the increasing demands of the marketplace. Those that haven't will fall by the wayside as one of those companies that should have known better but didn't.

ABOUT SAUCE LABS

Sauce Labs ensures the world's leading apps and websites work flawlessly on every browser, OS and device. Its award-winning Continuous Testing Cloud provides development and quality teams with instant access to the test coverage, scalability, and analytics they need to deliver a flawless digital experience. Sauce Labs is a privately held company funded by Toba Capital, Salesforce Ventures, Centerview Capital Technology, IVP and Adams Street Partners. For more information, please visit saucelabs.com.

SAUCELABS

SAUCE LABS INC. - HQ
116 NEW MONTGOMERY STREET, 3RD FL
SAN FRANCISCO, CA 94105 USA

SAUCE LABS EUROPE GMBH
NEUENDORFSTR. 18B
16761 HENNIGSDORF GERMANY

SAUCE LABS INC. - CANADA
134 ABBOTT ST #501
VANCOUVER, BC V6B 2K4 CANADA