GETTING STARTED WITH APPIUM

By Jonathan Lipps

Java Edition 2018.4

TABLE OF CONTENTS

PREFACE

This little e-book will help you get started with Appium using the Java programming language and the Java Appium client. It is the second in a new series of "bootcamp" guides, beginning with Ruby. The Ruby guide was loosely based on an earlier guide written by Dave Haeffner, who along with Matthew Edwards deserves a lot of thanks for all the work put into their first edition. This guide is similar in content and learning objectives, but focuses on issues specific to use of the Java client.

Appium is an open source project that is always changing, and guides like this one will never be accurate forever. When possible I will indicate which versions of various software are being used, which might help in ensuring reproducibility of the code samples used here.

As the Appium project lead, I benefit from the work of the entire community in being able to write a guide like this. Appium would not be what it is today without the maintainers and users who have decided to throw their lot in with our take on mobile automation. The credit for this book as well as for Appium as a whole go far and wide! Thanks especially to Sauce Labs who commissioned the writing of this guide, and the maintainers and contributors of the Appium Java client, including: @TikhomirovSergey, @SrinivasanTarget, and @mykola-mokhnach.

Jonathan Lipps
April 2018
Vancouver

Chapter 1

INTRODUCTION

Appium is a tool for automating apps. It has two components: the Appium server, which does the actual automating, and a set of Appium clients, one for every popular programming language. You write tests in your favorite language by importing the Appium client for that language and using its API to define a set of test steps. When you run the script, those steps are sent one-by-one to a running Appium server, which interprets them, performs the automation, and sends back a result if appropriate.

Appium was initially developed to automate mobile applications, first iOS and then Android. In recent years Appium has gone beyond mobile to support Desktop or even TV apps. This guide focuses on mobile testing for iOS and Android.

There are several kinds of mobile apps, and Appium lets you automate all of them:

1. Native apps — apps built using the native mobile SDKs and APIs
2. Web apps — websites accessed using a mobile browser
3. Hybrid apps — apps with a native container and one or more webviews embedded in that container. The webviews are little frameless browser windows that can show content from the web or from locally-stored HTML files. Hybrid apps allow the use of web technologies within a native-like user experience.

This guide focuses on automating native apps. Switching to web or hybrid automation is a breeze once you're familiar with the basic principles of Appium automation, and plenty of information can be found online about automating the other app modes.

THE APPIUM VISION

Appium is both more and less than an automation library. It is less than an automation library because Appium itself relies on other, more basic automation tools in order to run behaviors on mobile devices. The Appium team decided long ago not to compete on the fundamentals of functional automation. Apple or Google are well-positioned to release tools that efficiently automate one of their mobile devices. What Appium does bring is a standard interface on top of all of these disparate technologies.

Appium implements the WebDriver Protocol, a W3C standard defining browser automation. It's the same protocol that Selenium uses, meaning your Selenium knowledge will translate completely to Appium skill. So Appium is fundamentally about providing you access to the best automation technologies that are out there, within a standard WebDriver interface accessible from any programming language or test client.

Importantly, Appium is totally open source. Owned by the JS Foundation, Appium has open governance and contribution processes. The Appium team believes that open is the way to go, and the meteoric rise of Appium as a project is a testament to this approach.

THE APPIUM DRIVERS

How does Appium organize itself to meet its vision? Each automation technology provided by Appium is wrapped up into a bit of code called an *Appium driver*. Each driver knows how to translate the WebDriver protocol to that particular technology. And they all do quite a bit more than that, too—most of them take care of setting up and running the underlying technology as well.

What this means for you is that you are not just using Appium. You're using Appium in conjunction with one or more drivers. Even one platform (like Android), might have multiple supported Appium drivers, which target different fundamental automation technologies. For example, you can pick between the `appium-uiautomator2-driver` and the `appium-espresso-driver` when it comes to writing your Android tests. It's worth getting to know the different drivers so that you're sure you're using the best one for your tests. While Appium does its best to ensure automation commands do the same thing across different drivers, sometimes underlying differences make this impossible. For the Appium code samples in this guide, the iOS driver we'll be using is `appium-xcuitest-driver`, and the Android driver will be `appium-uiautomator2-driver`.

THE APPIUM CLIENTS

One of the great things about Appium is that you can write Appium scripts in any language. Because Appium is built around a client-server architecture, clients can be written in any programming language. These clients are simply fancy HTTP clients, which encapsulate HTTP calls to the Appium server inside nice user-facing methods (usually in an object-oriented fashion). This guide will be using the Appium Java client, which is distributed via Maven as

`io.appium/java-client`. The Appium Java client is not a standalone library: it is actually a wrapper around the standard Selenium Java client. So if you're already familiar with the Selenium client, you'll find it easy to understand the Appium version.

OK, time to get your system set up to run Appium tests!

Chapter 2

GETTING SET UP

Getting going with Appium itself is fairly straightforward. However, Appium depends on the various mobile SDKs and system dependencies in order to perform its magic. This means that even if you're not an app developer yourself, and don't plan on writing iOS or Android code, you'll need to get your system set up as if you were. Don't worry—this guide will walk you through it. I should point out for our Windows and Linux friends that this guide assumes a Mac environment, since iOS testing can only be done on a Mac. If you're only interested in Android testing and want to run Appium on Windows or Linux, refer to the Appium documentation for setup information specific to your host OS.

ASSUMED KNOWLEDGE

This guide is meant to be a reasonably in-depth introduction to Appium. However, we do assume certain kinds of knowledge. For example, we expect that you know your way around the command line terminal on your Mac, and already know the Java programming language well enough to follow along with simple examples. Finally, we'll assume enough familiarity with a Java IDE or other Java tools to create Java projects and run Java programs. If any of these assumptions are not true for you, stop here and do some digging on the Internet until you've found a good tutorial on those topics before you continue following this guide.

IOS-SPECIFIC SYSTEM SETUP

- Install Xcode
- Install Xcode's CLI tools (you'll be prompted the first time you open a fresh version of Xcode)
- Install Homebrew
- Install Carthage via Homebrew: `brew install carthage`

ANDROID-SPECIFIC SYSTEM SETUP

- Install Android Studio and SDK Tools
- Using the Android Studio SDK Manager, download the latest Android SDK, build tools, and emulator images

- Install the Java Development Kit (JDK)
- In your shell login file (`~/.bashrc, etc…`):
  - Export `$ANDROID_HOME` to the location of the Android SDK on disk. If you didn't set this manually, it's probably at the default location
  - Ensure that the appropriate directories are added to your `$PATH` so that the `adb` and `emulator` binaries are available from the command line
  - Export `$JAVA_HOME` to the `Contents/Home` directory inside of the newly-installed JDK (where does JDK get installed?)
  - Ensure that the appropriate directories are added to your `$PATH` so that the JDK's binaries are accessible from the command line
- Configure an Android Virtual Device (AVD) in Android Studio. The particular device doesn't matter. This will be the emulated device we use for Android testing in this guide.

APPIUM SETUP

There are two ways to install officially released versions of Appium: either from the command line via NPM or by installing Appium Desktop.

Appium From the Command Line

Appium is shipped as a Node.js package, so if you have Node.js and NPM installed on your system, you can simply run:

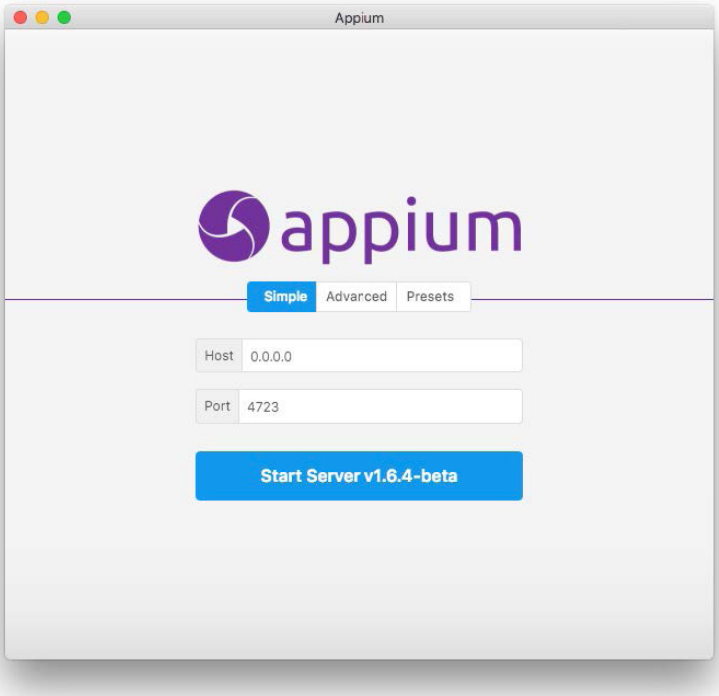
```
npm install -g appium
```

And the most recent version of Appium will be installed. You can then run Appium simply by typing `appium` from the command line.

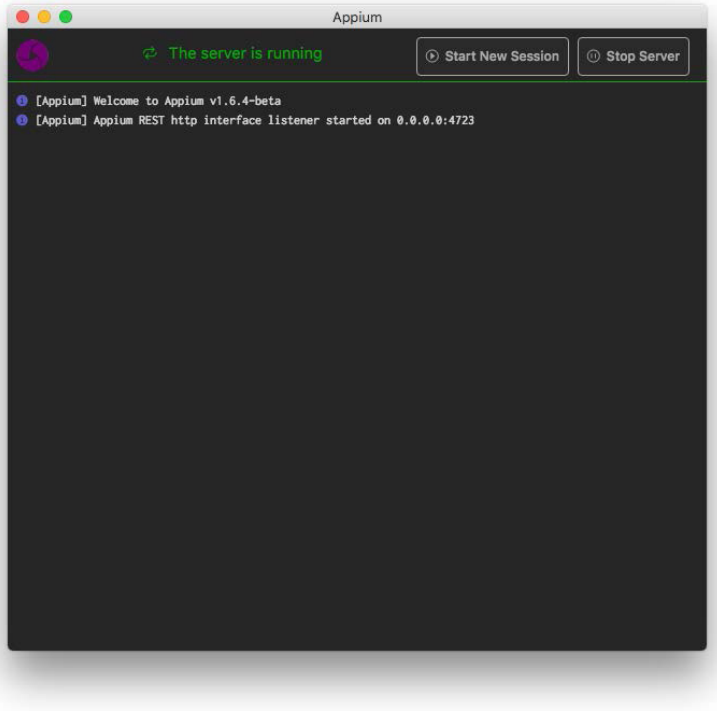
Appium From Appium Desktop

There is a very nice graphical front-end for Appium, maintained by the Appium Developers, called Appium Desktop. It bundles Appium along with a useful app inspector tool, so you can simply download Appium Desktop without worrying about any other system dependencies. You'll want Appium Desktop for this guide anyway, so go ahead and grab it from the releases page.

Once you've got it on your system and opened up, you should be greeted with a screen like this:



Now you can simply hit the "Start Server" button, and you'll see an Appium log window open up:



When we eventually begin running tests, this is where you'll see the output of what Appium is doing. Reading the Appium server logs can be a very useful way to understand what is happening under the hood! But for now, we just

want to make sure that everything is working. You can go ahead and stop the server if you want, or leave it running for later.

AN APPIUM CHECKUP

Once you've got Appium going, it's worthwhile to make sure your system is configured correctly before trying to run tests. There's a handy little tool called Appium Doctor that will do this for you:

```
npm install -g appium-doctor
appium-doctor
```

Running the `appium-doctor` command will output how well-configured Appium believes your system to be for the various platforms. If anything looks amiss, revisit the instructions above or seek help.

JAVA CLIENT SETUP

There is one Java package that we'll be using in this guide, hosted at Maven Central: io.appium/java-client. It contains a number of features, but primarily it's what we'll be using to speak the WebDriver protocol with the Appium server. We'll configure instructions for downloading and using the Java Client in the next section as part of our project setup, using Gradle.

PROJECT SETUP

For the rest of this guide we're going to be working on a Java project, starting from scratch. To get set up for the project, identify a path somewhere on your system that will be used for the project. It doesn't matter where it is. But in this guide, we're going to pretend it is `/path/to/project`, so anytime you see that path, just replace it with the one you're using. Next, initialize a new Java project in that directory, using Gradle as the build and dependency management tool. I'm using IntelliJ IDEA as my Java IDE while developing this guide, but it shouldn't matter which IDE you choose. (In general won't be spelling out instructions specific to the IDE, so find yourself a good guide to your IDE if you're new to it, for example setting up a Gradle project in IDEA). Give the project a group ID and artifact ID of your choosing (I will use `io.appium` and `bootcamp`, respectively).

At this point you should have a brand new Java project in `/path/to/project`, with some Gradle-related files (the `gradlew` wrapper, for example), and the typical Java file structure for our code (`src/main/java`, etc...).

Now, download a copy (one for each platform) of the test app we will use for this project. The app is called, *drumroll please*, The App and it's a silly little thing that will help us get going with automation. There's a version for both iOS and Android. Download each app from the v1.2.1 release page on GitHub. Put them in the `src/test/resources` directory (since we'll be using `test` as the place to put our test code, of course). At this point your project directory should look roughly like:

```
/path/to/project
├── build.gradle
├── gradle
├── gradlew
├── gradle.bat
├── settings.gradle
└── src
    ├── main
    │   ├── java
    │   └── resources
    └── test
        ├── java
        └── resources
            └── apps
                ├── TheApp-v1.2.1.apk
                └── TheApp-v1.2.1.app.zip
```

Next, we need to set up our project dependencies. Open up `build.gradle` and make sure the `dependencies` section looks like:

```
dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.12'
    testCompile group: 'io.appium', name: 'java-client', version: '6.0.0-BETA5'
    testCompile group: 'org.hamcrest', name: 'hamcrest-library', version: '1.3'
}
```

In other words, we're adding JUnit, the Appium Client, and Hamcrest (an assertion library) to our project, at the specified versions. Once you save the `build.gradle` file, you'll need to refresh the Gradle project (your IDE should helpfully suggest an easy way to do this).

We're now ready to begin! Head on over to the next chapter where we discuss how to open up your app and look for UI elements inside, to enable automation via Appium.

KNOWN WORKING VERSIONS

As an aside, it's worth mentioning what combination of software this book was written and tested with. If you want to guarantee that all the code samples will run without any modification, make sure to use this combination of tools and libraries:

Library	Version
Android	8.1, 7.1
Appium	1.8.0
Appium Desktop	1.6.0
Appium Java Client	6.0.0
iOS	11.2
Java JDK	1.8.0
Jenkins	2.89.4
JUnit	4.12
macOS	10.13.3
Sauce Java	2.1.23
The App	1.2.1
Xcode	9.3

Of course, I'll try to write code that won't go out of style too quickly. But Appium is a fast-moving project, so some Appium code might become outdated before too long, if you're keeping up with Appium server and driver releases. Always make sure you're reading the most recently published version of this guide.

Chapter 3

EXPLORING YOUR APP

Before you can test your app, you have to know how it's put together! You don't need to know the nitty-gritty of the app code, but you do need to know what the UI elements are that your test will operate on. In this chapter we're going to look at the recommended method of exploring the element hierarchy of our app, and figuring out how to find specific elements for use in testing. This method involves the visual inspector bundled with Appium Desktop.

Now, it's up to you to go explore the Android version of The App. All you need to do is fire up your AVD from Android Studio, and then run `arc` from the `android` subdirectory in your project. If all goes well, you'll be able to explore the Android version of this app in exactly the same way as we did for iOS.

USING THE APPIUM DESKTOP INSPECTOR

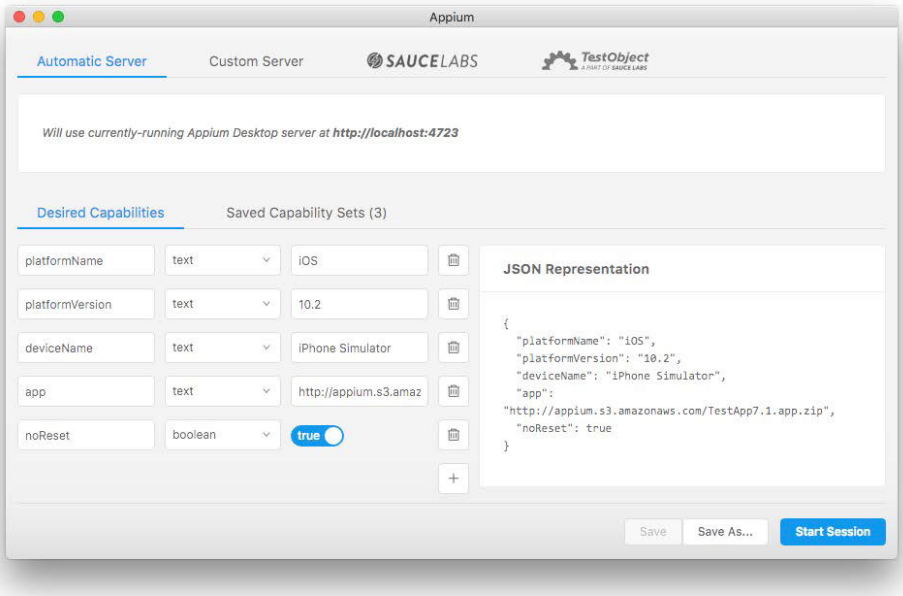
Load up Appium Desktop if you haven't already. Once it launches, simply click the big "Start Server" button to get Appium itself up and running. If you installed Appium via npm, and would prefer to use the command-line Appium server, you can also start it as you normally would (running `appium`).

You should now see Appium's welcome message, something like:

```
[Appium] Welcome to Appium v1.7.2
[Appium] Appium REST http interface listener started on 0.0.0.0:4723
```

This means Appium is alive and waiting for a new automation session to be requested. We're going to do that by tapping the magnifying glass icon ("Start Inspector Session") at the top of the log `window`[1]. Now a different window will pop up that will give you the ability to choose an Appium server, and enter desired capabilities for a new Appium session running on that server:

[1] If you started Appium outside of Appium Desktop, you can get to this session creation window by going up to the "Appium" menu and selecting "New Session", or hitting CMD+N). Either way, you'll be looking at this screen.



From Chapter 2, we already know that Appium is built on a client-server architecture, which means that our test sessions can speak to Appium servers anywhere on our machine or on the internet. The top part of the window here is for selecting from a variety of possible Appium server locations. For now, we'll just stick with the default, which is the server we have running through Appium desktop.[2]

In WebDriver-land, parameters used to start a session are known as *desired capabilities*, often abbreviated "caps". We're going to start by using 5 desired capabilities:

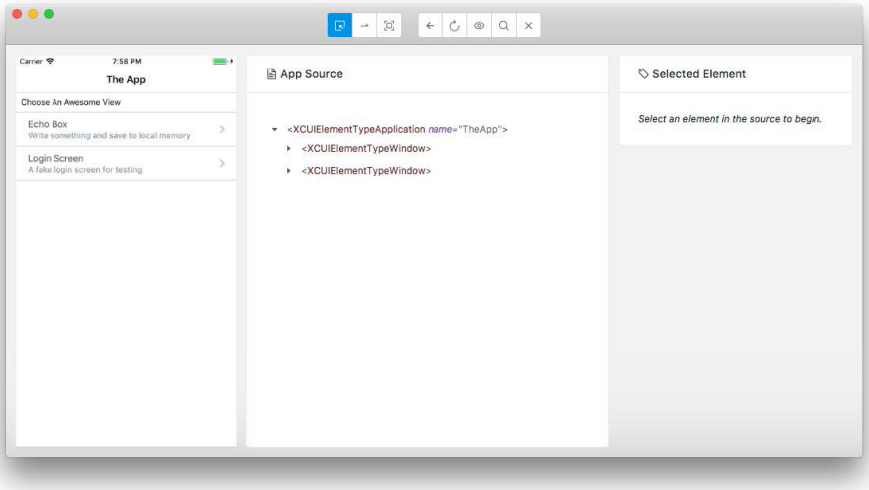
- `platformName`: which mobile platform we're running on ("iOS" or "Android")
- `platformVersion`: which mobile OS version (e.g., "11.1" for iOS)
- `deviceName`: the kind of device we wish to automate (e.g., "iPhone 8" or "Android Emulator")
- app: the path on your filesystem to the app you want to automate (e.g., the full path to the `.app.zip` or `.apk` you downloaded and put into the `resources` directory of your Java project
- `automationName`: which Appium driver to use (if different than the default for the platform specified). For iOS, this capability is not required since we'll use the default driver. For Android, set it to "UiAutomator2".

[2] If you're running Appium from the command line, here you'll need to choose "Custom Server", and then enter the host and port details, so that the Inspector can connect to the server you have running.

In the UI, use the lower portion to type the desired capabilities for iOS above into the boxes, creating new capability slots as you go. (The type for all of these capabilities is "text".) As you build the caps, you'll see a nice JSON representation so you can double-check your work. In fact, this JSON representation is ultimately what gets sent to the Appium server when you request a new session. So we'll be using these same desired capabilities when it comes time to write an Appium script in Java, too.

If you want, you can even save this set of caps so you can load it next time you launch Appium Desktop. When you're done, ensure again that the server selection panel up top has the correct server set (namely the "Automatic Server" if you're using the server that's built into Appium Desktop). Then, click "Start Session".

At this point Appium Desktop is starting a new session for you using the provided capabilities. If all goes well, the window will morph into the Inspector:



The Inspector consists of four sections:

1. A command bar with buttons that give access to different actions and features
2. A screenshot of the state of your app after the last command
3. The XML tree of your application hierarchy
4. A properties viewer that shows the details of a selected element, and lets you interact with a selected element

In the screenshot section, you can move your mouse over different elements and see them highlighted. If you click on one, it will open up in the property

viewer. Go ahead and click on the "Echo Box" list item. You'll see it focused in the source tree, as part of the XML structure. What is this structure? It is the UI hierarchy of your app, from the perspective of Appium, rendered as XML (so that it is similar to looking at the HTML source of a web page). Each node in the tree represents an element or container, and the tag name corresponds to the underlying UI element class name in either iOS or Android (for example, `XCUIElementTypeStaticText` corresponds to the class name of an iOS static text element).

Once you click on an element or node in the source tree, you'll also see a bunch of information retrieved for the element in the property viewer on the right. In the upper portion of the property viewer, there's a small table with recommended "locator strategies" and selectors for this element". The"locator strategy" is the method by which we could instruct Appium to find the element. We have to be specific here because there are a variety of strategies, and not all of them will find the elements we want. The "selector" is a string that describes the element in light of the chosen strategy.

There are a number of locator strategies:

Strategy	Description
`:accessibility_id`	Accessibility label
`:id`	Internal id
`:class_name`	UI class name
`:xpath`	XPath query based on source XML

When it comes time to code up our Appium script, we'll be using Java library methods that correspond to a strategy, and inputting the selector as a string, since it will be dependent on our particular app structure.

In the case of the "Echo Box" list item which we clicked on, Appium Desktop is recommending that we find this element by 'accessibility id', because it is the most reliable locator strategy to use for finding this element (even though we could also use 'xpath'). With the element selected, we can also run certain commands against it, to demonstrate the kind of actions available on element objects that Appium will return to our test script as well. By hitting 'Tap', for example, we see the appropriate action take place: the list item is tapped and we get to a new view. After this new view loads, the screenshot and source refresh so we can explore the current state. (If the app is slow and the refresh happens before new elements load, you can always hit the 'Refresh' button to update the Inspector).

Appium Desktop's Inspector is a powerful tool with many more features than we can go into in this guide, including the ability to record test actions and generate usable code. Please play with it and check out the Appium Desktop README for more information. What we've learned to do with it so far is still absolutely essential: how to navigate our app's hierarchy and figure out which selectors we should be using in our test code.

Before we end this tour of Appium Desktop, let's try one more feature: determining whether an element exists by inputting our own strategy and selector, rather than navigating around the screenshot or source tree. To do this, click the magnifying glass icon in the inspector ("Search for Element"). A window will appear that lets you choose a locator strategy and type in a selector. Assuming you're now in the Echo Box view of The App, choose the "Accessibility ID" locator strategy, and type in "messageSaveBtn" as the selector. When you click find, you'll see a results window with all elements that matched the criteria we selected. In my case, there are multiple elements with the same `messageSaveBtn` id. (This has to do with how React Native has built my app hierarchy). I can click on either of these and perform elements in the same modal window.

Now try again with an ID that certainly doesn't exist, like "foo". The dialog will happily tell us it couldn't find any elements. This technique is a useful way of using Appium Desktop to test whether certain elements can be found using certain locators, and is helpful especially when debugging tests where elements are not being found as expected.

Speaking of tests, it's high time we wrote one. Now that we're armed with the ability to figure out which elements are in our app and how to find them, we can move to our favorite IDE and write some Java code that actually perform a useful verification. If you want, go ahead and play around more with Appium Desktop, for example by switching the capabilities to the correct set for Android as described above, and inspecting the Android version of The App.

WRITING YOUR FIRST TEST

The test we're going to write in this chapter will exercise the (admittedly fake) "login" functionality of The App. It's worth thinking first about how we would test this manually before we begin to write code. Here's what we'd do to verify a successful login flow:

1. Open up the app
2. Tap the "Login Screen" item
3. Enter a valid username and password in the box
4. Tap the "Login" button
5. Verify that we can see something only logged-in users should see.

(Of course, to fully exercise the feature we'd also want a negative test: entering an invalid username or password and ensuring that we *cannot* see the logged-in area.)

Let's dive into the Java code that will test this flow. Let's compose our first test file, for iOS. Create a class called `IOSLoginTest` in the `src/test/java` directory of your project, so that you now have a file called `IOSLoginTest.java` in that directory, which looks like the following:

```
import io.appium.java_client.MobileBy;
import io.appium.java_client.ios.IOSDriver;
import java.io.File;
import java.net.MalformedURLException;
import java.net.URISyntaxException;
import java.net.URL;
import java.nio.file.Paths;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.remote.DesiredCapabilities;

public class IOSLoginTest {

    IOSDriver driver;

    @Before
    public void setUp() throws MalformedURLException, URISyntaxException {
```

```
        URL appiumUrl = new URL(“http://localhost:4723/wd/hub”);
        URL resource = getClass().getClassLoader().getResource(“apps/TheApp-
v1.2.1.app.zip”);
        File app = Paths.get(resource.toURI()).toFile();

        DesiredCapabilities caps = new DesiredCapabilities();
        caps.setCapability(“platformName”, “iOS”);
        caps.setCapability(“platformVersion”, “11.2”);
        caps.setCapability(“deviceName”, “iPhone 7”);
        caps.setCapability(“app”, app);

        driver = new IOSDriver(appiumUrl, caps);
    }

    @After
    public void tearDown() {
        try {
            driver.quit();
        } catch (Exception ignore) {}
    }

    @Test
    public void testLogin() {
        driver
            .findElement(MobileBy.AccessibilityId(“Login Screen”))
            .click();

        driver
            .findElement(By.xpath(“//XCUIElementTypeTextField[@
name=\”username\”]”))
            .sendKeys(“alice”);

        driver
            .findElement(By.xpath(“//XCUIElementTypeSecureTextField[@
name=\”password\”]”))
            .sendKeys(“mypassword”);

        driver
            .findElement(MobileBy.AccessibilityId(“loginBtn”))
            .click();

        driver.findElement(MobileBy.AccessibilityId(“You are logged in as
alice”));
    }
}
```

At this point, your directory tree should look like (omitting the Gradle stuff for the time being):

```
/path/to/project
└── src
    ├── main
    │   ├── java
    │   └── resources
    └── test
        ├── java
        │   └── IOSLoginTest.java
        └── resources
            └── apps
                ├── TheApp-v1.2.1.apk
                └── TheApp-v1.2.1.app.zip
```

You can *try* to run the test by using your IDE (typically right-clicking the test class will you give you an option), or by running the following from the command line in your project directory:

```
./gradlew cleanTest test --tests "IOSLoginTest.testLogin"
```

(This directs Gradle to run the test we just created). I italicized "try" before because, either way you do it, if you attempt to run this test, it might actually fail, with a message about an element not being found. Before we dig into why that failure would occur, let's break apart the test file and understand what we're attempting to do with it in the first place. And if the test passed, lucky you!

In general, the structure of the test class is:

```
// java imports up here
public class IOSLoginTest {

    IOSDriver driver;

    @Before
    public void setUp() {
        // set up the driver using desired capabilities
    }

    @After
    public void tearDown() {
        // clean up the session after each test
    }

    @Test
    public void testLogin() {
```

```
        // our actual test logic
    }
}
```

Essentially, we have a class with at least three methods

1. A method annotated with `@Before`, which will be run before each test in the class. This is where we put common test initialization code, for example starting an Appium session.

2. A method annotated with `@After`, which will be run after each test in the class. This is where we clean up after each test (in our case, quitting the session if it started, to make sure the Appium server is ready for our next session)

3. One or more methods annotated with `@Test`; these are the actual test cases, each of which will start life with whatever context we've set up in the `@Before` method.

Our `setUp` method has one job, despite everything that's going on, and that is to start the Appium session and create a reference to the session as an instance of an `AppiumDriver` class (i.e., `IOSDriver` or `AndroidDriver`):

```
URL appiumUrl = new URL("http://localhost:4723/wd/hub");
URL resource = getClass().getClassLoader().getResource("apps/TheApp-v1.2.1.app.zip");
File app = Paths.get(resource.toURI()).toFile();

DesiredCapabilities caps = new DesiredCapabilities();
caps.setCapability("platformName", "iOS");
caps.setCapability("platformVersion", "11.2");
caps.setCapability("deviceName", "iPhone 7");
caps.setCapability("app", app);

driver = new IOSDriver(appiumUrl, caps);
```

Basically we're telling the client where to find our running server, and where to find our app (namely in the resources location for our class). Of course we also have to set up the desired capability so Appium knows what manner of thing we want to automate.

The `tearDown` method is super simple: just close the session by calling `driver.quit()` (and wrap the whole thing in a `try` block because we don't care if the quit is successful or not; if it's not successful, it probably means we never started a session correctly, or there's nothing we can do otherwise in any case).

The meat is of course our test logic itself:

```
driver
    .findElement(MobileBy.AccessibilityId("Login Screen"))
    .click();

driver
    .findElement(By.xpath("//XCUIElementTypeTextField[@name=\"username\"]"))
    .sendKeys("alice");

driver
    .findElement(By.xpath("//XCUIElementTypeSecureTextField[@name=\"password\"]"))
    .sendKeys("mypassword");

driver
    .findElement(MobileBy.AccessibilityId("loginBtn"))
    .click();

driver.findElement(MobileBy.AccessibilityId("You are logged in as alice"));
```

There are 5 steps in our test logic here, each of which starts with a call on the `driver` instance to find an element that we want to interact with. The `findElement` method takes an instance of the `By` class as its parameter. The `By` class is a Java client encapsulation of the "locator strategy" concept we introduced in the previous chapter. Regular Selenium locator strategies are available on `By`, whereas Appium-specific strategies (like Accessibility ID) are available on `MobileBy`. We've already seen what the selectors are and how they can be used. In this test we're using two strategies, Accessibility ID and XPath, with the selectors we found using Appium Desktop.

Finally, on the result of the call to `findElement`, we're calling different API methods like `click` and `sendKeys`. These are methods available on instances of `WebElement`, which enable interaction with found elements. We're *not* calling a method on the last found element, however (the one that says "You are logged in as alice"). Why is that? Well, what we're really doing with this last `findElement` is making an assertion: that an element with the corresponding Accessibility ID actually exists and is present. It's an assertion because, if the element is *not* found, then an exception will be thrown and the test will fail. Thus it's doing its job just by being found, and we don't have to interact with it at all.

In fact, you may have encountered this "Element Not Found" exception already, if your test happened to fail earlier. Why would it have failed in this way? The answer is that Appium is sometimes too fast! (Or, too stupid—you

can look at it either way). When you tell Appium to find an element, it will try to find it right then and there, and throw an exception if it can't. The trouble is, sometimes your app is not quite ready for the element to be found. Maybe it's still loading, or transitioning from one view to another. So the element *would* be there, if only Appium waited a bit.

We could hard-code a "static wait" to make sure we don't run through our steps too early, but this would introduce an unnecessary minimum wait time before each step. Instead, we'll use something called a `WebDriverWait`. Check out the correct version of `IOSLoginTest.java`, this time using waits in order to make sure we don't fail before we've given it a good shot at finding all our elements:

```
import io.appium.java_client.MobileBy;
import io.appium.java_client.ios.IOSDriver;
import java.io.File;
import java.net.MalformedURLException;
import java.net.URISyntaxException;
import java.net.URL;
import java.nio.file.Paths;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

public class IOSLoginTest {

    IOSDriver driver;
    WebDriverWait wait;

    @Before
    public void setUp() throws MalformedURLException, URISyntaxException {
        URL appiumUrl = new URL("http://localhost:4723/wd/hub");
        URL resource = getClass().getClassLoader().getResource("apps/TheApp-
v1.2.1.app.zip");
        File app = Paths.get(resource.toURI()).toFile();

        DesiredCapabilities caps = new DesiredCapabilities();
        caps.setCapability("platformName", "iOS");
        caps.setCapability("platformVersion", "11.2");
        caps.setCapability("deviceName", "iPhone 7");
        caps.setCapability("app", app);

        driver = new IOSDriver(appiumUrl, caps);
```

```
        wait = new WebDriverWait(driver, 10);
    }

    @After
    public void tearDown() {
        try {
            driver.quit();
        } catch (Exception ignore) {}
    }

    private WebElement safeFind(By by) {
        return wait.until(ExpectedConditions.presenceOfElementLocated(by));
    }
    @Test
    public void testLogin() {
        safeFind(MobileBy.AccessibilityId("Login Screen"))
            .click();

        safeFind(By.xpath("//XCUIElementTypeTextField[@name=\"username\"]"))
            .sendKeys("alice");
        safeFind(By.xpath("//XCUIElementTypeSecureTextField[@name=\"password\"]"))
            .sendKeys("mypassword");

        safeFind(MobileBy.AccessibilityId("loginBtn"))
            .click();

        safeFind(MobileBy.AccessibilityId("You are logged in as alice"));
    }
}
```

What we've done here is introduced a `WebDriverWait` field called `wait`, which is configured with a timeout of 10 seconds. We've then constructed a library method `safeFind`, whose job is to apply the wait in concert with an `ExpectedCondition`, which checks for the presence of an element. Together, the wait and expected condition continually check for the presence of the element we want, up to 10 seconds. If it finds it, it returns the element for our use. If not, it will throw an exception. Ultimately, what this lets us do is replace calls to `driver.findElement` with calls to our own `safeFind` method, with the same result but the additional assurance that we will try to find the element periodically for up to 10 seconds before failing.

Assuming all this has been set up correctly, try to run the test again. This time you should see Appium navigating the views, filling in the username and password details, and ending the session. Awesome! This is a great start, but we can certainly do better with the code. We'll see how in the next chapter.

Chapter 5

INTRODUCTION TO PAGE OBJECTS

One issue with the test code we've written as it stands is that we are mixing information about our app (namely which elements can be found with which locators) and information about our test (which test steps constitute the flow we are trying to test). Another issue is that, as soon as we add a second login-related test, we'll begin duplicating our selector strings. Imagine if the app were to change its accessibility IDs—we'd have to go make a change in many different tests!

The common solution to these problems is to use something called the Page Object Model. A Page Object represents a view (for example our Login view), and exposes only high-level actions so that test code can deal in user-level behaviors rather mixing in low-level element finding logic. What do I mean by this? Imagine if the test method from `IOSLoginTest` looked like this:

```
public void testLogin() {
    final String username = "alice";
    final String password = "mypassword";

    mainView.navToLogin();
    loginView.login(username, password);
    String loggedInUsername = loggedInView.getLoggedInUsername();
    Assert.assertEquals(loggedInUsername, username);
}
```

This is a much more concise, readable, and maintainable bit of test code. It specifies everything we need to do, and nothing more than that, in high-level abstractions that are quasi-readable even to someone who's not a Java expert. "First, define the username and password we want to use to log in. Then, on the main view, navigate to the login view. On the login view, do the login with the supplied username and password. Finally, on the resulting logged-in view, check which username is displayed. Use that to verify it is the same as the user we logged in as." In this model, the responsibility of finding elements, waiting for them to appear, or getting data from them lies in the so-called "page objects", represented by the `mainView`, `loginView`, and `loggedInView` objects in the code above. Each of these objects are instances of individual classes, one per app view involved in our automation. (Since this technique was pioneered with web page automation, we have the term "page object"; in the mobile world we would more accurately call these classes "view objects"). What would one of these classes look like? Let's take a look at one that I've

called `MainView`, and added in a `page_objects` package directly beneath our test package as `MainView.java`:

```
package page_objects;

import io.appium.java_client.AppiumDriver;
import io.appium.java_client.pagefactory.AppiumFieldDecorator;
import io.appium.java_client.pagefactory.WithTimeout;
import io.appium.java_client.pagefactory.iOSFindBy;
import java.util.concurrent.TimeUnit;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.PageFactory;

public class MainView {
    @WithTimeout(time = 10, unit = TimeUnit.SECONDS)
    @iOSFindBy(accessibility = “Login Screen”)
    private WebElement navToLoginBtn;
    public MainView(AppiumDriver driver) {
        PageFactory.initElements(new AppiumFieldDecorator(driver), this);
    }

    public void navToLogin() {
        navToLoginBtn.click();
    }
}
```

Basically, we maintain the information about selectors in `WebElement` fields created as representations of the objects on the view we will want to work with. We use annotations like `@FindBy` or `@iOSFindBy` to denote the strategy and the selector finding a certain element. We can also use another annotation, `@WithTimeout`, to express how long we would like to wait for a particular element to appear. This is a much nicer and more compact way of doing what we were trying to do in the previous chapter with `WebDriverWait`. Once we have our fields annotated, we have only two more tasks in this file: first, to write up a constructor which takes an instance of `AppiumDriver` (so that our elements know which driver to use to find themselves), and initializes the `PageFactory` elements in our class. The final task is to expose user-level behaviors in methods on the object (in this case, the only thing we care about doing from the main view is navigating to the login view, so we simply expose a `navToLogin` method). Essentially, this `MainView` class encapsulates everything to do with that view, and only exposes as public the high-level methods a test might need. These means that all the plumbing is kept in this one place, making it much easier to maintain. Now, let’s look at

the implementation of the `LoginView` class (added alongside the other page object as `src/test/java/page_objects/LoginView.java`):

```
package page_objects;

import io.appium.java_client.AppiumDriver;
import io.appium.java_client.pagefactory.AppiumFieldDecorator;
import io.appium.java_client.pagefactory.WithTimeout;
import io.appium.java_client.pagefactory.iOSFindBy;
import java.util.concurrent.TimeUnit;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class LoginView {
    @WithTimeout(time = 10, unit = TimeUnit.SECONDS)
    @FindBy(xpath = "//XCUIElementTypeTextField[@name=\"username\"]")
    private WebElement usernameField;

    @FindBy(xpath = "//XCUIElementTypeSecureTextField[@name=\"password\"]")
    private WebElement passwordField;

    @iOSFindBy(accessibility = "loginBtn")
    private WebElement loginBtn;

    public LoginView(AppiumDriver driver) {
    PageFactory.initElements(new AppiumFieldDecorator(driver), this);
    }
    public void login(String username, String password) {
        usernameField.sendKeys(username);
        passwordField.sendKeys(password);
        loginBtn.click();
    }
}
```

This is a little meatier, but still nice and simple. We have information about finding our elements, and then the high-level `login` method exposed. And notice how even the implementation of the `login` method is clean and easy to read, because no selectors and strategies are mixed in with the login logic. Instead, the specifics of finding elements is left to the element annotations Let's round out our object models with the model for the logged-in user page:

```
package page_objects;

import io.appium.java_client.AppiumDriver;
import io.appium.java_client.pagefactory.AppiumFieldDecorator;
import io.appium.java_client.pagefactory.WithTimeout;
import io.appium.java_client.pagefactory.iOSFindBy;
import java.util.concurrent.TimeUnit;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.PageFactory;

public class LoggedInView {

    @WithTimeout(time = 5, unit = TimeUnit.SECONDS)
    @iOSFindBy(xpath = "//XCUIElementTypeOther[contains(@name, 'You are logged in
as')]")
    private WebElement loggedInMsg;

    public LoggedInView(AppiumDriver driver) {
        PageFactory.initElements(new AppiumFieldDecorator(driver), this);
    }

    public String getLoggedInUsername() {
        String text = loggedInMsg.getText();
        return text.replaceAll(".*You are logged in as ([^ ]+).*", "$1");
    }
}
```

What we want from this view is not to perform actions per se, but instead to provide a method that returns the name of the logged-in user which we want to use for verification. To achieve this, we have to do some clever string substitution on the text which is actually available to us from Appium's perspective of the app, but otherwise this object model is also very straightforward. Putting it all together, we simply need to pull these into our actual test code by instantiating these various classes using the `driver` instance which was there before, and of course to rewrite our actual test code to be like we said we wanted above. Happily, we can also be rid of all the `WebDriverWait` code which made things a bit too ugly for my taste before.

```
import io.appium.java_client.ios.IOSDriver;
import java.io.File;
import java.net.MalformedURLException;
import java.net.URISyntaxException;
import java.net.URL;
import java.nio.file.Paths;
import org.junit.After;
import org.junit.Assert;
```

```
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.remote.DesiredCapabilities;
import page_objects.LoggedInView;
import page_objects.LoginView;
import page_objects.MainView;

public class IOSLoginTest {

    private IOSDriver driver;
    private MainView mainView;
    private LoginView loginView;
    private LoggedInView loggedInView;

    @Before
    public void setUp() throws MalformedURLException, URISyntaxException {
        URL appiumUrl = new URL(“http://localhost:4723/wd/hub”);
        URL resource = getClass().getClassLoader().getResource(“apps/TheApp-
v1.2.1.app zip”);
        File app = Paths.get(resource.toURI()).toFile();

        DesiredCapabilities caps = new DesiredCapabilities();
        caps.setCapability(“platformName”, “iOS”);
        caps.setCapability(“platformVersion”, “11.2”);
        caps.setCapability(“deviceName”, “iPhone 7”);
        caps.setCapability(“app”, app);

        driver = new IOSDriver(appiumUrl, caps);

        mainView = new MainView(driver);
        loginView = new LoginView(driver);
        loggedInView = new LoggedInView(driver);
    }

    @After
    public void tearDown() {
        try {
            driver.quit();
        } catch (Exception ignore) {}
    }

    @Test
    public void testLogin() {
        final String username = “alice”;
        final String password = “mypassword”;

        mainView.navToLogin();
        loginView.login(username, password);
```

```
        String loggedInUsername = loggedInView.getLoggedInUsername();
        Assert.assertEquals(loggedInUsername, username);
    }
}
```

Despite the fact that we have some extra imports and setup boilerplate, our test code itself is now much more maintainable and easy to understand. Furthermore, in future tests we now have access to a growing library of high-level methods that we can reuse. Finally, if a selector changes for an app element, we have just one place to go to fix it without having to worry about where else it might be in our codebase.

Migrating to the Page Object Model was great. Go ahead and re-run the test to make sure it works despite the pretty large conceptual changes we've introduced. Our login test is now looking lean and clean. There's still some boilerplate in it, however, around setting up the driver.

Even though it's only a few lines, once we have more tests, we'll want to factor that out. Turn to the next chapter to see how this architectural change will help us by enabling the addition of Android tests in a cross-platform way.

Before we close this chapter, however, I'll include for reference how your directory structure should look after applying the changes we've made so far:

```
/path/to/project
└── src
    ├── main
    │   ├── java
    │   └── resources
    └── test
        ├── java
        │   ├── page_objects
        │   │   ├── LoggedInView.java
        │   │   ├── LoginView.java
        │   │   └── MainView.java
        │   └── IOSLoginTest.java
        └── resources
            └── apps
                ├── TheApp-v1.2.1.apk
                └── TheApp-v1.2.1.app.zip
```

ANDROID JOINS THE PARTY

Appium is a cross-platform automation tool, which means support for Android in addition to iOS. Also, The App is a cross-platform app, designed to work the same way on both iOS and Android. This makes it ideal for showcasing our ability to leverage code reuse with Appium. What we'd ideally like to see is complete test code reuse, such that the only difference between our iOS and Android testcases is the setup for each.

But first, let's do some refactoring to pave the way for this kind of cross-platform architecture. If all we were ever going to need was one test, our current setup would be just fine. But we're going to want many tests, spread across potentially many test files. If we just copy and paste the contents of our `IOSLoginTest.java` file into new files (say `IOSShoppingCartTest.java`), we would be duplicating a lot of boilerplate.

This gives us a clue that our test file has too many responsibilities. Right now, it not only knows about the test steps themselves, it also knows about platform information, Appium's server URL, paths to our apps, and desired capabilities. These are all extraneous from the perspective of the test itself, so let's factor them out into an abstract `BaseTest` class which takes these responsibilities, so actual testcase classes can be free of boilerplate. This is what `BaseTest.java` (at `src/test/java/BaseTest.java`) looks like:

```
import io.appium.java_client.AppiumDriver;
import io.appium.java_client.ios.IOSDriver;
import java.io.File;
import java.net.MalformedURLException;
import java.net.URISyntaxException;
import java.net.URL;
import java.nio.file.Paths;
import org.junit.After;
import org.junit.Before;
import org.openqa.selenium.remote.DesiredCapabilities;
import page_objects.LoggedInView;
import page_objects.LoginView;
import page_objects.MainView;

public abstract class BaseTest {

    private AppiumDriver driver;
```

```
    private final String appiumUrl = “http://localhost:4723/wd/hub”;

    protected MainView mainView;
    protected LoginView loginView;
    protected LoggedInView loggedInView;

    private File getAppFile(String app) throws URISyntaxException {
        URL resource = getClass()
            .getClassLoader()
            .getResource(“apps/” + app);
        return Paths
            .get(resource.toURI())
            .toFile();
    }

    @Before
    public void setUp() throws URISyntaxException, MalformedURLException {
        File app = getAppFile(“TheApp-v1.2.1.app.zip”);
        URL serverUrl = new URL(appiumUrl);

        DesiredCapabilities caps = new DesiredCapabilities();
        caps.setCapability(“app”, app);
        caps.setCapability(“platformName”, “iOS”);
        caps.setCapability(“platformVersion”, “11.2”);
        caps.setCapability(“deviceName”, “iPhone 7”);
        driver = new IOSDriver(serverUrl, caps);

        mainView = new MainView(driver);
        loginView = new LoginView(driver);
        loggedInView = new LoggedInView(driver);
    }

    @After
    public void tearDown() {
        try {
            driver.quit();
        } catch (Exception ignore) {}
    }
}
```

In addition to moving the setup and tear-down functionality into this class, I've also cleaned up the code a bit by creating a `getAppFile` method, which will make it easier to get apps for iOS and Android. With this `BaseTest` class, we can now rewrite our `IOSLoginTest` class to extend `BaseTest`, and do away with most of the code that was there previously:

```
import org.junit.Assert;
import org.junit.Test;

public class IOSLoginTest extends BaseTest {

    @Test
    public void testLogin() {
        final String username = “alice”;
        final String password = “mypassword”;

        mainView.navToLogin();
        loginView.login(username, password);
        String loggedInUsername = loggedInView.getLoggedInUsername();
        Assert.assertEquals(loggedInUsername, username);
    }
}
```

This file is now only 16 lines long, and it consists of just the test we care about. The test has access to the all the view objects (the “page objects” from the previous chapter) it might need to walk through its steps. Importantly, it now *only* has access to these objects; it can’t even access the `driver` object, which forces us into the clean use of the view models. Finally, you can see how easy it would be at this point to add a new testcase to this class, with zero boilerplate.

Go ahead and rerun the iOS login test, and prove that everything still works.

Now, we’re ready to add support for Android. Conceptually, there are three tasks we must complete to make it work, from the inside out:

1. The view models need to be updated with any Android-specific locators or logic. Each of their public methods should “just work” no matter which platform we’re on.

2. Since it’s the `BaseTest` class’s job to know about instantiating drivers based on platformspecific capabilities, it needs some branching logic so that it can set up the right environment for the right platform.

3. When we kick off our tests, we need some way to let our code know, from the outside, which platform we intend to run the test on, so that the appropriate drivers and capabilities are instantiated.

Let’s first tackle task #1. The Appium Java client comes not only with an `@iOSFindBy` annotation but also an @AndroidFindBy annotation. Anywhere that we use `@iOSFindBy`, we have simply to add an `@AndroidFindBy`

annotation with the correct selector. This is a trivial change, but in order to not repeat the use of constant string literals, I have also pulled out the selector string into its own field, for use with the annotations. Here's the `MainView` class:

```
package page_objects;

import io.appium.java_client.AppiumDriver;
import io.appium.java_client.pagefactory.AndroidFindBy;
import io.appium.java_client.pagefactory.AppiumFieldDecorator;
import io.appium.java_client.pagefactory.WithTimeout;
import io.appium.java_client.pagefactory.iOSFindBy;
import java.util.concurrent.TimeUnit;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.PageFactory;

public class MainView {
    private final String navToLoginBtnSel = “Login Screen”;
    @WithTimeout(time = 10, unit = TimeUnit.SECONDS)
    @iOSFindBy(accessibility = navToLoginBtnSel)
    @AndroidFindBy(accessibility = navToLoginBtnSel)
    private WebElement navToLoginBtn;

    public MainView(AppiumDriver driver) {
        PageFactory.initElements(new AppiumFieldDecorator(driver), this);
    }

    public void navToLogin() {
        navToLoginBtn.click();
    }
}
```

As you can see, all we've done is update the `navToLoginBtn` field with a new annotation.

Here, then, is our new `LoginView` class:

```
package page_objects;

import io.appium.java_client.AppiumDriver;
import io.appium.java_client.pagefactory.AndroidFindBy;
import io.appium.java_client.pagefactory.AppiumFieldDecorator;
import io.appium.java_client.pagefactory.WithTimeout;
import io.appium.java_client.pagefactory.iOSFindBy;
import java.util.concurrent.TimeUnit;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.PageFactory;
```

```
public class LoginView {
    private final String usernameFieldSel = “username”;
    @WithTimeout(time = 10, unit = TimeUnit.SECONDS)
    @iOSFindBy(accessibility = usernameFieldSel)
    @AndroidFindBy(accessibility = usernameFieldSel)
    private WebElement usernameField;

    private final String passwordFieldSel = “password”;
    @iOSFindBy(accessibility = passwordFieldSel)
    @AndroidFindBy(accessibility = passwordFieldSel)
    private WebElement passwordField;

    private final String loginBtnSel = “loginBtn”;
    @iOSFindBy(accessibility = loginBtnSel)
    @AndroidFindBy(accessibility = loginBtnSel)
    private WebElement loginBtn;

    public LoginView(AppiumDriver driver) {
        PageFactory.initElements(new AppiumFieldDecorator(driver), this);
    }

    public void login(String username, String password) {
        usernameField.sendKeys(username);
        passwordField.sendKeys(password);
        loginBtn.click();
    }
}
```

And finally the `LoggedInView` class:

```
package page_objects;

import io.appium.java_client.AppiumDriver;
import io.appium.java_client.pagefactory.AndroidFindBy;
import io.appium.java_client.pagefactory.AppiumFieldDecorator;
import io.appium.java_client.pagefactory.WithTimeout;
import io.appium.java_client.pagefactory.iOSFindBy;
import java.util.concurrent.TimeUnit;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.PageFactory;

public class MainView {
    private final String navToLoginBtnSel = “Login Screen”;
    @WithTimeout(time = 10, unit = TimeUnit.SECONDS)
    @iOSFindBy(accessibility = navToLoginBtnSel)
    @AndroidFindBy(accessibility = navToLoginBtnSel)
```

```
    private WebElement navToLoginBtn;

    public MainView(AppiumDriver driver) {
        PageFactory.initElements(new AppiumFieldDecorator(driver), this);
    }
    public void navToLogin() {
        navToLoginBtn.click();
    }
}
```

These updates take care of task #1 (making sure that our object models can work across both platforms). Luckily, all we had to do was update our selectors; no change in the actual logic was required to successfully implement our public methods. In a different app, we might have had to make changes there as well (say if it takes an extra button tap on one of the platforms).

Our strategy for #2 (teaching `BaseTest` about the different platforms) is going to involve a couple new classes. It's Java, after all! The first thing we want is an enum to represent the two platforms we're working with. I've added a `PlatformType` enum at `src/test/java/PlatformType.java`:

```
public enum PlatformType {
    ANDROID,
    IOS
}
```

It's nice and short, simply giving us a nice easy way to refer to these platforms throughout our code. The next thing we'll do is factor out the driver initialization code into a factory class, which knows how to generate the appropriate driver (iOS or Android), based on the platform type. This is in the same place as the previous file, and I've called it `DriverFactory.java`:

```
import io.appium.java_client.AppiumDriver;
import io.appium.java_client.android.AndroidDriver;
import io.appium.java_client.ios.IOSDriver;
import java.io.File;
import java.net.URL;
import org.openqa.selenium.remote.DesiredCapabilities;

public class DriverFactory {

    private PlatformType platformType;
    private File app;
    private URL serverUrl;
```

```
    public DriverFactory(PlatformType platformType, File app, URL serverUrl) {
        this.platformType = platformType;
        this.app = app;
        this.serverUrl = serverUrl;
    }

    public AppiumDriver getDriver() {
        DesiredCapabilities caps = new DesiredCapabilities();
        caps.setCapability("app", app);

        if (platformType == PlatformType.ANDROID) {
            caps.setCapability("platformName", "Android");
            caps.setCapability("deviceName", "Android Emulator");
            caps.setCapability("automationName", "UiAutomator2");
            return new AndroidDriver(serverUrl, caps);
        } else {
            caps.setCapability("platformName", "iOS");
            caps.setCapability("platformVersion", "11.2");
            caps.setCapability("deviceName", "iPhone 7");
            return new IOSDriver(serverUrl, caps);
        }
    }
}
```

Currently, we're sending in three bits of information from the outside in order to get an appropriate driver: the platform type, the app itself (since this factory could plausibly be used for any number of apps), and the location of the Appium server (since that is presumably variable based on external factors). With this information in hand, we simply return an instance of `IOSDriver` or `AndroidDriver` with appropriate capabilities set, based on the platform type which has been given. In the future, you could imagine extending the factory to also allow for different platform versions or device names.

Notice also that the Android capabilities differ slightly from the iOS ones, including the use of the `automationName` capability to specify that we want the (newer) UiAutomator2 driver available in Appium.

Finally (for task #2), we need to make use of our new factory by updating `BaseTest`, and also enabling `BaseTest` to pick up which platform tests should run on from some external state. In the case of this book, I've decided that we'll check for a system property called `platformType`, which could theoretically come in from a command line flag (like `-DplatformType`). Here, then, is the new `BaseTest.java`:

```
import io.appium.java_client.AppiumDriver;
import java.io.File;
import java.net.MalformedURLException;
import java.net.URISyntaxException;
import java.net.URL;
import java.nio.file.Paths;
import org.junit.After;
import org.junit.Before;
import page_objects.LoggedInView;
import page_objects.LoginView;
import page_objects.MainView;

public abstract class BaseTest {

    private AppiumDriver driver;
    private PlatformType platform;
    private final String appiumUrl = “http://localhost:4723/wd/hub”;

    protected MainView mainView;
    protected LoginView loginView;
    protected LoggedInView loggedInView;

    public BaseTest() {
        String cliArg = System.getProperty(“platformType”);
        if (cliArg == null) {
            // default to ios if no one set the platform type
            cliArg = “ios”;
        }
        if (cliArg.equalsIgnoreCase(PlatformType.ANDROID.name())) {
            platform = PlatformType.ANDROID;
        } else {
            platform = PlatformType.IOS;
        }
    }

    private File getAppFile(String app) throws URISyntaxException {
        URL resource = getClass()
            .getClassLoader()
            .getResource(“apps/” + app);
        return Paths
            .get(resource.toURI())
            .toFile();
    }

    @Before
    public void setUp() throws URISyntaxException, MalformedURLException {
        File app = getAppFile(“TheApp-v1.2.1.app.zip”);
        if (platform == PlatformType.ANDROID) {
```

```
            app = getAppFile("TheApp-v1.2.1.apk");
        }
        URL serverUrl = new URL(appiumUrl);
        DriverFactory driverFactory = new DriverFactory(platform, app,
serverUrl);
        driver = driverFactory.getDriver();

        mainView = new MainView(driver);
        loginView = new LoginView(driver);
        loggedInView = new LoggedInView(driver);
    }

    @After
    public void tearDown() {
        try {
            driver.quit();
        } catch (Exception ignore) {}
    }
}
```

As you can see, the only changes are:

1. We now use `System.getProperty` to check which platform we should run on (and fall back to iOS if nothing is provided).
2. We now select the Android version of The App in the case where we're running on Android.
3. We now utilize our new `DriverFactory` class to get our driver instance in a nice and clean way.

At this point I made one additional change, which is to rename `IOSLoginTest.java` to `LoginTest.java`, since it is now no longer iOS-specific (and renamed the class inside to `LoginTest`). Wait a second, don't we have to make any changes to the test file itself? The beauty of all the work we've done so far is that we don't! We already made sure the test was completely decoupled from anything remotely platform-specific, so we don't have to change it at all (apart from the name upgrade).

At this point, everything is ready to go. However, despite the fact that the code is set up for cross-platform testing, we have no real way to run an Android test. If you attempt to run `LoginTest.testLogin` from the IDE (by right-clicking on the test method, for example), it will work, but it will run the iOS version of the test. This is because we haven't set the system property `platformType` to anything, and so our platform selection logic is not getting triggered.

What we need to do is find some way to set the `platformType` system property explicitly, either to the string "`android`" or the string "`ios`". To do this, we're going to explore in the next chapter how to take advantage of the Gradle build system and finish up our cross-platformification of this little testsuite.

Before we close out this chapter, however, let's review what our project structure should look like at this point:

```
/path/to/project
└── src
    ├── main
    │   ├── java
    │   └── resources
    └── test
        ├── java
        │   ├── page_objects
        │   │   ├── LoggedInView.java
        │   │   ├── LoginView.java
        │   │   └── MainView.java
        │   ├── BaseTest.java
        │   ├── DriverFactory.java
        │   ├── LoginTest.java
        │   └── PlatformType.java
        └── resources
            └── apps
                ├── TheApp-v1.2.1.apk
                └── TheApp-v1.2.1.app.zip
```

RUNNING LOCAL TESTS WITH GRADLE

So far we've been running our tests just using our IDE directly (unless you intrepidly tried out the command-line options I gave a while back). But it's nice to know how to use a command-line task runner, as it's what we're going to ultimately need when we hook up our tests to CI later on. The task runner we've chosen is the same as our build tool, namely Gradle. We can edit the `build.gradle` file that we haven't touched since we added our initial project dependencies, and upgrade it to allow us to run our Android tests directly from the command line. What we need to do is add a set of new tasks to our project, one for running on iOS specifically and one for running on Android specifically. Each of these tasks should set the `platformType` system property to "ios" or "android", respectively. Let's take a look at what we want to add to `build.gradle`:

```
task testIOS(type: Test) {
    systemProperty "platformType", "ios"
}
task testAndroid(type: Test) {
    systemProperty "platformType", "android"
}
```

Essentially, we are defining two new tasks which inherit from the Test task. And all we have to do in each one is set the appropriate `platformType` via the Gradle `systemProperty` directive.

At this point you should reload your Gradle build via the IDE, to sync the tasks (and because the IDE will likely complain at you if you don't). If your IDE has nice Gradle integration, you might see the new tasks appear in the UI. We could choose to run them this way, but since the focus is on the command line, let's head to the terminal instead.

First, navigate to your project directory (`/path/to/project`). Next, use the Gradle wrapper to run the test using the new iOS task we created:

```
./gradlew testIOS
```

If all is set up correctly, you should see the exact same test running on your iOS simulator, and the command line output will let you know when it has completed successfully. Now, run exactly the same command again. What happens? Nothing! More specifically, the task completes instantly without

running the test. This is because Gradle knows it has just "built" that particular task and thinks it therefore doesn't need to do anything again, since nothing has changed in your code. We can get around this default behavior by always ensuring that we "clean" the task before running it, using a handy add-on task Gradle has given us for free:

```
./gradlew cleanTestIOS testIOS
```

We can happily run this command ad infinitum with the appropriate behavior.

Now let's finally attempt to run our cross-platform suite against an Android emulator. First, ensure you have an appropriate emulator booted up and running. We could have Appium start an emulator for us, but I like to have a bit more control over the emulator lifecycle when I'm running locally. Once it's ready, we can run:

```
./gradlew cleanTestAndroid testAndroid
```

Under the hood, this is setting the `platformType` system property to "android", and triggering the conditions in our code that deal with setting up an Android session via Appium. If all goes well, you'll see The App open up and run through the flow on Android, just as it did on iOS. Congratulations, you've automated a cross-platform app with a fully shared test codebase!

With knowledge of Gradle in hand, we'll move on to adding more task types and configuring our tests to run in the cloud.

Chapter 8

RUNNING TESTS IN THE SAUCE LABS CLOUD

There comes a time in the life of every testsuite when it becomes so large that it's impractical to run locally on your own machine, or requires platforms you no longer have easy access to. For these and a variety of other reasons, it's a good idea to invest in cloud Appium providers like Sauce Labs. `appium_lib` comes with built-in support for running tests in the Sauce cloud. To take advantage of this support, we'll need:

1. A Sauce Labs username and access key (if you don't already have one, you can start a free trial here, and find your access key at your dashboard after login)

2. Another Java dependency to help with the Sauce API, called `sauce_junit` (from the `com.saucelabs` group).

Let's update our `build.gradle` file with the new dependency, adding the line below to the `dependencies` block:

```
testCompile group: 'com.saucelabs', name: 'sauce_junit', version: '2.1.23'
```

Then, refresh your Gradle build so the dependency is added to the project and available in the code.

In order to make our test app accessible to the Sauce cloud, we have two options:

1. Host our app on the web somewhere and provide a url as the app capability

2. Use the `SauceREST` Sauce API client (bundled with `sauce_junit`) to upload our app to Sauce with the Sauce Storage API

Let's pursue option 2 here, since it is what you'd want to do in the context of a CI server. What we're going to do is upgrade our `BaseTest` to include Sauce-specific test setup and app uploading. Let's take a look at the new `BaseTest.java` file:

```
import com.saucelabs.saucerest.SauceREST;
import io.appium.java_client.AppiumDriver;
import java.io.File;
import java.io.IOException;
import java.net.URISyntaxException;
import java.net.URL;
import java.nio.file.Paths;
import org.junit.After;
import org.junit.Before;
```

```
import page_objects.LoggedInView;
import page_objects.LoginView;
import page_objects.MainView;

public abstract class BaseTest {
    private static String username = System.getenv(“SAUCE_USERNAME”);
    private static String accessKey = System.getenv(“SAUCE_ACCESS_KEY”);
    private static String localAppiumUrl = “http://localhost:4723/wd/hub”;
    private static String sauceServer = “@ondemand.saucelabs.com:80/wd/hub”;

    private AppiumDriver driver;
    private PlatformType platform;
    private ServerType server;

    protected MainView mainView;
    protected LoginView loginView;
    protected LoggedInView loggedInView;

    private SauceREST sauceAPI = new SauceREST(username, accessKey);

    public BaseTest() {
        setPlatformType();
        setServerType();
    }

    private void setPlatformType() {
        String platformArg = System.getProperty(“platformType”);
        if (platformArg == null) {
            // default to ios if no one set the platform type
            platformArg = “ios”;
        }
        if (platformArg.equalsIgnoreCase(PlatformType.ANDROID.name())) {
            platform = PlatformType.ANDROID;
        } else {
            platform = PlatformType.IOS;
        }
    }

    private void setServerType() {
        String serverArg = System.getProperty(“serverType”);
        if (serverArg == null) {
            // default to local if no one set the server type
            serverArg = “local”;
        }
        if (serverArg.equalsIgnoreCase(ServerType.SAUCE.name())) {
            if (username == null || accessKey == null) {
                System.out.println(“Username and access key were not set; running
locally”);
```

```
                server = ServerType.LOCAL;
                return;
            }
            server = ServerType.SAUCE;
        } else {
            server = ServerType.LOCAL;
        }
    }
    private File getApp(String app) throws URISyntaxException {
        URL resource = getClass()
            .getClassLoader()
            .getResource("apps/" + app);
        return Paths
            .get(resource.toURI())
            .toFile();
    }

    @Before
    public void setUp() throws URISyntaxException, IOException {
        File app = getApp("TheApp-v1.2.1.app.zip");
        if (platform == PlatformType.ANDROID) {
            app = getApp("TheApp-v1.2.1.apk");
        }
        String appStr = app.getAbsolutePath();
        URL serverUrl = new URL(localAppiumUrl);

        if (server == ServerType.SAUCE) {
            serverUrl = new URL("http://" + username + ":" + accessKey +
sauceServer);
            sauceAPI.uploadFile(app);
            appStr = "sauce-storage:" + app.getName();
        }
        DriverFactory driverFactory = new DriverFactory(platform, server, appStr,
serverUrl);
        driver = driverFactory.getDriver();

        mainView = new MainView(driver);
        loginView = new LoginView(driver);
        loggedInView = new LoggedInView(driver);
    }

    @After
    public void tearDown() {
        try {
            driver.quit();
        } catch (Exception ignore) {}
    }
}
```

There's a lot of stuff going on now! First of all, we've added some new static fields at the top of the class, which retrieve the Sauce Labs username and access key from the environment. This means that for running Sauce tests, we'll need to have the `SAUCE_USERNAME` and `SAUCE_ACCESS_KEY` environment variables set in our terminal. We can do this using the following commands:

```
export SAUCE_USERNAME="my_username"
export SAUCE_ACCESS_KEY="my_access_key"
```

This will store the variables for the duration of the current terminal session. If you want to store them permanently, it's a good idea to add the export commands to your `~/.bashrc`, `~/.bash_profile`, `~/.zshrc`, etc..., shell login file. That way you never have to remember them again.

Next, we've added a new field called server, of type `ServerType`. What is `ServerType`? It's a brand-new enum I just added at `src/test/java/ServerType.java`, with the following contents:

```
public enum ServerType {
    SAUCE,
    LOCAL
}
```

It does the same exact kind of work that `PlatformType` did for giving us the ability to distinguish between platforms, only now we're distinguishing between running tests locally and running them on Sauce. To use this new enum, I've factored out two setters, one for the old platform type logic (in `setPlatformType`), and a new one for setting the server type (`setServerType`). The only additional complexity to `setServerType` is that we double-check the username and access key are set appropriately; if not, we log a warning message and fall back to the local server.

The final change for `BaseTest` has to do with uploading our app to Sauce Storage. To enable this, we've added another field, namely an instance of `SauceREST`, which facilitates uploading apps to Sauce Storage. In our `setUp` method, we now simply check if we're running on Sauce versus local, and if we're running on Sauce, we call `uploadFile` on the API object, with the local file of the app we wish to upload. This takes care of sending it to the cloud and making it available for use in our Sauce tests. All that remains is to construct the correct server URL for Sauce (including the username and access key there, as well), and we're good to go.

You'll also notice that we've added an additional parameter to our `DriverFactory` instantiation, namely the type of server we're running on (Sauce vs local). This is because the capabilities we use do need to change in some small ways if we're running on Sauce. The new file should look like:

```
import io.appium.java_client.AppiumDriver;
import io.appium.java_client.android.AndroidDriver;
import io.appium.java_client.ios.IOSDriver;
import java.net.URL;
import org.openqa.selenium.remote.DesiredCapabilities;

public class DriverFactory {

    private PlatformType platformType;
    private ServerType serverType;
    private String app;
    private URL serverUrl;

    public DriverFactory(PlatformType platformType, ServerType serverType, String app,
            URL serverUrl) {
        this.platformType = platformType;
        this.app = app;
        this.serverUrl = serverUrl;
        this.serverType = serverType;
    }

    public AppiumDriver getDriver() {
        DesiredCapabilities caps = new DesiredCapabilities();
        caps.setCapability("app", app);

        if (serverType == ServerType.SAUCE) {
            caps.setCapability("appiumVersion", "1.7.2");
        }

        if (platformType == PlatformType.ANDROID) {
            String deviceName = "Android Emulator";
            caps.setCapability("platformName", "Android");
            if (serverType == ServerType.SAUCE) {
                deviceName = "Android GoogleAPI Emulator";
                caps.setCapability("platformVersion", "7.1");
            }
            caps.setCapability("deviceName", deviceName);
            caps.setCapability("automationName", "UiAutomator2");
            return new AndroidDriver(serverUrl, caps);
        } else {
            caps.setCapability("platformName", "iOS");
            caps.setCapability("platformVersion", "11.2");
            caps.setCapability("deviceName", "iPhone 7");
```

```
            return new IOSDriver(serverUrl, caps);
        }
    }
}
```

There are four key changes we made:

1. We made app a `String` rather than `File`, since the `sauce-storage:` app URL is not a File object.
2. We ensured we were passing in an `appiumVersion` capability, necessary on Sauce to specify which version of Appium we want to run on (at the time of writing, version 1.7.2 was the latest available on Sauce).
3. We added a `platformVersion` capability for Android, and set it to 7.1. We didn't have this before because I didn't want Appium to do a version check on the emulator; but for Sauce it's necessary, since Sauce needs to know which version of Android we care to test on.
4. We updated the `deviceName` capability for Android, again to follow Sauce conventions.

How do we know what capabilities we should use to select the appropriate Sauce platform? By using Sauce's Platform Configurator, a handy tool to walk you through getting the capabilities for the platforms you need.

These are all the changes we need to make our tests Sauce-aware. One step remains, however: how do we actually signify that we want to *run* the tests on Sauce instead of locally? One of the changes we made to `BaseTest` is the key: we now support the reading of a `serverType` system property to determine whether we should run on Sauce or locally (based on the contents of the string, whether "sauce" or "local"). And we already know how to use Gradle to create tasks with custom system properties. So let's update our `build.gradle` once again by adding the following Sauce-specific tasks (and updating the old tasks while we're at it):

```
task testIOS(type: Test) {
    systemProperty "platformType", "ios"
    systemProperty "serverType", "local"
}

task testAndroid(type: Test) {
    systemProperty "platformType", "android"
    systemProperty "serverType", "local"
}
```

```
task testIOS_Sauce(type: Test) {
    systemProperty “platformType”, “ios”
    systemProperty “serverType”, “sauce”
}

task testAndroid_Sauce(type: Test) {
    systemProperty “platformType”, “android”
    systemProperty “serverType”, “sauce”
}
```

When this change is imported, we can use Gradle to run, for example, the `testAndroid_Sauce` task, and the test will kick off on Sauce. Go ahead and run the Sauce versions:

```
./gradlew testAndroid_Sauce
./gradlew testIOS_Sauce
```

While you're running them, log onto the Sauce Labs website and you can see the tests running on your dashboard. If you click on a test, you will be greeted with a variety of details about it, including a stream of the running test (or a video if you catch it after it finishes).

You might notice that the name of the test (`unnamed test ....`) isn't very descriptive, and that we see a nasty gray question mark even though we know our test passed. Let's fix that in the next section. But before we move on, let's make one more upgrade to our `build.gradle` file, to fix a minor annoyance we encountered previously. Add these 3 lines to the bottom of the file:

```
[testIOS, testAndroid, testIOS_Sauce, testAndroid_Sauce].each {
    it.outputs.upToDateWhen {false}
}
```

What do these do? Basically, we loop through all our custom test tasks, and tell Gradle that none of them should ever be considered "up-to-date". This means we no longer have to worry about calling the various "clean" tasks in order to get these functional tests to re-run. A little change, but makes life a whole lot easier.

ANNOTATING TESTS ON SAUCE

By default, Sauce doesn't know what our test is called, or whether it passed or failed—it just knows what Appium commands we sent over, but that could mean anything! Luckily, JUnit knows these things, and we can use our helpful Sauce Java library methods to communicate this information back to Sauce, so we see prettier and more accurate results on our Sauce dashboard.

Let's take it one task at a time. First, in order to let Sauce know whether our test passed or failed, we will specify a JUnit `@Rule` inside of our `BaseTest` which is an instance of `SauceOnDemandTestWatcher`:

```
@Rule
public SauceOnDemandTestWatcher watcher = new SauceOnDemandTestWatcher(this, auth);
```

This sets up a hook on our tests which will get run when they finish (and pass or fail). What is `auth` here? It is defined just above:an instance of

```
private SauceOnDemandAuthentication auth = new SauceOnDemandAuthentication(username,
accessKey
```

This is just a helpful little container for our credentials which can be passed around. These two lines are enough to get the Sauce helper library to attempt to notify Sauce of the status of our completed test. It's not quite enough, however: Sauce needs to know the session ID of our test in order to update the status of the *correct* test. For this, we first need to declare that our class provides a session ID:

```
public abstract class BaseTest implements SauceOnDemandSessionIdProvider
```

Then, we need to create a `sessionId` field on our class, which is set once we have a driver object and therefore a session is in progress:

```
// just after the line 'driver = driverFactory.getDriver();'
sessionId = driver.getSessionId().toString();
```

Finally, we need to expose this field to the Sauce helpers, by adding a getter:

```
@Override
public String getSessionId() {
    return sessionId;
}
```

If you run the tests again with these changes, you'll notice that the test is getting appropriate marked as passed or failed in the Sauce UI. The name is still a bit of a problem though. How do we update that? The answer is with another JUnit `@Rule`, this time one to make the name of the test available to our `BaseTest` code:

```
@Rule
public TestName name = new TestName() {
    public String getMethodName() {
        return String.format("%s", super.getMethodName());
    }
};
```

This rule enables us to access the name field in the context of a test, and get that particular test's name. The way we use that information in conjunction with Sauce is to set the name desired capability before our test begins. That means we need to determine the test name inside our `setUp` routine, and pass the name to the `DriverFactory` which does all the capability management:

```
DriverFactory driverFactory = new DriverFactory(platform, server, appStr,
    serverUrl, name.getMethodName());
```

This in turn means we need to update `DriverFactory` to take a new String parameter, and then make use of this new parameter during capability setting (but only when we're running on Sauce):

```
if (serverType == ServerType.SAUCE) {
    caps.setCapability("appiumVersion", "1.7.2");
    caps.setCapability("name", testName);
}
```

There were a lot of ins and outs among all these changes, so let's once again have a look at the full files. First, `BaseTest.java`:

```
import com.saucelabs.common.SauceOnDemandAuthentication;
import com.saucelabs.common.SauceOnDemandSessionIdProvider;
import com.saucelabs.junit.SauceOnDemandTestWatcher;
import com.saucelabs.saucerest.SauceREST;
import io.appium.java_client.AppiumDriver;
import java.io.File;
import java.io.IOException;
import java.net.URISyntaxException;
import java.net.URL;
import java.nio.file.Paths;
import org.junit.After;
import org.junit.Before;
import org.junit.Rule;
import org.junit.rules.TestName;
import page_objects.LoggedInView;
import page_objects.LoginView;
import page_objects.MainView;

public abstract class BaseTest implements SauceOnDemandSessionIdProvider {
    private static String username = System.getenv("SAUCE_USERNAME");
    private static String accessKey = System.getenv("SAUCE_ACCESS_KEY");
    private static String localAppiumUrl = "http://localhost:4723/wd/hub";
    private static String sauceServer = "@ondemand.saucelabs.com:80/wd/hub";

    private AppiumDriver driver;
    private PlatformType platform;
```

```
    private ServerType server;
    private String sessionId;

    protected MainView mainView;
    protected LoginView loginView;
    protected LoggedInView loggedInView;

    private SauceREST sauceAPI = new SauceREST(username, accessKey);
    private SauceOnDemandAuthentication auth = new
SauceOnDemandAuthentication(username, accessKey);

    @Rule
    public SauceOnDemandTestWatcher watcher = new SauceOnDemandTestWatcher(this, auth);

    @Rule
    public TestName name = new TestName() {
        public String getMethodName() {
            return String.format("%s", super.getMethodName());
        }
    };
    public BaseTest() {
        setPlatformType();
        setServerType();
    }

    private void setPlatformType() {
        String platformArg = System.getProperty("platformType");
        if (platformArg == null) {
            // default to ios if no one set the platform type
            platformArg = "ios";
        }
        if (platformArg.equalsIgnoreCase(PlatformType.ANDROID.name())) {
            platform = PlatformType.ANDROID;
        } else {
            platform = PlatformType.IOS;
        }
    }

    private void setServerType() {
        String serverArg = System.getProperty("serverType");
        if (serverArg == null) {
            // default to local if no one set the server type
            serverArg = "local";
        }
        if (serverArg.equalsIgnoreCase(ServerType.SAUCE.name())) {
            if (username == null || accessKey == null) {
                System.out.println("Username and access key were not set; running
locally");
```

```
                server = ServerType.LOCAL;
                return;
            }
            server = ServerType.SAUCE;
        } else {
            server = ServerType.LOCAL;
        }
    }
    private File getApp(String app) throws URISyntaxException {
        URL resource = getClass()
            .getClassLoader()
            .getResource("apps/" + app);
        return Paths
            .get(resource.toURI())
            .toFile();
    }

    @Before
    public void setUp() throws URISyntaxException, IOException {
        File app = getApp("TheApp-v1.2.1.app.zip");
        if (platform == PlatformType.ANDROID) {
            app = getApp("TheApp-v1.2.1.apk");
        }
        String appStr = app.getAbsolutePath();
        URL serverUrl = new URL(localAppiumUrl);

        if (server == ServerType.SAUCE) {
            serverUrl = new URL("http://" + username + ":" + accessKey +
sauceServer);
            sauceAPI.uploadFile(app);
            appStr = "sauce-storage:" + app.getName();
        }

        DriverFactory driverFactory = new DriverFactory(platform, server, appStr,
            serverUrl, name.getMethodName());
        driver = driverFactory.getDriver();
        sessionId = driver.getSessionId().toString();
        mainView = new MainView(driver);
        loginView = new LoginView(driver);
        loggedInView = new LoggedInView(driver);
    }

    @After
    public void tearDown() {
        try {
            driver.quit();
        } catch (Exception ignore) {}
    }
```

```
    @Override
    public String getSessionId() {
        return sessionId;
    }
}
```

Then, `DriverFactory.java`:

```
import io.appium.java_client.AppiumDriver;
import io.appium.java_client.android.AndroidDriver;
import io.appium.java_client.ios.IOSDriver;
import java.net.URL;
import org.openqa.selenium.remote.DesiredCapabilities;

public class DriverFactory {

    private PlatformType platformType;
    private ServerType serverType;
    private String app;
    private URL serverUrl;
    private String testName;

    public DriverFactory(PlatformType platformType, ServerType serverType, String app,
            URL serverUrl, String testName) {
        this.platformType = platformType;
        this.app = app;
        this.serverUrl = serverUrl;
        this.serverType = serverType;
        this.testName = testName;
    }

    public AppiumDriver getDriver() {
        DesiredCapabilities caps = new DesiredCapabilities();
        caps.setCapability("app", app);

        if (serverType == ServerType.SAUCE) {
            caps.setCapability("appiumVersion", "1.7.2");
            caps.setCapability("name", testName);
        }
        if (platformType == PlatformType.ANDROID) {
            String deviceName = "Android Emulator";
            caps.setCapability("platformName", "Android");
            if (serverType == ServerType.SAUCE) {
                deviceName = "Android GoogleAPI Emulator";
                caps.setCapability("platformVersion", "7.1");
            }
            caps.setCapability("deviceName", deviceName);
            caps.setCapability("automationName", "UiAutomator2");
```

```
            return new AndroidDriver(serverUrl, caps);
        } else {
            caps.setCapability(“platformName”, “iOS”);
            caps.setCapability(“platformVersion”, “11.2”);
            caps.setCapability(“deviceName”, “iPhone 7”);
            return new IOSDriver(serverUrl, caps);
        }
    }
}
```

Now, when you run `./gradlew testIOS_Sauce` or `./gradlew testAndroid_Sauce`, you'll see a nice human-readable name show up in the Sauce dashboard, and you'll see a beautiful green checkmark when the test passes (or a correspondingly terrifying red X if it fails).

At this point, we've got a very robust and flexible setup that can work well for local development as well as extend to the cloud. The last step in our journey is to set our tests up to be run as part of a Continuous Integration server. Before we move on, let's have one last look at our project structure, which won't change from here on out:

```
/path/to/project
└── src
    ├── main
    │   ├── java
    │   └── resources
    │   test
    │   ├── java
    │   │   ├── page_objects
    │   │   │   ├── LoggedInView.java
    │   │   │   ├── LoginView.java
    │   │   │   └── MainView.java
    │   │   ├── BaseTest.java
    │   │   ├── DriverFactory.java
    │   │   ├── LoginTest.java
    │   │   ├── PlatformType.java
    │   │   └── ServerType.java
    │   └── resources
    │       └── apps
    │           ├── TheApp-v1.2.1.apk
    │           └── TheApp-v1.2.1.app.zip
    ├── build.gradle
    ├── gradlew
    └── settings.gradle
```

Chapter 9

AUTOMATING TEST RUNS WITH A CI SERVER

The point of automating your tests is to have total flexibility in how and when they are run. The ideal scenario is for your entire test suite to be run on every single code change, so that changes are gated on passing the entire test suite. Why allow code in to your app if it breaks something?

The way many teams work this out in practice is by using a "CI server". CI stands for "Continuous Integration", and refers to the process whereby new code is constantly added to the shippable version of your app. Typically developers use a branching version control system with one branch (`master` or `trunk`) always representing a known-working increment of the app. In other words, `master` is always primed for release. Developers work on their own forks or branches, and before their code is merged to `master`, it undergoes a battery of automated tests. This is where the CI server comes in: from an automated testing perspective, the CI server is responsible for figuring out when new code needs to be tested, testing that code, and then merging code which passes the tests into the main trunk. Of course, the CI server can do a whole lot more, for example building artifacts used in testing or for eventual release.

One popular open source CI server is Jenkins, and we'll set up a local version of Jenkins in this chapter, to see how easy it is to configure tests to run in CI.

SETTING UP JENKINS

Jenkins runs on any platform, and at some point you may want to run it in a Linux container or on a Linux host, but for now we will stick with macOS so that we can run Jenkins locally. There are a number of ways to install Jenkins, but we're going to stick with `homebrew` to make things easy:

```
brew install jenkins
```

You could follow `homebrew`'s instructions and set up Jenkins to run on server start, but since we're just playing around, let's instead run an instance of Jenkins right here from the command line:

```
jenkins
```

Jenkins will go through its startup routine and automatically launch itself on port 8080, so you can open up a browser and navigate to `http://localhost:8080`. Before you do that, take a look at the CLI output from

the Jenkins startup, and copy the admin password you will need to log in. Now head to your web browser and launch the URL.

At this point Jenkins will guide you through a little setup flow. First, paste in the admin password you copied from the command line, or cat it out using the terminal in order to set up the admin account. Then, bypass the "Customize Jenkins" wizard by clicking the close button at the top right. You might find it valuable to explore the ecosystem of Jenkins plugins after you're done with this guide.

CREATING AN ANDROID BUILD

Once you're all logged in, we're going to add a Project to house our Android tests:

1. Click "New Item" in the top left sidebar nav
2. For the item name, enter "Android Appium" (or some other clever moniker not suitable for publishing in a guide like this)
3. Click "Freestyle Project"
4. Click "OK"

The result will be a page with a host of options. This is where we would teach the Project how to read from our version control system, or when it should run itself (either via a remote trigger or on a schedule), etc... For now, let's just focus on the Build step itself:

5. Under "Build", click "Add Build Step"
6. Click "Execute shell"

This will open up a text input area where we can type commands as if we were running them from the terminal. Enter the following commands:

```
cd /path/to/project
./gradlew testAndroid
```

These are the same commands we would run if we were a new user getting started with running tests. Now let's run our Project. Head back to the main dashboard, and navigate to the Project we just created. On the left sidebar, click "Build Now". You'll see a little progress indicator pop up with a build number by it (#1). This is a "Job" representing an instance of building the Android Project you created. At the Job page, you'll see its status, and you

can follow along with what's happening by clicking on "Console Output" on the left sidebar. If all goes well, you'll be greeted by this output:

```
Started by user admin
Building in workspace /Users/user/.jenkins/workspace/Appium Android
[Appium Android Java] $ /bin/sh -xe /var/folders/gv/vdnhjfy96ps....
+ cd /path/to/project
+ ./gradlew testAndroid
:compileJava NO-SOURCE
:processResources NO-SOURCE
:classes UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources
:testClasses

:testAndroid

BUILD SUCCESSFUL in 22s
3 actionable tasks: 2 executed, 1 up-to-date
Finished: SUCCESS
```

This is the log of what happened during the job. As you can see, all we had to do was get into the right directory and run the Gradle command for our Android tests.

CREATING AN IOS BUILD

The procedure is exactly the same for running our iOS tests in Jenkins. Let's walk through it:

1. Go back to the main page and click "New Item"
2. Enter the name of the new Project (something more iOS-sounding, perhaps?)
3. This time, start typing the name of the Android project in the "Copy from" field, and select the existing project
4. Now, the shell commands are already pre-populated for us
5. Modify the shell commands, substituting `testIOS` for `testAndroid`
6. Save the Project, and click "Build Now" from the Project page
7. Verify that the iOS build completed successfully

Copying from an existing Project the way we did means we could rely on the existing shell commands and not have to start from a blank slate.

RUNNING ON SAUCE

Creating a project to run on Sauce is just as easy as running locally. To see how:

1. Follow the same steps as in the previous section to copy one of the existing Projects to a new Project called "Appium on Sauce"

2. In the shell command box, replace the existing `gradle` command with a new one: `./gradlew testAndroid_Sauce testIOS_Sauce` (this directs Gradle to run both Sauce tasks).

Now build this new Project and watch your tests run on Sauce!

For a real CI setup, you might consider checking out the Sauce Jenkins plugin which comes with a number of helpful features to make the experience of running on Sauce more integrated with your Jenkins server.

JENKINS FOR PRODUCTION

It's easy to see the power of something like Jenkins, once you imagine having these tests kicked off whenever a new commit comes in, or on an automated schedule of some sort. You have only to log back into your Jenkins server and see how your builds are doing, or explore the logs to debug any failures.

What we've done in this guide is run Jenkins as a toy under our own user. This would not be advisable in production. Instead, you'd want to follow one of many guides online (like this one) about setting Jenkins up for the appropriate host type in a secure and reliable fashion.

We also took advantage of the fact that Jenkins was running under our system user in order to keep our shell commands simple. In a real production setup, we'd have to worry about setting environment variables correctly (remember how we set `$ANDROID_HOME`, and all the rest? That needs to be done for Jenkins too). We'd also have to worry about emulator management for Android (for example, creating and tearing down emulators to ensure complete data isolation between builds), dependency management (ensuring Android and Xcode stay updated on build machines), etc... It's a lot! But getting CI set up for your team is always worth the effort.

HEADING OUT ON YOUR OWN

Here ends this guided tour of Appium. What have we covered?

1. What Appium is, and why we would use it
2. How to set up Appium for iOS and Android testing on macOS
3. How to interrogate iOS and Android apps using the Appium Desktop Inspector
4. How to create a simple Gradle-based test project in Java using the Appium Java Client
5. How to refactor and organize test code, leveraging Page Objects to make it fully crossplatform
6. How to use gradle to set up tasks for convenient test running
7. How to run tests in the cloud on Sauce Labs, leveraging different aspects of the Sauce API
8. How to set up a CI server we can use to run builds involving our Appium test suite

I hope you found it valuable, and I'll leave you with a few resources for further engagement with Appium.

RESOURCES

- The Appium Discussion Group is a great place to go and ask other Appium users for help
- The Appium Documentation has a set of guides, and a complete list of client commands, so it's a great place to go to figure out everything you can do with Appium
- Appium's GitHub Page contains links to the source code for the (many) Appium packages that make up the Appium server and clients
- Appium's Issue Tracker is where to go if you think you've found a bug and want to report it
- There's a handy reference card for Appium setup and commands

SUPPORT

Appium is community-supported via the discussion forums and GitHub. Sauce Labs customers can receive support for Sauce-related Appium issues via their account manager. And of course, there are a number of experienced Appium consultants out there who would be happy to offer paid support for tough issues.

Thanks for reading, and Happy Testing!

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