Getting Started with Appium for Mobile Native Application Testing

Appium was originally developed by Dan Cueller as a way to take advantage of the UIAutomation framework for Apple iOS to run tests against native mobile applications. Using the same syntax as Selenium, it shares Selenium’s ability to automate interaction with a website through a mobile browser, with additional features for interacting with elements on mobile devices, such as gestures. For this reason, while Appium can be used for website testing against mobile and desktop browsers, it is more commonly used for testing native and hybrid mobile applications for both iOS and Android.

This topic is intended to provide you with a quick overview of how Appium works, and the basic components of a native mobile application test script. For full documentation of Appium, with extensive examples of its syntax in the most popular scripting languages, check out the Appium.io website.

See the following sections for more information:

- Appium Architecture
- The Example Application
- Setting the Location of the Application to Test
- Creating an Instance of the Appium WebDriver Interface
- Locating an Element In the Application
- Performing an Action on an Application Element
- Anticipating Application Response
- Running Tests and Recording Test Results
- Concluding a Test
- Example with All Steps
- More Information
Appium Architecture

Like Selenium, Appium has a **client-server architecture**.

**Appium Client** includes:

- A set of **client libraries for various scripting languages** in which you will write your test scripts, which are based on the Selenium WebDriver API

**Appium Server** includes:

- A server component, based on node.js, which exposes the WebDriver API. In fact it exposes a superset of the WebDriver API known as the **Mobile JSON Wire Protocol**.
- A desktop application, available for both OS X and Windows, that includes everything you need to run Appium bundled in a single package, as well as the ability to inspect elements in running applications. Note that this is currently unsupported by the Appium core team.

The Seven Basic Steps of Native Application Testing

**Mobile Website Testing with Appium**

Automating mobile browsers to test websites with Appium is almost identical to the process you would use to test with Selenium, though there is a new set of desired capabilities and there are additional methods available for mobile-specific behaviors. Check out [Getting Started with Selenium for Automated Website Testing](#) for an overview of the process, and [Examples of Test Configuration Options for Website Tests](#) for examples of setting Appium desired capabilities for mobile web testing.

There are seven basic steps in creating an Appium test script, which apply to any test case and any application you want to test:

1. Set location of the application to test in the desired capabilities of the test script.
2. Create an Appium driver instance which points to a running Appium server (or Sauce Labs).
3. Locate an element within the native application.
4. Perform an action on the element.
5. Anticipate the application response to the action.
6. Run tests and record test results using a test framework.
7. Conclude the test.
## The Example Application

### Example Android Element

```xml
<android.widget.EditText index="13" text="We really would like to follow up!" class="android.widget.EditText" package="com.saucelabs.guineapig" content-desc="" checkable="false" checked="false" clickable="true" enabled="true" focusable="true" focused="true" scrollable="false" long-clickable="true" password="false" selected="false" bounds="[32,717][532,797]" resource-id="com.saucelabs.guineapig:id/fbemail" instance="1"/>
```

This is an example element from a Android application. It refers to a text input where a user could input their email address.

### Example iOS Element

```xml
<UIATextField name="fbemail" label="fbemail" value="We really would like to follow up!" dom="" enabled="true" valid="true" visible="true" hint="fbemail" path="/0/0/0/13" x="5" y="264" width="275" height="31"/>
```

This is the same element from an example iOS application.
Setting the Location of the Application to Test

When you write an Appium test script, the most basic component is the DesiredCapabilities object, which sets the parameters of your test, such as the mobile platform and operating system you want to test against. Within that object, one of the required capabilities is Application Path, or the `app` desired capability. One of the advantages of the Appium architecture is that the application you want to test can be hosted anywhere, from a local path to any other web host on the network, since the Appium server will send the commands it receives from the client to any application path you specify. Practically, you have three options.

Other Online Locations

Because the Appium server issues its commands over HTTP using the JSON wire protocol, it can interact with an application in any location where the application can receive those commands. Your application could be hosted on Amazon Web Services, Dropbox, or any other network-accessible location. For `app`, you would just specify the full URL to the application. Uploading Mobile Applications to Other Online Locations for Testing includes information on how you can use Sauce Labs to test applications hosted on other services.

Sauce Storage

Sauce Labs provides temporary secure storage for applications you want to test using our service. You upload the application as a `.zip`, `.ipa`, or `.apk` file, and then use the `sauce-storage` parameter for `app` to specify the application you want to test. Check out the topic Uploading Mobile Applications for Testing for more information.

Local

If you are running Appium server locally on the same machine where the application you want to test is located, you simply specify the absolute path to it for `app`. For example, `/abs/path/to/my.apk`. If your application is located a local http server, you can also use Sauce Connect to test it with Sauce Labs emulators, simulators, or real devices.
Creating an Instance of the Appium WebDriver Interface

The WebDriver instance is the starting point for all uses of the Mobile JSON Wire Protocol.

You create an instance of the WebDriver interface using a constructor for either Android or iOS. For mobile native application tests, you set both the platform and browser to test against by setting the BrowserName desired capability.

Once you have created an instance of the WebDriver interface, you use this instance to invoke methods, such as tap and swipe, to access other interfaces used in basic test steps. You do so by assigning the instance to a variable when you create it, and by using that variable to invoke methods.

Example

This example instantiates the Android WebDriver, and assigns it a variable named driver.

<table>
<thead>
<tr>
<th>Appium Driver Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebDriver driver = new AndroidDriver&lt;WebElement&gt;( new URL(&quot;http://SAUCE_USERNAME:<a href="mailto:SAUCE_ACCESS_KEY@ondemand.saucelabs.com">SAUCE_ACCESS_KEY@ondemand.saucelabs.com</a>:80/wd/hub&quot;), capabilities);</td>
</tr>
</tbody>
</table>

This is the same example instantiating the iOS WebDriver.

| WebDriver driver = new IOSDriver<WebElement>( new URL("http://SAUCE_USERNAME:SAUCE_ACCESS_KEY@ondemand.saucelabs.com:80/wd/hub"), capabilities); |
Locating an Element In the Application

Isaac Murchie has written a great post on the Sauce blog about the various strategies you can use to locate application elements. This section is an excerpt of the most basic approaches from that post.

In order to find elements in a mobile environment, Appium implements a number of locator strategies that are specific to, or adaptations for, the particulars of a mobile device. Three are available for both Android and iOS:

**class name**

The class name strategy is a string representing a UI element on the current view.

- For iOS it is the full name of a UI Automation class, and will begin with UIA, such as UIATextField for a text field. A full reference can be found here.
- For Android it is the fully qualified name of a UI Automator class, such as android.widget.EditText for a text field. A full reference can be found here.

The client libraries for Appium support getting a single element, or multiple elements, based on the class name. This functionality is in the Selenium clients (e.g., Python).

```python
# python
driver.find_element_by_class_name('android.widget.DatePicker')
```

**accessibility id**

The accessibility id locator strategy is designed to read a unique identifier for a UI element. This has the benefit of not changing during localization or any other process that might change text. In addition, it can be an aid in creating cross-platform tests, if elements that are functionally the same have the same accessibility id.

- For iOS this is the accessibility identifier laid out by Apple here.
- For Android the accessibility id maps to the content-description for the element, as described here.

For both platforms getting an element, or multiple elements, by their accessibility id is usually the best method. It is also the preferred way, in replacement of the deprecated name strategy.

The client libraries specific to Appium support getting elements by accessibility id. The methods are not yet implemented in the standard Selenium clients.

```python
# python
driver.find_element_by_accessibility_id('my_accessibility_identifier')
```

**xpath**

The xpath locator strategy is also available in the WebDriver protocol, and exposes the functionality of XPath language to locate elements within a mobile view. An XML representation of the view is created in Appium, and searches are made against that image.

The Selenium clients have methods for retrieving elements using the xpath locator strategy.

```python
# python
driver.find_element_by_xpath('//UIAApplication[1]/UIAWindow[1]/UIATextField[1]')
```

**id**

In the mobile environment, ids are not, as in WebDriver, CSS ids, but rather some form of native identifier.

- For iOS the situation is complicated. Appium will first search for an accessibility id that matches. If there is none found, a string match will be attempted on the element labels. Finally, if the id passed in is a localization key, it will search the localized string.
- For Android, the id is the element's android:id.

Example: Locate elements for username and password

This example invokes the findElement method on the driver variable, using the name attribute to locate the username and password text input elements, and (optionally) the id attribute to locate the form element.
import org.openqa.selenium.By;
import org.openqa.selenium.WebElement;
WebElement emailInput = driver.findElement(By.id("fbemail"));

Best Practices for Identifying Application Elements

It is always best to use an element locator that uniquely identifies the element, like an id or an accessibility id. Class names and xpath are best used only when IDs are not available. Multiple elements can have the same class name, and using xpath searches through the entire markup to find the element, which can slow down your tests.
Performing an Action on an Application Element

Once you've identified the mobile elements you want your test to interact with, the next step is to interact with them. You perform an action on a mobile element by invoking an interaction method on an instance of the WebElement interface.

The WebElement interface declares basic interaction methods including:

- The sendKeys method, to enter text
- The clear method, to clear entered text
- The submit method, to submit a form

Example

This example first invokes the sendKeys method to enter text in the username and password elements, and then invokes the submit method to submit the login form.

Enter a user name and a password

emailInput.sendKeys("SauceIsAwesome@email.com");

Submit the form

The submit method can be invoked either on any text input element on a form, or on the form element itself.

Submit Text Element

emailInput.submit();
Anticipating Application Response

When you click a Submit button, you know that you have to wait a second or two for your action to reach the server, and for the server to respond, before you do anything else. If you’re trying to test the response, and what happens afterwards, then you need to build that waiting time into your test. Otherwise, the test might fail because the elements that are expected for the next step haven’t loaded into the browser you. The WebDriver API supports two basic techniques for anticipating browser response by waiting: implicit waits and explicit waits.

**Do Not Mix Explicit and Implicit Waits**

Do not mix implicit and explicit waits. Doing so can cause unpredictable wait times. For example setting an implicit wait of 10s and an explicit wait of 15 seconds, could cause a timeout to occur after 20 seconds.

**Implicit Waits**

Implicit waits set a maximum time that the Appium server will continue trying to find an element. Using implicit waits is not a best practice because application response times are not definitely predictable and fixed elapsed times are not applicable to all interactions. Using explicit waits requires more technical sophistication, but is a Sauce Labs best practice.

**Explicit Waits**

Explicit waits wait until an expected condition occurs on the web page, or until a maximum wait time elapses. To use an explicit wait, you create an instance of the WebDriverWait class with a maximum wait time, and you invoke its until method with an expected condition.

The WebDriver API provides an ExpectedConditions class with methods for various standard types of expected condition. These methods return an instance of an expected condition class. You can pass an invocation of these standard expected-condition methods as argument values to until method. You can also pass - in ways that your programming language and its WebDriver API support - any function, code block, or closure that returns a boolean value or an object reference to a found web element as an argument value to the until method. How this is done varies over programming languages. The until method checks repeatedly, until the maximum wait time elapses, for a true boolean return value or a non-null object reference, as an indication that the expected condition has occurred.

**Example**

This example code illustrates how you could use either an explicit wait or an implicit wait to anticipate web browser response after submitting the login form.

**Explicit Wait**

```java
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
WebDriverWait wait = new WebDriverWait(driver, 10);
WebElement messageElement = wait.until( ExpectedConditions.
presenceOfElementLocated(By.id("loginResponse")) );
```

**Implicit Wait**

```java
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
```
Running Tests and Recording Test Results

Running tests and recording test results is the ultimate purpose of your test script: you run tests in an automated test script in order to evaluate function and performance in the AUT, without requiring human interaction.

Test Frameworks

To run test and to record test results, you use methods of a test framework for your programming language. There are many available test frameworks, including the frameworks in the so-called XUnit family, which includes:

- JUnit for Java
- NUnit for C#
- unittest or pyunit for Python
- RSpec for Ruby

For some programming languages, test frameworks other than those in the XUnit family are common - for example, the RSpec framework for Ruby. The Sauce Labs Sample Test Framework repos on GitHub contain over 60 examples of test frameworks set up to work with Sauce Labs.

Assertions

Most test frameworks implement the basic concept of an assertion, a method representing whether or not a logical condition holds after interaction with an AUT. Test frameworks generally declare methods whose names begin with the term assert and end with a term for a logical condition, e.g. assertEqual in JUnit. Generally, when the logical condition represented by an assert method does not hold, an exception for the condition is thrown. There are various approaches to using exceptions in most test frameworks. The SeleniumHQ documentation has more detailed information on using both assertions and verifications in your tests.

Recording Test Results

Recording of test results can be done in various ways, supported by the test framework or by a logging framework for the programming language, or by both together. Selenium also supports taking screenshots of web browser windows as a helpful additional type of recording. Because of the wide variations in recording technique, this beginning section omits recording, instead emphasizing a simple approach to applying a test using an assert method. The scripts in Sauce Labs Demonstration Scripts include examples of setting up reporting of test results to Sauce Labs, as do the framework scripts in the Sauce Labs Sample Test Frameworks github repos.

Example

The following example runs a test by asserting that the login response message is equal to an expected success message:

```java
import junit.framework.Assert;
import junit.framework.TestCase;

WebElement messageElement = driver.findElement(By.id("loginResponse"));
String message = messageElement.getText();
String successMsg = "Welcome to foo. You logged in successfully."
assertEquals (message, successMsg);
```
Concluding a Test

The quit Method

You conclude a test by invoking the quit method on an instance of the WebDriver interface, e.g. on the driver variable.

The quit method concludes a test by disposing of resources, which allows later tests to run without resources and application state affected by earlier tests. The quit method:

- quits the web browser application, closing all web pages
- quits the WebDriver server, which interacts with the web browser
- releases driver, the variable referencing the unique instance of the WebDriver interface.

Example

The following example invokes the quit method on the driver variable:

```javascript
driver.quit();
```
Example with All Steps

The following example includes code for all steps. The example also defines a Java test class `SampleSauceTest`, and its `main` method, so that the code can be run.

```java
package com.yourcompany;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.remote.DesiredCapabilities;
import io.appium.java_client.android.AndroidDriver;
import java.net.MalformedURLException;
import java.net.URL;
import junit.framework.Assert;
public class SampleSauceTest {
    public static void main() throws MalformedURLException {
        DesiredCapabilities capabilities = new DesiredCapabilities();
        capabilities.setCapability("platformName", "Android");
        capabilities.setCapability("deviceName", "Samsung Galaxy S4 Emulator");
        capabilities.setCapability("platformVersion", "4.4");
        capabilities.setCapability("app", "sauce-storage:GuineaPigApp-debug.apk");
        capabilities.setCapability("deviceOrientation", "portrait");
        capabilities.setCapability("appiumVersion", "1.5.1");
        WebDriver driver = new AndroidDriver<WebElement>(new URL("http://SAUCE_USERNAME:SAUCE_ACCESS_KEY@ondemand.saucelabs.com:80/wd/hub"), capabilities);
        WebElement emailInput = driver.findElement(By.id("fbemail"));
        emailInput.sendKeys("SauceIsAwesome@email.com");
        assertEquals(emailInput.getText(), "SauceIsAwesome@email.com");
        driver.quit();
    }
}
```
More Information

There are many additional resources available if you want to dive into more detail with Appium and mobile application testing.

- [http://appium.io](http://appium.io)
  The official Appium website and documentation
- [https://youtu.be/1J0aXDbjUE?list=PLSIUOFhmxEICODb8XQB-RUQ0RGNZ2yW7d](https://youtu.be/1J0aXDbjUE?list=PLSIUOFhmxEICODb8XQB-RUQ0RGNZ2yW7d)
  An introduction to Appium presented by Jonathan Lipps of Sauce Labs and the Appium project given at the 2013 Google Test Automation Conference
  A talk on the mobile JSON wire protocol presented by Jonathan Lipps at the 2015 Selenium Conference
  An in-depth tutorial by Jonathan Lipps covering Appium basics using Ruby and Sauce Labs