GETTING STARTED WITH APPIUM

By Jonathan Lipps

Ruby Edition 2018.1

TABLE OF CONTENTS

3 The Preface

4 Ch. 1: Introduction

4 The Appium Vision

5 The Appium Drivers

5 The Appium Clients

7 Ch. 2: Getting Set Up

7 Assumed Knowledge

7 iOS-specific System Setup

7 Android-specific System Setup

8 Appium Setup

8 Appium From the Command Line

8 Appium From Appium Desktop

10 Ruby Client Setup

10 Project Setup

11 Known Working Versions

12 Ch. 3: Exploring Your App

12 Using the Appium Ruby Console

17 Using the Appium Desktop Inspector

20 Ch. 4: Writing Your First Test

25 Ch. 5: Introduction to Page Objects

30 Ch. 6: Android Joins the Party

35 Ch. 7: Running Local Tests with Rake

36 Ch. 8: Running Tests in the Sauce Labs Cloud

40 Annotating Tests on Sauce

41 Ch. 9: Automating Test Runs with a CI Server

41 Setting up Jenkins

42 Creating an Android Build

44 Creating an iOS Build

44 Running on Sauce

44 Jenkins for Production

46 Heading Out on Your Own

46 Resources

47 Support

PREFACE

This little e-book will help you get started with Appium using the Ruby programming language. It is a complete revision from the ground up of an earlier guide written by Dave Haeffner, who along with Matthew Edwards deserves a lot of thanks for all the work put into the first edition.

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 @KazuCocoa, the current maintainer of the Appium Ruby libraries.

Jonathan Lipps
February 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 which 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 Ruby client, which is a rubygem named `appium_lib`.

The Appium Ruby client is not a standalone library: it is actually a wrapper around the standard Selenium Ruby 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 Ruby programming language well enough to follow along with simple examples. If either 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:

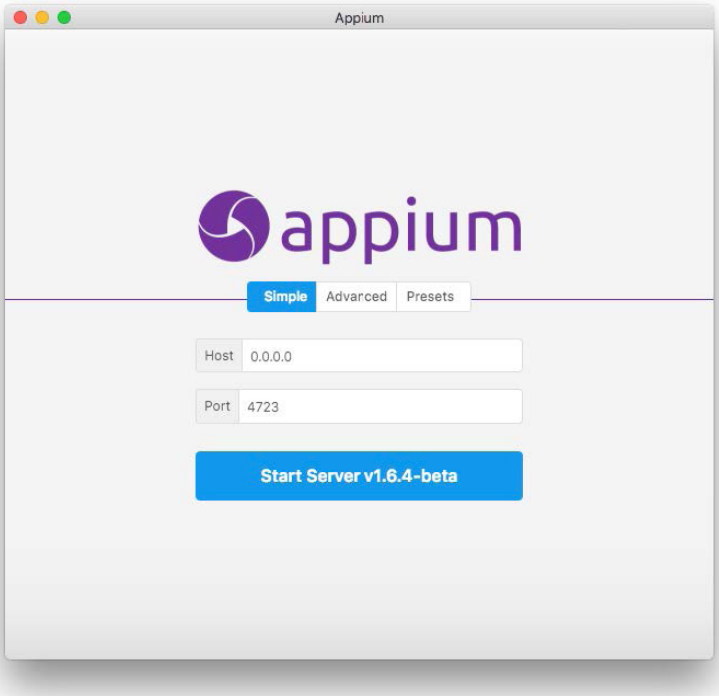
```
npm install -g appium
```

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

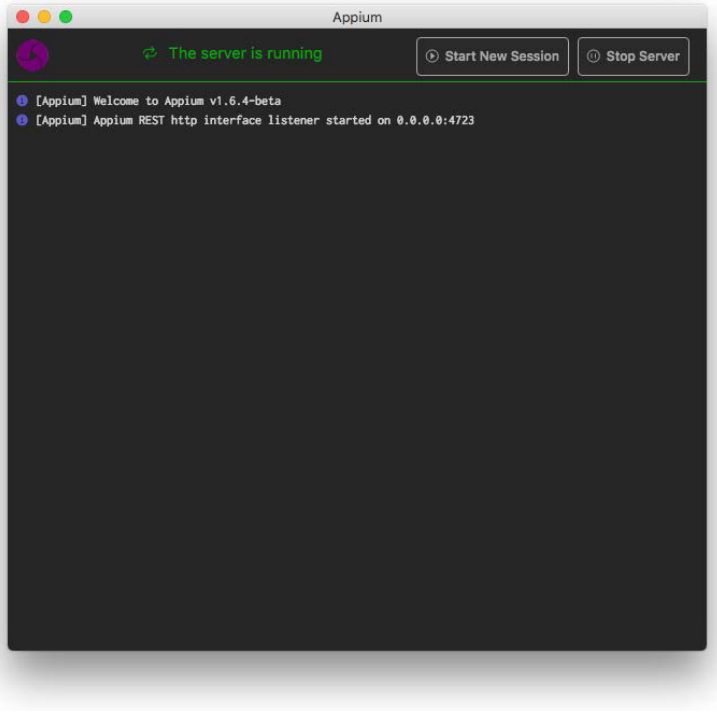
Appium From Appium Desktop

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.

RUBY CLIENT SETUP

There are two Ruby libraries we'll be using in this guide. The first is `appium_lib`, the Appium Ruby client. It's what we'll be using to speak the WebDriver protocol with the Appium server. The second library is `appium_console`, which is a command-line REPL we can use to experiment with the Ruby client in an easy fashion, outside the context of running an actual test. To get everything going:

- Get some system dependencies set up:
  - `brew install autoconf`
  - `brew install automake`
  - `brew install libtool`
- Make sure rubygems are up to date:
  - `gem update --system`
  - `gem update bundler`
- Now install the latest version of `appium_lib` and `appium_console`:
  - `gem uninstall -aIx appium_lib`
  - `gem uninstall -aIx appium_console`
  - `gem install --no-rdoc --no-ri appium_lib`
  - `gem install --no-rdoc --no-ri appium_console bond`

To be sure everything was installed, run `gem list | grep appium`. You should see output like the following:

```
appium_console (2.8.1)
appium_lib (9.10.0)
appium_lib_core (1.3.2)
```

PROJECT SETUP

For the rest of this guide we're going to be working on a Ruby project, starting from scratch. To get set up for the project, create a new directory somewhere

on your system. 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.

First, create two subdirectories, one called `ios` and one called `android`. Then, download a copy of the test app we will use for this project. The test app is called 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 the iOS app (the one ending with `.app.zip`) in the `ios` dir, and the Android app (the one ending with `.apk`) in the `android` dir. At this point your project directory should look like:

```
/path/to/project
├── android
│   └── TheApp-v1.2.1.apk
└── ios
    └── TheApp-v1.2.1.app.zip
```

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.

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	1.7.2
Appium Ruby Client	9.10.0
iOS	11.1
Java JDK	1.8.0
Jenkins	2.89.4
macOS	10.13.3
Ruby	2.4.1
The App	1.2.1

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 two ways of exploring the element hierarchy of our app, and figuring out how to find specific elements for use in testing. The first way is via the command-line Appium console, and the second is via the visual inspector bundled with Appium Desktop. (Be sure to read through both sections even if you only care about one, since I explain more about how Appium works along the way).

USING THE APPIUM RUBY CONSOLE

In the last chapter you installed a Rubygem called `appium_console`, which is a command-line REPL for using the Appium Ruby client to run sessions and explore apps. Let's get it going. The first thing to do is spin up the Appium server, so go ahead and click the "Start Server" button in Appium Desktop (with the default host and port), or run appium from the command line (depending on how you installed Appium in the previous chapter).

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 with the `appium_console` gem, which has installed an executable on your system called `arc` (**a**ppium **r**uby **c**onsole). In order to start a session using `arc`, we have to create a configuration file to encapsulate the parameters we will use to start the session. In WebDriver-land, these parameters 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

- `automationName`: which Appium driver to use (if different than the default for the platform specified)

In order to get arc to use these caps to start an Appium session, we create a file for it to read, called `appium.txt`. We're going to need two of these files: one for our iOS app and one for our Android app. So fill out an `appium.txt` file with the following contents, in the appropriate project directory.

In `/path/to/project/ios/appium.txt`:

```
[caps]
platformName = “iOS”
platformVersion = “11.2”
deviceName = “iPhone 7”
app = “/path/to/project/ios/TheApp-v1.2.1.app.zip”

[appium_lib]
sauce_username = false
```

In `/path/to/project/android/appium.txt`:

```
[caps]
platformName = “Android”
deviceName = “Android Emulator”
app = “/path/to/project/android/TheApp-v1.2.1.apk”
automationName = “UiAutomator2”

[appium_lib]
sauce_username = false
```

The only non-obvious differences between these two sets of caps are (1) in Android, we don't need to specify the `platformVersion`, since Appium will just use the running AVD we have created, and (2) in Android, we make sure to use the newer Android driver (namely `appium-uiautomator2-driver`).

Notice also that we have an `[appium_lib]` section with `sauce_username = false` in it. This just ensures that we are running tests using our local Appium server and not on Sauce. If you're already a Sauce user reading this guide, you might have certain environment variables set on your system that would otherwise trigger the Ruby client to try to run the test on Sauce. Not so fast, Ruby client!

Anyway, at this point, your project structure should look like:

```
/path/to/project
├── android
│   ├── appium.txt
│   └── TheApp-v1.2.1.apk
└── ios
    ├── appium.txt
    └── TheApp-v1.2.1.app.zip
```

To learn about `arc`, it doesn't matter which platform we begin with, so let's start with iOS. Navigate to the project's `ios` folder in your terminal, and then fire up `arc`:

```
cd /path/to/project/ios
arc
```

At this point, arc will attempt to start an iOS automation session on the running Appium server, using the caps you provided in the `appium.txt` file. If all goes you well, you will see a bunch of debug text filling the Appium server log window, letting you know that Appium is working on starting your session. If that process is successful, eventually an iOS simulator will pop up and load our test app, and wait for further instruction from you. If this does not happen, there will be error output from `arc`. Read it carefully, as it will likely contain a clue as to what went wrong (maybe the path to the application was not correct, for example). If it doesn't, move over to the Appium logs and read up from the bottom; likely there will be an error message that could contain more information. (If you end up stuck, head to the Appium forums and ask for help. You won't be able to continue this guide until you've got a working session!)

Once the session has started, `arc` will show you a prompt that allows you to start typing:

```
[1] pry(main)>
```

At this prompt you can use API methods from the Ruby client in order to find elements or examine the source tree of your application. What sorts of elements are there? Let's run the `page_class` command to find out:

```
pry(main)> page_class
17x XCUIElementTypeOther
2x XCUIElementTypeWindow
1x XCUIElementTypeStatusBar
1x XCUIElementTypeStaticText
1x XCUIElementTypeNavigationBar
1x XCUIElementTypeApplication
```

According to iOS, we've got an Application, a NavigationBar, a few other single-class elements, and a whole load of `XCUIElementTypeOther`. These are the elements that Apple's XCUITest library (which Appium is running under the hood) knows about. We can drill down into some details of these elements with the page command:

```
pry(main)> page :XCUIElementTypeOther
XCUIElementTypeOther
  visible: true
XCUIElementTypeOther
  name, label: The App
  visible: true
XCUIElementTypeOther
  visible: true
XCUIElementTypeOther
  isible: true
XCUIElementTypeOther
  visible: true
XCUIElementTypeOther
  name, label: Choose An Awesome View Echo Box Login Screen
  visible: true
```

(Note that we put a `:` in front of `XCUIElementTypeOther`, because we want it to be a Ruby symbol). Basically, this command walks through all of the Other elements and gives us some extra information about them, including a name. This name is important, because we can use it to find the element in our test, assuming it's unique. How do you find elements? By using the `find_element()` command! `find_element()` is an API method that takes two parameters: a "locator strategy" and a "selector". The strategy is the method by which we are instructing 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

Let's try out the first strategy, `:accessiblity_id`, which is always the recommended strategy since accessibility IDs can be added to elements by the app developer across platforms, so they're a good choice for a cross-platform locator. From the output of the page command earlier, we know that there is an element with the name `Echo Box`. On iOS, the accessibility ID is shown as the name attribute, so that's what we use:

```
pry(main)> find_element(:accessibility_id, “Echo Box”)
#<Selenium::WebDriver...>
```

The `#<Selenium::WebDriver...>` response is the representation of an object element. This means we found what we were looking for! Well, we found something. How can we be sure it's the element we want? We could query the element itself to see what text it's showing in the app:

```
pry(main)> find_element(:accessibility_id, “Echo Box”).text
“Echo Box”
```

Good. The fact that the `.text` command came back with "`Echo Box`" meant we found the element we were looking for. If we wanted, we could also perform an action on this list item, maybe tapping on it:

```
pry(main)> find_element(:accessibility_id, “Echo Box”).click
“”
```

If you watch the simulator while running this command, you'll see the view change as a result of the list item being tapped. We get an empty response from `arc` because the `click` action doesn't return any value; it simply does its thing and gives back control. Let's finish out this section by closing the session (don't want to leave it hanging around blocking future sessions or hogging resources). We can simply type `x` at the `arc` prompt to achieve this.

```
pry(main)> x
Closing appium session...
```

Believe it or not, this is basically everything you need to know about how to write an Appium test. Every Appium test consists of the following components:

1. Starting a session using desired capabilities
2. Finding elements
3. Performing actions on elements (including querying their state)
4. Ending the session

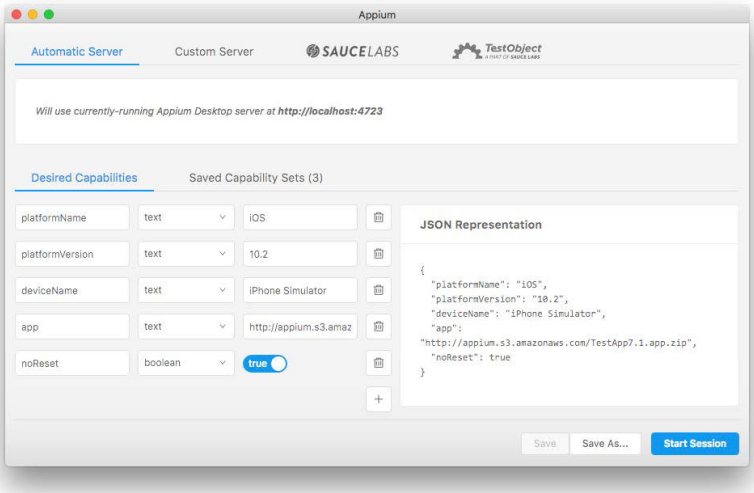
Steps 2 and 3 simply repeat as many times as necessary to walk through your app and generate the desired user behavior, making verifications in your test code along the way. Of course, there's a lot more to learn in terms of library methods available to you. But this is the basic pattern. The rest is details!

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

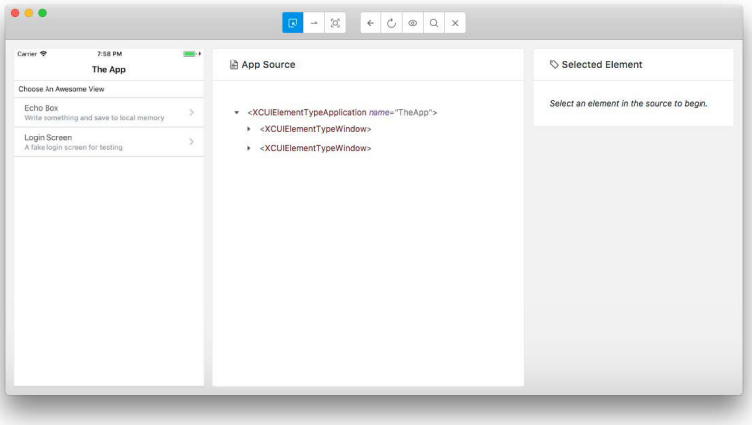
`arc` is pretty cool, and essential for playing with the Ruby-specific client commands to see how they work, or try something out without having to write a whole test. But for inspecting your app, finding elements, and a host of other reasons, Appium Desktop is the tool of choice.

If you've already got Appium Desktop's server running, simply tap the magnifying glass icon ("Start Inspector Session"), and a new window will pop up that will give you the ability to enter desired capabilities for a new Appium session:



Using this UI, port over the desired capabilities from the iOS `appium.txt`. As you build the caps, you'll see a nice JSON representation so you can double-check your work. 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 that the server selection panel has the correct server set (probably the "Automatic Server" if you're running from 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, and you'll also see a bunch of information retrieved for the element in the property viewer. In the upper portion of the property viewer, there's even a small table with recommended locator strategies and selectors for this element. In this case, Appium Desktop is recommending that we find this element by 'accessibility id', using the same selector we used within `arc`. Just like in 'arc', we can also run commands

on this element. By hitting 'Tap', we see the same action as before: 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.

Speaking of test code, it's high time we wrote some. 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 editor and write some Ruby scripts that actually perform a useful verification.

Chapter 4

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 Ruby code that will test this flow. First of all, because we want to assume we may be working on this test code as a team at some point in the future, we want to ensure that all the versions of our dependencies are set. We wouldn't want a teammate to try to run the test with a different set of gems, since it could lead to hard-to-debug errors. The way to achieve consistency here is to use the `bundler gem`. Install it if you don't have it already:

```
gem install bundler
```

Then, create a file called `Gemfile` in the root of our project directory. The `Gemfile` will hold the description of the gems and their versions:

```
source 'https://rubygems.org'
gem 'rspec', '~> 3.7.0'
gem 'appium_lib', '~> 9.10.0'
gem 'appium_console', '~> 2.8.1'
```

With this file in place we can run the `bundle` command in the project root directory:

```
bundle install
```

This will ensure we have all the gems installed at the appropriate versions, including the new `rspec` gem. RSpec is the test framework we will be using to develop our test suite. Now let's compose our first test file, for iOS. Create a file called `login_spec.rb` and add it to the `ios` directory:

```
require 'appium_lib'
describe 'Login' do
  before(:each) do
    appium_txt = File.join(Dir.pwd, 'appium.txt')
    caps = Appium.load_appium_txt(file: appium_txt)
    @driver = Appium::Driver.new(caps)
    @driver.start_driver
  end
  after(:each) do
    @driver.driver_quit
  end
  it 'Login with valid credentials' do
    @driver.wait {
      @driver.find_element(:accessibility_id, "Login Screen")
    }.click
    username = @driver.wait {
      @driver.find_element(:xpath,
        "//XCUIElementTypeTextField[@name=\"username\"]")
    }
    password = @driver.find_element(:xpath,
        "//XCUIElementTypeSecureTextField[@name=\"password\"]")
    username.send_keys("alice")
    password.send_keys("mypassword")
    @driver.find_element(:accessibility_id, "loginBtn").click
    @driver.wait {
      @driver.find_element(:accessibility_id,
        "You are logged in as alice")
    }
  end
end
```

At this point, your directory tree should look like:

```
/path/to/project
├── android
│   ├── appium.txt
│   └── TheApp-v1.2.1.apk
├── Gemfile
├── Gemfile.lock
└── ios
    ├── appium.txt
    ├── login_spec.rb
    └── TheApp-v1.2.1.app.zip
```

You can run the test by navigating to the `ios` directory in your terminal (if you're not still there), and running:

```
rspec login_spec.rb
```

Assuming all has been set up correctly, you'll see Appium navigating the views, filling in the username and password details, and ending the session. Now let's pick apart the code to see how it all worked! First, we make sure to require our Appium library, and set up the skeleton for our test using the RSpec structure:

```
require 'appium_lib'
describe 'Login' do
  before(:each) do
    ...
  end
  after(:each) do
    ...
  end
  it ... do
    ...
  end
end
```

What this structure indicates is that we have one or more tests belonging to a 'Login' category. We want RSpec to run some code before and after every single test, which we put into the `before(:each)` and `after(:each)` blocks. Finally, we specify the test itself using the `it` block. The way we're using this structure is to set up our Appium session in the `before` block, use the session for the test, and then tear it down in the `after` block. When we have multiple tests, we are going to be doing this for each test, so that every test has its own completely fresh Appium session.

How do we start an Appium session? Let's take a look:

```
appium_txt = File.join(Dir.pwd, 'appium.txt')
caps = Appium.load_appium_txt file: appium_txt
@driver = Appium::Driver.new(caps)
@driver.start_driver
```

We're taking advantage of the fact that we already have our desired capabilities set in the `appium.txt` file, and simply loading the caps from the file using the `Appium.load_appium_txt` helper utility. Then, we're storing a new `Appium::Driver` based on these caps into a Ruby instance variable, and starting the session with it. We're making an instance variable here because we want the driver to be accessible from other parts of our script (the test block itself, most importantly).

The way we end an Appium session is dead simple: just call `driver_quit` on our stored `@driver` object.

The test itself is where we see some complexity:

```
@driver.wait {
  @driver.find_element(:accessibility_id, "Login Screen")
}.click
username = @driver.wait {
  @driver.find_element(:xpath,
      "//XCUIElementTypeTextField[@name=\"username\"]")
}
password = @driver.find_element(:xpath,
  "//XCUIElementTypeSecureTextField[@name=\"password\"]")
username.send_keys("alice")
password.send_keys("mypassword")
@driver.find_element(:accessibility_id, "loginBtn").click
@driver.wait {
  @driver.find_element(:accessibility_id,
    "You are logged in as alice")
}
```

I'm making heavy use of a driver method called `wait`, which takes a block and returns (in this case) an element. What is the purpose of `wait`? When you ask Appium to find an element, it will do so immediately, exactly when you ask it to. That's all well and good, except for the fact that sometimes your app is not quite in a state where the element you are trying to find is ready. This can happen during view transitions, or while waiting on a network request, or for many other reasons. As humans, we're pretty good at watching the view until we notice it's in a good state for interaction.

As a robot, Appium is less sophisticated, and we have to be explicit about telling Appium to wait until conditions are right. We wouldn't want this test to fail just because Appium was too eager to try and find an element, and then declared it to be non-existent, when it just needed to wait a few more milliseconds! This is exactly what `wait` does: continually retries whatever is in the block for up to 30 seconds by default.

What the rest of the code does should be pretty self-explanatory given what we learned in the previous chapter. We're finding elements using a variety of locator strategies, and performing actions on them. The new action we encounter in this code is `send_keys`, which sends keystrokes to an input field.

The only thing apparently missing from this test is a verification. How are we proving that we have navigated to a logged-in area? Actually, a verification is hidden in the last chunk of the method. Telling Appium to wait until an element exists with the string `"You are logged in as alice"` is a verification, because if the element is present, we know we have reached the logged-in area. If the element is not present, the `wait` will eventually throw an error, and the test will be considered to have failed as a result.

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, 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 block from `login_spec.rb` were instead to read something like this:

```
@home_view.nav_to_login
@login_view.login("alice", "mypassword")
name = @user_view.get_logged_in_user
assert_equal(user, "alice")
```

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. It puts the responsibility of finding elements or getting data from them on these new objects we're referencing: `@home_view`, `@login_view`, and `@user_view`. These represent the Page Objects that I have been talking about. Let's take a look at what a Page Object might look like for the home view:

```
class HomeView
  @@login = [
    :accessibility_id,
    "Login Screen"
  ]
  def initialize(driver)
    @d = driver
  end
  def nav_to_login()
    @d.wait { @d.find_element(*@@login) }.click
  end
end
```

Basically, we maintain the information about selectors in class variables, and expose user-level behaviors in methods on the object. We encapsulate both

strategy and selector information with each element we care about, leading to super clean code. But this is very sparse and empty; all we care about doing with this view is getting to another view! So let's look at the Page Object Model for the login view now:

```
class LoginView
  @@username = [
    :accessibility_id,
    "username"
  ]
  @@password = [
    :accessibility_id,
    "password"
  ]
  @@login_btn = [
    :accessibility_id,
    "loginBtn"
  ]
  def initialize(driver)
    @d = driver
  end
  def login(username, password)
    @d.wait { @d.find_element(*@@username) }.send_keys(username)
    @d.find_element(*@@password).send_keys(password)
    @d.find_element(*@@login_btn).click
  end
end
```

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. Let's round out our object models with the model for the logged-in user page:

```
class UserView
  @@message = [
    :xpath,
    "//XCUIElementTypeOther[contains(@name, 'You are logged in as')]"
  ]
  def initialize(driver)
    @d = driver
  end
  def get_logged_in_user()
    message = @d.wait { @d.find_element(*@@message) }.text
    message.sub(/.*You are logged in as ([^ ]+).*/, '\1')
  end
end
```

Since what we want is to provide a method which returns the name of the logged-in user, we have to do some clever substitution on the text which is actually available to us, but otherwise this Object Model is also very straightforward. Assuming we have named these different files `home_view.rb`, `login_view.rb`, and `user_view.rb` respectively, we can update our main spec file to take advantage of these Object Models:

```
require 'appium_lib'
require 'test/unit/assertions'
require_relative './home_view.rb'
require_relative './login_view.rb'
require_relative './user_view.rb'
include Test::Unit::Assertions
describe 'Login' do
  before(:each) do
    appium_txt = File.join(Dir.pwd, 'appium.txt')
    caps = Appium.load_appium_txt(file: appium_txt)
    @driver = Appium::Driver.new(caps)
    @home_view = HomeView.new(@driver)
    @login_view = LoginView.new(@driver)
    @user_view = UserView.new(@driver)
    @driver.start_driver
  end
  after(:each) do
    @driver.driver_quit
  end
  it 'Login with valid credentials' do
    @home_view.nav_to_login
    @login_view.login("alice", "mypassword")
    user = @user_view.get_logged_in_user
    assert_equal(user, "alice")
  end
end
```

Despite the fact that we have some extra requires, 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. And if a selector changes for an app element, we have 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, but we can do a bit more cleanup. Right now our directory structure is getting a bit cluttered. Let's move the test file into its own directory called spec (paving the way for more tests), and the Page Objects into their own directory called views. Once we do that, we can also create a handy little file called `requires.rb` whose job is

simply to require everything we need so that we don't have to have a bunch of require calls cluttering up our test code. Assuming we've put `requires.rb` alongside our test code inside spec, it should look like:

```
require 'appium_lib'
require 'test/unit/assertions'
require_relative '../../common.rb'
require_relative '../views/home_view.rb'
require_relative '../views/login_view.rb'
require_relative '../views/user_view.rb'
include Test::Unit::Assertions
```

Finally, our login spec is getting pretty lean. 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. Let's do it now so that when we take a look at writing an Android test, everything will be ready. We can create a file called `common.rb` in the `project` directory, with the following contents:

```
def setup_driver
  appium_txt = File.join(Dir.pwd, '..', 'appium.txt')
  caps = Appium.load_appium_txt(file: appium_txt)
  Appium::Driver.new(caps)
end
```

Once we add this to our `requires.rb`, it's ready to use in our login spec file, which at last looks like this:

```
require_relative './requires.rb'
describe 'Login' do
  before(:each) do
    @driver = setup_driver
    @home_view = HomeView.new(@driver)
    @login_view = LoginView.new(@driver)
    @user_view = UserView.new(@driver)
    @driver.start_driver
  end
  after(:each) do
    @driver.driver_quit
  end
  it 'Login with valid credentials' do
    @home_view.nav_to_login
    @login_view.login("alice", "mypassword")
    user = @user_view.get_logged_in_user
    assert_equal(user, "alice")
  end
end
```

If everything is linked together correctly, you should be able to run rspec `login_spec.rb` in the `ios/spec` directory, and get the same passing testcase, only now with a much more beautified architecture. For reference, your project files should now look like:

```
/path/to/project
├── android
│   ├── appium.txt
│   └── TheApp-v1.2.1.apk
├── Gemfile
├── Gemfile.lock
└── ios
    ├── appium.txt
    ├── common.rb
    ├── spec
    │   ├── login_spec.rb
    │   └── requires.rb
    ├── TheApp-v1.2.1.app.zip
    └── views
        ├── home_view.rb
        ├── login_view.rb
        └── user_view.rb
```

Chapter 6

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. So rather than simply recreate the same structure we had for iOS, but now for Android, let's blow it up and put it back together in a cross-platform friendly way.

Previously, we were building a directory structure that ultimately would have looked like:

```
/path/to/project
├── android
│   ├── common.rb
│   ├── spec
│   │   ├── requires.rb
│   │   └── <specs here>
│   └── views
│       └── <views here>
└── ios
    ├── common.rb
    ├── spec
    │   ├── requires.rb
    │   └── <specs here>
    └── views
        └── <views here>
```

In other words, each of `android` and `ios` had their own whole test tree. Since we want to share as much code as possible, we're going to bring the `spec`, `views`, and `common` code up top, and distinguish between iOS and Android only inside spec:

```
/path/to/project
├── apps
├── common
├── spec
│   ├── android
│   ├── base
│   └── ios
└── views
```

Essentially, we'll be putting the test logic itself into spec files inside the `base` dir, and merely referencing those from the `android` and `ios` spec files, which will of course be neighbors to their platform-specific `appium.txt` and test app. To make sure we share as much code as possible, we'll also start using RSpec's configuration feature, which will hide away all the `before(:each)` and `after(:each)` boilerplate. Let's create a file inside our new common directory called `spec_helper.rb`. (If you haven't created the new directory structure above yet, do so now). `spec_helper.rb` should look like:

```
require 'rspec'
require 'appium_lib'
RSpec.configure do |config|
  config.before(:all) do
      appium_txt = File.join(Dir.pwd, 'appium.txt')
      caps = Appium.load_appium_txt(file: appium_txt)
      @driver = Appium::Driver.new(caps)
      @home_view = HomeView.new(@driver)
      @login_view = LoginView.new(@driver)
      if caps[:caps][:platformName].downcase == "ios"
        @user_view = IOSUserView.new(@driver)
    else
        @user_view = AndroidUserView.new(@driver)
    end
  end
  config.before(:each) do
    @driver.start_driver
  end
  config.after(:each) do
    @driver.driver_quit
  end
end
```

What we're doing in this file is basically creating `before` and `after` blocks that will be run in any spec file which includes this helper. There are two areas of our test setup that are platformspecific: (1) our `appium.txt` file which contains our platform-specific capabilities, and (2) the Page Object Models. We differentiate which `appium.txt` we're using automatically in virtue of relying on `Dir.pwd`; whichever directory we're running the rspec command from is where we'll look for `appium.txt`. So when running the command from android, it'll find the appropriate file to use for caps, and likewise for iOS. As for the Page Object Models, we'll have a closer look momentarily. Two of the three models are completely cross-platform, whereas one has minor

differences between platforms, so we need to choose the right model based on the capabilities we're using for the test.

Great. The next bit of code reuse is to make sure that both iOS and Android spec files use the same test code. Before, we had it in `ios/login_spec.rb`. Now, we're going to move it to `spec/base/login.rb`, and simply expose the test as a plain old method:

```
require_relative './requires.rb'
def login_with_valid_credentials
  it 'Login with valid credentials' do
    @home_view.nav_to_login
    @login_view.login("alice", "mypassword")
    user = @user_view.get_logged_in_user
    assert_equal(user, "alice")
  end
end
```

Notice that the method returns an `it` block. This is so we can simply call the method from inside our spec file's `describe` block without having to specify the block again. Notice also that our `requires.rb` is now in this same base directory. It's basically the same, but with updated paths:

```
require 'appium_lib'
require 'test/unit/assertions'
require_relative '../../common/spec_helper.rb'
require_relative '../../views/home_view.rb'
require_relative '../../views/login_view.rb'
require_relative '../../views/user_view.rb'
include Test::Unit::Assertions
```

With these changes, we're able to construct our new platform-specific spec files:

```
# /path/to/project/spec/ios/login_spec.rb
require_relative '../base/login.rb'
describe 'Login - iOS' do
  login_with_valid_credentials
end
```

```
# /path/to/project/spec/ios/login_spec.rb
require_relative '../base/login.rb'
describe 'Login - Android' do
  login_with_valid_credentials
end
```

These two files are extremely short—a mere 4 lines of code, including the `require`. We're relying on our `spec_helper` to decorate the `it` block with the appropriate before and after methods, and then we're simply mixing in the desired test behavior as the `login_with_valid_credentials` method. As we write more tests, we can simply add their method calls here.

The last real difference we need to look at is the `user_view.rb` Page Object Model. We now need two versions, because we can't, unfortunately, use completely identical locators across platforms. We can, however take advantage of Ruby's class inheritance to only change what we absolutely must:

```
class IOSUserView
  def message
    [:xpath,
    "//XCUIElementTypeOther[contains(@name, 'You are logged')]"]
  end
  def initialize(driver)
    @d = driver
  end
  def get_logged_in_user()
    message = @d.wait { @d.find_element(*self.message) }.text
    message.sub(/.*You are logged in as ([^ ]+).*/, '\1')
  end
end
class AndroidUserView < IOSUserView
  def message
    [:xpath,
    "//android.widget.TextView[contains(@text, 'You are logged')]"]
  end
end
```

Here we've decided that the 'default' view is `IOSUserView`, and then we extend it to `AndroidUserView` below. We've also changed from using class variables `(@@message)` to instance methods, because class variables don't play nice with inheritance in Ruby. So what we end up with is a set of Page Object Models that is platform-specific, but shares all the possible logic. We wire this conditionality into our test via the `spec_helper.rb` code we saw above. With this addition, our new full-blown project setup looks like:

```
/path/to/project
├── apps
│   ├── TheApp-v1.2.1.apk
│   └── TheApp-v1.2.1.app.zip
├── common
│   └── spec_helper.rb
    spec
│   ├── android
│   │   ├── appium.txt
│   │   └── login_spec.rb
│   ├── base
│   │   ├── login.rb
│   │   └── requires.rb
│   └── ios
│       ├── appium.txt
│       └── login_spec.rb
├── views
│   ├── home_view.rb
│   ├── login_view.rb
│   └── user_view.rb
├── Gemfile
└── Gemfile.lock
```

Given that we've decided to put our `apps` in an apps directory, we'll also need to make the changes in `appium.txt` to point to `/path/to/project/apps` instead of the platform-specific directory. Once you've done that, we can test our refactoring to make sure we haven't broken the iOS test:

```
cd /path/to/project/spec/ios
rspec login_spec.rb
```

And then, because we've made everything totally cross-platform, we can run the same test on Android:

```
cd /path/to/project/spec/android
rspec login_spec.rb
```

Don't forget to launch your emulator before attempting to run the test! And that's it. Of course, if you aren't dealing with a cross-platform app in your own testing, you may not want to go to such lengths to share code the way we did here. We went through this exercise to drive home the point that it's possible, and a good idea, to think about your test code as a software product just as much in need of refactoring and code reuse as your app itself. Keep it clean and keep it pretty!

RUNNING LOCAL TESTS WITH RAKE

So far we've been running our tests locally using the `rspec` test runner directly. But Ruby projects often use a task runner called `rake` (**R**uby's version of M**ake**), which comes with some niceties for command line execution of tasks like our tests. Let's upgrade our project for use with `rake`.

First, we need to add `rake` to our Gemfile:

```
source 'https://rubygems.org'
gem 'rspec', '~> 3.7.0'
gem 'appium_lib', '~> 9.10.0'
gem 'appium_console', '~> 2.8.1'
gem 'rake', '~> 12.3.0'
```

Make sure to run `bundle install` at this point to get the new gem on your system. Now we can build a `Rakefile` which specifies the kinds of tasks we want `rake` to run for us:

```
desc 'Run iOS tests'
task :ios do
  Dir.chdir 'spec/ios'
  exec 'rspec *_spec.rb'
end
desc 'Run Android tests'
task :android do
  Dir.chdir 'spec/android'
  exec 'rspec *_spec.rb'
end
```

This `Rakefile` tells rake that when we want to run the `ios` task, to first switch to the `spec/ios` directory, and then to execute the `rspec` command with the appropriate arguments to make it run any spec files it finds nearby. To make sure our `Rakefile` is put together correctly, we can query it for tasks:

```
> rake -T
rake android   # Run Android tests
rake ios       # Run iOS tests
```

Go ahead and give each of these commands a try, to verify the tests still pass successfully. Next, we'll move on to adding more `rake` tasks and running our tests in the cloud!

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 Rubygem to help with the Sauce API, called `sauce_whisk`.

Let's update our Gemfile accordingly (and run `bundle install`):

```
source 'https://rubygems.org'
gem 'rspec', '~> 3.7.0'
gem 'appium_lib', '~> 9.10.0'
gem 'appium_console', '~> 2.8.1'
gem 'rake', '~> 12.3.0'
gem 'sauce_whisk', '~> 0.1.0'
```

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 `sauce_whisk` 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 `spec_helper.rb` to include Sauce-specific test setup and app uploading:

```
require ‘rspec’
require ‘appium_lib’
require ‘sauce_whisk’
def using_sauce?
  user = ENV[‘SAUCE_USERNAME’]
  key = ENV[‘SAUCE_ACCESS_KEY’]
  user && !user.empty? && key && !key.empty?
end
def upload_app(app)
  storage = SauceWhisk::Storage.new({debug: true})
  storage.upload(app)
  “sauce-storage:#{File.basename(app)}”
end
RSpec.configure do |config|
  config.before(:all) do
    appium_txt = File.join(Dir.pwd, ‘appium.txt’)
    caps = Appium.load_appium_txt(file: appium_txt)
    if using_sauce?
      caps[:caps][:app] = upload_app(caps[:caps][:app])
    end
    @driver = Appium::Driver.new(caps)
    @home_view = HomeView.new(@driver)
    @login_view = LoginView.new(@driver)
    if caps[:caps][:platformName].downcase == “ios”
      @user_view = IOSUserView.new(@driver)
  else
      @user_view = AndroidUserView.new(@driver)
    end
  end
  config.before(:each) do
    @driver.start_driver
  end
  config.after(:each) do
    @driver.driver_quit
  end
end
```

What we’ve done is add two new methods, one to check if we want to run on Sauce by querying the environment variables, and another to upload an app to Sauce Storage using Sauce Whisk. We then use these methods to reset the app capability to the Sauce Storage location if we’re running on Sauce.

Next, we need to update our `appium.txt` for both iOS and Android, ensuring we’re using capabilities that are valid on Sauce. We also need to add the `appiumVersion` capability, because when we run tests on Sauce, we can choose between any version of Appium currently hosted on the provider.

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. Using this tool, I've updated the `appium.txt` files (and also removed our hack to prevent tests from running on Sauce before we wanted to).

For iOS:

```
[caps]
appiumVersion = "1.7.2"
platformName = "iOS"
platformVersion = "11.1"
deviceName = "iPhone 7"
app = "/path/to/project/apps/TheApp-v1.2.1.app.zip"
```

And for Android:

```
[caps]
appiumVersion = "1.7.2"
platformName = "Android"
platformVersion = "7.1"
deviceName = "Android GoogleAPI Emulator"
app = "/path/to/project/apps/TheApp-v1.2.1.apk"
automationName = "UiAutomator2"
```

Finally, we need to update our Rakefile, so that we can decide on the command line whether we'd like to run a test locally (as before), or opt in to the new Sauce support. We do this by parameterizing the tasks with a `:where` argument. This argument dictates whether we allow the Sauce environment variables to stay set. If we want a local test, we simply unset those variables, and our `spec_helper.rb` will leave off the Sauce logic.

```
desc ‘Run iOS tests’
task :ios, :where do |t, args|
  setup_env args[:where]
  Dir.chdir ‘spec/ios’
  exec ‘rspec *_spec.rb’
end
desc ‘Run Android tests’
task :android, :where do |t, args|
  setup_env args[:where]
  Dir.chdir ‘spec/android’
  exec ‘rspec *_spec.rb’
end
def setup_env(where)
  if where != “sauce”
    ENV[‘SAUCE_USERNAME’] = nil
    ENV[‘SAUCE_ACCESS_KEY’] = nil
  end
end
```

To successfully run a test on Sauce, we now need to first ensure our environment variables are set correctly. We can do this in our terminal 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.

Now, we can use our new `rake` task parameters to decide whether we want to launch a Sauce session or a local one:

```
rake android            # run android local
rake android[‘sauce’]   # run android on sauce
rake ios                # run ios local
rake ios[‘sauce’]       # run ios on sauce
```

Go ahead and run the Sauce versions. 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.

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, RSpec knows these things, and we can use Sauce Whisk to communicate them back to Sauce so we see prettier information on our Sauce dashboard.

Let's once again upgrade our `spec_helper.rb`. This time we just need to modify the `before(:each)` and `after(:each)` blocks to take a parameter we'll call `test`. This is RSpec's way of giving us access to information about the testcase. We can use this information to set the name capability before the test begins, and to set the status of the Sauce job after it has finished:

```
config.before(:each) do |test|
  @driver.caps[:name] = test.metadata[:full_description] if using_sauce?
  @driver.start_driver
end
config.after(:each) do |test|
  if using_sauce?
    SauceWhisk::Jobs.change_status(@driver.driver.session_id,
                                   test.exception.nil?)
  end
  @driver.driver_quit
end
```

Now, when you run `rake ios['sauce']` or `rake android['sauce']`, you'll see a nice humanreadable 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.

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
bundle update
rake android
```

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] $ /bin/sh -xe /var/folders/gv/vdnhjfy96ps3gb6vz8x...
+ cd /path/to/project
+ bundle update
Fetching gem metadata from https://rubygems.org/..........
Resolving dependencies...
Using rake 12.3.0
Using json 2.1.0
Using ffi 1.9.23
Using childprocess 0.8.0
Using rubyzip 1.2.1
Using selenium-webdriver 3.9.0
Using appium_lib_core 1.3.2
Using mini_portile2 2.3.0
Using nokogiri 1.8.2
Using tomlrb 1.2.6
Using appium_lib 9.10.0
Using awesome_print 1.8.0
Using bond 0.5.1
Using coderay 1.1.2
Using method_source 0.9.0
Using pry 0.11.3
Using numerizer 0.1.1
Using chronic_duration 0.10.6
Using spec 5.3.4
Using thor 0.20.0
Using appium_console 2.8.1
Using bundler 1.16.1
Using diff-lcs 1.3
Using rspec-support 3.7.1
Using rspec-core 3.7.1
Using rspec-expectations 3.7.0
Using rspec-mocks 3.7.0
Using rspec 3.7.0
Bundle updated!
+ rake android
.
Finished in 27.69 seconds (files took 1.42 seconds to load)
1 example, 0 failures
Finished: SUCCESS
```

This is the log of what happened during the job. As you can see, we made sure dependencies were up to date (in case our Gemfile had changed since the last build), and ran the android tests locally.

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 `rake ios` for `rake android`
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 rake command with two new ones: `rake ios['sauce'] && rake android['sauce']`

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 command line and the Appium Desktop Inspector
4. How to write simple tests in Ruby and RSpec using the Appium Ruby client
5. How to refactor and organize test code, leveraging Page Objects to make it fully crossplatform
6. How to use `rake` 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