The Beginner's Guide of Using Appium for Mobile App Testing

a publication of
bitbar.

Table of content

Introduction

Appium has quickly become one of the most prominent test automation framework for mobile app, game and web testing. As an open source test automation framework, it drives Android and iOS apps, regardless of native, hybrid or mobile web apps, using the WebDriver protocol.

What is Appium Mobile Testing - Why Should You Care?

It works well on native apps – the ones that are written using the iOS or Android SDKs, mobile web apps that are accessed using a mobile browser as well as hybrid apps that are utilizing webview and are wrapped inside your app. For example, Appium is also very popular among mobile game developers who typically use some advanced techniques for testing input-driven mobile games even running on two different platforms at the same time and by the same script.

As one of the most widely used frameworks in Testdroid Cloud and with many years of experiences of supporting it, we have decided to compose the very first Appium ebook and share it with all of you.

Chapter

01

How to Get Started, Setup and Run Your First Tests

To get started with using Appium for mobile app testing, you need to get set on very basic things, such as setting up the environment, installing Appium pieces, selecting the preferable programming language, etc. This chapter will start with every basic thing.

Getting Started

To get started, you can download the versatile Appium samples here. Depending on which programming language you will be using, you can select the appropriate samples and client library as follows.

One more thing before we enter the details, you need to get Github installed on your machine, as most of the samples we created are located in our Github repositories. If you are new to Git, you can follow the instructions here to install Git on popular operating system. Below is the abbreviated version of the installation procedures.

Mac OS X

Download the latest git command line tool from http://git-scm.com/download/mac install it using normal Mac installation procedure.

Linux

Use the following command to get Git installed on your Linux machine:

```
$ sudo apt-get install git
```

Windows

The easiest and most straightforward way is to install the Github Windows application.

Once you get Github installed, you need to get one of the four programming languages installed. We'll go through them one by one.

Python

Mac OS X and Linux

Our Python samples have been created with 2.7.x version in use, so best compatibility can be expected with the same. Check that newest Python 2.7.x version is installed by using the following command:

```
$ python --version
```

If not installed, you can install Python by using the following commands:

```
Linux: $ sudo apt-get install python2.7
OSX: $ brew install python
```

Brew is a handy package manager tool, similar to apt-get. If you don't have it, check the brew website for its one-liner installation.

Then, check if 'pip' module is installed. Use the following command:

```
$ pip --version
```

If 'pip' is appropriately installed, you'll see something like this:

```
pip 1.5.6 from /Library/Python/2.7/site-packages (Python 2.7)
```

If not, use the following command to install it:

```
Linux: $ sudo apt-get install python-pip
OSX: $ sudo easy_install pip
```

Next, install Selenium module for Python:

```
$ pip install selenium
```

And finally, verify that Selenium got installed:

```
$ pip list | grep selenium
```

Windows

Ensure that newest Python 2.7.x version is installed. Go to command line and use the following:

```
> python --version
```

If Python 2.7 or newer is not installed, download and run the setup from Python's download center. To add Python into environment variables, go to Windows "System properties" › "Advanced System Settings" › "Environment Variables" › "System Variables" › choose "Path" and press "Edit..." and then insert (assuming you have installed Python in the default location) ;C:Python27;C:Python27Scripts at the end separating each path with semicolon ; . Make sure to re-launch the command prompt to bring new environment variables into effect.

Then, check whether Python's pip module is already installed:

```
> pip --version
```

Install pip if it's not already (we assume here that you have cURL installed. If not, check this out):

```
> curl https://raw.github.com/pypa/pip/master/contrib/get-pip.py > get-pip.py
> python get-pip.py
> del get-pip.py
```

Install Python's Selenium module:

```
$ pip install selenium
```

› Java

Appium testing supports writing tests in multiple programming languages, including Java. Testing can be done against native or hybrid apps or responsive web pages on both iOS and Android devices. Only the test setups differ.

With Java, things are pretty simple and you only need to configure your test file accordingly. You can use SampleAppium(iOS)Test.java as an example/template.

In case you don't have an IDE with Maven included and would like to launch the example from command line, you will need to make sure that Maven is properly installed. Here's a link to the installation instructions.

› C#

Windows

Launch the AppiumTest.sln on Visual Studio and make sure that NUnit Test Adapter is

installed through the Extension Manager. Use Test Explorer to run your tests.

Linux/OS X

First, install Monodevelop for C# support. Then, download dependencies using Nuget:

```
$ nuget install Test123/packages.config -OutputDirectory
packages
```

To build the package, simply use the following command on correct path:

```
$ xbuild
```

Ruby

First, install the latest stable release of Ruby:

```
$ curl -sSL https://get.rvm.io | bash -s stable
$ rvm install ruby
```

Then, make sure RVM is using the correct Ruby by default:

```
$ rvm list
$ rvm --default use 2.1.1
```

In case you have an old Ruby/RVM, you can upgrade those with the following commands:

```
$ rvm get head
$ rvm autolibs homebrew
$ rvm install ruby
```

Check that it's installed properly by printing out the Ruby version:

```
$ ruby --version
```

Update RubyGems and Bundler:

```
$ gem update --system
$ gem install --no-rdoc --no-ri bundler
$ gem update
$ gem cleanup
```

Check that RubyGems is >= 2.1.5

```
$ gem --version
```

Run bundler at the Ruby example directory to install dependencies:

```
$ bundle install
```

All Programming Languages: How to Configure Testdroid Specific Settings

If you have used any of those example files as a template, add your Testdroid user credentials in this script. You can also use environmental variables `TESTDROID_USERNAME` and `TESTDROID_PASSWORD` to get your credentials used:

```
String testdroid_username = env.get("TESTDROID_USERNAME");
String testdroid_password = env.get("TESTDROID_PASSWORD");
```

Or alternatively, you can edit `testdroid_username` and `testdroid_password` in your source file:

```
capabilities.setCapability("testdroid_username", 'john.doe@
bitbar.com');
capabilities.setCapability("testdroid_password",
'secretPassword123');
```

If you are new with desired capabilities or if you are looking for more information on how to use those efficiently, take a look at this article.

Also, if you want to run tests against your application, make sure to change a file path

to your application binary (whether you are running against APK or IPA):

```
private static final String TARGET_APP_PATH = "../../../apps/
builds/BitbarSampleApp.apk";
```

In addition, there are lots of possible ways to configure your test run for our devices. In order to do this, you need to configure those desired capabilities. The current examples are as follows:

```
DesiredCapabilities capabilities = new DesiredCapabilities();
capabilities.setCapability("platformName", "Android");
capabilities.setCapability("testdroid_target", "Android");
capabilities.setCapability("deviceName", "Android Device");

capabilities.setCapability("testdroid_project",
"LocalAppium");

capabilities.setCapability("testdroid_testrun", "Android Run
1");

// See available devices at: https://cloud.testdroid.
com/#public/devices

capabilities.setCapability("testdroid_device", "Samsung
Galaxy Nexus GT-I9250 4.2.2");

capabilities.setCapability("testdroid_app", fileUUID);
```

In this Java example, you only really need to configure `testdroid_username` and `testdroid_password`, since sample application upload is included.

Running Your First Test

Before the test can start you need to upload your application to Testdroid Cloud. This can be done either via API or you can do it manually as well.

› Python

To upload your app file (either APK or IPA) to Testdroid Cloud, open and configure the `upload.py` script. As we walked it through, you only need to configure your username (email) and password that you registered with to Testdroid Cloud. Also, you need to set the full path to your mobile app. This can be an Android or iOS application. Then execute this:

```
$ python upload.py
```

To run a test:

```
$ python testdroid_android.py
```

Java

You can run test from your IDE or directly from command line using Maven:

```
> mvn clean test -Dtest=SampleAppiumTest
```

or to be more precise:

```
> mvn clean test -Dtest=com.testdroid.appium.android.sample.
SampleAppiumTest
```

or run all the tests:

```
> mvn clean test
```

C#

To run tests, either launch them in Visual Studio via the Text Explorer or use the nunit console command:

```
$ nunit-console Test123/bin/Debug/TestdroidAndroidSample.dll
```

Ruby

Run the tests with rspec:

```
$ rspec testdroid_android.rb
```

Chapter

02

How to Get Started, Setup and Run Your First Tests

Depending on which language you've selected and installed, we'll go through each of the programming languages one by one. This chapter will cover how to use Python for Appium app testing.

If you properly went through the first chapter with all basic setup, installation of Python, and some other things, you can continue with the basic code example.

Python is a great script language choice for those who want to get down to business quick and simple. The syntax is really easy to learn and implement and there isn't typically a huge cluster of different files required. When you want to test some untapped Appium features, you can use python to check it out.

Create and Run An Upload Script

Before the actual testing can happen, you need to upload you mobile application to the cloud (create an account here, if you don't have one). The app can be in form of APK or IPA – and to get you an effective start, we'll create a brief upload script here. The upload script example is also available at Github. For IPA files, please also make sure they're properly built for use in Testdroid Cloud.

```
import requests
import base64

username = 'USERNAME'
password = 'PASSWORD'
upload_url = 'http://appium.testdroid.com/upload'
myfile = '../../../apps/builds/Testdroid.apk'

def build_headers():
  return { 'Authorization' : 'Basic %s' % base64.
b64encode(username+":"+password) }

files = {'file': ('Testdroid.apk', open(myfile, 'rb'),
'application/octet-stream')}
r = requests.post(upload_url, files=files, headers=build_
headers())
print r.text
```

Configure your `username`, `password` and `myfile` attributes with your Testdroid Cloud login credentials, and file path for your application (this can be APK or IPA).

Now, you can run the upload script using the following command:

```
$ python update.py
```

As an output you'll get something like this:

```
{"status":0,"sessionId":"99becd25-3183-4c7f-998e-
84ef307de7a5","value":{"message":"uploads successful",
"uploadCount":1,"rejectCount":0,"expiresIn":1800,"uploads":
{"file":"99becd25-3183-4c7f-998e-84ef307de7a5/test.apk"},
"rejects":{}}}
```

From this response message you need to store the application's ID and file name in Testdroid Cloud. The above example is `'99becd25-3183-4c7f-998e-84ef307de7a5/test.apk'`.

Creating and Running a Test Script

Generic Settings

There are some common settings that you need to set in all scripts regardless of app type that you are testing. Each `testdroid_*.py` file needs to be updated with the appropriate values. For this example, here are all the values that you need to edit:

`screenshotDir` – where should screenshots be stored on your local drive

`testdroid_username` – your email that you registered with to Testdroid Cloud

`testdroid_password` – your Testdroid Cloud password

`testdroid_project` – the name of the project you want to use in Testdroid Cloud. Each project must have a unique name, which can also be modified later. The project will be automatically created, if it doesn't already exist in cloud.

`testdroid_testrun` – name of this test run. Test run names can also be modified afterwards and they don't have to be unique.

`testdroid_app` – the location of the app you uploaded to cloud. Eg. if you uploaded your app using the upload.py script this would look like `'99becd25-3183-4c7f-998e-84ef307de7a5/test.apk'`. Alternative way to this is uploading the app through the cloud portal and then use the value `'latest'` instead of an id path.

Native Android Specific Settings

You can use our template file as an example: testdroid_android.py from Github.

In order to configure this script for your testing needs and for your own app, you can edit two additional capabilities as follows:

`appPackage` – Java package of the Android app you want to run

`appActivity` – Activity name for the Android activity you want to launch from your package. Typically this is the main activity.

Native iOS Specific Settings

You can use our template file as an example: testdroid_ios.py from Github.

NOTE! In addition to other configurations and desired capabilities, Appium requires a bundle id to be included in the test script:

`bundleId` – this is your application's unique name. For example, it can be used in your Python test script as follows:

```
    'bundleId': 'com.bitbar.testdroid.BitbarIOSSample'
```

An Example Template For Desired Capabilities and Setup

You can find all the template scripts from our Github repository. There are examples for Android, iOS, Safari, Chrome, and additionally Android hybrid for hybrid apps.

Okay, let's look at a concrete Python test script example then and what goes in the code:

Imported libraries, bundles and other libraries/dependencies:

```
import os
import sys
import time
import unittest
from time import sleep
from appium import webdriver
from device_finder import DeviceFinder
from selenium.common.exceptions import WebDriverException
```

Next, let's create a class with the device, login and configuration specific desired capabilities. For example:

```
class AndroidTest(unittest.TestCase):
  def setUp(self):
  self.screenshotDir = os.environ.get('TESTDROID_
SCREENSHOTS') or "/absolute/path/directory"
  testdroid_url = os.environ.get('TESTDROID_URL') or
"https://cloud.testdroid.com"
  testdroid_username = os.environ.get('TESTDROID_USERNAME')
or "user@email.com"
  testdroid_password = os.environ.get('TESTDROID_PASSWORD')
or "password"
  appium_url = os.environ.get('TESTDROID_APPIUM_URL') or
'http://appium.testdroid.com/wd/hub'
```

To choose a device from Testdroid Cloud, you can search for the desired model from Device Groups page. Once you found the model you want to use, simply copy the full name of the device to `testdroid_device` variable.

You can also use DeviceFinder class to automatically find an available freemium device for your test run at Testdroid Cloud:

```
deviceFinder = None
testdroid_device = os.environ.get('TESTDROID_DEVICE')

deviceFinder = DeviceFinder(username=testdroid_username,
password=testdroid_password, url=testdroid_url)
if testdroid_device == "":
  # Loop will not exit until free device is found
  while testdroid_device == "":
    testdroid_device = deviceFinder.available_free_android_
    device()
```

Then, to choose correct automation backend for the server, you need to fetch the API level of the device which will be used for a test run:

```
apiLevel = deviceFinder.device_API_level(testdroid_device)
```

Finally, let's take a look at desired capabilities configuration with your Python script.

```
desired_capabilities_cloud = {}
desired_capabilities_cloud['testdroid_username'] = testdroid_
username
desired_capabilities_cloud['testdroid_password'] = testdroid_
password
if apiLevel > 17:
  desired_capabilities_cloud['testdroid_target'] = ,Android'
else:
  desired_capabilities_cloud['testdroid_target'] =
  'Selendroid'

desired_capabilities_cloud['testdroid_project'] = os.environ.
get('TESTDROID_PROJECT') # eg.'Demo'
desired_capabilities_cloud['testdroid_testrun'] = os.environ.
get('TESTDROID_TESTRUN') # eg.'MyTest'
desired_capabilities_cloud['testdroid_device'] = testdroid_
device
desired_capabilities_cloud['testdroid_app'] = 'sample/
BitbarSampleApp.apk'
desired_capabilities_cloud['platformName'] = 'Android'
desired_capabilities_cloud['deviceName'] = 'Android Phone'
desired_capabilities_cloud['appPackage'] = 'com.bitbar.
testdroid'
desired_capabilities_cloud['appActivity'] =
'.BitbarSampleApplicationActivity'
```

One more thing! Let's initiate and set up WebDriver:

```
desired_caps = desired_capabilities_cloud;
self.driver = webdriver.Remote(appium_url, desired_caps)
```

Tips!

You can set Environment Variables to store all the desired capabilities that won't be changing often. This logic is already included in the sample scripts so all you need to do is set the variables! For example, to set your Testdroid Cloud username, simply apply `TESTDROID_USERNAME=user@testdroidcloud.com` to your environment and the script will then find it with the `os.environ.get()` function. If the variable doesn't exist, the script will move on to the next declaration after the `'or'` key word.

Playing around with the Android appPackage and appActivity values may be useful when creating tests to automate specific activity screens of your app. There is also a command to launch another app while one is already being tested! This functionality is available for Android only though.

Chapter

03

Java Integration with Real Devices on Cloud Service

Now it's time to move forward and see how things are getting done on Java. The beauty of Java is in its extensibility and long existing history. Most developers these days have at least some experience in Java, which makes it easy to approach. You should check the Appium Java-client README, if you haven't already!

Before you attempt to launch the Testdroid Java samples, remember to check that all the desired capabilities are correct. It may be a good idea to also set as many capabilities in Environment Variables as possible to reduce the amount of editing when creating additional test scripts. Frankly, this will save a lot of time and also makes your test run launching a way more robust (no typos or flaky issues with the script config).

> *Tip! Put as many desired capabilities in environment variables as possible. It saves time and reduces errors due wrong configuration or typos in desired caps.*

Similar to the Python sample, our Java implementation uses the Environment Variables for capabilities such as `testdroid_username`, `testdroid_password`, app file location and Appium server:

```
private static final String TARGET_APP_PATH = "../../../
apps/builds/BitbarSampleApp.apk";

private static final String TESTDROID_SERVER = "http://
appium.testdroid.com";
```

```
String testdroid_username = env.get("TESTDROID_USERNAME");
String testdroid_password = env.get("TESTDROID_PASSWORD");
```

Notice how we create the test class by extending to another class called BaseTest.java:

```
public class SampleAppiumTest  extends com.testdroid.
appium.BaseTest.
```

Thanks to this extend, we get to use some of our self created code behind the scenes of the test class.

Let's Take a Closer Look at the Basetest.java

Since we have both Android and iOS samples combined in the same Java project, we decided to create the convenience class called BaseTest.java. The class includes functions for both application upload to Testdroid Cloud and taking screenshots. Appium java-client's screenshot implementation is based on Selenium Webdriver's TakesScreenshot interface, which is basically too cumbersome to write multiple times.

While writing your own tests, extending to use such a convenience class can be a real time saver.

Typically our samples are built for Maven build automation tool. This tool is really handy as it automates downloading all required dependencies before building and executing the project. All the required dependencies and other project settings are described in the `pom.xml` file.

Running Tests in Testdroid Cloud

You can run test from your IDE or directly from command line using maven:

```
>mvn clean test -Dtest=SampleAppiumTest
```

or to be more precise:

```
>mvn clean test -Dtest=com.testdroid.appium.android.sample.
SampleAppiumTest
```

or run all the tests:

```
>mvn clean test
```

More Tips!

When you have more than one test case to run, you should be aware of the fact that every test session to Testdroid Cloud can use only one instance of the Appium Webdriver. If you use driver.quit(), the current session towards the cloud will also end.

In other words, bundling multiple test cases to one test run session requires you to keep using the same webdriver instance in each of your test cases. To make sure each of your test case starts off from a clean slate, take a look at the driver command resetApp.

If you're unsure of your possibilities with the java-client, you may find some ideas by checking the Appium Java-client API documentation.

Chapter

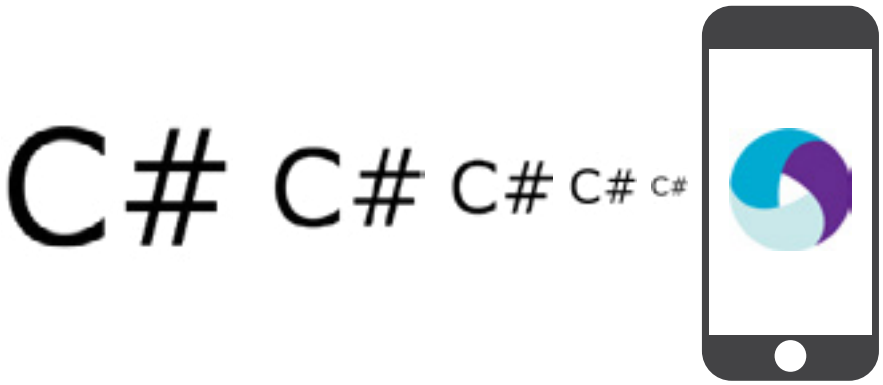
04

Mastering C# for Your Tests, Setup and Some Basics

Next we'll look at C# (or csharp). In short, C# is a multi-paradigm programming language and was created to be as simple, modern and general-purpose as possible, not forgetting the object-oriented approach as a programming language.

Similar to the syntax of Java, C# is a widely used language for all kinds of programming and scripting needs. The primary reasons to choose C# as THE scripting language for Appium tests are those benefits and advantages also close to what Java programmers have. The recommended full IDE for C# scripting is Visual Studio for Windows and or if you use MAC/UNIX there are other options available as well. Anyway, we recommend using an IDE, as having any type of smart auto-completion mechanism with proper listings of available methods is always a win and makes your day easier.

For Visual Studio you should make sure that the NUnit Test Adapter is installed through the Extension Manager. In order to install the NUnit Test Adapter in the Extension Manager, select **Tools** › **Online Extensions** (panel on left) and then locate NUnit Test Adapter and highlight it.



For running those tests, you can use Test Explorer. Using UNIX, first download the dependencies of the solution by following these steps:

1. Download All Dependencies - With One Command

We use NuGet to install a package using specific resources. Using the Testdroid example, you have a folder "Test123" that includes packages.config. The following components will get fetched:

```
<packages>
<package id="Appium.WebDriver" version="1.3.0.1"
targetFramework="net45" />
<package id="Newtonsoft.Json" version="7.0.1"
targetFramework="net45" />
<package id="NUnit" version="2.6.4" targetFramework="net45"
/>
<package id="Selenium.Support" version="2.46.0"
targetFramework="net45" />
<package id="Selenium.WebDriver" version="2.46.0"
targetFramework="net45" />
</packages>
```

To get these components installed, use the following command line:

```
> nuget install Test123/packages.config -OutputDirectory
packages
```

Now, all required bits and pieces are downloaded, installed and you are ready to build the package.

2. Build Test Package

Use the following command to build the test package for your local setup:

```
> xbuild
```

3. Run Your Tests

Then, use the following command to run recently built test package:

```
> nunit-console Test123/bin/Debug/TestdroidAndroidSample.dll
```

What About Those Test Results?

When you begin your Appium testing with Testdroid, you may notice that by default the portal doesn't show detailed information regarding your test run successes and failures. The reason for this is, of course, that you need to upload the results first!

Why is this so? During a test run, Appium server doesn't keep count of succeeding and failing test cases. To put it simply, the server's job is to take in the commands from the client, execute those commands on the connected mobile device and finally report back to client the results of those commands. Any test suite formatting, reporting and all that hassle is all handled by the client. So any test results will only exist on the client side.

Let's take a look of how you can upload those test details to cloud to ensure everything is shown properly in the right context.

Uploading Your Test Results XML to Testdroid Cloud

From our help pages, the test results upload goes like this:

Uploading your Test Suite output to Cloud

If your test suite generates a JUnit XML results file, you can upload the XML to Testdroid Cloud. When you do this, all your test cases and their statuses will be shown on the Testdroid Cloud testrun view and after this, the complete test report is downloadable. Couple of good tips to ensure everything goes fine:

- Add the testdroid_junitWaitTime Desired Capability in your TestScript.
- Get Appium sessionId from your script using <code>driver.session_id</code>. Note that this only can be done after the WebDriver connection has been properly established.
- After your test run has finished, and JUnit XML has been generated, use `cURL` to upload the XML to Testdroid Cloud:

```
curl -s -F result=@"/absolute/file/path/TestOutput.
xml" "http://appium.testdroid.com/upload/
result?sessionId=<sessionId>"
```

But wait, Webdriver in C# Doesn't Seem to Have Driver.Session_ID

Yikes! That's what I was wondering too while I was preparing this post. After some digging though, I found a way to print out the session id.

Fortunately this StackOverflow post has a well working solution to make the session_id

value available. Here what you should do – first add the function for session_id:

```
public class CustomRemoteDriver : RemoteWebDriver
{
  public CustomRemoteDriver(Uri remoteAddress, ICapabilities
  desiredCapabilities)
    : base(remoteAddress, desiredCapabilities, TimeSpan.
    FromSeconds(400))
    {
    }
  public String getSessionId()
  {
    return this.SessionId.ToString();
  }
}
```

Then, you should add <code>testdroid_junitWaitTime</code> in desired capabilities:

```
capabilities.SetCapability("testdroid_junitWaitTime", 120);
...
driver = new CustomRemoteDriver(new Uri("http://appium.
testdroid.com/wd/hub"), capabilities);
```

Finally, use the following line to print out the session id:

```
Console.WriteLine (driver.getSessionId());
```

These edits that I did to print out the session id can be also found here.

So Far So Good.. And Tips

One last thing to consider is that Testdroid Cloud expects the test results xml file to be in JUnit format, as the desired capability leads to believe. NUnit and JUnit result files are very close to each other, so using some converter is more than possible. For example, here's an open sourced Jenkins plugin for NUnit reporting, which includes JUnit conversion as one of its functionalities.

Great! I hope you learnt something about how to get in started or if you use it already, make things working smoother with your Appium and Testdroid Cloud combination on C#.

Chapter

05

Rumble with Ruby

Ruby has become popular in past years as a similar scripting language to Python and Ruby is a dynamic, reflective, object-oriented general-purpose programming language using in variety of different purposes.

In a nutshell, Ruby is a language comparable to Python in its simplicity and clearness, although it does require a bit more investment of time to learn. Once you get used to it though, Ruby can become a very powerful tool due to its well defined and robust design.

Getting Started

First and the foremost, you should install Bundler to ensure all necessary gems and dependencies gets installed properly. Then update your test script (`testdroid_*.rb`) with necessary information. Always do remember to modify your desired capabilities before you start a test. This will always save you time and effort as you don't have to run a test to realize there was something wrong with desired capabilities.

Ok, here is how to install:

```
$ gem install bundler
$ bundle install
```

Launch a test:

```
$ rspec testdroid_android.rb
```

NOTE! There are some common settings that you need to set in all scripts regardless of the app type that you are testing. Each `testdroid_*.rb` file needs to be updated with the following values.

Here are all the values that you need to configure:

- `screen_shot_dir` – where should screenshots be stored on your local drive
- `testdroid_username` – your email that you registered in Testdroid Cloud
- `testdroid_password` – your Testdroid Cloud password
- `testdroid_project` – has a default value, but you might want to add your own name to this. Project name is visible in your project view in Testdroid Cloud. Each project must have a unique name
- `testdroid_testrun` – name or number of this test run. Test run names can be modified even at every test run
- `testdroid_app` – should be the name of the app you uploaded to cloud. Eg. if you

uploaded your app using a script this would look something like this: `'f4660af0-10f3-46e9-932b-0622f497b0d2/Testdroid.apk'` If you uploaded your app through the Testdroid Cloud web UI, you can use here the value 'latest' that refers to the latest uploaded app file.

Native iOS Specific Settings

Example script: testdroid_appiumdriver_ios.rb

To run your Appium tests against your native iOS application with real devices you need to edit the testdroid_appiumdriver_ios.rb script.

In addition to the above mentioned Appium capabilities for iOS testing you need set:

- `bundleId` – this is your application's unique name

Two Tips for Ruby on Appium

(1) You can launch your Appium test to use the latest app file you uploaded through Testdroid Cloud portal by setting the testdroid_app desired capability value to `'latest'`.

(2) To automate device finding from Testdroid cloud, you could check this Ruby tool contributed by bootstraponline. The tool is actually a port from our Python device_finder.py script based on the Testdroid Cloud API. The API can be utilized to do basically all the things our Portal allows you to do, and then a bit more! Documentation for the API can be found here.

Chapter

06

Jazzing Javascript with Node.js

JavaScript has become a de-facto programming language in web-based applications and it is widely and commonly used in other software domains, such as in game development and naturally server-side network programming, as well. Appium can be run with JavaScript through Node.js – thanks to different testing frameworks such as Mocha and Nightwatch.

Getting Started

There is one awesome example (with some source code) by Aaron Colby using Nightwatch framework and this example provides an insight on how to launch Safari browser and use your tests on navigating to a specific website. Nightwatch is the framework for browsers and web apps specifically. Take a look at the example first – it's available at https://github.com/acolby/Testdroid_Example.

To recap some of the essential things from Aaron's example and some further things to consider:

Add Dependencies

1. npm dependencies

```
$ npm install
```

2. install mocha globally

```
$ npm install -g mocha
```

Add Credentials to ./.Creds.json

You will need to add an object `testdroid` to the file `./.creds.json`, as it is .gitignored:

```
{
  "testdroid": {
    "username": "TESTDROID_USERNAME",
    "password": "TESTDROID_PASSWORD"
  }
}
```

› Run the Test

```
$ mocha ios_safari.js
```

All dependencies MUST be installed as instructed in the readme file. Then, create a new file called .creds.json and add your Testdroid credentials as illustrated and exampled above.

You also have to create a directory named as "`temp`" for all screenshots that will be taken during the test session. Explore the `ios_safari.js` file to see how the test itself is built. In order to make the device finder look for freemium devices, add the `creditsPrice` value as 0 (or 1 for premium devices) to the criteria, as follows:

```
if(devices.data[i].locked === false && devices.data[i].
creditsPrice === 0)
{
  device = devices.data[i];
  i = devices.data.length;
}
```

Example of Appium for Mocha Based Node.js Tests

Let's use the earlier sample as a base to create a native iOS app test next! The difference here is that we won't be needing the Nightwatch framework at all.

DESIRED CAPABILITIES

Edit the desired capabilities, namely the testdroid_target, testdroid_app and bundleId. Also, make sure to remove the 'browser' capability.

```
desired.testdroid_username = creds.testdroid.username;
desired.testdroid_password = creds.testdroid.password;
desired.testdroid_target = 'ios';
desired.testdroid_project = 'Test iOS Mocha';
desired.testdroid_testrun = 'TestRun A';
desired.testdroid_device = device.displayName;
desired.platformName = 'iOS';
desired.deviceName = 'iOS Device';
desired.testdroid_app = 'sample/BitbarIOSSample.ipa';
desired.bundleId = 'com.bitbar.testdroid.BitbarIOSSample';
```

Using `testdroid_app = ,sample/BitbarIOSSample.ipa'` you can default to the pre-existing sample app which is always available in cloud! For android the same would be `'sample/BitbarSampleApp.apk'`.

Test Case

Let's edit the test case to do something with the native sample app, instead of the safari targeted commands:

```
it("should set text to 'Tester'", function() {
return driver
.sleep(4000)
.saveScreenshot(SCREEN_SHOT_PATH + '/test.png')
.sleep(4000)
.elementByClassName('UIATextField')
.click()
.keys('Tester')
.saveScreenshot(SCREEN_SHOT_PATH + '/test2.png')
.hideKeyboard('Return')
.sleep(4000)
.saveScreenshot(SCREEN_SHOT_PATH + '/test3.png')
.sleep(4000)
});
```

You may of course change the name of the `ios_safari.js` file to something that makes more sense, like `ios_native.js` or so. The test case function `describe("ios safari", function() {` should probably be renamed at this point too.

Documentation

For the Selenium based driver commands, best place to refer would be the Selenium Webdriver's Node.js documentation's full jsonwire mapping.

Chapter

07

Mastering Appium Through the Command Line

Now let's look how to use command line for your Appium tests and testing. To get this to work, you need to have everything prepared beforehand.

Get the Latest Version of Appium

To get the latest version of Appium, you should pull the source code from Github and then compile it on your designated host machine. Appium is provided with a script to handle the build process, although it does require some prepping when doing so for the first time. Once all pre-requisites are found on your system, building shouldn't be a problem from there on.

PREREQUISITES FOR APPIUM

First, depending on what you intend to automate it's probably easiest to see if your machine already has all the requirements installed. You can try this by navigating to Appium directory and running the `appium-doctor.js` script found at the bin directory. If you don't intend to automate both iOS and Android devices, you can safely ignore any warnings of the platform you don't need.

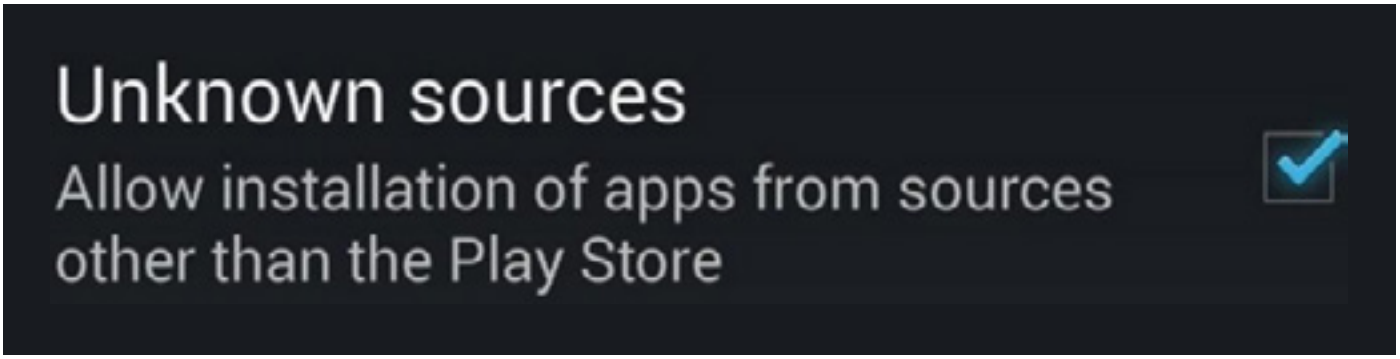
Here in Testdroid when we want to build Appium on a Mac device to support all the usual automation needs for both iOS and Android devices, we use this command:

```
$ ./reset.sh --android --selendroid --ios --real-safari
--verbose
```

The `--verbose` flag is important, if you suspect that you might be missing any dependencies as it shows detailed information on the build process.

Prepping the Test Devices

Once you have Appium server built up and ready to go, the next step is to connect your test device(s) to the machine and accept any trust dialogs the devices may show. On Android devices you need to enable USB debugging in the Developer options and also allow the device to install APK files from "unknown sources".



iOS devices require UI Automation to be enabled in the Developer settings, which should be visible as long as you can see the device in Xcode Devices window (cmd+shift+2 on Mac). To make sure your device is activated for test automation, you should also open up Instruments from the Xcode and select the devices you have as active devices.

Using Appium from Command Line

To launch Appium, use the `appium.js` script found in the bin directory to fire up the server. For iOS you will need to add the `-U`, or `--udid`, flag with the udid of your target device.

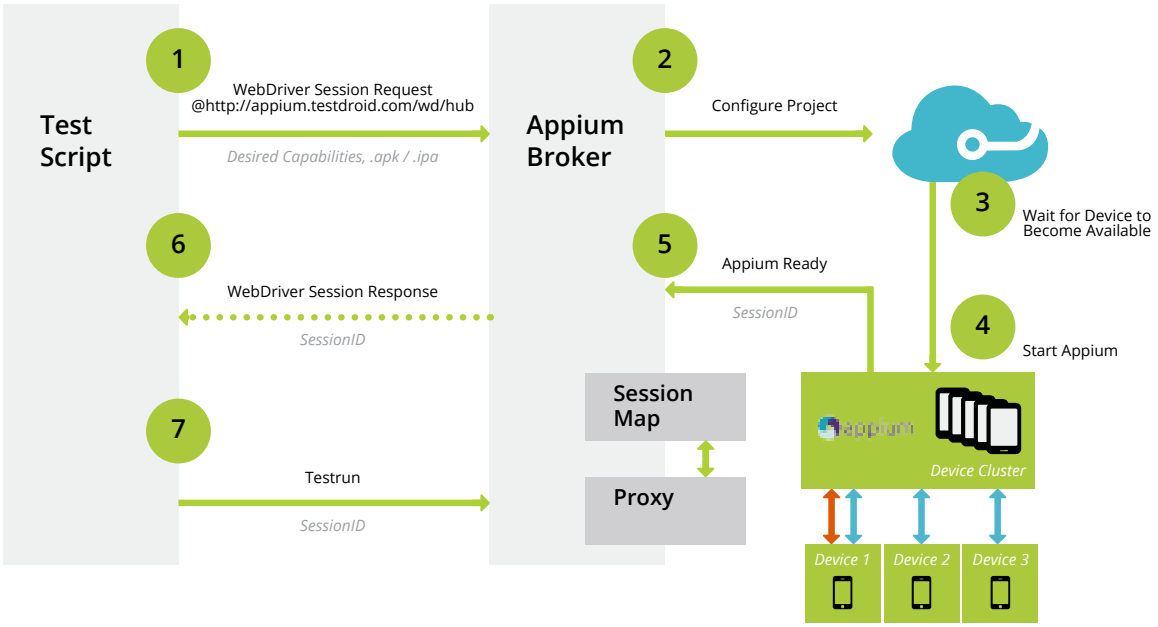
If you want to automate multiple devices at the same time, simply fire up multiple instances of the Appium server, all with different ports in use. With multiple devices, you need to set `--port` flag for the actual Appium port (default 4723) and a `--bootstrap-port` for communication with the different Android devices. Selendroid and ChromeDriver also need their dedicated ports when automating multiple devices so don't forget to set `--selendroid-port` and `--chromedriver-port` as well, if you need Selendroid or Chrome.

You should note that no more than one iOS device can be automated in a single host machine due to restrictions in Xcode.

Launching Tests Toward Your Server

If the server runs in your localhost, and in default port, you can set the webdriver to initialize towards `http://localhost:4723/wd/hub`. If you have multiple servers running, simply make sure the IP address and port number are matching to the server you want to use.

Behind the Scenes



In Testdroid Cloud we have developed our very own Appium Broker, which handles the communications of all on-going sessions between clients and Appium servers. In other words, we handle the creation of all the Appium server instances with their dedicated ports completely on the cloud side so that all you need to do is launch your tests towards `http://appium.testdroid.com/wd/hub`.

Conclusion

There's a myriad of different ways you can setup, build and run your Appium tests. The ones provided here are just one example of how things can be done.

For Appium beginners, we hope this ebook has shedded some light on the basics of Appium, including the definition of Appium, the supported programming languages, etc. And for those who have been using Appium many years, the ebook has consolidated your understanding of this open source and provided more useful tips.