GETTING STARTED WITH

Appium

BY **DAVE HAEFFNER**
REVIEWED BY **JONATHAN LIPPS,** APPIUM PROJECT LEAD
AND **DAN CUELLAR,** APPIUM CREATOR

CONTENTS

WHAT IS APPIUM?

Appium is a free and open-source mobile automation framework used for native, hybrid, and mobile web apps. It works on iOS, Android, Mac, and Windows apps using the WebDriver protocol. WebDriver (the API to automate browsers, maintained by the Selenium project) is currently going through a W3C (World Wide Web Consortium) specification.

GETTING STARTED

In order to get up and running on your local machine, you need to download an Appium server and client bindings for your preferred programming language. There are Appium language bindings for multiple programming languages. The officially supported ones (in alphabetical order) are:

- C# (.NET)
- Java
- JavaScript (Node.js)
- PHP
- Python
- Ruby

NOTE: *There are also some platform specific dependencies you'll need for testing on iOS and Android. For an up-to-date reference, be sure to check out the Appium documentation.*

APPIUM SERVER

There are two approaches you can take for getting the Appium Server onto your machine. You can use the command-line server available through `npm` and install it with `npm install -g appium`.

Alternatively, you can use the Appium Desktop app, which is an open source app for Mac, Windows, and Linux which gives you the Appium server in a simple and flexible UI (along with some extra functionality). You can download and install the latest version here.

NOTE: *If you have any questions about Appium Desktop be sure to check out its documentation.*

After you have the server installed, be sure to run it (`appium` if using the command-line server, or launch the Appium Desktop app and click `Start Server`), then download and install the client bindings for your preferred programming language.

C# (WITH NUGET)

Use the following commands from the Package Manager Console window in Visual Studio to install the Appium C# bindings (which is an extension of the Selenium client bindings).

```
Install-Package Selenium.WebDriver
Install-Package Newtonsoft.Json
Install-Package Selenium.Support
Install-Package Castle.Core
```

NOTE: *You will need to install Microsoft Visual Studio and NuGet to install these libraries and build your project. For more information on the Appium C# bindings checkout the API documentation.*

JAVA (WITH MAVEN)

In your test project, add the following code to your `pom.xml`. Once that's done you can either let your IDE (Integrated Development Environment) use Maven to import the dependencies or open a command-prompt, `cd` into the project directory, and run `mvn clean test-compile`.

```
<!-- https://mvnrepository.com/artifact/io.appium/java-client -->
<dependency>
    <groupId>io.appium</groupId>
    <artifactId>java-client</artifactId>
    <version>LATEST</version>
    <scope>test</scope>
</dependency>
```

NOTE: *You will need to have the Java Development Kit (version 7+, or 8+) and Maven installed on your machine. For more information on the Appium Java bindings, check out the API documentation.*

A brief history of web and mobile app testing.

Find out how Sauce Labs can accelerate your testing to the speed of awesome.

For a demo, please visit saucelabs.com/demo
Email sales@saucelabs.com or call (855) 677-0011 to learn more.

SAUCELABS
Testing at the speed of awesome.

JAVASCRIPT (NPM)

Type the following command into a command-prompt to install the JavaScript bindings for Appium:

```
npm install webdriverio
npm install wdio-appium-service --save-dev
```

NOTE: *You will need to have Node.js and NPM installed on your machine. For more information about the Appium JavaScript client bindings, check out the API documentation.*

PHP (WITH COMPOSER)

Add `"appium/appium-php": "dev-master"` to the `require` section of your `composer.json` file, as well as the Appium PHP GitHub repository to the `repositories` section.

```
{
    "name": "username/my-php-project",
    "repositories": [
        {
            "type": "vcs",
            "url": "https://github.com/appium/php-client"
        }
    ],
    "require": {
        "appium/php-client": "dev-master"
    }
}
```

Then install the dependencies and run your tests.

```
composer install
vendor/phpunit/phpunit/phpunit <mytest.php>
```

NOTE: *For more details on the PHP Appium Client bindings, be sure to check out the documentation.*

PYTHON

The following command installs the Python bindings for Selenium:

```
pip install Appium-Python-Client
```

NOTE: *You will need to install Python, pip, and setuptools in order for this to work properly. For more information on the Appium Python bindings, check out the API documentation.*

RUBY

Type the following command to install the Appium Ruby bindings:

```
gem install appium_lib
```

NOTE: *You will need to install a current version of Ruby which comes with RubyGems. You can find instructions on the Ruby project website. For more information on the Appium Ruby bindings, check out the API documentation.*

PLATFORM DEPENDENCIES (IOS)

For testing on iOS, you'll need to install Xcode and the Xcode Command Line Tools Package. The Command Line Tools Package can be installed with the `xcode-select --install` command in your terminal once Xcode has been installed.

You'll also need to authorize Appium to use the iOS Simulator, which you can do with the following commands:

```
npm install -g authorize-ios
sudo authorize-ios
```

If you're using Xcode 8 & iOS 10 (or newer), you'll also need to install the following libraries:

```
brew install carthage
brew install libimobiledevice
npm install -g ios-deploy
```

NOTE: *For additional information on system setup requirements (since your needs might be different), be sure to check out the Appium documentation.*

PLATFORM DEPENDENCIES (ANDROID)

For testing on Android, you'll need to download the Android Command Line Tools, unzip them to a known location, and specify that location in an `ANDROID_HOME` environment variable and your system path.

```
export ANDROID_HOME="$HOME/android"
export PATH="$ANDROID_HOME/tools:$PATH"
export PATH="$ANDROID_HOME/platform-tools:$PATH"
```

Once done, run `android sdk` from the command-line to select and install the SDK version that you'd like to use (e.g., `Android SDK Build Tools 23.0.3`). Then run `android avd` from the command-line to create an emulator for your tests to use (for the examples that follow we'll be using an `avd` called `example`).

NOTE: *For additional information on system setup requirements (since your needs might be different), be sure to check out the Appium documentation.*

VERIFY YOUR MACHINE IS SETUP

To verify that all of Appium's dependencies are met, you can use appium-doctor.

To install it, use `npm install -g appium-doctor`. Then run it with `appium-doctor`. You can supply it with `--ios` or `--android` flags to verify that all of the dependencies are set up correctly for that particular platform.

NOTE: *The remaining examples will be shown using Ruby.*

SAMPLE APPS

Don't have a test app? Don't sweat it. There are pre-compiled test apps available to kick the tires with. You can grab the iOS app here and the Android app here.

Just make sure to put your test app in a known location, because you'll need to reference the path to it next.

APP CONFIGURATION

When it comes to configuring your app to run on Appium there are a lot of similarities to Selenium -- namely the use of Capabilities (e.g., "caps" for short). You can specify the necessary configurations of your app through caps by storing them in a file called `appium.txt`.

Here's what `appium.txt` looks like for the iOS test app to run in an iPhone simulator:

```
[caps]
platformName = "iOS"
deviceName = "iPhone Simulator"
app = "/path/to/UICatalog.app"
automationName = "XCUITest"

[appium_lib]
sauce_username = false
sauce_access_key = false
```

And here's what `appium.txt` looks like for Android on a local emulator:

```
[caps]
platformName = "ANDROID"
deviceName = "Android 6.0"
app = "/path/to/api.apk"
avd = "example"

[appium_lib]
sauce_username = false
sauce_access_key = false
```

For Android, note the use of `avd`. The `"example"` value is for the Android Virtual Device that was mentioned previously in the `Platform Dependencies (Android)` section. This is necessary for Appium to auto-launch the emulator and connect to it. This type of configuration is not necessary for iOS.

NOTE: *You can see a full list of available* `caps` *here.*

Go ahead and create an `appium.txt` file with the caps for your app.

INTERROGATING YOUR APP

Writing automated scripts to drive an app in Appium is very similar to how it's done in Selenium. You first need to choose a locator which finds an element that you can then perform an action against.

In Appium, there are two approaches to interrogating an app to find the best locators to work with — through the command-line with Ruby's `irb` (a.k.a. interactive Ruby) or through Appium Desktop. Here are some examples of how to use each of these approaches to decompose and understand your app:

USING THE COMMAND-LINE

INITIAL SETUP

1. Confirm you have the Appium Ruby client bindings installed (e.g., `gem install appium_lib`)
2. Open a terminal window and run the Appium server (e.g., `appium`)
3. Open another terminal window, change its directory to where your `appium.txt` file is located.
4. Type `irb` to launch the interactive Ruby terminal session
5. Input the following commands one at a time (hitting `Enter` after each and waiting for the prompt to return control to you)

```
require 'appium_lib'
caps = Appium.load_appium_txt file: File.join(Dir.pwd,
'appium.txt')
Appium::Driver.new(caps).start_driver
Appium.promote_appium_methods Object
```

NOTE: *There is a library called Appium Ruby Console which takes care of this initial setup for you, but it's not officially supported by the Appium project. Using* `irb` *is recommended since it will remain stable and more closely resembles the code you need to write in your actual tests.*

After you've executed all of the commands, you will have a window with your app loaded that you can interact with manually, as well as an interactive command-prompt for Appium that you can issue commands to. To end the Appium client session, you can issue a `driver.quit` command and end the `irb` session with the `quit` command.

AN IOS EXAMPLE

To get a quick birds eye view of our iOS app structure, let's get a list of the various element classes available. With the `page_class` command we can do just that.

```
[1] pry(main)> page_class
20x XCUIElementTypeStaticText
10x XCUIElementTypeCell
10x XCUIElementTypeOther
2x XCUIElementTypeWindow
1x XCUIElementTypeStatusBar
1x XCUIElementTypeNavigationBar
1x XCUIElementTypeTable
1x XCUIElementTypeApplication
```

With the `page` command we can specify a class name and see all of the elements for that type. When specifying the element class name, we can either specify it as a string, or a symbol (e.g., `'XCUIElementTypeStaticText'` or `:XCUIElementTypeStaticText`).

```
[2] pry(main)> page :XCUIElementTypeStaticText
XCUIElementTypeStaticText
   name, label, value: UICatalog
   hint:
XCUIElementTypeStaticText
   name, label, value: Action Sheets
   hint:
XCUIElementTypeStaticText
   name, label, value: AAPLActionSheetViewController
   hint:
XCUIElementTypeStaticText
   name, label, value: Activity Indicators
   hint:
...
```

Within each element of the list, notice their properties — things like `name`, `label`, and `value`. This is the kind of information we will want to reference in order to interact with the app. Let's take the second element for example:

```
XCUIElementTypeStaticText
   name, label, value: Action Sheets
   hint:
```

In order to find this element and interact with it, we can can search for it with a couple of different commands: `find`, `text`, or `text_exact`.

```
[3] pry(main)> find('Action Sheets')
#<Selenium::WebDriver::Element:0x43a1194018243038
id="71760D5F-FFA7-4379-BF4A-41048BB87F99">
```

We'll know that we successfully found an element when we see a `Selenium::WebDriver::Element` object returned. To verify that we have the element we expect, let's access the name attribute for it.

```
[4] pry(main)> find('Action Sheets').name
"Action Sheets"
```

AN ANDROID EXAMPLE

To get a quick birds eye view of our Android app structure, let's get a list of the various element classes available. With the `page_class` command we can do just that.

```
[1] pry(main)> page_class
11x android.widget.TextView
5x android.widget.FrameLayout
2x android.widget.LinearLayout
2x android.view.ViewGroup
1x android.widget.ListView
1x android.widget.ImageView
1x hierarchy
```

With the `page` command, we can specify a class name and see all of the elements for that type. When specifying the element class name, we can specify it as a string (e.g., `'android.widget.TextView'`).

```
[2] pry(main)> page 'android.widget.TextView'

android.widget.TextView (0)
  text: API Demos
  strings.xml: activity_sample_code

android.widget.TextView (1)
  text, desc: Access'ibility
  id: android:id/text1

android.widget.TextView (2)
  text, desc: Accessibility
  id: android:id/text1

android.widget.TextView (3)
  text, desc: Animation
  id: android:id/text1
...
```

Within each element of the list, notice their properties -- things like `name`, `label`, and `value`. This is the kind of information we will want to reference in order interact with the app. Let's take the third element for example:

```
android.widget.TextView (2)
  text, desc: Accessibility
  id: android:id/text1
```

In order to find that element and interact with it, we can search for it by text or by id. Since there are no unique IDs, we'll use `text`.

```
[3] pry(main)> text('Accessibility')
#<Selenium::WebDriver::Element:0x21934249d0953622 id="1">
```

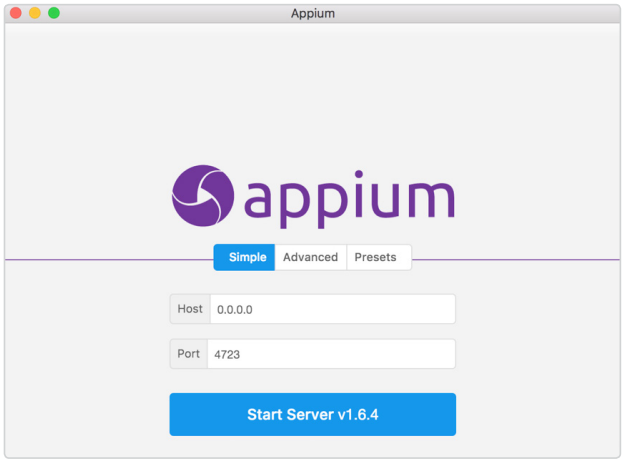
We'll know that we successfully found an element when we see a `Selenium::WebDriver::Element` object returned. To verify that we have the element we expect, let's access the name attribute for it.

```
[4] pry(main)> text('Accessibility').name
"Accessibility"
```

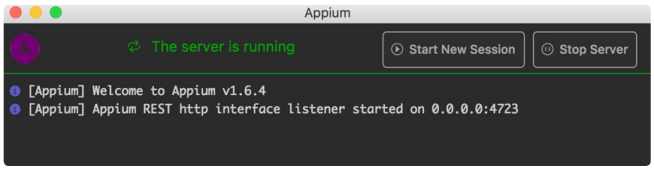
USING THE APPIUM DESKTOP APP

Run the Appium server through Appium Desktop app by clicking `Start Server` with the defaults provided. Once it's running, you should see a log window listing the latest server activity.

NOTE: *If you have Appium running in a terminal window you'll need to kill it by issuing a* `CTRL+C` *command.*



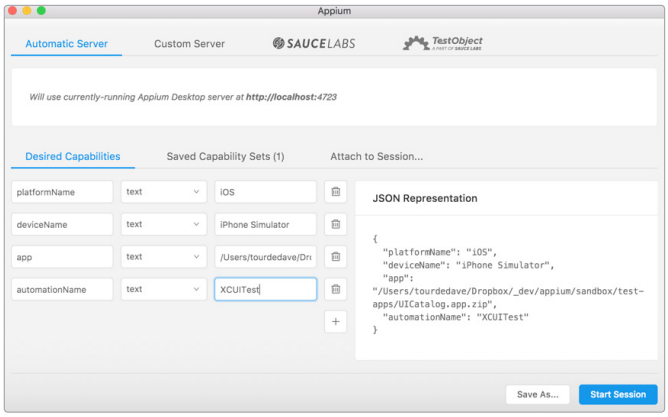
APPIUM DESKTOP INIT



APPIUM DESKTOP RUNNING

Click `Start New Session` at the top of the screen and specify your capabilities manually (the same as you've specified in your `appium.txt` file). Then click `Start Session`.

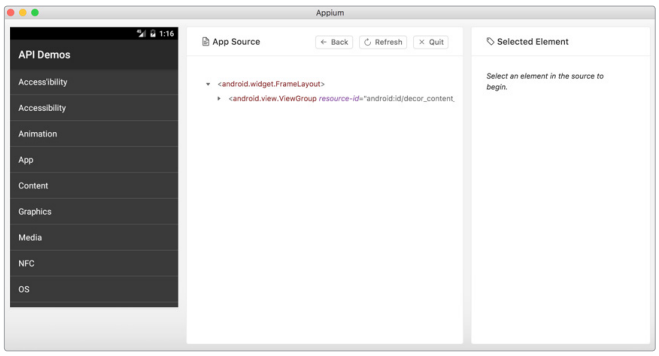
NOTE: *You can save your configuration by clicking the* `Save As` *button next to* `Start Session` *and giving it a helpful name. Then you can refer to this configuration easily later.*



APPIUM DESKTOP WITH MANUAL CAPS

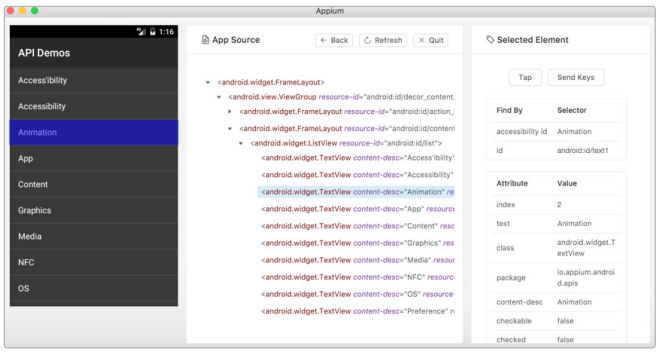
NOTE: Alternatively, you can take advantage of Appium Desktop's ability to connect to an already running session. To do that, you'll need to grab the session of ID of the already running Appium session (which is available in the Appium Server log output), click `Start New Session`*, paste the ID into the* `Attach to Session...` *menu option, and click* `Attach to Session`*.*

When the session is fully loaded, a three-pane inspector window will open that shows a screenshot of the app on the left, the underlying source code of the app's UI in the center, and details about the element you are attempting to interact with on the right.



APPIUM DESKTOP INSPECTOR INIT

In the left-pane you can click on an element you'd like to interact with. When you do, the middle pane will update with the source code. The right-pane will then show details about the element and offer actions you can take against it (e.g., `Tap` or `Send Keys`).



APPIUM DESKTOP INSPECTOR DRILL DOWN

COMMANDS AND OPERATIONS

The most common operations you'll end up doing in Appium are finding an element (or a set of elements) and performing actions with those elements (e.g. tap, type text, swipe, etc.). You can also ask questions about the elements (e.g. Is it displayed? Is it enabled? etc.), pull information out of the element (e.g. the text of an element or the text of a specific attribute within an element), or perform additional gestures.

```
# find just one, the first one Appium finds
element = find('locator')

# find all instances of the element on the page
element = finds('locator')
```

WORK WITH A FOUND ELEMENT

```
# chain actions togethers
find('locator').click

# store the element and then click it
element = find('locator')
element.click
```

PERFORM MULTIPLE COMMANDS

```
element.click # clicks an element
element.clear # clears an input field
element.send_keys('input text') # type text into an input
  field
```

ASK A QUESTION

```
element.displayed?  # is it visible to the user?
element.enabled?    # can it be selected by the user?
```

RETRIEVE INFORMATION

```
# directly from an element
element.text

# by an attribute name
element.attribute('clickable')
```

GESTURES

```
# Swipe (by Coordinates)
swipe start_x: 75, start_y: 500, end_x: 75, end_y: 0,
duration: 0.8

# Swipe (by Locators)
start   = find('starting-element-locator').location.to_h
finish  = find('ending-element-locator').location.to_h
Appium::TouchAction.new.press(start).wait(200).move_
to(finish).release.perform

# Long Press
element = find('locator').location.to_h
location.merge!(fingers: 1, duration: 2000
Appium::TouchAction.new.long_press(button).release.
perform
```

NOTE: For a full list of available commands and operations, be sure to check out the documentation.

AN EXAMPLE TEST

To tie all of these concepts together, here is a simple test that shows how to use Appium to exercise common functionality by launching an app, navigating through it, triggering and dismissing an alert, and waiting for the app to return to the previous screen.

```
require 'appium_lib'

describe 'Alert' do

  before(:each) do
    caps = Appium.load_appium_txt file: File.join(
      File.dirname(__FILE__), 'appium.txt')
    Appium::Driver.new(caps).start_driver
    Appium.promote_appium_methods
      RSpec::Core::ExampleGroup
  end

  after(:each) do
    driver.quit
  end

  it 'open and closed' do
    text('Alerts').click
    text('Show Simple').click
    id('OK').click
    wait { text('Show Simple') }
  end

end
```

AVAILABLE DRIVERS

To run your Appium tests on a different platform, platform version, or driver, you'll need to specify certain capabilities. Here's a breakdown.

PLATFORM NAME	AUTOMATION NAME	DESCRIPTION
iOS	not required	instruments driver (up to iOS 9.3)
iOS	XCUITest	XCUITest driver (for iOS 9.3+)
iOS	YouiEngine	You.i.Engine driver
Android	not required	UiAutomator driver
Android	UiAutomator2	UiAutomator2 driver
Android	Selendroid	Selendroid driver
Android	YouiEngine	You.i.Engine driver
Windows	not required	Windows WinAppDriver
Mac	not required	Appium4Mac driver

NOTE: *For more details on what's supported within Appium, take a look at the Appium Design documentation.*

APPIUM SERVICE PROVIDERS

Rather than take on the overhead of standing up and maintaining a test infrastructure, you can easily outsource these services to a third-party cloud provider like Sauce Labs. With Sauce Labs you'll also be able to get access to real devices as well as simulators and emulators.

NOTE: *You'll need an account to use Sauce Labs. Their free trial offers enough to get you started. And if you're signing up because you want to test an open source project, then be sure to check out their Open Sauce account.*

In the Appium Ruby bindings this is a turn-key thing to enable within `appium.txt`. You just need to provide your Sauce username (which you use to log in) and access key (which is available under the `My Account` page in your Account Dashboard). You will also need to specify where your test app is located (either at a publicly available URL or uploaded into the Sauce cloud).

```
app = "https://github.com/appium/ruby_lib/raw/master/
android_tests/api.apk"
# ...
[appium_lib]
sauce_username = 'your-sauce-username'
sauce_access_key = 'your-sauce-access-key'
```

Alternatively, you can bypass the `appium.txt` file and specify your configuration as part of your test setup code.

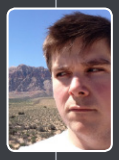
```
# ...
before(:each) do
  caps = {
    caps: {
      platformName: "Android",
      deviceName: "Android Emulator",
      platformVersion: '6.0',
      app: "https://github.com/appium/ruby_lib/raw/
        master/android_tests/api.apk" },
    appium_lib: {
      sauce_username: ENV['SAUCE_USERNAME'],
      sauce_password: ENV['SAUCE_PASSWORD'] }
  }

  Appium::Driver.new(caps).start_driver
  Appium.promote_appium_methods
    RSpec::Core::ExampleGroup
end
# ...
```

NOTE: *Also, you can spin up a Sauce Labs session from within the Appium Desktop app's* `Start New Session` *menu. After providing your credientials, specify the caps for the setup you need and click* `Start Session`.

NOTE: *You can see a full list of Sauce Labs' available platform options here. There's also a handy configuration generator which will tell you what values to plug into your test. Be sure to check out Sauce Labs' documentation portal for more details.*

ABOUT THE AUTHOR



DAVE HAEFFNER is the writer of Elemental Selenium—a free, once weekly Selenium tip newsletter that's read by thousands of testing professionals. He's also the creator and maintainer of the-internet (an open-source web app that's perfect for writing automated tests against), and author of The Selenium Guidebook. He's helped numerous companies successfully implement automated acceptance testing; including The Motley Fool, ManTech International, Sittercity, and Animoto. He's also an organizer of Selenium Conf, an active member of the Selenium project, and has spoken at numerous conferences and meet-ups around the world about automated functional testing.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513

888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com