EuroSTAR
eBook Series
JUNE 2018

Appium Tips & Tricks: Mobile Application Testing Made Easy

Nishant Verma

EuroSTAR Software Testing | Huddle

Skill Level

This eBook is for an audience of all skill levels.

Abstract

Appium is the most widely adopted mobile test automation tool. The community support has been vibrant, however, there is a lack of a structured and step-by-step guide or documentation around building a framework. The book is an attempt to bridge that gap and serve as a handheld guide to each tester who wants to build his or her own mobile test automation framework from scratch.

It is intended towards developers and testers who want to learn mobile app testing and automation using Appium. The book takes you through the journey of understanding Appium and slowly get you started with test automation ecosystem. Cucumber is one of the most promising technologies and rising in popularity due to wide adoption of Agile and Behavior Driven Development methodology.

This book introduces you to the concept of Cucumber and how you can build your own testing framework in Cucumber and Appium from scratch. It contains example code snippets right from creating a sample project, write first Appium tests, evolving test framework, setting up Jenkins etc.

The full book can be purchased from Amazon here.

About The Author

Nishant Verma is a Co-founder of TestVagrant Technologies living in Bangalore, India.

Nishant has more than 12 years of experience in testing and development where he has worked on a variety of tools and technologies. He has hands on experience with test automation tool like WebDriver, Calabash, Frank, Appium, HP UFT (formerly QTP), and DeviceAnywhere. He specializes in mobile testing and automation with open source tool Appium.

He has successfully built and delivered mobile test automation solution using Appium with various clients in different business domains.
At TestVagrant, he is instrumental in building one of the most advanced and intelligent solutions in mobile testing space. Optimus is the flagship-testing product of the company that addresses the lack of standard framework in mobile test automation space. Optimus is available to download for free from the company site.

He actively maintains his own website on testing techniques, agile testing, automation techniques and general learning.

Twitter: @nishuverma
www.nishantverma.com

11
Appium Tips and Tricks

In the last chapter, we looked at how to set up Jenkins and have a test run in an automated way. We also learned how to put the code into GitHub and then configure the Jenkins task for the purpose. We have almost come to the end of this book; in this chapter, we will learn some tips and tricks of Appium and automation in general, which can help improve our test automation and make it a little more intelligent both from the system and testing points of view.

In this chapter, we will take a detailed look at the following:

- Switching between WebView and Native
- Taking screenshots
- Recording video execution
- Interaction with an other app
- Approach for running the test in parallel
- Simulating various network conditions

Switching between views - web and native

While testing an app, we often find the need to switch between the Web and native views. A typical example is the Facebook sign-in page in many apps or an intermediate payment page. In those situations, we need to change the application context to `WEBVIEW` or `NATIVE`. Use the following code snippet to switch to WebView:

```
public static void changeDriverContextToWeb(AppiumDriver driver) {
    Set<String> allContext = driver.getContextHandles();
    for (String context : allContext) {
        if (context.contains("WEBVIEW"))
            driver.context(context);
        }
    }
```

It tries to get a list of all the context handles, checks whether there is any context that contains WebView, and then the driver switches to that context.
The following code snippet switches to native on a similar logic:

```
    public static void changeDriverContextToNative(AppiumDriver driver) {
        Set<String> contextNames = driver.getContextHandles();
        for (String contextName : contextNames) {
            if (contextName.contains("NATIVE"))
                driver.context(contextName);
        }
    }
```

Generally, switching between a WebView and native view happens across the app on different pages, so it will make more sense to have this method created in BasePage. The advantages of this approach are as follows:

- Easy access to call from any page
- Avoid duplication of the implementation

We can use the preceding code for reference and may tweak it, if need be. The next tip is taking a screenshot of the app while under execution.

Taking screenshots

A picture speaks a thousand words, but in our case it can save a thousand seconds. It's a good practice to take an image at the point of test failure as it will help us save a lot of time, which is needed to go through the error logs. Also, sometimes images are needed as part of the test case itself. Here are two approaches:

- Embedding a snapshot at the point of failure
- Taking a screenshot and saving it for later use or reference

Embedding a snapshot in a cucumber report becomes fa+irly easy. Cucumber exposes you to the `Scenario` interface, which makes it slightly easier to query whether the scenario has failed or passed. For example, refer to the following snapshot of code; we are doing the following step by step:

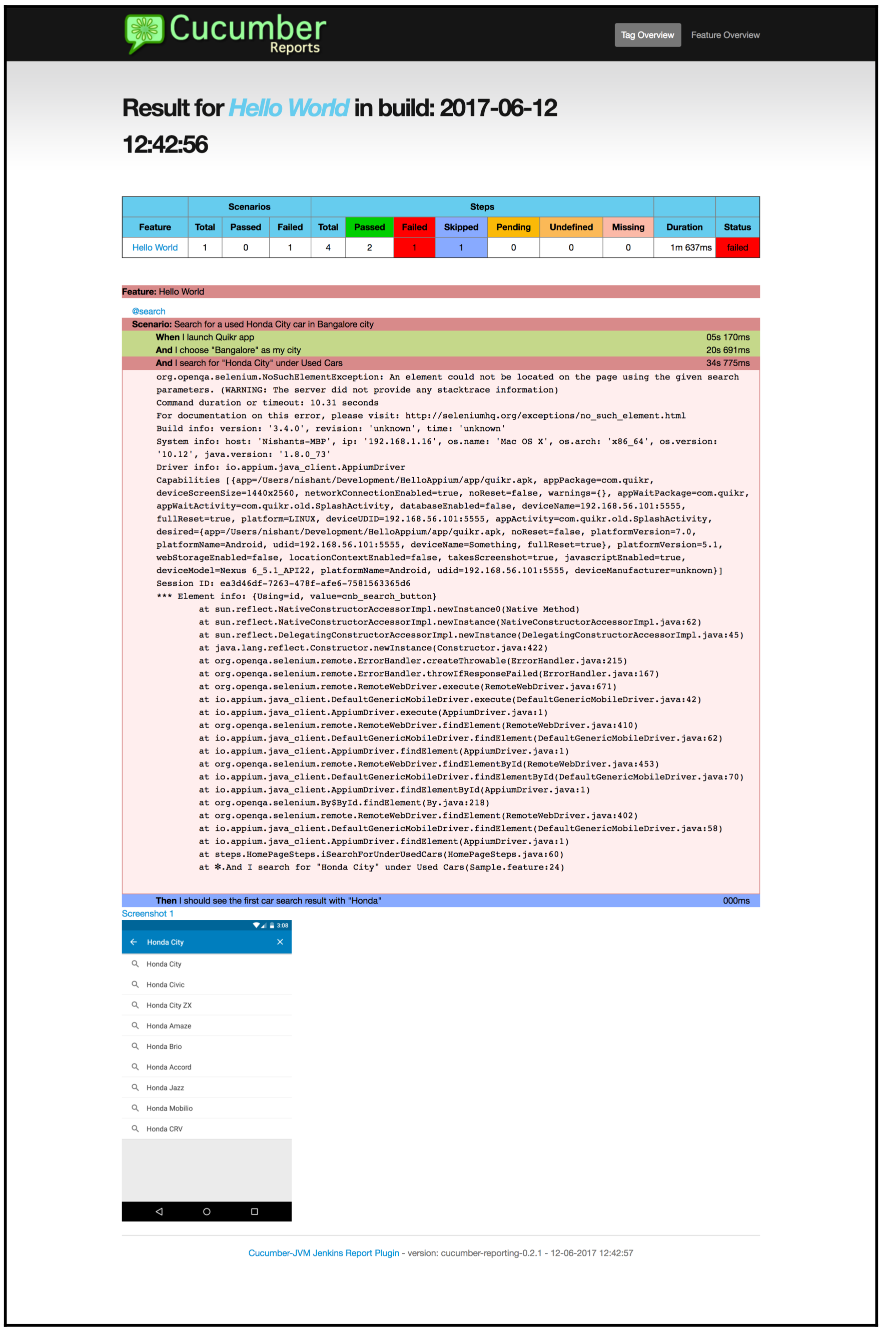
- The conditional statement helps us check whether the scenario has passed or failed
- We are checking for a failure condition in respect of the scenario
- We instruct the driver instance to take a screenshot at the point of failure:

```
if (scenario.isFailed()) {
    final byte[] screenshot = driver
            .getScreenshotAs(OutputType.BYTES);
    scenario.embed(screenshot, "image/png");
}
```

We can include the preceding code in the tear-down method. So, this will keep probing the scenario and, if it fails, it will take a screenshot and embed it in cucumber reports. If we edit the current tear-down method, it will be as shown below:

```
@After
public void tearDown(Scenario scenario) {
    try {
        if (scenario.isFailed()) {
            final byte[] screenshot = appiumDriver
                    .getScreenshotAs(OutputType.BYTES);
            scenario.embed(screenshot, "image/png");
        }
        appiumService.stop();
        appiumDriver.quit();
    } catch (Exception e) {
        System.out.println("Exception while running Tear down :"
        + e.getMessage());
    }
}
```

When we embed the failure snapshot in the current test report, it becomes more informative. Here's how a sample report with image embedding will look:



Result for *Hello World* in build: 2017-06-12 12:42:56

	Scenarios			Steps								
Feature	Total	Passed	Failed	Total	Passed	Failed	Skipped	Pending	Undefined	Missing	Duration	Status
Hello World	1	0	1	4	2	1	1	0	0	0	1m 637ms	failed

Feature: Hello World

@search

Scenario: Search for a used Honda City car in Bangalore city

When I launch Quikr app 05s 170ms

And I choose "Bangalore" as my city 20s 691ms

And I search for "Honda City" under Used Cars 34s 775ms

```
org.openqa.selenium.NoSuchElementException: An element could not be located on the page using the given search
parameters. (WARNING: The server did not provide any stacktrace information)
Command duration or timeout: 10.31 seconds
For documentation on this error, please visit: http://seleniumhq.org/exceptions/no_such_element.html
Build info: version: '3.4.0', revision: 'unknown', time: 'unknown'
System info: host: 'Nishants-MBP', ip: '192.168.1.16', os.name: 'Mac OS X', os.arch: 'x86_64', os.version:
'10.12', java.version: '1.8.0_73'
Driver info: io.appium.java_client.AppiumDriver
Capabilities [{app=/Users/nishant/Development/HelloAppium/app/quikr.apk, appPackage=com.quikr,
deviceScreenSize=1440x2560, networkConnectionEnabled=true, noReset=false, warnings={}, appWaitPackage=com.quikr,
appWaitActivity=com.quikr.old.SplashActivity, databaseEnabled=false, deviceName=192.168.56.101:5555,
fullReset=true, platform=LINUX, deviceUDID=192.168.56.101:5555, appActivity=com.quikr.old.SplashActivity,
desired={app=/Users/nishant/Development/HelloAppium/app/quikr.apk, noReset=false, platformVersion=7.0,
platformName=Android, udid=192.168.56.101:5555, deviceName=Something, fullReset=true}, platformVersion=5.1,
webStorageEnabled=false, locationContextEnabled=false, takesScreenshot=true, javascriptEnabled=true,
deviceModel=Nexus 6_5.1_API22, platformName=Android, udid=192.168.56.101:5555, deviceManufacturer=unknown}]
Session ID: ea3d46df-7263-478f-afe6-7581563365d6
*** Element info: {Using=id, value=cnb_search_button}
        at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
        at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:62)
        at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)
        at java.lang.reflect.Constructor.newInstance(Constructor.java:422)
        at org.openqa.selenium.remote.ErrorHandler.createThrowable(ErrorHandler.java:215)
        at org.openqa.selenium.remote.ErrorHandler.throwIfResponseFailed(ErrorHandler.java:167)
        at org.openqa.selenium.remote.RemoteWebDriver.execute(RemoteWebDriver.java:671)
        at io.appium.java_client.DefaultGenericMobileDriver.execute(DefaultGenericMobileDriver.java:42)
        at io.appium.java_client.AppiumDriver.execute(AppiumDriver.java:1)
        at org.openqa.selenium.remote.RemoteWebDriver.findElement(RemoteWebDriver.java:410)
        at io.appium.java_client.DefaultGenericMobileDriver.findElement(DefaultGenericMobileDriver.java:62)
        at io.appium.java_client.AppiumDriver.findElement(AppiumDriver.java:1)
        at org.openqa.selenium.remote.RemoteWebDriver.findElementById(RemoteWebDriver.java:453)
        at io.appium.java_client.DefaultGenericMobileDriver.findElementById(DefaultGenericMobileDriver.java:70)
        at io.appium.java_client.AppiumDriver.findElementById(AppiumDriver.java:1)
        at org.openqa.selenium.By$ById.findElement(By.java:218)
        at org.openqa.selenium.remote.RemoteWebDriver.findElement(RemoteWebDriver.java:402)
        at io.appium.java_client.DefaultGenericMobileDriver.findElement(DefaultGenericMobileDriver.java:58)
        at io.appium.java_client.AppiumDriver.findElement(AppiumDriver.java:1)
        at steps.HomePageSteps.iSearchForUnderUsedCars(HomePageSteps.java:60)
        at ✽.And I search for "Honda City" under Used Cars(Sample.feature:24)
```

Then I should see the first car search result with "Honda" 000ms

Screenshot 1

Cucumber-JVM Jenkins Report Plugin - version: cucumber-reporting-0.2.1 - 12-06-2017 12:42:57

To get the above report or a nicely formatted cucumber report, we can use an external JAR listed here: `https://github.com/damianszczepanik/cucumber-sandwich`.

The second use case for taking a screenshot is to use it for manual verification. For instance, a use case would help UX team give a page by page snapshot of the app to verify the look and feel. We can use the described `getScreenshotAs()` method to taking the screenshot and store the output as a file in some predefined path. The format we are using is `.jpg`:

```
public void getScreenshot(String imageFolder) throws IOException {
    File srcImgFile=driver.getScreenshotAs(OutputType.FILE);
    String filename= UUID.randomUUID().toString();
    File targetImgFile=new File(imageFolder + filename +".jpg");
    FileUtils.copyFile(srcImgFile,targetImgFile);
}
```

Having a large number of screenshots at different points in the execution and publishing them as part of build artefacts might eat up the Jenkins agent space (assuming that the Jenkins slaves are less powerful and scaled down machine versions). We should be careful with this feature.

The next tip is to record the video execution of scenarios.

Recording video execution

Often, there is an inherent need to capture the playback when we execute a test so that we can actually see how the scenario fared. There can be a few reasons for this, one of which is the documentation. It might also be for demonstration purposes in the product team, or to see what happened on the device in the case of any failure.

Android ADB gives screen recording functionality only and not the audio capture. This should suffice for most functional test automation needs, which doesn't really require the audio component to be captured. ADB gives you a way to capture the display of Android devices, running Android 4.4 (API Level 19) or upward. The API is `adb shell screenrecord [options] <filename>`.

- Let's look at a usage example--`adb shell screenrecord /sdcard/demoVideo.mp4`:
    - The screen recording automatically stops after 3 minutes or by the `--time-limit` option, if set API usage for time limits--`adb shell screenrecord --time-limit <TIME_IN_SECONDS>`.

- The usage example for this is `adb shell screenrecord --time-limit 240`:
    - The screen record API gives the option to rotate the output by 90 degrees; however, this is just an experimental feature.
- API usage for rotate--`adb shell screenrecord --rotate`:
    - The screen record API gives the option to display log information. By default, this is off.
- API usage for displaying log info--`adb shell screenrecord --verbose`:
    - The screen record API gives the option of setting the bit rate for the video, in megabits per second. The default value is 4 Mbps. The higher the bit rate, the greater the size of video and vice versa.

API usage for bit rate--`adb shell screenrecord --bit-rate <RATE>`.

An example of this `adb shell screenrecord --bit-rate 6000000 /sdcard/demoVideo.mp4`.

A handy tip for recording video execution is to start the recording when you start the scenario; so an ideal place to call it would be in the `setup` method with the `@Before` tag. Also, adb makes only 3 minutes of screen recording; so if a scenario exceeds 3 minutes, we need to write our own logic to capture the remaining execution.

The next tip is about how we launch a different app when we have started a session with a specified app under test.

Interacting with another app

Most of the time, when we test a mobile application, it requires interaction with another app. For example, an app might need integration with the Contacts app or the SMS app. Sometimes, while testing, we might need to simulate the geo location, which can be done via an external app installed on the device/emulator (or it can even be done using Android `adb` commands).

When we start an Appium session for testing, generally it is tied to an app as we are passing the `app` parameter in the desired capabilities, so we can't really pass two apps in the desired capabilities. If we recall our code, we are using this line:

```
capabilities.setCapability("app",
"/Users/nishant/Development/HelloAppium/app/quikr.apk");
```

One way to switch between the apps is when we know the target app's **package name** and **activity name**. Android driver exposes a method, `startActivity(Activity activity)`, which basically takes an activity as input and starts it. So, a sample code snippet to start the Contacts app on a device will look like this:

```
Activity activity = new Activity("com.android.contacts",
".ContactsListActivity");
androidDriver.startActivity(activity);
```

Once we are done with the test steps we want on this app, we can use the BACK key to traverse back to the application under test:

```
androidDriver.pressKeyCode(AndroidKeyCode.BACK);
```

The `StartActivity()` method is available only for `androidDriver` and not for `AppiumDriver`. On iOS devices/simulators we can't automate two apps in one session due to a limitation from the Apple itself. The only way we can do this:

- Initiate a session 1
- Run through the steps for app 1
- Close session 1
- Start another session 2
- Run through the steps for app 2
- Close the session 2.

One thing we need to keep in mind is to set the Desired Capability `noReset` to be `true` while creating the driver instance.

Let's take a look at how we can run the test in parallel.

Running the test in parallel

Let's go back a bit and see what we used in `Chapter 4`, *Understanding Desired Capabilities*: the Refactoring -2 section. Here's the code snippet we used:

```
@Before
public void startAppiumServer() throws IOException {

    int port = 4723;
    String nodeJS_Path = "C:/Program Files/NodeJS/node.exe";
    String appiumJS_Path = "C:/Program
    Files/Appium/node_modules/appium/bin/appium.js";

    String osName = System.getProperty("os.name");

    if (osName.contains("Mac")) {
        appiumService = AppiumDriverLocalService.buildService(new
        AppiumServiceBuilder()
                .usingDriverExecutable(new File("/usr/local/bin/node"))
                .withAppiumJS(new File("/usr/local/bin/appium"))
                .withIPAddress("0.0.0.0")
                .usingPort(port)
                .withArgument(GeneralServerFlag.SESSION_OVERRIDE)
                .withLogFile(new File("build/appium.log")));
    } else if (osName.contains("Windows")) {
        appiumService = AppiumDriverLocalService.buildService(new
        AppiumServiceBuilder()
                .usingDriverExecutable(new File(nodeJS_Path))
                .withAppiumJS(new File(appiumJS_Path))
                .withIPAddress("0.0.0.0")
                .usingPort(port)
                .withArgument(GeneralServerFlag.SESSION_OVERRIDE)
                .withLogFile(new File("build/appium.log")));
    }
    appiumService.start();
}
```

We discussed the `ROBOT_ADDRESS` capability, but didn't use it then. This capability holds the key to have Appium tests run in parallel.
We can follow these steps to implement test parallelization for Appium:

1. Create a method to start Appium service by passing `port` and `udid` as parameters.

2. Once we parameterize the preceding, we can actually start as many Appium servers as we have devices connected. The following code takes `port` and `udid` as parameters, starts the Appium service, and ties it to a particular `port` and `udid`:

```
appiumService = AppiumDriverLocalService.buildService(new
AppiumServiceBuilder()
.usingDriverExecutable(new File("/usr/local/bin/node"))
.withAppiumJS(new File("/usr/local/bin/appium"))
.withIPAddress("127.0.0.1")
.usingPort(port)
.withArgument(GeneralServerFlag.ROBOT_ADDRESS, udid as String)
.withArgument(AndroidServerFlag.BOOTSTRAP_PORT_NUMBER,
(port + 2) as String)
.withArgument(SESSION_OVERRIDE)
.withLogFile(new File("build/${udid}.log")));

appiumService.start();
```

3. Create a method to read the output of the `adb devices` command:
    - Iterate the preceding method to start the Appium service for each `udid` (Android device connected)
    - Use the following method to read the output of the preceding command:

```
public List<String> attachedDevicesAndEmulators() {
    List<String> devices = new ArrayList<>();
    String line;
    StringBuilder log = new StringBuilder();
    Process process;
    Runtime rt = Runtime.getRuntime();
    try {
        process = rt.exec(new String[]
        {"adb", "devices", "-l"});
        BufferedReader stdInput = new BufferedReader(new
        InputStreamReader(
                process.getInputStream()));
        BufferedReader stdError = new BufferedReader(new
        InputStreamReader(
                process.getErrorStream()));


        while ((line = stdInput.readLine()) != null) {
            log.append(line);
            log.append(System.getProperty
            ("line.separator"));
```

```
                    }
                    while ((line = stdError.readLine()) != null) {
                        log.append(line);
                        log.append(System.getProperty
                        ("line.separator"));
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
                Scanner scan = new Scanner(String.valueOf(log));
                while (scan.hasNextLine()) {
                    String oneLine = scan.nextLine();
                    if (oneLine.contains("model")) {
                        devices.add(oneLine.split("device")[0].trim());
                    }
                }
                return devices;
            }
```

4. Create a method to build the desired capability based on the device UDID as the parameter
5. Create a properties file to save the mapping of tags and devices to pick at runtime
6. Create a method in Gradle to read from the properties file and run the test

For the preceding steps, you need to implement your own code.

Network conditioning

Mostly, we test a mobile app in a perfect condition of best and fast network; however, in reality the devices might be moving and the network may be fluctuating between Edge connections (2G), 3G, or even LTE. Sometimes, the automation test has to run at a lower data speed or even test some offline functionality.

Appium exposes the `driver.setConnection()` method, which can help in setting the network condition between WiFi, airplane, data, or none. Any of the following statements can be used, based on which data connectivity you want to set up:

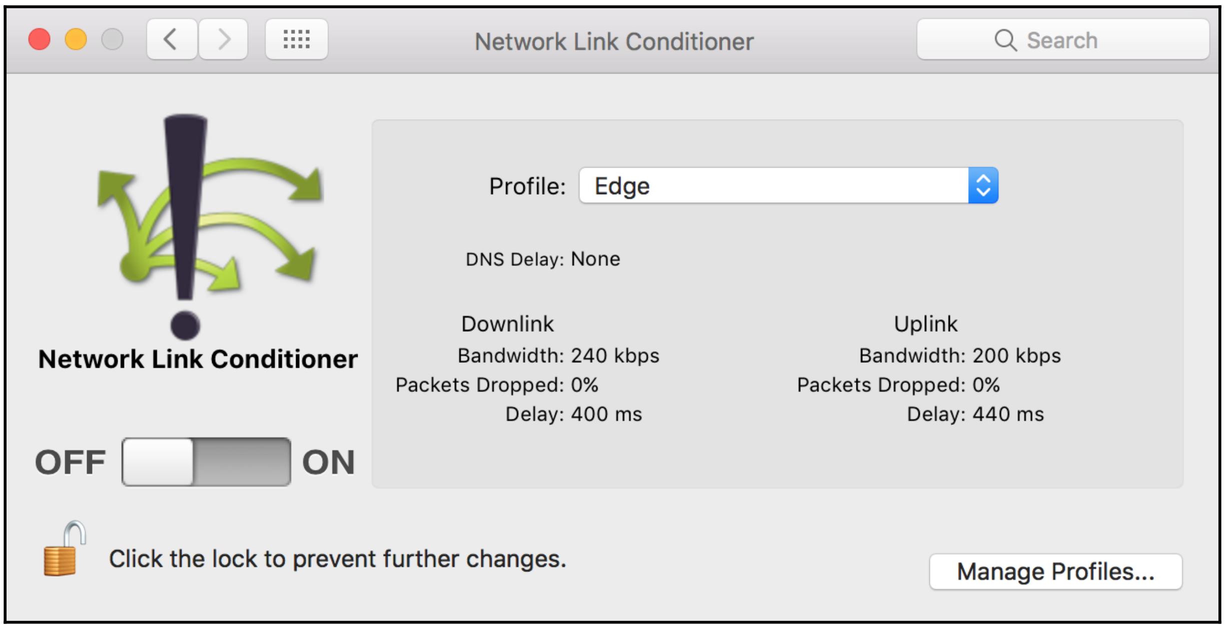
```
driver.setConnection(Connection.AIRPLANE);
driver.setConnection(Connection.WIFI);
driver.setConnection(Connection.DATA);
driver.setConnection(Connection.NONE);
driver.setConnection(Connection.ALL );
```

The connection is an `enum` that defines these bit masks:

Connection Type	Bit Mask
`NONE`	0
`AIRPLANE`	1
`WIFI`	2
`DATA`	4
`ALL`	6

Once the value is set, it persists for the life of the driver instance, so we must reset it back to the data connectivity we want for the test suite.

On macOS, one can install the **Network Link Conditioner** app to simulate the various network conditions. It can be downloaded as part of the Hardware IO tools package (for more information visit: `https://developer.apple.com/download/more/?q=Hardware%20IO%20Tools`). The following screenshot shows what the app looks like. This helps simulate the network speed on the simulator. One thing to keep in mind is that it impacts the hosting device network speed as well, so we have to be careful while using it:



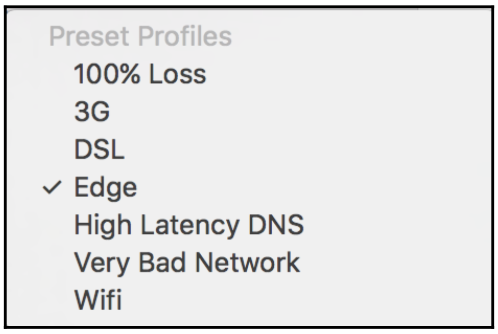
Profile: Edge

DNS Delay: None

Downlink
Bandwidth: 240 kbps
Packets Dropped: 0%
Delay: 400 ms

Uplink
Bandwidth: 200 kbps
Packets Dropped: 0%
Delay: 440 ms

Profile lets you select between different network speeds, such as low latency and 3G:



However, on a real iOS device, it's already built in and can be accessed by navigating to **Settings** > **Developer** > **Network link Conditioner**.

Summary

This chapter completes our journey learning mobile test automation with Appium. It took us on a tour where we understood the importance of mobile app testing and automation. We learned about the mobile testing ecosystem, how to set up a machine, and how to install the respective software and tools. We learned how to use the Appium app, find locators, and author tests. We also learned how to automate gestures and how to introduce synchronization in tests. We saw how to run these tests on devices and emulators, including setting up Genymotion emulators. We also discussed how to set up Jenkins and have tests automated when the source code is checked in Github.

Lastly, in this chapter, we learned some Appium tricks for switching between WebView and Native, taking screenshots, and embedding them in the report. We explored how to record the test execution device screen and also how to vary the quality of the recording. We learned how to interact with other apps and traverse back to the app under test. We learned the approach for parallel test execution and how to implement it. We also learned how to simulate the various network conditions to simulate 2G, 3G, or LTE conditions on the device while running the functional test.

With this knowledge, we are good to go out, set up our own automation framework from scratch, and drive it to solve our testing needs. I wish good luck and happy learning to you all!

Enjoyed this eBook and want to read more?
Check out our extensive library on Huddle.

Click Me To Visit
Huddle

EuroSTAR Software Testing | Huddle

www.eurostarsoftwaretesting.com