APPIUM STEP-BY-STEP GUIDE

MAKING THE MOVE TO AUTOMATION TESTING WITH APPIUM

Including viewpoints from industry leaders Paul Grizzaffi, Jonathan Lipps, Joe Colantonio, Mush Honda and Patrik Patel

Making the move to automation testing with Appium

A COMPREHENSIVE INTRODUCTION TO MOBILE TEST AUTOMATION

Kobiton, Inc.
Atlanta, GA

www.kobiton.com
marketing@kobiton.com

Ordering Information:
Printed copies or Quantity sales. Printed versions of this book are available. Contact us for details Special discounts are available on quantity purchases by corporations, associations, and others. For details, contact the us at the address above.

Making the Move to Automation Testing with Appium — 1st ed.
ISBN Pending
Technical/Testing

Table of Contents

Forward

In recent years the prevalence of mobile testing has increased dramatically, just like we expected it would from observations of the mobile ecosystem. Mobile test automation is still a young art, and there are many pitfalls in pursuit of useful and reliable automation systems.

There is therefore a huge need at the present time for comprehensive instruction, especially for those who are transitioning to mobile testing from web or desktop testing.

While there are an overwhelming number of similarities between these practices, the world of mobile testing comes with its own prerequisites for understanding, and these are often ignored by newcomers.

This book is a welcome addition to the expanding set of resources available to the new mobile tester.

-- **Jonathan Lipps, Founding Principal, Cloudgrey.io**

About: Jonathan Lipps has been making things out of code as long as he can remember. Jonathan is an Appium lead maintainer and architect and founding principal of Cloud Grey, the mobile automation consultancy. Before founding Cloud Grey, he was Director of Open Source at Sauce Labs. He has worked as a programmer in tech startups for over a decade, but is also passionate about academic discussion in various fields. Jonathan has master's degrees in philosophy and linguistics, from Stanford and Oxford respectively. Jonathan lives in Vancouver, Canada.

Welcome

Mobile test automation is more important than ever. Companies have embraced a "mobile-first" philosophy, and customers are more demanding than ever. Delivering **perfect mobile experiences is no longer optional. It's imperative.**

This makes it an exciting time to be a mobile test automation engineer. They're in high-demand and often working on high-visibility projects.

Appium is by far the most used and widely adopted test automation framework for mobile. And for good reason. Its maturity and open-sourced approach has led to a large community of Appium users who continue to improve the product.

It is our hope that this book will serve as a comprehensive guide, taking you from Appium Newbie to Appium Expert.

As with any undertaking as large as writing this guide, we anticipate errors slipping through the editing process, or inaccuracies based on ever-changing technology. If you spot something that doesn't seem right, please email us at marketing@kobiton.com. We welcome all comments!

Enjoy your journey!

Introduction

Congratulations on taking the first step to automated mobile testing with Appium. We're excited to have you with us on this journey.

With most organizations adopting a 'mobile-first' policy and the increasing criticality that mobile devices play in our everyday lives, the importance of rigourous mobile testing is as important than ever.

And with the increasing pressure to release more, faster and with better quality, companies need to integrate automated testing into their Quality Assurance process. A task often made more difficult thanks to the fragmented mobile ecosystem requiring development and testing across many different device and operating system combinations.

There are plenty of testing tools out there to test app functionality (Especially User Interface testing) of these hybrid, mobile web, and native mobile applications.

So why Appium?

Ideally, to answer that question we need to understand that there are certain characteristics that must be present in an Automation tool, including:

- We should not have to change the app to apply automation testing
- It should not be language and platform specific.
- It should be open-source so it can have a large community and better updates.
- We should be able to automate both Hybrid and Native apps.

Appium checks all these boxes, and that's why it's considered to be the leading mobile test automation tool.

Appium is wrapper built upon Selenium WebDriver that translates Selenium commands into iOS and Android specific commands, making the Selenium WebDriver compatible with mobile. Selenium supports Java, Python C#, Ruby, JavaScript, PHP.

Appium is using vendor-specific frameworks which are:

- iOS 9.3 and above: Apple's XCUITest
- iOS 9.3 and lower: Apple's UIAutomation
- Android 4.2+: Google's UiAutomator/UiAutomator2
- Android 2.3+: Google's Instrumentation. (Instrumentation support is provided by bundling a separate project, Selendroid)
- Windows: Microsoft's WinAppDriver

In the coming chapters we'll be doing a deep-dive into Appium. By the time you work through this book, you will be a very accomplished and capable Appium test engineer. The first place to start your journey is setting up your environment. So grab a cup of coffee and let's get started.

Chapter-1: Setting Up Your Testing Environment

Appium is platform independent so executing Appium scripts is largely consistent across all the major platforms (Linux, Mac, and Windows). In this section, we will discuss how can you setup Appium and it's dependencies on different platforms. Refer to the section relevant to your environment.

Installation on Windows

Software required:

1. Java
2. Android SDK (Android Studio)
3. Node.js
4. Appium Desktop Server

1) Install the JDK software and set `JAVA_HOME`

1. Install the Java Development Kit Software.
    - Go to: http://java.sun.com/javase/downloads/index.jsp
    - Select the appropriate JDK software and click Download.
    - The JDK software is installed on your computer, for example, at `C:\Program Files\Java\jdk1.6.0_02.` You can move the JDK software to another location if desired.
2. Set `JAVA_HOME`:
    - Right click My Computer and select Properties.
    - On the Advanced tab, select Environment Variables, and then edit `JAVA_HOME` to point to where the JDK software is located, like: `C:\Program Files\Java\jdk1.6.0_02.`
    - You can check it by typing `$ java -version` command at the command prompt

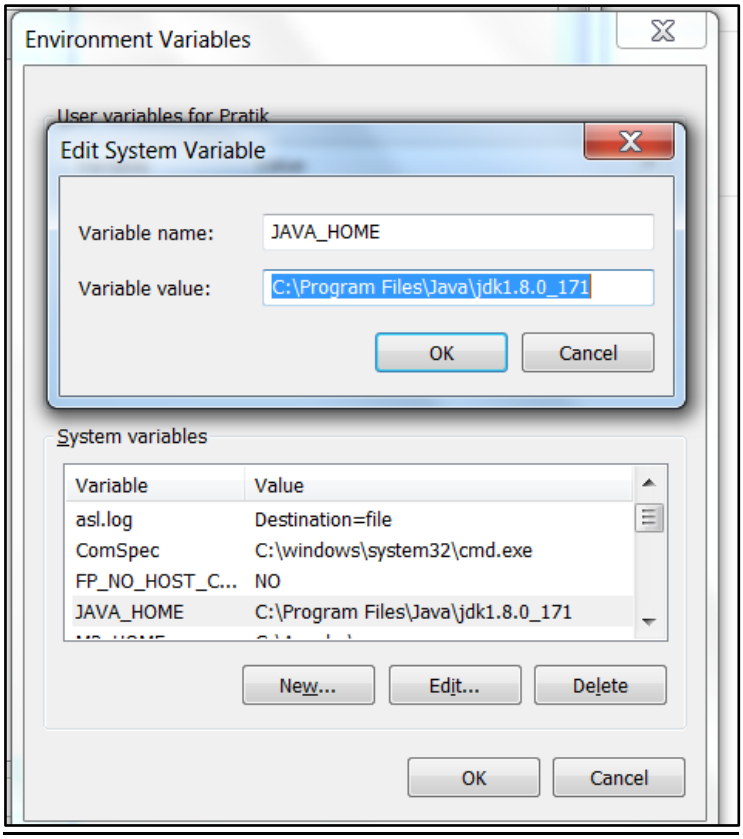

Figure-1: JAVA_HOME System Variable.

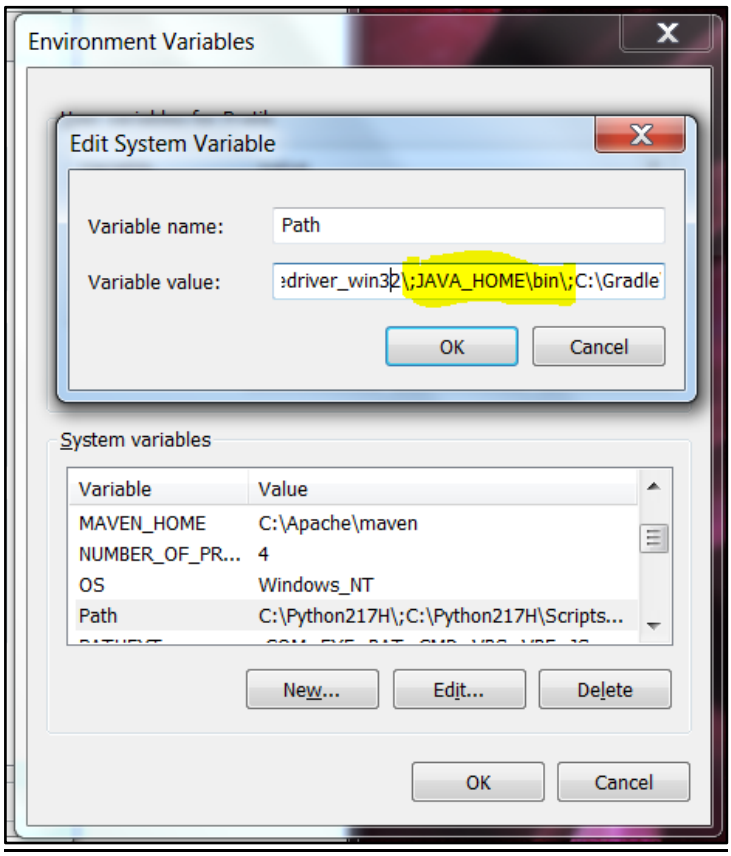

Figure-2: PATH System Variable.

2) Install Android Studio and set `ANDROID_HOME`

1. Install Android Studio and the SDK:
    - Go to: https://developer.android.com/studio/index.html
    - Download and Install Android Studio
    - Open Android Studio and then download the needed Android SDK files from **Tools > Android > SDK Manager**
2. Set ANDROID_HOME:
    - Right click My Computer and select Properties.
    - On the Advanced tab, select Environment Variables, and then add `ANDROID_HOME` to point to where the Android SDK files is located, like: `D:\Android\sdk\`
    - Verify it on the Command prompt using `$ echo %ANDROID_HOME%` command. Output must display the SDK path.

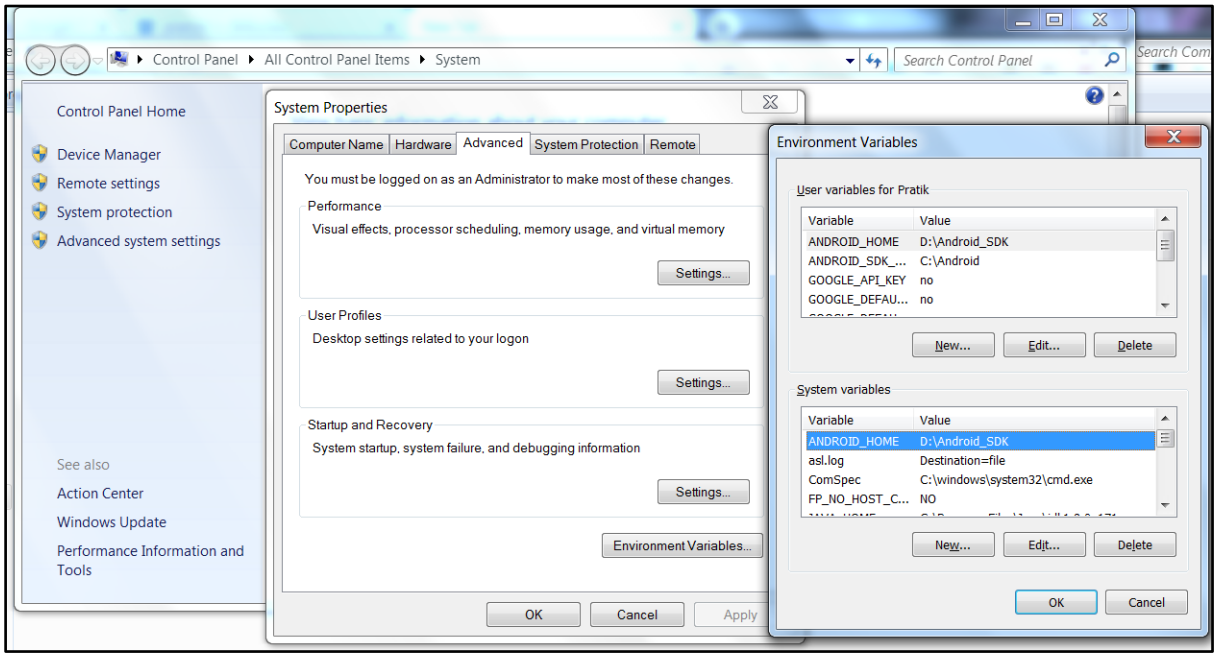

Figure-3: ANDROID_HOME System Variable.

3) Installation of Node.js

1. Install Node.js from: https://nodejs.org/en/download/

2. You can verify installation by entering `$ npm -v` command at the command prompt and it will display the version.

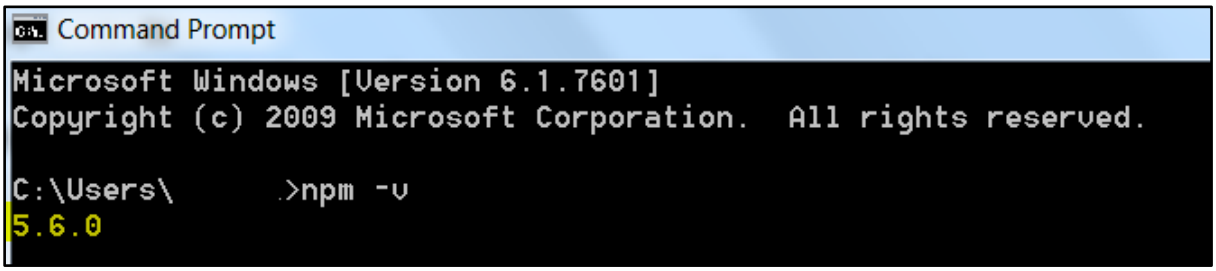

Figure-4: Node.js version

4) Installation of Appium desktop server

1. Go to the Appium github project and Download the relevant Appium Desktop .exe file.

2. Install it and Open ***Appium.exe*** file and start the server

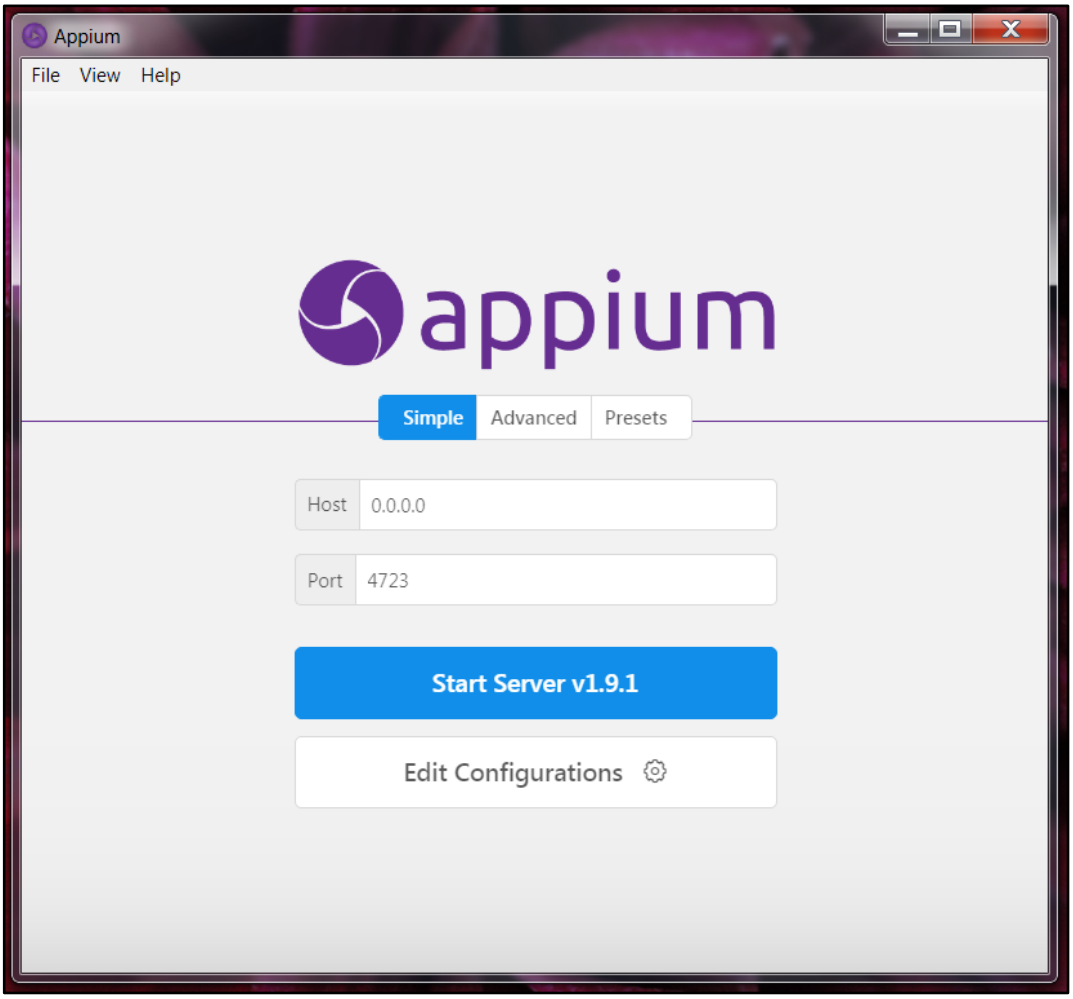

Figure-5: Appium Desktop Application

3. A Terminal should appear saying *'The server is running'*

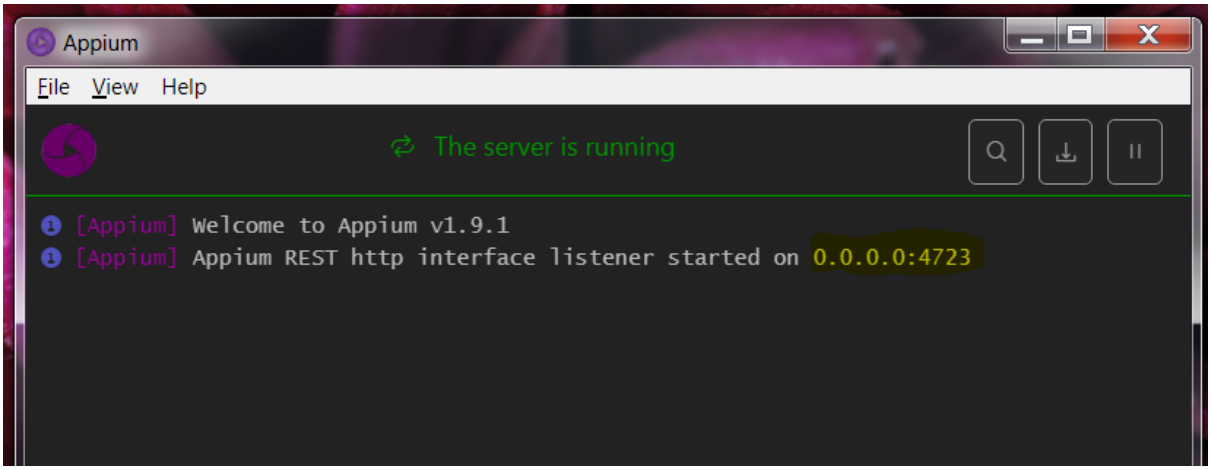

Figure-6: Appium Server is running on 0.0.0.0:4723

Please follow the above steps correctly in order to have a seamless Appium setup experience.

Installation on Mac

Software required:

1. Java
2. Android SDK (Android Studio)
3. Node.js
4. XCode
5. Appium Desktop Server

1) Install the JDK software

1. Install the Java Development Kit Software from: http://java.sun.com/javase/downloads/index.jsp

2. Select the appropriate JDK software and click Download.

3. Set `JAVA_HOME`:
   - Right click My Computer and select Properties.
   - On the Advanced tab, select Environment Variables, and then edit `JAVA_HOME` to point to where the JDK software is located, like: `C:\Program Files\Java\jdk1.6.0_02.`

2) Install Android Studio & Android SDK

1. Install Android Studio from: https://developer.android.com/studio/index.html

2. Open Android Studio and then download the needed Android SDK files from **Tools > Android > SDK Manager**

3) Set JAVA_HOME & ANDROID_HOME

1. We need to store Environment variables in `.bash_profile` file so open Terminal and enter this command to open the bash_profile:

```
$ vi ~/.bash_profile
```

2. Now to you need to go into insert mode by pressing the `**i**` key from the keyboard, and write the following text at the end of the file.

```
export ANDROID_HOME=/Users/username/Library/Android/sdk
export ANDROID_SDK=$ANDROID_HOME
PATH=$PATH:$ANDROID_HOME/build-tools
PATH=$PATH:$ANDROID_HOME/platform-tools
PATH=$PATH:$ANDROID_HOME/tools

export PATH
export
JAVA_HOME="`/System/Library/Frameworks/JavaVM.framework/Versi
ons/Current/Commands/java_home`"
```

3. Press ESC key followed by **:wq** which will save the `.bash_profile` file.

4. You can check that `JAVA_HOME` & `ANDROID_HOME` are properly set by executing commands `$ java -version` & `$ echo $ANDROID_HOME` Respectively.

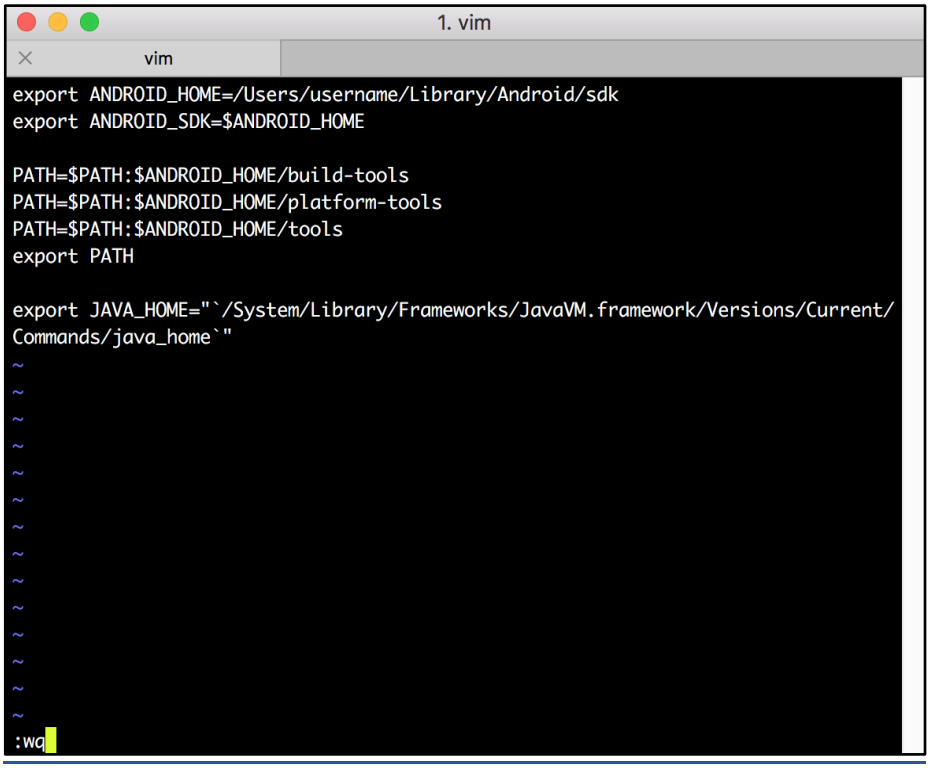

Figure-7: ANDROID_HOME & JAVA_HOME environment variables.

4) Installation of Node.js

1. Install Node.js from: https://nodejs.org/en/download/

2. You can verify the installation by entering `$ npm -v` command at the command prompt and it will display the version.

5) Installation of the Appium desktop server

1. Go to the Appium github page and download the relevant Appium Desktop .dmg file.

2. Install it and Open *Appium* and start the server

3. A Terminal should appear saying *'The server is running'*

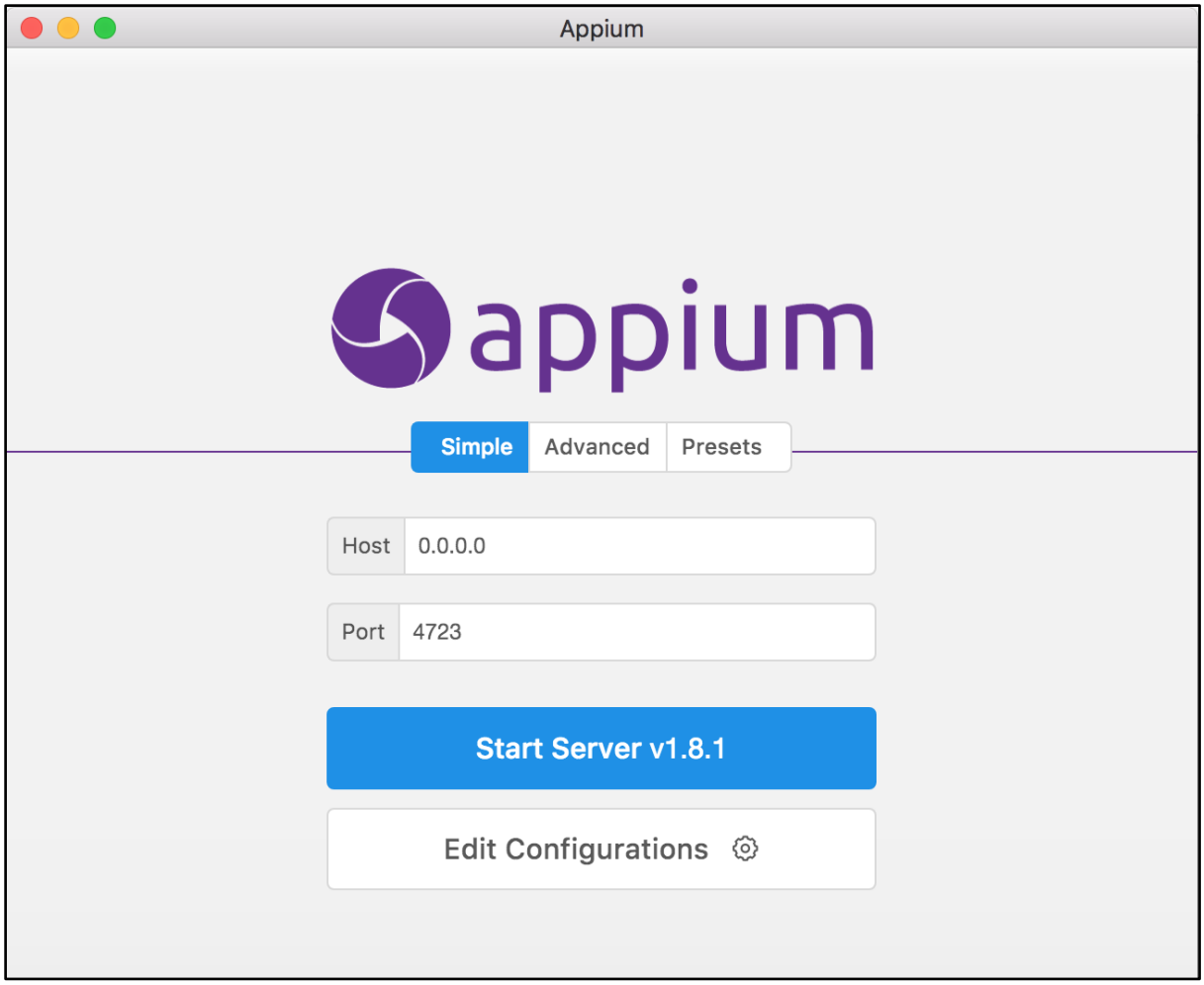

Figure-8: Appium Desktop Application

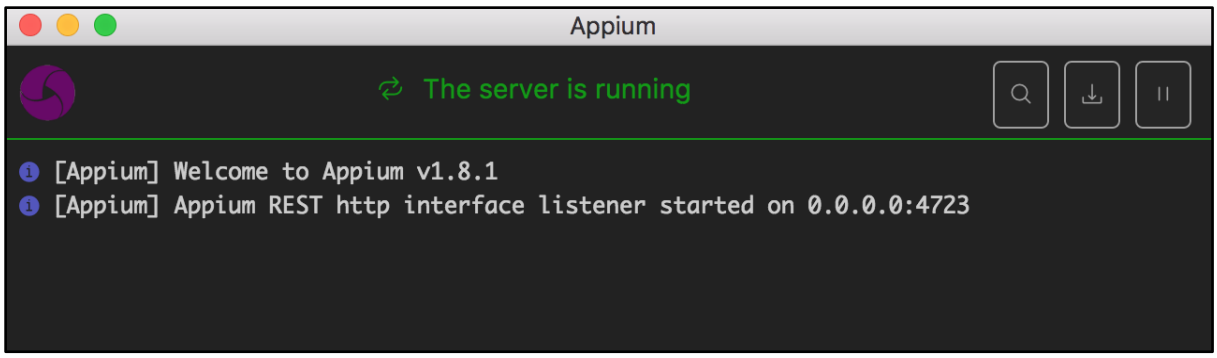

Figure-9: Appium Server is running on 0.0.0.0:4723

6) XCode with Appium libraries setup

1. **Install XCode:** https://developer.apple.com/download/

2. **Install Xcode Command line tools:**
   - Execute below command on Terminal:
     `$ xcode-select --install`

3. **Install Brew(If it's not installed already):**
    - Execute below command on Terminal:
      `$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"`

4. **Install libimobiledevice:**
    - Execute below command on Terminal:
      `$ brew install libimobiledevice --HEAD`

5. **Install ios-deploy:**
    - Execute below command on Terminal:
      `$ npm install -g ios-deploy`

6. **Install carthage:**
    - Execute below command on Terminal:
      `$ brew install carthage`

NOTE: *Now you are set to run your iOS Appium Script on Simulator*

7) WebDriverAgentRunner setup(Setting up iOS real devices tests with XCUITest)

This is the most crucial and important step of Appium setup for iOS.
If you don't follow these steps correctly then you might not be able to run the Appium Automation scripts on your real iOS devices.

There are 2 ways to configure project in Appium:

1) Basic (automatic) configuration
2) Basic (manual) configuration

1. Automatic configuration

The easiest way to get up-and-running with Appium's XCUITest support on iOS real devices is to use the automatic configuration strategy. There are two ways to do this:

1) Use the `xcodeOrgId` and `xcodeSigningId` desired capabilities:

```
{
   "xcodeOrgId": "<Team ID>",
```

```
    "xcodeSigningId": "iPhone Developer"
}
```

In Java, the code will look like:

```
desiredCapabilities.setCapability("xcodeOrgId", <<Team ID>>);
desiredCapabilities.setCapability("xcodeSigningId", "iPhone
Developer");
```

NOTE: You will learn about desiredCapabilities in a subsequent chapter. You can always come back to this section after you've progressed in your Appium knowledge a bit further and need to start testing on real iOS devices (as compared to Emulators).

or

2) Create a `.xcconfig` file somewhere on your file system and add the following to it

```
DEVELOPMENT_TEAM = <Team ID>
CODE_SIGN_IDENTITY = iPhone Developer
```

After this you need to set the desired capabilities and set the path to `.xcconfig` file:

```
desiredCapabilities.setCapability("xcodeConfigFile",
"path/to/.xcconfig")
```

In either case, the ***Team ID*** is a unique 10-character string generated by Apple that is assigned to your team. You can find your Team ID using your developer account. Sign in to www.developer.apple.com/account , and click Membership in the sidebar. Your Team ID appears in the Membership Information section under the team name. You can also find your team ID listed under the "*Organizational Unit*" field in your iPhone Developer certificate in your keychain.

NOTE: These are mutually exclusive strategies; use either the `xcodeConfigFile` capability or the combination of `xcodeOrgId` and

`xcodeSigningId`. For more details you can visit this link.

2. Manual configuration

There are many cases in which the basic automatic configuration is not enough. This usually has to do with code signing and the configuration of the project to be able to be run on the real device under test. Often this happens when the development account being used is a "*Free*" one, in which case it is not possible to create a wildcard provisioning profile, and will often not create one for the default application bundle.

Please follow these steps to Manually configure the **WebDriverAgent** XCode project.

- Move to the: WebDriverAgent.xcodeproj(Make sure you have installed Appium Desktop application properly):

```
$ cd
/Applications/Appium.app/Contents/Resources/app/node_modules/
appium/node_modules/appium-xcuitest-driver/WebDriverAgent
```

- Execute the `Scripts/bootstrap.sh` script using:

```
$ sh Scripts/bootstrap.sh
```

- Open the **WebDriverAgent.xcodeproj** project in Xcode.

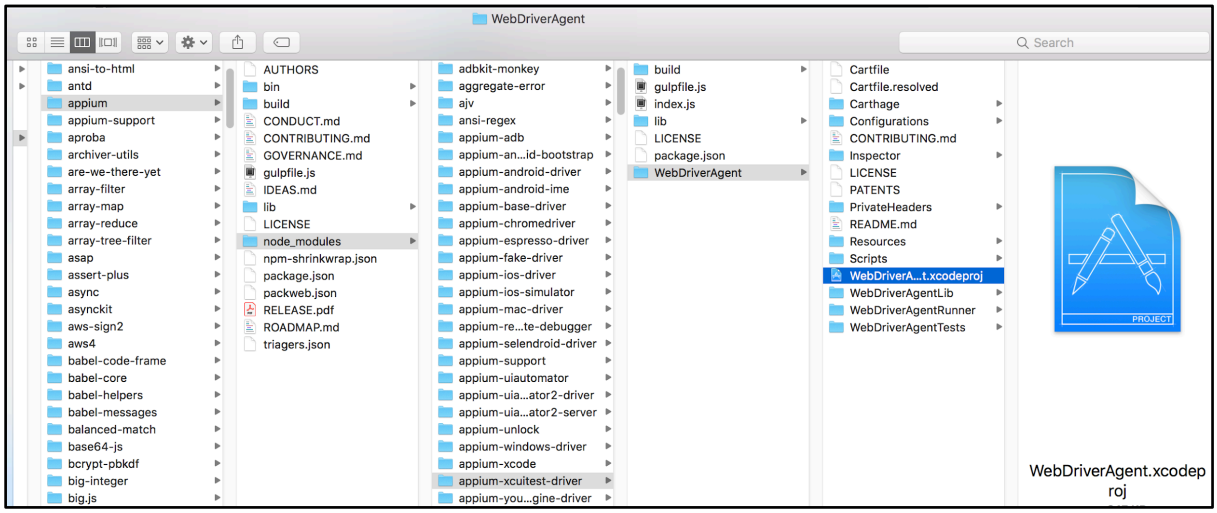

Figure-10: WebDriverAgent.xcodeproj Project in Finder window..

- Select **WebDriverAgentRunner** under **TARGETS.**

- Now when you move to **WebDriverAgentRunner**, you would face an error that **Xcode failed to create provisioning profile**:

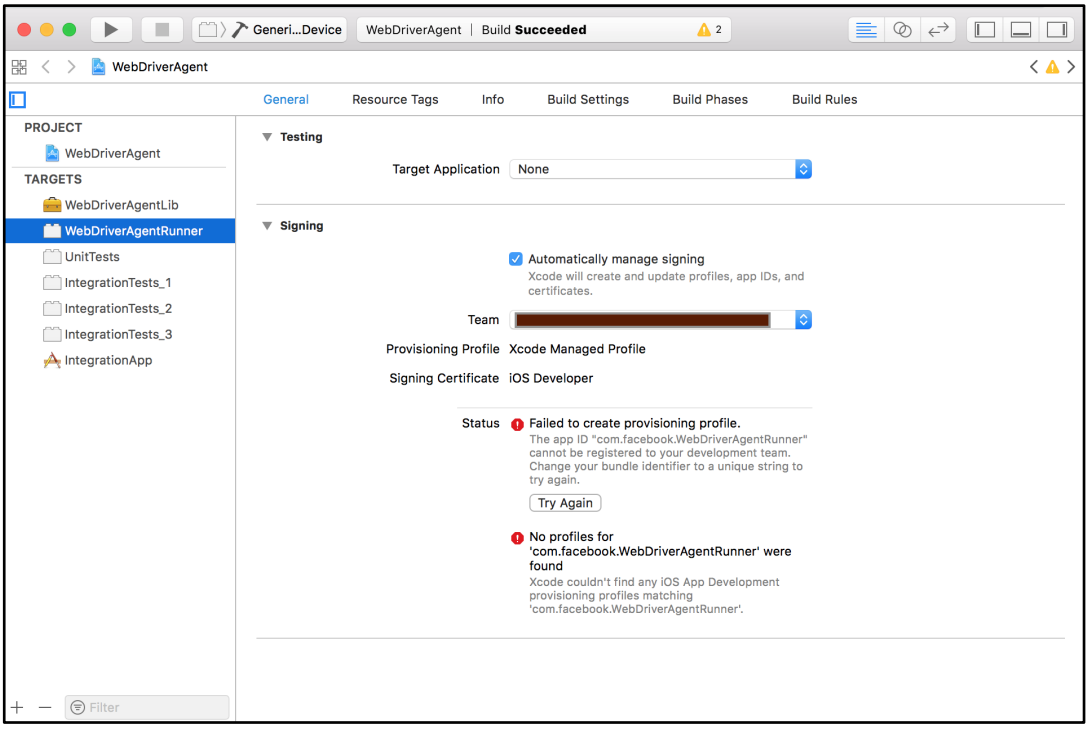

Figure-11: XCode: WebDriverAgentRunner Project in not able to find Provisioning profile.

- The easiest way to resolve that is 1) Select **WebDriverAgentLib** under **TARGETS**, 2) select **Automatically manage signing**, 3) select valid **Team** and most important 4) change the **Bundle Identifier** and put the Bundle Identifier of your existing valid XCode project the purpose here to put something that Xcode will accept.

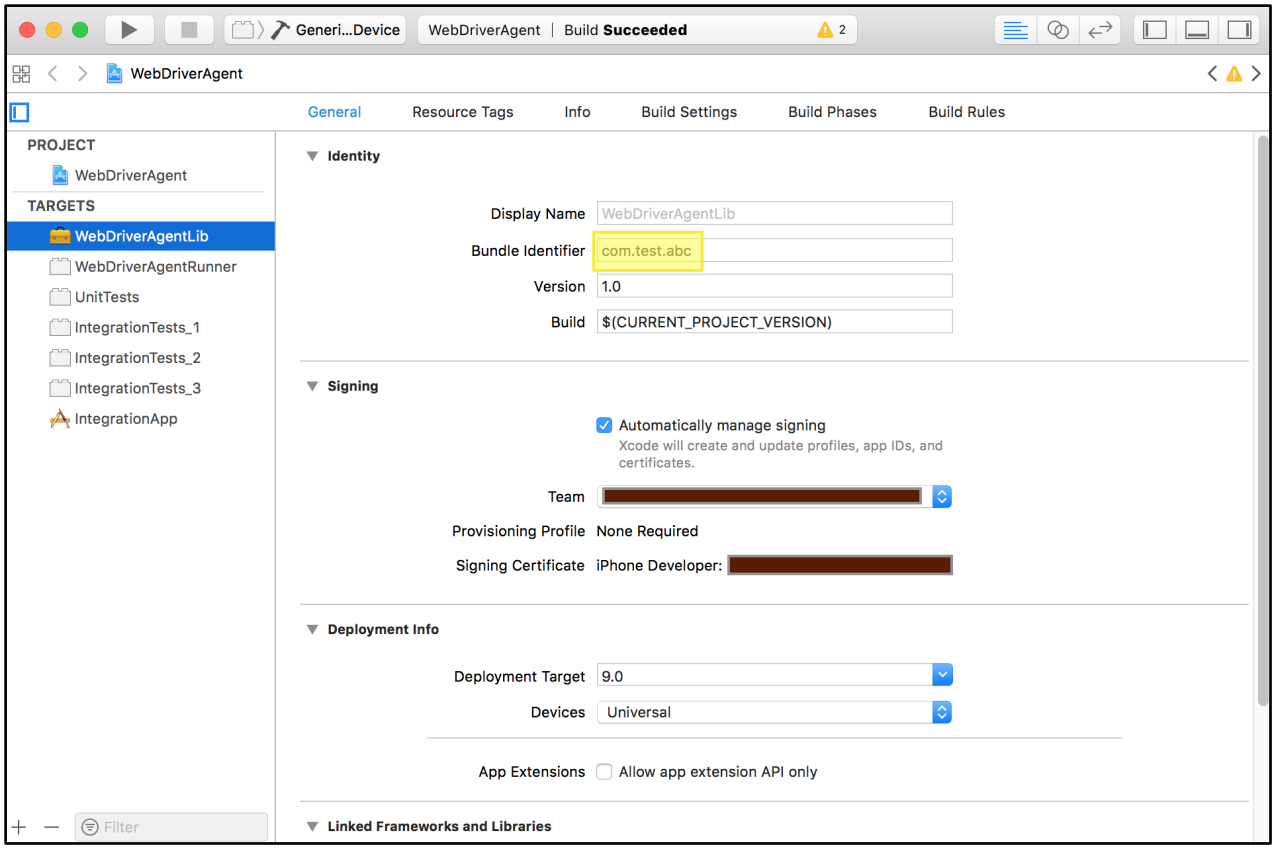

Figure-12: XCode: Change Bundle Id, Select and Select Valid Team Project

- Also ensure that you should have installed the valid Provisioning profile*(Of course compatible with entered Certificate and Bundle Identifier)*. Now move to **WebDriverAgentRunner** again and 1) Select valid Provisioning Profile under Signing (Debug) and 2) Select valid Provisioning Profile under Signing (Release).

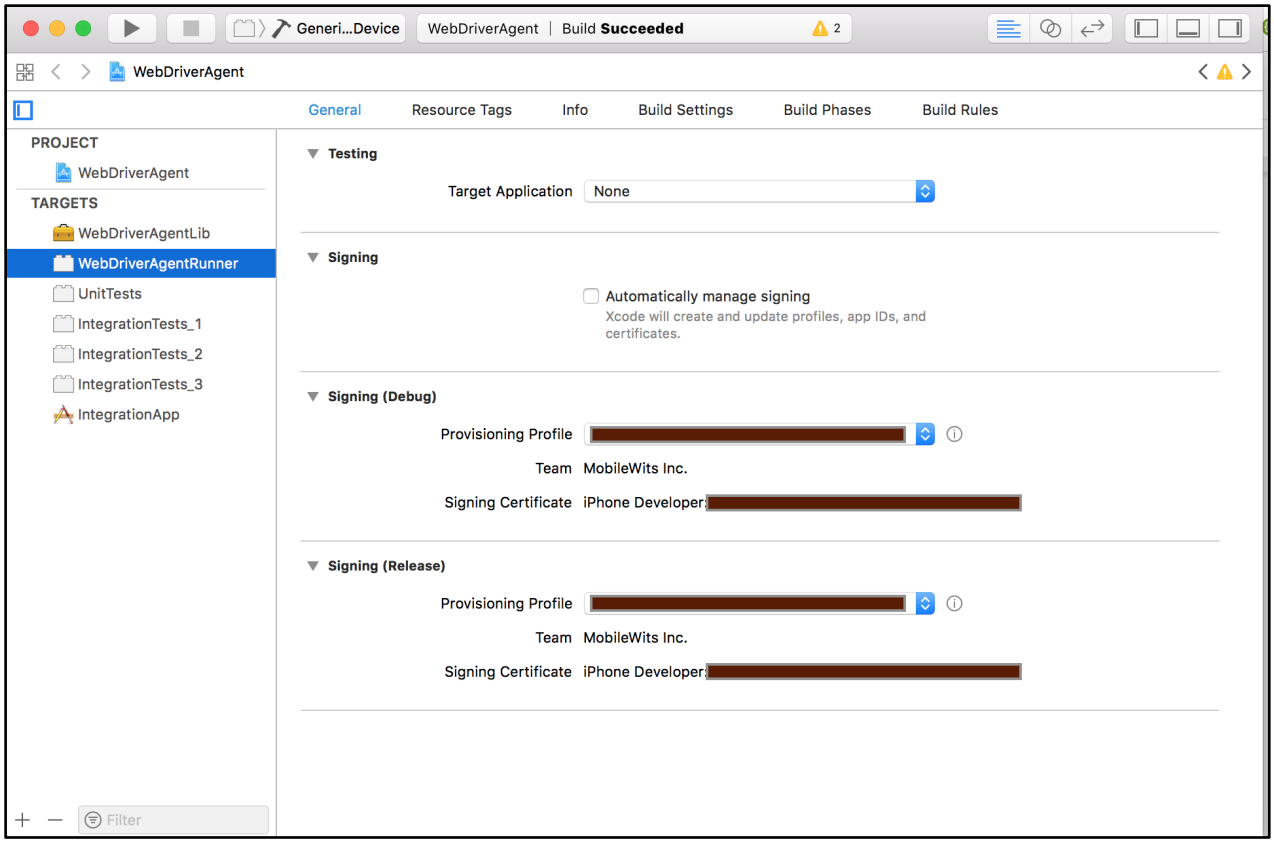

Figure-13: XCode: Select Valid Provisioning Profile for WebDriverAgentRunner

- Now to be on safer side repeat above step same for UnitTests, IntegrationTests_1, IntegrationTests_2, IntegrationTests_3 and IntegrationApp

- Connect valid iPhone device to your Mac machine*(Please ensure device is included in selected provisioning profile).*

- Select **WebDriverAgentRunner** under TARGETS and click on Test button to execute build on your connected device.

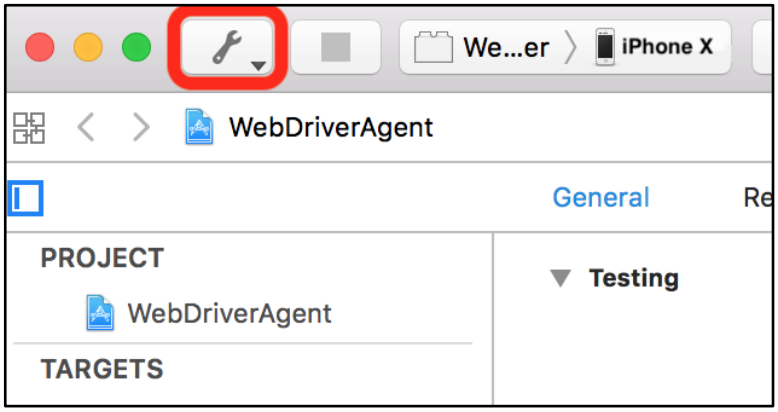

Figure-14: XCode: Test the Project on connected iPhone device

- You can observe that when you click on Test/Run button the WebDriverAgent application will be installed to iOS device and it will open and give you the black screen for a moment and automatically closed. That means Success. **Now you can able to Run Appium script on this device**.*(In fact it applies to all the valid devices registered under selected provisioning profile).*

NOTE: Please install **Appium-Doctor(Node Utility)** using npm, It will diagnose and fix common Node, iOS and Android configuration issues before starting Appium.

```
$ npm install -g appium-doctor
//then
$ appium-doctor
//it will give checklist of which things are okay and
which are not
```

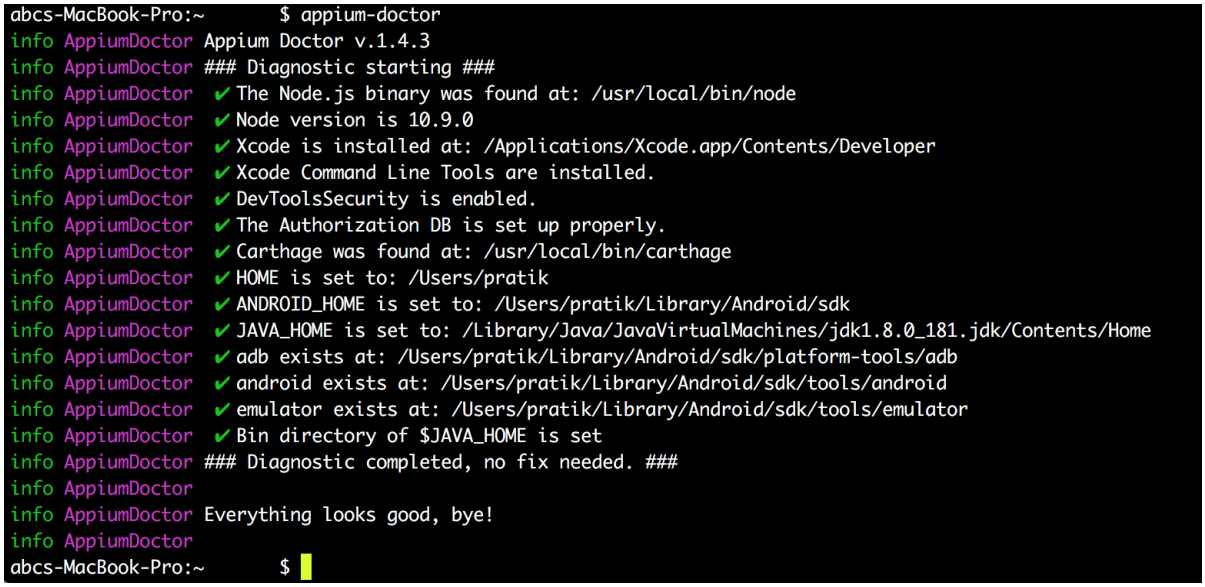

Figure-15: Appium Doctor Node Utility on iOS

Installation on Ubuntu(Linux)

Software required:

1. Java
2. Android SDK (Android Studio)
3. Node.js
4. Appium Desktop Server

1) Install JAVA(JDK/JRE) & set JAVA_HOME

- Install Java Development Kit Software from:

```
$ sudo apt-get update
$ sudo apt-get install default-jre
$ sudo apt-get install default-jdk
//to install oracle jdk

$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

- In order to set JAVA_HOME on linux you need to edit the `.bashrc` and need to specify the path of java directory.

```
$ vi ~/.bashrc
```

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export PATH=${PATH}:${JAVA_HOME}/bin
```

- Run below command to verify that recently saved environment variables are displaying correct path or not.

```
$ echo $JAVA_HOME
$ echo $PATH
```

NOTE: Use the command: `$ which java` to find out exact path to which java executable

2) Install Node.js without using sudo

- Do not install node.js through *apt-get*, which will need sudo rights and appium will not work if node.js is installed as sudo user.

  If you have already installed remove it using commands:

```
$ sudo apt-get remove nodejs
$ sudo apt-get remove npm
```

- Download latest nodejs linux binaries from https://nodejs.org/download/release/latest/ into a folder for example /home/username/Downloads.

```
$ cd /usr/local tar --strip-components 1 -xzf
```

```
/home/username/Downloads/node-v8.2.1-linux-x64.tar.gz
```

- For Verification use below commands:

```
$ which node //it should give you output like:
usr/local/bin/node
$ node -v  //it should give you output v8.2.1 (or whichever
version you have installed)
```

3) Install Android Studio

- Download and install Android Studio from Official Google Website: https://developer.android.com/studio/
- Open terminal and Enter:

```
$ cd android-studio/bin
$ . studio.sh //To execute the studi.sh script
```

- Now android studio will open.Click next and let it download required things.

- Once android sdk is installed add ANDROID_HOME to environment variable using:

```
$ vi /.bashrc
```

```
export ANDROID_HOME=/home/user_name/Android/Sdk
export PATH=$PATH:/home/user_name/Android/Sdk/tools
export PATH=$PATH:/home/user_name/Android/Sdk/platform-tools
```

4) Install Appium globally:

- Open terminal and enter below commands in order to install the appium globally.

```
$ npm install -g appium
$ appium //To start the Appium server
```

5) Install appium-doctor to troubleshoot the errors if any using

- Please install Appium-Doctor(Node Utility) using npm, It will diagnose and fix common Node, iOS and Android configuration issues before starting Appium.

```
$ npm install -g appium-doctor
//then
$ appium-doctor
//it will give checklist of which things are okay & which aren't
```

Chapter-2: Writing Your First Test Case

And now the moment you've been waiting for: Writing your first test case. You'll quickly grasp the basics of Appium and see how it can be used for test automation.

We're going to cover a lot of ground in this chapter but it will be well worth it. By the end of this chapter you'll have grasped the basics of Appium and writing test cases.

Appium supports Native, Hybrid and Web application testing, and you can execute Appium scripts on Real Physical devices(iOS/Android), simulators(iOS) and emulators(Android).

The best thing about Appium is it has no dependency on the Mobile OS or Mobile Application, meaning you can Appium scripts can run everywhere. For Automation Testing with Appium you will just need the APK or IPA file.

Under the hood, Appium is just wrapper that translates Selenium Webdriver commands into XCUITest for iOS and UiAutomator2 for Android. XCUITest and UiAutomator2 are Test frameworks for XCode and Android Studio respectively.

Appium supports all the programming languages which Selenium supports such as Java, C#, Python, Ruby, Javascript with Node.js etc.

For the examples that follow, we will be using Java since Appium was written in Java, and you can find many resources online on Appium and Java, which makes your learning journey a little easier. In our examples we will be using a Mac, but of course you will be fine on any supported operating system.

Please make sure you followed the previous chapter and installed Appium properly. Now that Appium is installed, we'll be installing our development environment. If you already have a development environment installed, you can skim through the steps that follow. However, it may be easier for you to follow these instructions and have your environment mimic ours for easier reference.

Note: The steps may seem a little daunting especially when you want to write a simple test case. However, this is really a one-time effort. Once your environment is configured and you're comfortable with concepts such as dependency management, you'll find writing the Appium scripts is a relatively straightforward task. So stay with us through this section, and it will get easier, we promise.

We will start by installing an IDE to create Java based Appium Scripts. You can either select IntelliJ IDEA or Eclipse IDE (or any IDE of your choice). We will be using IntelliJ.

To summarize, we will use the following to create our ***Appium Script:***

1. **Programming Language:** Java
2. **Test Framework:** TestNG
3. **IDE:** IntelliJ IDEA
4. **Project Dependency Tool:** Gradle

There are 2 primary things to accomplish in this chapter:

1) Setup the IDE (IntelliJ IDEA).
2) Create the First Automation Test Case. Note that Appium is used to drive or control the underlying mobile application in order to perform automation on it. However, you still need some testing framework for implementation of the actual tests. We will be using the TestNG framework for this purpose.

Setup the IDE (IntelliJ IDEA)

In order to sethup the IDE you need to:

1) Install the intelliJ IDEA
2) Install the TestNG plugin on IntelliJ IDEA.

1) Installation of IntelliJ IDEA

1. You can download and install IntelliJ IDEA Community Edition from here: https://www.jetbrains.com/idea/download/#section=mac

2. Install it by dragging and dropping:

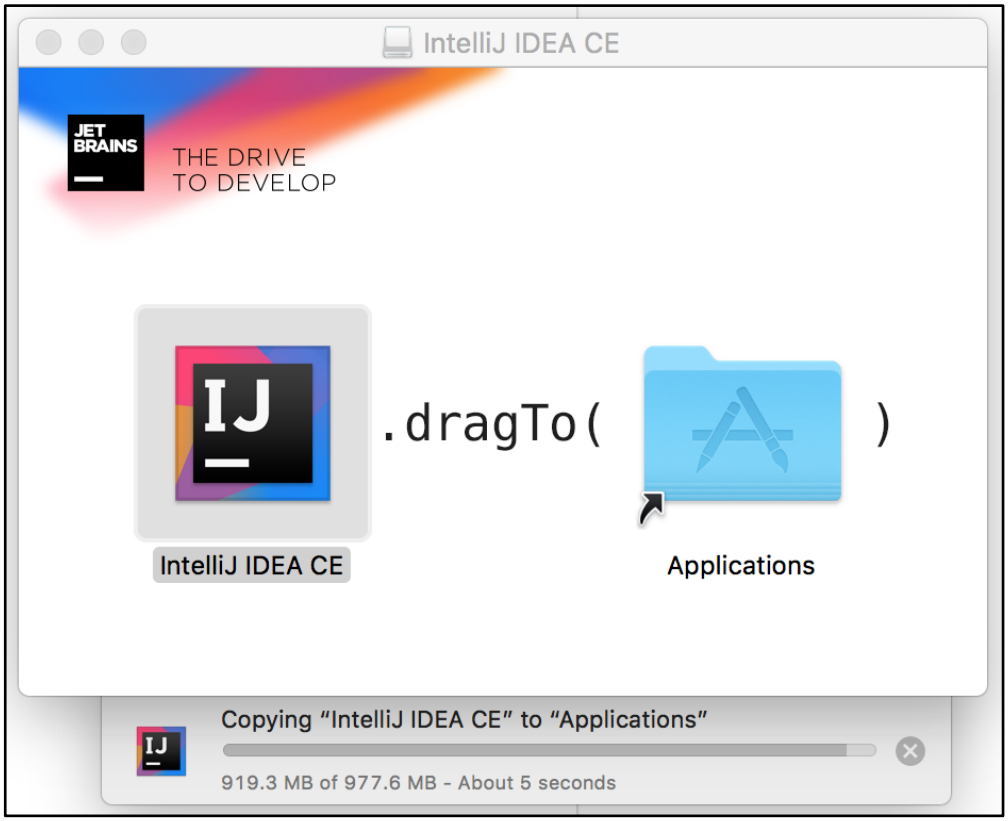

Figure-1: Installation of IntelliJ IDEA Community Edition.

3. If you are installing IntelliJ Idea first time, then you need to select the " Do not import settings" option.

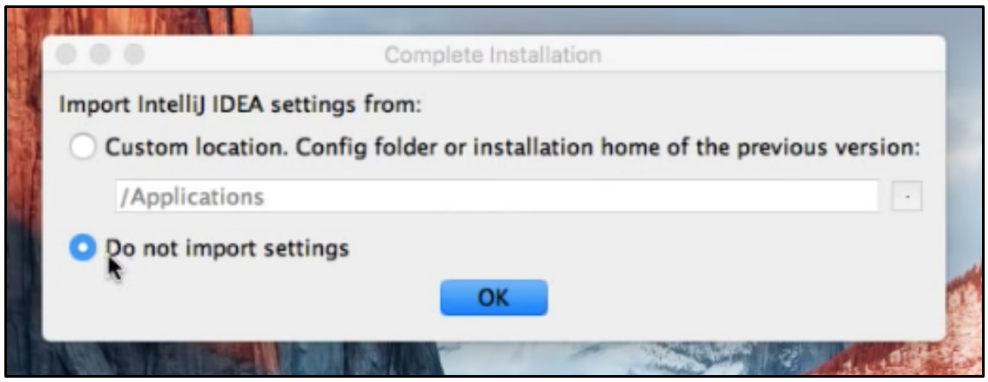

Figure-2: Complete the Installation.

4. Accept the IntelliJ IDEA Privacy Policy Agreement:

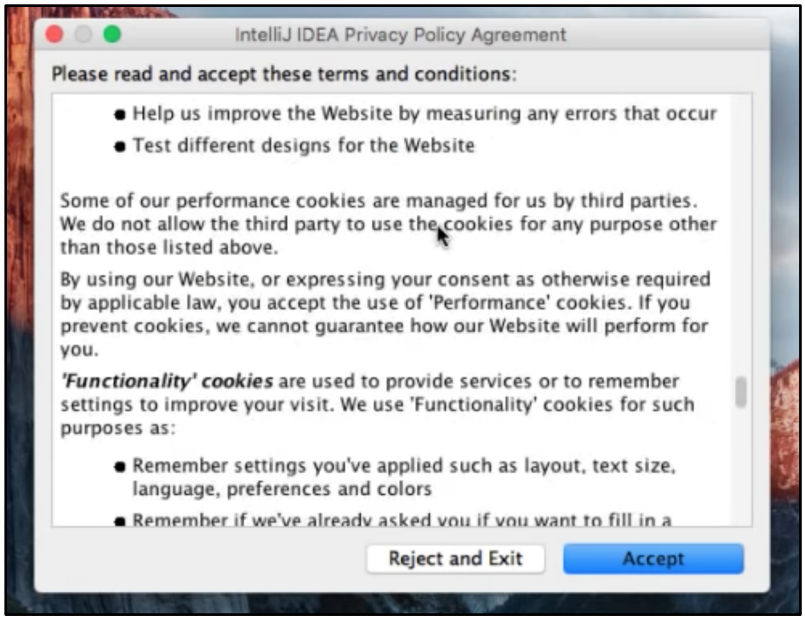

Figure-3: Policy Agreement.

5. Set your preferred theme and click on Next:

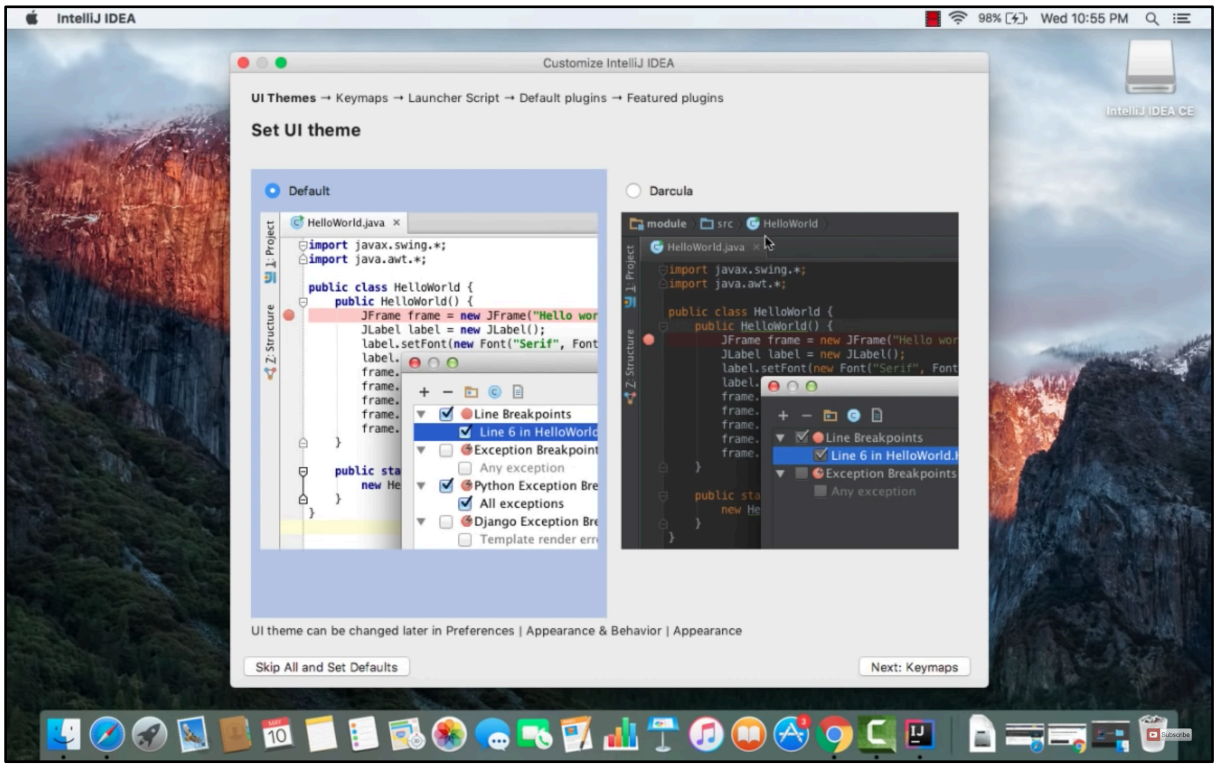

Figure-4: Set UI Theme.

6. Select the default plugins, and Finish the setup.

2) Install the TestNG plugin on IntelliJ IDEA

By default the TestNG plugin is installed in IntelliJ IDEA. You can check it in several ways but the best way is to check it is in Plugins.

1. Open IntelliJ Idea and click on Configure > Plugins.

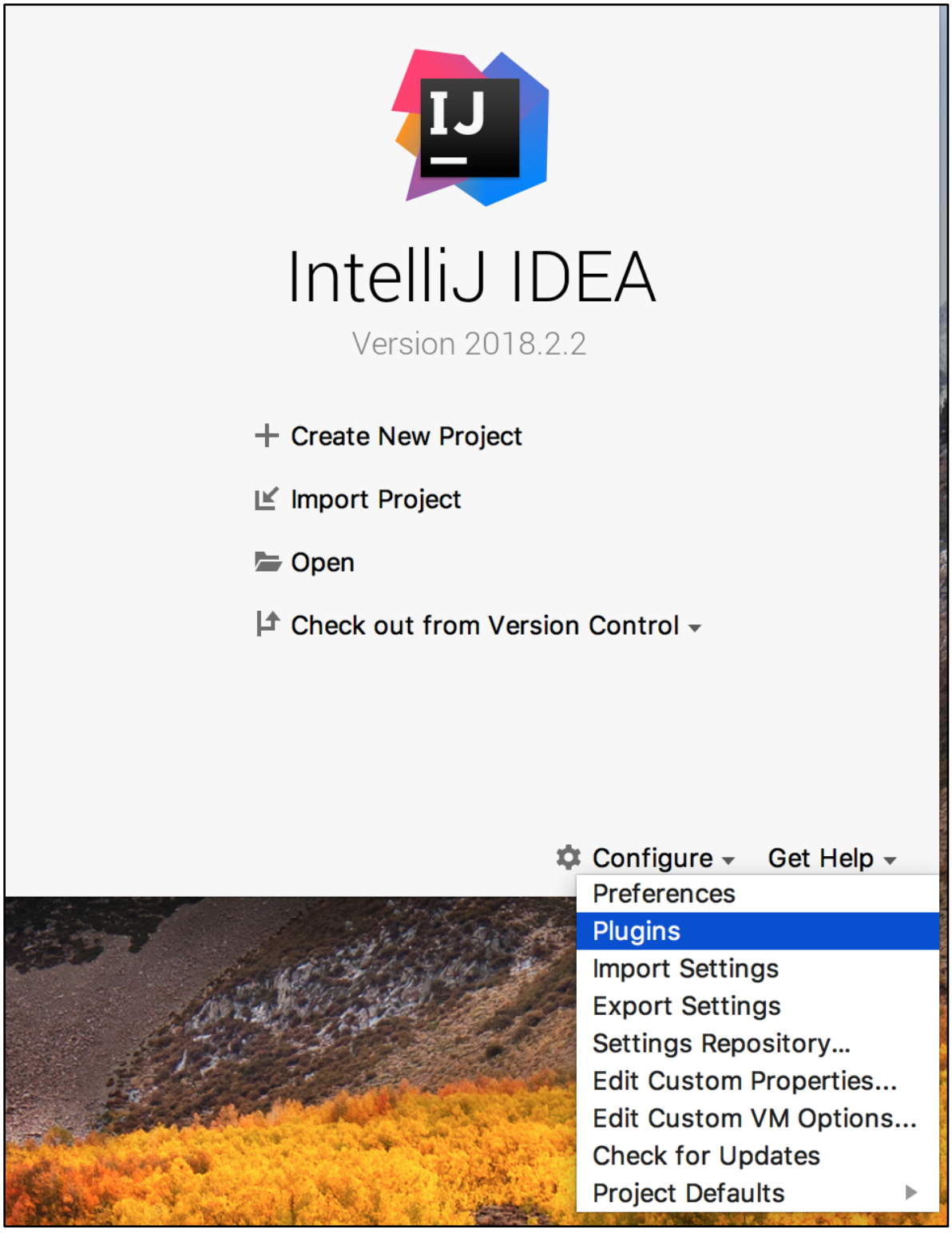

Figure-5: Go to IntelliJ IDEA Plugins.

2. Type 'testng' and search it, if it is installed properly you can see the Right tick icon right next to 'TestNg' text. And if it is not installed you need to install by clicking on **Install JetBrains plugin… > Search for 'testng' > Install it.**

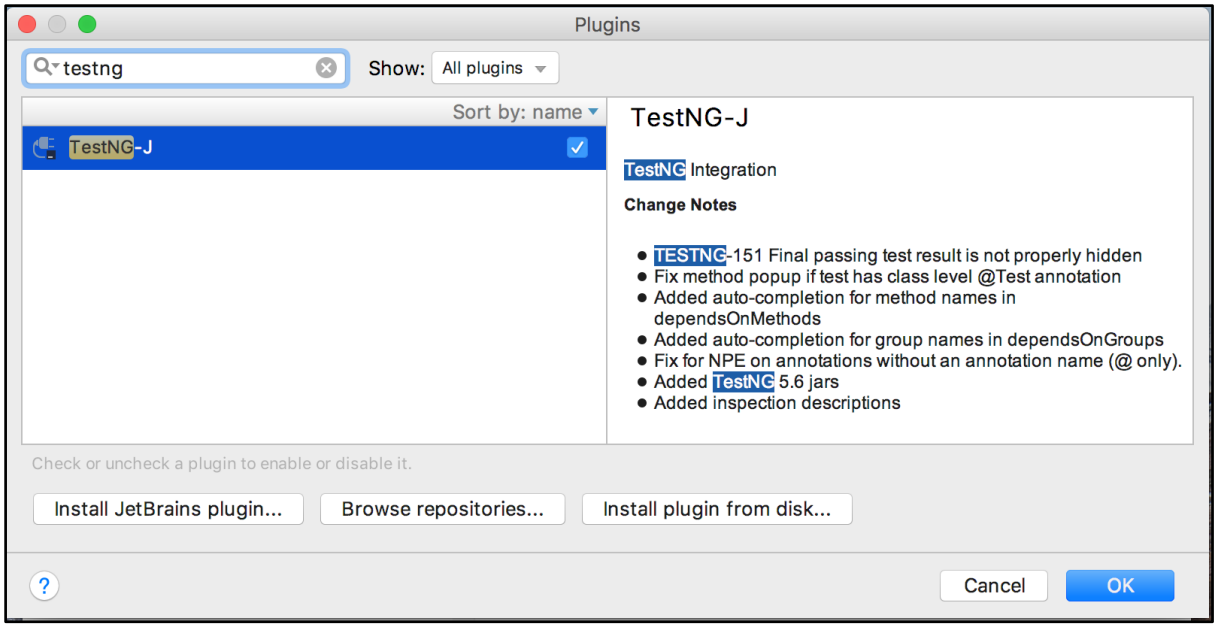

Figure-6: Check IntelliJ Plugin.

Create your first automation test case

We need to create a new project and than need to setup the Automation script, so we can divide this into 2 sections:

1) Create New Project.
2) Setup the Automation Case.

1) Create new project

1. Open IntelliJ IDEA and click on **`Create New Project`**

Figure-7: Create New Project.

2. Now we need to select the project configuration such as Project type, Java version, and Build tool. We can either use Maven OR Gradle, but since Gradle is more flexible we will be using that for our tutorial.

However you can use Maven if you prefer (We just need to add few dependencies).

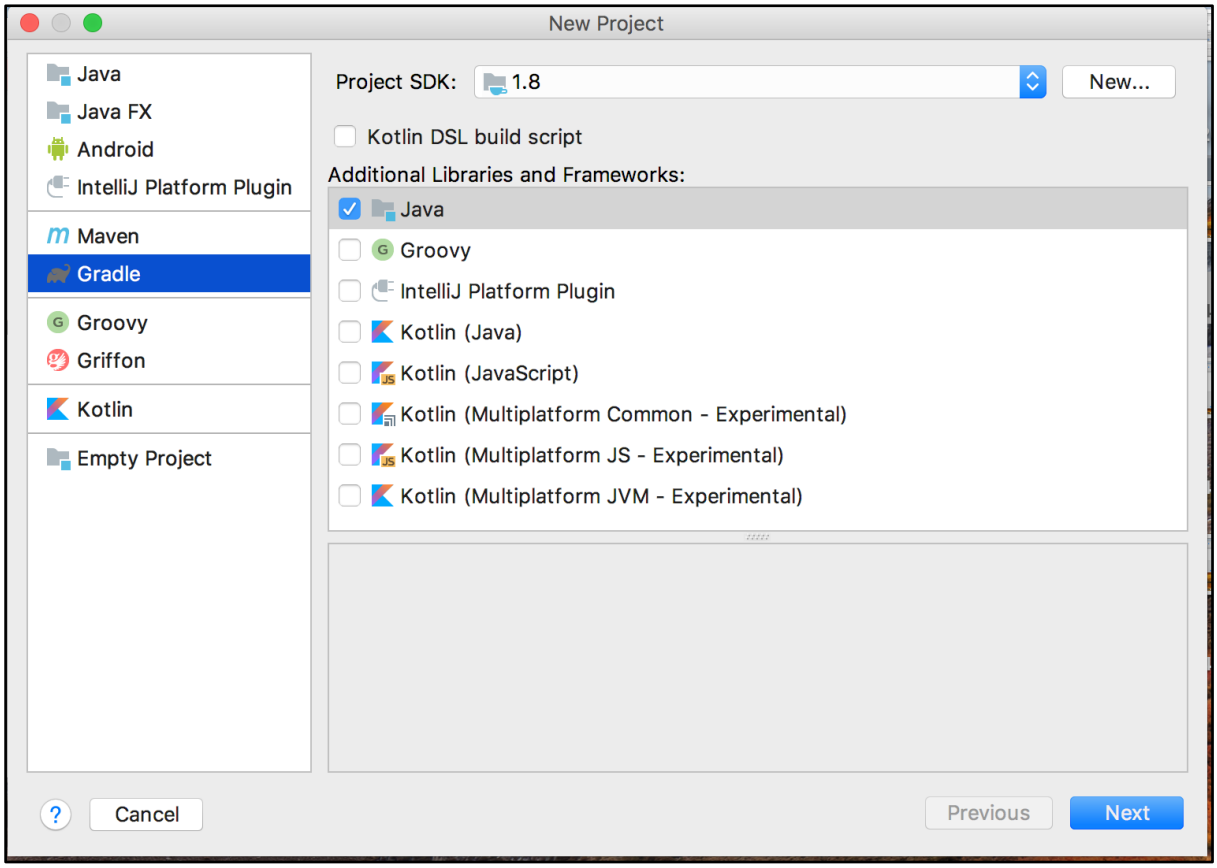

Figure-8: Create New Project.

3. Give the proper GroupId and ArtifactId(Project Name) and Version.

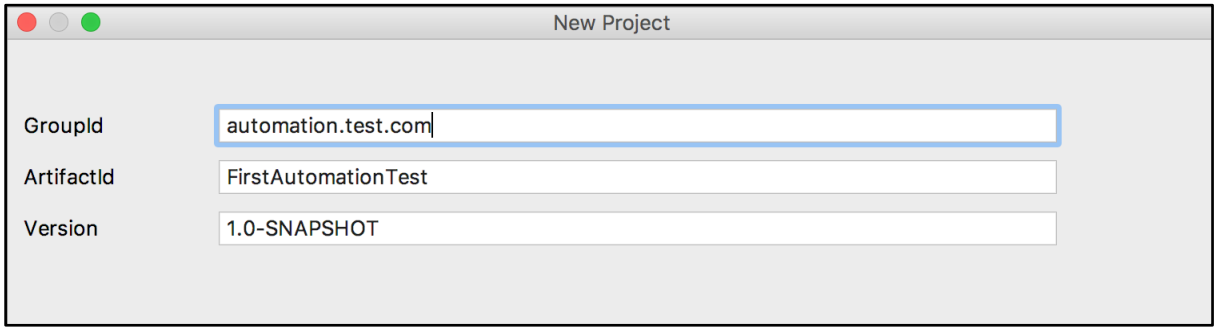

Figure-9: Enter GropuId, ArtifactId and Version.

***NOTE:** For better project management it is better to give a proper Group Id and Artifact Id.*

Visit: http://maven.apache.org/guides/mini/guide-naming-conventions.html to learn more about it.

4. This is the Gradle selection dialog, If you haven't installed Gradle explicitly it is recommended to the **Use default gradle wrapper (recommended)**

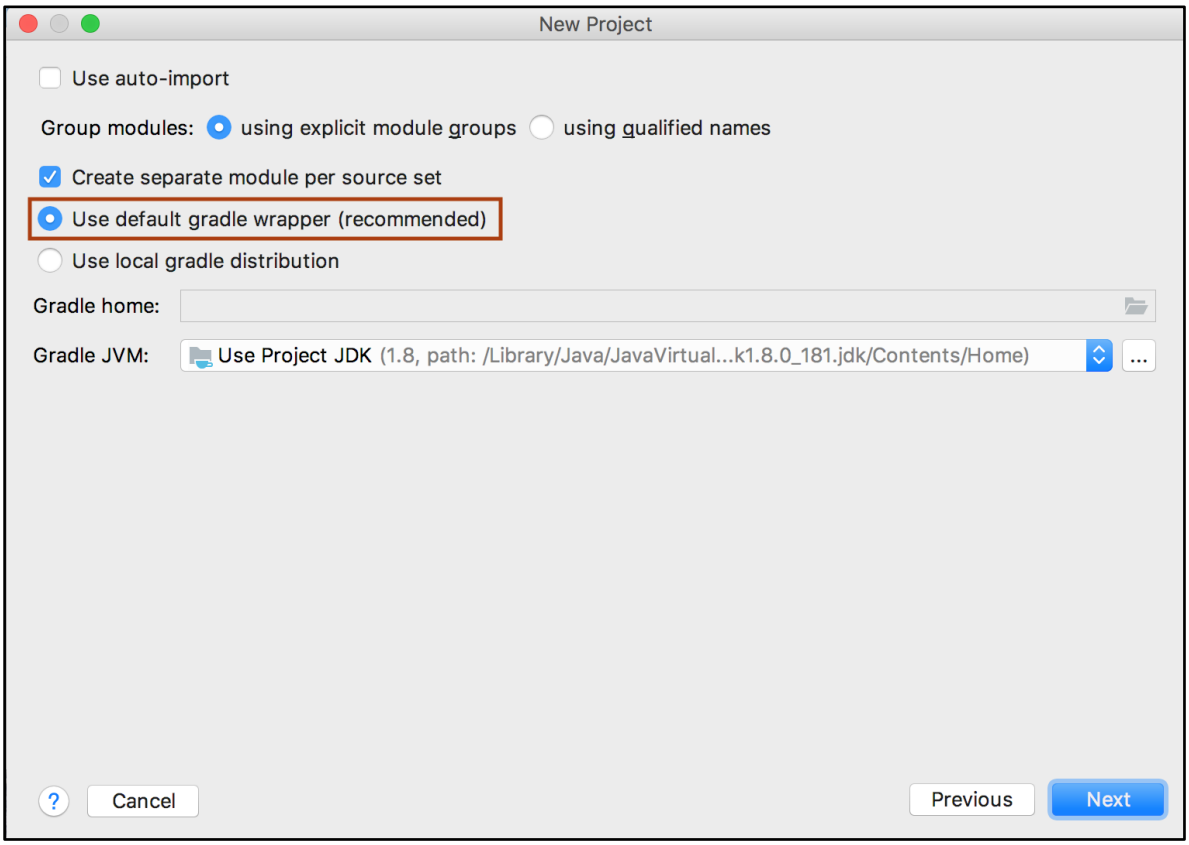

Figure-10: Select 'Use default gradle wrapper (recommended)'

5. Confirm all the project details and Finish the setup.

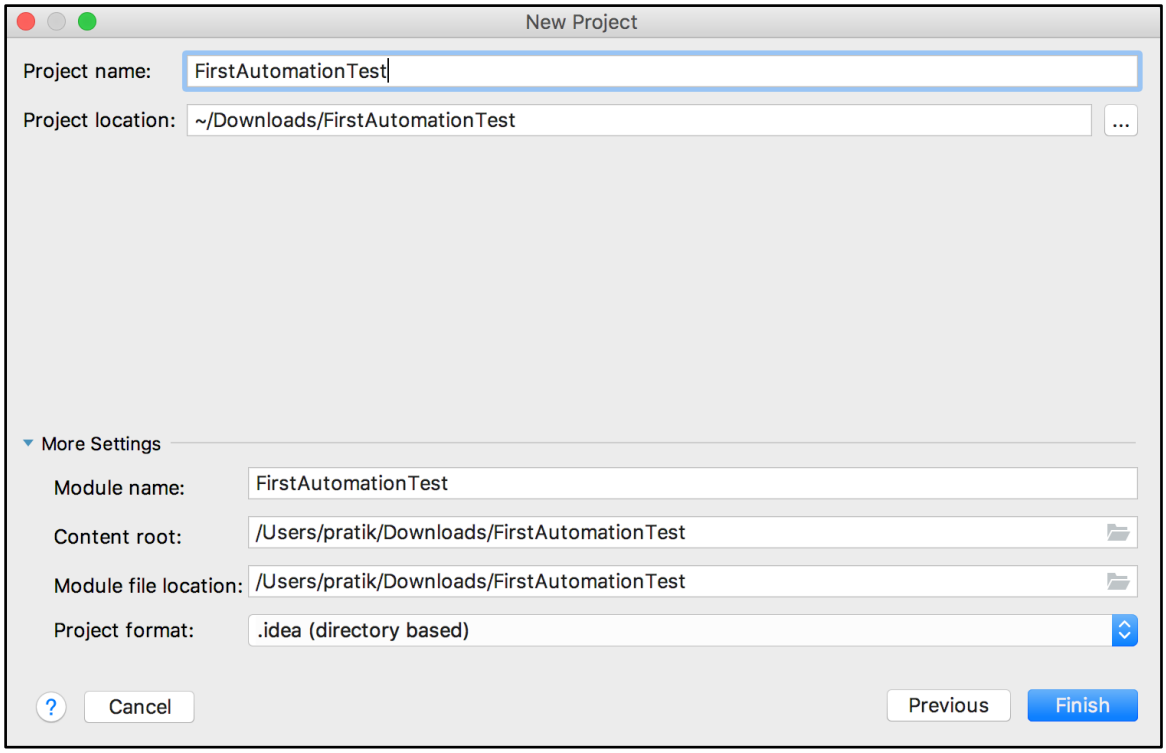

Figure-11: Confirm Project Details.

6. After the setup is finished, gradle will build the project and it may take some time (especially for the first time, because it will download the Gradle .zip file). After the sync up is done you can see the Project Directory as per the below image.

   Now we are ready to add Appium dependencies and then start coding the first automation test case.

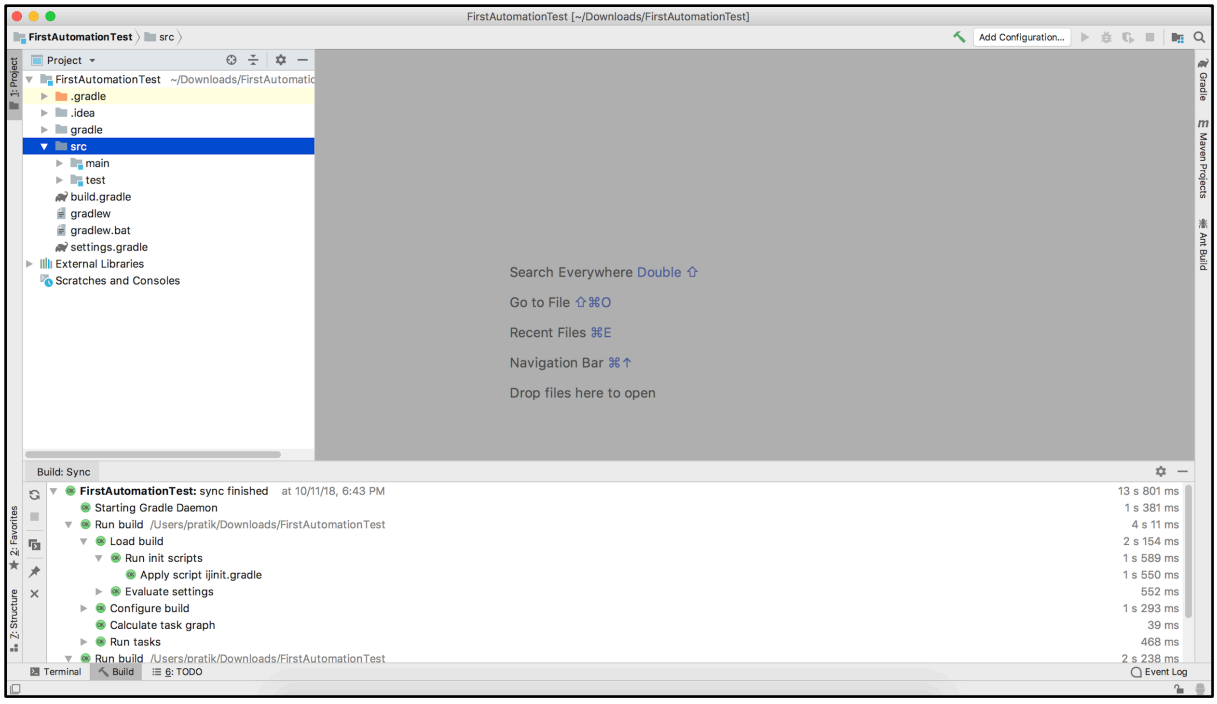

Figure-12: IntelliJ IDEA: Project - First Automation Test

2) Setup the automation case

The first step of any project setup is to download and link the needed libraries/files referred to as dependencies. Gradle and Maven are dependency management tools and have a large number of remote dependencies.

So let's understand how Gradle works:

- Modern software projects rarely build code in isolation. Projects reference modules for the purpose of reusing existing and proven functionality.
- Selected versions of modules are downloaded from dedicated repositories(from remote servers)
- And they are stored in the dependency cache to avoid unnecessary network traffic.

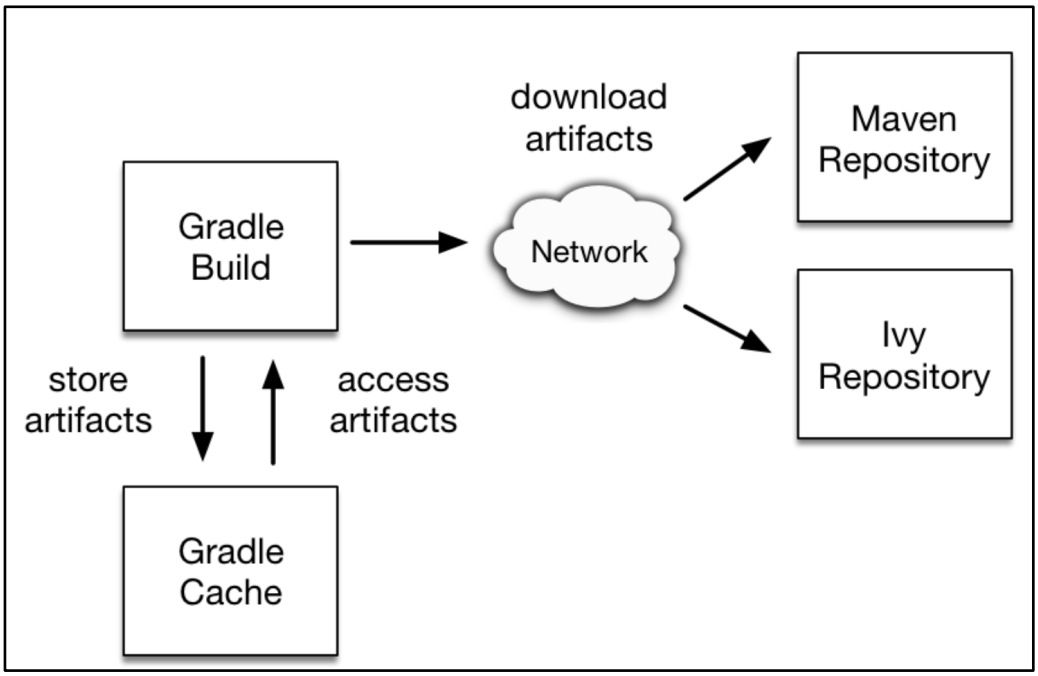

Figure-13: Gradle Build System

Declaring a concrete version of a dependency:

- A typical example for such a library in a Java project is the Automation testing library which is Selenium.

- The following code snippet declares a compile-time dependency on the Spring web module by its coordinates:

build.gradle:

```
sourceCompatibility = 1.8
repositories {
    mavenCentral()
}
dependencies {
    compile group: 'org.seleniumhq.selenium', name:
'selenium-java', version: '3.14.0'
}
```

- The above code snippet declares a compile-time dependency of Selenium library(.JAR file), So when you build the project Gradle will download the *Selenium library having version 3.14.0* from mavenCentral(remote registry) and store it in the Gradle cache, so when you mention this same library with the same version next time on a different project, Gradle will link the library from the Gradle cache.

  Here we need 2 dependencies:

  1) **Appium Java Client** - Mobile Automation Appium Library.
  2) **TestNG** - Test Framework.

Now continue to our project, open **build.gradle** file and add the following dependencies to the build.gradle file.

```
dependencies {
testCompile group: 'io.appium', name: 'java-client', version:
'6.1.0'
testCompile group: 'org.testng', name: 'testng', version:
'6.14.3'
}
```

NOTE-1: *As we already have the TestNG plugin installed, we don't need to mention the TestNG dependency but to be on the safer side and so that this project can also be imported to other IDEs such as Eclipse, and to run through the command line we need to have the dependency mentioned in* ***build.gradle****.*

NOTE-2: *We will be adding the Automation script under **src/test** directory, so in order to link the Appium and TestNG dependencies to that directory we need to use the **testCompile** keyword instead of the **compile** keyword(which compiles the dependencies and made them accessible to **src/main** directory).*

After mentioning the dependencies in the build.gradle, the project will be built and the mentioned dependencies will be downloaded, so now you can use Appium and TestNG classes inside the ***test*** directory.

Figure-14: build.gradle

After dependency management we need to start working on our Appium script. We will start by setting the correct set of Desired Capabilities.

What is "Desired capabilities"?

Desired Capabilities are core to Appium. They are actually a set of keys and values sent to the Appium server to tell the server what kind of automation session should be started. There are various capabilities to modify the behavior of the server during automation.

We have a dedicated chapter on desired capabilities where we will explore them in-depth, but for the sake of getting our first case test we'll use the following desired capabilities:

Android:

```
{
  "platformName": "Android",
  "platformVersion": "8.0",
  "app": "/Users/username/Downloads/sample.apk",
  "deviceName": "c4e3f3cda"
```

```
}
```

iOS:

```
{
  "platformName": "iOS",
  "platformVersion": "11.4.1",
  "app": "/Users/username/Downloads/sample.ipa",
  "deviceName": "John's iPhone",
  "udid": "bea36e2b0262ae4b77bd3463bd462922ee935d24"
}
```

Now let's understand these capabilities:

1. `platformName`- Specifies the Mobile Device Platform to use. (iOS or Android)
2. `platformVersion`- Mobile OS version (8.0, 11.4, 12.1)
3. `app`- The absolute path to the location of the app to test, apk/ipa.(For this example it is under ***src/test/resource*** directory)
4. `deviceName`- Can either refer to an actual mobile device or to an Emulator/Simulator. For Android you can find it using `$ adb devices` command and for iOS you may use `$ instruments -s devices`
5. `udid`- It is the Unique device identifier of the connected physical device.

We will look at both an Android and iOS test case. However, please review both sections even if you are not testing on a particular platform. For each, we explore a different scenario and you will be exposed to different features. If you don't have the specific platform, just read along so that you can get the gist of what we are doing.

Android

1. We will look at an **Android** test first. After setting the valid DesiredCapabilities, we need to pass them to the AndroidDriver class along with the Appium Server URL(By default it is: *http://127.0.0.1:4723/wd/hub*)

AndroidDriver is the main class we will be working with. We will create an instance of AndroidDriver and we will interact with all the UI Elements of the app using that object every time.

And as we are using the TestNG framework we will put these initialization steps before the test starts, so our code will look like this:

FirstAutomationTest | AndroidSampleTest.java

```
1  import io.appium.java_client.AppiumDriver;
2  import io.appium.java_client.android.AndroidDriver;
3  import org.openqa.selenium.By;
4  import org.openqa.selenium.remote.DesiredCapabilities;
5  import org.testng.annotations.BeforeTest;
6  import org.testng.annotations.Test;
7
8  import java.net.MalformedURLException;
9  import java.net.URL;
10
11 public class AndroidSampleTest {
12
13     public AndroidDriver driver;
14
15     @BeforeTest
16     public void setUp() throws MalformedURLException {
17         String appiumServerURL = "http://127.0.0.1:4723/wd/hub";
18
19         DesiredCapabilities dc = new DesiredCapabilities();
20         dc.setCapability( capabilityName: "platformName", value: "Android");
21         dc.setCapability( capabilityName: "platformVersion", value: "8.0");
22         dc.setCapability( capabilityName: "app", value: "/Users/test/Downloads/FirstAutomationTest/src/test/resources/DemoApp.apk");
23         dc.setCapability( capabilityName: "deviceName", value: "c4e3f3cd");
24         dc.setCapability( capabilityName: "automationName", value: "UiAutomator2");
25
26         driver = new AndroidDriver(new URL(appiumServerURL), dc);
27     }
28
```

Figure-15: AndroidDriver initialization and Desired Capabilities.

TestNG provides annotations such as @BeforeTest, @BeforeMethod, @AfterTest, @AfterMethod, @Test etc.

In the above screenshot, ***@BeforeTest*** means that the method will be called before the test, and only once. So it is standard practice to put the AndroidDriver initialization code over there, so the object of AppiumDriver becomes accessible at the end of that method and before the test.

2. Now let's create the first sample Appium Test Case.

   The **@Test** annotation(provided by TestNG) is used to create the individual test case. So in the below code ***firstTest*** is an individual test case.

```
@Test
public void firstTest(){}
```

3. So let's automate a simple scenario. In the below screen we want to click(tap) on Login Screen item from list.

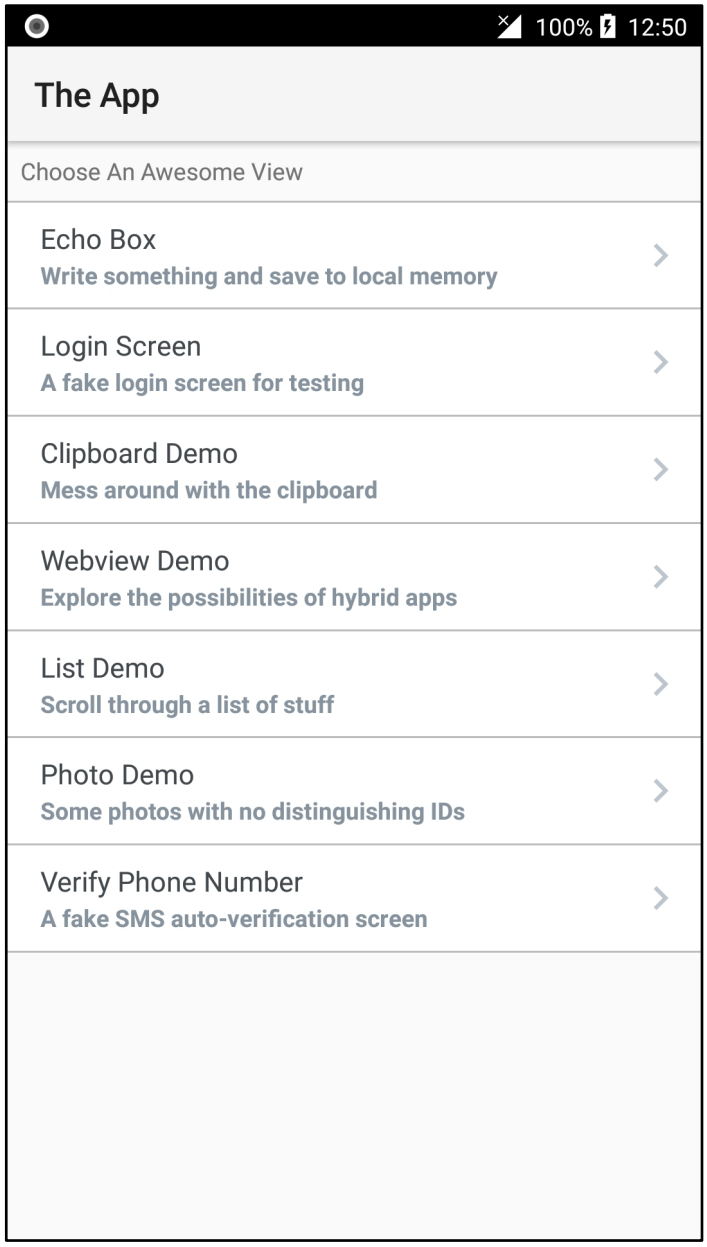

Figure-16: Android - Sample App.

4. After creating the test case we need to add the Appium logic to interact with the UI elements. *In Appium we need each element's locator to interact with*. If you want to tap on some button, you need to find the locator of that button first and then after that you can perform a `click()` action upon it. We will be exploring locators in detail in a subsequent chapter.

   This code will find the Login Screen textview locator and simply click on it:

```
driver.findElement(By.id("Login Screen")).click();
```

Now our First Appium Automation Script is ready to execute, below is the complete code:

```
import io.appium.java_client.AppiumDriver;
import io.appium.java_client.android.AndroidDriver;
import org.openqa.selenium.By;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;

import java.net.MalformedURLException;
import java.net.URL;

public class AndroidSampleTest {

    public AndroidDriver driver;

    @BeforeTest
    public void setUp() throws MalformedURLException {
        String appiumServerURL =
"http://127.0.0.1:4723/wd/hub";

        DesiredCapabilities dc = new DesiredCapabilities();
        dc.setCapability("platformName", "Android");
        dc.setCapability("platformVersion", "8.0");
        dc.setCapability("app",
"/Users/test/Downloads/FirstAutomationTest/src/test/resource
s/DemoApp.apk");
        dc.setCapability("deviceName", "c4e3f3cd");
        dc.setCapability("automationName", "UiAutomator2");

        driver = new AndroidDriver(new URL(appiumServerURL),
dc);
    }

    @Test
    public void firstTest() throws InterruptedException {
        driver.findElement(By.id("Login Screen")).click();
    }
```

```
}
```

5. We are ready to execute it on a real device, so follow these steps:

   a. Move to the Appium Desktop Application and Start the Server.

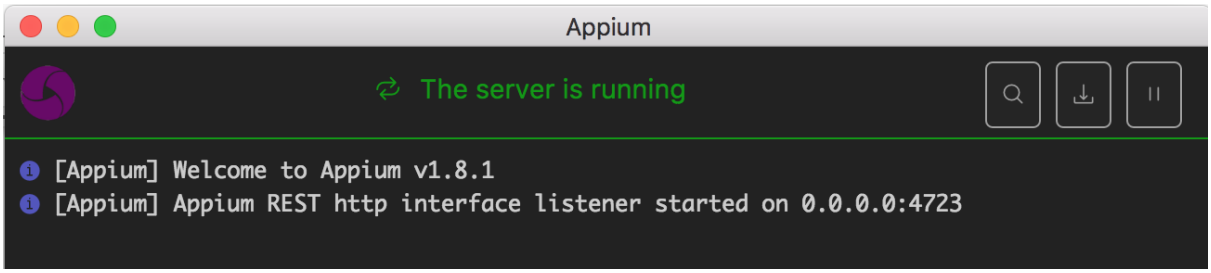

Figure-17: Appium Server is Running on 0.0.0.0:4723

   b. Connect your Android Mobile device to your computer and check that it is connected properly by executing `$ adb devices` command. And also check the **deviceName** capability has the same name of the device which is showing up in the terminal.

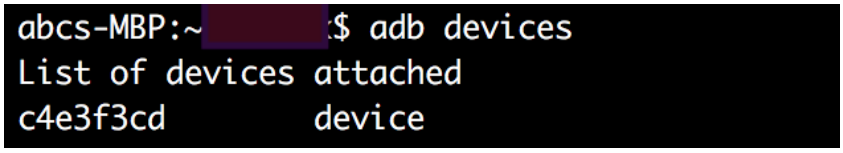

Figure-18: Android device is connected.

   c. Please make sure that device screen is unlocked and that it's connected properly. Now move to intelliJ Idea and select the test case name > Right click on it > Run 'firstTest()'

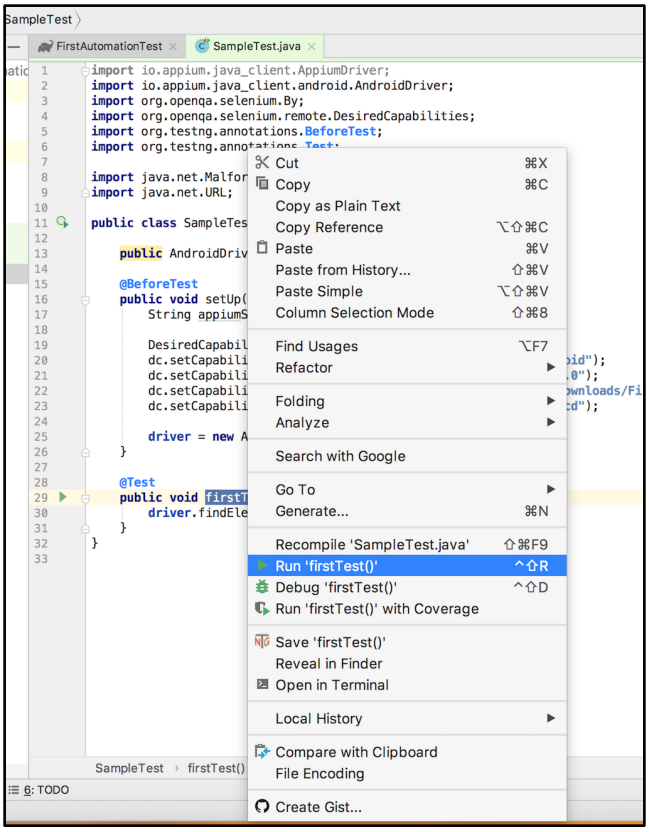

Figure-19: Run the test case.

d. Observe the Test Result and confirm the navigation on your device. It was a simple test case but you've actually accomplished a lot! From here, you get to explore all the cool features that Appium offers.

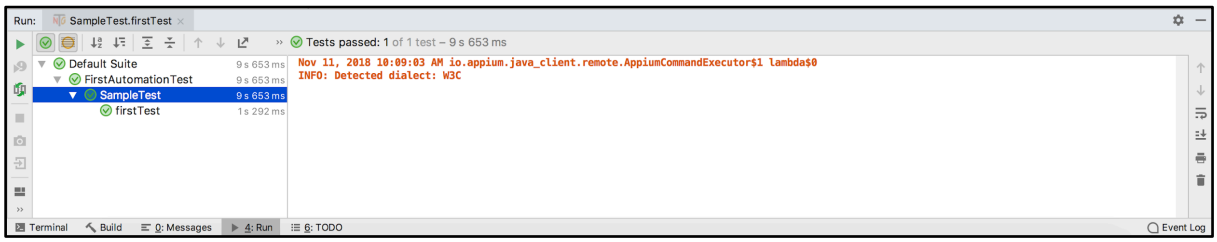

Figure-20: Test Result.

Although we will be turning to iOS next, be sure to read this section even if you are not doing iOS testing. In this example, we get just a little more sophisticated with our test and also expose you to using an *assert* statement.

iOS

1. Let's make our test case a little more sophisticated, while also looking how we can work with iOS. Again, if if you are not planning on using iOS, we suggest you read this section as we'll be introducing new

concepts applicable to both iOS and Android. For our iOS sample test case we will create a separate Test Case file named **iOSSampleTest**

2. As we discussed above we need to put the iOS capabilities instead of Android capabilities, and define an *IOSDriver* class instead of an *AndroidDriver* class.

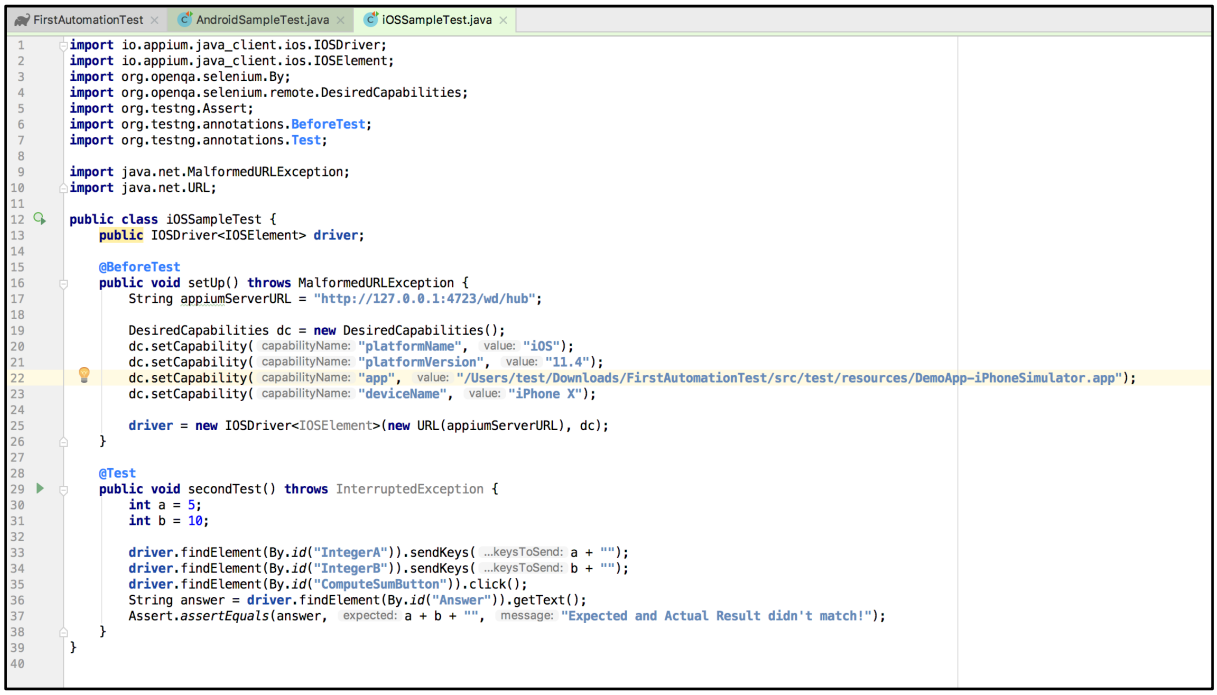

Figure-21:iOSDriver initialization and Desired Capabilities.

3. After specifying the desired capabilities we can write the Automation test case.We have a sample app(.app file, which will work on iOS Simulator only) for automation. In this app there is a feature where you can add 2 integer numbers and can get the the results. So we will automate this feature.

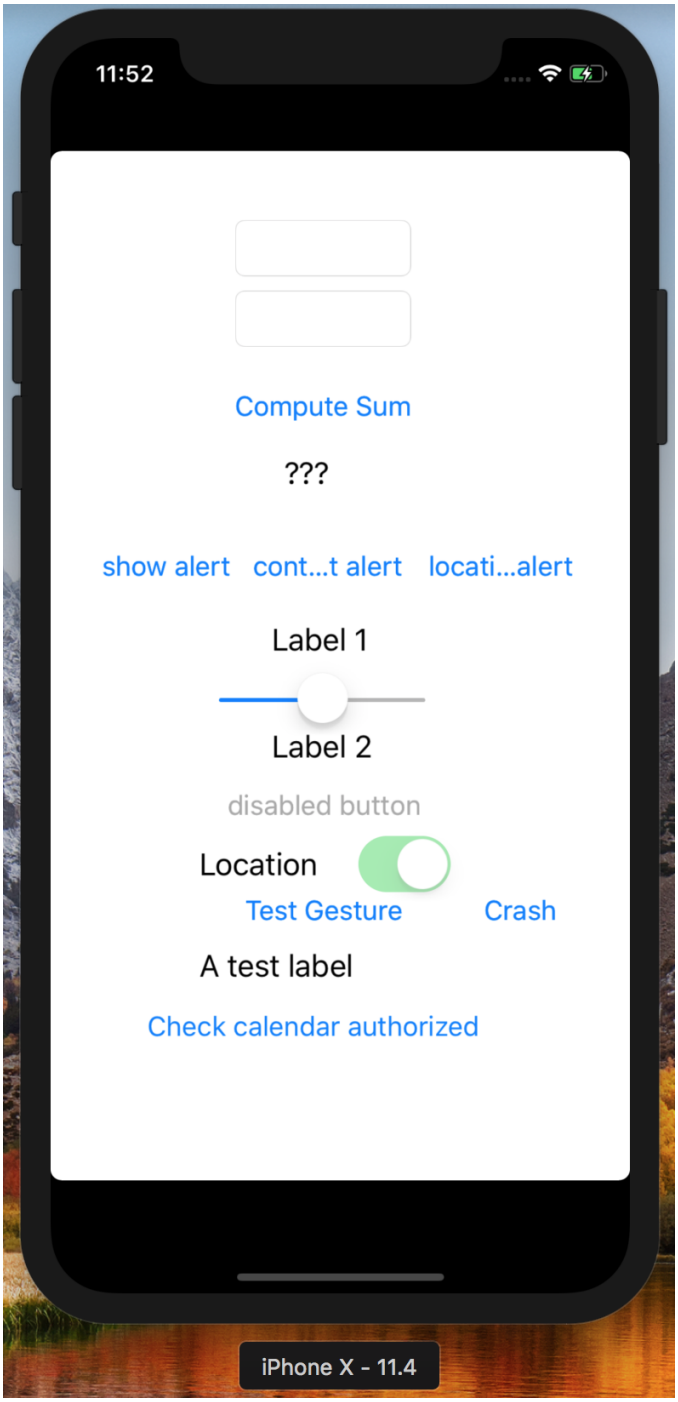

Figure-22: iOS Sample Application

The steps to automate this would be:

a. Find the locator of TextField A and enter the value (ie. Send keys) from the keyboard.

```
driver.findElement(By.id("IntegerA")).sendKeys(5 + "");
```

NOTE: The `sendKeys()` method accepts only ***String*** parameter so we have converted the Integer value to a String by appending a blank String value.

b. Find the locator of TextField B and enter the second value from the keyboard.

```
driver.findElement(By.id("IntegerB")).sendKeys(10 + "");
```

c. Find the locator of *'Compute Sum'* and click on it, so the result would be displayed below the *'Compute Sum'* textview.

```
driver.findElement(By.id("ComputeSumButton")).click();
String answer =
driver.findElement(By.id("Answer")).getText();
```

NOTE: The `getText()` method is used to get the Text(in String format) from UI Elements.

d. Get the text of the result and compare it with the expected result, so if you enter 5 into TextField A, 10 into TextField B and when you click on *'Compute Sum'* textview the result 15 should be displayed under *'Compute Sum'.*

```
Assert.assertEquals(answer, 15 + "", "Expected and Actual
Result didn't match!");
```

NOTE: *`Assert.assertEquals(expected, actual, error_message)` is a TestNG method used to compare the Expected and Actual values. This is the most important step of any test case, because this is how automation test will know whether values are being rendered on UI is correct and as expected or not. You will see us using Assertions throughout this guide.*

TestNG is the Testing framework and work best with Appium(Mobile Automation) and Selenium(Website Automation), you can learn more about the TestNG Annotations and methods here: https://testng.org/doc/index.html

4. Below is the full code of our test which will enter 2 values into text fields, click on *'Compute Result'*, get the result from app and compare it with the expected result.

```
import io.appium.java_client.ios.IOSDriver;
import io.appium.java_client.ios.IOSElement;
import org.openqa.selenium.By;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.testng.Assert;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;

import java.net.MalformedURLException;
import java.net.URL;

public class iOSSampleTest {
    public IOSDriver<IOSElement> driver;

    @BeforeTest
    public void setUp() throws MalformedURLException {
        String appiumServerURL =
"http://127.0.0.1:4723/wd/hub";

        DesiredCapabilities dc = new
DesiredCapabilities();
        dc.setCapability("platformName", "iOS");
        dc.setCapability("platformVersion", "11.4");
        dc.setCapability("app",
"/Users/pratik/Downloads/FirstAutomationTest/src/test/reso
urces/DemoApp-iPhoneSimulator.app");
        dc.setCapability("deviceName", "iPhone X");

        driver = new IOSDriver<IOSElement>(new
URL(appiumServerURL), dc);
    }

    @Test
    public void secondTest() throws InterruptedException {
        int a = 5;
        int b = 10;

        driver.findElement(By.id("IntegerA")).sendKeys(a +
"");
        driver.findElement(By.id("IntegerB")).sendKeys(b +
```

```
        "");

        driver.findElement(By.id("ComputeSumButton")).click();
                String answer =
        driver.findElement(By.id("Answer")).getText();
                Assert.assertEquals(answer, a + b + "", "Expected
        and Actual Result didn't match!");
            }
        }
```

You can get this example code on our github page.

Phew! We covered a lot of material in this chapter.

We learned the following:

1. Installation of IntelliJ IDEA.
2. TestNG plugin installation on IntelliJ IDEA.
3. Setting up the Appium Project on IntelliJ IDEA.
4. Writing the first Automation Test on Android Real Device.
5. Writing the first Automation Test on iOS Simulator.

Along the way you learned a little bit about desired capabilities, locators and assertions. All of this is a great grounding to continue your education into the world of Automated testing and Appium.

Take a break and let's continue our journey when you get back.

Chapter-3: Understanding the Desired Capabilities

In the previous chapter you were briefly introduced to Desired Capabilities. This is a core capability of Appium. In this chapter we will take a deep dive into this feature.

Desired Capabilities help us to configure the Appium server and provide the criteria which we wish to use for running our automation script. For example, we can request the environment (emulator or real-device), which version of the operating system to run the test on, and more. Desired Capabilities are key/value pairs encoded in JSON format and are sent to the Appium Server by the Appium client when a new automation session is requested.

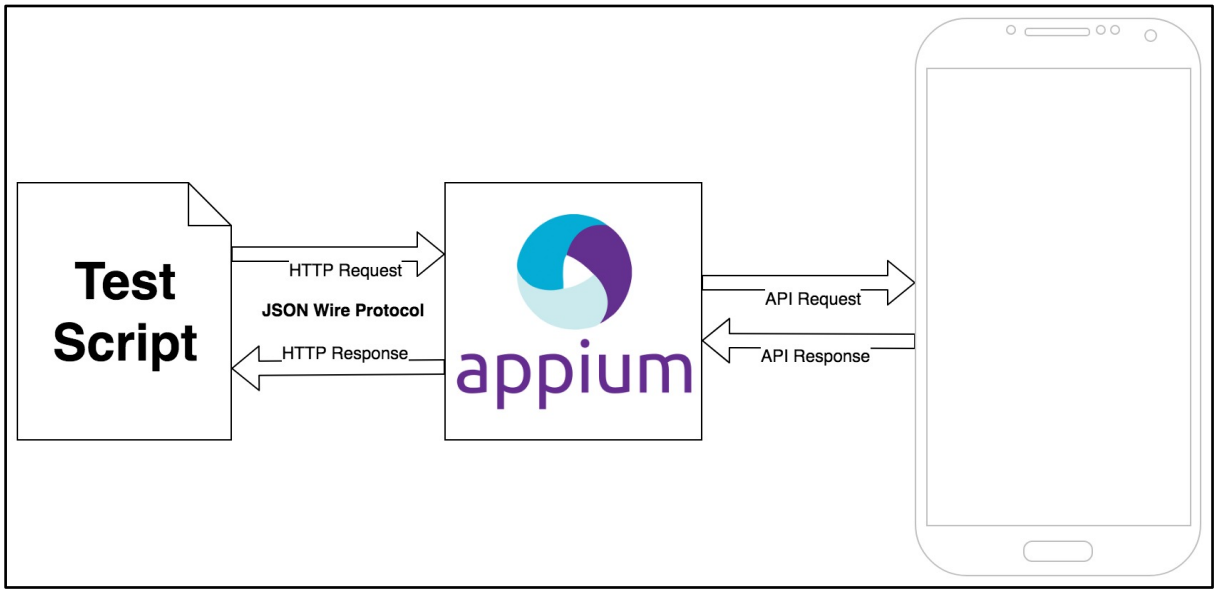

Figure-1:Appium Architecture.

DesiredCapabilities is a predefined library and in order to use it you need to import:

```
Org.openqa.selenium.remote.DesiredCapabilities
```

As Appium supports both Android and iOS, There are separate capabilities for both. However most of the capabilities remain common to both platforms.

Android:

```
{
  "platformName": "Android",
  "platformVersion": "8.0",
  "app": "/Users/username/Downloads/sample.apk",
  "deviceName": "c4e3f3cda"
  "automationName": "UiAutomator2"
}
```

iOS:

```
{
  "platformName": "iOS",
  "platformVersion": "11.4.1",
  "app": "/path/to/.ipa/file",
  "deviceName": "John's iPhone",
  "udid": "bea36e2b0262ae4b77bd3463bd462922ee935d24"
  "automationName": "XCUITest"
}
```

The above is the JSON representation. To use this in code you can define the Desired Capabilities (for Android) as below:

```
DesiredCapabilities dc = new DesiredCapabilities();
dc.setCapability("platformName", "Android");
dc.setCapability("platformVersion", "8.0");
dc.setCapability("app",
"/Users/test/Downloads/FirstAutomationTest/src/test/resources
/DemoApp.apk");
dc.setCapability("deviceName", "c4e3f3cd");
dc.setCapability("automationName", "UiAutomator2");
```

In the above code snippet instead of defining Capability Name in a String you can use the Appium predefined interfaces such as *MobileCapabilityType*, *IOSMobileCapabilityType* and *AndroidMobileCapabilityType* to get Capability Names, so the above code snippet can be written in a better way:

```
DesiredCapabilities dc = new DesiredCapabilities();
dc.setCapability(MobileCapabilityType.PLATFORM_NAME,
```

```
"Android");
dc.setCapability(MobileCapabilityType.PLATFORM_VERSION,
"8.0");
dc.setCapability(MobileCapabilityType.APP,
"/Users/test/Downloads/FirstAutomationTest/src/test/resour
ces/DemoApp.apk");
dc.setCapability(MobileCapabilityType.DEVICE_NAME,
"c4e3f3cd");
dc.setCapability(MobileCapabilityType.AUTOMATION_NAME,
"UiAutomator2");
```

Desired capabilities for iOS and Android

We have many desired capabilities at our disposal depending on what we are trying to accomplish. However, there are specific capabilities we need to set based on what we are trying to test:

1) Mobile Web Android.
2) Mobile Web iOS.
3) Mobile Native Android.
4) Mobile Native iOS.

1) Mobile web - Android

If you want to automate the Chrome browser on Android device then you need to use these Desired Capabilities.

```
DesiredCapabilities dc = new DesiredCapabilities();
dc.setCapability(MobileCapabilityType.PLATFORM_NAME,
"Android");
dc.setCapability(MobileCapabilityType.PLATFORM_VERSION,
"OS version of your test device/simulator");
dc.setCapability(MobileCapabilityType.DEVICE_NAME, "Name
of your test device");
// This capability will open the Chrome browser instead of
Native app.
dc.setCapability(MobileCapabilityType.BROWSER_NAME,
MobileBrowserType.CHROME);
dc.setCapability(MobileCapabilityType.AUTOMATION_NAME,
"UiAutomator2");
```

2) Mobile web - iOS

```
DesiredCapabilities dc = new DesiredCapabilities();
dc.setCapability(MobileCapabilityType.PLATFORM_NAME,
"iOS");
dc.setCapability(MobileCapabilityType.PLATFORM_VERSION,
"OS version of your test device/simulator");
dc.setCapability(MobileCapabilityType.DEVICE_NAME, "Name
of your test device");
// This capability will open the Safari browser instead of
Native app.
dc.setCapability(MobileCapabilityType.BROWSER_NAME,
MobileBrowserType.SAFARI);
dc.setCapability(MobileCapabilityType.AUTOMATION_NAME,
"XCUITest");
// If you are using Real iPhone device then you need to
specify "udid" of device.
dc.setCapability("udid", "UDID of your test device");
```

3) Mobile native - Android

```
DesiredCapabilities dc = new DesiredCapabilities();
dc.setCapability(MobileCapabilityType.PLATFORM_NAME,
"Android");
dc.setCapability(MobileCapabilityType.PLATFORM_VERSION,
"OS version of your test device/simulator");
dc.setCapability(MobileCapabilityType.APP,
"/path/to/.apk/file");
dc.setCapability(MobileCapabilityType.DEVICE_NAME, "Name
of your test device");
dc.setCapability(MobileCapabilityType.AUTOMATION_NAME,
"UiAutomator2");
```

4) Mobile native - iOS

```
DesiredCapabilities dc = new DesiredCapabilities();
dc.setCapability(MobileCapabilityType.PLATFORM_NAME, "iOS");
dc.setCapability(MobileCapabilityType.PLATFORM_VERSION, "OS
version of your test device/simulator");
dc.setCapability(MobileCapabilityType.APP, "/path/to/.app or
.ipa/file");
dc.setCapability(MobileCapabilityType.DEVICE_NAME, "Name of
your test device");
```

```
dc.setCapability(MobileCapabilityType.AUTOMATION_NAME,
"XCUITest");
```

List of all capabilities

There are many, many capabilities that Appium supports. We can categorize the Capabilities into 3 parts:

1) General Capabilities.
2) iOS Capabilities.
3) Android Capabilities.

While you clearly don't need to memorize all of these, we do suggest you spend the time to familiarize yourself with these capabilities. As you use Appium more and more and continue to consult this list, you will eventually have a good sense of what capabilities are available to you.

General capabilities

Capability	Description	Values
automationName	Which automation engine to use	Appium (default) or Selendroid or UiAutomator2 or Espresso for Android or XCUITest for iOS
platformName	Which mobile OS platform to use	iOS, Android, or Firefox OS
platformVersion	Mobile OS version	e.g., 7.1, 4.4
deviceName	The kind of mobile device or emulator to use	iPhone Simulator, iPad Simulator, iPhone Retina 4-inch, Android Emulator, Galaxy S4, etc.... On

		iOS, this should be one of the valid devices returned by instruments with instruments -s devices. On Android this capability is currently ignored, though it remains required.
app	The absolute local path *or* remote http URL to a .ipa file (IOS), .app folder (IOS Simulator) or .apk file (Android), or a .zip file containing one of these (for .app, the .app folder must be the root of the zip file). Appium will attempt to install this app binary on the appropriate device first. Note that this capability is not required for Android if you specify appPackage and appActivity capabilities (see below). Incompatible with browserName.	/abs/path/to/my.apk or http://myapp.com/app.ipa
browserName	Name of mobile web browser to automate. Should be an empty string if automating an app instead.	'Safari' for iOS and 'Chrome', 'Chromium', or 'Browser' for Android
newCommandTimeout	How long (in seconds) Appium will wait for a new command from the client before assuming the client quit and ending the session	e.g. 60
language	(Sim/Emu-only) Language to set for the simulator / emulator. On Android, available only on API levels 22 and below	e.g. fr
locale	(Sim/Emu-only) Locale to set for the	e.g. fr_CA

	simulator / emulator.	
udid	Unique device identifier of the connected physical device	e.g. 1ae203187fc012g
orientation	(Sim/Emu-only) start in a certain orientation	LANDSCAPE or PORTRAIT
autoWebview	Move directly into Webview context. Default false	true, false
noReset	Don't reset app state before this session. See here for more details	true, false
fullReset	Perform a complete reset. See here for more details	true, false
eventTimings	Enable or disable the reporting of the timings for various Appium-internal events (e.g., the start and end of each command, etc.). Defaults to false. To enable, use true. The timings are then reported as events property on response to querying the current session. See the event timing docs for the the structure of this response.	e.g., true
enablePerformanceLogging	(Web and webview only) Enable Chromedriver's (on Android) or Safari's (on iOS) performance logging (default false)	true, false
printPageSourceOnFindFailure	When a find operation fails, print the current page source. Defaults to false.	e.g., true

Android capabilities

These Capabilities are available only on Android-based drivers (like UiAutomator2 for example).

Capability	Description	Values
appActivity	Activity name for the Android activity you want to launch from your package. This often needs to be preceded by a . (e.g., .MainActivity instead of MainActivity). By default this capability is received from the package manifest (action: android.intent.action.MAIN , category: android.intent.category.LAUNCHER)	MainActivity, .Settings
appPackage	Java package of the Android app you want to run. By default this capability is received from the package manifest (@package attribute value)	com.example.android.myApp, com.android.settings
appWaitActivity	Activity name/names, comma separated, for the Android activity you want to wait for. By default the value of this capability is the same as for appActivity. You must set it to the very first focused application activity name in case it is different from the one which is set as appActivity if your capability has appActivity and appPackage.	SplashActivity, SplashActivity,OtherActivity, *, *.SplashActivity
appWaitPackage	Java package of the Android app you want to wait for. By default the value of this capability is the same as for	com.example.android.myApp, com.android.settings

	appActivity	
appWaitDuration	Timeout in milliseconds used to wait for the appWaitActivity to launch (default 20000)	30000
deviceReadyTimeout	Timeout in seconds while waiting for device to become ready	5
androidCoverage	Fully qualified instrumentation class. Passed to -w in adb shell am instrument -e coverage true -w	com.my.Pkg/com.my.Pkg.instrumentation.MyInstrumentation
androidCoverageEndIntent	A broadcast action implemented by yourself which is used to dump coverage into file system. Passed to -a in adb shell am broadcast -a	com.example.pkg.END_EMMA
androidDeviceReadyTimeout	Timeout in seconds used to wait for a device to become ready after booting	e.g., 30
androidInstallTimeout	Timeout in milliseconds used to wait for an apk to install to the device. Defaults to 90000	e.g., 90000
androidInstallPath	The name of the directory on the device in which the apk will be push before install. Defaults to /data/local/tmp	e.g. /sdcard/Downloads/
adbPort	Port used to connect to the ADB server (default 5037)	5037

systemPort	systemPort used to connect to appium-uiautomator2-server, default is 8200 in general and selects one port from 8200 to 8299. When you run tests in parallel, you must adjust the port to avoid conflicts. Read Parallel Testing Setup Guide for more details.	e.g., 8201
remoteAdbHost	Optional remote ADB server host	e.g.: 192.168.0.101
androidDeviceSocket	Devtools socket name. Needed only when tested app is a Chromium embedding browser. The socket is open by the browser and Chromedriver connects to it as a devtools client.	e.g., chrome_devtools_remote
avd	Name of avd to launch	e.g., api19
avdLaunchTimeout	How long to wait in milliseconds for an avd to launch and connect to ADB (default 120000)	300000
avdReadyTimeout	How long to wait in milliseconds for an avd to finish its boot animations (default 120000)	300000
avdArgs	Additional emulator arguments used when launching an avd	e.g., -netfast
useKeystore	Use a custom keystore to sign apks, default false	true or false

keystorePath	Path to custom keystore, default ~/.android/debug.keystore	e.g., /path/to.keystore
keystorePassword	Password for custom keystore	e.g., foo
keyAlias	Alias for key	e.g., androiddebugkey
keyPassword	Password for key	e.g., foo
chromedriverExecutable	The absolute local path to webdriver executable (if Chromium embedder provides its own webdriver, it should be used instead of original chromedriver bundled with Appium)	/abs/path/to/webdriver
chromedriverExecutableDir	The absolute path to a directory to look for Chromedriver executables in, for automatic discovery of compatible Chromedrivers. Ignored if chromedriverUseSystemExecutableis true	/abs/path/to/chromedriver/directory
chromedriverChromeMappingFile	The absolute path to a file which maps Chromedriver versions to the minimum Chrome that it supports. Ignored if chromedriverUseSystemExecutableis true	/abs/path/to/mapping.json
chromedriverUseSystemExecutable	If true, bypasses automatic Chromedriver configuration and uses the version that comes downloaded with Appium. Ignored if chromedriverExecutable is set. Defaults to false	e.g., true
autoWebviewTimeout	Amount of time to wait for Webview context to become active, in ms. Defaults to 2000	e.g. 4

intentAction	Intent action which will be used to start activity (default android.intent.action.MAIN)	e.g.android.intent.action.MAIN, android.intent.action.VIEW
intentCategory	Intent category which will be used to start activity (default android.intent.category.LAUNCHER)	e.g. android.intent.category.LAUNCHER, android.intent.category.APP_CONTACTS
intentFlags	Flags that will be used to start activity (default 0x10200000)	e.g. 0x10200000
optionalIntentArguments	Additional intent arguments that will be used to start activity. See Intent arguments	e.g. --esn <EXTRA_KEY>, --ez <EXTRA_KEY> <EXTRA_BOOLEAN_VALUE>, etc.
dontStopAppOnReset	Doesn't stop the process of the app under test, before starting the app using adb. If the app under test is created by another anchor app, setting this false, allows the process of the anchor app to be still alive, during the start of the test app using adb. In other words, with dontStopAppOnReset set to true, we will not include the -Sflag in the adb shell am start call. With this capability omitted or set to false, we include the -S flag. Default false	true or false
unicodeKeyboard	Enable Unicode input, default false	true or false
resetKeyboard	Reset keyboard to its original state, after running Unicode tests with unicodeKeyboard capability. Ignored if	true or false

	used alone. Default false	
noSign	Skip checking and signing of app with debug keys, will work only with UiAutomator and not with selendroid, default false	true or false
ignoreUnimportantViews	Calls the setCompressedLayoutHierarchy()uiautomator function. This capability can speed up test execution, since Accessibility commands will run faster ignoring some elements. The ignored elements will not be findable, which is why this capability has also been implemented as a toggle-able *setting* as well as a capability. Defaults to false	true or false
disableAndroidWatchers	Disables android watchers that watch for application not responding and application crash, this will reduce cpu usage on android device/emulator. This capability will work only with UiAutomator and not with selendroid, default false	true or false
chromeOptions	Allows passing chromeOptions capability for ChromeDriver. For more information see chromeOptions	chromeOptions: {args: ['--disable-popup-blocking']}
recreateChromeDriverSessions	Kill ChromeDriver session when moving to a non-ChromeDriver webview. Defaults to false	true or false
nativeWebScreenshot	In a web context, use native (adb) method for taking a screenshot, rather than proxying to ChromeDriver. Defaults	true or false

	to false	
androidScreenshotPath	The name of the directory on the device in which the screenshot will be put. Defaults to /data/local/tmp	e.g. /sdcard/screenshots/
autoGrantPermissions	Have Appium automatically determine which permissions your app requires and grant them to the app on install. Defaults to false. If noReset is true, this capability doesn't work.	true or false
networkSpeed	Set the network speed emulation. Specify the maximum network upload and download speeds. Defaults to full	['full','gsm', 'edge', 'hscsd', 'gprs', 'umts', 'hsdpa', 'lte', 'evdo']Check -netspeed option more info about speed emulation for avds
gpsEnabled	Toggle gps location provider for emulators before starting the session. By default the emulator will have this option enabled or not according to how it has been provisioned.	true or false
isHeadless	Set this capability to true to run the Emulator headless when device display is not needed to be visible. false is the default value. *isHeadless* is also support for iOS, check XCUITest-specific capabilities.	e.g., true
uiautomator2ServerLaunchTimeout	Timeout in milliseconds used to wait for an uiAutomator2 server to launch.	e.g., 20000

	Defaults to 20000	
uiautomator2ServerInstallTimeout	Timeout in milliseconds used to wait for an uiAutomator2 server to be installed. Defaults to 20000	e.g., 20000
otherApps	App or list of apps (as a JSON array) to install prior to running tests	e.g., "/path/to/app.apk", https://www.example.com/url/to/app.apk, ["/path/to/app-a.apk", "/path/to/app-b.apk"]

iOS capabilities

These Capabilities are available only on the XCUITest Driver and the deprecated UIAutomation Driver.

Capability	Description	Values
calendarFormat	(Sim-only) Calendar format to set for the iOS Simulator	e.g. gregorian
bundleId	Bundle ID of the app under test. Useful for starting an app on a real device or for using other caps which require the bundle ID during test startup. To run a test on a real device using the bundle ID, you may omit the 'app' capability, but you must provide 'udid'.	e.g. io.appium.TestApp
udid	Unique device identifier of the connected physical device	e.g. 1ae203187fc012g

launchTimeout	Amount of time in ms to wait for instruments before assuming it hung and failing the session	e.g. 20000
locationServicesEnabled	(Sim-only) Force location services to be either on or off. Default is to keep current sim setting.	true or false
locationServicesAuthorized	(Sim-only) Set location services to be authorized or not authorized for app via plist, so that location services alert doesn't pop up. Default is to keep current sim setting. Note that if you use this setting you MUST also use the bundleId capability to send in your app's bundle ID.	true or false
autoAcceptAlerts	Accept all iOS alerts automatically if they pop up. This includes privacy access permission alerts (e.g., location, contacts, photos). Default is false. Does not work on XCUITest-based tests.	true or false
autoDismissAlerts	Dismiss all iOS alerts automatically if they pop up. This includes privacy access permission alerts (e.g., location, contacts, photos). Default is false. Does not work on XCUITest-based tests.	true or false
nativeInstrumentsLib	Use native intruments lib (ie disable instruments-without-delay).	true or false
nativeWebTap	(Sim-only) Enable "real", non-javascript-based web taps in Safari. Default: false. Warning: depending on viewport size/ratio this might not accurately tap an element	true or false

safariInitialUrl	(Sim-only) (>= 8.1) Initial safari url, default is a local welcome page	e.g. https://www.github.com
safariAllowPopups	(Sim-only) Allow javascript to open new windows in Safari. Default keeps current sim setting	true or false
safariIgnoreFraudWarning	(Sim-only) Prevent Safari from showing a fraudulent website warning. Default keeps current sim setting.	true or false
safariOpenLinksInBackground	(Sim-only) Whether Safari should allow links to open in new windows. Default keeps current sim setting.	true or false
keepKeyChains	(Sim-only) Whether to keep keychains (Library/Keychains) when appium session is started/finished	true or false
localizableStringsDir	Where to look for localizable strings. Default en.lproj	en.lproj
processArguments	Arguments to pass to the AUT using instruments	e.g., -myflag
interKeyDelay	The delay, in ms, between keystrokes sent to an element when typing.	e.g., 100
showIOSLog	Whether to show any logs captured from a device in the appium logs. Default false	true or false

sendKeyStrategy	strategy to use to type test into a test field. Simulator default: oneByOne. Real device default: grouped	oneByOne, grouped or setValue
screenshotWaitTimeout	Max timeout in sec to wait for a screenshot to be generated. default: 10	e.g., 5
waitForAppScript	The ios automation script used to determined if the app has been launched, by default the system wait for the page source not to be empty. The result must be a boolean	e.g. true;, target.elements().length > 0;, $.delay(5000); true;
webviewConnectRetries	Number of times to send connection message to remote debugger, to get webview. Default: 8	e.g., 12
appName	The display name of the application under test. Used to automate backgrounding the app in iOS 9+.	e.g., UICatalog
customSSLCert	(Sim only) Add an SSL certificate to IOS Simulator.	e.g. -----BEGIN CERTIFICATE----- MIIFWjCCBEKg... -----END CERTIFICATE-----
webkitResponseTimeout	(Real device only) Set the time, in ms, to wait for a response from WebKit in a Safari session. Defaults to 5000	e.g., 10000

remoteDebugProxy	(Sim only, <= 11.2) If set, Appium sends and receives remote debugging messages through a proxy on either the local port (Sim only, <= 11.2) or a proxy on this unix socket (Sim only >= 11.3) instead of communicating with the iOS remote debugger directly.	e.g. 12000 or "/tmp/my.proxy.socket"

Important capabilities

1) Reset strategies

In Mobile Application Automation, most of the execution time is spent on Application installation. Sometimes you do not want to reinstall the application (like between tests) so Appium has provided 2 capabilities named **noReset** and **fullReset** which provides control over application installation and you can leverage the right combination of the two flags.

noReset	fullReset	Result on iOS	Result on Android
true	true	Error: The 'noReset' and 'fullReset' capabilities are mutually exclusive and should not both be set to true	
true	false	Do not destroy or shut down simulator after test. Start tests running on whichever simulator is running, or device is plugged in.	Do not stop app, do not clear app data, and do not uninstall apk.
false	true	Uninstall app after real device test, destroy Simulator after sim test.	Stop app, clear app data and uninstall apk after test.
false	false	Shut down simulator after test. Do not destroy simulator. Do not uninstall app from real device.	Stop and clear app data after test. Do not uninstall apk

NOTE: *You can know more about Appium Capabilities on Official Appium Docs periodically: http://appium.io/docs/en/writing-running-appium/caps/*

The following additional capabilities are reprinted with permission from Jonathan Lipps, Founding Principal of Cloud Grey, a mobile testing services company. Refer to footnote for the source link.

2) Android-specific capabilities[1]

disableAndroidWatchers:
The only way to check for toast messages on Android is for the Appium UiAutomator2 driver to run a loop constantly checking the state of the device. Running a loop like this takes up valuable CPU cycles and has been observed to make scrolling less consistent, for example. If you don't need the features that require the watcher loop (like toast verification), then set this cap to true to turn it off entirely and save your device some cycles.

autoGrantPermission:
Set to true to have Appium attempt to automatically determine your app permissions and grant them, for example to avoid system pop ups asking for permission later on in the test.

skipUnlock:
Appium doesn't assume that your device is unlocked, and it should be to successfully run tests. So it installs and runs a little helper app that tries to unlock the screen before a test. Sometimes this works, and sometimes this doesn't. But that's beside the point: either way, it takes time! If you know your screen is unlocked, because you're managing screen state with something other than Appium, tell Appium not to bother with this little startup routine and save yourself a second or three, by setting this cap to true.

appWaitPackage and appWaitActivity:
Android activities can be kind of funny. In many apps, the activity used to launch the app is not the same as the activity which is active when the user initially interacts with the application. Typically it's this latter activity you care about when you run an Appium test. You want to make sure that Appium doesn't consider the session started until this activity is active, regardless of what happened to the launch activity.

In this scenario, you need to tell Appium to wait for the correct activity, since the one it automatically retrieves from your app manifest will be the launch activity. You can use the appWaitPackage and appWaitActivity to tell Appium to consider a session started (and hence return control to your test code) only when the package and activity specified have become active. This can

[1] From *Jonathan Lipps' blog:* https://appiumpro.com/editions/24

greatly help the stability of session start, because your test code can assume your app is on the activity expects when the session starts.

ignoreUnimportantViews:
Android has two modes for expressing its layout hierarchy: normal and "compressed". The compressed layout hierarchy is a subset of the hierarchy that the OS itself sees, restricted to elements which the OS thinks are more relevant for users, for example elements with accessibility information set on them. Because compressed mode generates a smaller XML file, and perhaps for other Android-internal reasons, it's often faster to get the hierarchy in compressed mode. If you're running into page source queries taking a very long time, you might try setting this cap to true.

Note that the XML returned in the different modes is ... different. Which means that XPath queries that worked in one mode will likely not work in the other. Make sure you don't change this back and forth if you rely on XPath!

3) iOS-specific capabilities[2]

usePrebuiltWDA and derivedDataPath:
Typically, Appium uses xcodebuild under the hood to both build WebDriverAgent and kick off the XCUITest process that powers the test session. If you have a prebuilt WebDriverAgent binary and would like to save some time on startup, set the usePrebuiltWDA cap to true. This cap could be used in conjunction with derivedDataPath, which is the path to the derived data folder where your WebDriverAgent binary is dumped by Xcode.

useJSONSource:
For large applications, it can be faster for Appium to deal with the app hierarchy internally as JSON, rather than XML, and convert it to XML at the "edge", so to speak---in the Appium driver itself, rather than lower in the stack. Basically, give this a try if getting the iOS app source is taking forever.

iosInstallPause:
Sometimes, large iOS applications can take a while to launch, but there's no way for Appium to automatically detect when an app is ready for use or not. If you have such an app, set this cap to the number of milliseconds you'd like Appium to wait after WebDriverAgent thinks the app is online, before Appium hands back control to your test script. It might help make session startup a bit more stable.

maxTypingFrequency:
If you notice errors during typing, for example the wrong keys being pressed

[2] *Jonathan Lipps' blog: https://appiumpro.com/editions/24*

or visual oddities you notice while watching a test, try slowing the typing down. Set this cap to an integer and play around with the value until things work. Lower is slower, higher is faster! The default is 60.

realDeviceScreenshotter:
Appium has its own methods for capturing screenshots from simulators and devices, but especially on real devices this can be slow and/or flaky. If you're a fan of the libimobiledevice suite and happen to have idevicescreenshot on your system, you can use this cap to let Appium know you'd prefer to retrieve the screenshot via a call to that binary instead of using its own internal methods. To make it happen, simply set this cap to the string "idevicescreenshot"!

simpleIsVisibleCheck:
Element visibility checks in XCUITest are fraught with flakiness and complexity. By default, the visibility checks available don't always do a great job. Appium implemented another type of visibility check inside of WebDriverAgent that might be more reliable for your app, though it comes with the downside that the checks could take longer for some apps. As with many things in life, we sometimes have to make trade-offs between speed and reliability.

NOTE: *You can learn more about Android & iOS specific Capabilities on Jonathan Lipps' (Appium project lead and architect) blog:*
https://appiumpro.com/editions/24

We suggest reviewing these capabilities and familiarize yourself with them. It isn't necessary to memorize them, but as you get more sophisticated testing needs it will be good to keep coming back here to see which one will do the trick. We'll be using various forms of these capabilities throughout the rest of the book so you will start getting more familiar with them as we work through more examples.

Chapter-4: Appium Locator Finding Strategies

Understanding how to properly use Locators is key to building your automation scripts. After all, if you're unable to "find" the UI element, you cannot control it (such as clicking a button).

In Mobile (or Web) Automation Testing automating any scenario follows these 2 steps:

1) **Find the UI element locators (uniquely).**
2) **Perform an action on that element.**

In this chapter we focus on the first step and will look into all the available Locator Finding Strategies and discuss each strategy's pros and cons.

So What is an Element Locator?

An Element Locator is nothing but an address that identifies a UI Element on a Mobile App (or Website). As there are many UI elements present on a single mobile application screen there can be a chance that same (generic)address can refer to more than one element. This means that we need to find a unique address for the element. As you will see, sometimes this is easy, and other times you have to do some further exploration to uniquely identify your UI element. The way in which you uniquely identify the element is called a locator strategy. Appium makes many different strategies available.

If you recall our simple test cases in Chapter 2, our Android example used the following code for identifying the Textview:

```
driver.findElement(By.id("Login Screen")).click();
```

Here `id` is the **Locator strategy** and `Login Screen` is the unique id(address). Think of reading it as "Finding the element by <locator strategy> <element unique id>". So in this example we're telling Appium to use the "id" strategy (used for finding elements by unique ID) and the ID we're using is "Login Screen".

The below image describes how can you find the Textview element for any android application (in Java).

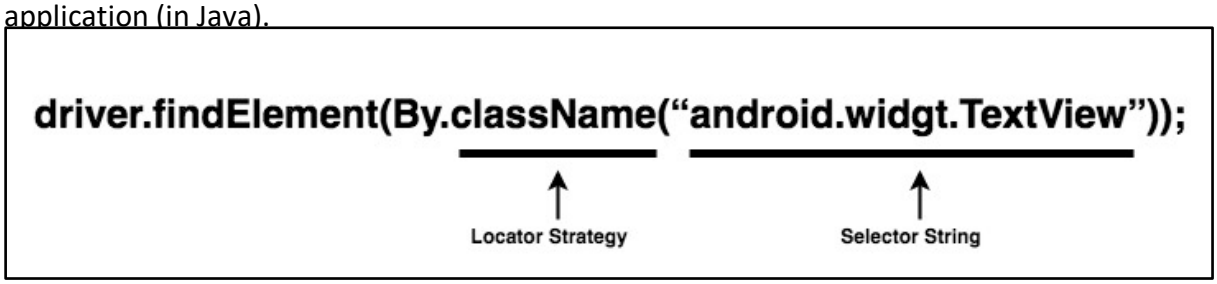

Figure-1:Locator Strategy(Java Example).

As you may expect, there are many different locator strategies available to you, including:

1) Accessibility ID
2) Class name
3) ID
4) Name
5) XPath
6) Image (Recently Introduced)
7) Android UiAutomator (UiAutomator2 only)
8) Android View Tag (Espresso only)
9) IOS UIAutomation

Learning which type of Locator Strategy to use is part of the learning process of becoming comfortable with Appium. We will go through all Locator Strategies and discuss them in detail. Don't worry about memorizing all of them … at this stage in your journey you just need to become familiar with them and eventually you'll understand which are best to use in which scenarios. In fact, there are some tricks coming up later that will automatically suggest which strategy to use! Often during your script development you'll wrestle with trying to identify a UI element. When that happens, refer back to these different locator strategies to see which might best fit your needs.

NOTE: *All of the above Locator Strategies can be inspected using the* ***Appium Inspector Tool****(for Android and iOS both). We will learn about that tool in the next chapter. The screenshots that follow are using this Inspector to illustrate the locator. The mobile app is depicted in the leftmost pane and when clicking an element we see the attributes in the rightmost pane.*

1) Accessibility ID

- ***This is the best preferred locator strategy in Appium. Always use this one if you can.***

- ***It is a Cross-platform locator strategy*** as this works in a similar way on iOS and Android which makes your code more reusable.

- **iOS:** If the `accessibility id` property(attribute) value is set at development time (by the app developers) then you can inspect it using the ***Appium Inspector***(Android & iOS) or ***UiAutomatorViewer***(Android). When `Accessibility Id` property value is not defined by developer, it is by default equals to the ***Name*** of that UI Element.

- **Android:** ***Accessibility Id*** property is equals to `content-desc` property(attribute) on Android.

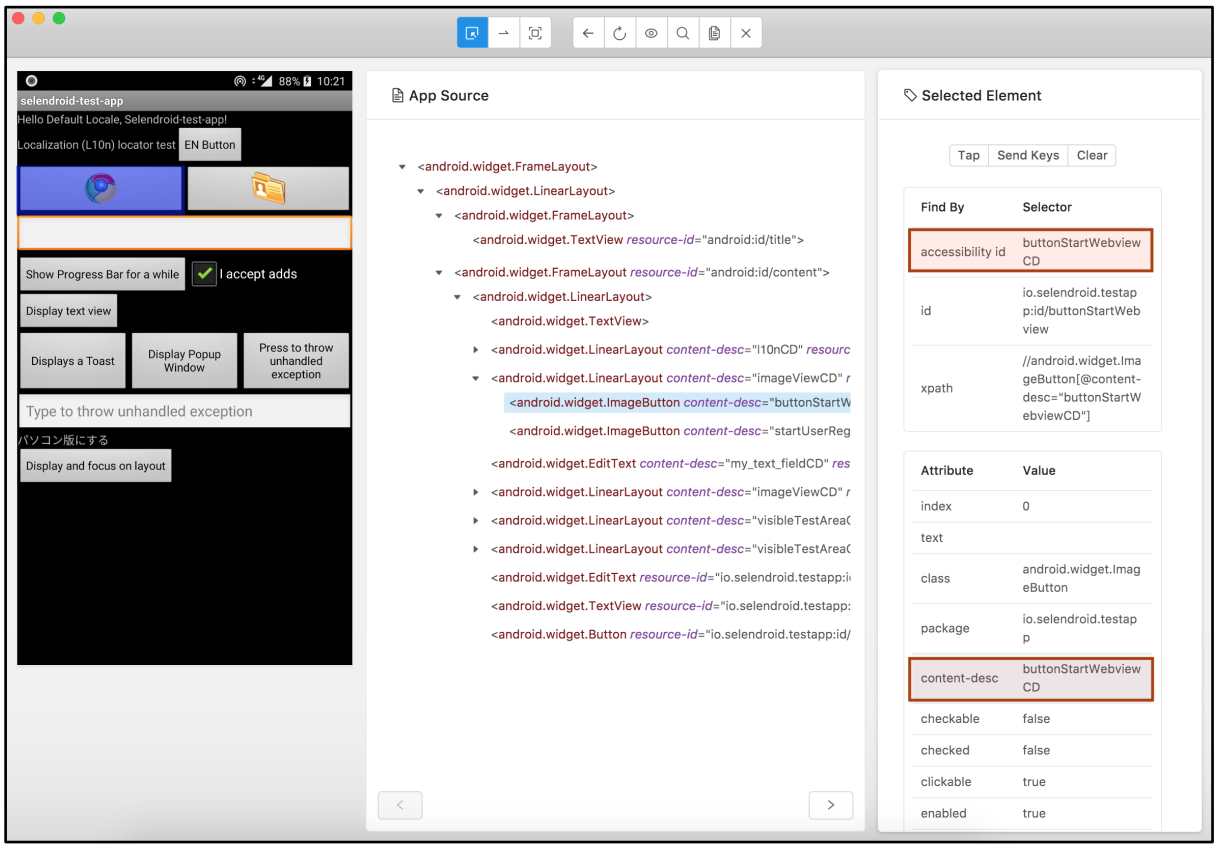

Figure-2:Locator Type: Accessibility Id on Android Sample Application.

Example Usage in different programming languages:

1) Java:

```
WebElement chromeButtonElement =
driver.findElementByAccessibilityId(“buttonStartWebviewCD”);
MobileElement mobileElement =
```

```
(MobileElement)chromeButtonElement;
```

2) **Python:**

```
element = self.driver.
find_element_by_accessibility_id(“buttonStartWebviewCD”)
```

3) **JavaScript:**

```
let element = await driver.
elementByAccessibilityId(“buttonStartWebviewCD”);
```

4) **Ruby:**

```
@driver.find_element(:accessibility_id,
"~buttonStartWebviewCD")
```

5) **PHP:**

```
$els = $this->element($this->using('accessibility id')-
>value(‘buttonStartWebviewCD’));
```

2) Class Name

- Finding an element using Class Name is generic and it does not guarantee to find the unique element because many elements have the same class name.

- **iOS:** In iOS the class name is the fully qualified name of *UIAutomation* class, and it starts with “***UIA***” keyword such as `UIAButton`, `UIARadioButton` and `UIATextField` for old versions of iPhone Apps, and on recent versions made on Swift programming language you can find the “***XCUITest”*** keyword.

- **Android:** In Android, the class name is the fully qualified name of the UIAutomator class and these are examples of it: `android.widget.TextView` , `android.widget.Button`, `android.widget.ImageButton`, `android.widget.CheckBox` etc.

- Now, in the above image (fig. 2), as you can see for the Chrome Button the class name is `android.widget.ImageButton` which is

same for the User Registry button. Which leaves the question, how do you get the right button? The answer is using **"Indexing"**

- In figure-2 above the index value of the Chrome Image Button is 0 while in below image you can see the index value of User Registry Image button is 1, so by combination of locator and Index you can get the needed Unique UI locator. *This is NOT ADVISABLE as it does not provide stability. There is a strong likelihood of indexes changing, for example if a new Image button is added to the screen!)*.

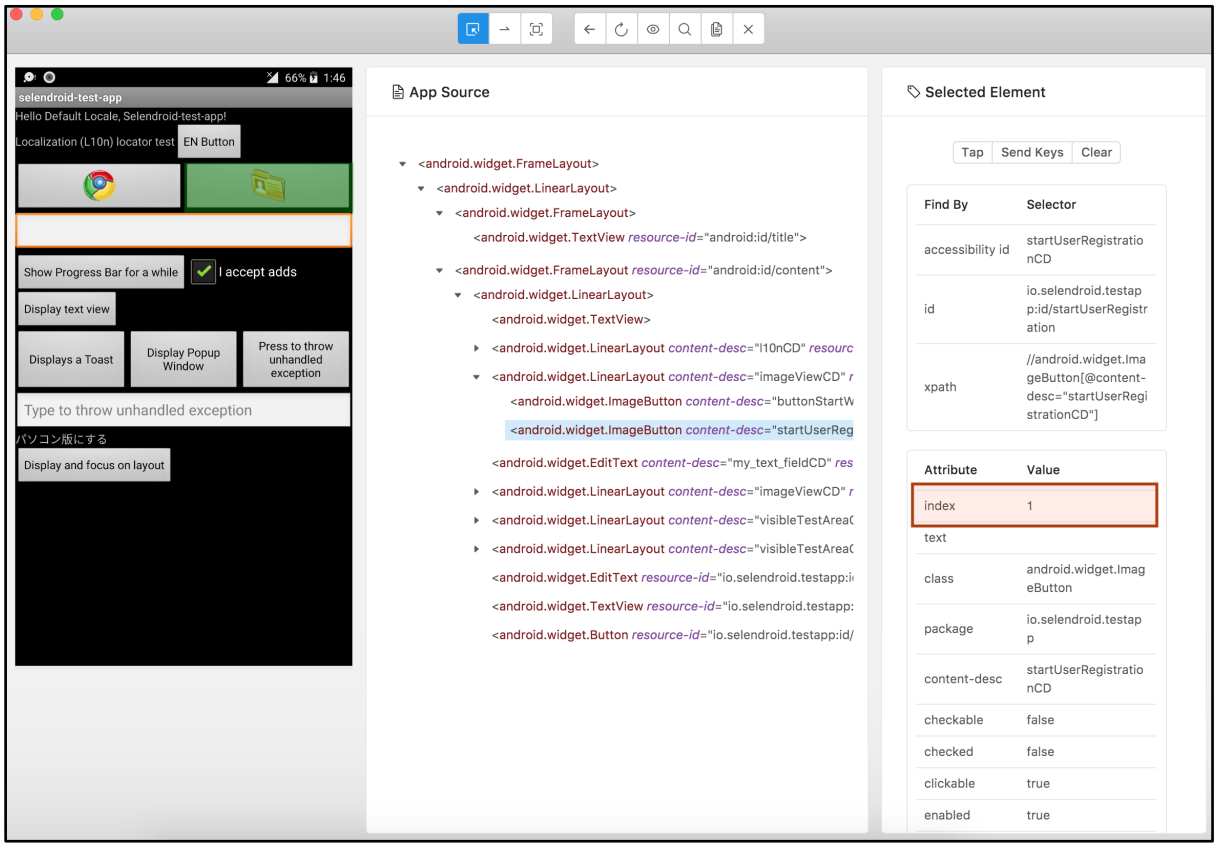

Figure-3: Index of class Name: android.widget.ImageButton

- You can get the indexed values using the relevant programming languages methods.

- This JAVA code will get the User Registry Image Button which has **Class name**= `android.widget.ImageButton` and **Index**=2.

```
List<MobileElement> mobileElements = (MobileElement)
driver.findElementsByClassName("android.widget.ImageButton");

MobileElement mobileElement = mobileElement.get(1);
```

NOTE: Actually You can get locators by two ways in Appium (for `id`, `name`, `className`, and `xpath`).

1) Using Selenium Methods:

```
WebElement element =
driver.findElement(By.className(“android.widget.ImageButton”)
);

// OR

WebElement element =
driver.findElementByClassName(“android.widget.ImageButton”);
```

2) Using Appium (Selenium Wrapper) Methods:

```
MobileElement mobileElement = (MobileElement)
driver.findElement(By.className(“android.widget.ImageButton”)
);

// OR

MobileElement mobileElement = (MobileElement)
driver.findElementByClassName(“android.widget.ImageButton”);
```

3) ID

In Mobile Application Automation `id` is are in form of Native context, it is not similar to Selenium WebDriver’s CSS id.

- `id` are also cross-platform locator strategy similar like `accessibility id`.

- **iOS:** It will find elements by `name` and `label` attribute but before that Appium will try to search for a `accessibility id` that will match with the given id string.

- For Figure-4 screenshot below both locator strategies are valid.

```
driver.findElementById("IntegerA");

// OR

driver.findElementById("TextField1");
```

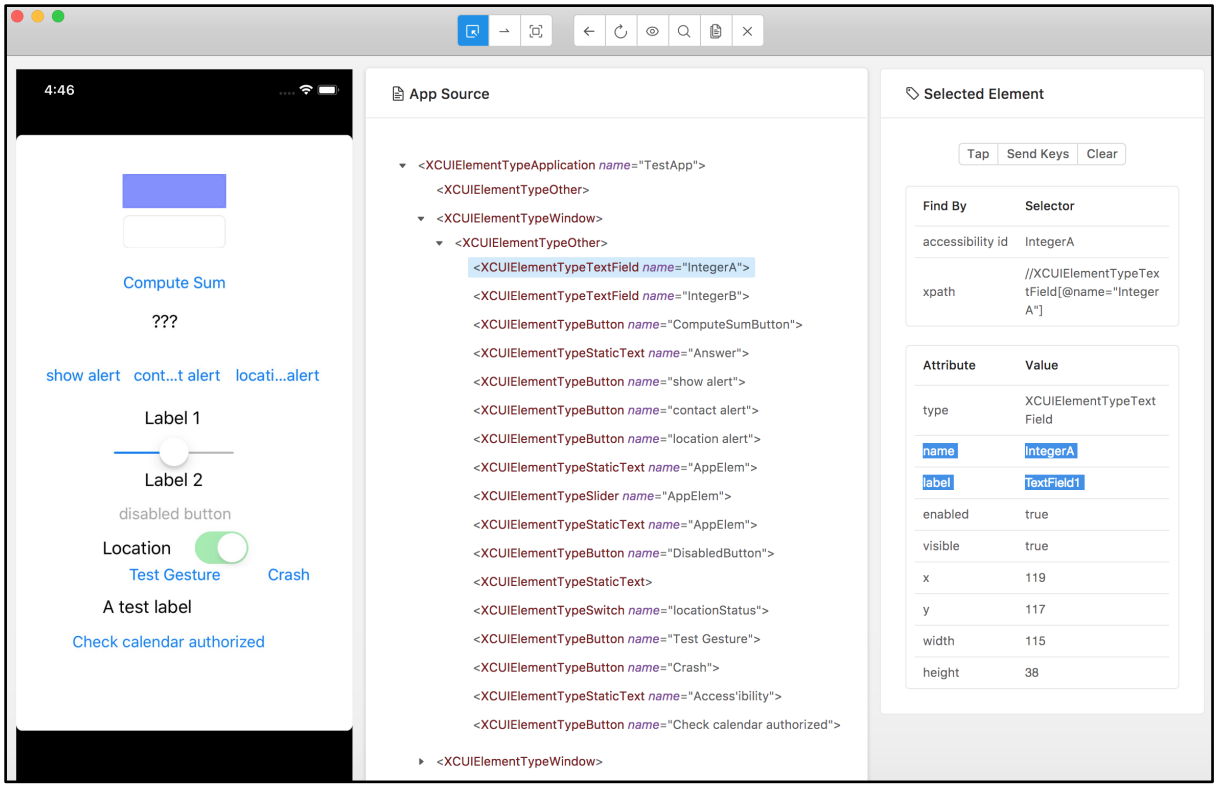

Figure-4:Locator Type:Id on iOS Sample Application.

- **Android:** In Android, it's `resource-id` attribute. It contains common ***<package-name>:id/<id-name>*** string format.

- You can use either that full string *(ex. io.selendroid.testapp:id/startUserRegistration)* or only ***<id-name>*** *(startUserRegistration)*. So in the below code both options are valid.

```
driver.
findElementById("io.selendroid.testapp:id/startUserRegistrati
on");

// OR

driver.findElementById("startUserRegistration");
```

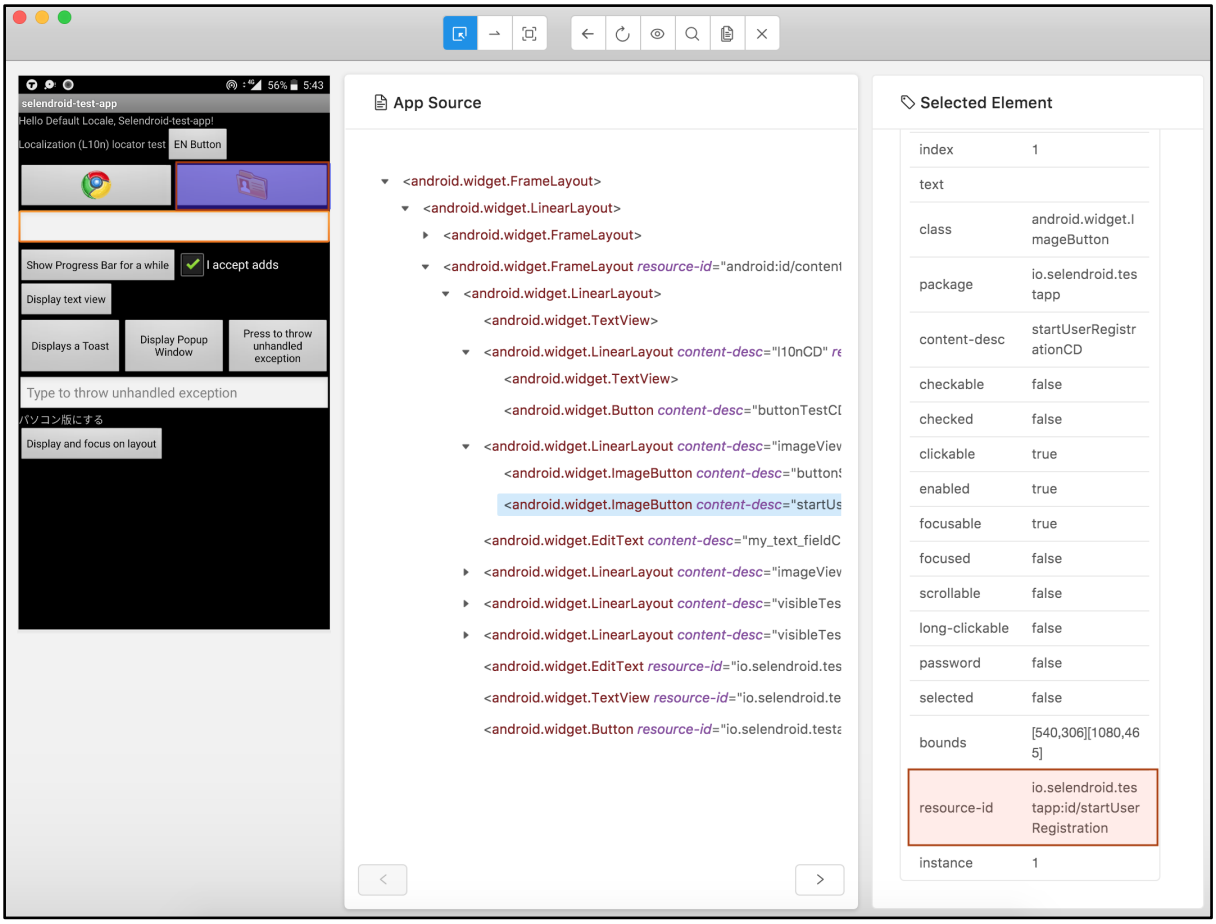

Figure-5:Locator Type:Id on Android Sample Application.

4) Name

- **iOS & Android:** It's the Name of the element on both platforms. This isn't used as often as `accessibility id` and `id` strategies are mostly used.

- In below image you can find the Name attribute using:

```
MobileElement element = driver.findElementByName("IntegerA");
```

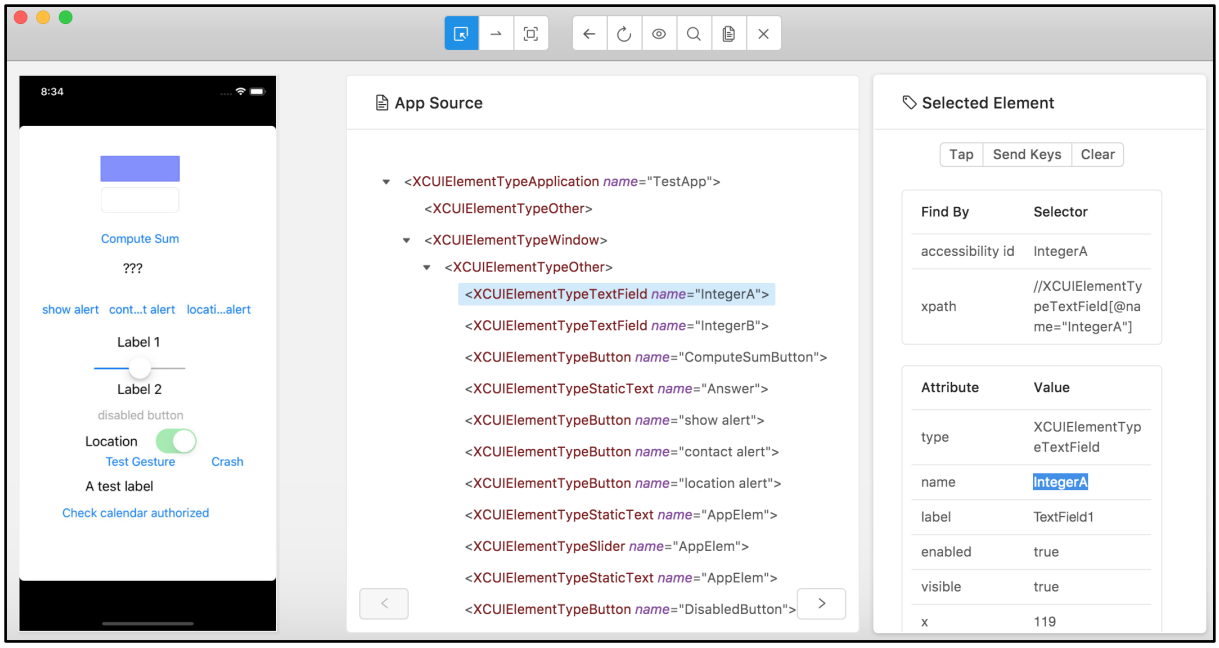

Figure-6: Locator Type:Name on iOS Sample Application.

5) XPath

- This locator strategy analyzes the XML structure of the app and locates the element with respect to the other elements.

- The XPath is originally designed to allow for the navigation of XML data with the purpose of locating unique UI elements.

- ***XPath selectors are not cross-platform.***

- ***This strategy should only used when there is no Accessibility Id, Id or Name assigned to an UI Element. XPath has performance and stability issues but is very "brittle" changing across platforms and even device manufacturers.***

- This strategy comes to the rescue when you've tried the above strategies and failed. As it depends on Parent XML nodes it's really very fragile because when any new UI element gets added or removed, the XML structure is changed rendering your locators broken.

- **Now the question is why you would ever use XPath ?**
    - *It allows for the formulation of complex queries.*
    - *It can literally find any UI element in the XML structure available to Appium. So even if no ID or Name is present, you can still find it with XPath.*

- If you are using the Appium Inspector for inspection of the Application XML structure then Appium will give you the XPath directly without any extra effort.

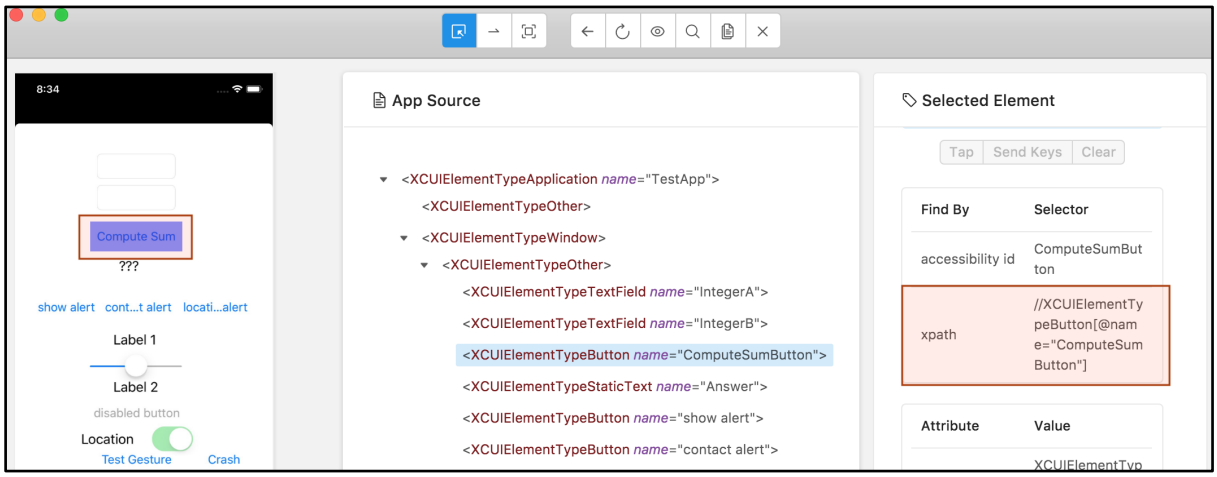

Figure-7: Locator Type:XPath on iOS Sample Application.

- Using XPath you can use any attribute or/and combination of attributes in order to find the element uniquely. Apart from the above Xpath in the screenshot, all of the following XPaths are valid and find the ***Compute Sum*** button uniquely.

```
MobileElement computeSumButton = driver.findElementByXPath
("//XCUIElementTypeButton[@name="ComputeSumButton"]");

// OR

MobileElement computeSumButton =
driver.findElementByXPath("(//XCUIElementTypeButton)[1]");

// OR

MobileElement computeSumButton = driver.findElementByXPath
("//XCUIElementTypeButton[@label='Compute Sum']");
```

- You can learn more about how you can find the Xpath from: https://www.w3schools.com/xml/xpath_syntax.asp, and from this link you can learn more about how you can properly use XPath with Appium:

  http://www.software-testing-tutorials-automation.com/2015/10/ui-automator-viewer-get-android-app.htm

6) Image

- Appium supports Image Comparison as a locator strategy which is using the OpenCV library in the backend.
- The strings which are being used by this locator strategy are Base64-encoded image files.
- So you need to convert image files into Base64-encoded image files first and you need to pass that String into the locator.
- Below is the example:

```
String base64Image = //Code which will to convert Image file
to Base-64 String
WebElement element = driver.findElementByImage(base64Image);
```

- We will look into this locator strategy in more details in a dedicated chapter later on.

7) Android UiAutomator (UiAutomator2 only)

- This is an Android Platform specific locator strategy.
- This is rarely used to find the element locators as it requires to have prior knowledge of the UiSelector API (and of course it's Android only).
- It's performance is slightly better than XPath.
- In this example we find the first ***Button*** element having the text ***Login***:

```
String selector = "new UiSelector().text(“Cancel”))
.className(“android.widget.Button”))";

MobileElement element = (MobileElement)
driver.findElement(MobileBy.AndroidUIAutomator(selector));
```

8) Android View Tag (Espresso only)

- This is also an Android Platform specific locator strategy.

- It locates an element using it's view tag.

9) IOS UIAutomation

- This is iOS Platform specific locator strategy. It uses Apple's Instruments framework.

- It performs better than XPath.

- Example:

```
String selector = "**/XCUIElementTypeCell[`name BEGINSWITH
"P"`]/XCUIElementTypeButton[4]";

MobileElement element = (MobileElement)
driver.findElement(MobileBy.iOSClassChain(selector));
```

- In the above example we will find the 4th button anywhere under the UI hierarchy and whose name begins with the character 'P'.

Hopefully you're starting to get an idea of when to use which locator. Don't worry if it isn't immediately clear … the more you start building Appium scripts and the more you keep reviewing these it will become more intuitive.

Generally speaking you will find you should mostly likely use **Accessibly Id** and **Id** automation strategies**. XPath** is incredibly flexible as a fallback when no ID exists, but tends to be brittle. Your best option is to work with the developers to add unique IDs if they don't exist. This will make for far more robust test scripts.

Chapter-5: The Appium Inspector

As you learned in the previous chapter, In order to perform automation it's necessary to locate the unique selectors for:

1) Mobile Applications and
2) Mobile Web Browser(Chrome).

However, if you haven't already realized (you soon will!), finding these unique elements can sometimes be painful. And this is where the Appium Inspector comes into play.

Appium inspection is known by many names such as Element Extraction, UI Element Identification, Locator Finding etc. It is the process by which which you can locate or find elements in your mobile application (native only).

Appium inspection is a standard procedure to identify the UI elements of a mobile app uniquely. It works with both real devices or simulators(iOS) or emulators(Android).

NOTE: The *Appium Inspection tool does not support finding the locators on Web Browser(Chrome) as it is specifically designed to fetch the attributes for Native Mobile Application only.*

The Appium Desktop Application is a combination of the Appium server itself and the Element inspector, which is designed to help you discover all the visible elements of your mobile application while developing your test scripts.

You can use it for native apps...

1) **To identify and understand the element hierarchy:** For developers this may be trivial but for testers it is definitely useful information on how certain UI elements are aligned with each other and what other layers/fragments/etc the app may have.

2) **To find the name, description, value and other attributes of the element/object:** Objects have certain characteristics that can be identified through this tool and then used with the xpath command.

3) **To record your manual actions with the app:** In order to record your actions, you need either the Appium Inspector or some other tool that can access those elements.

Different element inspector tools that helps you to identify elements in mobile app

There are many different tools that help you inspect elements in mobile apps. But we will cover the most important and used Element Inspectors:

1) **Appium Inspector:**

   You can use this inspector for both Android and iOS apps (for iOS apps, you would need a Mac)

2) **UiAutomatorViewer(Android):**

   This is a tool provided by Android Studio that lets you inspect elements in your mobile app.

   There is one important factor in that the way you inspect elements in mobile app is exactly the same in UIAutomatorViewer and Appium Desktop Inspector. The are slight differences in the UI of both the tools, but the underlying logic of identifying elements remains the same.

   Appium Desktop Inspector uses the same methods as UI Automator Viewer to identify the elements in your mobile app:

   - Find element by **ID**
   - Find element by **ClassName**
   - Find element by **Accessibility ID**
   - Find element by **XPath**

   Also, the properties of the mobile elements, such as **resource-id, content-desc, text** etc, will be the same in both the tools. We will explore **UIAutomatorViewer(For Android)** in the next chapter.

3) **Accessibility Inspector(iOS):**

   The Accessibility Inspector is a tool that shows all of the properties and values, methods (actions that can occur from elements on the screen), and position of the object that's currently being selected on the screen.

In this chapter we will discuss the most used and popular tools to find the unique and correct element locator.

Element extraction on mobile native applications using Appium Inspector

Now let's discuss how you can extract the elements using Appium Inspector:

1) Open the Appium Desktop Application:

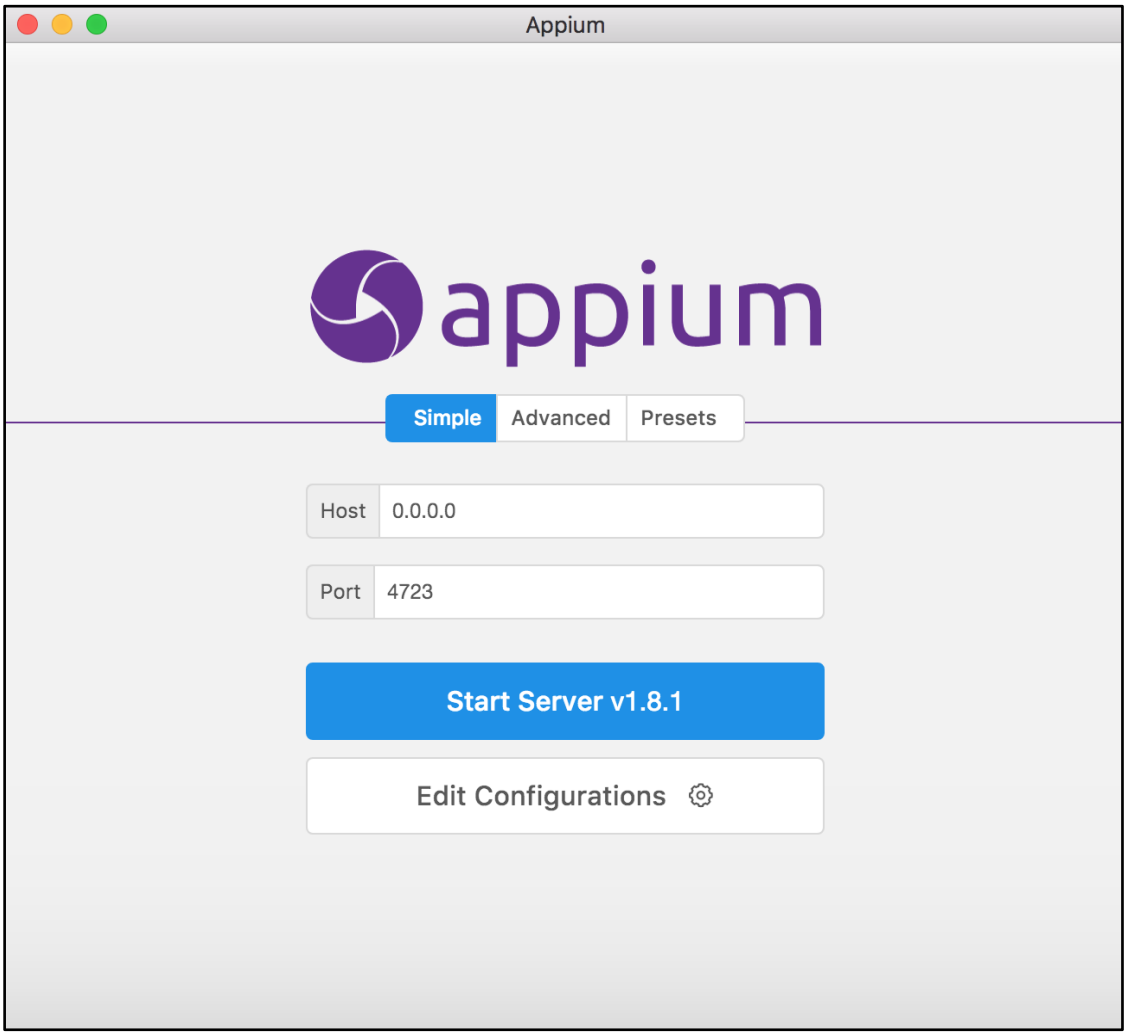

Figure-1: Appium Desktop Application.

2) Start the Server by clicking on the ***Start Server*** button

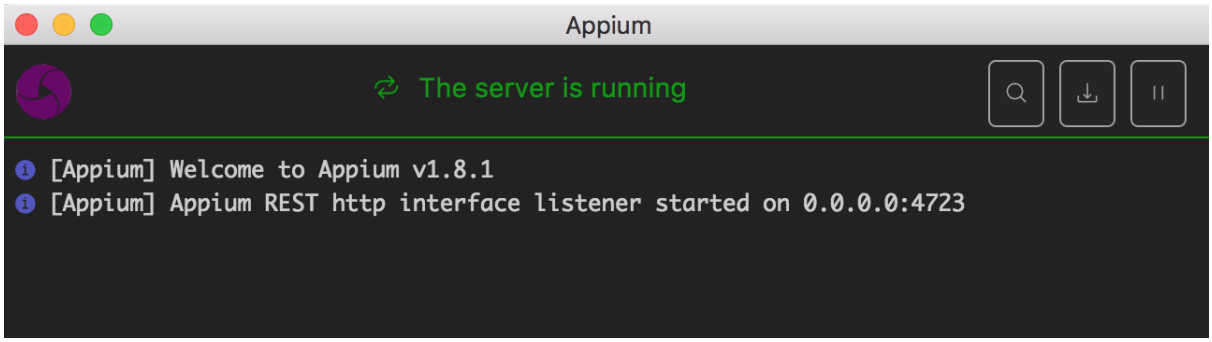

Figure-2: Appium Server Is Running.

3) Click on the Search button and open the Appium Inspector Session.

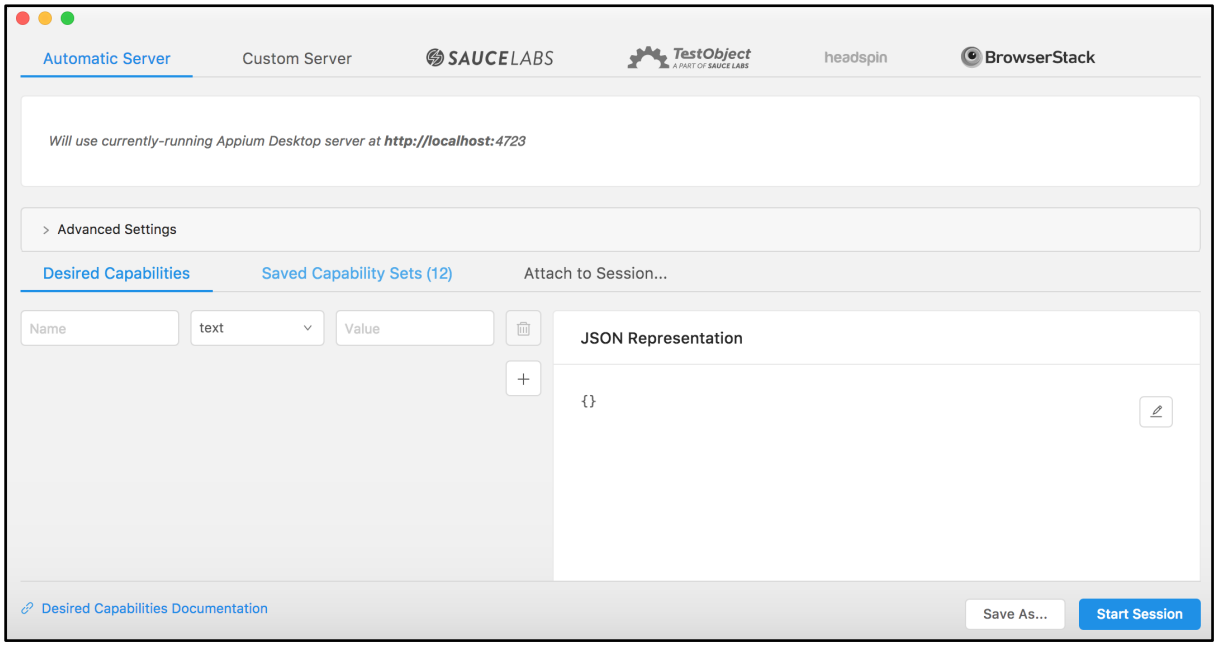

Figure-3: Appium Inspector Session.

4) As we discussed in Chapter 3 you need to provide the correct set of Desired Capabilities.

Android:

```
{
  "platformName": "Android",
  "platformVersion": "8.1", //<<Android Version of connected
Device>>
  "app": "/path/to/.apk/file",
  "deviceName": "c33143r", //<<get device name using: $ adb
devices>>
  "automationName": "UiAutomator2"
}
```

iOS(Real Device):

```
{
  "platformName": "iOS",
  "platformVersion": "11.4",
  "app": "/path/to/.ipa/file",
  "deviceName": "John's iPhone", //<<get it using iTunes>>
  "udid": "bea36e2b0262ae4b77bd3463bd462922ee935d24", //<<get
it using iTunes>>
  "automationName": "XCUITest"
}
```

iOS(Simulator):

```
{
  "platformName": "iOS",
  "platformVersion": "11.4",
  "app": "/Users/username/Downloads/sample.ipa",
  "deviceName": "iPhone X", //<<(iPhone 7, iPhone 7 Plus
etc..)You can get devices from: "$ instruments -s devices">>
  "automationName": "XCUITest"
}
```

Below are the screenshots of Desired Capabilities for Android(Real Device) and iOS(Real Device and Simulator).

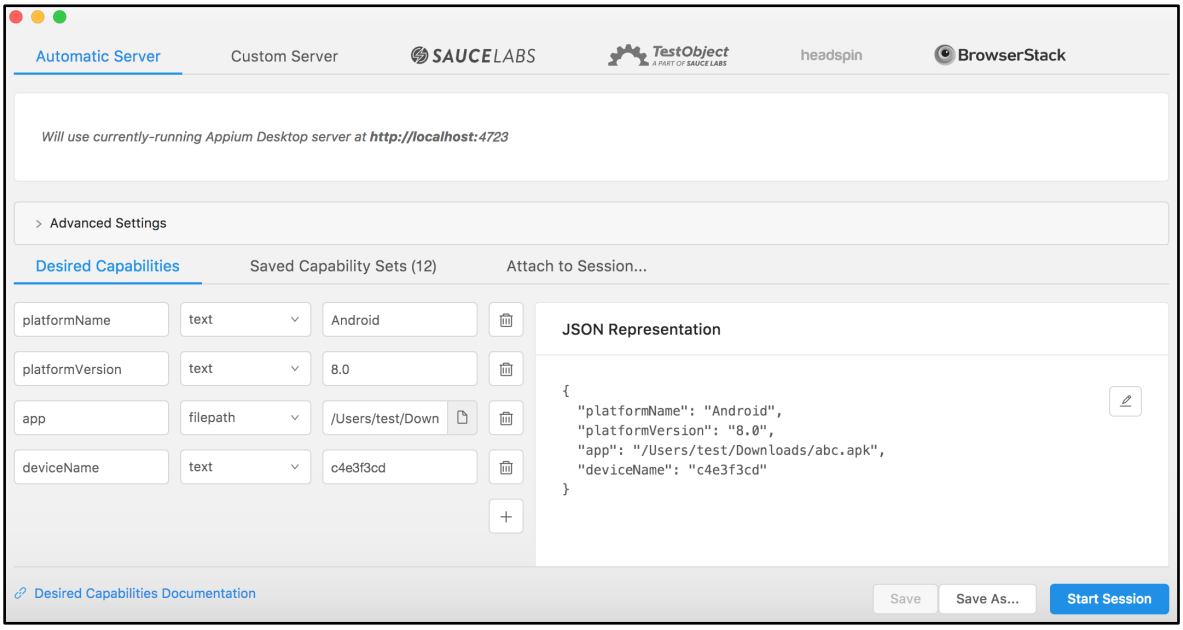

Figure-4: Android Desired Capabilities for Real Android Device.

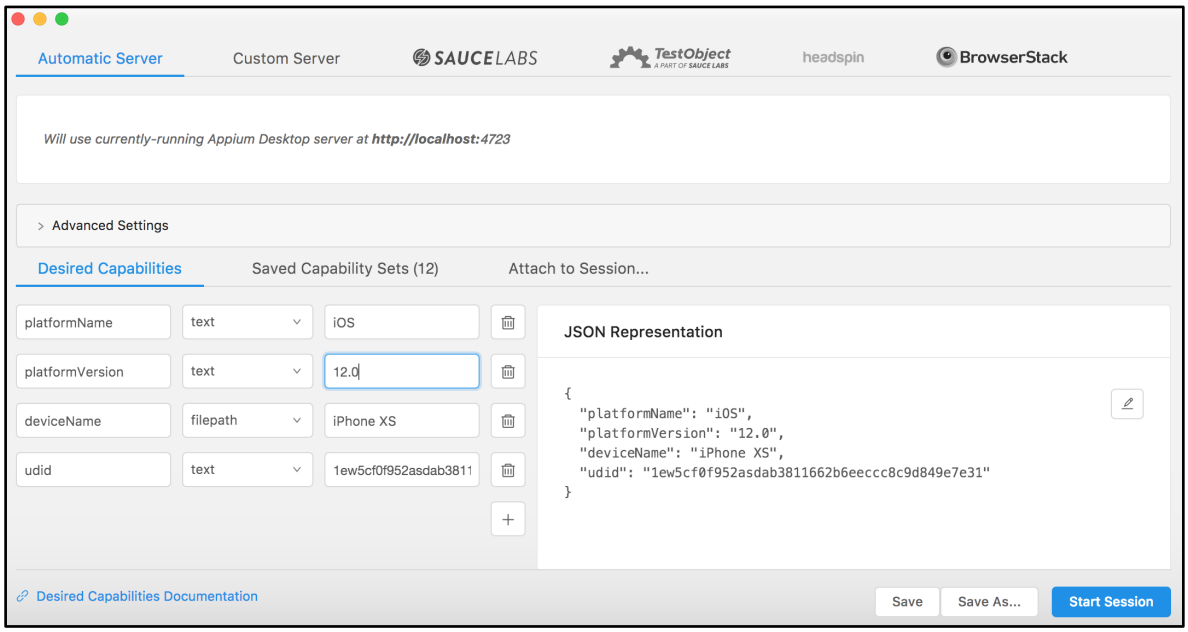

Figure-5: iOS Desired Capabilities for Real iOS Device.

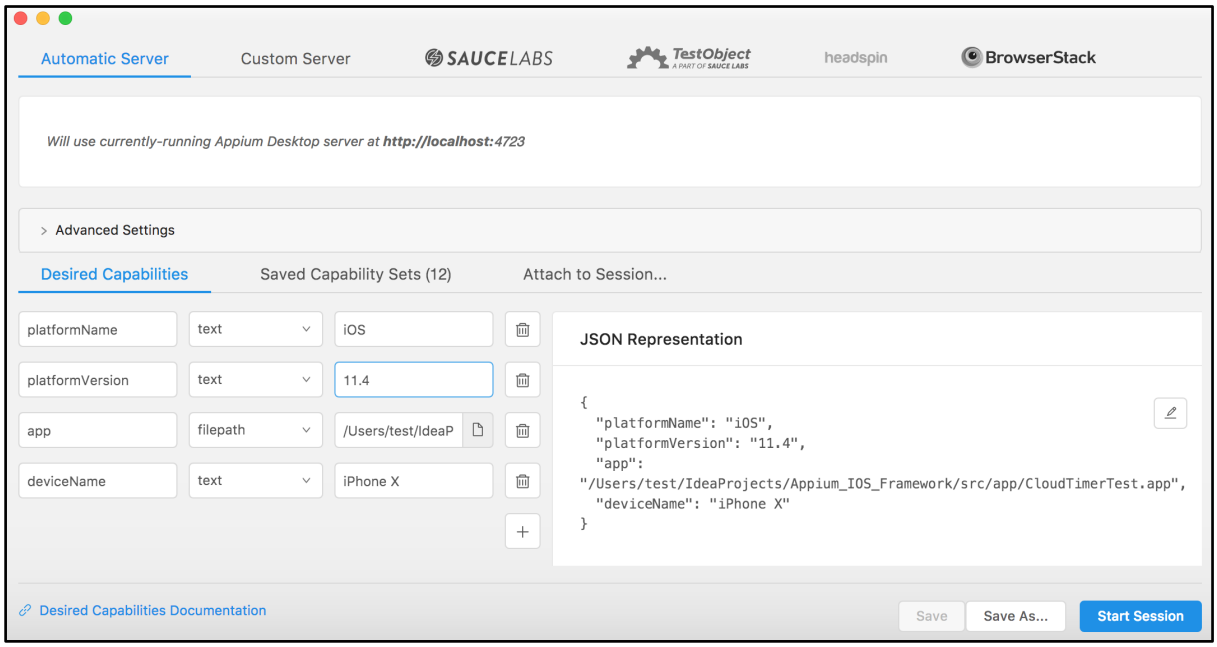

Figure-6: iOS Desired Capabilities for iOS Simulator.

5) You can also save the Desired Capabilities for a particular configuration by clicking on the **Save/Save As..** button.

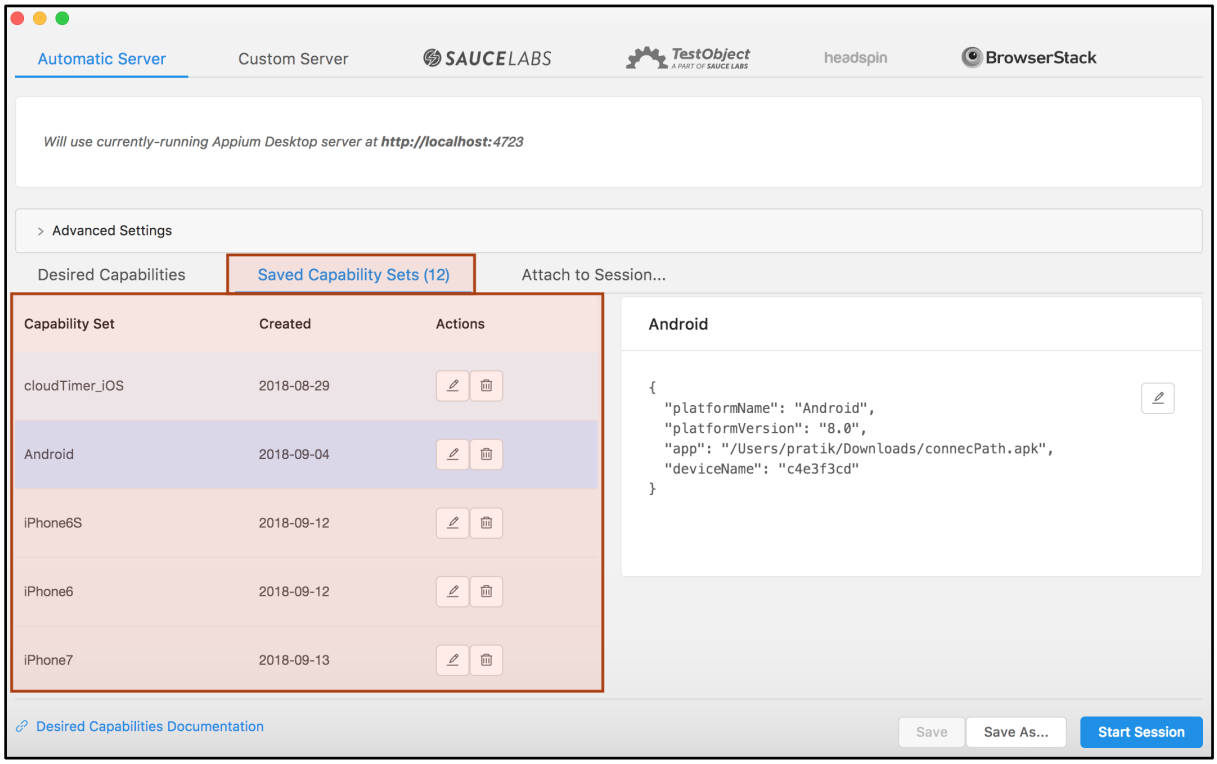

Figure-7:Saved Capabilities Set.

6) Click on the **Start Session** button - it will take some time because the Appium server will install the mentioned app to your connected device/simulator and then it will analyze the Application XML and underlying structure. After some time you can see the a window similar to:

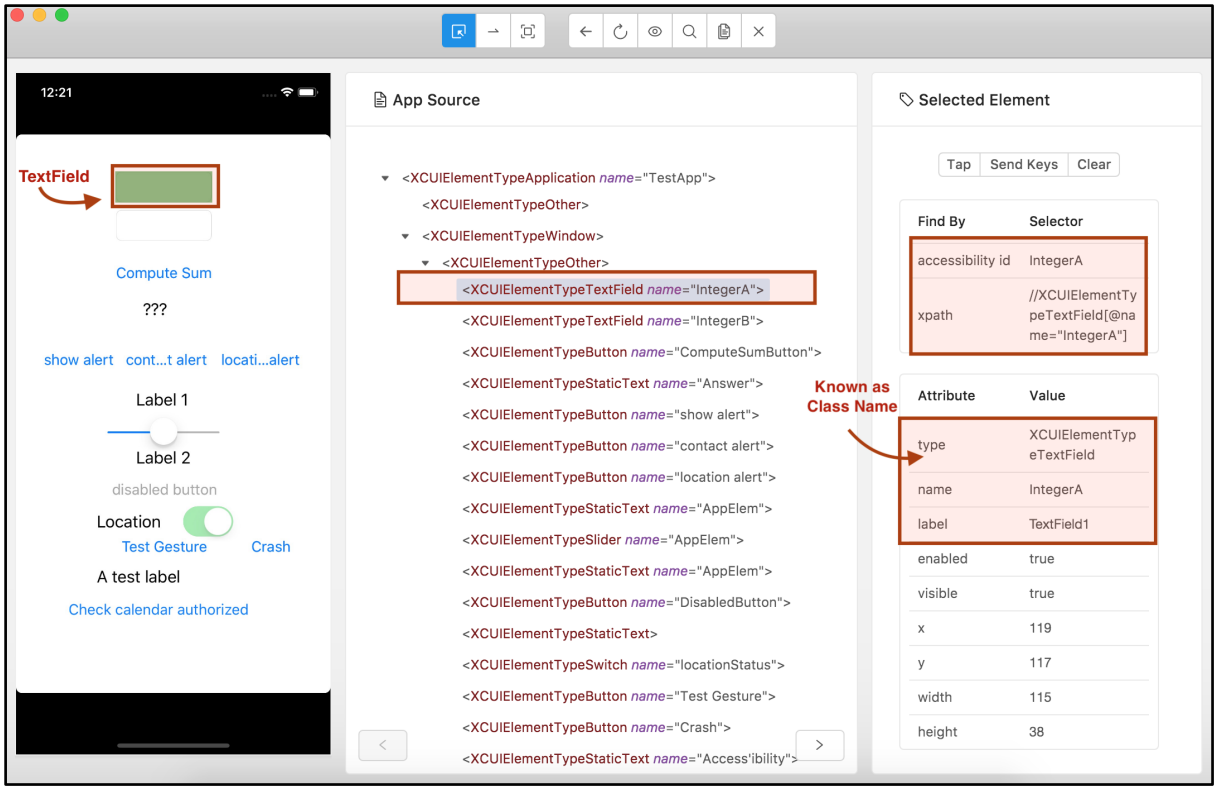

Figure-8: Appium Inspector Session(iOS).

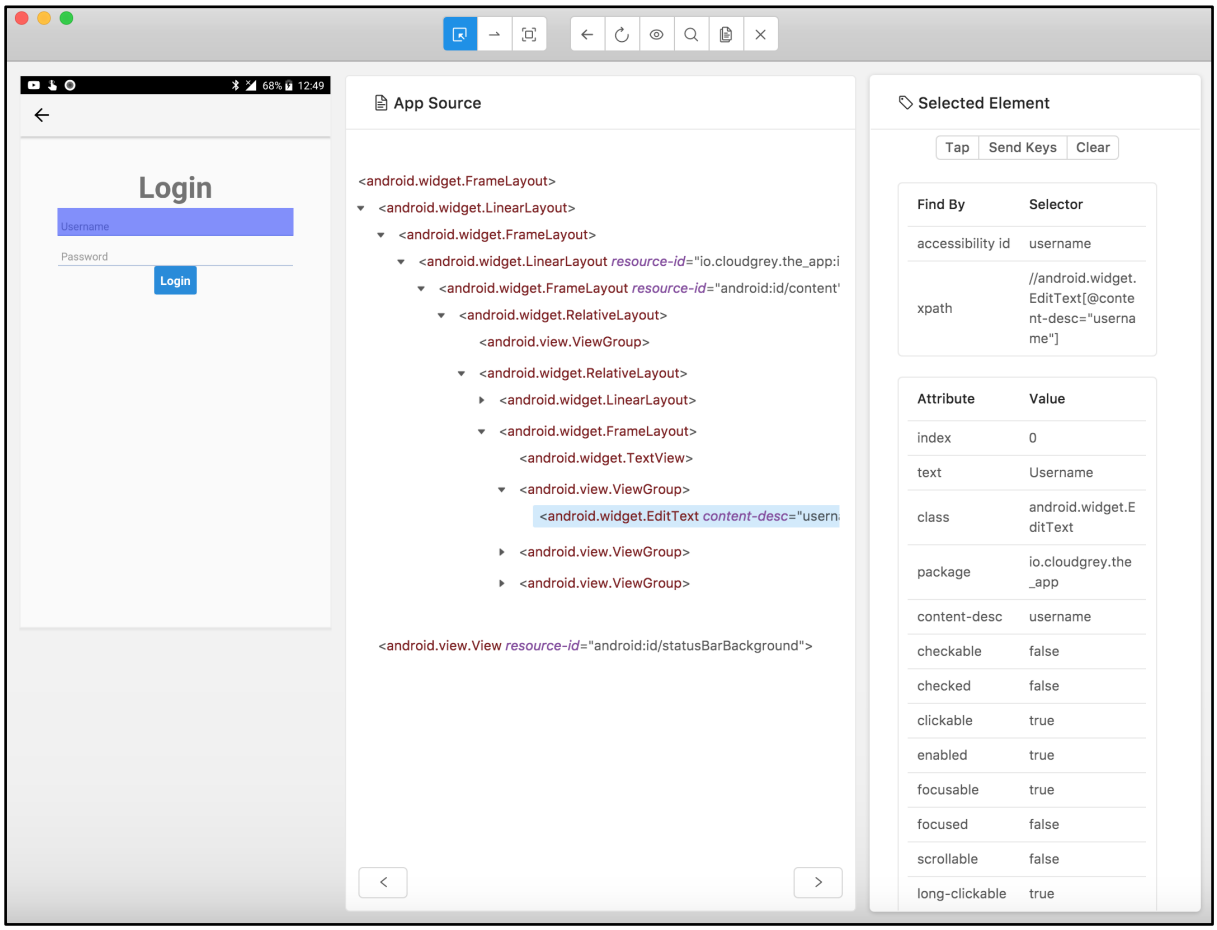

Figure-8.1: Appium Inspector Session(Android).

As you can see you can get the XML structure of all the visible elements on the screen. Using the best or most appropriate locators strategy(accessibility id, id, class name, xpath etc.) you can get the valid unique locators. Additionally, Appium inspector supports many features:

- A great feature of Appium is it will give you the best locator strategies listed automatically, so in the above image you can see that Appium is suggesting *accessibility id* and *xpath selectors* for the selected TextField on the screen. Isn't that convenient?

- But we have only looked at the first initialized screen.

  What if you are navigating to another page and want to find elements on that screen? Would Appium fetch the new screenshot and extract the XML structure automatically?

  Unfortunately the answer is NO, you need to manually click on Refresh Source & Screenshot button after changing the screen in order to get that screen's elements selectors.

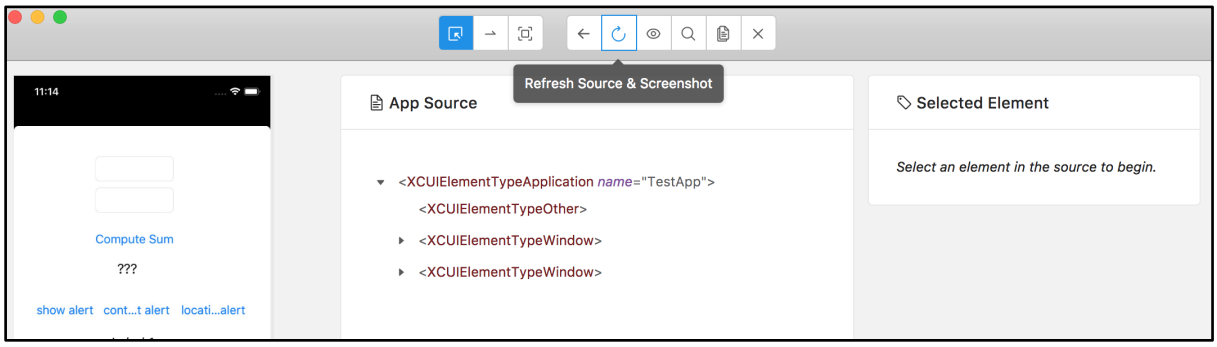

Figure-9:Refresh Source & Screenshot.

- Appium Inspector also provides some actions on elements such as Swiping, Tap on Coordinates etc.

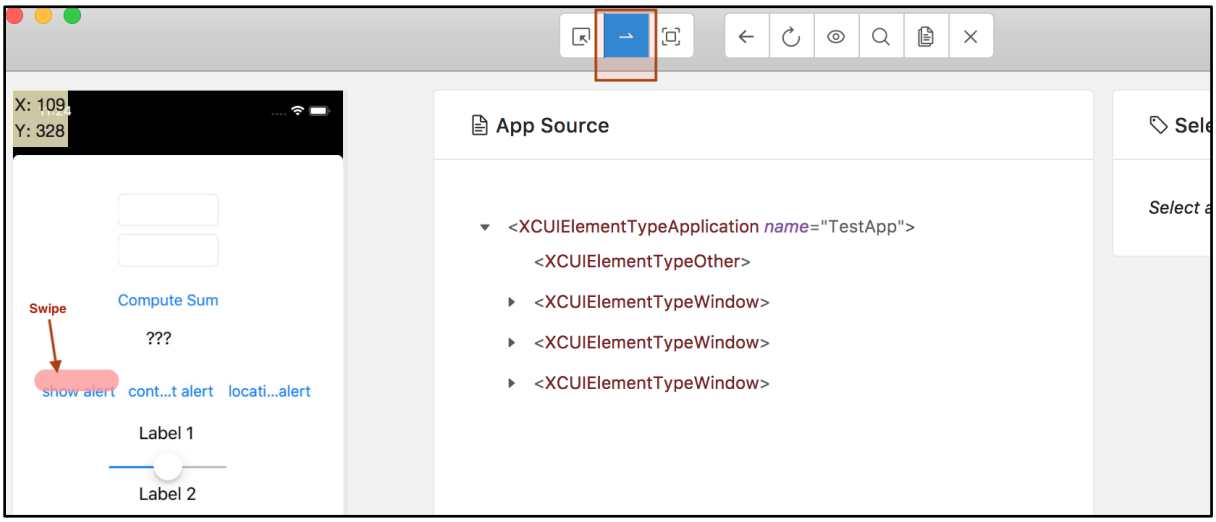

Figure-10: Swipe By coordinates.

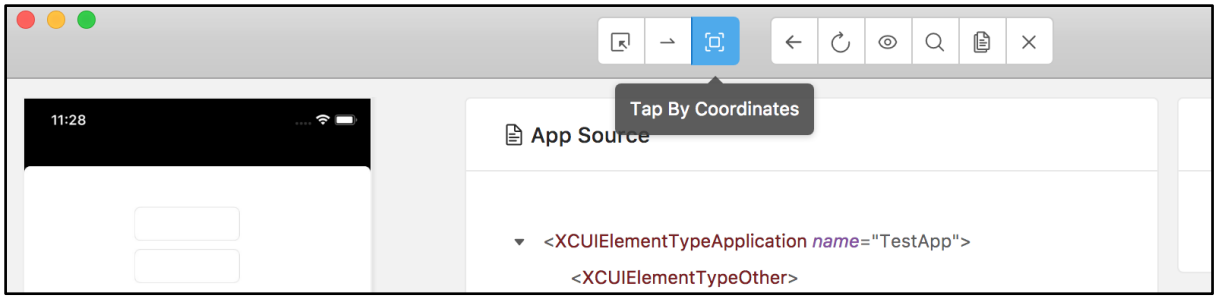

Figure-11: Tap By Coordinates.

- The Appium Inspector session also provides script recording functionality which can save a lot of time.

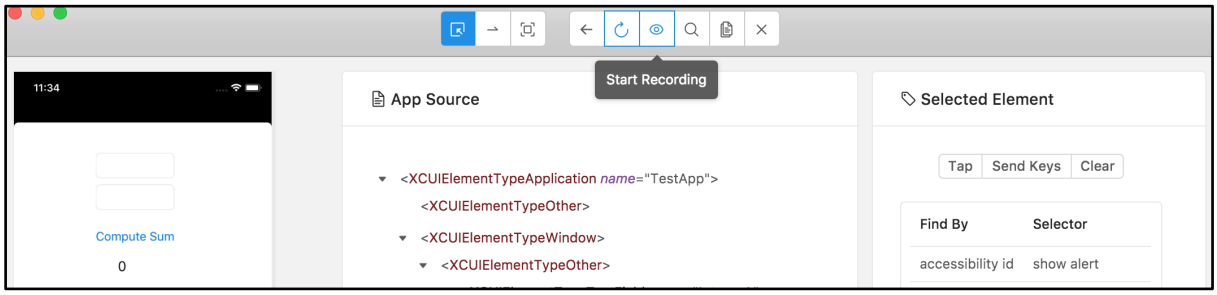

Figure-12: Start Recording.

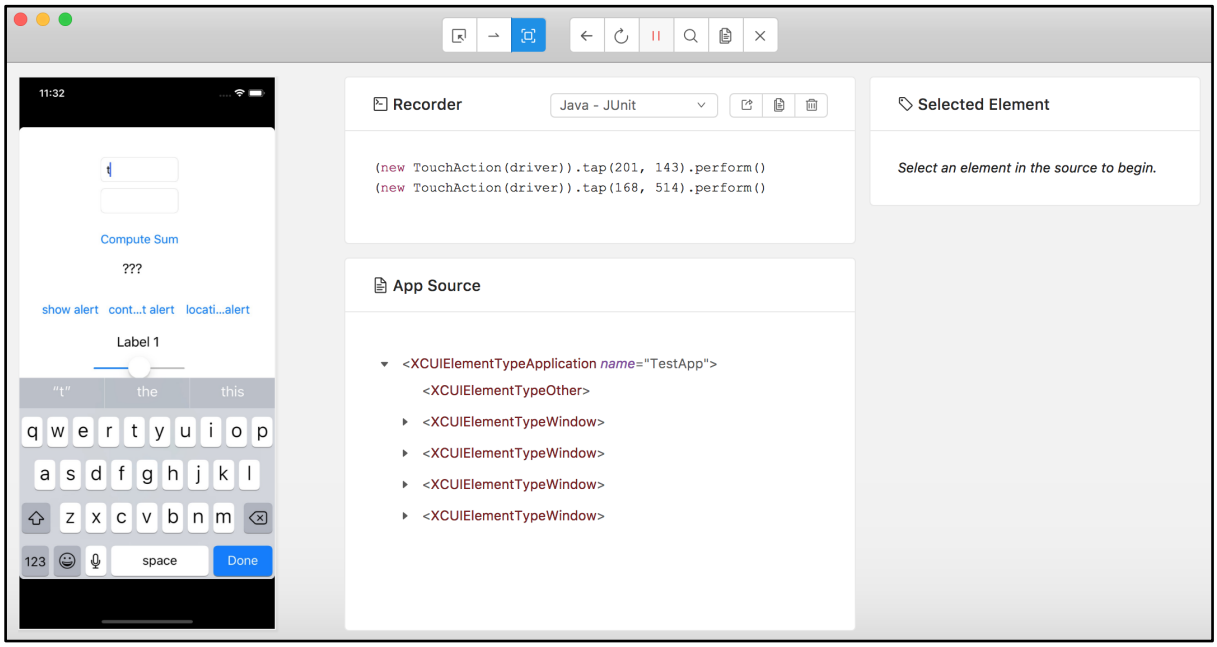

Figure-13: Recording started.

- Appium Inspector also supports the reverse case - meaning thay by using the locator you can search for the element on the UI.

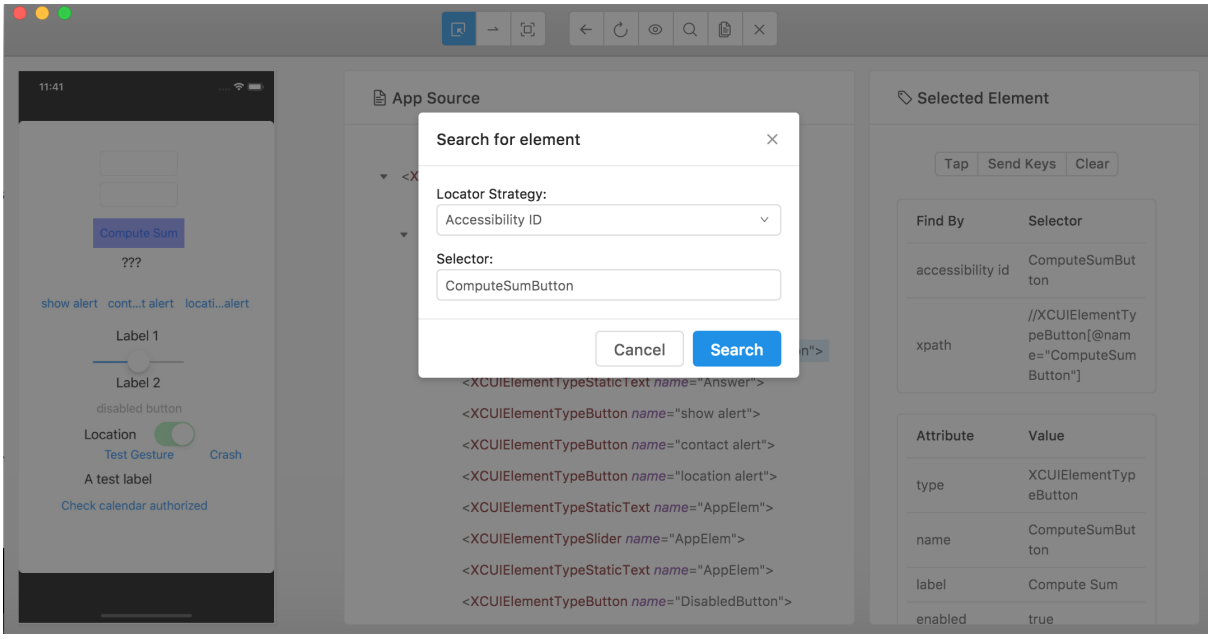

Figure-14: Search for element.

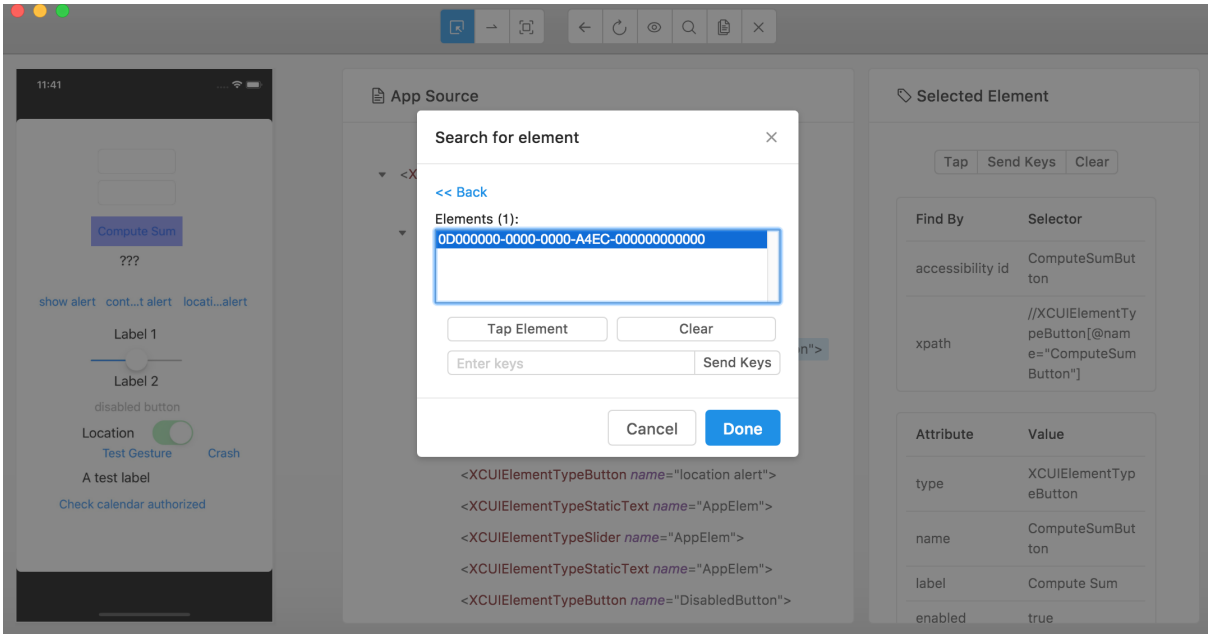

Figure-15: Searched Locator.

7) There is also a feature where you can **Attach to existing Session**. You need to provide just the session-id (as shown in the following screenshot). It is useful for when you already have an Appium session running. This (attaching session) is possible because the inspector is an Appium client, not Appium server.

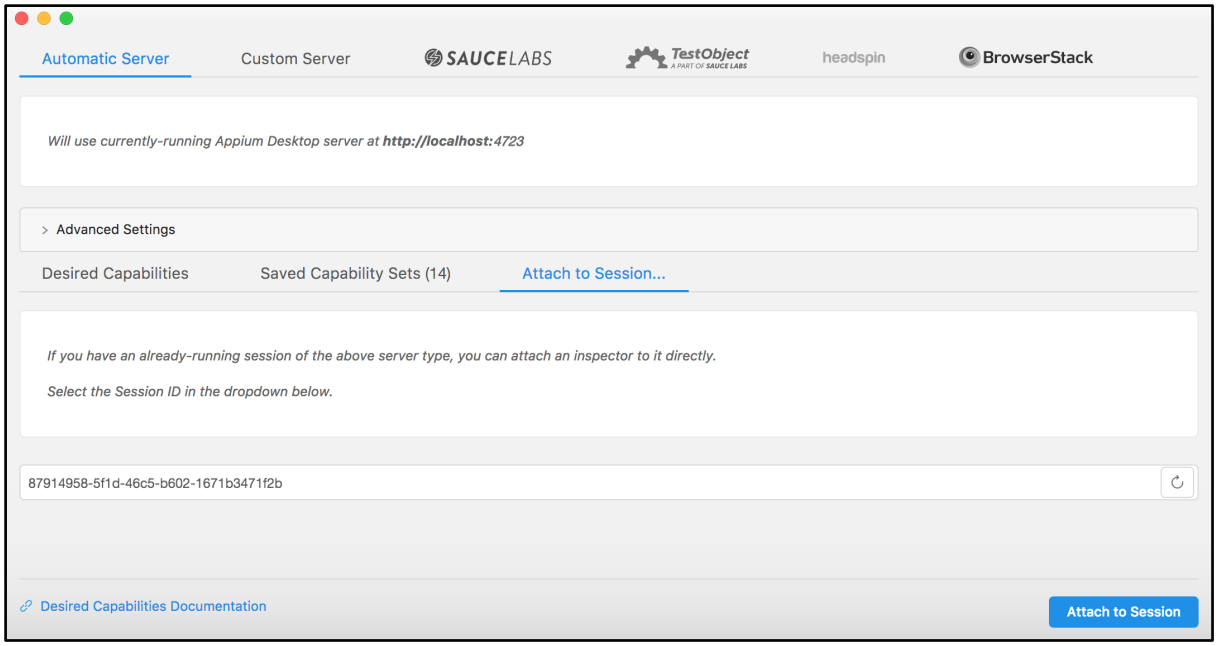

Figure-16: Attach to Session...

8) You can inspect elements on another custom appium server by providing the following information under the "Custom Server" tab:
    1) Remote Host Ip address.
    2) Remote Host Port.
    3) Remote Host Path.

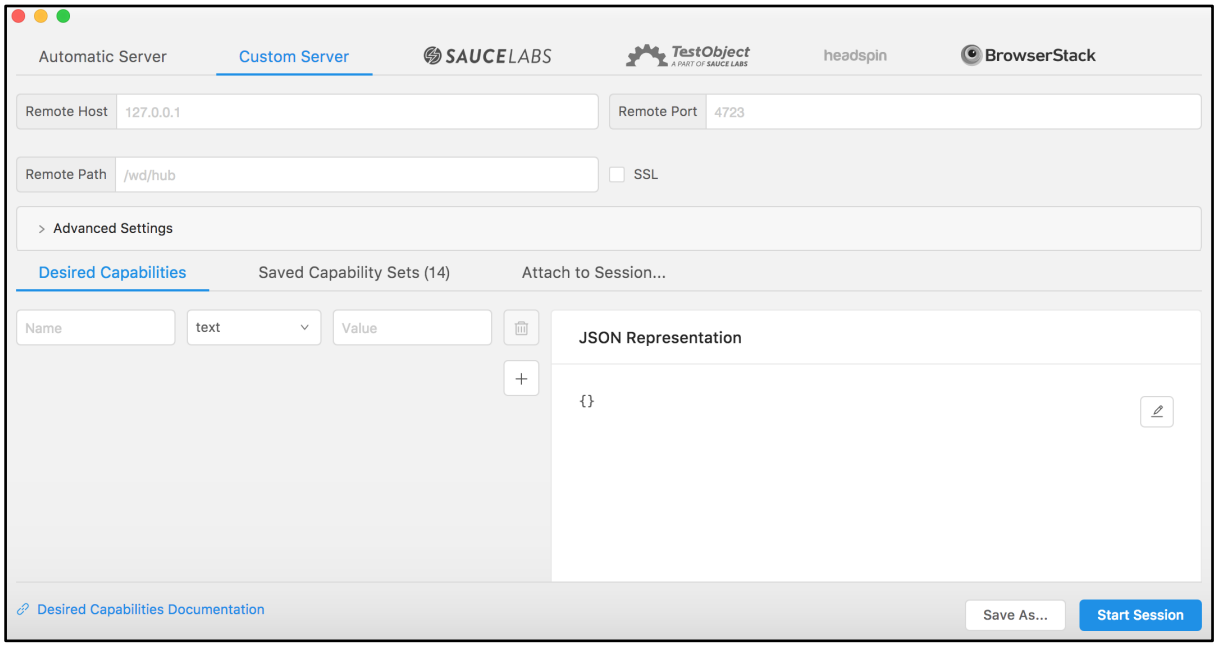

Figure-17: Custom Server Configuration.

9) Appium supports element inspection on remote devices.

    There are number of vendors that provide cloud based real-devices for testing mobile apps, including:

1) Kobiton,
2) Saucelabs,
3) TestObject,
4) Headspin,
5) BrowserStack,
6) Bitbar cloud,
7) Kobiton.

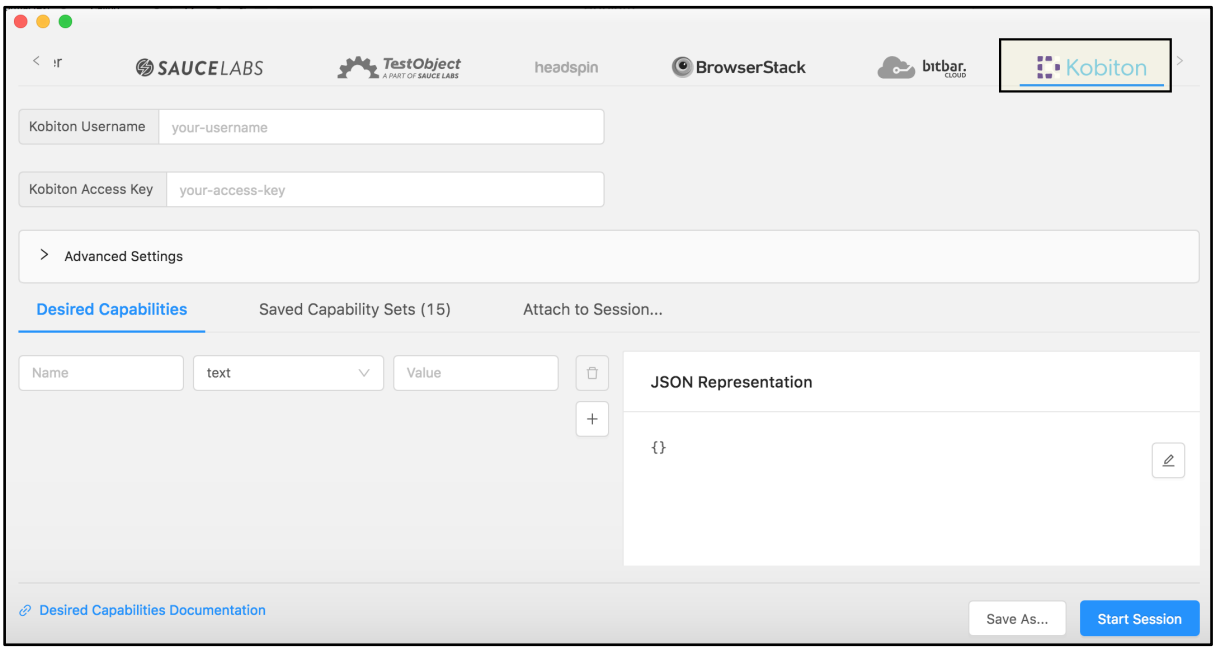

Figure-21: Kobiton Remote Inspection Configuration.

Comparison between iOS & Android locator strategy

Below is the mapping between Attributes from Appium Inspector(or UiAutomatorViewer) and Appium Locator Strategy for Android/iOS.

Android

Attribute	Locator Strategy(Android)
text	Name
resource-id	Id
class	Class Name
content-desc	Accessibility Id

iOS

Attribute	Locator Strategy(Android)
name	Name, Id
label	Id
type	Class Name
accessibility id	Accessibility Id

In this chapter we learned how to find elements using the Appium Inspector tool. In the next chapter we are going to see an alternative option UiAutomationViewer (only for Android) to inspect the UI elements. Before that, let's turn to how we can use Appium to extract elements on the mobile web browser.

Element extraction on a mobile web browser

In this section we are going to talk about how we can get the UI locators for the websites which we will be automating in a Mobile Chrome browser.

Make sure you use the mobile version of the site you're looking to test. For example http://m.facebook.com is the mobile website, while http://www.facebook.com is the default website on Desktop. However you can open http://m.facebook.com on your Desktop and can get the mobile view on your desktop. So ultimately the first thing is we need to get the Mobile website which we are interested in to be automated.

After getting the Website URL we need to find the locators of the elements we will be interacting with. This is a bit different to getting the elements from a Mobile Native application. On a Mobile Native application we can get the elements using the Appium Inspector while for Website automation we can get the UI elements from the browser itself, we don't need to rely on any third party tool.

If you're familiar with Selenium then you already know how to get website UI elements.

Below are the locator strategies to get the UI Element locators:

- ID
- Name
- Class Name
- CSS Locators

- XPath
- LinkText
- Partial Link Text

1) ID

- **Id = "m_login_email"** for the "Mobile number or email address" textfield

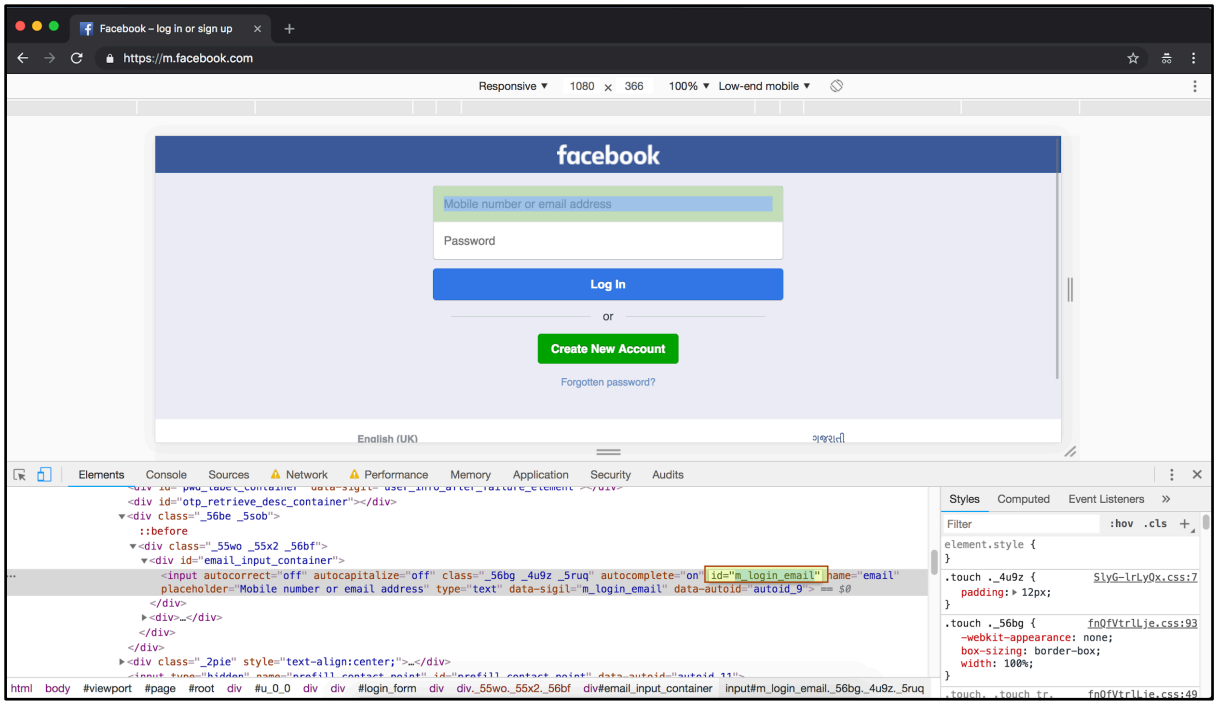

Figure-22: Id: Locator Strategy.

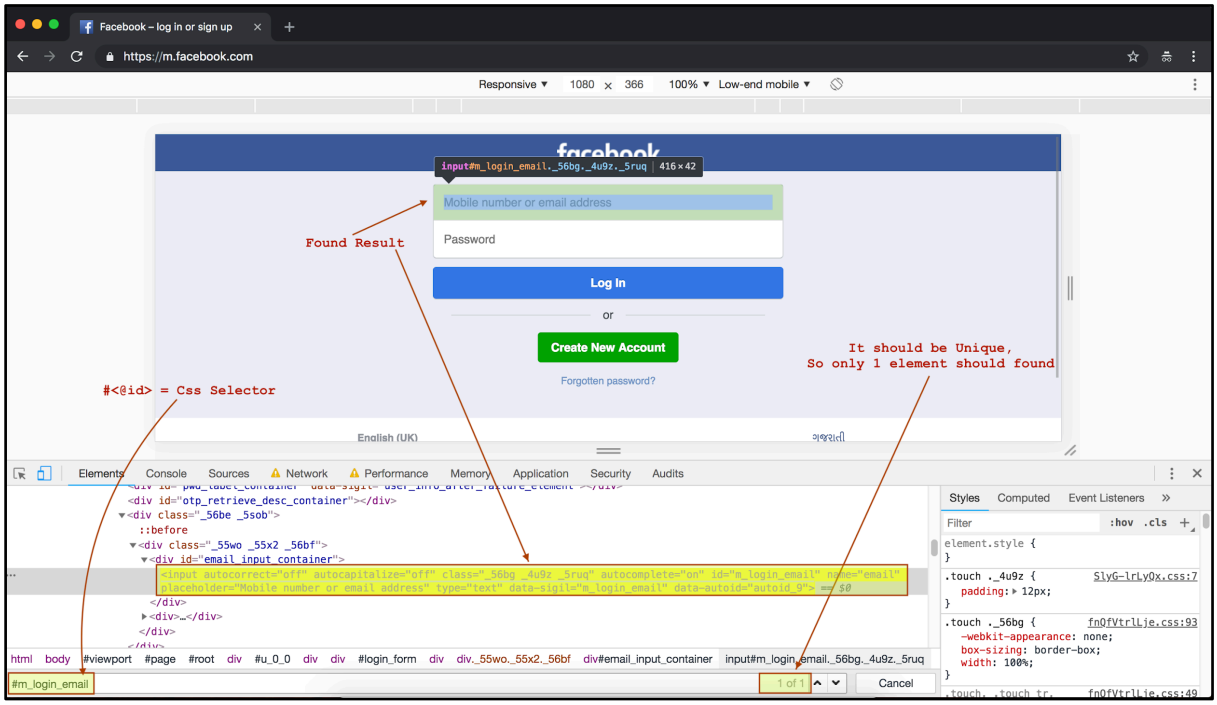

Figure-23: Id: Find the Id Locator Strategy on DOM.

- **Selector Code:**

```
driver.findElement(By.id("m_login_email"));
```

NOTE: *If you assign '#' before id name, it becomes the CSS Selector.*

2) Name

- **Name = "email"** for the "Mobile number or email address" textfield

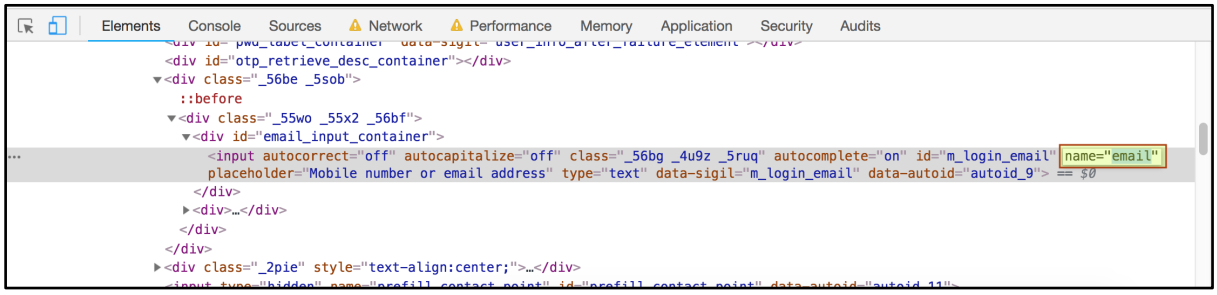

Figure-24: Name: Locator Strategy.

- **Selector Code:**

```
driver.findElement(By.name("email"));
```

3) Class Name

- **Class Name = "_56bg _4u9z _5ruq"** for the "Mobile number or email address" textfield. Please note that use this locator when the class name is defined only once in DOM. If more than one class name found in DOM, please don't use it.

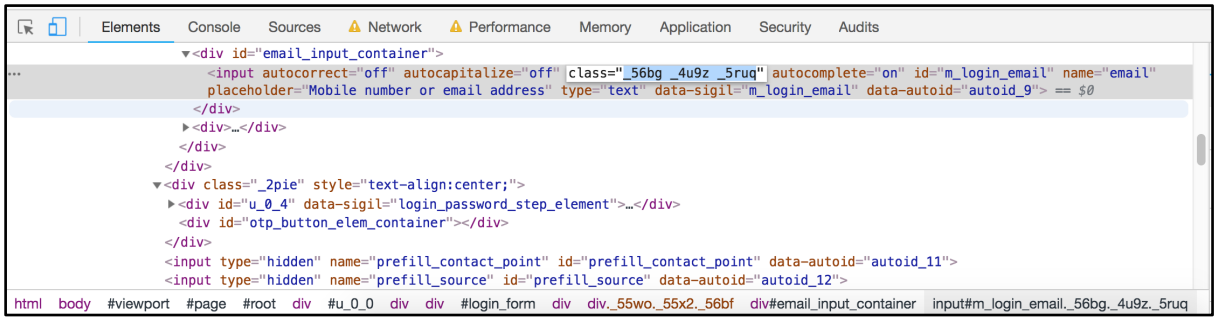

Figure-25: Class Name: Locator Strategy.

- **Selector Code:**

```
driver.findElement(By.className("_56bg _4u9z _5ruq"));
```

4) CSS Selector

- **CSS Selector = "._56bg._4u9z._5ruq"** for the "Mobile number or email address" textfield.

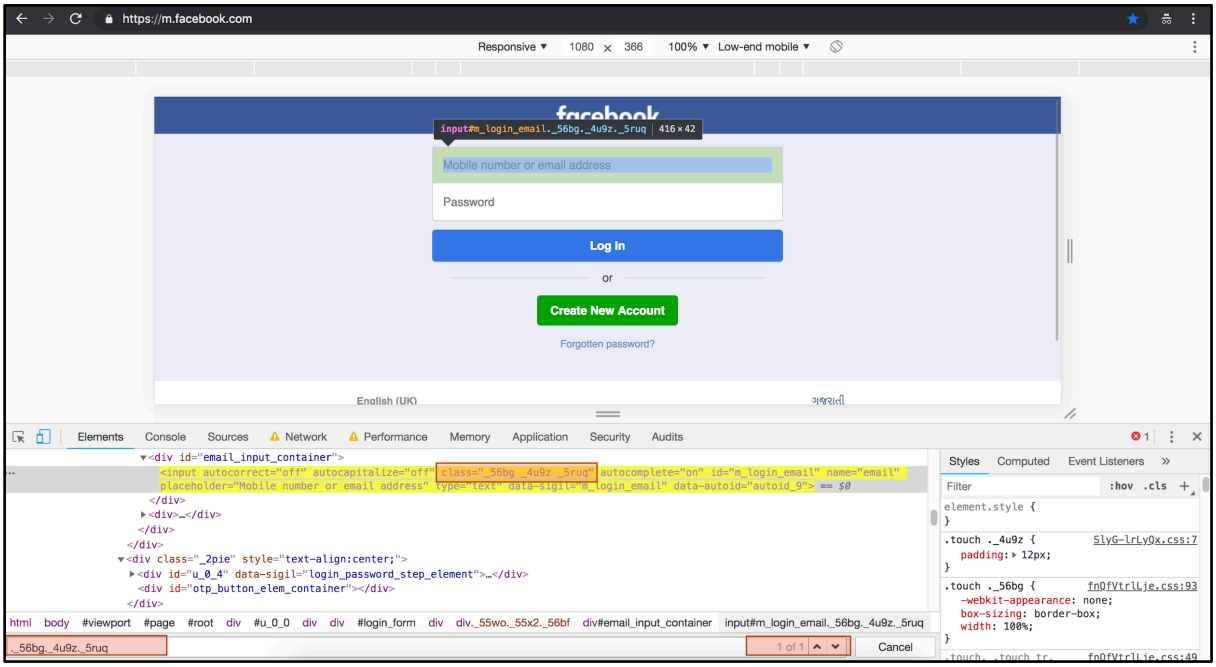

Figure-26: CSS Selector: Locator Strategy.

- In order to use **CSS Selector using Class Name**, You need to remember that **.** should be placed at the first letter of class name and every space in class name must be replaced by **.** So as we seen on previous image that class name = "_56bg _4u9z _5ruq" for "Mobile number or email address" textfield.
  But we need to place **.** before the class name so it would be **"._56bg _4u9z _5ruq"** and we also need to replace space with **.** so the final CSS Selector would be **"._56bg._4u9z._5ruq"**

- In order to use **CSS Selector using Id,** You need to place **#** before the Id.
  So for `Id = "m_login_email"` the `CSS Selector = "#m_login_email"`

- In order to use **CSS Selector using Name or other Attribute,** You need to define the attribute in square brackets like:
  `[attribute _name = "attribute_value"]`
  For Name = ""
  So for `Name = "email"` the `CSS Selector = "[name='email']"`

- **Selector Code:**

```
driver.findElement(By.cssSelector("._56bg._4u9z._5ruq"));
```

NOTE: *You can learn more about CSS selectors here:* *https://www.softwaretestinghelp.com/css-selector-selenium-locator-selenium-tutorial-6/*

5) XPath

- **XPath Selector = "//*[@class='_56bg _4u9z _5ruq']"** OR **"//input[@id='m_login_email']"** OR **"//input[@name='email']"** for the "Mobile number or email address" textfield.

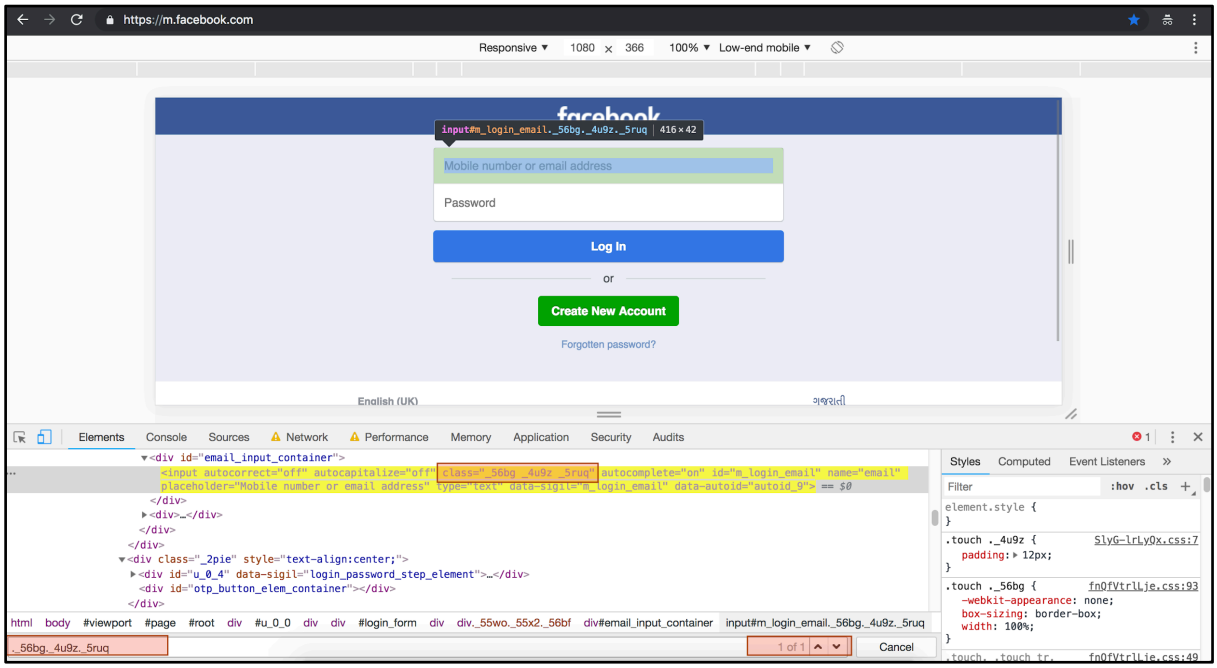

Figure-27: XPath Selector: Locator Strategy.

- **This locator should be your last option to use as it is also unreliable, unstable and has performance issues like Native Mobile Applications.**

- XPath is relative Path strategy so In order to create **XPath using Class Name**, You need to put the **//** first thing first, it means it will search the element anywhere on DOM. After **//** you can put the Tag name such as **input, a, div etc.** And last you need to put attributes in this format:

  `[@attribute _name = "attribute_value"]` so the final element XPath selector would be: `//*[@class="_56bg _4u9z _5ruq"]` OR `//input[@class="_56bg _4u9z _5ruq"]`

- **Selector Code:**

```
driver.findElement(By.xpath("//*[@class='_56bg _4u9z
_5ruq']"));
```

NOTE: *You can learn more about XPath selectors here:*
https://www.guru99.com/xpath-selenium.html

6) LinkText and Partial LinkText

- **LinkText Selector = "Help Centre"** for the Help Center link shown in the image below.

- **Partial LinkText Selector = "Help Cent"** will work the same way as above (full) LinkText locator.

- This locator strategy applies to get the UI Locator for the Link Text.

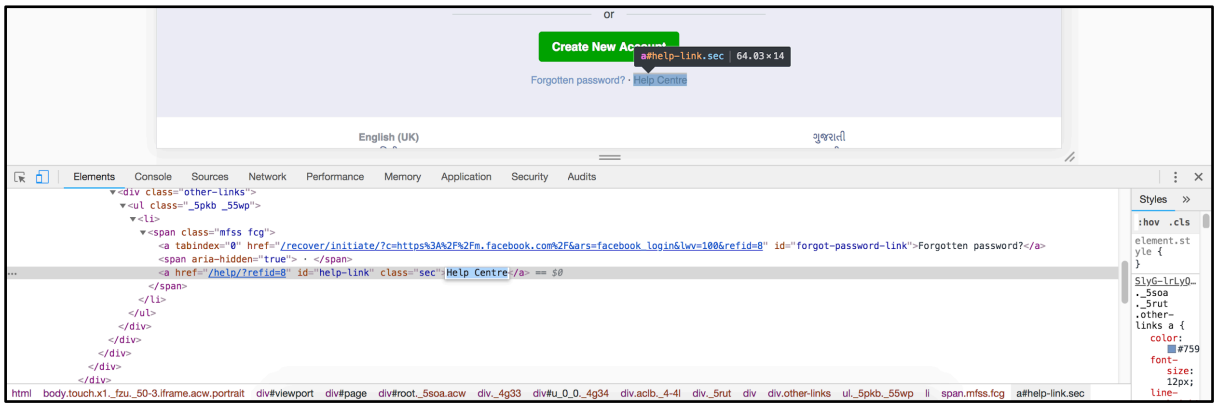

Figure-28: LinkText: Locator Strategy.

- **Selector Code(Link Text):**

```
driver.findElement(By.linkText("Help Centre")).click();
```

- **Selector Code(Partial Link Text):**

```
driver.findElement(By.partialLinkText("Help Cen")).click();
```

Mobile browser automation - Sample test case

Let's put all that theory into some practice to help make the concept more concrete. We have created a sample automation test case/script which will:

1) Open Chrome/Safari Browser on the relevant device.

2) Fill the Username and Password.
3) Click on the Login button.

Android

You can find the Android Code below:

```
import io.appium.java_client.AppiumDriver;
import io.appium.java_client.android.AndroidDriver;
import
io.appium.java_client.remote.AndroidMobileCapabilityType;
import io.appium.java_client.remote.IOSMobileCapabilityType;
import io.appium.java_client.remote.MobileCapabilityType;
import org.openqa.selenium.By;
import org.openqa.selenium.remote.BrowserType;
import org.openqa.selenium.remote.CapabilityType;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;

import java.net.MalformedURLException;
import java.net.URL;

public class AndroidBrowserTest {

   public AndroidDriver driver;

   @BeforeTest
   public void setUp() throws MalformedURLException {
       String appiumServerURL =
"http://127.0.0.1:4723/wd/hub";

       DesiredCapabilities dc = new DesiredCapabilities();
       dc.setCapability(MobileCapabilityType.PLATFORM_NAME,
"Android");

dc.setCapability(MobileCapabilityType.PLATFORM_VERSION,
"8.0");
       dc.setCapability(MobileCapabilityType.BROWSER_NAME,
BrowserType.CHROME);
       dc.setCapability(MobileCapabilityType.DEVICE_NAME,
"c4e3f3cd");
```

```
        dc.setCapability(MobileCapabilityType.AUTOMATION_NAME,
"UiAutomator2");

        driver = new AndroidDriver(new URL(appiumServerURL),
dc);
    }

    @Test
    public void verifyUserCanLoginToFaceBook() throws
InterruptedException {
        String username = ""; // Enter your valid facebook
username
        String password = ""; // Enter your valid facebook
password

        driver.get("https://m.facebook.com/");

driver.findElement(By.id("m_login_email")).sendKeys(username)
;

driver.findElement(By.id("m_login_password")).sendKeys(passwo
rd);
        driver.findElement(By.name("login")).click();
    }
}
```

iOS

You can find the iOS Code below:

```
import io.appium.java_client.ios.IOSDriver;
import io.appium.java_client.ios.IOSElement;
import io.appium.java_client.remote.MobileCapabilityType;
import org.openqa.selenium.By;
import org.openqa.selenium.remote.BrowserType;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.testng.Assert;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;

import java.net.MalformedURLException;
import java.net.URL;
```

```
public class iOSBrowserTest {
    public IOSDriver<IOSElement> driver;

    @BeforeTest
    public void setUp() throws MalformedURLException {
        String appiumServerURL =
"http://127.0.0.1:4723/wd/hub";

        DesiredCapabilities dc = new DesiredCapabilities();
        dc.setCapability(MobileCapabilityType.PLATFORM_NAME,
"iOS");

dc.setCapability(MobileCapabilityType.PLATFORM_VERSION,
"11.4");
        dc.setCapability(MobileCapabilityType.BROWSER_NAME,
BrowserType.SAFARI);
        dc.setCapability(MobileCapabilityType.DEVICE_NAME,
"iPhone X");

        driver = new IOSDriver<IOSElement>(new
URL(appiumServerURL), dc);
    }

    @Test
    public void verifyValidUserCanLoginToFaceBook() throws
InterruptedException {
        String username = ""; // Enter your valid facebook
username
        String password = ""; // Enter your valid facebook
password

        driver.get("https://m.facebook.com/");

driver.findElement(By.id("m_login_email")).sendKeys(username)
;

driver.findElement(By.id("m_login_password")).sendKeys(passwo
rd);
        driver.findElement(By.name("login")).click();
    }
}
```

You can find this project on our Github page.

Chapter-6: Walkthrough of UIAutomator for Android and Accessibility Inspector for iOS for Element Extraction.

In the last chapter we discussed the Appium Inspector tool and learned how we can extract the elements for any application. Appium Inspector is a great tool to extract elements from Android and iOS both. However, it can take some time to do so.

There is another way: UiAutomatorViewer(Mac & Windows) and Accessibility Inspector(Mac) are handy tools to extract the elements from Android and iOS Applications respectively.

The reason why UiAutomatorViewer and Accessibility Inspector is fast because it does not involve the application installation part, it just fetches the XML structure of the current screen on the device, no matter which application is open and displayed.

In this chapter we will look into *UiAutomatorViewer and Accessibility Inspector* tools.

1. UiAutomatorViewer:

UiAutomatorViewer is the Android SDK part and it's packaged with it, so you don't need to install it separately. It's a tool like Appium Inspector which lets you inspect the UI(XML Structure) of the application and gives you the attributes of UI element.

NOTE: *Before using this tool make sure SDK is properly downloaded and the PATH is set properly.*

1) The first step to work with UiAutomatorViewer is you need to connect the Real(Physical) Android device to your computer (using USB cable).

2) Once you connect the device you can find the device name using: `$ adb devices`

```
abcs-MBP:~       $ adb devices
List of devices attached
c4e3f3cd         device
```

Figure-1: Android device is connected.

If your device is not connected properly then you might get error: **"No Android devices were detected by adb."**

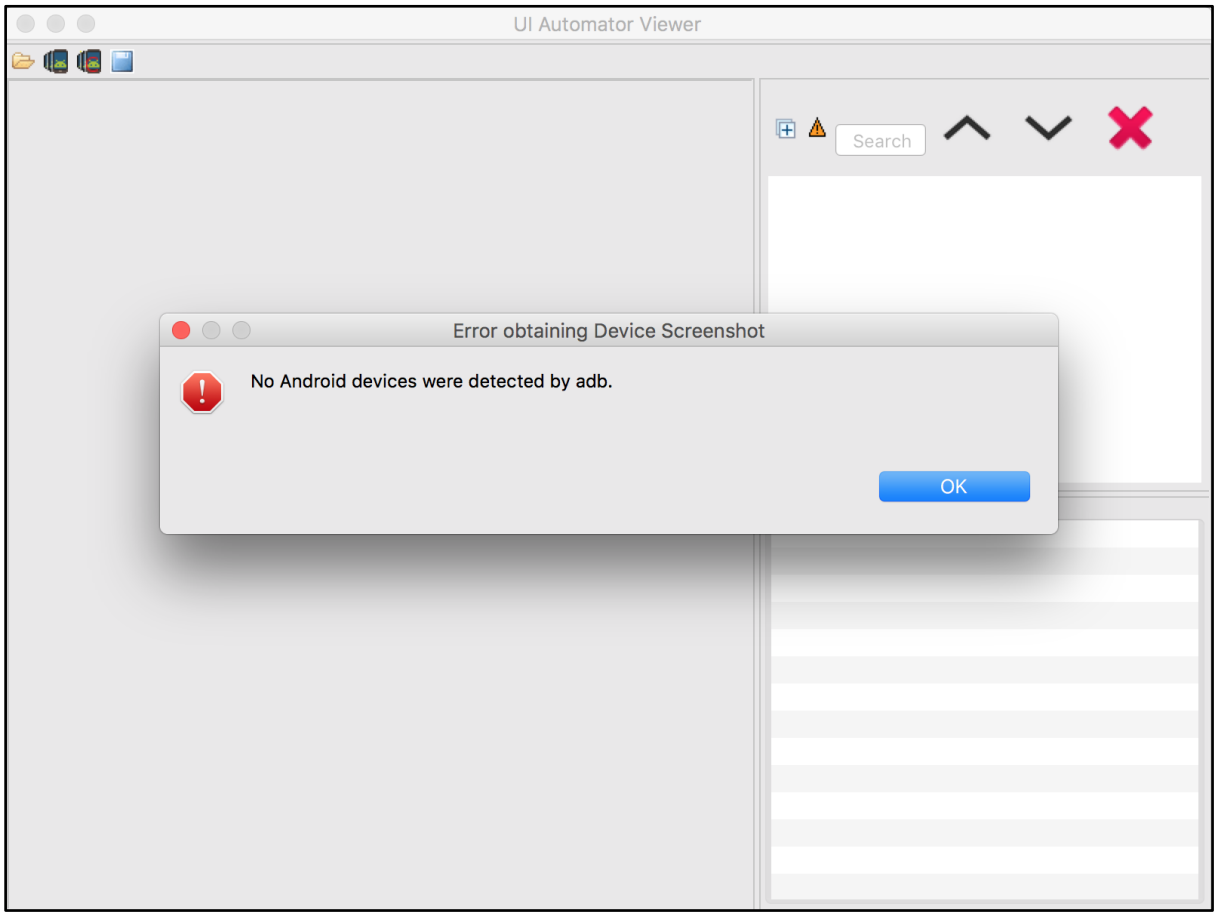

Figure-2: No Android device is connected error dialog.

3) After connecting the device, you need to open the UiAutomatorViewer from SDK directory. You can find UiAutomatorViewer under *Android SDK>tools>bin* directory.

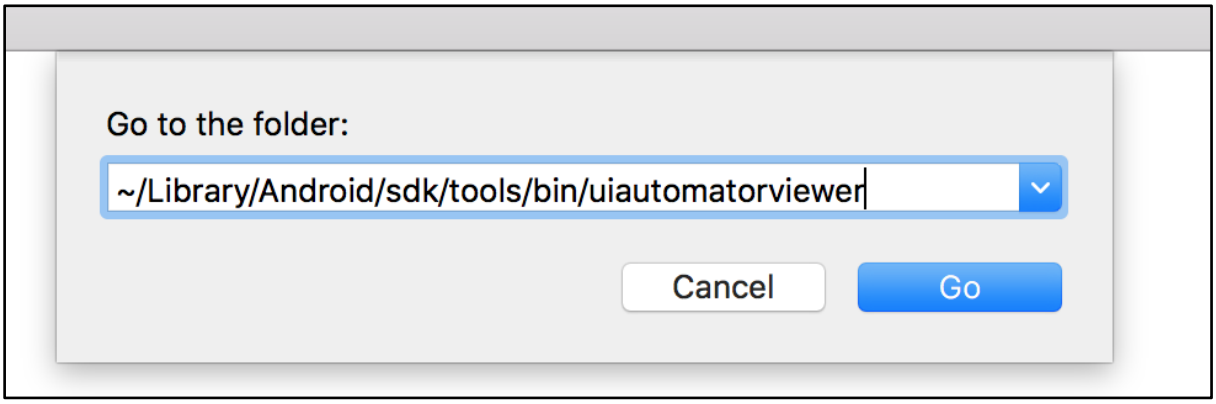

Figure-3: Path to UiAutomatorViewer (on Mac).

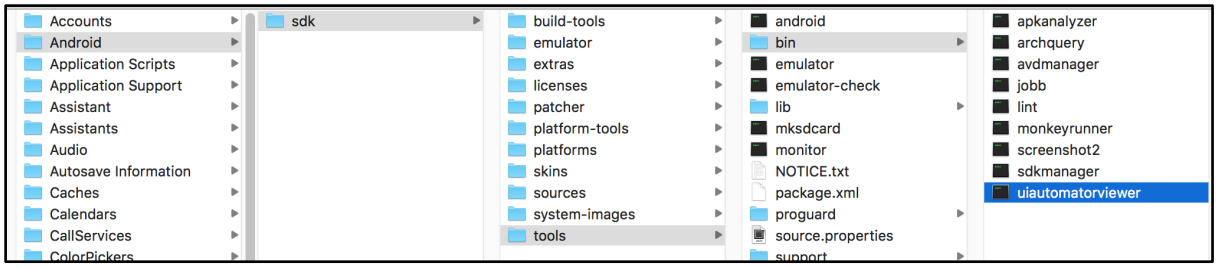

Figure-4: UiAutomatorViewer (on Mac).

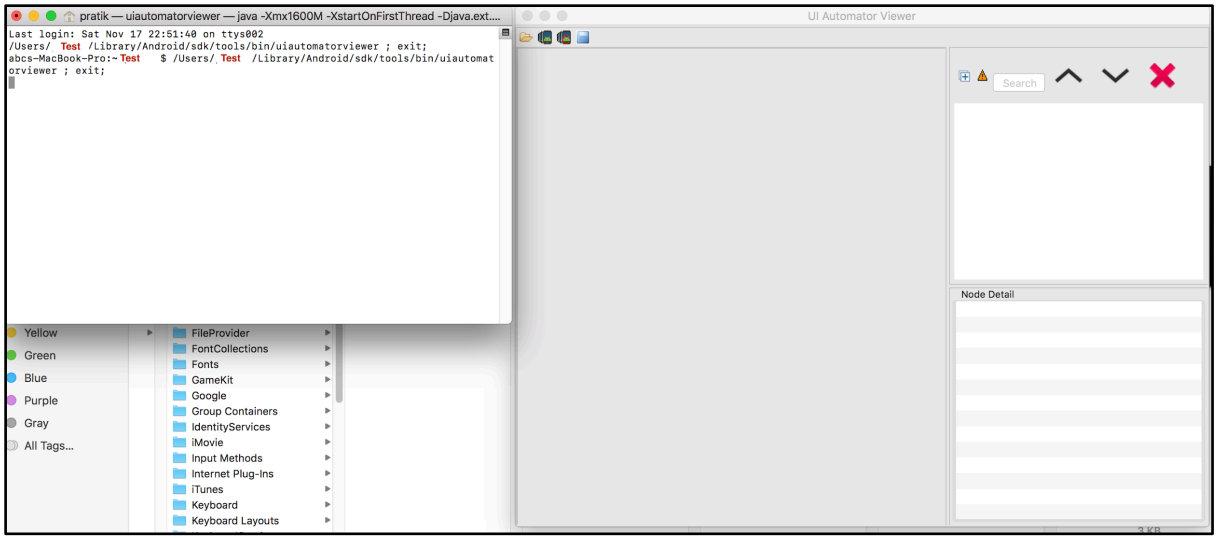

Figure-5: UiAutomatorViewer Window (on Mac).

4) Now open the specific application screen on the connected device for which you want to extract the elements.

5) In order to fetch that screen's XML structure you need to press the (second) mobile icon on UiAutomatorViewer window.

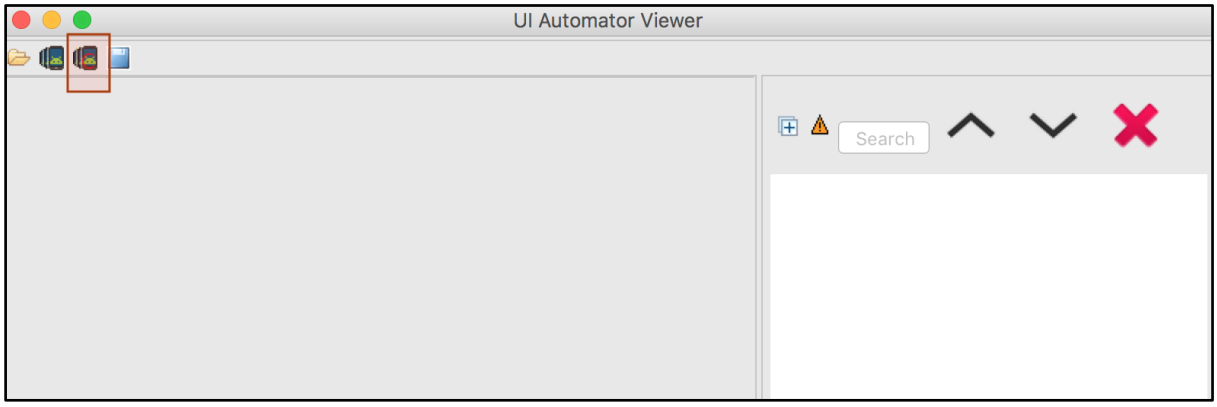

Figure-6: Click on second Mobile icon.

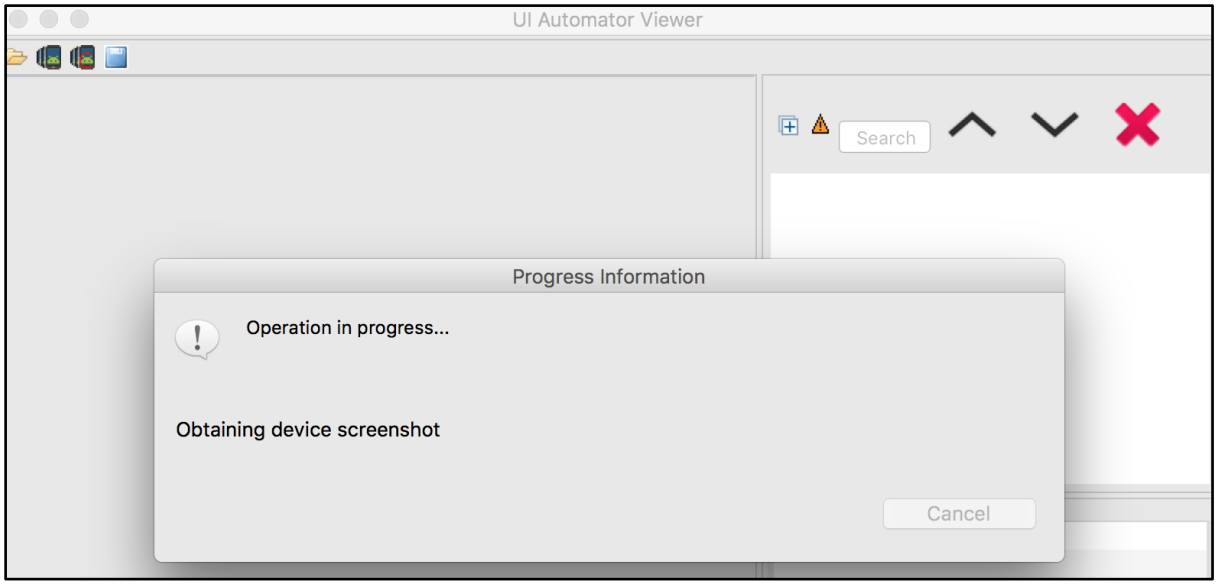

Figure-7: Obtaining Device screenshot and fetching XML structure.

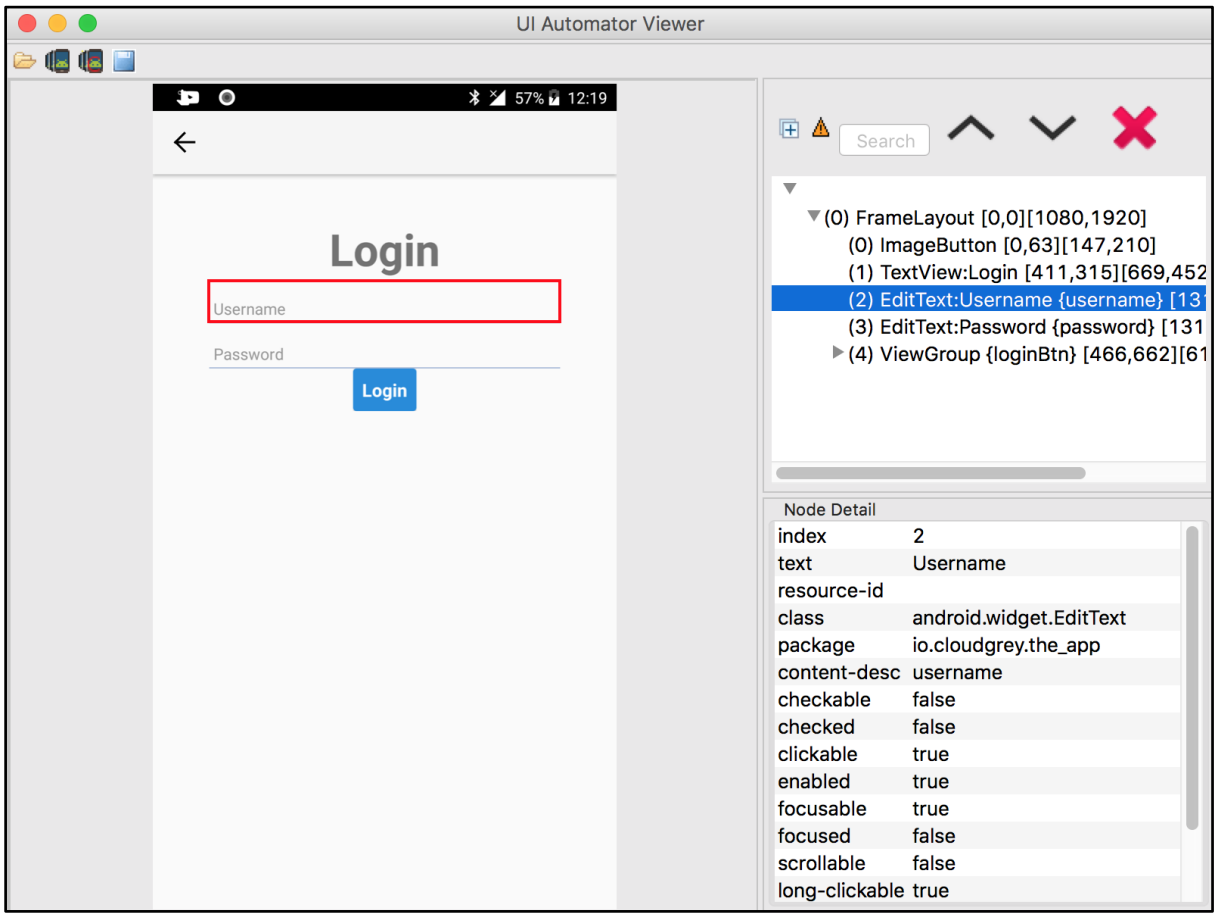

Figure-8: Sample application screen.

You can observe here that left side part gives you the screenshot of the current screen from your connected device and the right side is divided into 2 parts.

1) The upper half contains the XML hierarchy of Screen and selected node.
2) The lower half contains the selected node's attributes with their values.

Now you can get the valid selectors such as cont-desc(accessibility id), id, class name, xpath etc. from the attributes section and start automating the application right away.

The following table gives the mapping between attributes and Appium locator strategies:

Attribute	Locator Strategy
text	Name
resource-id	Id
class	Class Name
content-desc	Accessibility Id

So, as you can see this tool is similar to the Appium Inspector, but the only difference is this tool doesn't take much time to get UI element locators.

2. Accessibility Inspector:

Accessibility Inspector is a common Inspector tool included in XCode and specially designed for Mac OS to get the basic details such as Label, Title, Value and Type for any UI element from opened application on Mac OS.

It does not give many details and attributes of UI elements but it is handy tool to get the basic information about the element rapidly.

1) On Spotlight Search, search for the Xcode and open it.

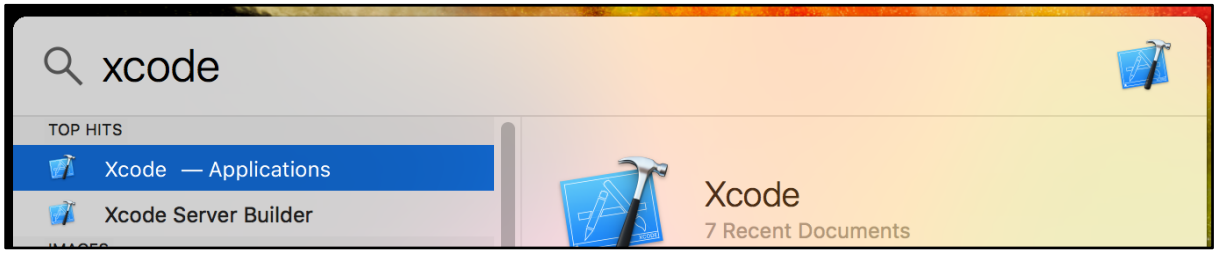

Figure-9: XCode on Spotlight Search.

2) After XCode is open, on the Menu bar Select the *XCode > Open Developer Tool > Accessibility Inspector*

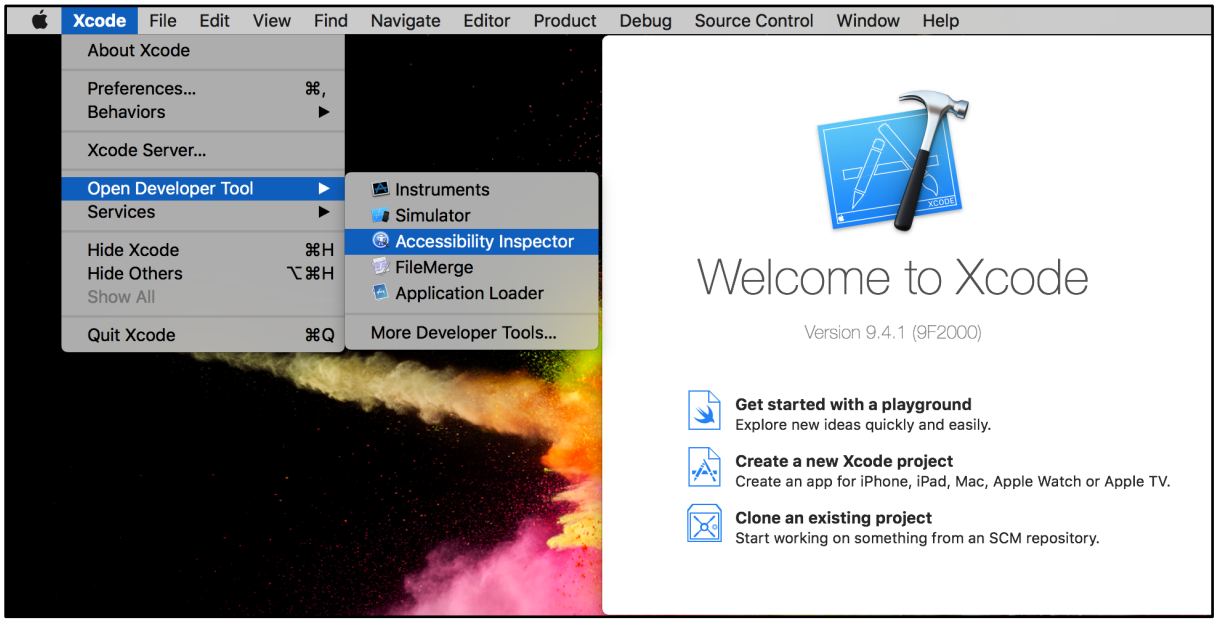

Figure-10: Accessibility Inspector on XCode.

3) Below is the dialog of Accessibility Inspector.

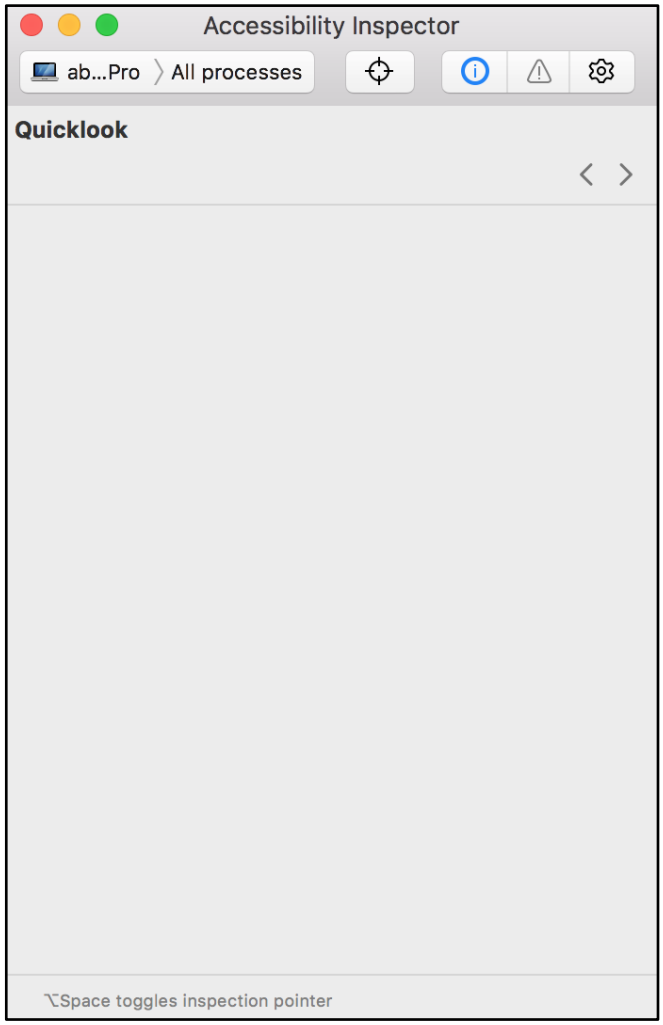

Figure-11: Accessibility Inspector.

4) Now open iOS Simulator and open the application for which you want to extract the locators.

5) Now click on this icon:  and select the ***"Compute Sum"*** textfield on Simulator.

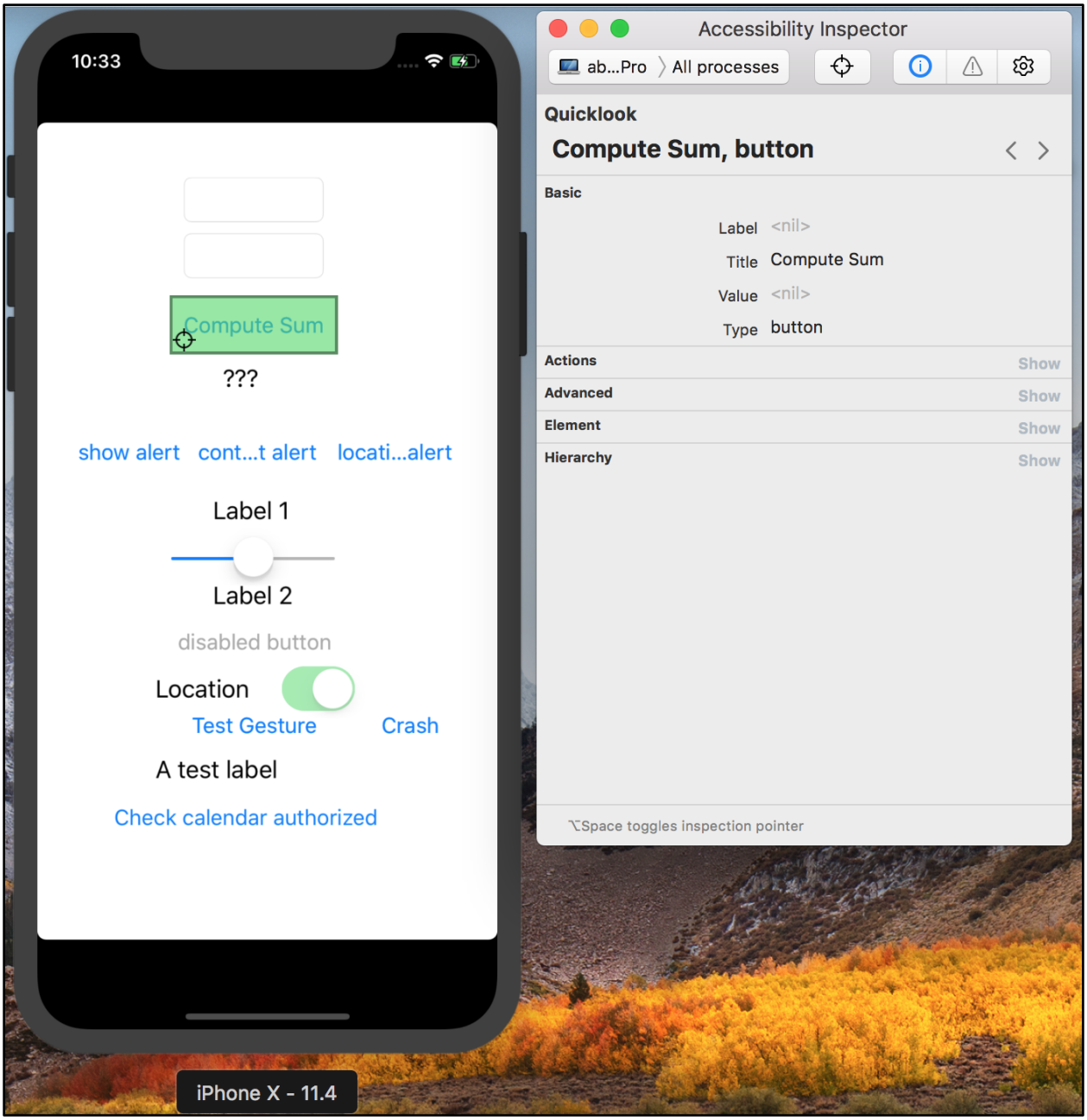

Figure-12: Locate the selector on Accessibility Inspector.

As you can see you can get the Basic attributes such as Label, Title, Value and Type.

While with the Appium Inspector you get many more attributes:

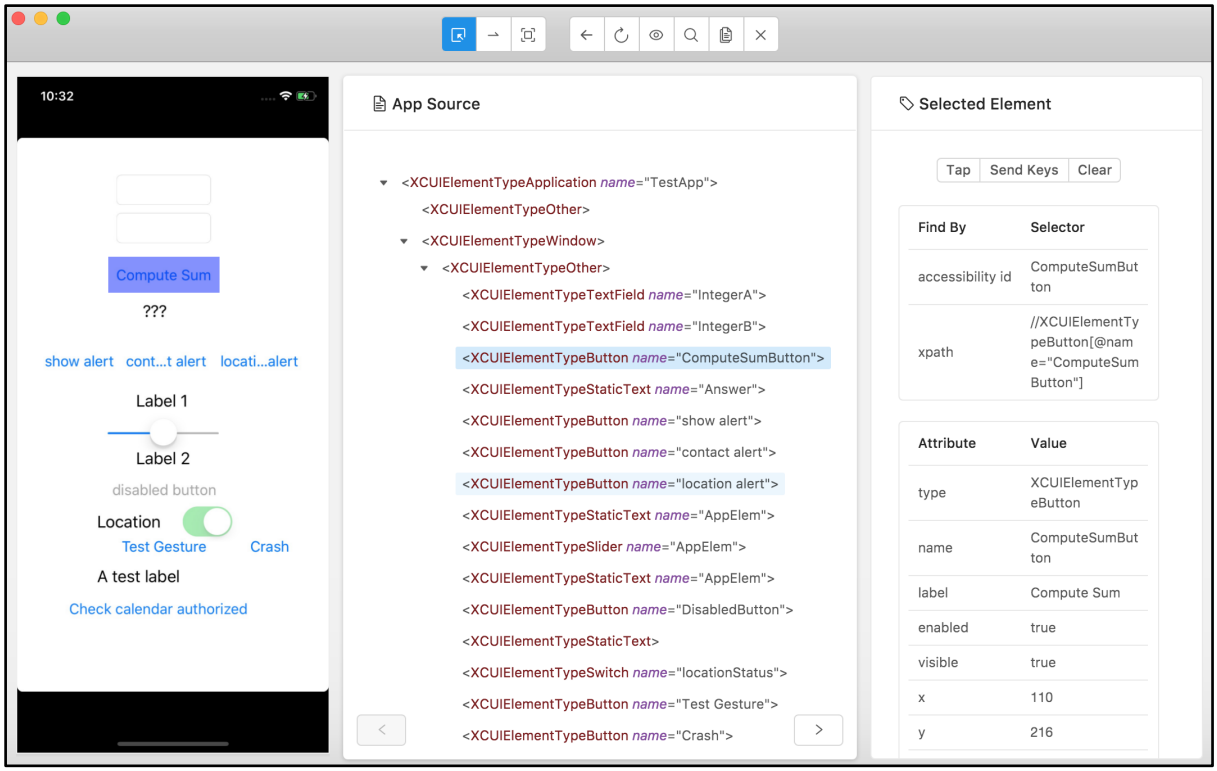

Figure-13: Locate the selector on Appium Inspector.

Here, Appium Inspector adds more value to the iOS part, whereas the Accessibility Inspector provides only basics values which in some case may not be sufficient.

There are also many third party software and tools available which helps to identify the UI locators.

Ultimately you will figure out which tool best fits into your workflow, but it is a good idea to have a broad understanding of the choices at your disposal.

Chapter-7: Developing a Test Automation Framework for Appium using Page Object Modeling(POM).

Automation Testing with Appium fundamentally boils down to a simple 2-step process:

1. Identify the UI Element locator(address)
2. Perform an action on it.

So far we have looked into the basics of Appium and learned how can you build a simple test case.

But that was just the beginning! It's time to ramp our skills up a notch.

Now that you know how to locate elements, we could continue exploring the different actions you can perform on those elements. However, we're going to leave that for a later section, and cover an important topic. Technically, the material covered here is optional, but we'd urge you to follow along.

In the real world, Appium is used to automate an entire mobile application and the simple idea to put all element locators, and interactions with those locators, into one file (as we have been doing) won't help us. It's just bad design...when we inevitably go back to increase automation test coverage we would likely end up with an unmaintainable project - large project files, complex code and duplicate usage of element locators will become the bane of your daily automation life.

Moreover, even a small change in the application UI would break the existing working locators, and if we use the linear structure in our test code it will become so difficult to fix that locator because we need to replace the invalid locator from each place in the code.

For Example, most apps or websites have a 'home' page, such as a dashboard, containing a number of menu options. Many automation test cases might click on these menu options to perform a specific task. Now imagine that the UI is changed/revamped and menu buttons are relocated and some of them removed -

this will lead to automation tests failure because scripts will not be able to find the particular element.

So in order to reduce that pain we need to use some kind of structure which can eliminate those difficulties. And that masterpiece code structure (or framework as it's more commonly called!) is known as **Page Object Modeling**.

Page Object Modeling(POM)

Page Object Model is a popular and widely used Design Pattern in Appium (and Selenium) Test Automation.

It is popular because it enhances test maintenance and reduces code duplication. The main logic behind the Page Object Model framework is to keep Locators and Tests cases separate from each other. This allows you to easily update locators when the app changes, without affecting your test cases. It's a cleaner abstraction of duties.

Page Object is an object-oriented class that keeps all the element locators referring to a particular page of your Application Under Test and it has interaction methods for all relevant locators that have been defined. These will be used by the Test Cases in a particular order according to the test requirements of the feature being tested.

The main advantage to using this is that whenever a UI change causes a test script failure, you only need to apply changes on Page Object classes to fix the automation script.

The basic structure of the Page Object Model framework is depicted below:

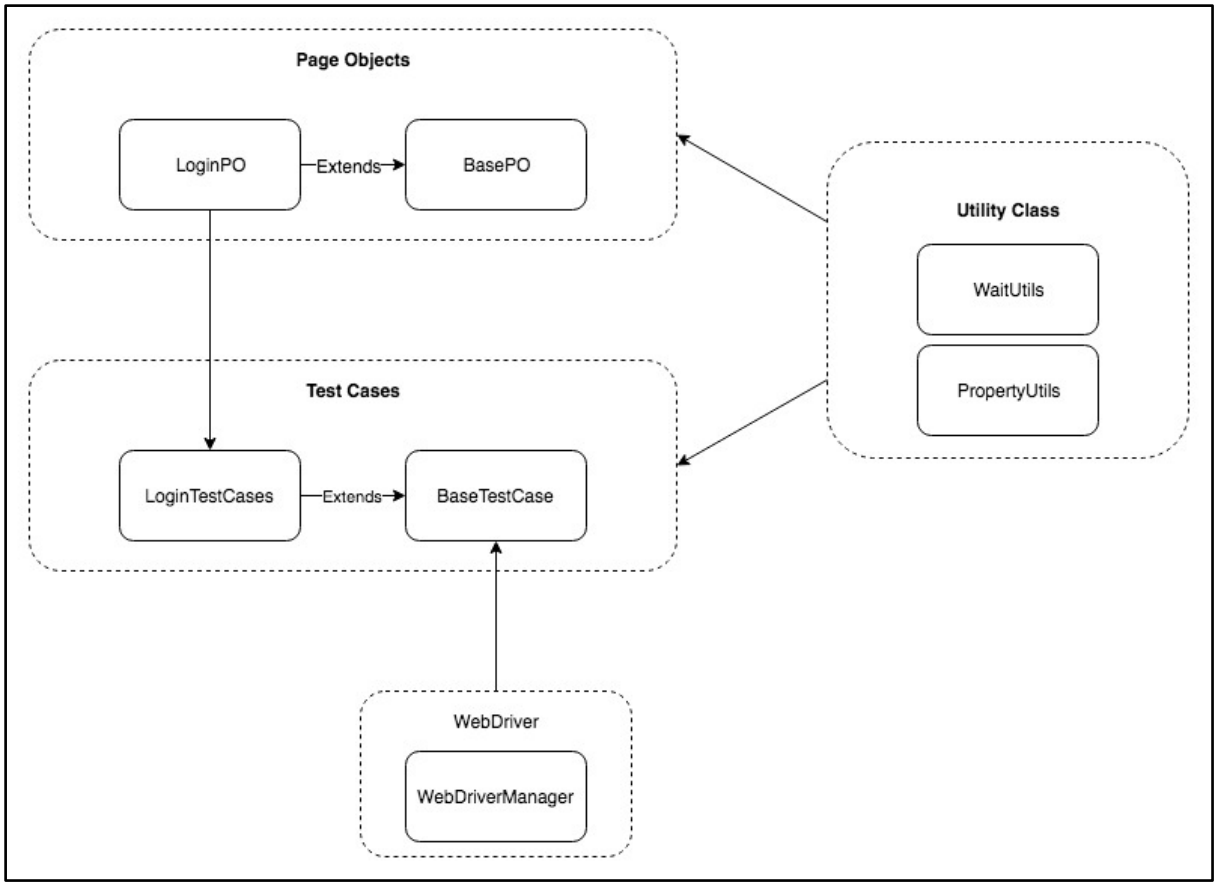

Figure-1: Page Object Modeling Structure(POM).

NOTE: *The above structure just illustrates one possible Page Object Model structure - It may vary according to the needs of your app and test cases, and POM works best for multi page applications.*

You don't need quite as complex a POM structure as shown above, especially when you're learning. In this tutorial we will use a "lighter" version of POM illustrated below:

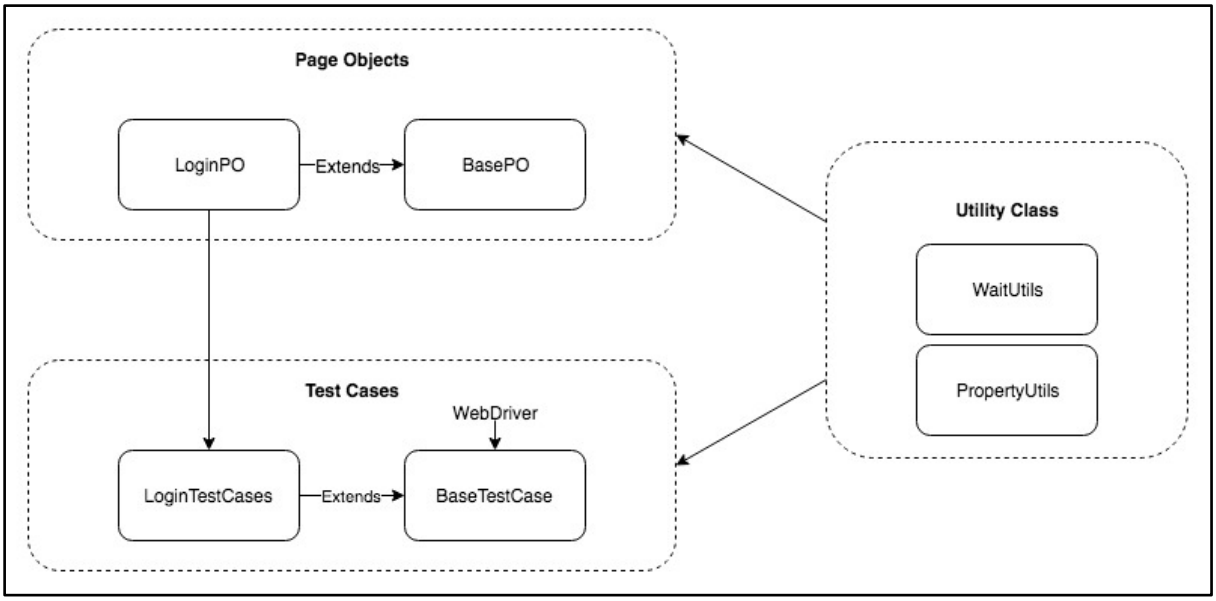

Figure-2: Page Object Modeling Structure(Light-weight).

We have removed the WebDriver as separate entity and included it in the BaseTestCase. The reason for doing so is we want to simplify things, and WebDriverManager makes more sense when we are working with many different test execution clients(browsers).

It is especially useful to have the WebDriverManager class when working with Selenium as there are many browsers out there such as Chrome, Firefox, Safari, Opera etc and each have some specific Desired Capabilities.

But here we are dealing with iOS and Android Devices only so in lieu of creating a separate WebDriverManager class we can include the WebDriver creation logic inside of BaseTestCase.

Follow the below steps to implement this (steps shown below remain same for Intellij IDEA and Eclipse IDE):

1) Create a new Java Project:

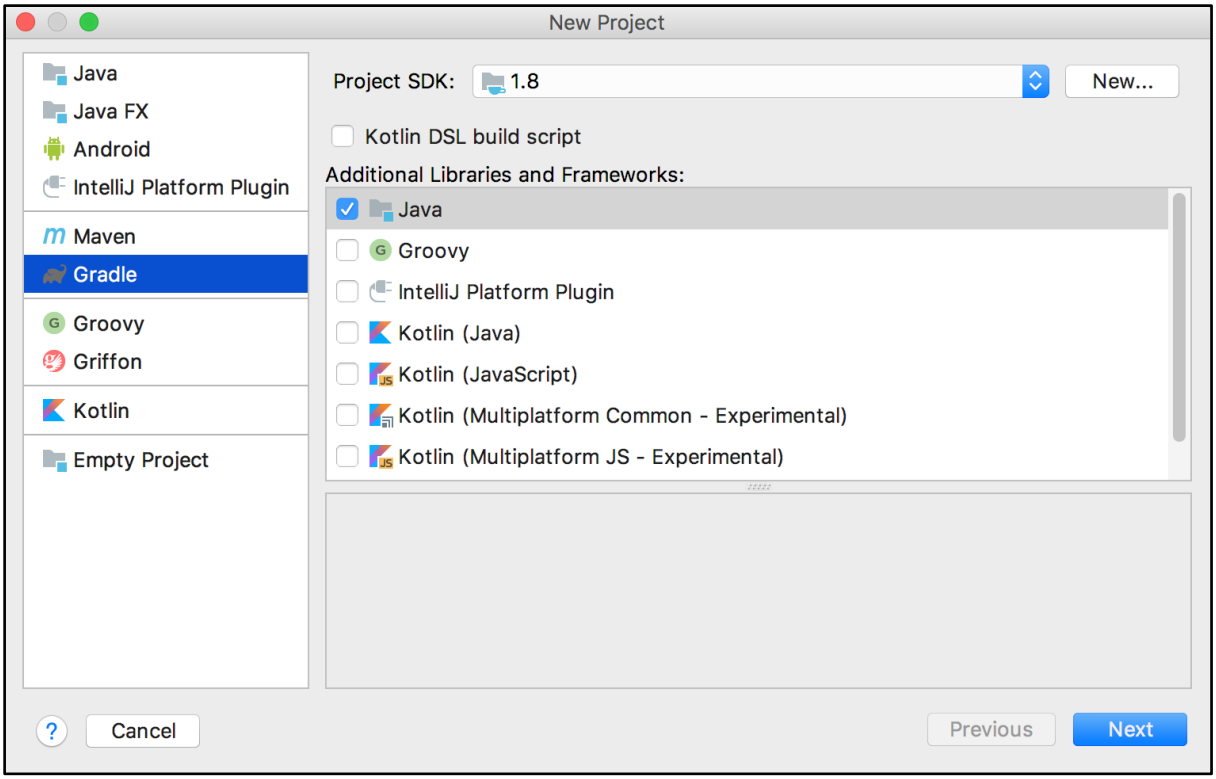

Figure-3: Create Java Project.

2) Give a valid GroupId and ArtifactId:

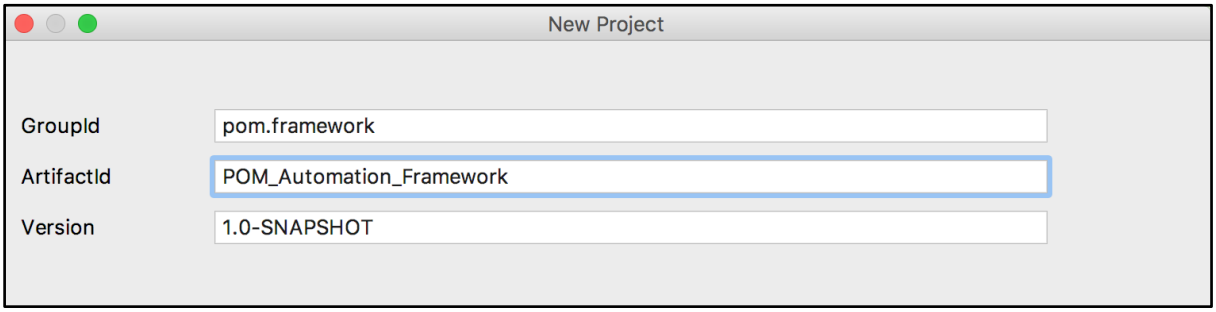

Figure-4: GroupId and ArtifactId.

3) Check the configuration and click on Next button.

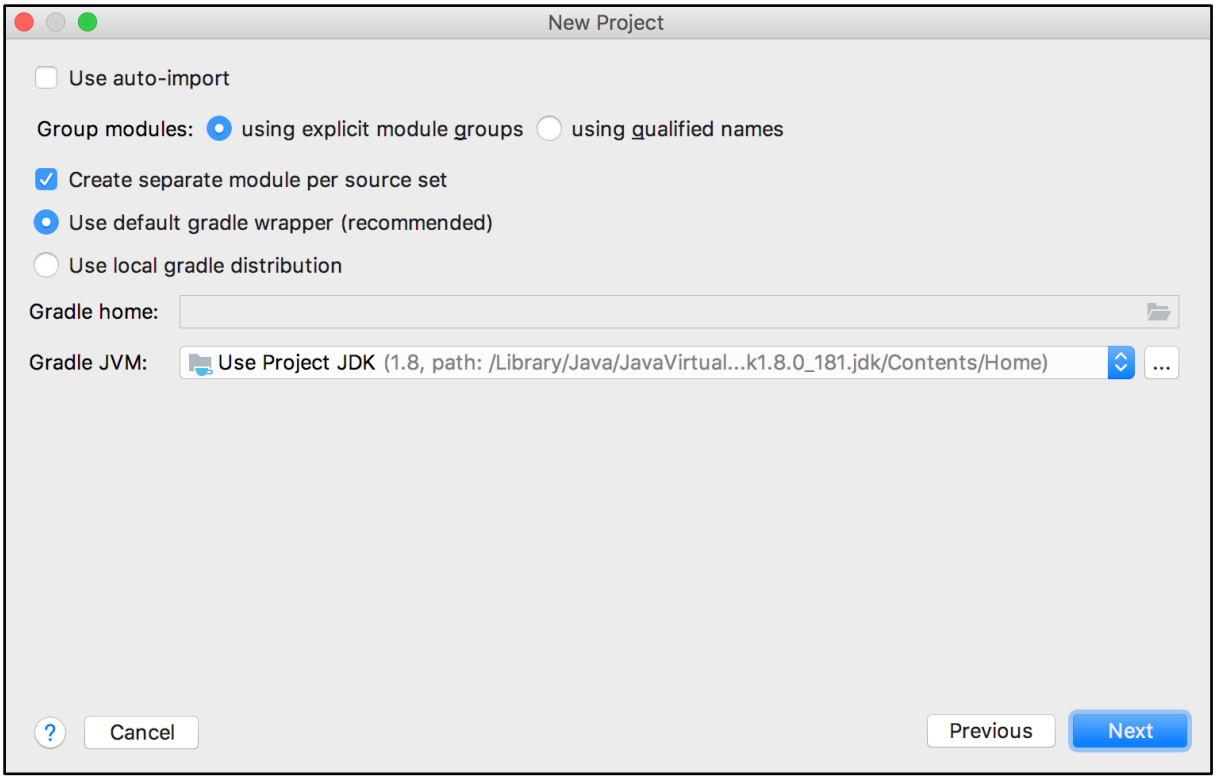

Figure-5: Check the Configuration.

4) Check the Project Name, Project Location and More Settings.

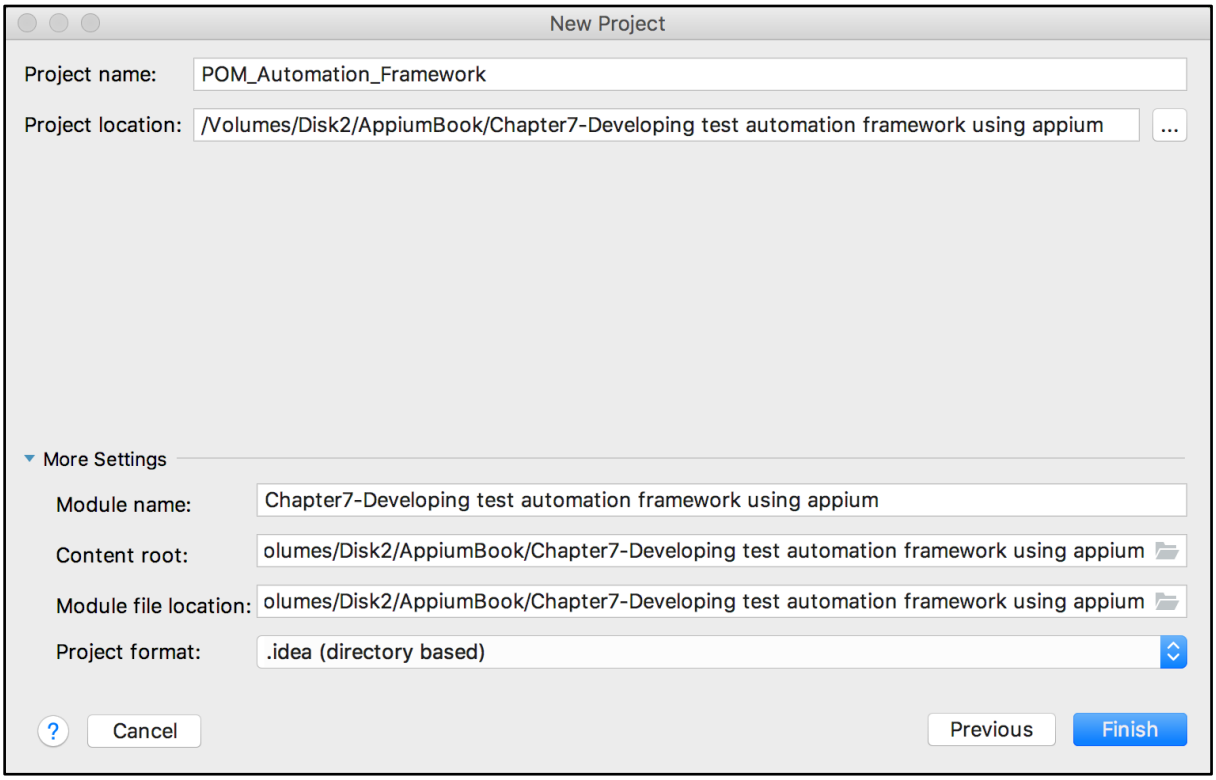

Figure-6: Name of New Project and other details.

5) When you click on the Finish button it will build the whole project and link the default dependencies.

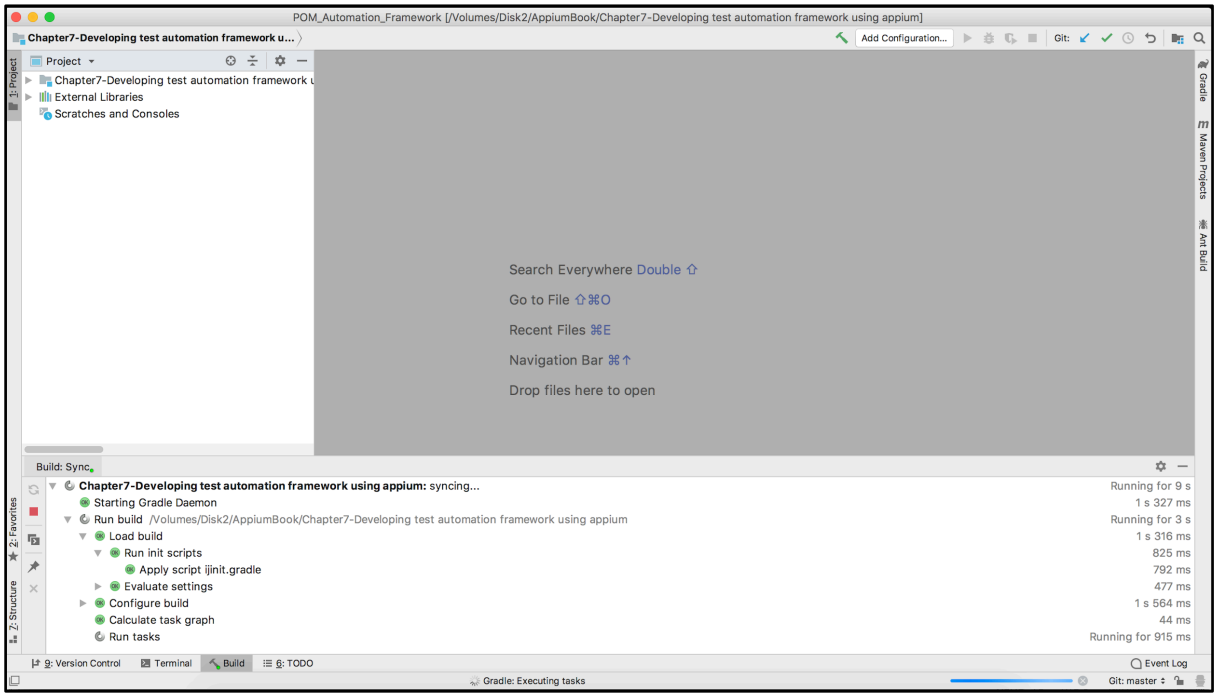

Figure-7: Gradle Build.

6) As we are using Gradle as a build tool(No offense to Maven), the first thing you need to do is to add the gradle dependencies and Import the changes.

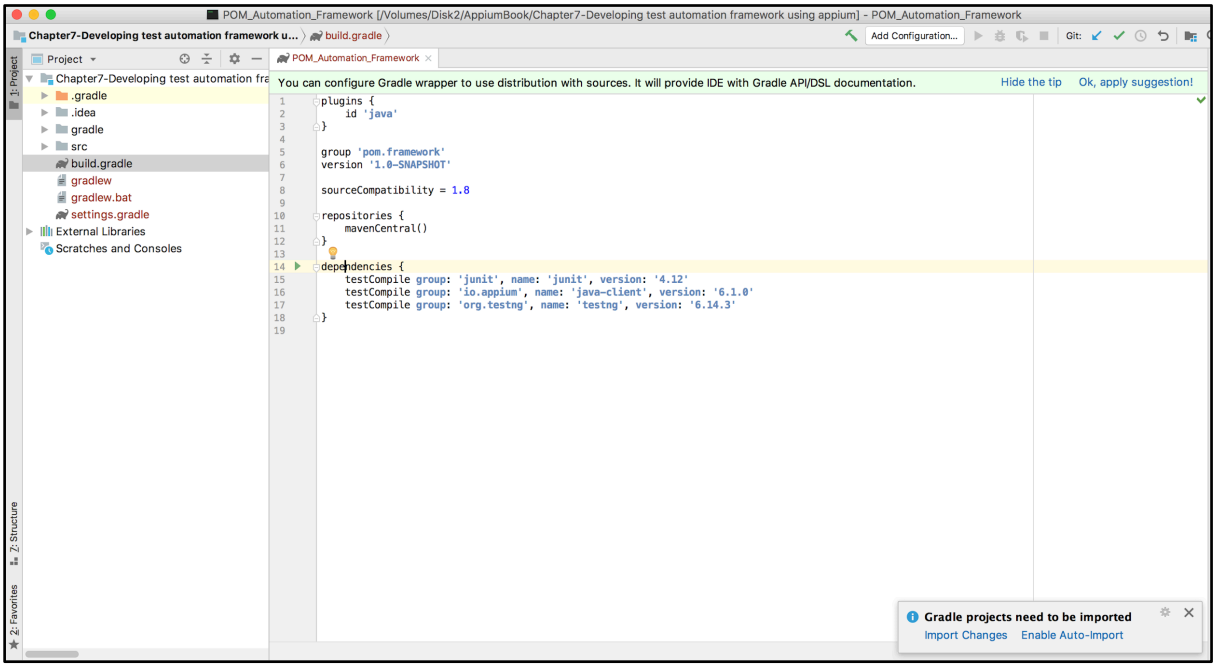

Figure-8: build.gradle

7) You can observe that there is a **src** directory created by default (By Gradle).

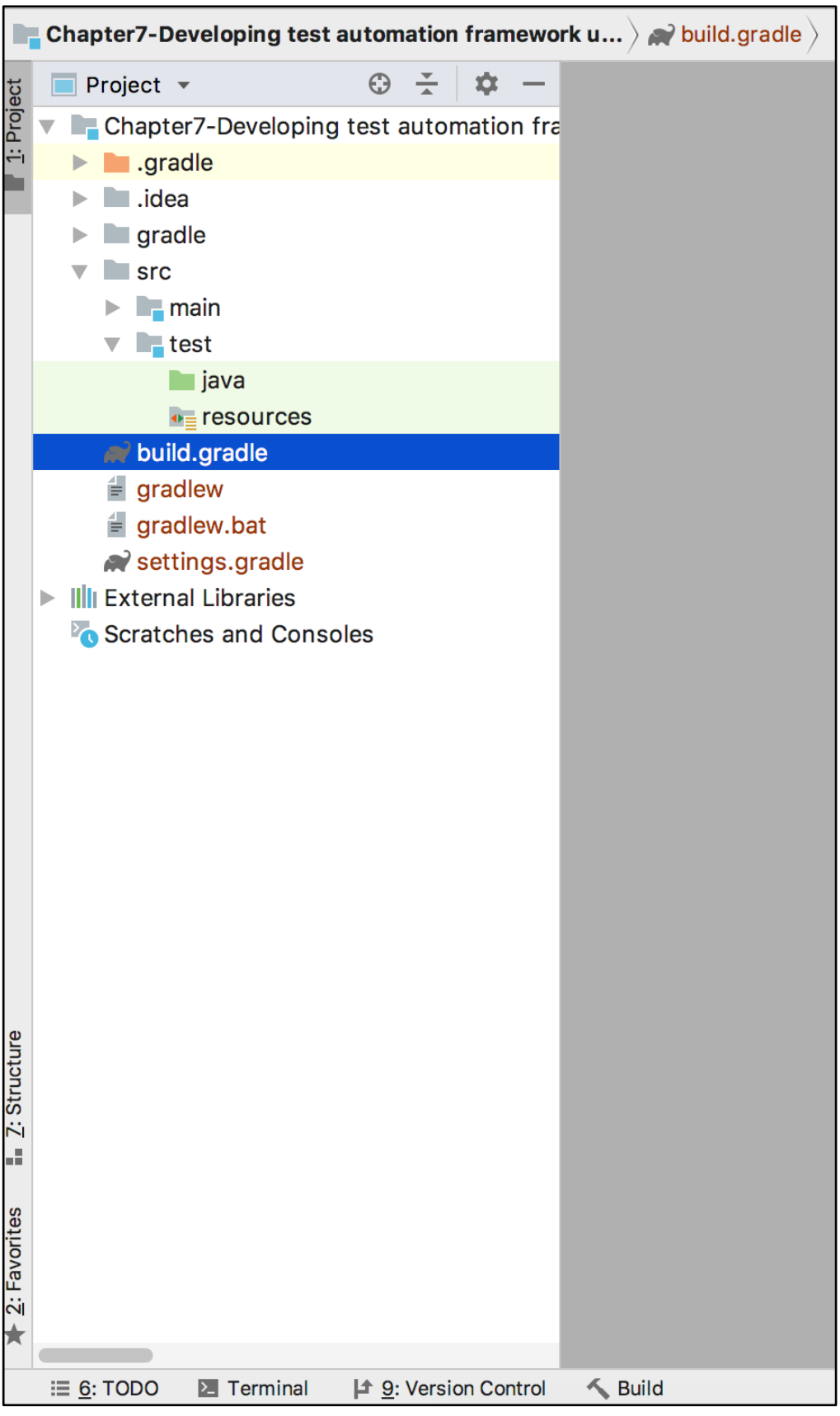

Figure-9: Directories created by Gradle.

8) After adding the Appium and TestNG dependencies we will create the Page Object package inside ***src > test > java***

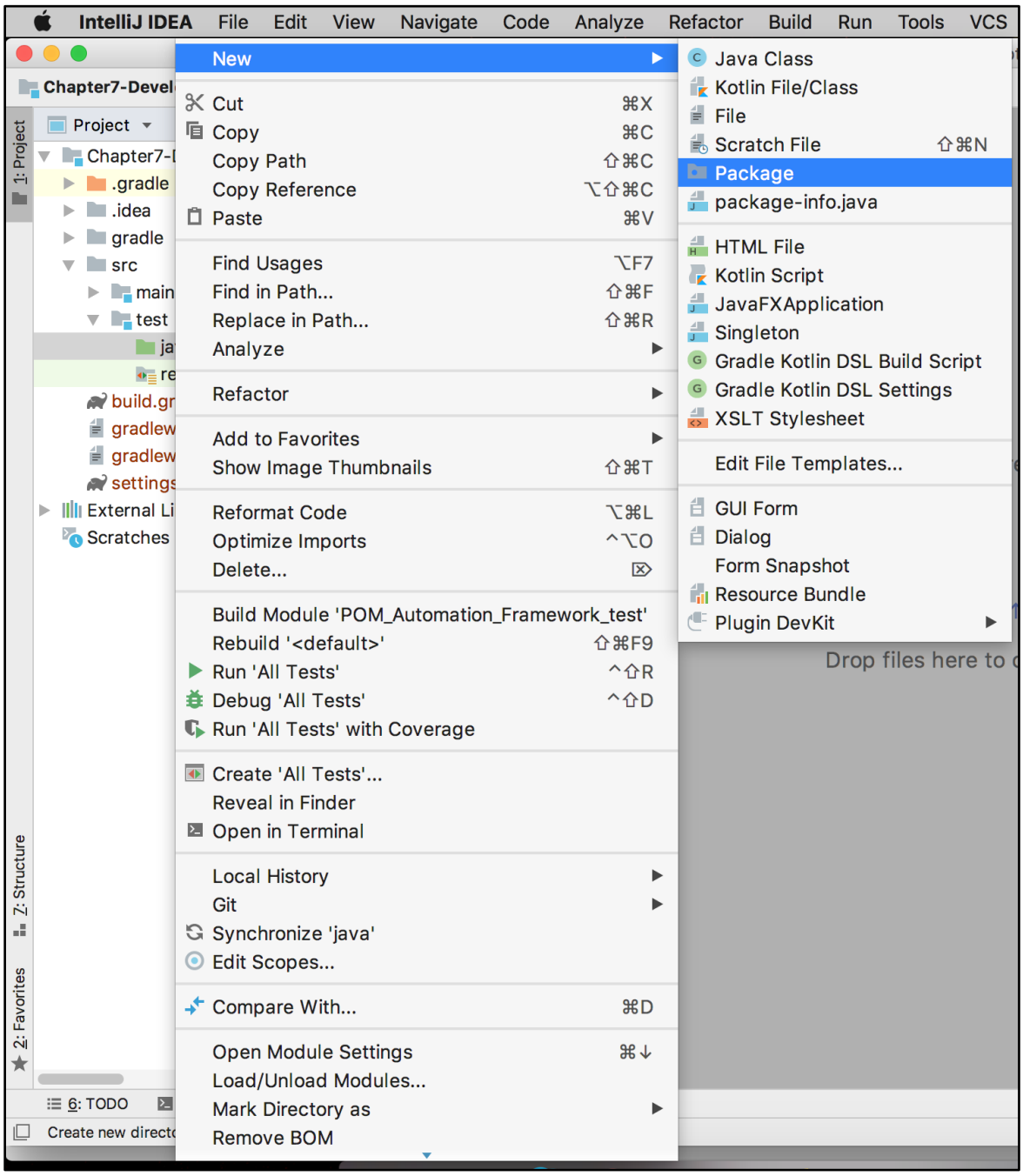

Figure-10: Adding new package inside src/test/java.

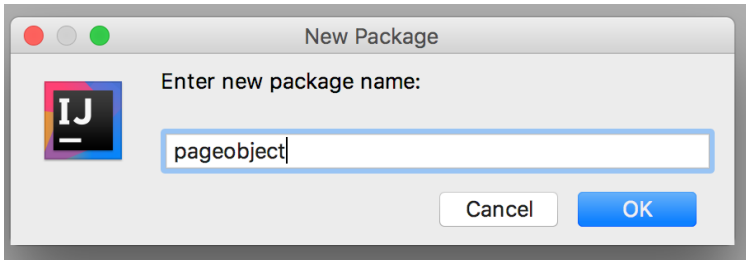

Figure-11: Package Name.

9) Using the same approach add a ***testcases*** and utils ***package***.

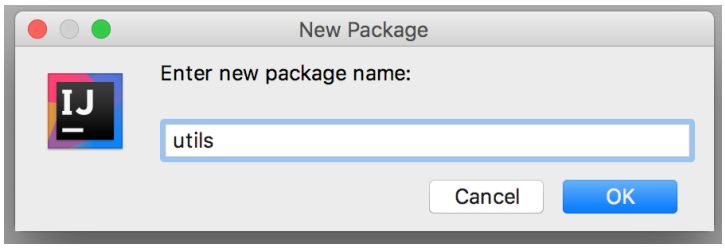

Figure-12: Package Name: utils.

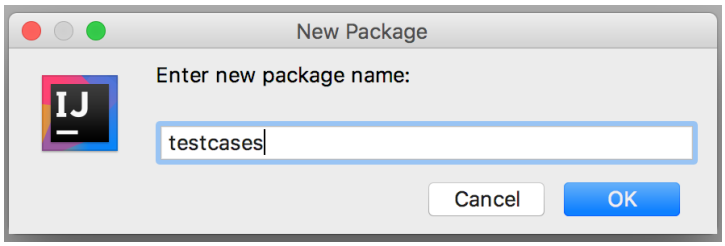

Figure-13: Package Name: testcases.

10) The first step is to add the **configuration.properties** file under ***resources/*** directory. Now what do we mean by this? According to the Page Object Model we need to put all the configuration related values in one file and in Java we can use a ***.properties*** file for it.

A ***.properties*** file contains key/value pairs so if we want to change some configuration like changing the connected device name, we just need to change the value of **android.device.name** key. This means less changes to our code when things change.

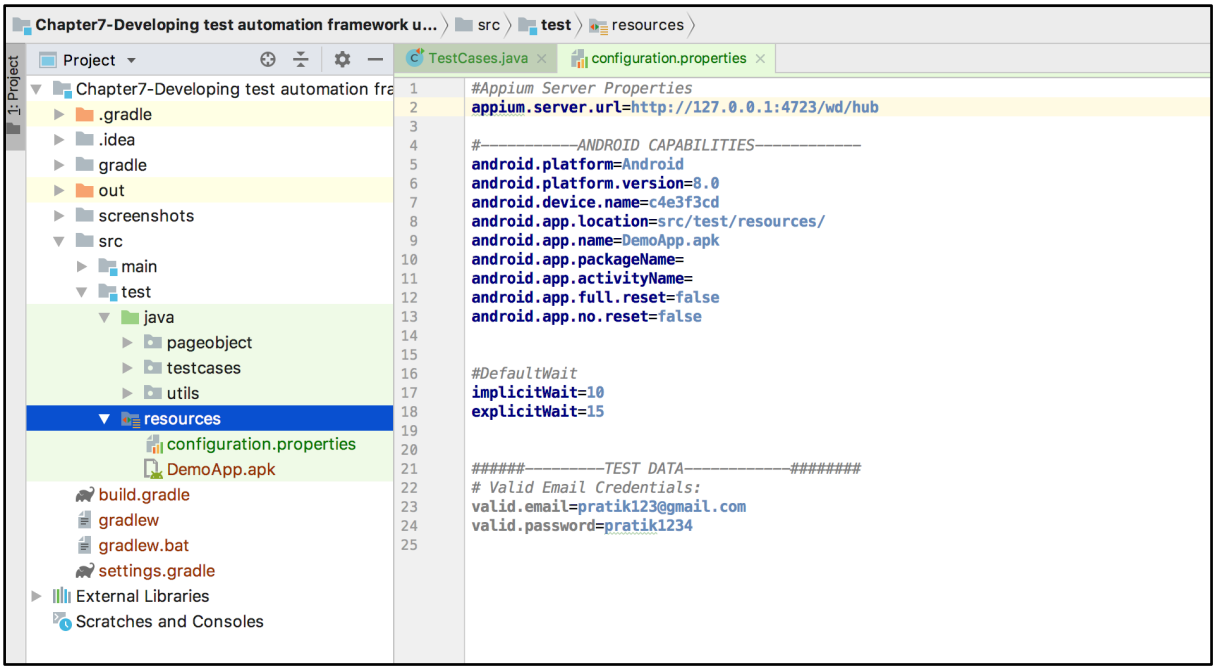

Figure-14: configuration.properties

To get the value of a key from the .properties file we need to implement methods and we will use the PropertyUtils class for that.

11) Now we need to add Utility classes such as PropertyUtils, WaitUtils etc.

- In test automation **Wait** has got a vital role. In application automation when you navigate to another screen or page, there will be a delay due to resource loading and if we don't apply the wait then Automation test scripts can break. That is, the script is attempting to operate on elements that have not yet rendered or loaded. So to avoid that we need to tell Appium's webdriver object that we are expecting some delay to get the mobile element on the new screen/page and this is known as a **Wait.**

  There are basically 3 Types of waits in Appium (or Selenium) and we need to use various forms of them at various times, so we can create useful methods based on the wait and put them together in the WaitUtility Class. Then, whenever we need to use the wait we will use the WaitUtility class.

  *We will see more about **Wait** in an upcoming chapter. Here is our code which you can use right now, and you will learn more about how it works in a subsequent section. For right now, just know that you have this WaitUtility class that you will use.*

WaitUtils.java

```
package utils;

import io.appium.java_client.ios.IOSElement;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

import java.util.List;
import java.util.concurrent.TimeUnit;

/**
 * This will contain all wait related utility methods.
 *
 * @author prat3ik
 */
public class WaitUtils {

    public final int explicitWaitDefault =
PropertyUtils.getIntegerProperty("explicitWait", 10);

    /**
     * This method is for static wait
     *
     * @param millis
     */
    public void staticWait(final long millis) {
        try {
            TimeUnit.MILLISECONDS.sleep(millis);
        } catch (final InterruptedException e) {
        }
    }

    /**
     * To wait for button to be clickable
```

```
     *
     * @param driver
     * @param element
     */
    public void waitForElementToBeClickable(final WebElement
element, final WebDriver driver) {
        new WebDriverWait(driver, this.explicitWaitDefault)

.until(ExpectedConditions.elementToBeClickable(element));
    }

    /**
     * To wait for element (By) to be invisible
     *
     * @param driver
     * @param locator
     */
    public void waitForElementToBeInvisible(final By locator,
final WebDriver driver) {
        long s = System.currentTimeMillis();
        new WebDriverWait(driver, this.explicitWaitDefault)

.until(ExpectedConditions.invisibilityOfElementLocated(locato
r));
    }

    /**
     * To wait for given element (By) to be present
     *
     * @param driver
     * @param locator
     */
    public void waitForElementToBePresent(final By locator,
final WebDriver driver) {
        new WebDriverWait(driver, this.explicitWaitDefault)

.until(ExpectedConditions.presenceOfElementLocated(locator));
    }

    /**
```

```
     * To wait for element (By) to be visible
     *
     * @param driver
     * @param locator
     */
    public void waitForElementToBeVisible(final By locator,
final WebDriver driver) {
        new WebDriverWait(driver, this.explicitWaitDefault)

.until(ExpectedConditions.visibilityOfElementLocated(locator)
);
    }

    /**
     * To wait for element to be visible
     *
     * @param driver
     * @param element
     */
    public void waitForElementToBeVisible(final WebElement
element, final WebDriver driver) {
        long s = System.currentTimeMillis();
        new WebDriverWait(driver,
this.explicitWaitDefault).until(ExpectedConditions.visibility
Of(element));
    }

    /**
     * To wait for element to be visible for given amount of
time
     *
     * @param element
     * @param driver
     * @param time
     */
    public void waitForElementToBeVisible(final IOSElement
element, final WebDriver driver, int time) {
        long s = System.currentTimeMillis();
        new WebDriverWait(driver,
time).until(ExpectedConditions.visibilityOf(element));
    }
```

```
    public void waitForElementsToBeInvisible(final
List<WebElement> elements, final WebDriver driver) {
        final long s = System.currentTimeMillis();
        new WebDriverWait(driver, this.explicitWaitDefault)

.until(ExpectedConditions.invisibilityOfAllElements(elements)
);
    }

    public void waitForElementToBeNotPresent(final By
element, WebDriver driver) {
        long s = System.currentTimeMillis();
        new WebDriverWait(driver, this.explicitWaitDefault)

.until(ExpectedConditions.not(ExpectedConditions.presenceOfAl
lElementsLocatedBy(element)));
    }

    public void waitUntilNestedElementPresent(WebElement
element, By locator, WebDriver driver) {
        new WebDriverWait(driver, explicitWaitDefault)

.until(ExpectedConditions.presenceOfNestedElementLocatedBy(el
ement, locator));
    }
}
```

- Now add PropertyUtils Class under utils directory, which will be responsible to get the property values from ***resources/configuration.properties*** file.

PropertyUtils.java

```
package utils;

import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;

/**
```

```
 * This class is used to get the configuration properties
from the .properties file
 */
public class PropertyUtils {

    private static PropertyUtils INSTANCE = null;
    private final Properties props = new Properties();

    private PropertyUtils() {
        this.loadProperties("configuration.properties");
        this.props.putAll(System.getProperties());
    }

    private static PropertyUtils getInstance() {
        if (PropertyUtils.INSTANCE == null) {
            PropertyUtils.INSTANCE = new PropertyUtils();
        }
        return PropertyUtils.INSTANCE;
    }

    /**
     * This method can read Property value for any given key
     *
     * @param key
     * @return
     */
    public static String getProperty(final String key) {
        return
PropertyUtils.getInstance().props.getProperty(key);
    }

    /**
     * This method will read any integer property value
     *
     * @param key
     * @param defaultValue
     * @return
     */
    public static int getIntegerProperty(final String key,
final int defaultValue) {
```

```
        int integerValue = 0;
        final String value =
PropertyUtils.getInstance().props.getProperty(key);
        if (value == null) {
            return defaultValue;
        }
        integerValue = Integer.parseInt(value);
        return integerValue;
    }

    /**
     * If key couldn't be found then it will return default
value
     *
     * @param key
     * @param defaultValue
     * @return
     */
    public static String getProperty(final String key, final
String defaultValue) {
        return
PropertyUtils.getInstance().props.getProperty(key,
defaultValue);
    }

    /**
     * This method will load properties file in Properties
object
     *
     * @param path
     */
    public void loadProperties(final String path) {
        InputStream inputStream = null;
        try {
            inputStream =
ClassLoader.getSystemResourceAsStream(path);
            System.out.println(inputStream);
            if (inputStream != null) {
                this.props.load(inputStream);
            } else {
```

```
                throw new
UnableToLoadPropertiesException("property file '" + path + "'
not found in the classpath");
            }
        } catch (final Exception e) {
            e.printStackTrace();
        } finally {
            try {
                inputStream.close();
            } catch (final IOException e) {
                e.printStackTrace();
            }
        }
        return;
    }

    /**
     * @return Properties
     */
    public static Properties getProps() {
        return PropertyUtils.getInstance().props;
    }

}

class UnableToLoadPropertiesException extends
RuntimeException {

    UnableToLoadPropertiesException(final String s) {
        super(s);
    }

    public UnableToLoadPropertiesException(final String
string, final Exception ex) {
        super(string, ex);
    }
}
```

12) Now the most important step is to create our BaseTestCase class and include the logic of WebDriver creation which would be the responsible to manage the WebDriver object throughout the automation project.

- Create a BaseTestCase class file under the testcases package.

- Add **TestNG(Test Framework for Java)** default methods such as ***@BeforeSuite, @BeforeClass, @BeforeTest, @AfterClass*** and ***@AfterTest*** methods.

  The ***@BeforeMethod*** will be executed every time before test case starts - Let's look at some simple code regarding its usage.

```
public class TestCases {

    @BeforeMethod
    public void setUp() {
        System.out.println("Before Method executed..!");
    }

    @Test
    public void test() {
        System.out.println("Test");
    }

    @AfterMethod
    public void tearDown() {
        System.out.println("After Method executed..!");
    }
}
```

It's pretty straight-forward code - the ***@Test*** method is the actual testing method you will use for your test cases.

@***BeforeMethod*** will be executed before the execution of your ***@test*** method and likewise @***AfterMethod*** will be executed after the test execution **everytime.** It does not matter if you are calling these methods or not.

So when you execute the above test by *selecting the* `test` *method > Right Click > Run 'test()'* it will execute the setUp() method before test executes → test() method → tearDown() method after the test executes.

Have a look at this screenshot to see how to execute the test case and refer to the second screenshot for the output:

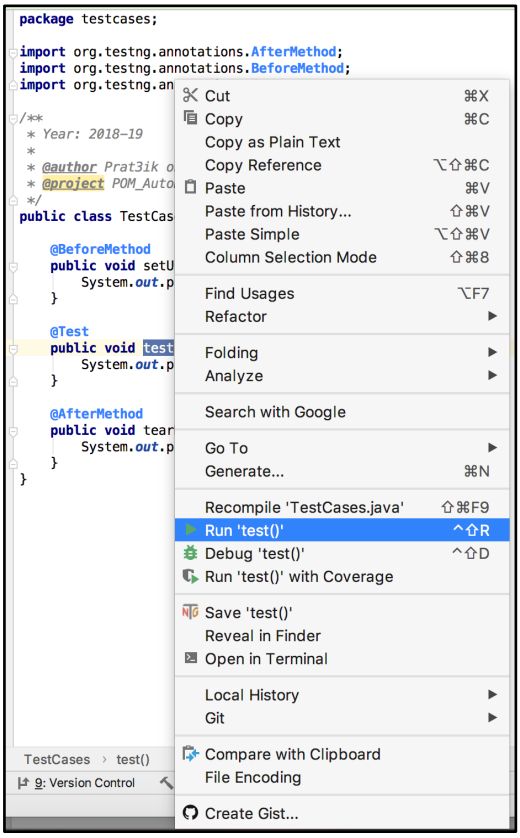

Figure-15: Execute the TestNG test..

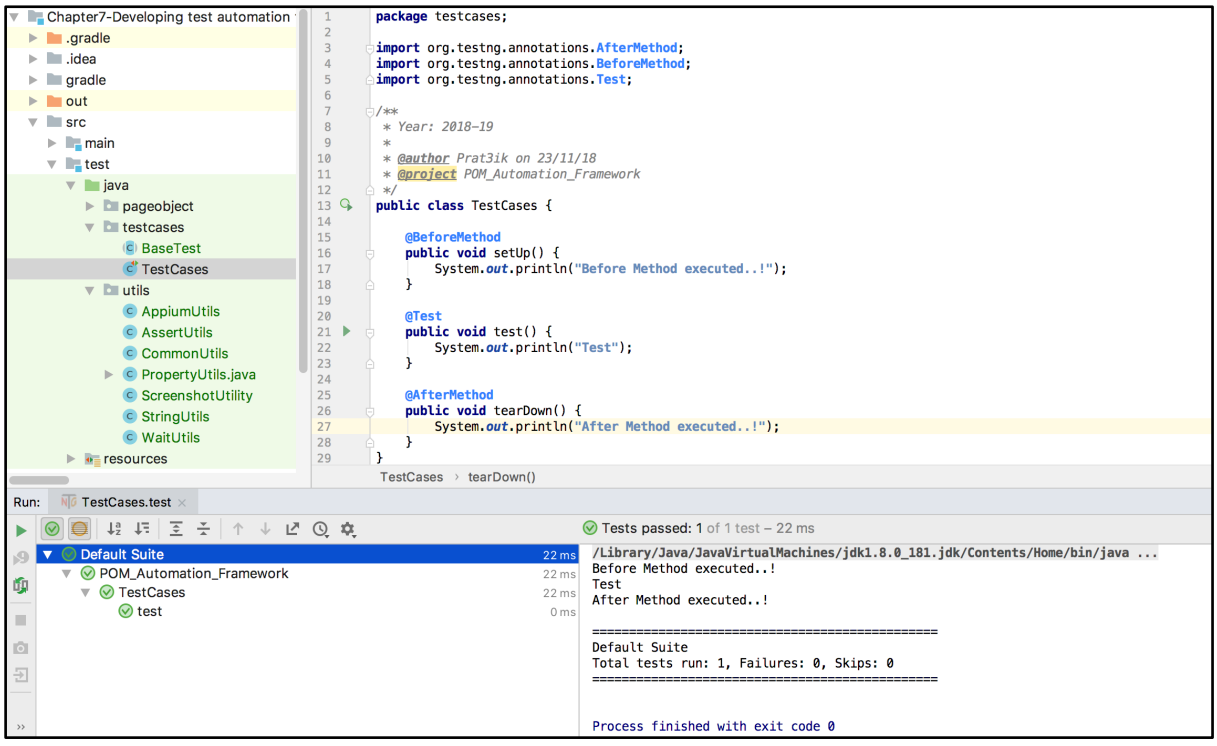

Figure-16: Execution of Simple Program.

You can learn more about TestNG Annotations here: https://www.tutorialspoint.com/testng/testng_basic_annotations.htm

- The idea is to put the code for creation of the WebDriver object inside the ***@BeforeMethod***, because we want the webdriver object in place before starting the test case(method).

```
@BeforeMethod
public void setUpAppium() throws MalformedURLException {
    DesiredCapabilities capabilities = new
DesiredCapabilities();
    setDesiredCapabilitiesForAndroid(capabilities);
    driver = new AppiumDriver(new URL(APPIUM_SERVER_URL),
capabilities);
}


/**
 * It will set the DesiredCapabilities for the local
execution
 *
 * @param desiredCapabilities
 */
private void
setDesiredCapabilitiesForAndroid(DesiredCapabilities
desiredCapabilities) {
String PLATFORM_NAME = PropertyUtils.getProperty("android.plat
String PLATFORM_VERSION = PropertyUtils.getProperty("android.pl
    String APP_NAME = PropertyUtils.getProperty("android.app.n
String APP_RELATIVE_PATH = PropertyUtils.getProperty("android.a
               String APP_PATH = getAbsolutePath(APP_RELATIVE_
String DEVICE_NAME = PropertyUtils.getProperty("android.device
String APP_PACKAGE_NAME = PropertyUtils.getProperty("android.ap
String APP_ACTIVITY_NAME = PropertyUtils.getProperty("android.a
String APP_FULL_RESET = PropertyUtils.getProperty("android.app
String APP_NO_RESET = PropertyUtils.getProperty("android.app.no

desiredCapabilities.setCapability(MobileCapabilityType.AUTOMATI
```

```
desiredCapabilities.setCapability(MobileCapabilityType.DEVICE_N
desiredCapabilities.setCapability(MobileCapabilityType.PLATFORN
desiredCapabilities.setCapability(MobileCapabilityType.PLATFORN
desiredCapabilities.setCapability(MobileCapabilityType.APP, APP
desiredCapabilities.setCapability(AndroidMobileCapabilityType.A
desiredCapabilities.setCapability(AndroidMobileCapabilityType.A
desiredCapabilities.setCapability(MobileCapabilityType.FULL_RES
desiredCapabilities.setCapability(MobileCapabilityType.NO_RESET
desiredCapabilities.setCapability(AndroidMobileCapabilityType.A
}

// To get Absolute Path from Relative Path
private      static      String      getAbsolutePath(String
appRelativePath){
    File file = new File(appRelativePath);
    return file.getAbsolutePath();
}

/**
* This will quite the android driver instance
*/
private void quitDriver() {
    try {
                                                this.driver.q
                                            } catch (Exceptio



}
```

This code illustrates how you can leverage the ***@BeforeMethod and @AfterMethod*** to leverage WebDriver object creation and deletion.

Every Test Case class file will extend this BaseTest Class file, so all the methods in the BaseTest class file are available to your test cases. You just have to focus on Test methods creation on TestCases.java file.

13) Add a ***BasePO*** Class under the ***pageobject*** package.

- BasePO is the class containing the PageFactory method.

- **What is Page Factory?:**
  - It is an inbuilt Page Object Model concept for Selenium WebDriver and it is used to initialize the web elements (Utilizing the concept of lazy loading: Initialize elements only when they are needed to be used) that are defined in Page Objects.

- Below code is responsible to initialize the web elements:

```
private void initElements() {
   PageFactory.initElements(new AppiumFieldDecorator(driver,
Duration.ofSeconds(IMPLICIT_WAIT)), this);
}
```

- Now every other Page Object class like LoginPO will extend the BasePO class, so the constructor of BasePO is always called first and the `initElements` method will be called on BasePO constructor. In simple language `initElements` will always called first whenever any Page Object class gets called.

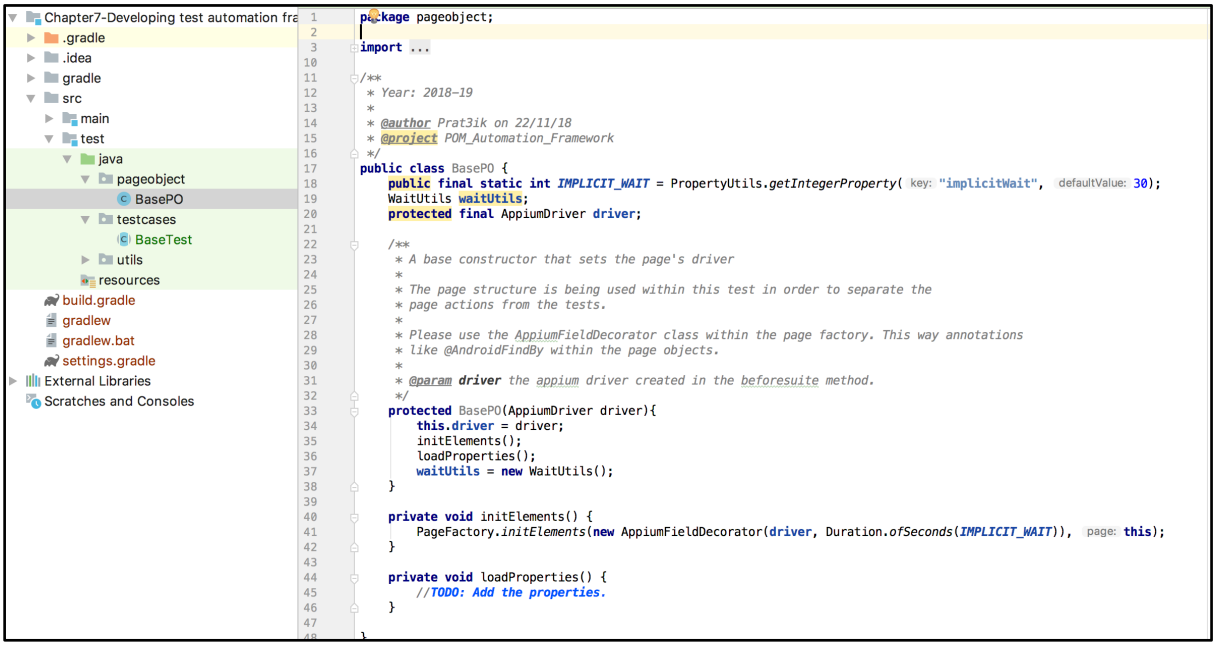

Figure-17: BasePO.

Here you can also see some objects and variables defined. `IMPLICIT_WAIT` is getting the values defined in properties file of java which will be stored under ***resources/*** dir.

- In the Page Object class you need to define the element's locators using the below approach:

For iOS:

```
@iOSFindBy(xpath = "//XCUIElementTypeTextField")
IOSElement emailTextField;
```

For Android:

```
@AndroidFindBy(xpath =
"//android.widget.TextView[@text='Login Screen']")
AndroidElement loginScreenTextView;
```

- In this example we will work with the sample Android application and we will use the UiAutomatorViewer inspection tool as it is a speedy way to get the element's locator.

Here we get the locator for ***Login Screen*** Textview:

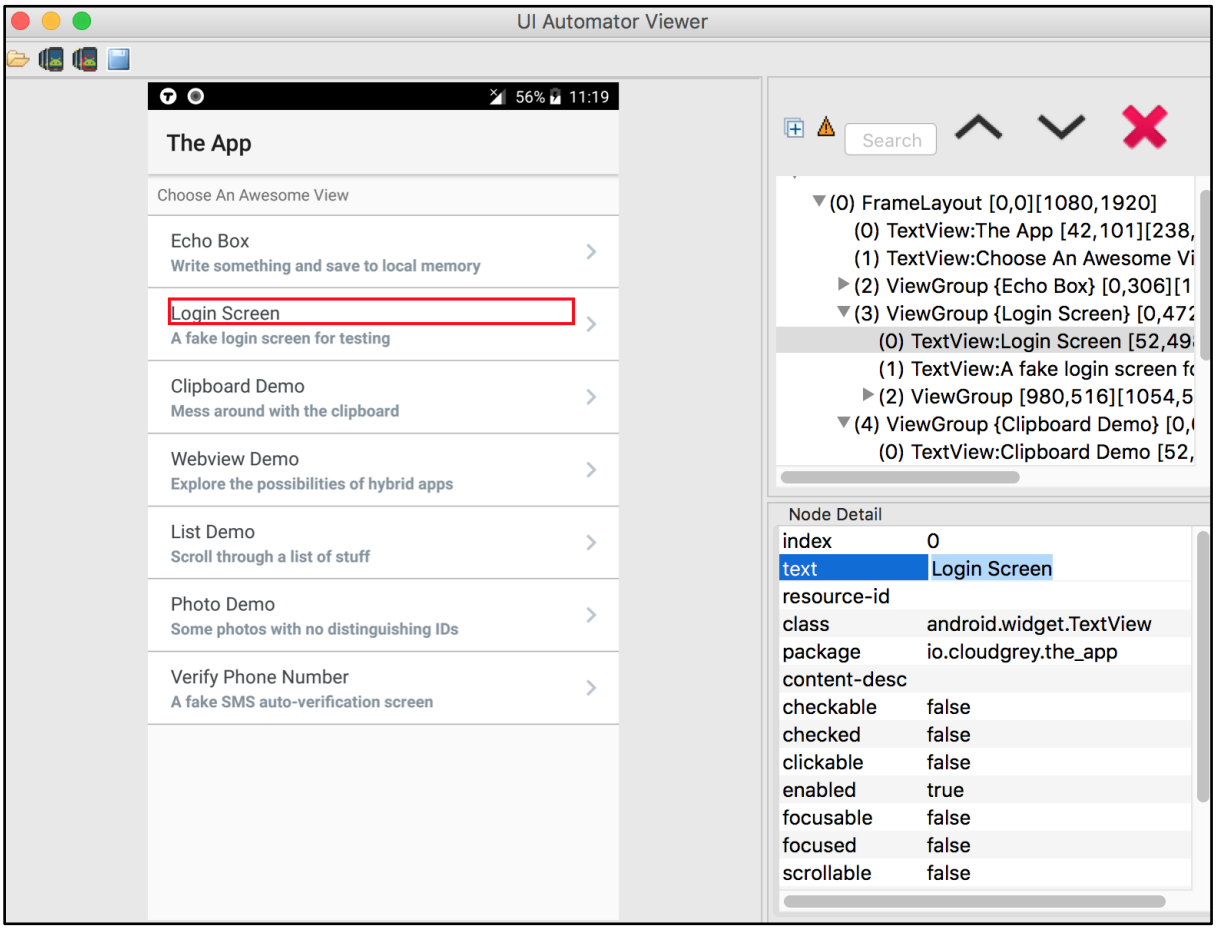

Figure-18: UiAutomatorViewer for Android Application.

- And you can create methods in the Page Object class(like in LoginPO) for the element locators for example if you want to Tap on Login TextView then you can create method such as tapOnLoginScreenTextView().

HomeScreenPO.java

```
package pageobject;
import io.appium.java_client.AppiumDriver;
import io.appium.java_client.android.AndroidElement;
import io.appium.java_client.pagefactory.AndroidFindBy;

public class HomeScreenPO extends BasePO {

    public HomeScreenPO(AppiumDriver driver) {
        super(driver);
    }

    @AndroidFindBy(xpath =
"//android.widget.TextView[@text='Login Screen']")
    AndroidElement loginScreenTextView;

    /**
     * This method will click on Login Screen textview.
     */
    public void tapOnLoginScreenTextView(){
        loginScreenTextView.click();
    }
}
```

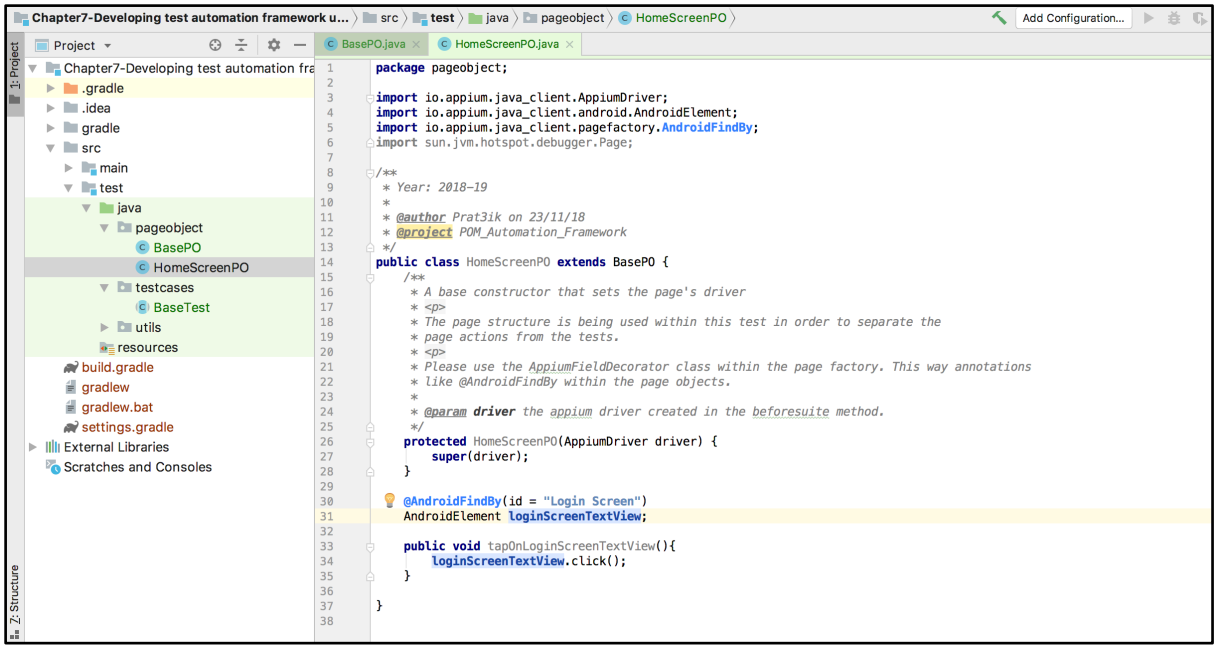

Figure-19: HomeScreenPO.

14) Now that we have created our first Page Object class and added our first locator into it, we are ready to create a very basic simple test on ***TestCase***

15) Before creating the test case you need to provide the correct path to the application. You can either:

1) Provide the application locally in the code.

```
desiredCapabilities.setCapability(MobileCapabilityType.APP
, "path/to/.apk or .ipa(or .app) file");
```

2) Provide the URL of the application.

```
desiredCapabilities.setCapability(MobileCapabilityType.APP
, "https://github.com/cloudgrey-io/the-
app/releases/download/v1.7.0/TheApp-v1.7.0.apk");
```

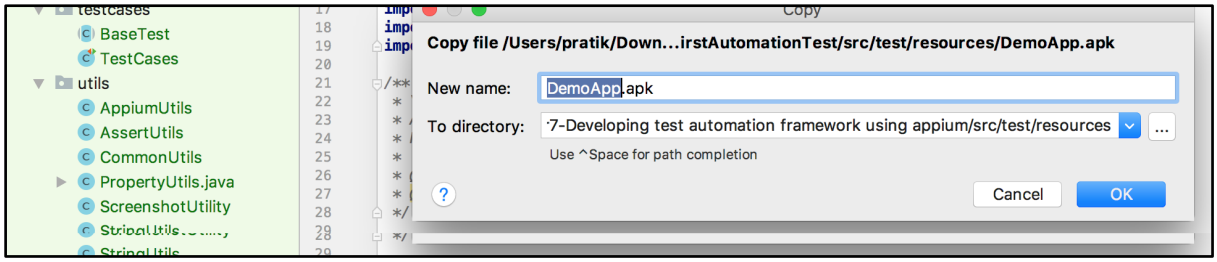

Figure-20: Local .apk file.

In our example we will use the local .apk file.

16) Now finally it's time to create the ***TestCase*** java file under the ***testcases*** package and it should extend the ***BaseTest*** java file which controls the webdriver creation before the test starts and deletion at the end of test execution. So you don't have to take care of that in the ***TestCase*** file.

NOTE: *Before executing the Test Case make sure Appium server is running on* ***http://127.0.0.1:4723/wd/hub***

TestCases.java

```
public class TestCases extends BaseTest{
    @Test
    public void test() {
        HomeScreenPO homeScreenPO = new HomeScreenPO(driver);
        homeScreenPO.tapOnLoginScreenTextView();
    }
    @BeforeTest
    @Override
    public void setUpPage() {}
}
```

As you can see here, our test case is not concerned with how to locate the element and how to perform the action. That is taken care of by our homeScreenPO object. Our test case can focus on the business logic or what we are actually testing.

If the element location changes, you can update the homeScreenPO object and your test case remains unaffected.

Execute the above test by *selecting the* `test` *method > Right Click > Run 'test()'*

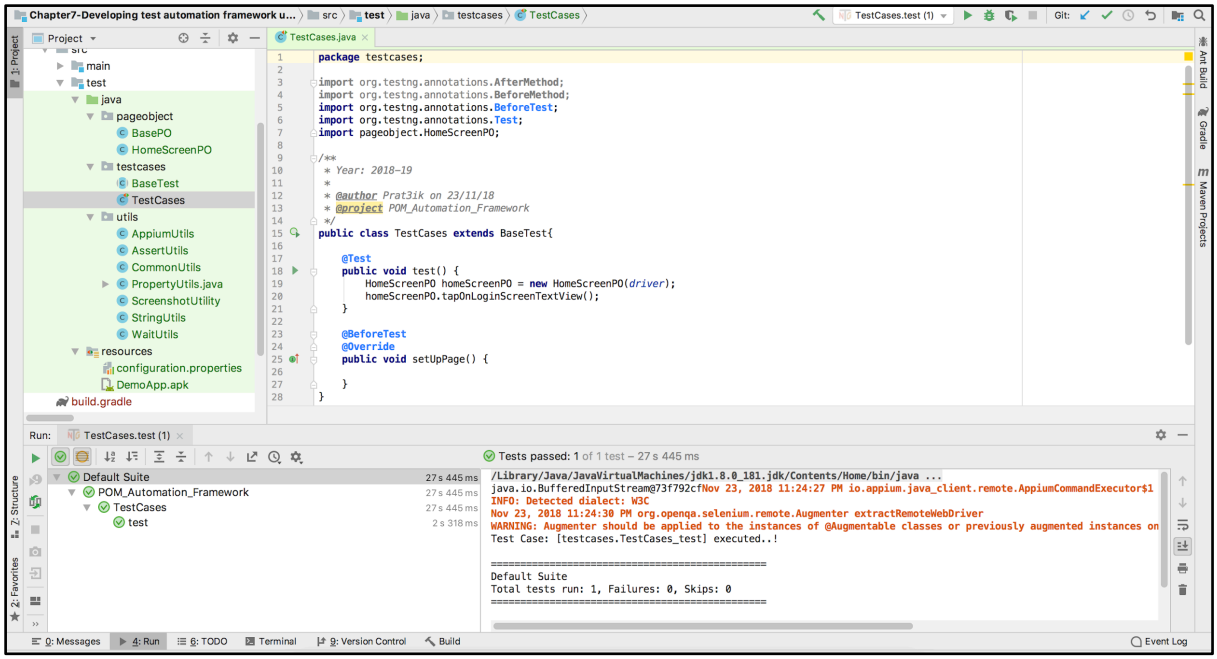

Figure-21: Test Case successful execution.

If you can execute the test case successfully then you will get the above screenshot.

You can get the code of above explained framework from our github project.

Phew! That was a ride! Right now you may be thinking that was a lot of work to create a simple test case - but that simple test case is misleading. You've done all the groundwork - adding more test cases reusing those elements are much quicker.

Fixing the locator when the application changes

Now the question is how can you benefit from all of this work you just did? Think about your next release when the element locator of the application changes.

Thanks to structuring your code properly, changes to elements just need to be relfected in the PageObject class. As easy as 1-2-3:

1) Open the particular page object class

Let say the locator of Login TextView changed in a new iteration(version) of the application and Login TextView is part of the Home Screen so you need to move to **HomeScreenPO.java** file.

2) Get the new locator

Using Appium Inspector, UiAutomator (Android) or Accessibility Inspector (iOS) you can get the Locator. So move to the particular screen on the application where the element is located and fetch the correct locator.

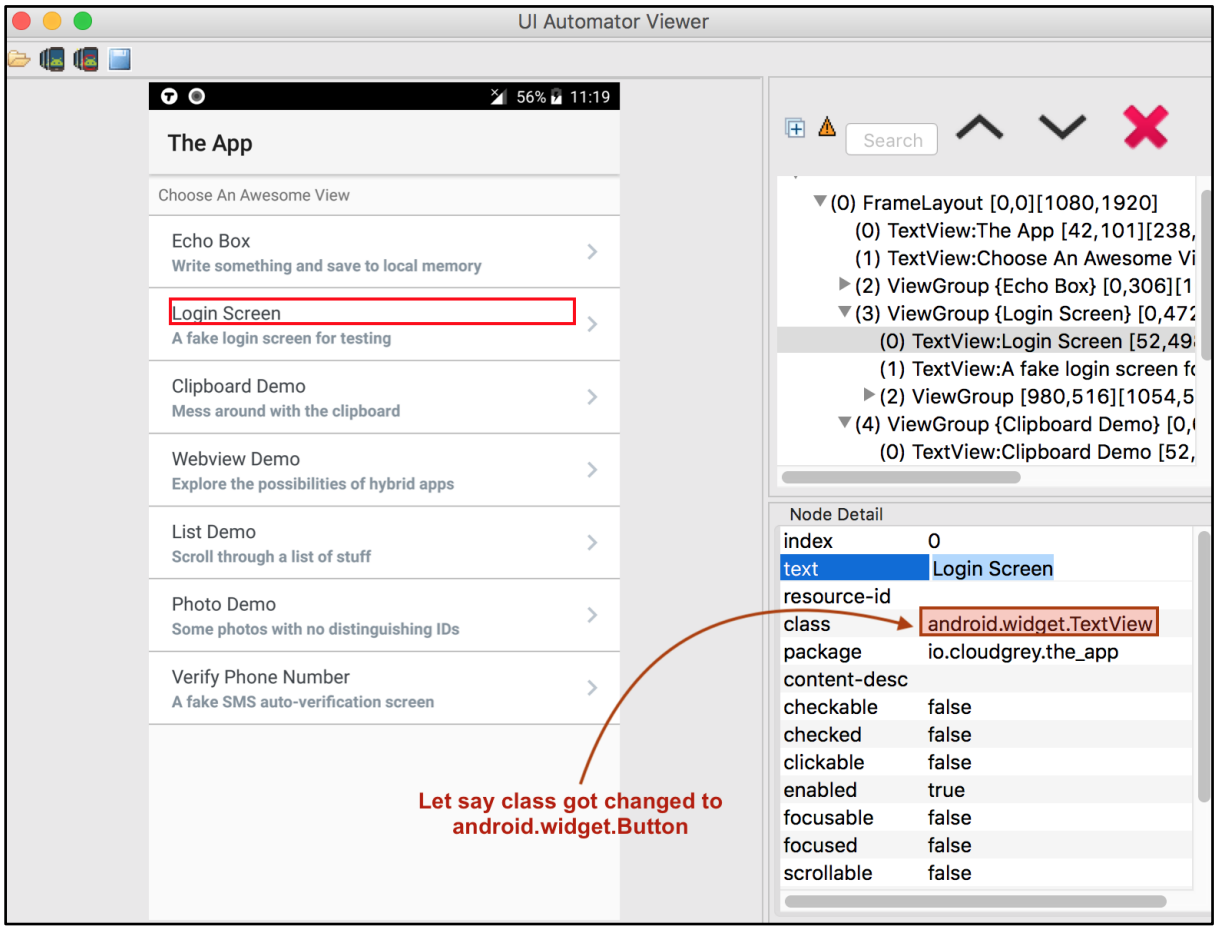

Figure-22: Android Application Locator Change.

3) Change the locator

After getting the right locator you just need to replace the old and incorrect locator. In **HomeScreenPO.java** file you just need to replace the old locator with the new locator.

```
@AndroidFindBy(xpath =
"//android.widget.Button[@text='Login Screen']")
AndroidElement loginScreenTextView;
```

That's it! As you can see we just need to change the locator in the ***Page Object class*** and everything works normally again - that is the beauty of using this framework! If you are not using the framework then you might need to change the locator from every affected place in code, which is not advisable and can also break something else.

Structuring your test cases like this will make for a far more maintainable test automation suite. Learning the rigors to do it following this design pattern will benefit you and your organization for years to come!

Chapter-8: Test Synchronization

In the last chapter we explored the Page Object Model in some detail. If you recall, we touched on the **wait** method and its significance and we promised to get back to it. So in this chapter we will understand how wait (or Synchronization) performs a vital role in Automation.

If two or more components are working together in parallel at the same pace or rate, synchronization comes into play.

We see it in almost every application whenever the screen changes it takes a few milliseconds (or seconds) to load, and if you do not manage the proper synchronization in your code then you might face the dreaded "ElementNotVisibleException" or "NoSuchElementException" exceptions. This is because the screen hasn't finished loading and is not synchronized with your test code. That is, your test code is over eager and starts trying to perform an action on an element that hasn't been loaded yet. To avoid this we need to implement proper synchronization in our automation script.

We can categorize **synchronization** in two types:

1) Unconditional,
2) Conditional.

Now let's discuss each of them.

1) Unconditional synchronization

Unconditional Synchronization is also known as ***Static Synchronization*** or ***Static Wait***.

As the name suggests, it specifies a particular fixed (static) time to wait before starting the execution. Here Appium(or any program) will wait the specified amount of time and then it will resume the execution.

The standard example of Unconditional Synchronization is below:

```
try {
                                                    Thread.sleep(
} catch (InterruptedException e) {
                                                  e.printStackTr
}
```

Here the `Thread.sleep(1000)` function would take 1000 ms to execute.
Key to note is that this wait will be absolute, even if the underlying condition you were waiting on has been met. For example, you may put in a wait for 3 seconds waiting for the screen to load. Even if that screen loads in 2 seconds, the system will still wait for the additional second. The converse is also true - Sometimes the wait finishes before the underlying operation and execution proceeds. In test automation, for example, limited network connectivity may slow the mobile application response time and a screen change may now take 5 seconds while the script only waits 3 seconds. Once again, you'll face "ElementNotVisibleException" or "NoSuchElementException" exceptions.

So unconditional synchronization or static wait is not the preferred way to deal with dynamic responses.

However it is a viable strategy to use it when you are working with some 3rd party interfaces and where you can not identify the underlying condition you need to wait on OR you are sure about the response time.

2) Conditional synchronization

Conditional synchronization depends on some underlying condition. So in addition to a specified absolute time to wait, the condition is also passed into the method. Here the script(or program) will resume execution as soon as the condition is met - or, in the event the condition isn't met, it will resume after the specified time.

Appium (or Selenium) provides 3 (mainly 2) types of conditional synchronization.

1) **Implicit wait**
2) **Explicit wait**
3) **Fluent wait**

1) Implicit wait

Implicit wait tells the Appium's webdriver object to poll the DOM for the specified amount of time while trying to find the element before throwing an "ElementNotVisibleException" or "NoSuchElementException" exceptions.

The big advantage of using Implicit wait is it's lifespan. As we apply Implicit wait on the Webdriver object, it will be valid for the webdriver object's lifespan.

Below is the code to apply the Implicit wait of 10 seconds on the Webdriver object.

```
// Define AppiumDriver(WebDriver)
AppiumDriver driver = new AppiumDriver(new
URL(APPIUM_SERVER_URL), capabilities);

// Set Implicit wait upon AppiumDriver(WebDriver)
driver.manage().timeouts().implicitlyWait(10,
TimeUnit.SECONDS);
```

NOTE: ***Ideally you should set the implicit wait as soon as you initialize the WebDriver.***

Also you can use other Time Units such as
`TimeUnit.NANOSECONDS`
`TimeUnit.MICROSECONDS`
`TimeUnit.MILLISECONDS`

Please remember that Implicit wait works with only `driver.findElement(...)` and `driver.findElements(...)` methods - it won't work for other methods.

Let's look at a simple example for better understanding.

You want to click on the Login button on the home screen but the home screen itself takes some to appear when the Appium script runs.

So ideally you need to specify a condition that Appium should locate the Login Button element(on the home screen) within 10 seconds after starting the script and if the element is not present (or the home screen has not appeared) after 10 seconds, only then throw Exceptions.

You can write the following statement after writing the above (webdriver initialization and set implicit wait of 10 sec.) code:

```
// Click on Login Button from Home screen.
driver.findElement(By.id("login")).click();
```

You will notice that we don't need to specify anything else to the code as we already added the 10 seconds implicit wait to the AppiumDriver object. Now

it will be polling the DOM for 10 seconds until the Login Button is found and as soon as the button is found, it will be clicked.

However there are a few limitations to using Implicit wait:

1) As you know it's only useful for `driver.findElement(...)` and `driver.findElements(...)` methods - we can't check other conditions. For example if you want to wait until a particular button is displayed on the screen as well as on the DOM, you can't check it with Implicit wait. There is a chance that the particular button is in the DOM but it's hidden or it's not visible on the screen, so in that case Implicit wait executes successfully but would not give us the accurate answer about the element's visibility.

2) There is a chance that the time for the implicit wait isn't enough. For example as we mentioned earlier where we're waiting 5 seconds but limited network connectivity causes the screen to take 10 seconds to load. In that case Implicit wait will break.

These limitations are resolved by the Explicit wait.

2) Explicit wait

Explicit waits are the best synchronization methods for dynamic responses in the application.

Explicit wait informs the AppiumDriver(WebDriver) to wait

1) Until the specified condition is met OR
2) The specified time has elapsed

...before throwing the "ElementNotVisibleException" or "NoSuchElementException" exceptions.

And if the AppiumDriver is able to meet the condition within the specified amount of time then the code will get executed.

In explicit wait we need to tell the WebDriver object to wait for a specific condition using the `ExpectedConditions` class. So, actually this wait is specific to a particular single element rather than the whole WebDriver object (unlike implicit wait).

The `WebDriverWait` class will call the **ExpectedCondition** every 500 milliseconds by default until the output is True. So if you have given 10 seconds of timeout, **ExpectedCondition** would be called 20 times at 500 milliseconds intervals to check if the condition has been met.

Now let's take one simple example to understand the use of Explicit wait.

In almost all mobile applications when you perform a Login it takes some time to load the dashboard or home screen and its elements. For example purposes, let's say there is a menu button on the dashboard screen. You can use an Explicit wait with the condition of wait till menu button element is visible: (`ExpectedConditions.visibilityOf(<menu_button_element>)`) on Dashboard screen.

```
WebDriverWait webDriverWait = new WebDriverWait(driver,
30);
webDriverWait.until(ExpectedConditions.visibilityOfElement
Located(By.id("menubutton")));
```

When we initialize new object of `WebDriverWait` class, we need to pass 2 parameters:

1) WebDriver object.
2) Number of seconds

After creation of the WebDriver object you need to call the `until()` method and need to pass the `ExpectedConditions.<condition_name>()` inside it.

There are many conditions are defined in `ExpectedConditions` class, but we can list down a few popular ones[3]:

Condition Name	Purpose
elementToBeClickable(By locator) **Example:**	An expectation for checking an element is visible and enabled such that you can click it. In this method

[3] *Official Selenium API docs on the Github project.*

ExpectedConditions.elementToBeClickable(By.id("loginButton"));	you need to pass the object of `By` class.
elementToBeClickable(WebElement element) **Example:** ExpectedConditions.elementToBeClickable(driver.findElement(By.id("menubutton")));	An expectation for checking an element is visible and enabled such that you can click it. In this method you need to pass the object of `WebElement` class.
presenceOfElementLocated(By locator)	An expectation for checking that an element is present on the DOM of a page. This does not necessarily mean that the element is visible.
visibilityOfElementLocated(By locator)	An expectation for checking that an element is either invisible or not present on the DOM.
elementToBeSelected(WebElement element)	An expectation for checking if the given element is selected.
numberOfElementsToBe(By locator, java.lang.Integer number)	An expectation for checking number of WebElements with given locator.
titleIs(java.lang.String title)	An expectation for checking the title of a page. ***This is not applicable to Appium Mobile Application.***
textToBePresentInElement(WebElement element, java.lang.String text)	An expectation for checking if the given text is present in the specified element.

You can find more details about `ExpectedConditions` class and it's methods on Official Selenium docs, but remember not all are applicable to Mobile Applications, however all the methods are valid for Mobile Web Browser(Chrome/Safari):

3) Fluent wait

Fluent wait is part of WebDriverWait, The only difference is it's more configurable than Explicit wait.

You can configure the:

1) **Poll frequency:** The is the Time Interval to check whether the expected condition for the webelement is met or not. So if poll frequency is 1 second and total wait time is 10 seconds, fluent will check if the condition is met or not at every 1 second for a maximum of 10 times.

2) **Ignore the Exception:** If you want to ignore a specific exception such as ***NoSuchElementExceptions*** while searching for an element.

3) **Maximum wait time:** The total maximum amount of time to wait for a condition is met before throwing an exception.

Below is the example of Fluent wait:

```
FluentWait<AppiumDriver> webDriverWait = new
FluentWait<AppiumDriver>(driver);

webDriverWait.pollingEvery(Duration.ofSeconds(1));
webDriverWait.ignoring(NoSuchElementException.class);
webDriverWait.withTimeout(Duration.ofSeconds(10));
```

Which types of Wait you should use When?

Wait Type	Purpose
Implicit	When you need to apply common wait without any condition.
Explicit	When you need to test expected condition for an element.
Fluent	When you need to test expected condition for an element after a specific amount of time like every x seconds/minutes.

Synchronization in our automation framework (WaitUtils.java)

In Chapter-7, we defined the Wait Utility in order to handle synchronization in our tests. In this section we will look into the usage of it.

We created the WaitUtils object at 2 places to leverage the synchronization at Page Object or Test Class level:

1) **BasePO.java**

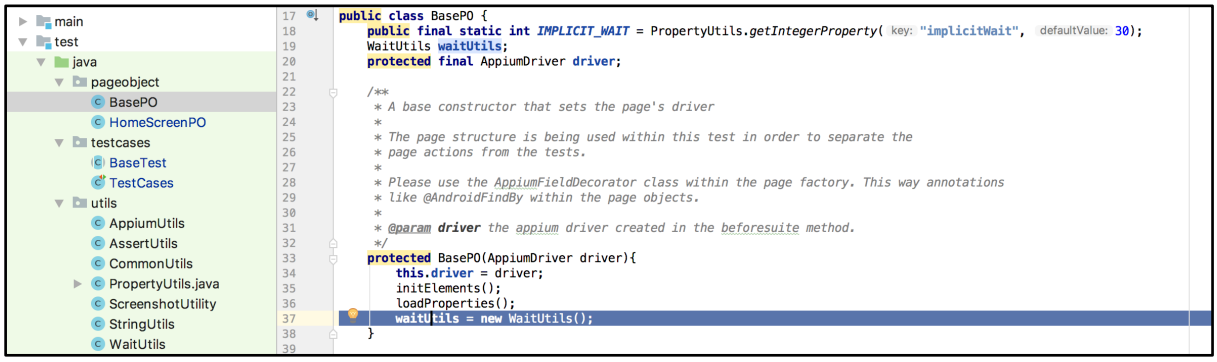

Figure-1:WaitUtils object in BasePO.

2) **BaseTest.java**

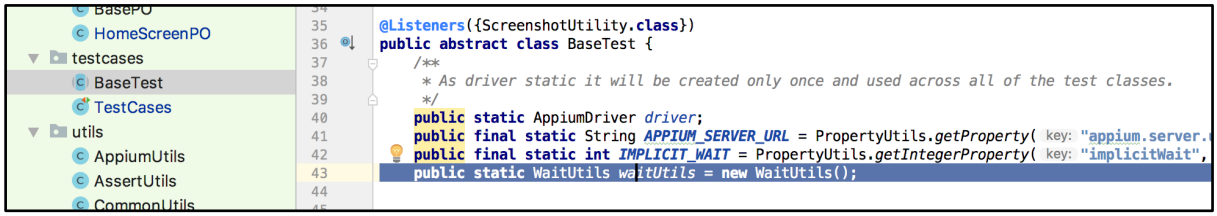

Figure-2: WaitUtils object in BaseTest.

You can see there are many wait methods(of Implicit and Explicit wait) are defined in **WaitUtils.java** file, Now let's create a simple example to use WaitUtils using the WaitUtils object at the Page Object and Test Class level.

WaitUtils.java

```
public class WaitUtils {
……
……
    public void staticWait(final long millis) {
        try {
            TimeUnit.MILLISECONDS.sleep(millis);
        } catch (final InterruptedException e) {
        }
    }
……
……
}
```

1) WaitUtils usage on TestCases.java

Here the script is using the `staticWait(long milliSeconds)` method, so after tapping on the Login Screen Text View, the script will pause execution for 2 seconds.

```
public class TestCases extends BaseTest{

    @Test
    public void test() {
        HomeScreenPO homeScreenPO = new
HomeScreenPO(driver);
        homeScreenPO.tapOnLoginScreenTextView();
        waitUtils.staticWait(2000);
    }
…….
…….
}
```

2) WaitUtils usage on .java

Here the script is using the same `staticWait(long milliSeconds)` method of **WaitUtils.java** class. In **HomeScreenPO.java** we have defined the `tapOnLoginScreenTextView()` method which clicks on the Login TextView - after tapping on Login Screen Text View, execution would pause for 2 seconds.

```
public class HomeScreenPO extends BasePO {
…….
…….
    /**
     * This method will click on Login Screen textview.
     */
    public void tapOnLoginScreenTextView(){
        loginScreenTextView.click();
        waitUtils.staticWait(3000);
    }
…….
…….
}
```

So both of them doing the same work but at different places!

Many beginner Appium developers sometimes wonder when to wait and for how long. Using wait in your scripts will become intuitive the more you use them and the more you run into certain conditions. And don’t worry, you’ll be greeted with exceptions when you forget to include the proper wait!

Chapter-9: Parallel Test Execution on Simulators and Emulators.

Up until now we've learned all the Appium basics, including how to extract elements and executing the tests on devices and emulators.

Although automation is great, in today's fast moving world there is a constant demand to execute tests faster and on more devices. So the idea of executing these tests in a linear fashion (ie. one at a time) seems somewhat antiquated.

In this chapter we are going to discuss how you can leverage parallel test execution on simulators and emulators to test more, faster.

We will be using **Java + TestNG,** a great combination in the world of Automation Testing allowing us to run tests on parallel.

Before going too deep into parallelization details we will go through the basics of TestNG.

TestNG

TestNG is a open source testing framework written in **Java,** featuring:

- Annotations support, which reduces complexity at the class level.
- It is written in Java and has clean Object Oriented features.
- All tests can be run from one place via the ***testng.xml*** file that specifies the tests to be executed.
- It has a grouping feature so you can group the test cases and can run them group wise.
- Supports multi threading and parallel testing at the 1) method(test), 2) class and 3) test suite level.
- It also supports Data Providers so code duplication can be reduced.

For example, if you want to execute the same test cases having different data each time, you don't need to create separate test cases with different data.

In this chapter we will focus on the Parallel testing feature.

Let's take a refresher on how you can execute test cases from a Test Class file in IntelliJ Idea. You will recall that you just need to select the ***Test Method Name > Right click on it > Run.***

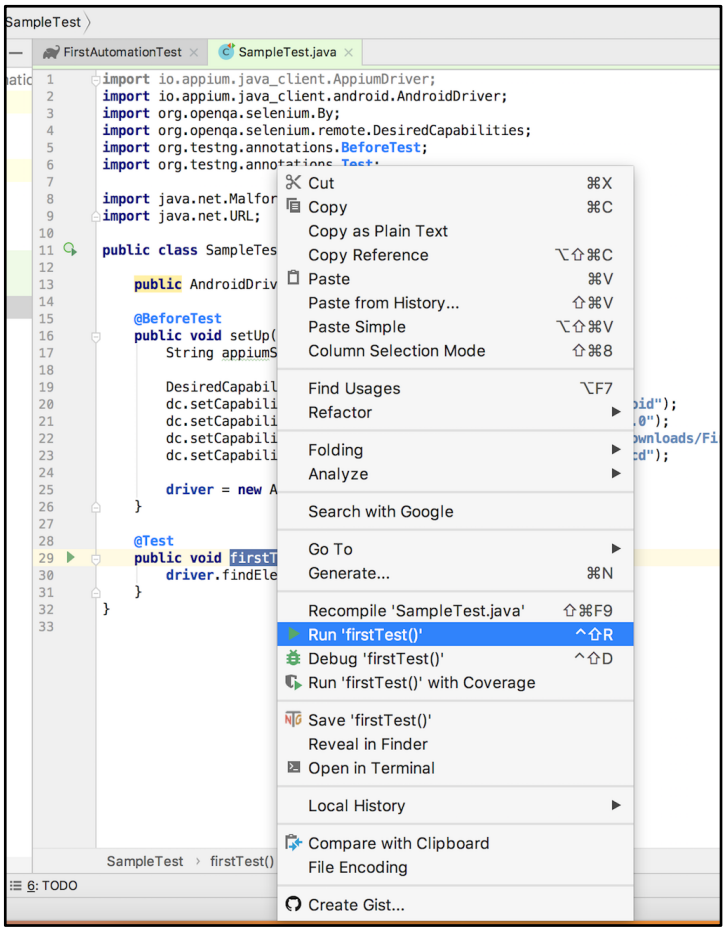

Figure-1: Run the test case.

Now before jumping into parallel testing, let's revisit the following simple code for iOS we used in an earlier chapter. This is the code we will be using as an example for parallel execution:

```
import io.appium.java_client.ios.IOSDriver;
import io.appium.java_client.ios.IOSElement;
import org.openqa.selenium.By;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.testng.Assert;
import org.testng.annotations.BeforeTest;
```

```
import org.testng.annotations.Test;

import java.net.MalformedURLException;
import java.net.URL;

public class IOSTestCases {
    public IOSDriver<IOSElement> driver;

    @BeforeTest
    public void setUp() throws MalformedURLException {
        String appiumServerURL =
"http://127.0.0.1:4723/wd/hub";

        DesiredCapabilities dc = new
DesiredCapabilities();
        dc.setCapability("platformName", "iOS");
        dc.setCapability("platformVersion", "11.4");
        dc.setCapability("app",
"/Users/pratik/Downloads/FirstAutomationTest/src/test/reso
urces/DemoApp-iPhoneSimulator.app");
        dc.setCapability("deviceName", "iPhone X");

        driver = new IOSDriver<IOSElement>(new
URL(appiumServerURL), dc);
    }

    @Test
    public void sampleTestCase() throws
InterruptedException {
        int a = 5;
        int b = 10;

        driver.findElement(By.id("IntegerA")).sendKeys(a +
"");
        driver.findElement(By.id("IntegerB")).sendKeys(b +
"");

driver.findElement(By.id("ComputeSumButton")).click();
        String answer =
driver.findElement(By.id("Answer")).getText();
        Assert.assertEquals(answer, a + b + "", "Expected
```

```
and Actual Result didn't match!");
    }
}
```

After putting the above test script into a IntelliJ java project, we can execute the test cases. But right now we will use the **testng.xml method** for execution.

Creation of testng.xml

Unfortunately, by default **testng.xml** is not available to the project so we need to create it. But you can do it in 2 ways:

1) Manually create testng.xml

In this approach, You have to create the testng.xml file manually in project. You can

create/configure the testng.xml file by many ways such as:

a) Specifying the Package name, to execute tests from whole package.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="All Test Suite">
    <test verbose="2" preserve-order="true"
          name="/Volumes/Disk2/AppiumBook/Chapter9-Test
Execution on Parallel simulators and emulators">
            <packages>
              <package name="testcases" />
            </packages>
    </test>
</suite>
```

b) Specifying the Test Class name, to execute tests for particular Test Class.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="All Test Suite">
    <test verbose="2" preserve-order="true"
          name="/Volumes/Disk2/AppiumBook/Chapter9-Test
Execution on Parallel simulators and emulators">
        <classes>
            <class name="testcases.IOSTestCases">
```

```
            </class>
        </classes>
    </test>
</suite>
```

c) Specifying the Test Method name, to execute a particular test case.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="All Test Suite">
    <test verbose="2" preserve-order="true"
          name="/Volumes/Disk2/AppiumBook/Chapter9-Test
Execution on Parallel simulators and emulators">
        <classes>
            <class name="testcases.IOSTestCases">
                <methods>
                    <include name="sampleTestCase"/>
                </methods>
            </class>
        </classes>
    </test>
</suite>
```

d) Specifying the specific groups to be included or excluded. Here you need to set the group to Test case first like:

```
@Test(groups = {"sample"})
public void sampleTestCase() {
….
….
}
```

And after that you can create a testng.xml file to run the test cases for a **sample** group.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="All Test Suite">
    <test verbose="2" preserve-order="true"
          name="/Volumes/Disk2/AppiumBook/Chapter9-Test
Execution on Parallel simulators and emulators">
```

```
        <groups>
            <run>
                <include name="sample"/>
            </run>
        </groups>

        <classes>
            <class name="testcases.IOSTestCases"/>
        </classes>
    </test>
</suite>
```

That may seem a little confusing right now since you don't yet know the details of

what all of that does. Hang in there, it'll start making sense in a bit.

2) Manually create testng.xml

This is the recommended approach in IntelliJ Idea, as you can create **testng.xml** using an IntelliJ Idea plugin. For other IDEs, consult their plug-in marketplace or ecosystem.

Creating it via IntelliJ involves the following steps:

- Move to IntelliJ Idea project and open **Preferences.**

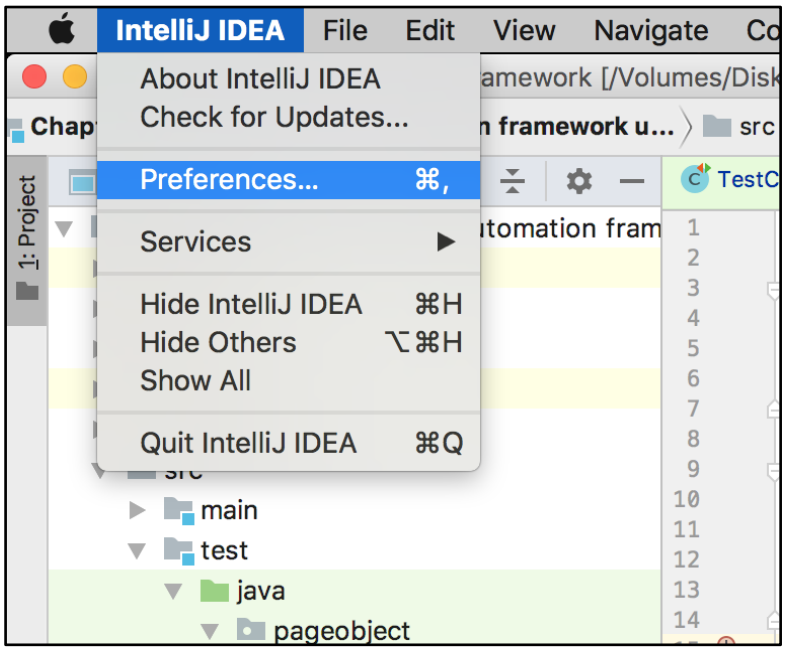

Figure-2: Open Preferences.

- Now move to **plugin** section and click on **Browse repositories…** which will takes you to the Plugins dialog.

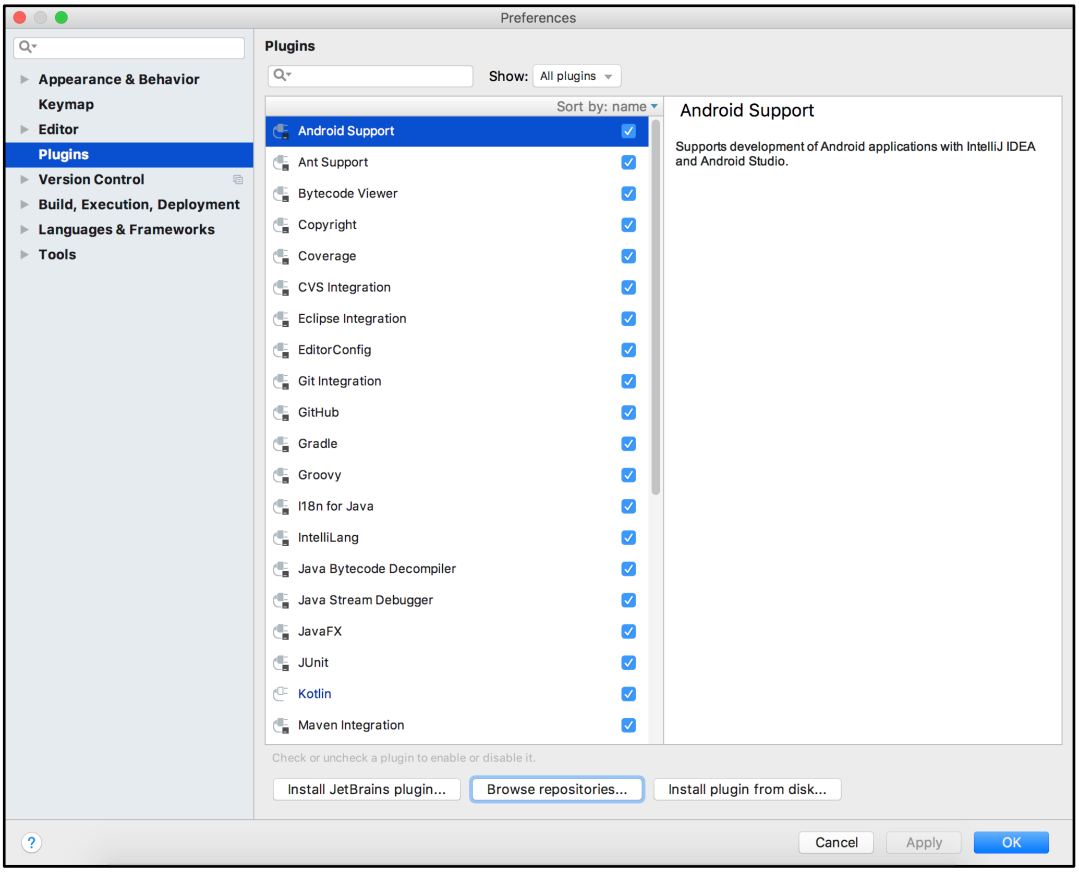

Figure-3: IntelliJ Plugins.

- Search for “Create testng” type string and you will find the plugin named “Create TestNG XML”

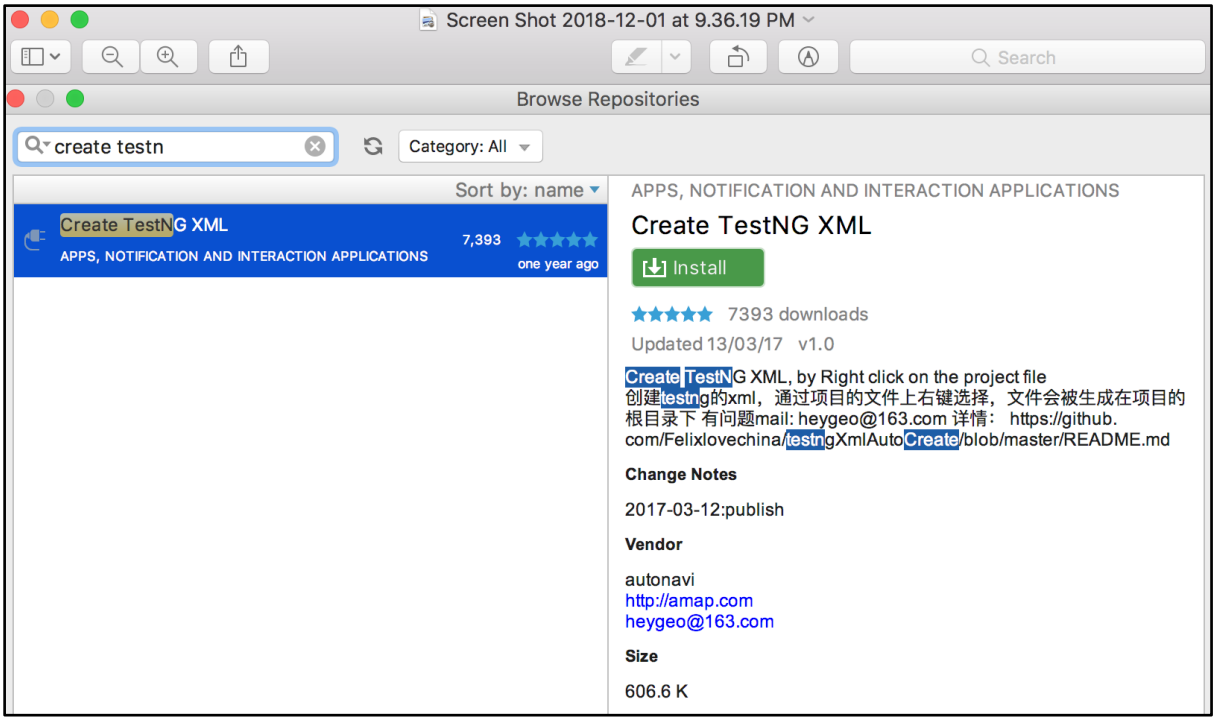

Figure-4: Create TestNG XML plugin.

- After installing the plugin, you need to restart IntelliJ IDEA.

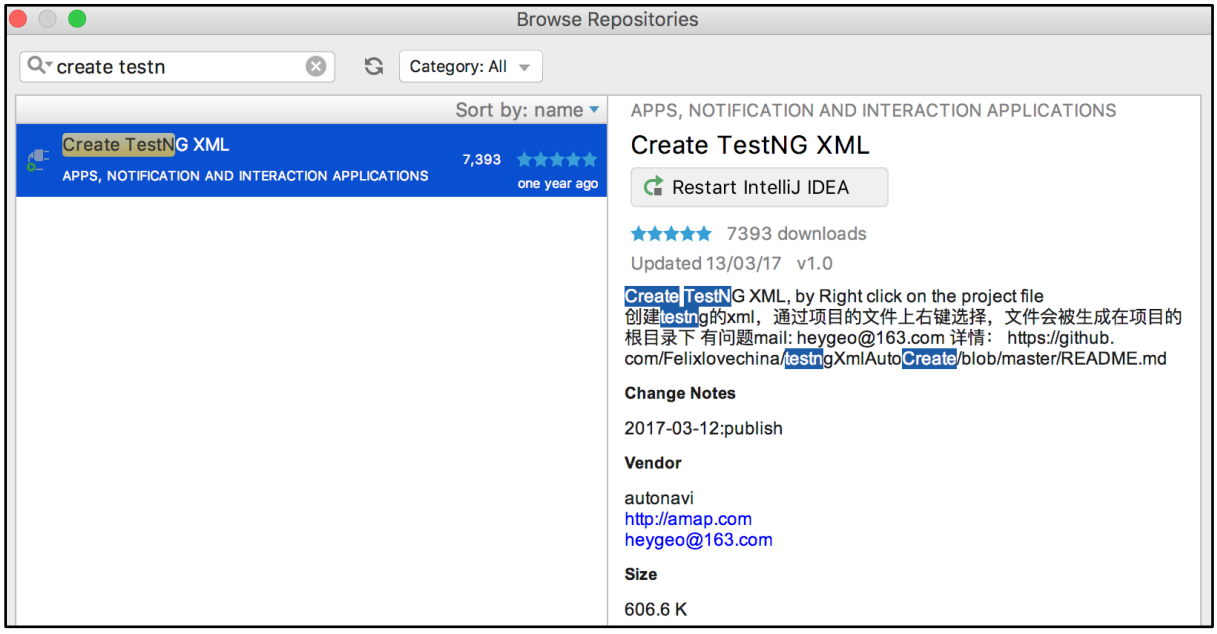

Figure-5: Restart IntelliJ IDEA after installing plugin.

- Now after restarting the IntelliJ IDEA, you are able to generate a TestNG xml file.

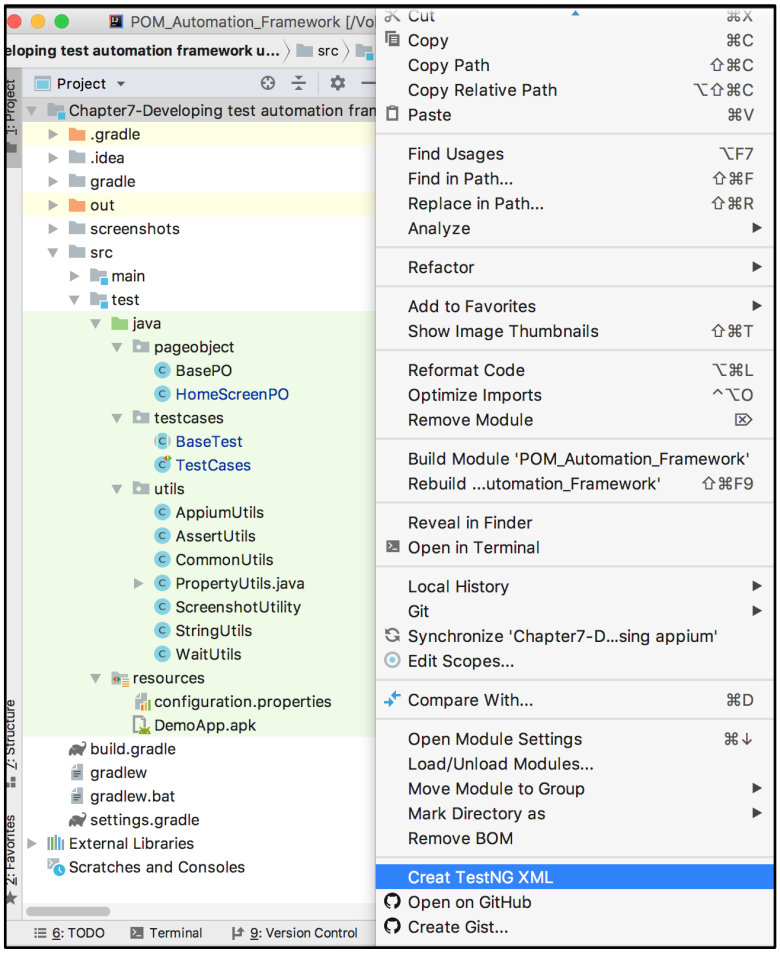

Figure-5: Create TestNG XML.

- In a fraction of a second **testng.xml** will be created and you will get the below confirmation modal dialog.

Figure-6: TestNG created successfully.

- By default **testng.xml** will be created under the project root directory, so it won't be identified by the java compiler at run/compile time. So you need to move it to **test/resources.**

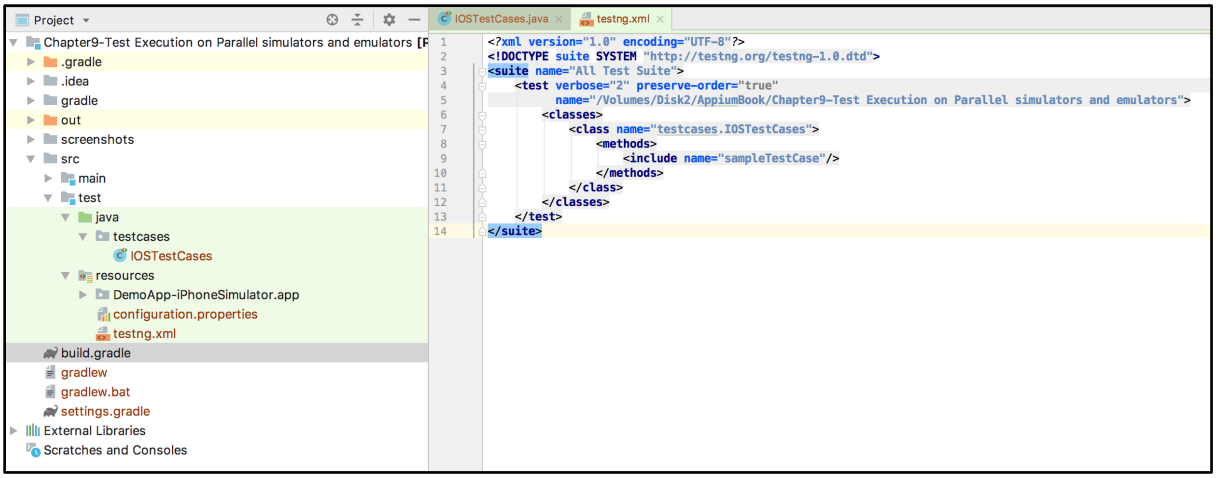

Figure-7: testng.xml under test/resources directory.

So as you can see in above screenshot, there are some pre-defined XML tags present. Here by default **class** tag has **name="testcases.IOSTestCases"** attribute, which means TestNG will execute all test cases under this test class only. And as we have only one test case(sampleTestCase) is defined under **testcases.IOSTestCases** it will run/execute only one test case.

You can also change the testng.xml and make it run at the package, class, method and group level.

How to run the testng.xml?

Well, the answer is just a 2 step process:

1) Right click on testng.xml
2) Select Run

And that's it, your tests under the **IOSTestCases** will start the execution sequentially.

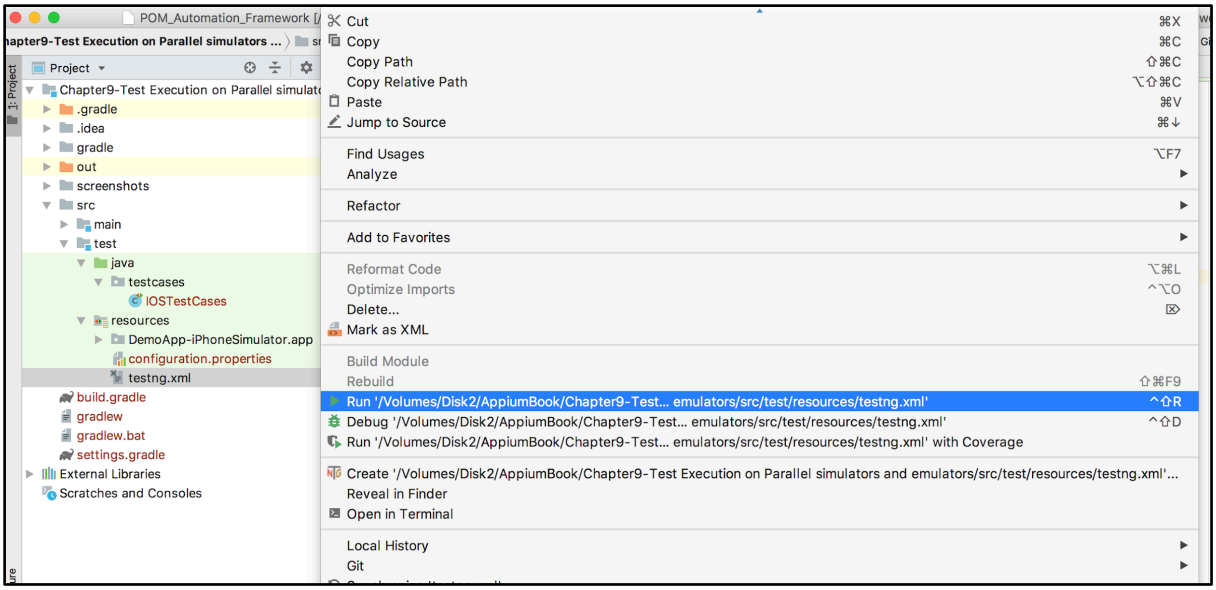

Figure-8: Run the testng.xml.

Parallel execution of automation tests is a really important concept - by executing the tests in parallel we can save a lot of time.

1) Parallel execution of tests on iOS simulators

Our goal is to execute a single test case (sampleTestCase) written in IOSTestCase.java on 3 iOS Simulators(iPhone 7, iPhone 8 and iPhone X) in parallel.

In order to achieve that we need to understand 2 important concepts:

1) **We need to manage the Appium Server from Code:**

   Up until now we were using the Appium Desktop Application to start the Appium server. Normally, one Appium server is bound to one Appium Session (or you can say to one device/simulator), but now as we need to run the test case on 3 iOS simulators at the same time, we need ***3 Appium servers running on three different ports.*** Now you can't rely on the Appium Desktop Application as it will be able to run only one Appium Server.

   So the best option left to us is to create and run 3 Appium servers at runtime, and using Java you can easily create the Runtime Appium Server by mentioning the particular port. In our case we will need 3 different ports to start 3 different Appium Servers(or Sessions).

   Below is the code which will start and stop the Appium server for port 4725.

```
// Create AppiumDriverLocalService object with specifying the
port.
AppiumDriverLocalService service = new
```

```
AppiumServiceBuilder().usingPort(4725).build();
// To start Appium server.
service.start();

// To End Appium server.
service.start();
```

2) **We need to use parameters in Test Class and testng.xml**

We need to start 3 Appium Servers on 3 different ports, and each device should be assigned to a single Appium Server with a unique WDA port.

What is a WDA Port?

It's nothing but used to forward traffic from the Mac host to the iOS Simulator to real ios devices over USB

This table may make it more clear:

No.	Appium Server Port	Device Name (Simulator)	WDA Local Port
1	4725	iPhone 7	8100
2	4726	iPhone 8	8200
3	4727	iPhone X	8300

Now we need to pass these values from testng.xml to our Test Class before all initialization of the WebDriver takes place. And we will use the TestNG **@Parameters** annotation for that.

Please consider the below example to understand how @Parameter at the Test Class level and <parameter> at testng.xml level works.

IOSTestCases.java

```
@Parameters({"wda", "deviceName", "port"})
@BeforeTest
public void setUp(long wda, String deviceName, String port){

AppiumDriverLocalService service = new
AppiumServiceBuilder().usingPort(Integer.valueOf(port)).build
();
```

```
service.start();

DesiredCapabilities dc = new DesiredCapabilities();
dc.setCapability(IOSMobileCapabilityType.WDA_LOCAL_PORT,
wda);
dc.setCapability(MobileCapabilityType.DEVICE_NAME,
deviceName);
….
….
}
```

testng.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="All Test Suite">
    <test name="test1">
        <parameter name="wda" value="8100"/>
        <parameter name="deviceName" value="iPhone 7"/>
        <parameter name="port" value="4725"/>
        <classes>
            <class name="testcases.IOSTestCases"/>
        </classes>
    </test>
</suite>
```

In the above example we are passing 3 values from testng.xml to the IOSTestClass.java:

1) **wda:** Which will be passed to Desired capabilities `IOSMobileCapabilityType.WDA_LOCAL_PORT`

2) **deviceName:** Which also would be passed to Desired capabilities of `MobileCapabilityType.DEVICE_NAME`

3) **port:** Used to create the Appium server.

Now let's come to the parallelization part. If you want to run test cases in parallel then you need to use attributes along with the `<suite>` tag.

1) `parallel`: It has a number of possible values such as ***tests, classes, method*** and ***instances***. If you want to run parallelization at `<test>` then you can use `parallel="tests"`

2) `thread-count`: No of threads to execute in parallel. If you want to execute 5 test cases in parallel then `thread-count="5"`

```
<suite name="All Test Suite" parallel="tests" thread-
count="5">
```

To reach our Goal we need to execute test cases from `IOSTestCases.java` on 3 iOS simulators in parallel, so below is the testng.xml file we can use:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="All Test Suite" parallel="tests" thread-
count="3">

    <test name="test1">
        <parameter name="wda" value="8100"/>
        <parameter name="deviceName" value="iPhone 7"/>
        <parameter name="port" value="4725"/>
        <classes>
            <class name="testcases.IOSTestCases"/>
        </classes>
    </test>

    <test name="test2">
        <parameter name="wda" value="8200"/>
        <parameter name="deviceName" value="iPhone 8"/>
        <parameter name="port" value="4726"/>
        <classes>
            <class name="testcases.IOSTestCases"/>
        </classes>
    </test>
    <test name="test3">
        <parameter name="wda" value="8300"/>
        <parameter name="deviceName" value="iPhone X"/>
        <parameter name="port" value="4727"/>
        <classes>
            <class name="testcases.IOSTestCases"/>
        </classes>
```

```
    </test>

</suite>
```

After adding this to testng.xml you just need to ***Right click on it and select Run option,*** and you will see the 3 iOS Simulators will open and each of them will execute test cases in parallel. Great!

We have discussed only one possible way to achieve parallelization. There are many other ways out there and you can also create your own.

2) Parallel execution of tests on real iOS devices

In the previous section we looked at test execution on iOS Simulators, but what if you want to execute tests on Real Devices? In the next chapter we will look at using a cloud testing service like **Kobiton**, but for now, let's look at how you may need to run parallel tests using the real-devices on-hand.

The answer is pretty simple - we just need to pass **UDID** as a 4th parameter.

Let's understand this by way of an example. Let's say we have 2 real iOS Devices connected to our Mac host and we want to run the test cases on that in parallel. Please look at the table below for Device capability and Port information.

No.	Appium Server Port	Real Device Name	UDID	WDA Local Port
1	4725	John's iPhone	2b6f0cc904d137be2e1730235f5664094b831186	8100
2	4726	iPhone	d137be2e12b6f0cc90473031186235f5664094b8	8200
3	4727	iPhone X	137b30235f5664094b831186e22b6f0cc904de17	8300

IOSTestCases.java and **testng.xml** will look like below.

IOSTestCases.java

```
@Parameters({"wda", "udid", "deviceName", "port"})
@BeforeTest
public void setUp(long wda, String udid, String deviceName,
String port){
```

```
AppiumDriverLocalService service = new
AppiumServiceBuilder().usingPort(Integer.valueOf(port)).build
();
service.start();

DesiredCapabilities dc = new DesiredCapabilities();
dc.setCapability(IOSMobileCapabilityType.WDA_LOCAL_PORT,
wda);
dc.setCapability(MobileCapabilityType.UDID, udid);
dc.setCapability(MobileCapabilityType.DEVICE_NAME,
deviceName);
….
….
}
```

testng.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="All Test Suite" parallel="tests" thread-
count="2">

    <test name="test1">
        <parameter name="wda" value="8100"/>
        <parameter name="udid"
value="2b6f0cc904d137be2e1730235f5664094b831186"/>
        <parameter name="deviceName" value="John’s
iPhone"/>
        <parameter name="port" value="4725"/>
        <classes>
            <class name="testcases.IOSTestCases"/>
        </classes>
    </test>

    <test name="test2">
        <parameter name="wda" value="8200"/>
        <parameter name="udid"
value="d137be2e12b6f0cc90473031186235f5664094b8"/>
        <parameter name="deviceName" value="iPhone"/>
        <parameter name="port" value="4726"/>
        <classes>
```

```
            <class name="testcases.IOSTestCases"/>
        </classes>
    </test>

    <test name="test3">
        <parameter name="wda" value="8300"/>
        <parameter name="udid"
value="137b30235f5664094b831186e22b6f0cc904de17"/>
        <parameter name="deviceName" value="iPhone X"/>
        <parameter name="port" value="4727"/>
        <classes>
            <class name="testcases.IOSTestCases"/>
        </classes>
    </test>

</suite>
```

3) Parallel execution of tests on Android emulators

Parallel execution of tests on Android emulators is largely the same mechanism we just explored as with iOS device. The main advantage with parallel execution on Android is that, unlike with iOS, you don't need to provide a wda port.

AndroidTestCases.java

```
@Parameters({"platformVersion", "deviceName", "port"})
@BeforeTest
public void setUp(String platformVersion, String deviceName,
String port) throws MalformedURLException {

AppiumDriverLocalService service = new
AppiumServiceBuilder().usingPort(Integer.valueOf(port)).build
();
service.start();

DesiredCapabilities dc = new DesiredCapabilities();
dc.setCapability(MobileCapabilityType.PLATFORM_VERSION,
platformVersion);
dc.setCapability(MobileCapabilityType.DEVICE_NAME,
deviceName);
….
```

```
….
}
```

testng.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="Android Test Suite" parallel="tests" thread-
count="2">
    <test name="test1">
        <parameter name="platformVersion" value="8.0"/>
        <parameter name="deviceName" value="emulator-
5554"/>
        <parameter name="port" value="4729"/>
        <classes>
            <class name="testcases.AndroidTestCases"/>
        </classes>
    </test>

    <test name="test2">
        <parameter name="platformVersion" value="9.0"/>
        <parameter name="deviceName" value="emulator-
5556"/>
        <parameter name="port" value="4730"/>
        <classes>
            <class name="testcases.AndroidTestCases"/>
        </classes>
    </test>
</suite>
```

Everything else remains the same.

4) Parallel execution of tests on real Android devices

You just need to change the device name in testng.xml. Using `$ adb devices` you can get the connected real device names.

testng.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
```

```
<suite name="Android Test Suite" parallel="tests" thread-
count="2">
    <test name="test1">
        <parameter name="platformVersion" value="7.0"/>
        <parameter name="deviceName"
value="B6AUTSZDYPSOZD7S"/>
        <parameter name="port" value="4739"/>
        <classes>
            <class name="testcases.AndroidTestCases"/>
        </classes>
    </test>
    <test name="test2">
        <parameter name="platformVersion" value="8.0"/>
        <parameter name="deviceName" value="c4e3f3cd"/>
        <parameter name="port" value="4740"/>
        <classes>
            <class name="testcases.AndroidTestCases"/>
        </classes>
    </test>
</suite>
```

When you execute the above testng.xml you can get output result similar to the below image.

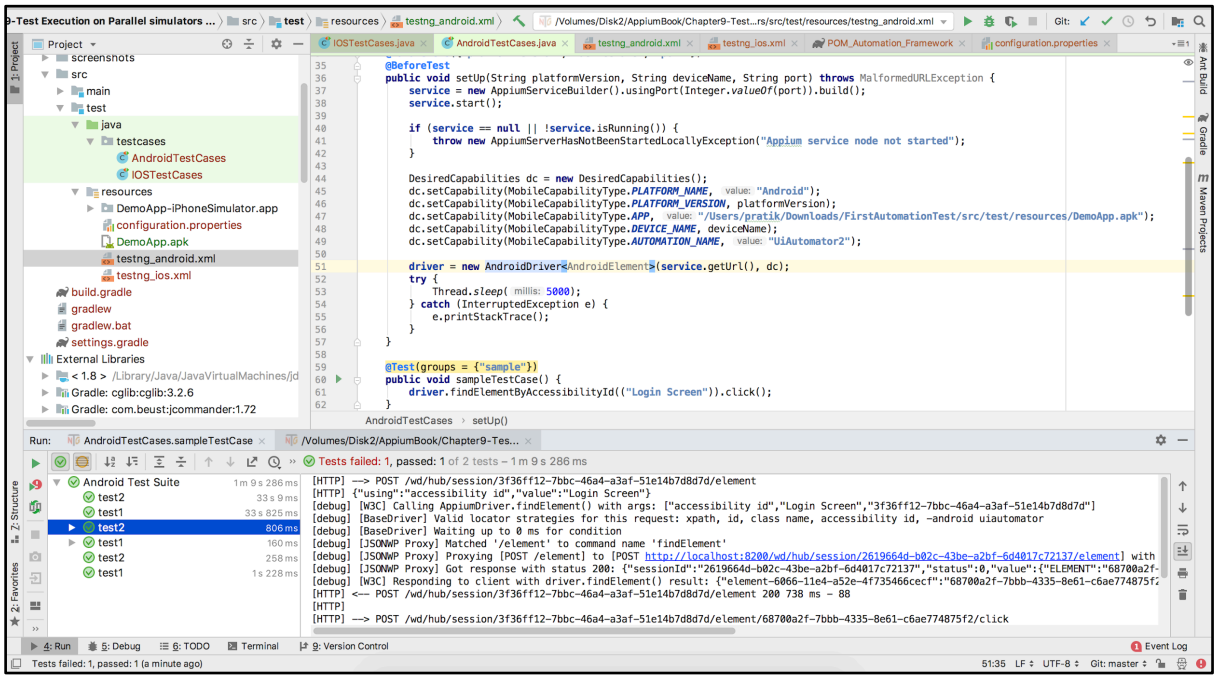

Figure-9: Parallel execution on Real Android Devices.

You can also find this example on our Github page as well:

As we discussed there are many other ways to achieve parallelization using TestNG, so using your custom logic you can even execute tests for both iOS and Android parallelly.

In the next chapter we'll look at how you can use the **Kobiton cloud testing service** to execute your test scripts against real-devices in the cloud.

Chapter-10: Test Execution on Real Devices Using Kobiton

Up until now, you've learned the basics of using Appium and testing against emulators and simulators. However, for production apps it is imperative to test on real devices to ensure the behavior is as expected for your end-users. The mobile ecosystem is particularly fragmented given the wide range of device manufacturers and different versions of operating systems. The only way of knowing that it *really* works on real-devices is to test your app on a wide variety of devices.

It is imperative therefore to introduce real-device testing into your quality process.

The question then becomes "how"? After all, it would seem that working with simulators/emulators is much simpler than buying physical devices which are constantly being released. Fortunately, there are services which makes this a breeze.

A Cloud Device Provider provides a set of real devices on the cloud at scale. The main purpose of any cloud device provider is to provide the testing infrastructure upon which the test cases will be executed on, so the team is can be focused on producing the test scripts.

The main advantage to use a cloud device service is that you will always have access to the latest (and older) set of devices.

Another advantage is that users can see the test execution details such as screenshots, execution logs, failure log details and even video. And if you have a geographically dispersed team, everybody has access to the devices and the session logs.

Popular cloud device providers include Kobiton, Perfecto, Browserstack and SauceLabs. In this chapter we will look at using Kobiton.

Introduction to Kobiton

Kobiton focuses exclusively on real devices rather than emulators and simulators. Kobiton uses its own Appium Server so you don't need to worry about starting your own Appium server - you focus on the test scripts!

Kobiton provides[4]:

[4] Kobiton Website: https://kobiton.com/real-device-testing/

1) Rich test logs for true Root Cause Analysis

Kobiton provides various analysis and reporting features such as:

- Full video recording.
- Screenshots.
- Capture user interactions.
- System metrics reporting.
- Full device logs.
- Script execution results.

2) Integration with your favorite tools

You can also integrate it with various tools such as:

- JIRA
- Github
- HockeyApp
- Jenkins
- Travis CI
- TeamCity

3) Powerful APIs

Kobiton provides the full support of Appium with major programming languages such as Java, Python, C#, Node.js, Ruby and PHP and they have also did partnership with Katalon Studio(A highly acclaimed and innovative test automation tool).

4) Manual, Automated and Parallel testing supported

It provides support for parallel execution which is necessary for CI/CD process and also supports:

- On-demand manual testing.
- Automation testing.
- Supports Appium and Selenium.
- Mobile web and cross browser testing.
- UI testing.
- Supports parallel script execution.
- Full DevOps integration.

Now let’s see how easy it is to test your Appium script on real-devices in the cloud.

Step-by-step guide

1) Visit https://kobiton.com/ and register as a new user.

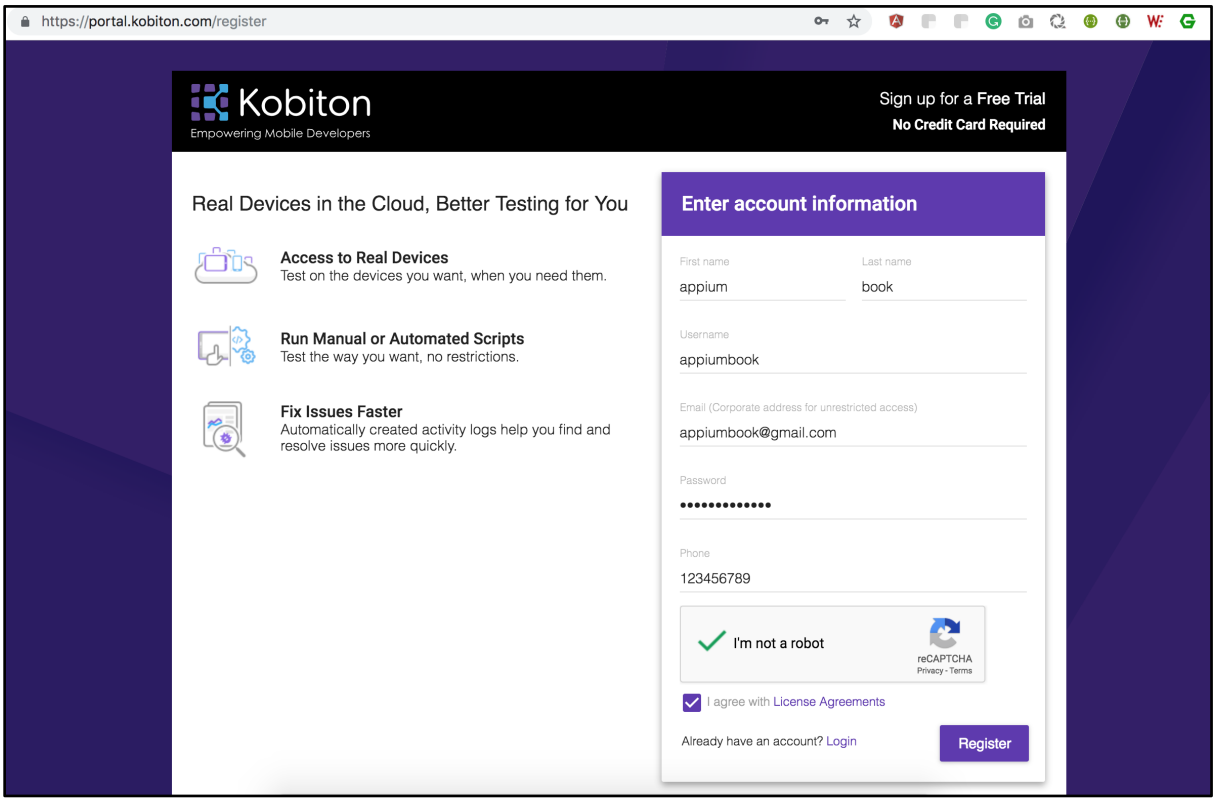

Figure-1: Kobiton: Registration Page.

2) After the successful registration you will get an email for the confirmation from the Kobiton so you can confirm it.

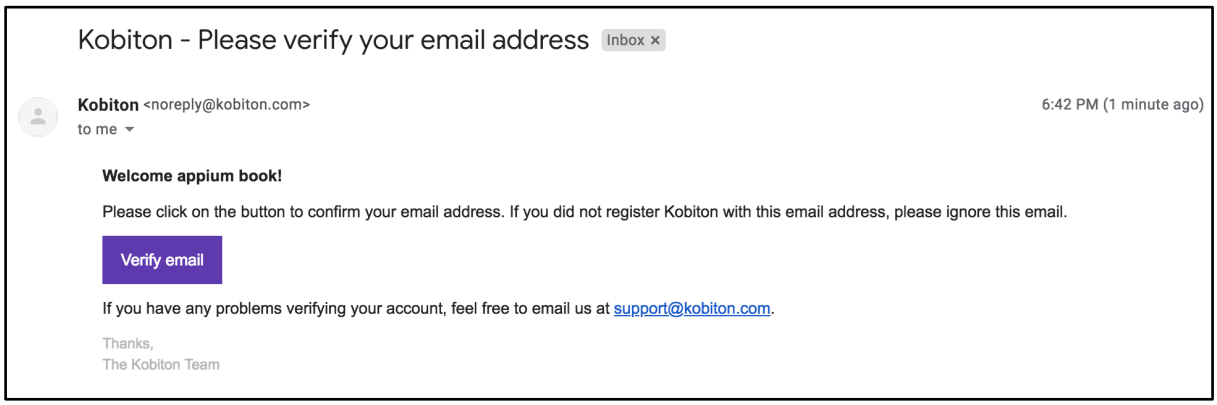

Figure-2: Kobiton: Confirmation Email.

3) Now you can access the Kobiton portal by providing valid credentials.

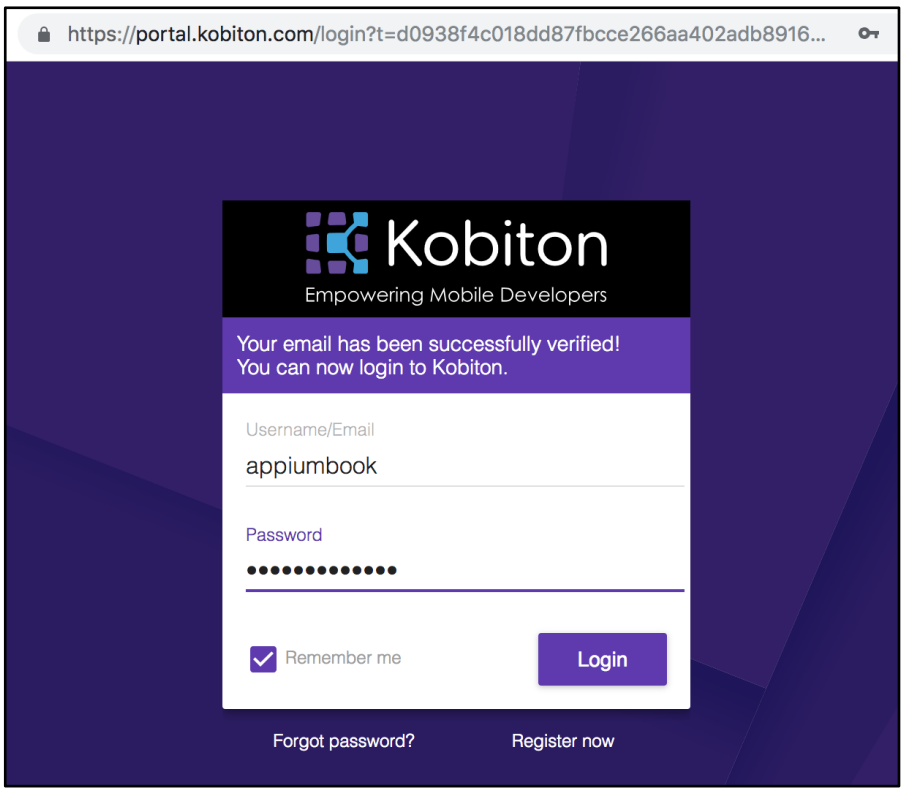

Figure-3: Kobiton: Login Page.

4) As part of the free trial, you get 120 minutes. By completing the survey you can get an additional 30 minutes of testing.

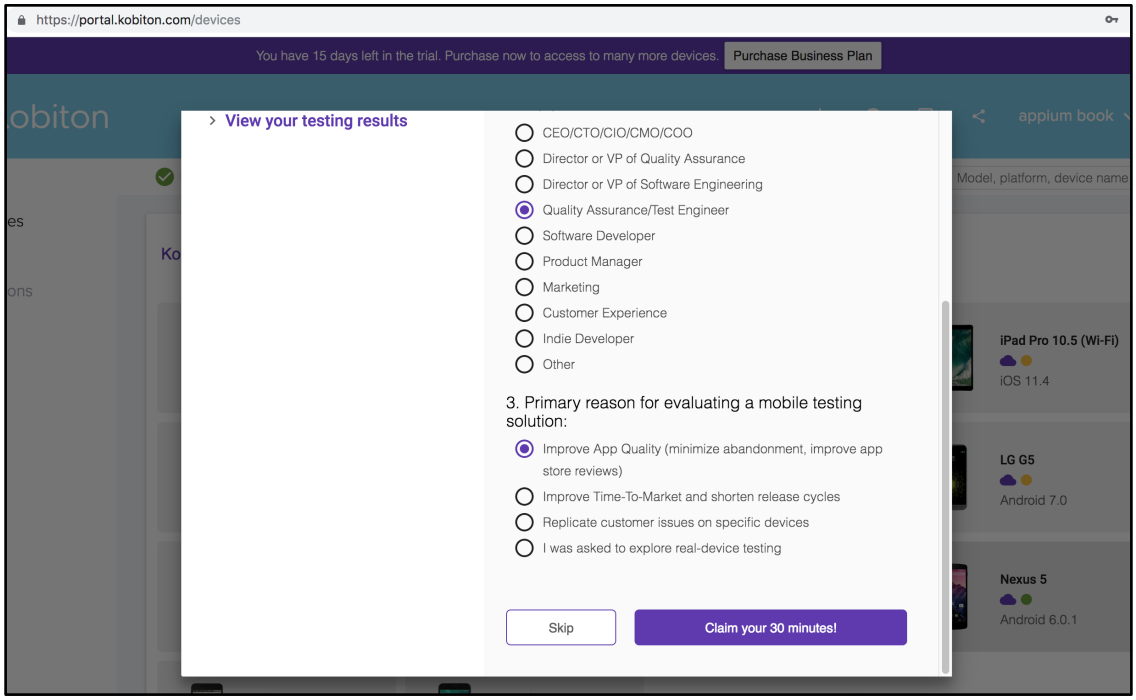

Figure-4: Select the preferences.

5) Once you click on “Claim your 30 minutes!” you will be redirected to the dashboard where you can see the devices.

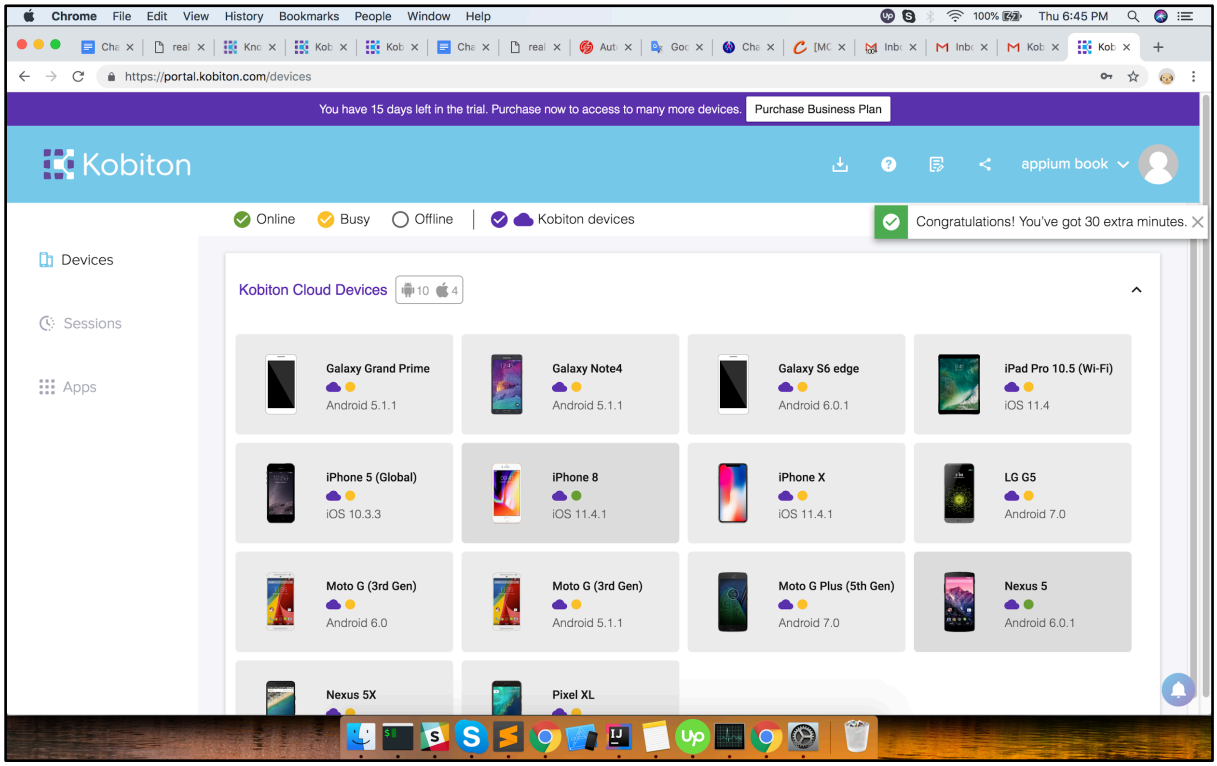

Figure-5: Kobiton: Dashboard.

NOTE: *Please make sure the device upon which you want to start execution is Online (Available for execution) otherwise you will get the exception saying: org.openqa.selenium.SessionNotCreatedException: No device matching the desired capabilities. Note that Kobiton provides exclusive/private devices if needed.*

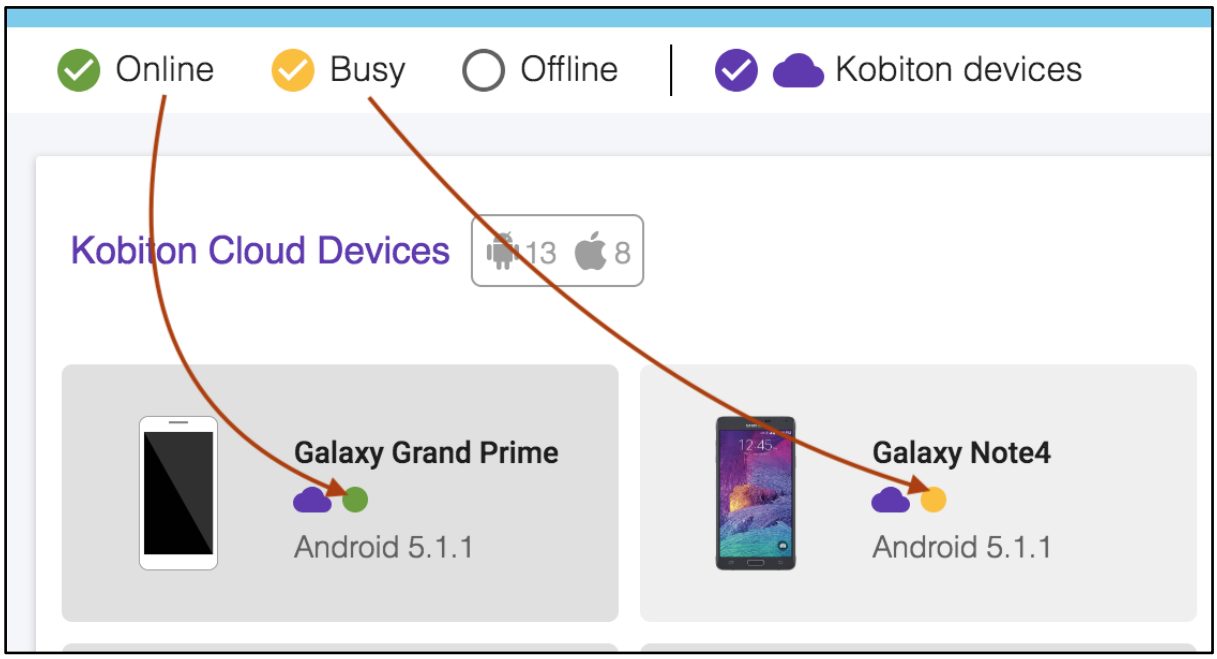

Figure-6: Device Availability.

6) Now on the dashboard page you can choose any device for test execution. But before you choose the device for execution you need to upload the mobile application which you want to automate by going into the Apps section.

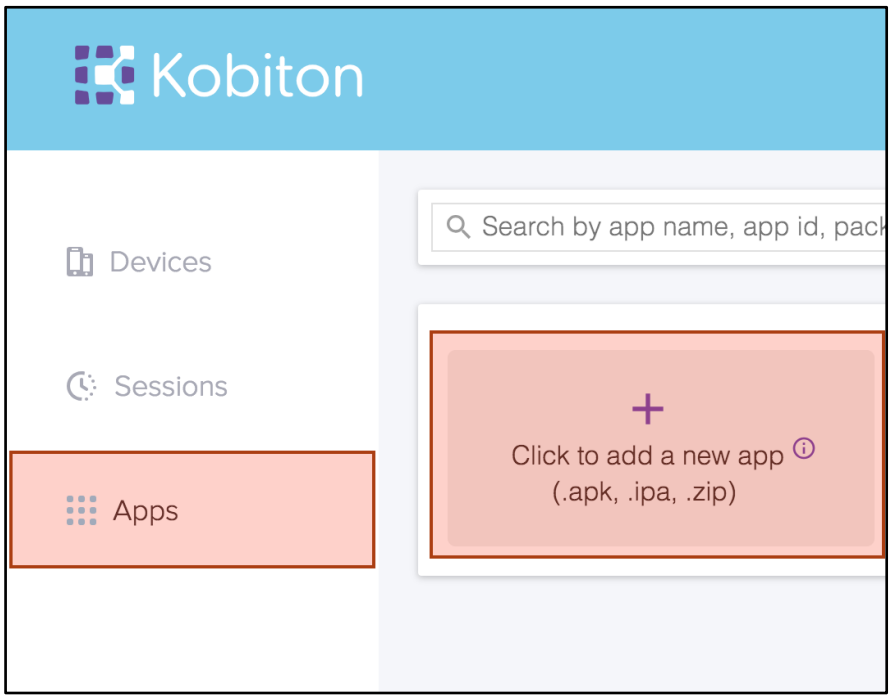

Figure-7: Add a new app.

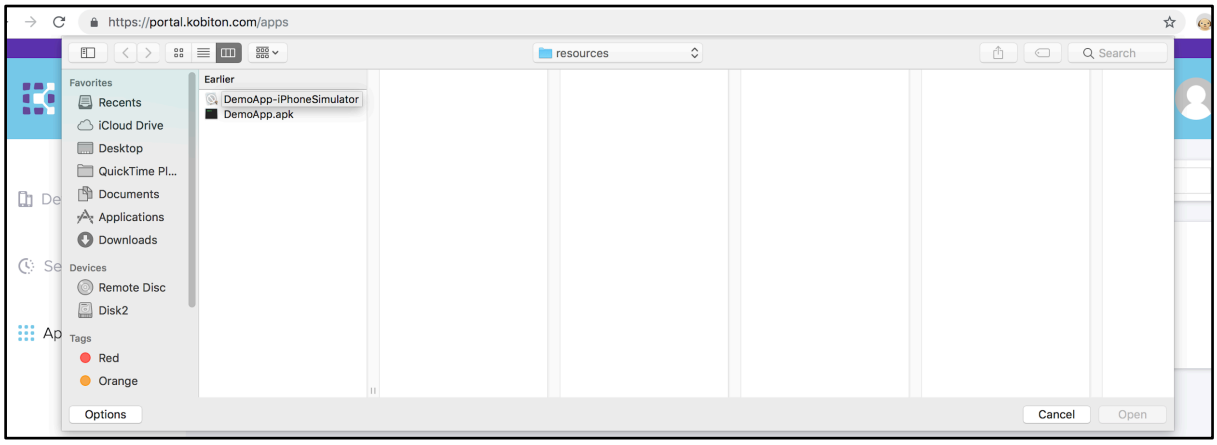

Figure-8: Select the application from finder window.

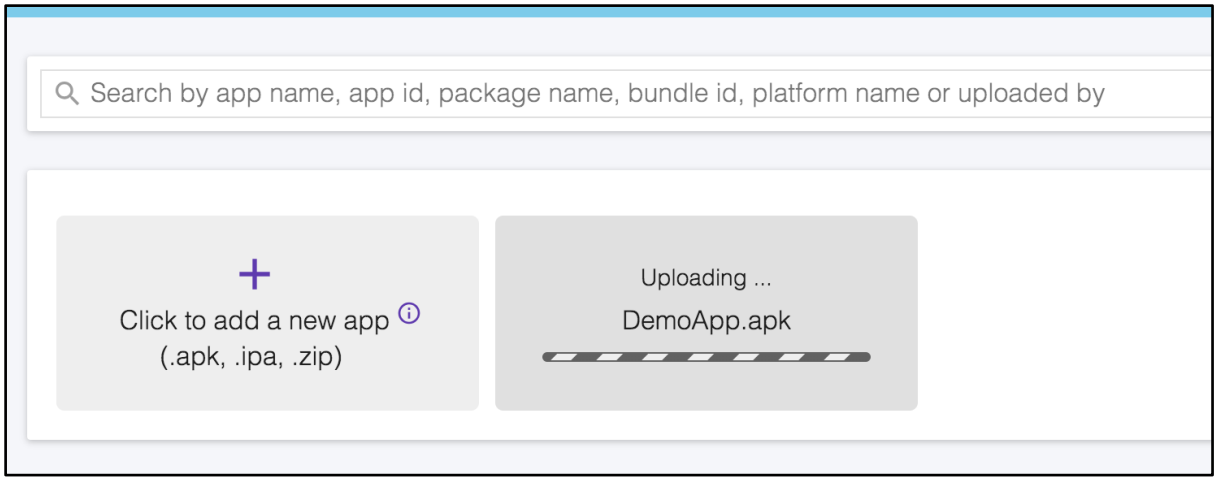

Figure-9: Uploading the new .apk file

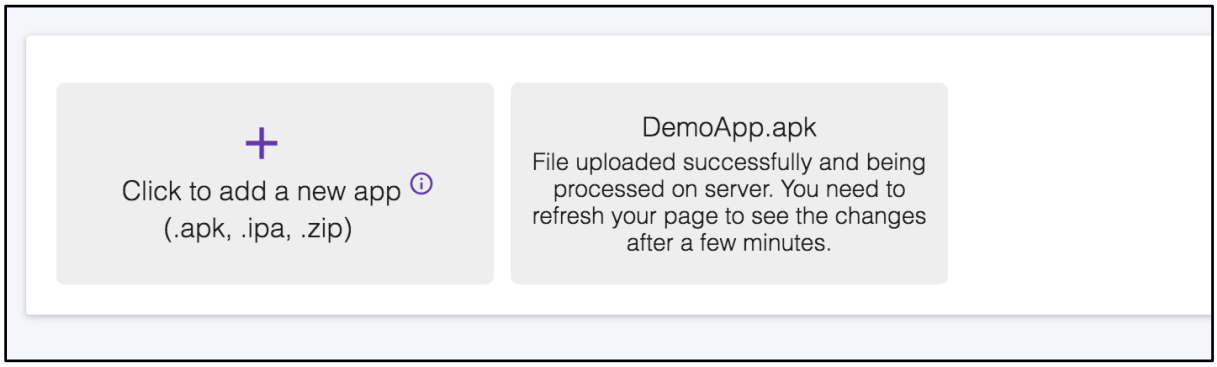

Figure-10: .apk file uploaded.

7) Once the application is uploaded successfully you can get the app id for the uploaded application because you need to pass this id as value of `app` key in Desired Capabilities.

```
capabilities.setCapability("app", "kobiton-store:22304");
```

Or you can also select the application in the automation settings dialog, so that you don't need to manually write the `app` value in desired capabilities.

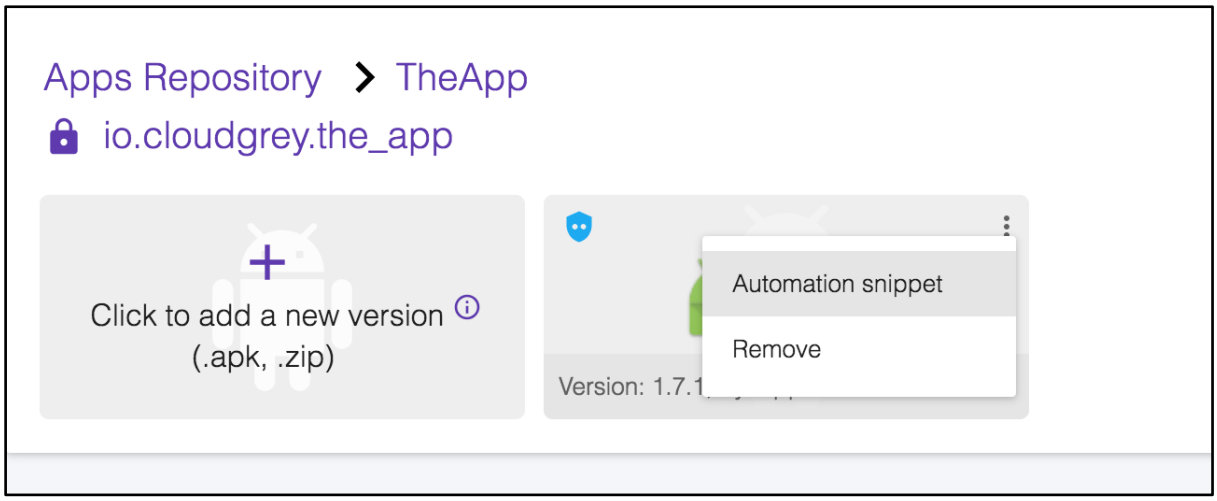

Figure-11: Get Automation snippet.

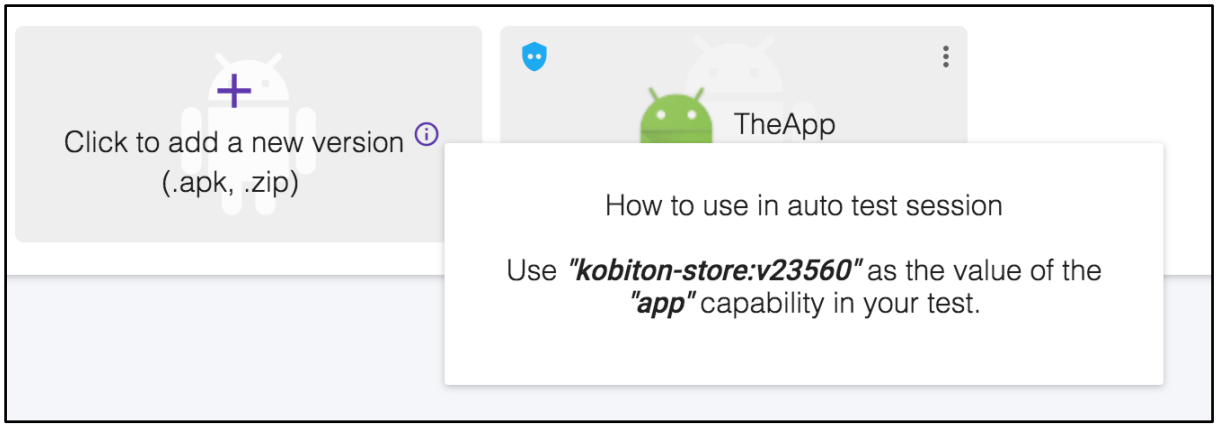

Figure-12: app capability value

8) After noting down the app value you can move to the Devices section and in the automation settings you will get the set of the Desired Capabilities (for every programming language) which you need to use in order to execute the test case on Kobiton device.

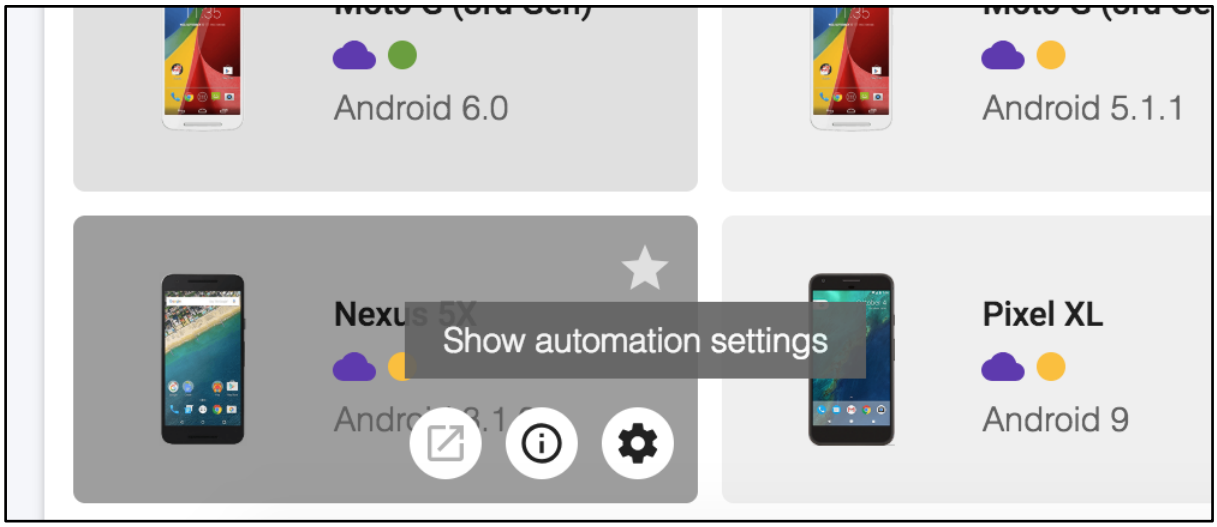

Figure-13: Show automation settings for Device.

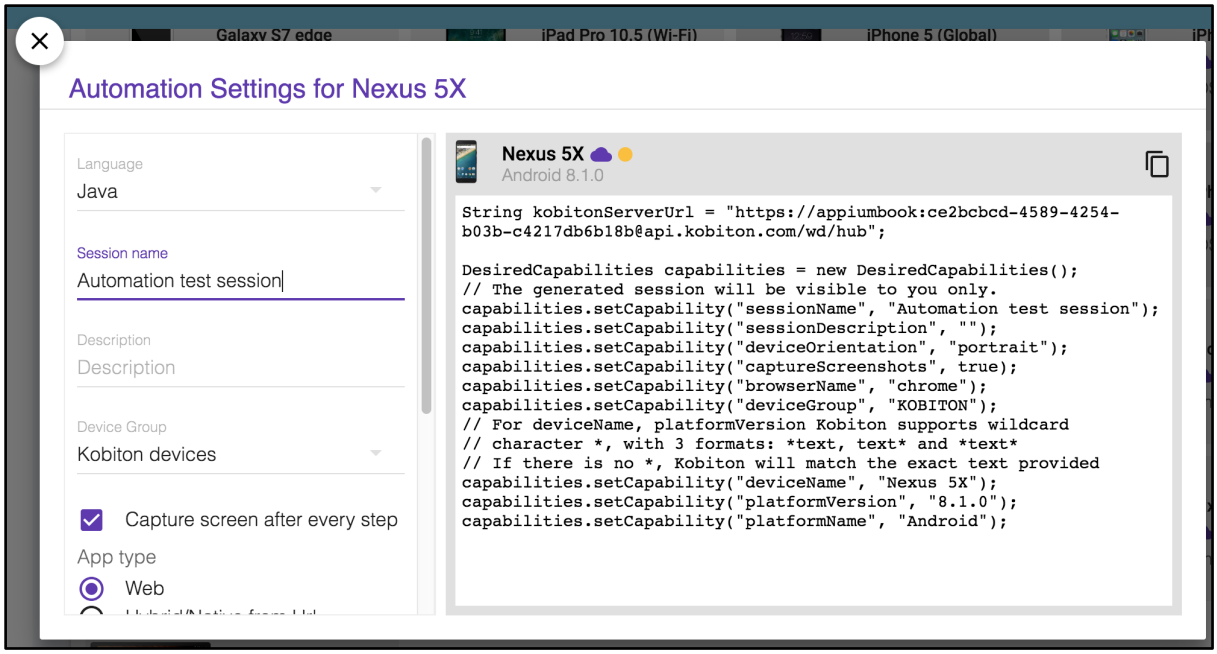

Figure-14: Automation Settings for Device.

As we want to execute test cases for a mobile application, you need to select the ***"Hybrid/Native from Apps"*** option.

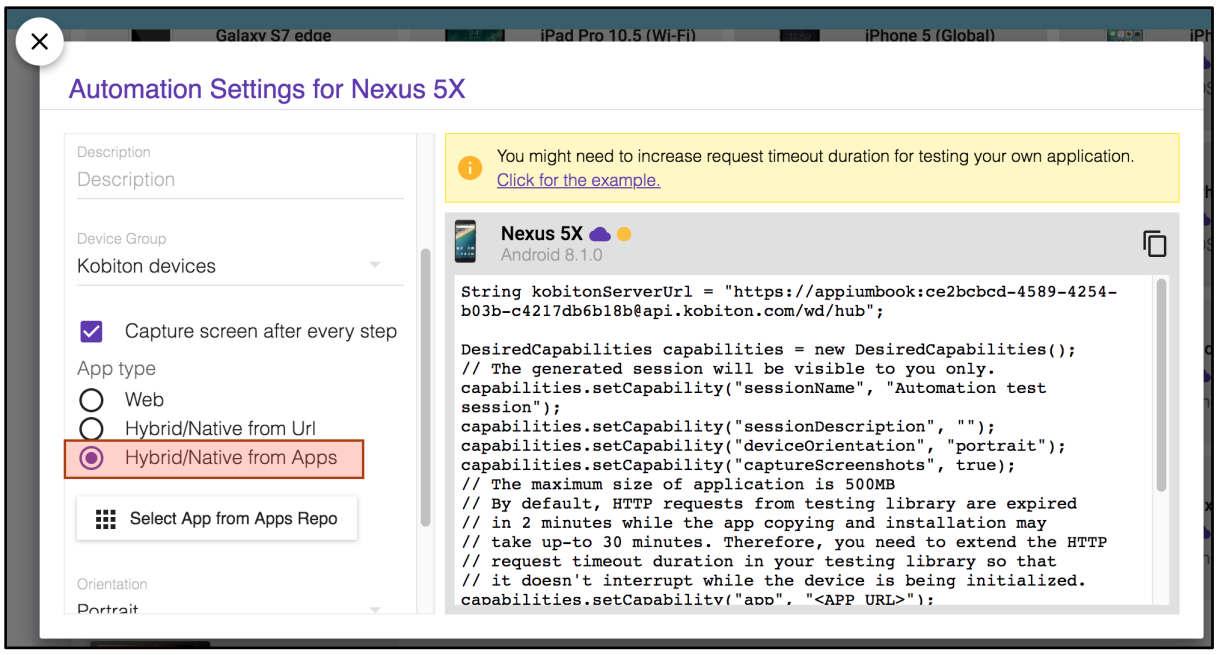

Figure-15:Choose Hybrid/Native from Apps.

Now as we have already uploaded the application to Kobiton server, so you just have to click on ***"Select App from Apps Repo"*** button and select the uploaded app.

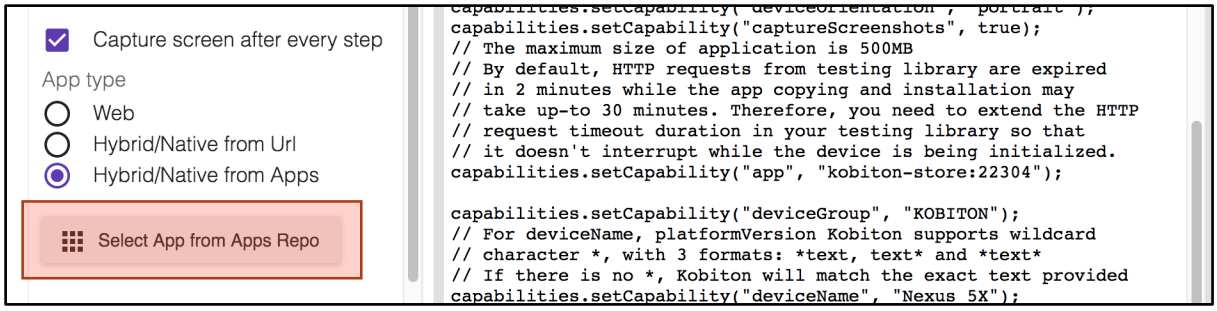

Figure-16: Click on Select App from Apps Repo button.

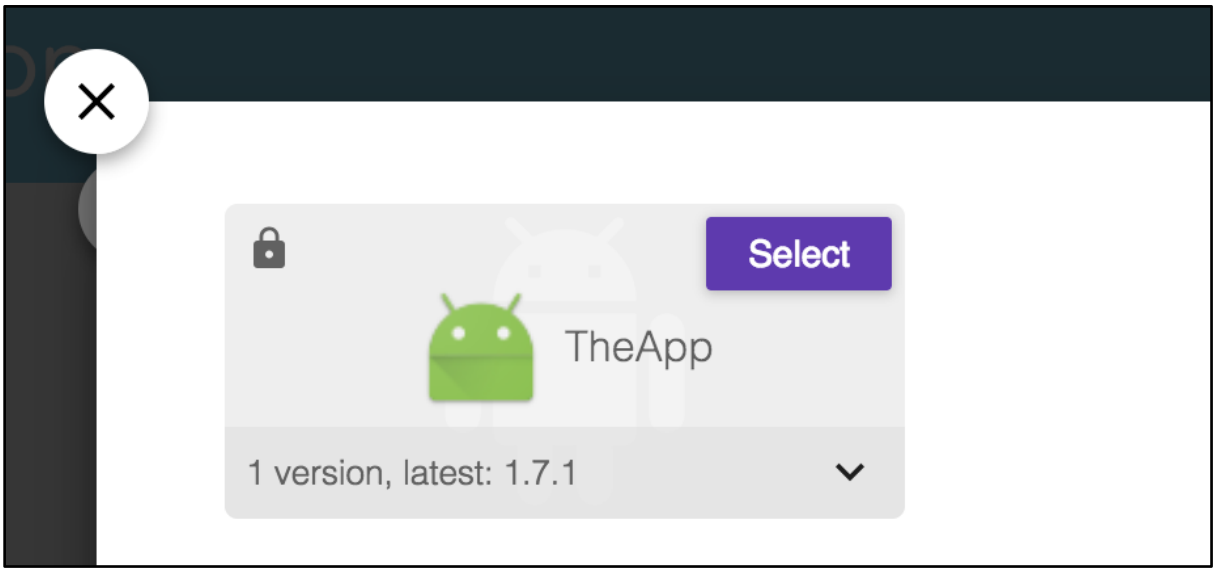

Figure-17: Select uploaded app.

And at last copy the set of Desired Capabilities.

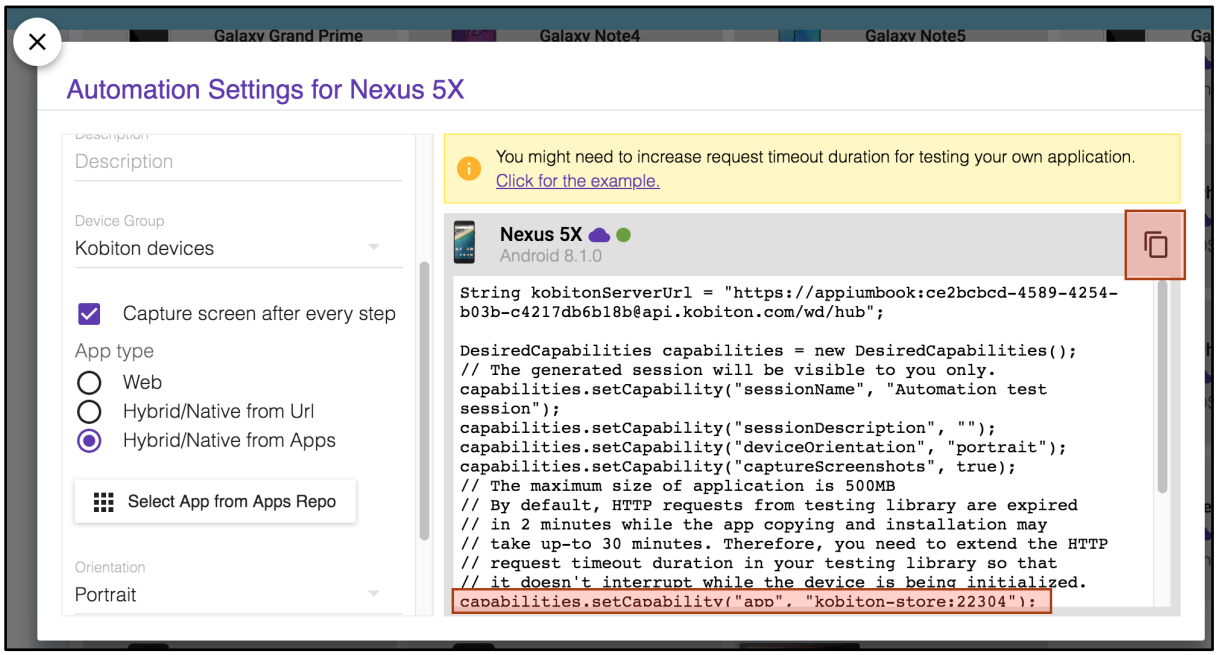

Figure-18: Copy Desired Capabilities.

9) Now all we need to do is paste the desired capabilities to our test script and run it. And here we are using the same android script which we have gone through in early chapters.

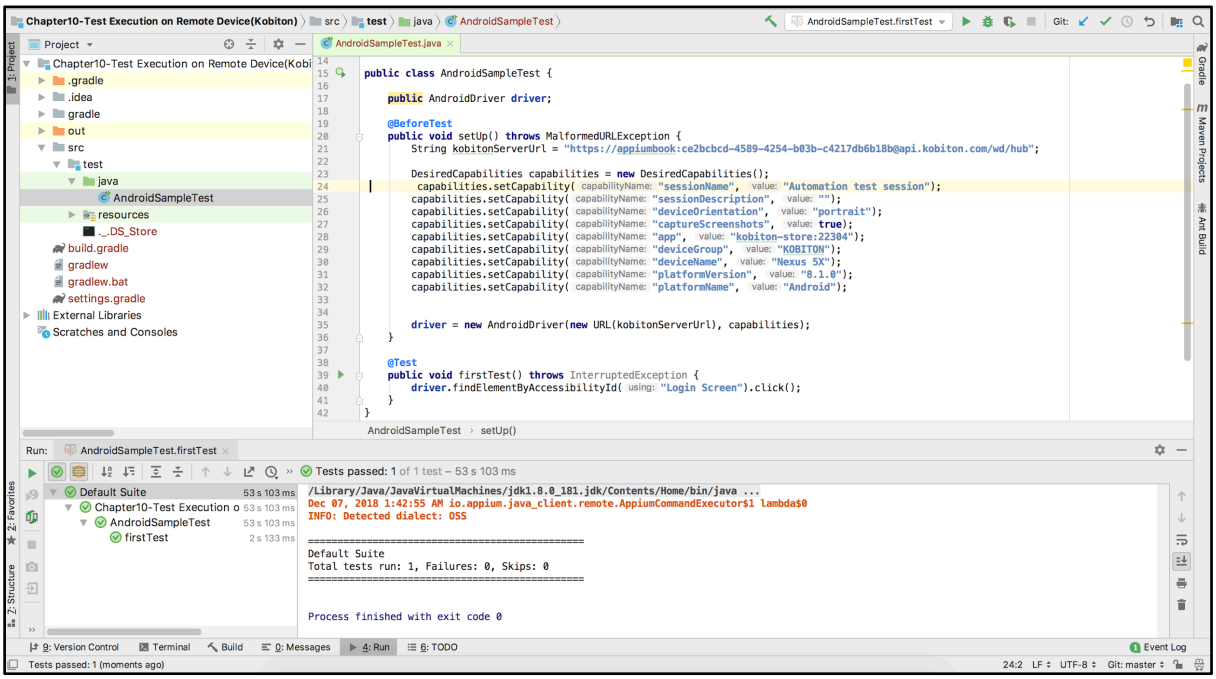

Figure-19: Paste Desired Capabilities in Test Script.

10) Now when you run the script, you can see the session is created for the selected Kobiton device.

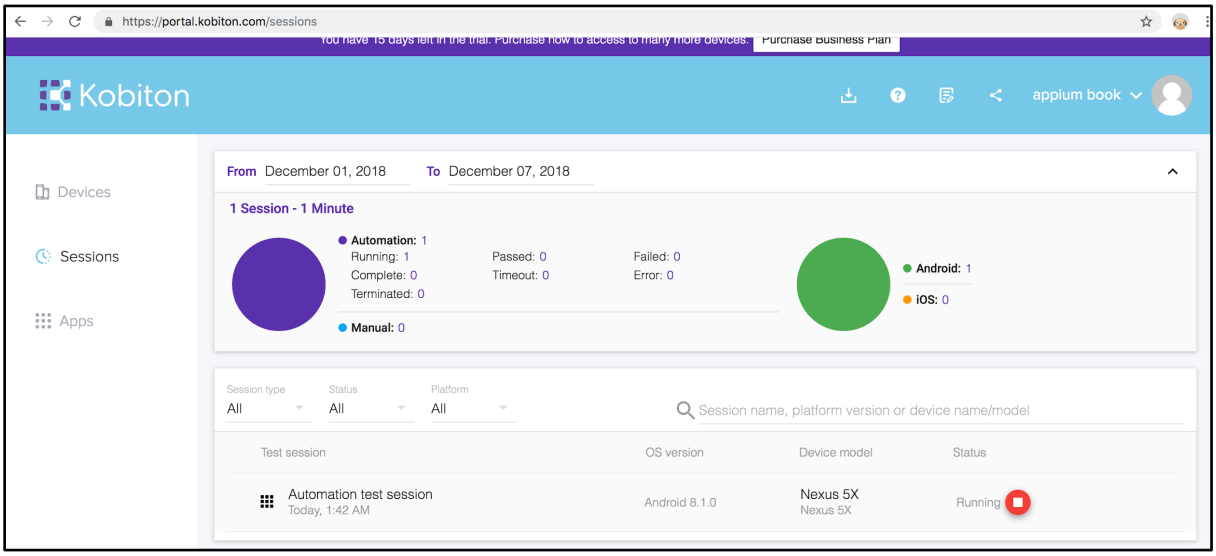

Figure-20: Kobiton Session is created.

11) Once execution is finished, you are able to analyze the logs, screenshots, video and even HTTP commands.

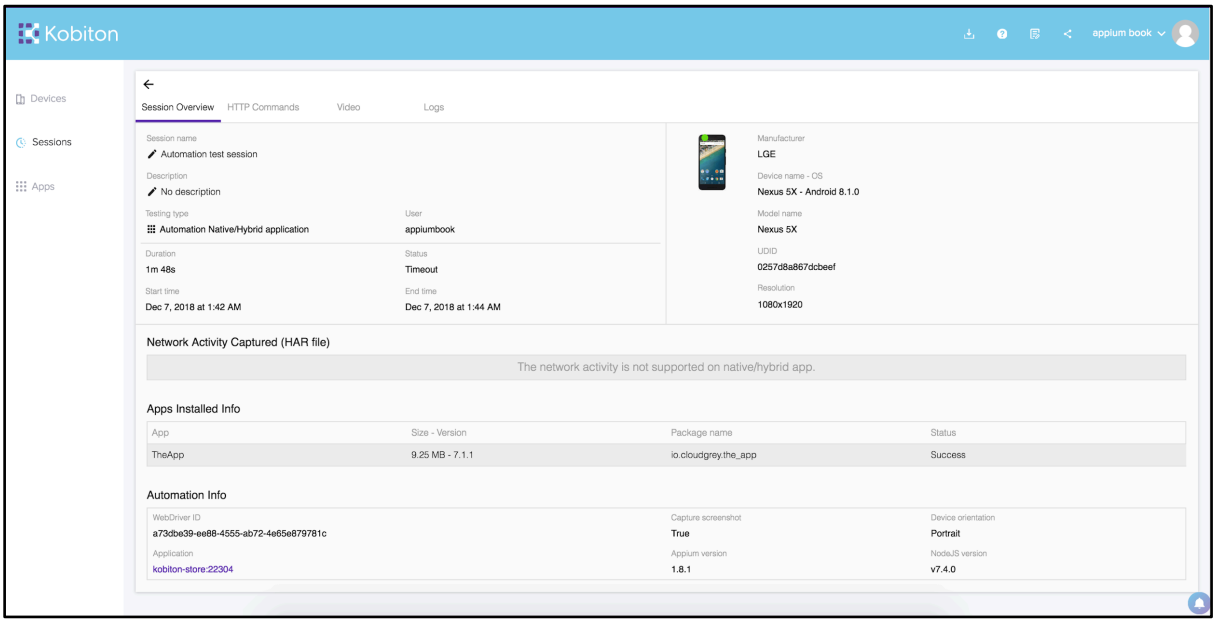

Figure-21: Session Overview.

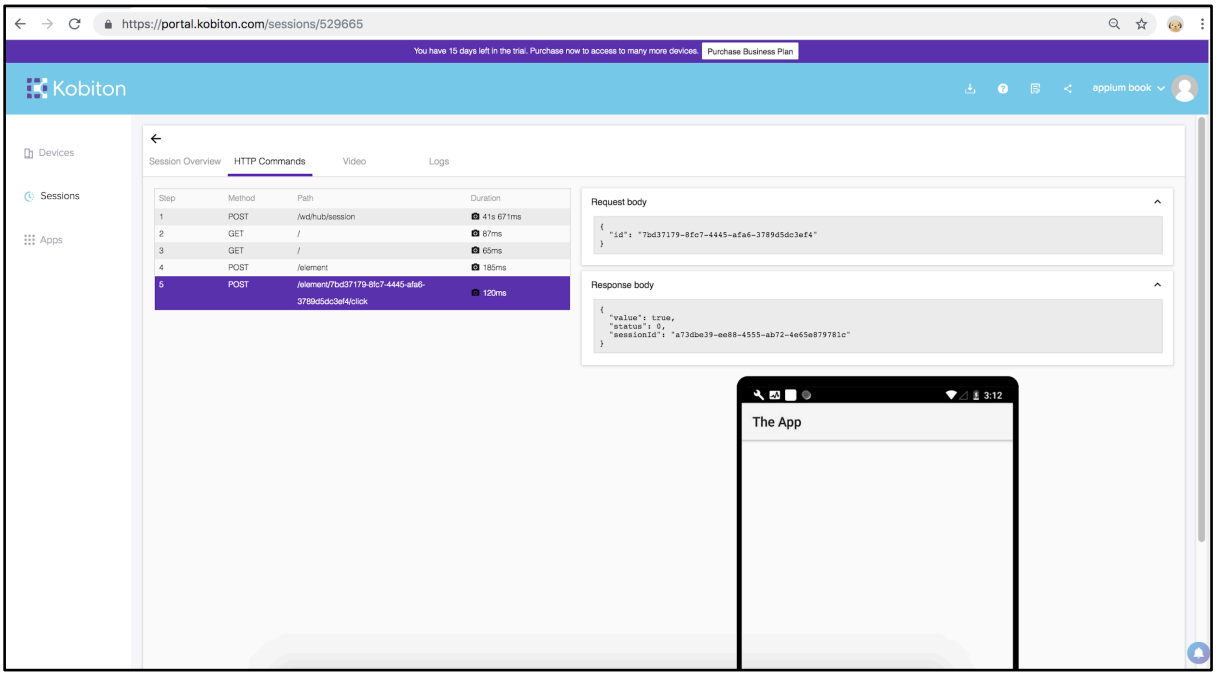

Figure-22: HTTP Commands.

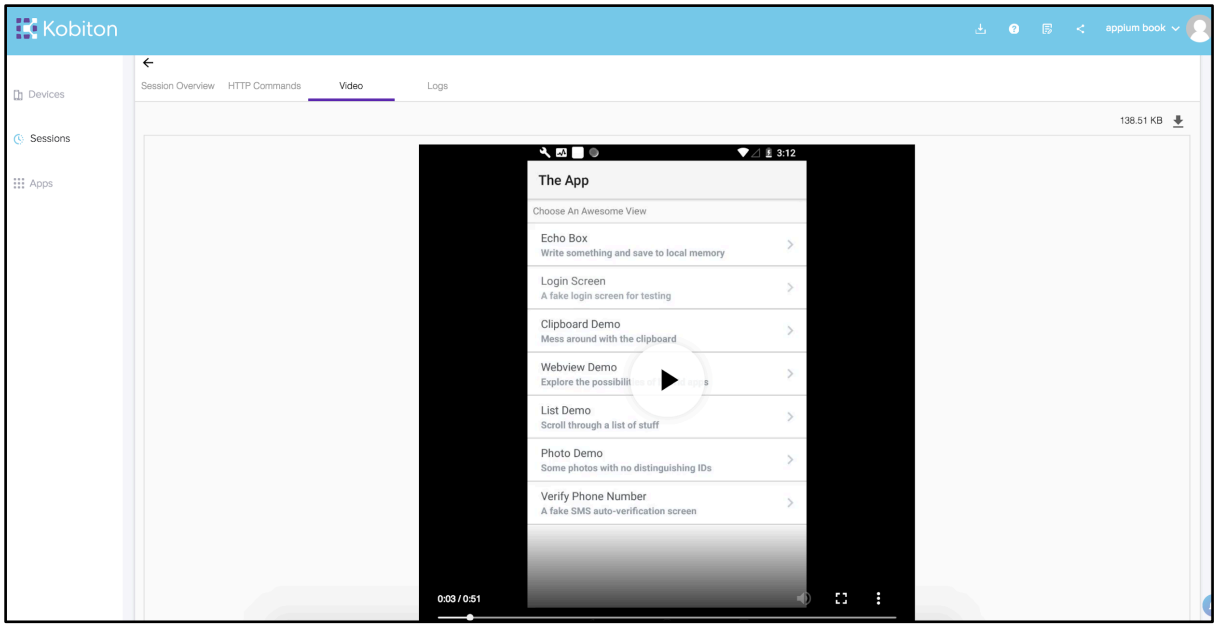

Figure-23: Execution Video.

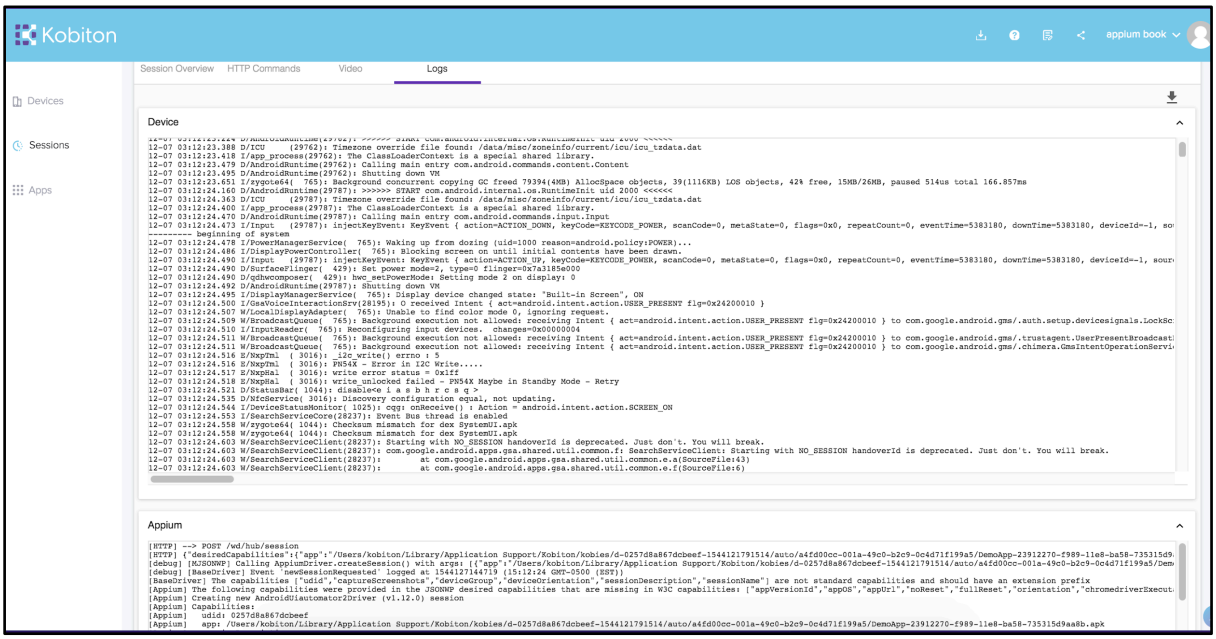

Figure-24: Execution Logs.

It's that easy!

You can find the above test code on our github page:

Execute test cases on a Mobile Browser

Kobiton also supports mobile web testing in addition to native apps.

Testing for mobile web is really easy - you just need to select the ***"Web"*** option in the Automation settings for the device and simply copy the desired capabilities. And

because we're testing a web application in this case, there is no need to upload an app.

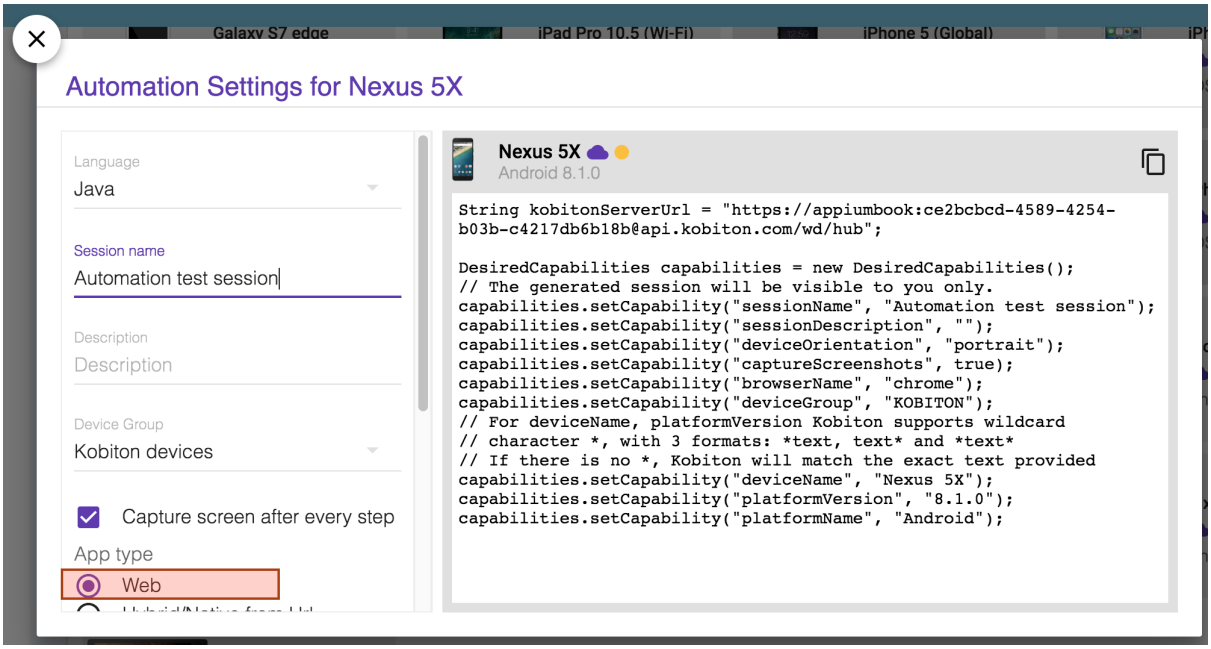

Figure-25: Automation Settings for Mobile Browser execution.

Moreover, if you want to manually test your mobile application on a Kobiton device you can easily do that, you just need to select the device click on the Launch button.

Figure-26: Launch device.

In this manual mode you can control the real device the same as if you had the device in-hand.

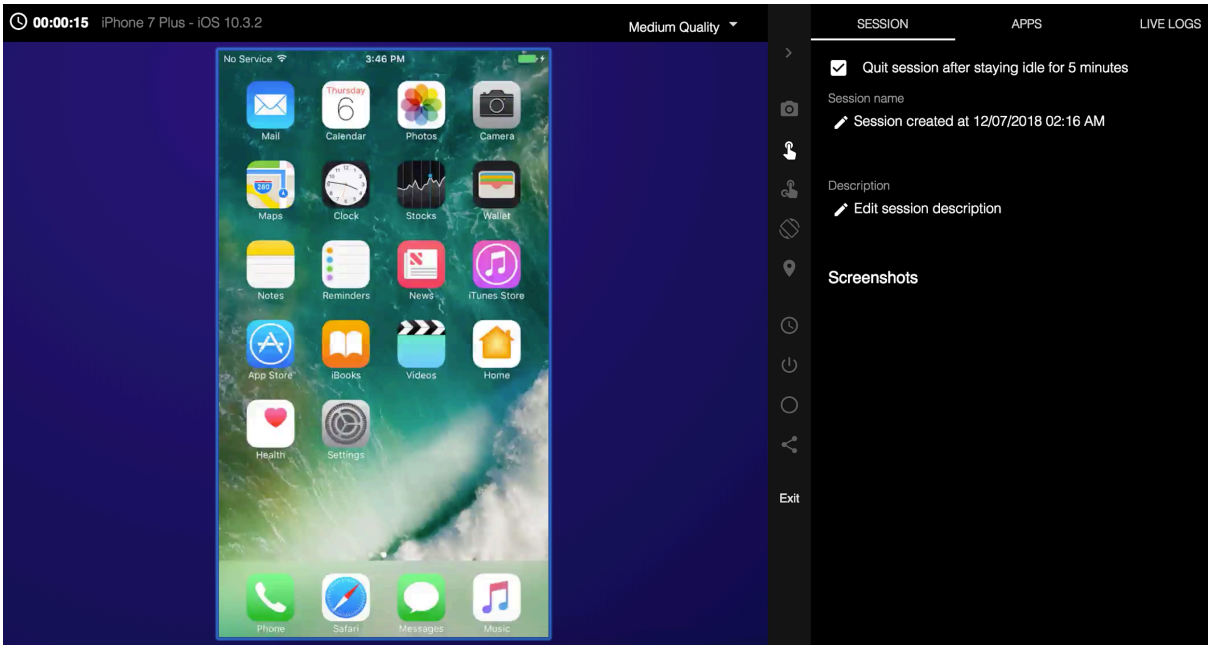

Figure-27: Manual testing on device.

Kobiton offers many features including the ability to combine your own on-premises devices with the cloud devices. A detailed review of all the Kobiton capabilities are beyond the scope of this book but if you visit https://docs.kobiton.com/ you can find lots of additional information and services offered by Kobiton.

For more information about Automation testing with Kobiton visit:

https://docs.kobiton.com/automation-testing/using-kobiton-for-automation-testing/

And for Manual testing with Kobiton you can find good documentation here:

https://docs.kobiton.com/manual-testing/overview/

Real-device testing should be a key part of your quality process. Fortunately this is made easy by cloud device providers. Other providers apart from Kobiton include Perfecto, Browserstack and Saucelabs.

Chapter-11: Automating Gestures

Up until now we have looked into basic Appium automation, such as finding and clicking on a button or typing text into a text field. However, "real world" mobile applications are more sophisticated and contain many complex UI elements that require user interactions such as double tap, long press, swipe left/right, pull up/down and even multi-touch actions.

Appium supports the following gestures:

- Tap on an element.
- Tap on x, y coordinates.
- Press an element for a particular duration.
- Press x, y coordinates for a particular duration.
- Horizontal swipe: Using start and end percentage of the screen height and width.
- Vertical swipe: Using start and end percentage of the screen height and width.
- Drag(Swipe) one element to another element.
- Multitouch for an element.

Appium supports these gestures using the **TouchActions** class.

```
TouchAction touchAction = new TouchAction(driver);
```

Some of the supported methods are:[5]

Method Name	Purpose
press(PointOption pressOptions)	Press action on the screen.
longPress(LongPressOptions longPressOptions)	Press and hold the at the center of an element until the context menu event has fired.
tap(PointOption tapOptions)	Tap on a position.
moveTo(PointOption moveToOptions)	Moves current touch to a new position.

[5] *Official Appium API docs from the github page:*

cancel()	Cancel this action, if it was partially completed by the performsTouchActions.
perform()	Perform this chain of actions on the performsTouchActions.

Before exploring each mentioned action we need to understand the significance of `perform()` as it plays a vital role. The Appium client simply records all the instructions and actions on the client side and stores the intermediate values in a local data structure. The `perform()` method is used to send all actions to the appium server - as soon as `perform()` is called, the intermediate actions and instructions are converted to JSON and sent to the appium server, and then the actual action is being performed. So for any gesture code the last method called would be `perform()`

Note: *This is a common omission during development, forgetting to call `perform()` and wondering why your test isn't working!*

Appium fully supports native applications. So if the application is designed natively for the platform (iOS or Android), then you can easily find the unique selectors for automation, but there are case when you use cross-platform development technologies such as react native, ionic or xamarin framework. In this instance, sometime elements are not present for a particular screen or even a whole application.

For example, most games are coded using the Unity3D platform rather than native coding, so there would not be a single element that can be located by any tool or even by the appium inspector. However we are not talking about game automation right now.

The main takeaway here is that if you are not able to get the selector for any element for any reason then only one survival option remains. Which is to get the x, y coordinates for that element.

NOTE: *Please remember that you can only click on that element using appium.*

Now the question is how can you get the x,y coordinate?

- It depends...
- Because ***you can get the Pointer location in Android*** but ***you can not get it in iOS devices***.

Getting the pointer location in Android:

1) Move to Settings > Developer options
2) Enable the Pointer location.
3) Now move to any application for which you need the coordinates of a particular location. Tap on the location and you will get the coordinates for that spot at the top of the screen.

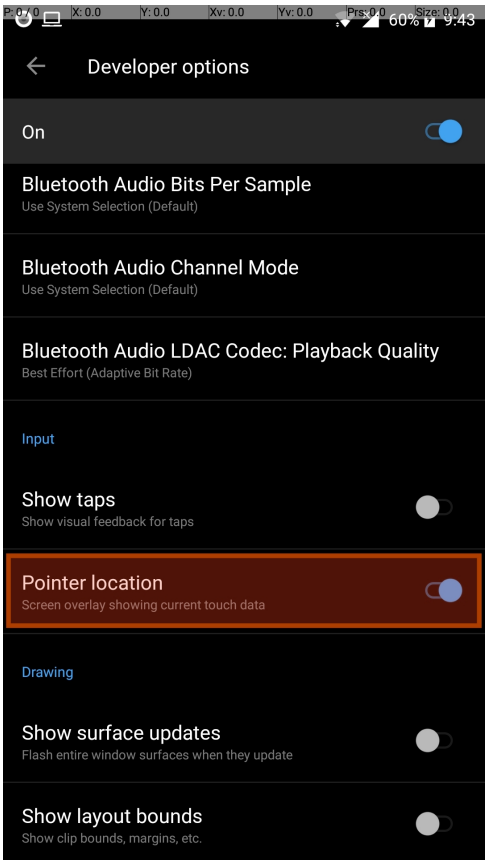

Figure-1: Enable the Pointer location.

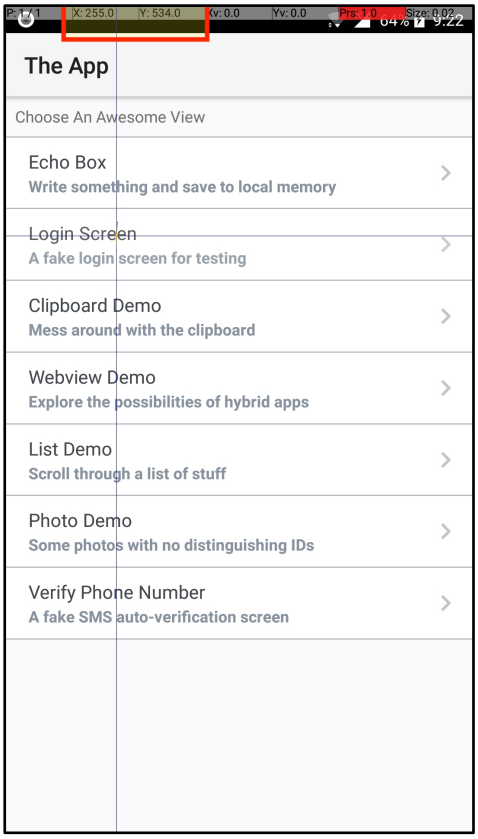

Figure-2: Get coordinates.

Getting the pointer location in iOS:

iOS does not support the pointer location and there aren't even any third party apps or tools which come to the rescue. Therefore you need to calculate it using screen resolution and a little bit of prediction. In case you don't get success at first, you can use trial and error to get the needed location.

Now let's look into each gesture one by one:

1) Tap on element

Method: `tap(TapOptions tapOptions)`

Usage: It is the simplest action, as the name suggests it will simply click/tap on a particular location. It is a combination of `press()` and `release()`

Example:

```
TouchAction touchAction = new TouchAction(driver);
touchAction.tap(tapOptions()
          .withElement(element(androidElement)))
          .perform()
```

NOTE: *Here you can also put the wait along with the tap action, for example:*

```
new TouchAction(driver)
.tap(tapOptions().withElement(element(androidElement)))
.waitAction(waitOptions(Duration.ofMillis(millis)))
.perform();
```

2) Tap on x, y coordinates

Method: `tap(PointOption pointOptions)`

Usage: It is used to tap on a particular x,y coordinate point.

Example:

```
TouchAction touchAction = new TouchAction(driver);
touchAction.tap(PointOption.point(1280, 1013))
.perform()
```

NOTE: *Similar like Tap on element you can put the wait along with the tap action, for example:*

```
new TouchAction(driver)
.tap(point(x, y))
.waitAction(waitOptions(Duration.ofMillis(millis)))
.perform();
```

3) Press an element for a particular duration

Method: `press(PointOption pressOptions)`

Usage: It is used to apply the press action. After the press action you also need to release so that the state would be in press mode. You do so by calling the `release()` function after calling `press().`

Example:

```
TouchAction touchAction = new TouchAction(driver);
touchAction.press(element(element))
.waitAction(waitOptions(ofSeconds(seconds)))
.release()
```

```
.perform();
```

4) Press x, y coordinates for a particular duration

Method: `press(PointOption pressOptions)`

Usage: Similar to Press(ing) an element for a particular duration, here you just need to pass x, y coordinates instead of an element and don't forget to call the `release()` function after calling `press().`

Example:

```
TouchAction touchAction = new TouchAction(driver);
touchAction.press(point(x,y))
.waitAction(waitOptions(ofSeconds(seconds)))
.release()
.perform();
```

Automating swipe actions

Before we look into the Horizontal swipe let's understand how we can automate

swipe actions generally.

Swiping is a combination of tapping + moving actions. Appium does not provide a direct method for swiping, so you need to combine a few methods in order to achieve swiping. For example if you want to perform swiping then first you have to press on a particular point and then specify the particular amount of time during which you want to perform the swiping action and at last you to move to another point - and don't forget to call the release method which used to release all the actions. So it's actually simple: first press -> wait(duration of swiping) -> move to (moveTo()) particular location.

You might be wondering why can't we directly use moveTo() ?

If you recall, using the press method requires you to eventually call the release method. So we are basically mimicking a swipe by entering the press state, moving to a location, and THEN releasing.

Swiping can have an up/down/right/left direction so you need to apply the right logic and have to provide the x, y coordinates for the press() and

moveTo() methods.

And also please note these appium methods to get the device screen measurements:

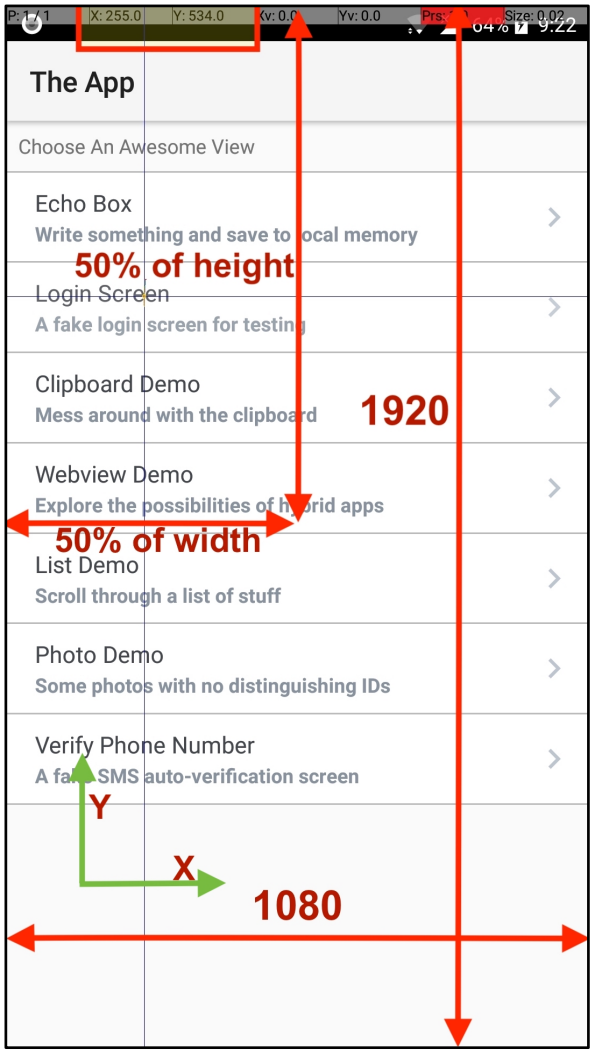

Figure-3: Screen Measurements.

```
int heightOfScreen =
driver.manage().window().getSize().getHeight();
int widthOfScreen =
driver.manage().window().getSize().getWidth();

int middleHeightOfScreen = heightOfScreen/2;

// To get 50% of width
int x = widthOfScreen * 0.5;

// To get 50% of height
int y = heightOfScreen * 0.5;
```

Here Width → X and Height → Y coordinates.

5) Horizontal swipe: Using start and end percentage of the screen height and width

Method and Usage: As we discussed above there is no particular method for Horizontal swipe, and you need to perform the combination of press()->wait()->moveTo(). The `moveTo()` method is new to us - it is used to move to particular location. Its syntax is: `moveTo(PointOption pressOptions)`

The secret to moveTo lies in the coordinates - you need to mention the starting and ending x,y coordinates in such a way that swiping can done from left → right OR right → left direction. Note that when we say swipe right, we mean moving the content from right to left, but the physical gesture is is moving to the left. See figure 4 below for clarification.

Example: Let say we want to swipe right on the screen, so in practice you need to press on the right side and, without taking your finger off of the device, move your finger to left side. So we need to move from in the Right to Left direction in order to make the Right Swipe.

Referring to figure 4 below, we have the swiping UI element placed on the screen from location (0,360) to (1080, 780). Now in order to attempt swipe in the right direction you have to first press anywhere in swipe area, for example let's say (972, 500), and now without taking away the press action you need to move to left side suppose (108, 500) [Please note that Y coordinate is constant as we just need to change the X coordinates for swiping]. At that point we have achieved the swipe and now we can able to release the action and at last call the perform method to send all commands to the Appium server to perform on the UI.

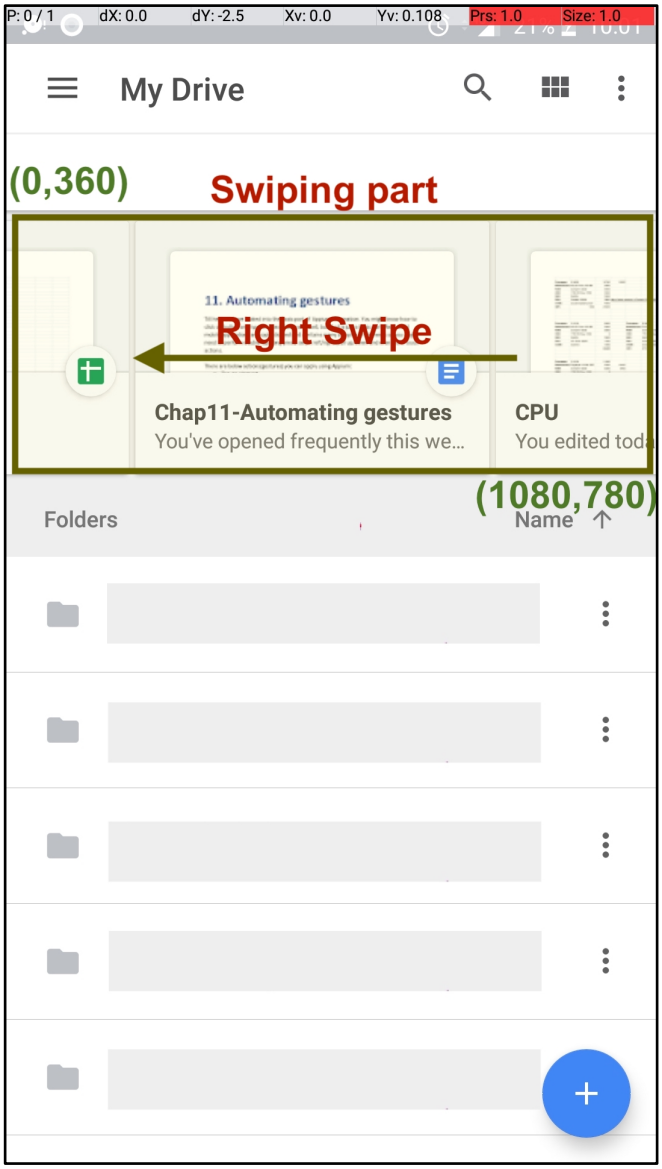

Figure-4: Swiping in action.

This is just one scenario for achieving a swipe gesture. Ideally first we need to make a decision as to what is the "right side of the screen". We do this by considering 90% of the screen width. For example, if the screen resolution is 1920 x 1080, 1080 is the width of Screen and 90% of that width would equals to 972, so we have got our X coordinate for what we consider the "Right" side. In a similar manner we will need the X coordinate for the Left side and this time we can consider 10% of the width which would give us an X coordinate of about 108. So we have got X coordinates for Left and Right direction. For the Y coordinate we can choose any value as long as it falls in the swiping area - for example, let's say our swiping area is between (0, 360) to (1080, 780), so you can choose any value for the Y coordinate in between 360 to 780.

NOTE: *It is important that the Y coordinate have same value because we are*

focusing on only swiping (not scrolling) so only the X coordinate will change during the process and Y will remain constant. Ideally you should choose the half height of swiping area for Y coordinate.

Finally, you can perform the swipe gesture:

```
TouchAction swipe = new TouchAction(driver)
.press(PointOption.point(972,500))
.waitAction(waitOptions(ofMillis(800)))
.moveTo(PointOption.point(108,500))
.release()
.perform();
```

Same way if you want to swipe in the left direction you have to first press on the left side and move to the right side.

NOTE: *This method works in a similar way on Android and iOS but the location differs according to Mobile device being used (given the swipe is dependent on coordinates which is dependent on screen resolution). Moreover on the iOS side you can't find the location directly so you will need to use a trial and error approach.*

6) Vertical swipe(scroll): Using start and end percentage of the screen height and width

Usage: Scroll is the same as swipe but the direction is different. In swiping we are dealing with horizontal direction where as in scrolling we are dealing with vertical direction - but the rest of the logic will remain the same.

Scrolling can done in the up → down OR down → up direction.

Example: Let's look at scrolling Down. On a mobile device in order to scroll down on screen we swipe in the "down to up" direction. It's actually 3 steps we need to complete:

1) **Find the swiping area.**
    - Starting point = (0,360)
    - Ending point = (1080,1920)

2) **Mark the scrolling points** (We will use the height from scrolling area

only. As per below image, the scrolling area is starting from approximately 30% of the screen height and ending at the end of the screen).

- Down area point:
  - i) X = Middle of Screen= 0.5 x 1080 = 540
    (This will be same for starting and ending location)
  - ii) Y = 95% height of Screen = 0.95 x 1920 = 1824.

  Location = (540,1824)
- Up area point:
  - i) X = Middle of Screen=540
  - ii) Y = 35% height of Screen = 0.35 x 1920 = 672(Percentage value must be >30%).

  Location = (540,672)

3) Perform scroll action using Appium.

```
TouchAction swipe = new TouchAction(driver)
.press(PointOption.point(540,1824))
.waitAction(waitOptions(ofMillis(800)))
.moveTo(PointOption.point(540,672))
.release()
.perform();
```

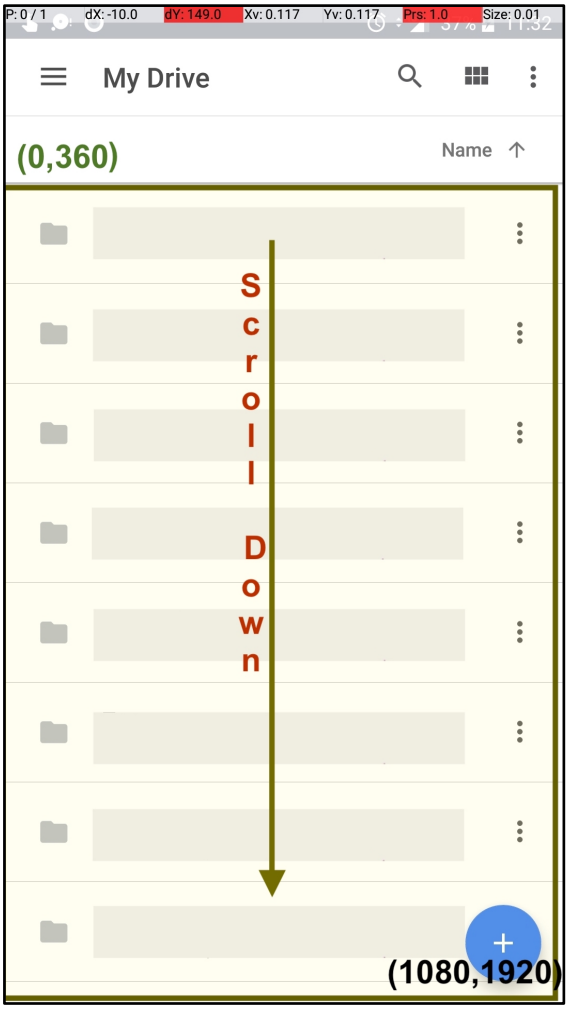

Figure-5: Scrolling in action.

Scroll up will work the same but with different location points.

7) Drag(swipe) one element to an another element

Dragging one element to another element is one kind of swiping action.

But here location in coordinates would not matter as we have both of the elements(1. Element which needs to be dragged, 2. Element upon which another element will be dragged).

```
TouchAction swipe = new TouchAction(driver)
.press(ElementOption.element(element1))
.waitAction(waitOptions(ofSeconds(2)))
.moveTo(ElementOption.element(element2))
.release()
.perform();
```

8) MultiTouch

As the name suggests it means multiple touches happening at the same time.

For example on iOS if you want to move to the Main screen, you need to use 5 fingers and do a swipe.

Multi Touch is handled by the `MultiTouchAction` class. It has a `add(TouchActions touchActions)` method so in which we need to pass a TouchActions object.

So let say you want to press on 5 different points at a time then first you need to create 5 TouchActions, but here the important thing is we are not having a perform method at the end. We just need to call the release method for the TouchAction object, and then pass those values into the add method of the MultiTouchAction class.

You can perform Multi Touch for:

1) **Multiple touches at a time.**

```
TouchAction touchActionOne = new TouchAction();
touchActionOne.press(PointOption.point(100, 100));
touchActionOne.release();

TouchAction touchActionTwo = new TouchAction();
touchActionTwo.press(PointOption.point(200, 200));
touchActionTwo.release();

MultiTouchAction action = new MultiTouchAction();
action.add(touchActionOne);
action.add(touchActionTwo);
action.perform();
```

OR if you want to perform multi touch on particular elements then use the below code snippet.

```
TouchAction touchAction1 = new TouchAction(driver)
.tap(ElementOption.element(e1))
.release();

TouchAction touchAction2 = new TouchAction(driver)
.tap(ElementOption.element(e2))
.release();
```

```
MultiTouchAction action = new MultiTouchAction();
action.add(touchAction1);
action.add(touchAction2);
action.perform();
```

2) Swiping using multiple fingers.

```
TouchAction touchActionOne = new TouchAction();
touchActionOne.press(PointOption.point(100, 100));
touchActionOne.moveTo(PointOption.point(500, 100));
touchActionOne.release();

TouchAction touchActionTwo = new TouchAction();
touchActionTwo.press(PointOption.point(100, 200));
touchActionTwo.moveTo(PointOption.point(500, 100));
touchActionTwo.release();

MultiTouchAction action = new MultiTouchAction();
action.add(touchActionOne);
action.add(touchActionTwo);
action.perform();
```

NOTE: *As mentioned earlier only* `MultiTouchAction` *should call the perform() method at the end. For* `TouchActions` *, the perform() method should not be called otherwise instructions will be sent to the Appium server and the click will happen before the Multi Touch action.*

In this chapter we have looked into the most used scenarios in the Appium world. These methods all work on both Android and iOS.

More details about all the different `TouchAction` methods can be found on the official appium docs on the github project.

Chapter-12: Appium Tips and Tricks

You've made it this far! By now you're a full-fledged Appium ninja. It's time to round out some of your new skills with additional tips and tricks.

Appium is a sophisticated and ever updating testing platform, and with a growing user base forming, you're bound to see some cool tricks by monitoring discussion boards and blogs.

In this chapter, we'll look at some practical tips that you can put into practice immediately to improve your test case authoring. Let's get started!

1) How to check whether an Android app is already installed or not?

There are scenarios in which you want to check out whether your Android app is already installed on the device or not, *but why would you need that?*

A very simple (but common!) use case requires testing a part of the app that is only accessible to logged-in users. So every time you execute your test case, the first step would be to login which is a time consuming operation (from an execution perspective).

We can skip the login if we've already logged in on the previous test execution, as it takes unnecessary time. We can do 2 things:

1) Set the desiredcapabilities for **"noReset"** as true and **"fullReset"** as false. So if your app has already been installed, Appium neither uninstalls it nor clear the cache data. It will simply open the app every time, So login will take place only at once, when you open the app first time after installing it.

2) To avoid reinstalling the app every time, fire `$ adb shell pm list packages` on your terminal - it lists all the packages of installed apps. You can create test logic like if the app you want to automate is displayed on that list then you can skip the installation.

Now the question is how would you fire this command within your programming language?

In Java this is done using the `ProcessBuilder` class:

```
String line;
Process p = Runtime.getRuntime().exec("adb shell pm list
packages");
BufferedReader input = new BufferedReader(new
InputStreamReader(p.getInputStream()));
while ((line = input.readLine()) != null) {
    System.out.println(line);
}
input.close();
```

2) How to enable mouse pointer location on Android at runtime ?

As we discussed in an earlier chapter, if there is no unique locator assigned to a UI element then the only option is to tap on a particular point. By enabling “pointer location” in developer options in Android we can get the x, y coordinates for any point.

Moreover having the mouse pointer location helps so much in debugging, especially whenever you are dealing with swipe, touch and scroll functions in Appium.

But what if you are dealing with remote devices(such as Kobiton or BrowserStack devices)?

There are case in which you cannot access the remote device before executing the automation test cases. So you will need some way to enable the pointer location on that devices at run time.

If you want to enable mouse pointer location using the terminal then fire command:

`$ adb shell settings put system pointer_location 1` [Use 0 for disable]

In Java you can use this code:

```
public static void main(String[] args) throws IOException,
InterruptedException {
    ProcessBuilder pb = new ProcessBuilder("adb", "shell",
"settings", "put", "system", "pointer_location", "1");
```

```
    Process pc = pb.start();
    pc.waitFor();
    System.out.println("Finish!");
}
```

You can find this example on our github page on our github page:

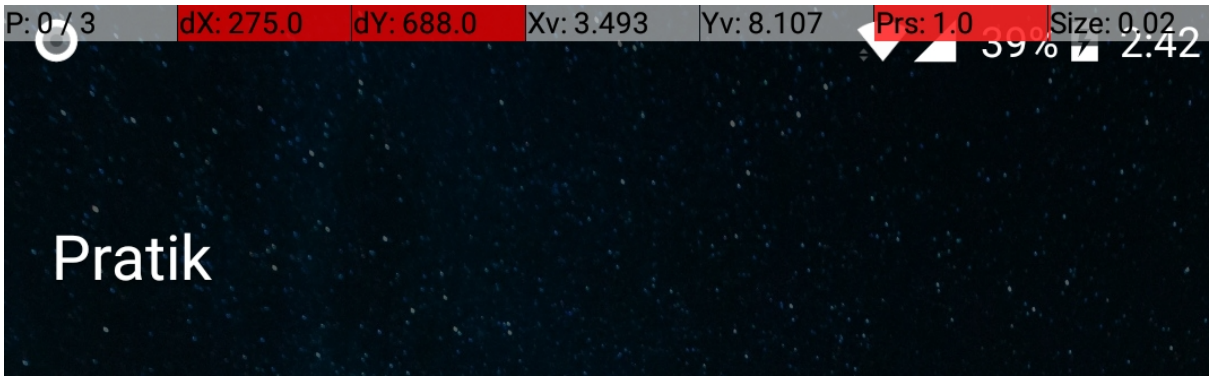

Figure-1: Android Point Location Enabled.

3) How to capture Screenshots On Test Failure?

This is the most vital feature of Appium (and Selenium). Practically speaking, with automation you are not observing the execution of all test cases, so in case a failure occurs you won't have the exact details of the failure. That's why this feature comes in handy, because it will capture the screenshot whenever a failure occurs. And by looking at the screenshot sometimes you can quickly figure out the failure cause without taking a look at the error log.

Note: *Cloud device platforms like Kobiton offer the ability to automatically take not only screenshots of every action, but also record full video of your test sessions as well as detailed Appium and device logs.*

Below is the code you use to capture a screenshot:

```
private void takeLocalScreenshot(String imageName) throws
IOException {
    File scrFile = ((TakesScreenshot)driver).
getScreenshotAs(OutputType.FILE);
    FileUtils.copyFile(scrFile, new
File("failureScreenshots/" + imageName + ".png"));
}
```

But we need to call this method only if test cases are failing or being skipped. Thus we need to check the status of the test case, and, as we are using the

TestNG test framework, we can use its methods to check the status of the executing test case.

```
@AfterMethod(alwaysRun = true)
public void afterMethod(final ITestResult result) throws
IOException {
if (result.getStatus() == ITestResult.FAILURE ||
result.getStatus() == ITestResult.SKIP) {
takeScreenshot(result.getMethod().getMethodName().toLowerCase
() + "_" + System.currentTimeMillis());
}
```

This code example works for both iOS and Android. The code is implemented in the github repository ***BaseTest.java***

4) How to dismiss dialogs/alerts automatically ?

In mobile application notifications, popups and dialogs occur at random so in order to access the app we must handle that immediately as soon as they appear.

Appium can handle the system dialogs/alerts in 2 ways:

1) **Manually:** In this approach you have to find the locators of the allow/deny button of the element and then perform a `click()` action on it.
2) **Automatically:** You can set the desired capabilities to auto accept or deny the alert/dialogs.

Auto Accept the Alerts:

```
capabilities.SetCapability("autoAcceptAlerts", true);
```

Auto Dismiss the Alerts:

```
capabilities.SetCapability("autoDismissAlerts", true);
```

In some scenarios when you are not able to get the dialog elements, you can leverage the Appium image comparison feature and find the element by image. We will look more into that feature in a subsequent chapter.

5) How to handle notifications in Android?

Push notification assertion is a common exercise you need to be familiar with because many apps send a push notification while you are accessing the app.

Appium provides a super easy way to open notifications - you just have to call the `openNotifications()` method.

You can get the title of the notification using `android:id/title` and the content of the notification using `android:id/text` locators.

For example:

```
driver.openNotifications();
List<AndroidElement> titleElement =
driver.findElements(By.id("android:id/title"));
List<AndroidElement> contentElement =
driver.findElements(By.id("android:id/text"));

for (int i = 0; i < titleElement.size(); i++) {
    System.out.println(titleElement.get(i).getText() + " : "
+ contentElement.get(i).getText());
}
```

See ***AndroidTricks.java*** on our github repository.

NOTE: *`driver.openNotifications()` method only applies to AndroidDriver object. It is not available for IOSDriver.*

6) How to make test cases fail fast in order to quickly get an error message?

As strange as the name of this trick may sound, there are cases when you want to fail your test cases faster in order to get the error message to fix the test case.

Using the `newCommandTimeout` desired capability you can specify the time in seconds for which Appium will wait for a new command from the client before assuming the client quit and ending the session.

```
capabilities.setCapability("newCommandTimeout", 15);
```

If you are executing the test cases locally you want them to fail quickly, so ideally you should wait for 10 to 20 seconds (depends on app) because during that time app will load all the resources. With remote devices, due to network latency and other reasons, apps on remote device may take more time to load the resources so for remote execution that time period should be closer to 60 seconds.

7) How to handle the hide_keyboard() method?

Essentially, it's as simple as calling `hide_keyboard()`. However, the `hide_keyboard()` method works differently on iOS and Android because the internal architecture of the soft keyboard on both platforms is not the same. It also depends on the physical device model and its operating system version. Therefore, standard practice is you should put the `hide_keyboard()` method in a try/catch block.

```
try {
driver.hide_keyboard();
} catch(Exception e) {}
```

8) How can you write test cases faster?

Unfortunately QA engineers are facing increasing pressure to deliver more, faster. In this Agile world, requirements and releases are changing faster than ever which means your Application Under Test is constantly changing.

And with continuous deployment, there is a good chance your test case becomes obsolete before that glorious bug is discovered.

The following tips will help you to develop test cases at a faster pace:

- **Use a local appium server and local device:** Appium performs the best with a locally installed appium server and having the script execute on a physically connected real device. Save the cloud device testing for increased coverage testing and for full regression tests.

- **Extract all the UI element locators of the application at first go:** You can simply navigate the whole app with Appium Inspector (or another tool) and note down all relevant Ids, text or class name for the UI elements. The objective here is to save time by getting all the element locators at first go, so you won't have to find the locator of element when you are in the middle of writing your test case.

- **Communicate with developers and assign a valid, and unique, ID to UI elements:** This is the most common scenario where no unique id, class name or text is assigned to a UI element so you have to use the XPath locator which might contain the index. But relying on XPath indexes is brutal especially when your application under test is under development, because minor changes in the UI can change the index of all elements. As a point of cooperation between QA and Development, a naming convention or best practice should exist the ensure unique IDs are used. Whenever you find yourself using an

XPath, stop to ask why and see if you can have the dev team provide unique IDs instead.

10) How to handle to mobile data, wifi and airplane mode in Android?

In order to automate various connectivities such as mobile data, wifi and airplane mode, Appium provides a `ConnectionState` class for setting and getting the network connection for a connected android device/emulator.

For example if you want to turn on only wifi(Not mobile data and airplane mode), you can use this command:

```
ConnectionState state = driver.setConnection(new
ConnectionStateBuilder().withWiFiEnabled().build());
```

You can see the example on our github page.

This API works based on the android device OS versions so please go through the official appium documentation in order to get more information.

NOTE: *The above mentioned APIs are not available for iOS.*

11) How to switch context?

There are two main types of context in Appium:

1) WEBVIEW
2) NATIVE

While we are working with a native application, the context will be NATIVE. And when webview is being used on some screen it will have the WEBVIEW context. Sometimes you actually have both in a single application. You might have experienced the webview when you are dealing with payments in an application- generally the payments page is integrated the in form of webview (being provided by the payment gateway). So in that case you need to change the application context from NATIVE to WEBVIEW in order to get control of WEBVIEW elements.

Using the below code you can get the current context of your app:

```
String context = driver.getContext();
```

In a similar way you can get the all contexts available to automate using:

```
Set<String> contextNames = driver.getContextHandles();
```

And using this code you can change the context to WEBVIEW:

```
driver.context("WEBVIEW");
```

You can use the this method in order to change the context to WEBVIEW if it is found:

```
public void changeDriverContextToWebView(AppiumDriver driver)
{
          Set<String> contextHandles =
     driver.getContextHandles();
          for (String name: contextHandles) {
              if (name.equals("WEBVIEW"))
                  driver.context(name);
        }
}
```

12) How can you minimize and reopen the app again?

There are some scenarios where you need to minimize the application and need to re open it without killing the current session. Using the `runAppInBackground(Duration time)` method you can hide your application for a particular duration.
For example if you want to minimize application for 5 seconds then you might use:

```
driver.runAppInBackground(Duration.ofSeconds(5));
```

13) How to start Appium Server programmatically?

In all our examples up until now we have presumed that you have started the appium server explicitly using the command line or using the Appium Desktop application.

Now when you are thinking to integrate appium UI test cases with Jenkins (or generally within in a CI/CD pipeline) you have two options to start the Jenkins server:

1) Specify the command(`$appium&`) in the build section of Jenkins to run the appium server.
2) Start the Appium server by writing the code in the test framework in such a way that the server starts before the tests execute, and quits automatically at the end of the execution.

Here, option 1 is not advisable because if a test failure occurs then the Appium server will be continuously running and occupying memory of the physical server. While in option 2 we have server quit code written at the end of execution, so even if failure occurs it will quit the Appium server and free the space in memory.

You can start the server using 2 classes:

1) `AppiumDriverLocalService`: It is simply used to start and stop the appium server.
2) `AppiumServiceBuilder`: This class is used to build the appium service, here you can specify the appium server url, port, desired capabilities and some other parameters. This is recommended if you really want to customize the server details.

So let's look into the example of both classes:

AppiumDriverLocalService:

```
AppiumDriverLocalService appiumDriverLocalService =
AppiumDriverLocalService.buildDefaultService();

public void setUpPage() throws IOException
{
    AppiumDriverLocalService appiumDriverLocalService =
AppiumDriverLocalService.buildDefaultService();
    appiumDriverLocalService.start();
}

public void tearDownAppium()
{
    super.tearDownAppium();
    appiumDriverLocalService.stop();
}
```

In the above example Appium will start the server using default url and port. You can find this example at our github project.

AppiumServiceBuilder:

```
AppiumServiceBuilder builder;
AppiumDriverLocalService appiumDriverLocalService;

public DesiredCapabilities getDesiredCapabilities()
{
   DesiredCapabilities desiredCapabilities = new
DesiredCapabilities();
      ...
          return desiredCapabilities;
}

public void startServer() throws IOException {
    DesiredCapabilities desiredCapabilities =
getDesiredCapabilities();
    AppiumServiceBuilder builder = new
      AppiumServiceBuilder();
    builder.withIPAddress("127.0.0.1");
    builder.usingPort(4729);
    builder.withCapabilities(desiredCapabilities);
    builder.withArgument(GeneralServerFlag.SESSION_OVERRIDE);
    builder.withArgument(GeneralServerFlag.LOG_LEVEL,
      "error");
    appiumDriverLocalService =
      AppiumDriverLocalService.buildService(builder);
    appiumDriverLocalService.start();
}

public void stopServer() {
    appiumDriverLocalService.stop();
}
```

Here as you can see appium will use Port: 4729, URL: 127.0.0.1 with some Desired Capabilities and flags.

You can find this code at out github page.

Hopefully you found (or will find!) these tips and tricks useful as you progress your journey of being an Appium expert. The Appium community is constantly expanding and you'll find tremendous tips and tricks shared by other developers and testers. Keep reading forums and blogs to keep your knowledge up to date.

Jonathan Lipps, the project lead of Appium, has created an amazing blog at https://appiumpro.com in which he has compiled some of the best practices and tricks to use Appium in an effective way. We strongly recommend you visit this blog and subscribe to the helpful mailing list to continue learning interesting things about Appium.

Chapter-13: Image Comparison Using Appium

The mass advancement in machine learning and artificial intelligence is affecting every sector of every industry, and test automation is no exception.

AI is being used in multiple areas of software testing, including:

1) **Visually automated testing:** Drive testing through the UI by using Image Comparison
2) **Automated API Testing:** Using machine learning algorithms we can analyze the API calls in more effective ways
3) **Test Coverage:** Knowing what to test is a science unto itsel - sometimes a small change has a large impact or vice versa. Using AI tools you can know what areas of the app changed based on source code analysis and which test cases should be executed (or updated)
4) **Self-Healing Test Scripts:** The most common scenario in automation test cases is test case failure due to locator changes. In order to fix that we need to find the valid locator again which is time-consuming and happens frequently. Machine learning/AI algorithms can learn and observe the changes of an application's domain object modeling structure and can automatically suggest the new locators to use.
5) **Automatically write the test cases:** As AI keeps improving, we will see an increased ability for test cases to be automatically created, based either on self exploring or by user observation.

Most of the AI solutions are being developed by 3rd party vendors and not within the Appium framework directly. However, the Appium community recently introduced the Image comparison feature which is great for testing at the UI level and comparing images, which makes our test scripts less brittle. Essentially, we get the benefit of a new Image locator strategy we can use in our test scripts.

The feature was developed by incorporating OpenCV, one of the leading image comparison libraries.

This chapter is split into 2 sections:

1) **Setup and Linking OpenCV with Appium**
2) **Using the Image comparison feature in automation**

Setup and Linking OpenCV with Appium

In order to link the OpenCV library to Appium, we must install the Appium CLI. This is 3 step process:

1) Install the Appium CLI.
2) Install the OpenCV library.
3) Link the OpenCV library with Appium.

So let's look into all options one-by-one.

1) Install Appium CLI

1) Make sure you have installed node and npm. If you have brew installed on your mac then you can just execute: `$ brew install node` to install node.js along with npm.

2) Install appium: `$ npm install appium`

3) Verify that appium is installed correctly using the command: `$ appium -v`

4) Also check the path where Appium is installed: `$ which appium` and move to that path.

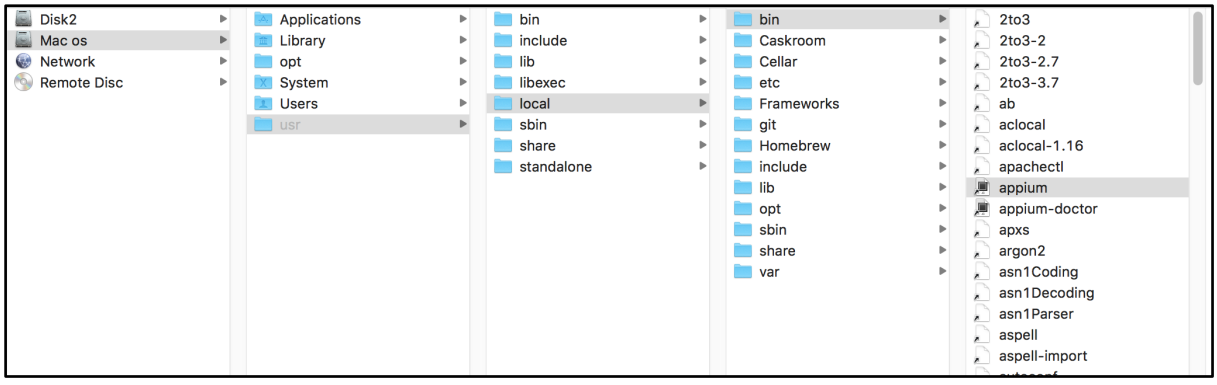

Figure-1: Appium location.

5) Now find the actual path of the appium binary and move to it.

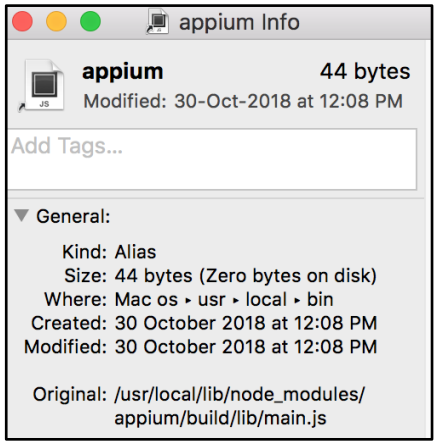

Figure-2: Appium binary location.

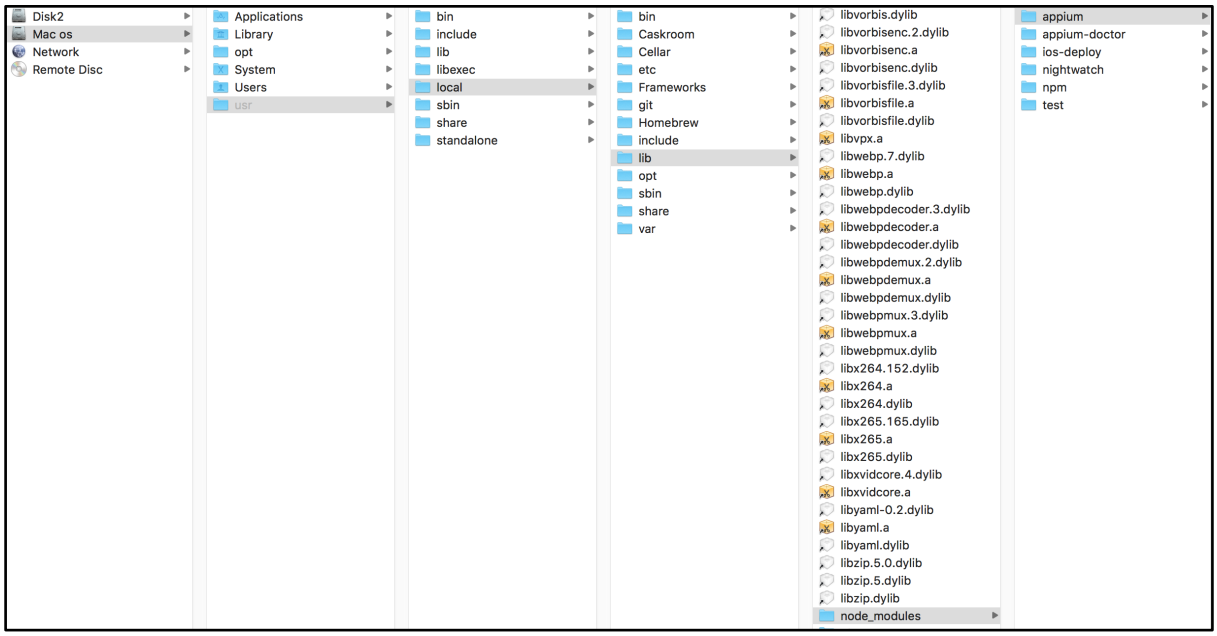

Figure-3: Appium Node module.

As you can see in the above screenshot all globally defined node modules are present under ***/usr/local/lib/node_modules*** location, so you have to install the OpenCV library at this global location.

2) Install the OpenCV library

1) As we discussed previously we need to install the OpenCV module globally so we will use the **-g** flag. Use this command to install the OpenCV library for node: `$ npm i -g opencv4nodejs`

   After successful installation you can find the `opencv4nodejs` library under the same global location(***/usr/local/lib/node_modules***).

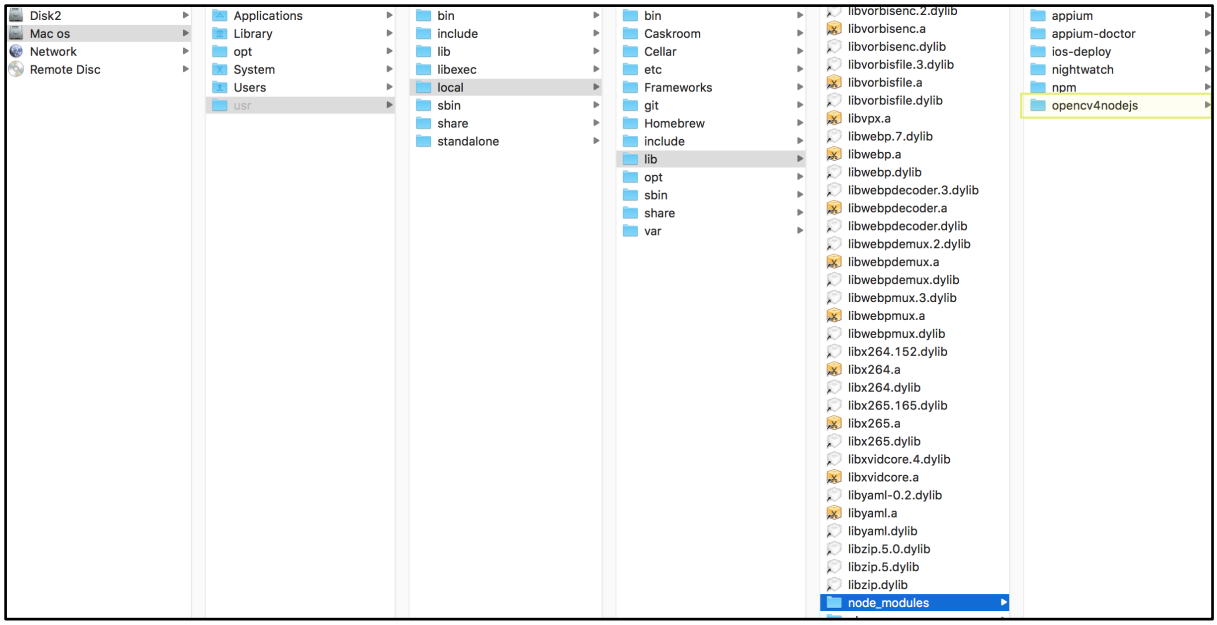

Figure-4: opencv4nodejs Node module.

3) Link the OpenCV library with Appium

1) Now we need to link this module with Appium, using:
   `$ npm link opencv4nodejs`

   NOTE: *You can also install the opencv4nodejs package inside the appium > node_modules directory instead of linking.*

2) Now move to the `appium` node module and move to the **node_modules** directory, there you will find that `opencv4nodejs` is linked.

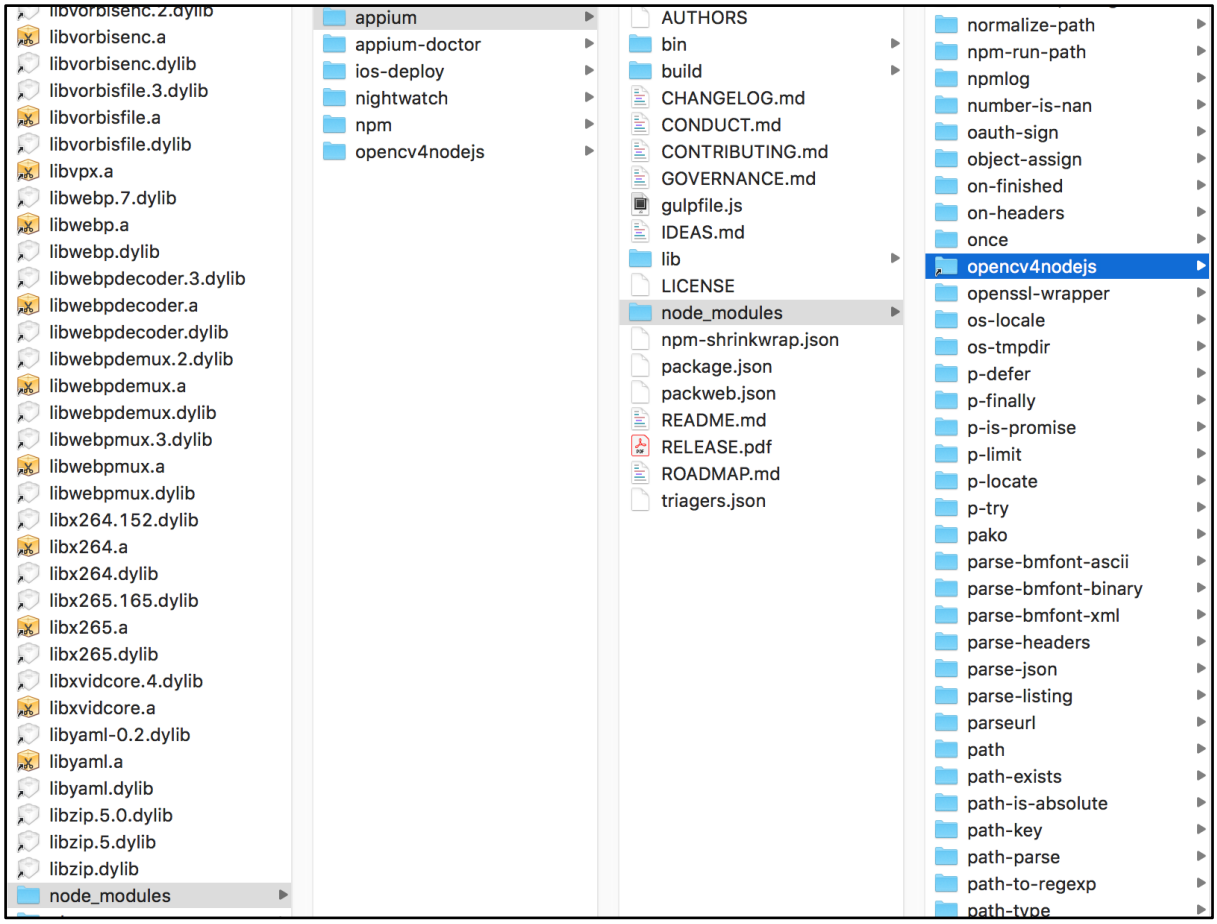

Figure-5: opencv4nodejs module is linked with appium.

That's it, now you are ready to use Image comparison feature in Appium.

NOTE: *You can also directly install the* `opencv4nodejs` *node module under Appium node_modules, the steps would be:*

1) Move to appium directory: `$ cd /usr/local/lib/node_modules/appium`

2) Install opencv4nodejs: `$ npm install opencv4nodejs`

Using the Image comparison feature in automation

Why do we need the image comparison feature in appium?

There are cases when in the application there is no unique locator present for a particular UI element, and in that case, you can't do anything except tapping at a particular location which is a very fragile operation (that is, prone to failure). It is workable only if the location of that UI element is static every time - if the UI element is changing its location every time you open your application or based on the device it's executing on, then tapping on a particular location isn't the preferred

option. With the new image comparison feature inside Appium, we have an image locator strategy. Using this locator we locate elements based on their image.

With the image locator strategy, instead of typical unique locators, you need to pass the string which is the Base64 encoded format of the image.

There are many ways to get the Base64 encoded version of the Image, but we use the simplest.

Use this method to convert an image(.png/.jpg file) to Base64 String format.

```
public String getReferenceImageB64(String imgPath) throws
URISyntaxException, IOException {
    URL refImgUrl =
getClass().getClassLoader().getResource(imgPath);
    File refImgFile = Paths.get(refImgUrl.toURI()).toFile();
    return
Base64.getEncoder().encodeToString(Files.readAllBytes(refImgF
ile.toPath()));
}
```

Now that we have the image encoded as Base64, we can locate elements by this image.

Image Locator Strategy:

```
MobileElement elementByImage = (MobileElement)
driver.findElementByImage(refImageBase64);
```

NOTE: *Just as with the other locator strategies we can perform actions like* `click()`, `getText()`, `sendKeys()` *etc…*

Now to understand this better let's take one practical automation test case which uses the image locator strategy.

Image Comparison Automation Test Case

We have taken one simple image application into consideration in order to understand the image comparison feature of Appium. That application will display the image from the given image URL.

Now the main problem is that the DOM structure contains no details for the image view, so we won't be able to verify whether image view is visible or not after clicking on submit. Moreover even if we verify that image view is visible it won't give us confidence that the displayed images are being fetched from given image URL. So in order to check this use case, we need to use Appium's image comparison functionality.

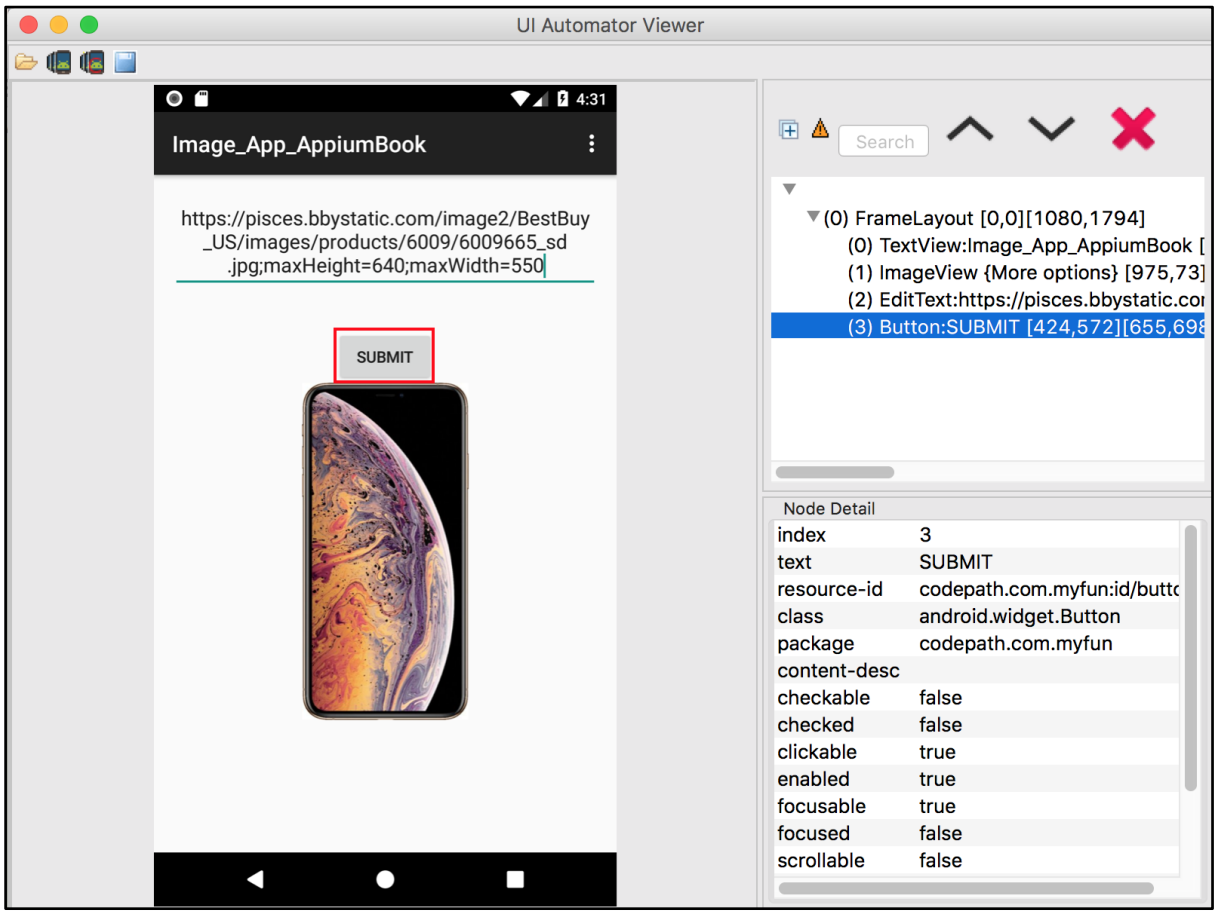

Figure-6: ImageView is not available for the appeared image.

Here we will download the image from the given URL and will check that the downloaded image is present on the application(after clicking the submit button).

Follow a 2 step process:

1) Download the image from the image URL.
2) Verify that the downloaded image is present on the app after entering the image URL and clicking on the Submit button.

But before moving to the business logic part we need to integrate the latest version of the Appium java-client(7.0.0) which supports the new image locator strategy.

build.gradle

```
. . .
```

```
dependencies {
...
   testCompile group: 'io.appium', name: 'java-client',
version: '7.0.0'
}
...
```

After defining the dependency we can proceed.

1) Download the image from the image URL and convert it to Base64-encoded version:

In the image locator strategy we are dealing with the base64-encoded format of the image so downloading the image is not sufficient, we need to convert it to base64-encoded format. Below method will convert any image from it's URL to base64-encoded format.

```
public String getBase64FormatOfImageFromURL(String imageURL)
throws IOException, URISyntaxException {
    URL url = new URL(imageURL);
    try {
        InputStream is = url.openStream();
        byte[] bytes =
org.apache.commons.io.IOUtils.toByteArray(is);
        return
org.apache.commons.codec.binary.Base64.encodeBase64String(byt
es);
    } catch (Exception e) {
        throw new RuntimeException("Please check the network
on your server! It seems disconnected.");
      }
}
```

2) Verify downloaded image is present on the app (after entering the image URL and clicking on the Submit button).

After getting the base64 format of the downloaded image, you need to check that base64-encoded string of that image is present on the application (after entering the image URL and clicking on the submit button on the app).

Please find the below code for reference:

```
public void isImageAppearOnApplication(String
base64FormatImage) throws IOException, URISyntaxException {
```

```
    waitUtils.staticWait(5000);
    try {

Assert.assertTrue(AppiumUtils.isElementDisplayed((AndroidElem
ent) driver.findElementByImage(base64FormatImage)), "Expected
Image did not appear on dashboard Screen.");
    } catch (NoSuchElementException e) {
        throw new RuntimeException("Expected image didn't
display on Application!");
    }
}
```

In the above code, notice how we have put a static wait of 5 seconds - this is because as soon as the SUBMIT button is tapped, it will take some time to fetch and display the image on the app from the image URL.

NOTE: *Here we could have put the dynamic wait instead of static if Image view was present in DOM for the displayed image.*

Using Image Comparison to Locate an Element

A more common use case for Image comparison is to find an element that you can't find in any other way. We can use image comparison to find, for example, a button by comparing the image of that button and then perform an action on it.

We're going to use a somewhat contrived sample application but it will help demonstate the concept. In this sample application there are 3 icons (pretend that they're buttons!) and none of them have any id or unique locator assigned (You *could* use the indexing but we want to focus on the image locator strategy). When you click on the first image the textview will be visible saying "First Icon clicked!". Similarly when you click on the second or third image, the textview will visible accordingly.

We want to click on the first button and want to *assert* that the "First Icon clicked!" textview is visible. This will be 3 step process:

1) Get the image file of the button.
2) Get the element using the image.
3) Click on the element.

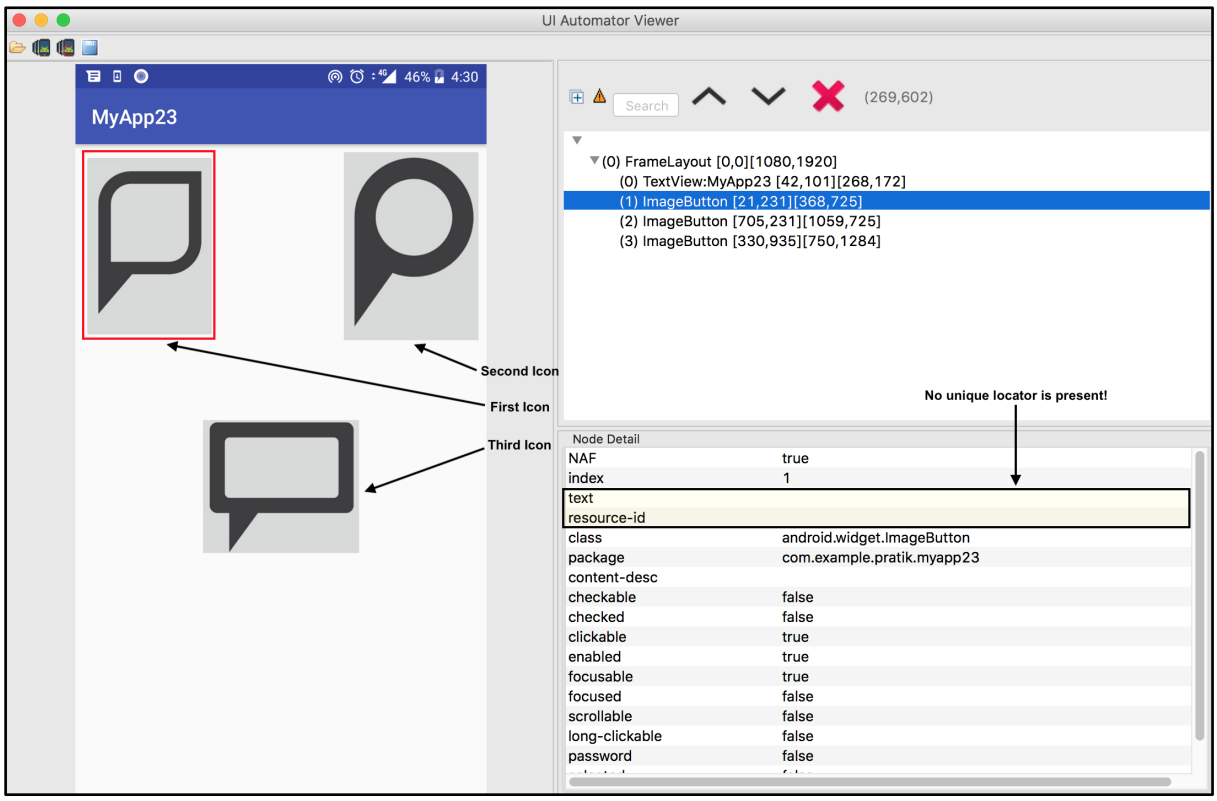

Figure-7:Android application having 3 buttons.

So let's understand the each step:

1) Get the image file of the button

There are many ways you can get the image for the any of element.

1. Capture the screenshot of the application and crop the needed element's image.
2. Using Appium Inspector/uiautomatorviewer

Here we can use the 2nd approach since it is more convenient for our example. We can inspect the elements along with the image screenshot in the uiautomatorviewer tool and using snipping tool(windows)/image capture(mac) we can simple capture the image of selected area easily. Going with a capture is going to be inaccurate - your best bet is to get the source image file from the developers or UI/UX team.

2) Get the element using the image

After getting the image we need to convert the image into Base64 encoded format. In java below code will convert any image file(.jpg/.png) to Base64 encoded format.

```
public static String getBase64StringFormatOfImage(String
imgName) throws URISyntaxException, IOException {
    URL refImgUrl = ImageUtils.class.getClassLoader().
getResource(imgName);
    File refImgFile = Paths.get(refImgUrl.toURI()).toFile();
    return Base64.getEncoder().
encodeToString(Files.readAllBytes(refImgFile.toPath()));
}
```

Here we will pass the image name as an argument in the method which we have from step 1. And using this code we can get the element from image.

```
String base64FormatOfImageFromImage =
ImageUtils.getBase64StringFormatOfImage(imageName);

WebElement iconButton =
androidDriver.findElementByImage(base64FormatOfImageFromImage
);
```

3) Click on the element

After getting an element you just need to click on the element. Using: `iconButton.click();`

And on the last step we will make assertion that “First Icon clicked!” textview is visible when click() action performs.

```
String expectedText = "First Icon clicked!";

String actualText =
androidDriver.findElement(By.id("com.example.pratik.myapp23:i
d/textView")).getText();

Assert.assertEquals(expectedText, actualText, "Actual and
Expected Text didn't match!");
```

This feature opens the door to new possibilities because there were cases when the traditional locator strategy could not help us. But now, by using the Image locator strategy you can literally find any UI element and apply an action upon it.

Image matching: Find occurrence of partial image in the full image

Another important capability in the image comparison feature is to match the partial image on the full image. Appium provides a method:

```
findImageOccurrence(byte[] fullImage, byte[] partialImage,
@Nullable OccurrenceMatchingOptions options)
```

which serves this purpose.

We will use a simple example to demonstrate this. We will use an Android application showing the full size of an image and we will verify that the partial image of that full image is present.

These are the steps to *assert* that the partial image is present in the full image:

1) First of all we need to convert both partial and full images to the Base64 encoded byte format. Below method is responsible to convert any image to it's Base64 encoded byte format.

```
/**
* This method is used to convert Image(.png file) to Base64
Byte format.
*
* @param imgName
* @return
* @throws URISyntaxException
* @throws IOException
*/
public static byte[] getBase64ByteFormatOfImage(String
imgName) throws URISyntaxException, IOException {
    URL refImgUrl =
ImageUtils.class.getClassLoader().getResource(imgName);
    File refImgFile = Paths.get(refImgUrl.toURI()).toFile();
    return Files.readAllBytes(refImgFile.toPath());
}
```

2) Now we need to call the `findImageOccurrence()` method with appropriate parameters.

```
OccurrenceMatchingResult imageOccurrence =
androidDriver.findImageOccurrence(fullImage, partialImage,
new
OccurrenceMatchingOptions().withThreshold(0.1).withEnabledVis
ualization());
```

Here we need to pass 3 parameters:

1. Base64 encoded byte format of Full Image
2. Base64 encoded byte format of Partial Image
3. New `OccurrenceMatchingOptions()` with `withThreshold(0.1)` & `withEnabledVisualization()`

`withEnabledVisualization()`: In order to enable the visualization so that you can get the visualization length.

`withThreshold(0.1)`: This is the threshold value for image matching. If you don't use this method and you have only small portion of the full image to compare then you might face an error: "Cannot find any occurrences of the partial image in the full image above the threshold of 0.5". Here we have used this method with a 0.1 value which means we have put a minimum threshold of 0.1 so we won't get the exception.

3) Add the assertion by checking that the length of visualization is greater than 0.

```
Assert.assertTrue(imageOccurrence.getVisualization().length >
0 , "Partial image is not present!");
```

VisualizationTest.java

```
@Test
public void testVisulization() throws IOException,
URISyntaxException {
    byte[] fullImage =
Base64.encodeBase64(ImageUtils.getBase64ByteFormatOfImage("ab
stractFullImage.jpg"));
```

```
    byte[] partialImage =
Base64.encodeBase64(ImageUtils.getBase64ByteFormatOfImage("ab
stractPartialImage.png"));

    OccurrenceMatchingResult imageOccurrence =
androidDriver.findImageOccurrence(fullImage, partialImage,
new
OccurrenceMatchingOptions().withThreshold(0.1).withEnabledVis
ualization());


System.out.println(imageOccurrence.getRect().getDimension());

System.out.println("X:"+imageOccurrence.getRect().getX());

System.out.println("Y:"+imageOccurrence.getRect().getY());
    System.out.println("Height:"+imageOccurrence.getRect().
getHeight());
    System.out.println("Width:"+imageOccurrence.getRect().
getWidth());

System.out.println(imageOccurrence.getVisualization().length)
;


Assert.assertTrue(imageOccurrence.getVisualization().length >
0 , "Partial image is not present!");
}
```

In order to get more details on image comparison visit the official [appium docs.]

You can find the above complete examples on our GitHub [page].

If you want to know more about the image locator strategy, please review this article on Jonathan Lipps' blog: [https://appiumpro.com/editions/32]

Chapter-14: End-to-End Testing

We've covered a lot up until this point. If you've stuck with us, you should hopefully feel far more knowledgeable about Appium. Don't worry if you feel it hasn't all quite "come together" yet. Start small and write some simple test cases. You don't need to use the advanced concepts like our design patterns until you become more comfortable with the basics of Appium.

This chapter aims to help you bring in all the moving pieces and combines everything you've learned up to now by showing you how to apply all your knowledge in an "end-to-end" test. At times we'll reference previous sections, and we'll also rehash some information we've previously covered. Seeing coverage from another angle will help solidify your knowledge.

This chapter is divided into 4 sections:

1) Setting up Appium.
2) Test Planning.
3) Test Setup.
4) Test Case Writing and Execution.

1) Setting up Appium

This is a section we won't repeat here as it was covered extensively in the first chapter. Setting up environments isn't ever any fun but once it's setup you're good to go. We're going to assume you have everything setup - and refer to the first chapter if you need some help.

2) Test Planning

As the expression goes - "Measure twice, cut once". Some planning up-front is going to save you a lot of headache later on. And good test planning means a little experimentation with manual testing before starting automation. Specifically:

1) The first thing is you need to check application compatibility or suitability for automation testing. You can check that by verifying the values of selectors using the locator inspection tool (Appium accessibility inspector or UiAutomator). You can quickly assess the locators of basic UI elements such as username field, password field, login button etc. and if you don't find unique locators for them you're going to need to use Xpath which is less than ideal. If at all possible,

work with the development team to see if they can assign unique locators to UI elements.

2) After verifying the availability of unique locators, you need to explore the whole application thoroughly, you need to understand each and every feature of the app and need to prepare the list of the most important ones. After preparing the list you can prepare manual test cases. With automation and parallel execution, keep in mind that your test could be executed in a random order at any point of time so it should be very granular, and it is important that the test cases you design are modular and independent.

3) Test Environment Setup

In order to perform automation using Appium you need to:

1) *Write the code which will find the UI element on the screen.*
2) *After getting the element, write the code which will perform an action upon it.*

This is all done within your test code - the Appium server will interact with the application artifacts(.ipa or .apk) and your test code:

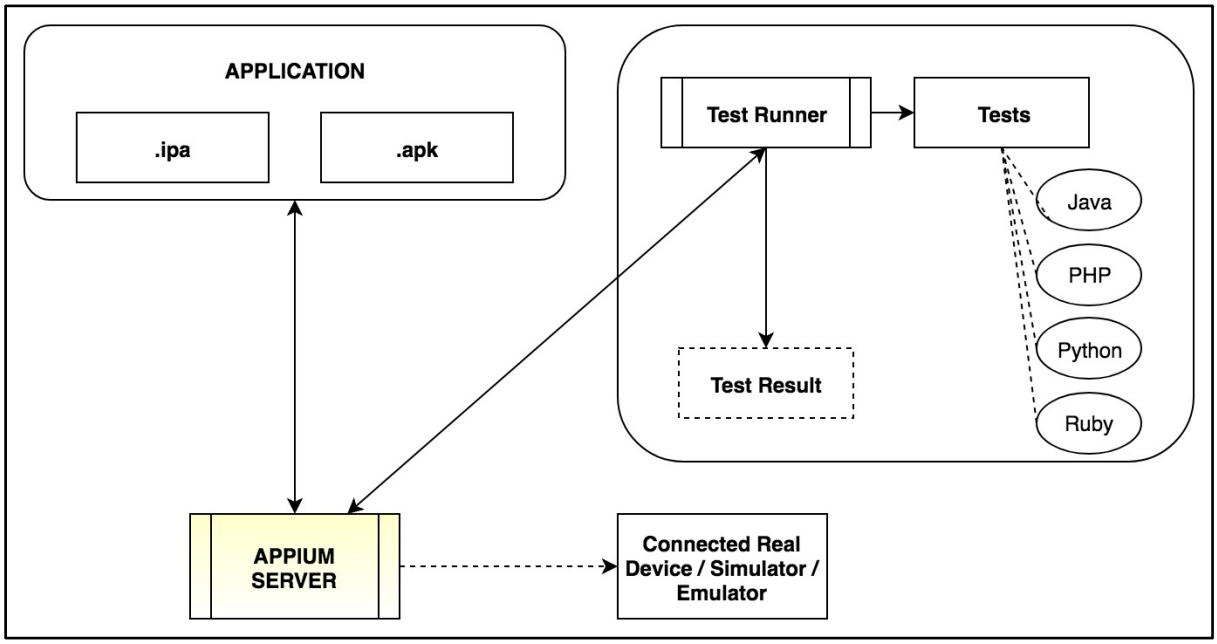

Figure-1: Appium Process.

So you can create a native mobile application on XCode(for iOS) and Android Studio(for Android) OR you can also use the build (.ipa/.apk) directly, write the UI test cases in your preferred programming language and executes them manually from the IntelliJ IDEA, Eclipse IDE, Visual Studio or IntelliJ PHPStorm.

Moreover, if you want to achieve end-to-end automation and want UI test cases as the part of your CI/CD process then you can also integrate them with tools such as Jenkins and BitBucket.

So first thing first is you need to get the build(.ipa/.apk), choose the programming language for your automation code and get the physical device/Simulator/Emulator for testing.

4) Test Case Writing

After designing the test cases and setting up the test environment you can start writing the test cases. For illustrative purposes, we will be automating the Android messaging app from Google. To reemphasize best practices, we will utilize the page object model to create a cleaner and more modular solution. Because we already discussed the page object model framework in detail previously, we can save time by cloning the automation framework project we used previously which you can find on github here.

Now let's discuss the scenario which we want to automate. We have selected the Google message application(v 3.9.039) for automation.

But before starting the automation we will need to plan our test. So as per our above discussion, we need to take care of 2 things.

1) We need to quickly check the app feasibility for automation and to do so we need to verify that the locators of the app are unique. Here we will use uiautomatorviewer to find the unique locators of UI elements as it is quick on Android compared to the Appium inspection tool.

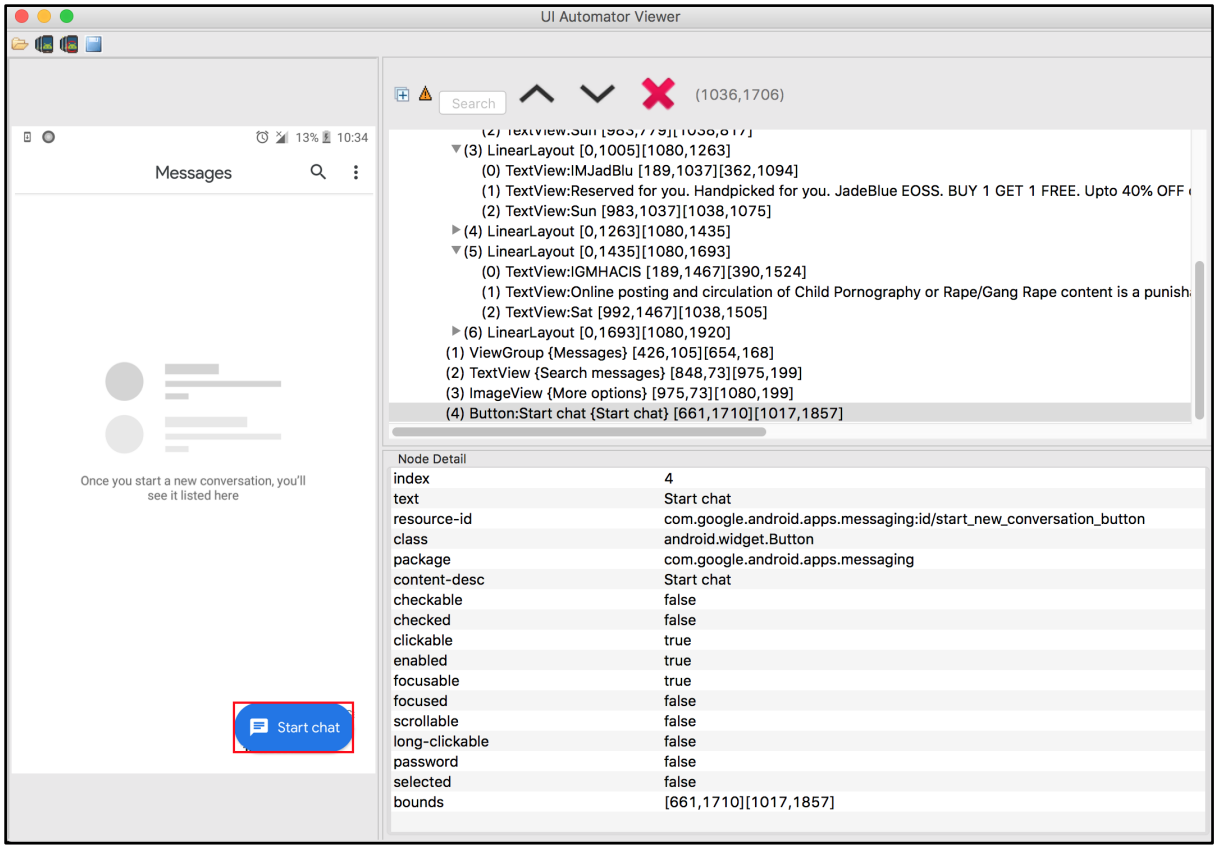

Figure-2: Quick check of locators on messaging app.

As you can see above, the button has a unique resource-id - The Google developers have assigned unique ids to all UI elements, so the first step is clear.

2) The Google message app is straightforward - you can explore the whole application and can get an idea of each feature. As the application is mainly designed to send an SMS to contacts we will need to automate the most important scenario which is sending the SMS to a particular contact.

Here are the manual steps we will perform and then use this to create the automation:

No.	Test Steps	Expected Output
1	Open the google message application.	An app should open.
2	Tap on 'Start chat' OR '+' button.	New conversion screen should appear.

3	Type the contact no. in 'To' textfield.	'Send to contact no.' row should be visible.
4	Tap on the suggested contact below search box.(like 'Send to 111-1111')	Conversation screen should appear.
5	Type into the message textfield.	A message should be typed correctly.
6	Tap on 'SMS' button	Message should be sent to the recipient.

Now, both the conditions are satisfied so we can move forward and start writing the automation test case. We will use the page object model framework for writing the automation test case and using it involves just a few steps - it may seem like overkill for this test case but we're putting in a good baseline for a larger more sophisticated automation project:

1) You can get the POM-based automation framework from our GitHub page.

2) Import as gradle project in IntelliJ IDEA/Eclipse IDE.

3) Run the build.gradle file in order to download all dependencies.

4) Move to the `configuration.properties` file and change these properties as per the connected Android device.
`android.platform.version=<get android version of device>`
`android.device.name=<get device name using $ adb devices>`

5) Set the proper desired capabilities.

6) Get the unique locators for UI elements on the app.

7) Create the page objects of different screens.

8) Write the automation test case using the created page object's methods.

The First 4 steps are straight forward - now we get to the desired capabilities part!

Set desired capabilities

Because the app we have chosen comes pre-installed on Android devices, we can skip the installation and go straight to opening the app.

```
DesiredCapabilities desiredCapabilities = new
DesiredCapabilities();
desiredCapabilities.setCapability(MobileCapabilityType.AUTOMA
TION_NAME, "uiautomator2");
desiredCapabilities.setCapability(MobileCapabilityType.DEVICE
_NAME, "c4e3f3cd");
desiredCapabilities.setCapability(MobileCapabilityType.PLATFO
RM_NAME, "Android");
desiredCapabilities.setCapability(MobileCapabilityType.PLATFO
RM_VERSION, "8.0");
desiredCapabilities.setCapability(AndroidMobileCapabilityType
.APP_PACKAGE, "com.google.android.apps.messaging");
desiredCapabilities.setCapability(AndroidMobileCapabilityType
.APP_ACTIVITY,
"com.google.android.apps.messaging.ui.ConversationListActivit
y");
desiredCapabilities.setCapability(MobileCapabilityType.FULL_R
ESET, false);
desiredCapabilities.setCapability(MobileCapabilityType.NO_RES
ET, true);
desiredCapabilities.setCapability(AndroidMobileCapabilityType
.AUTO_GRANT_PERMISSIONS, true);
```

Here, we don't use the `desiredCapabilities.setCapability(MobileCapabilityType.AP P, <path-to-app>);` capability because we are not installing the application.

Getting the unique locators

After setting the desired capabilities you need to extract the selectors using uiautomatorviewer(or Appium inspector) for all the UI elements we need to control.

NOTE: The *Google message application will change periodically so there are chances that the element IDs may change over time, so please modify the automation script accordingly.*

There are 4 screens we need to take care of for automating our scenario:

1) Messages (Dashboard) screen.
2) New conversation screen.
3) New conversation screen for a new contact.
4) Conversation screen.

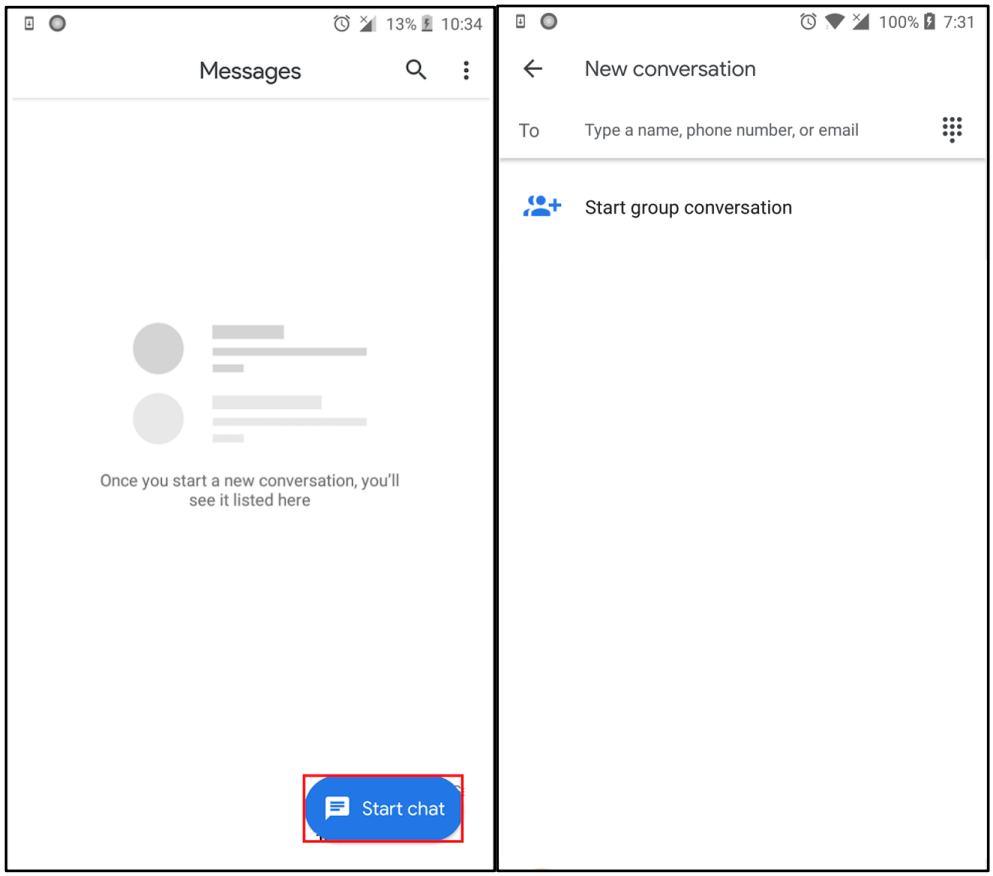

Figure-3.1: Google message app: Message & New conversation screens.

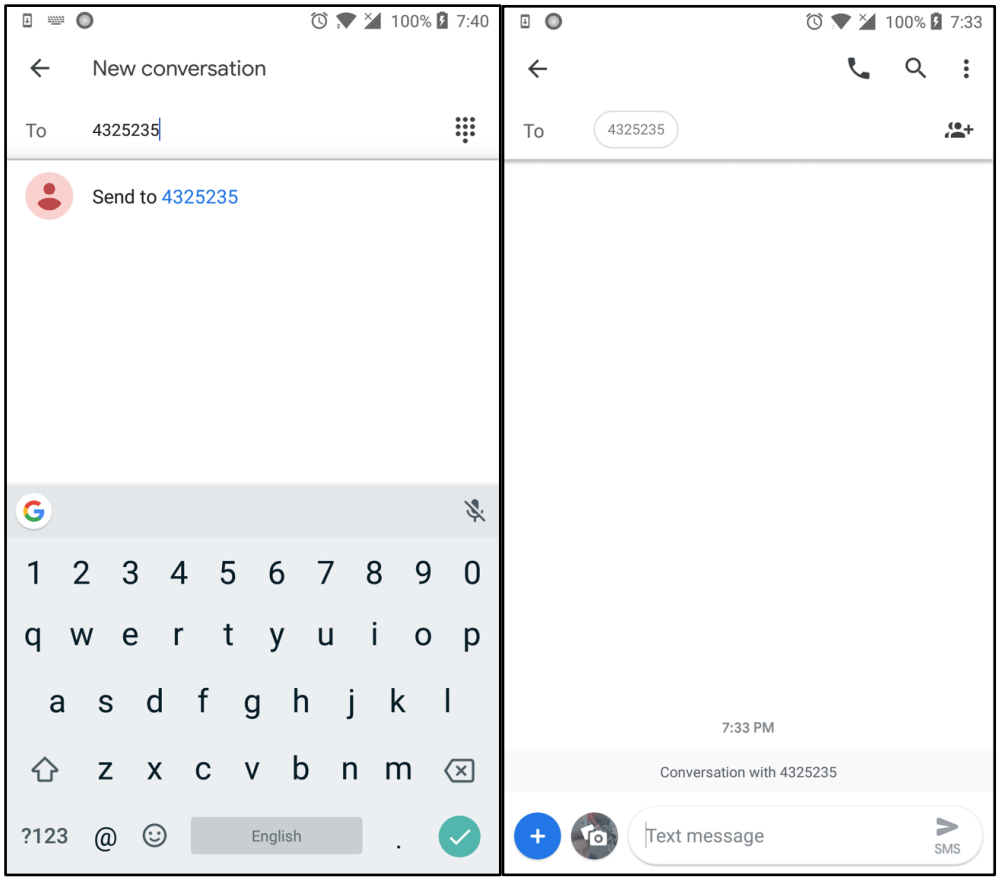

Figure-3.2: Google message app: New conversation & Conversation screens.

After identifying the screens, we need to get the selectors of each element from all screens which will be needed while automating our test case.

So let's make a list:

Screen Name	Element	Locator
Messages	'Start chat' button	**ID:** start_new_conversation_button Or **ID:** com.google.android.apps.messaging:id/start_new_conversation_button
New conversation	'To' textfield	**ID:** recipient_text_view
New conversation	'Send to 432-5235' textview	***Not present on DOM, but we can skip this by tapping on ENTER from keyboard.***
Conversation	SMS Message textfield	**ID:** compose_message_text

Conversation	'SMS' button	**ID:** send_message_button_container
Conversation	Sent Message layout	**XPath:** //android.support.v7.widget.RecyclerView/android.widget.FrameLayout

NOTE: *You can use the **ID** selector in these formats:*

1) *start_new_conversation_button*
2) *com.google.android.apps.messaging:id/start_new_conversation_button*

If you look at the above table closely you can notice that most of elements have unique Ids, however a few elements have some issues with their id:

1) 'Send to 432-5235' textview is not present in DOM at all so we can't locate that element.

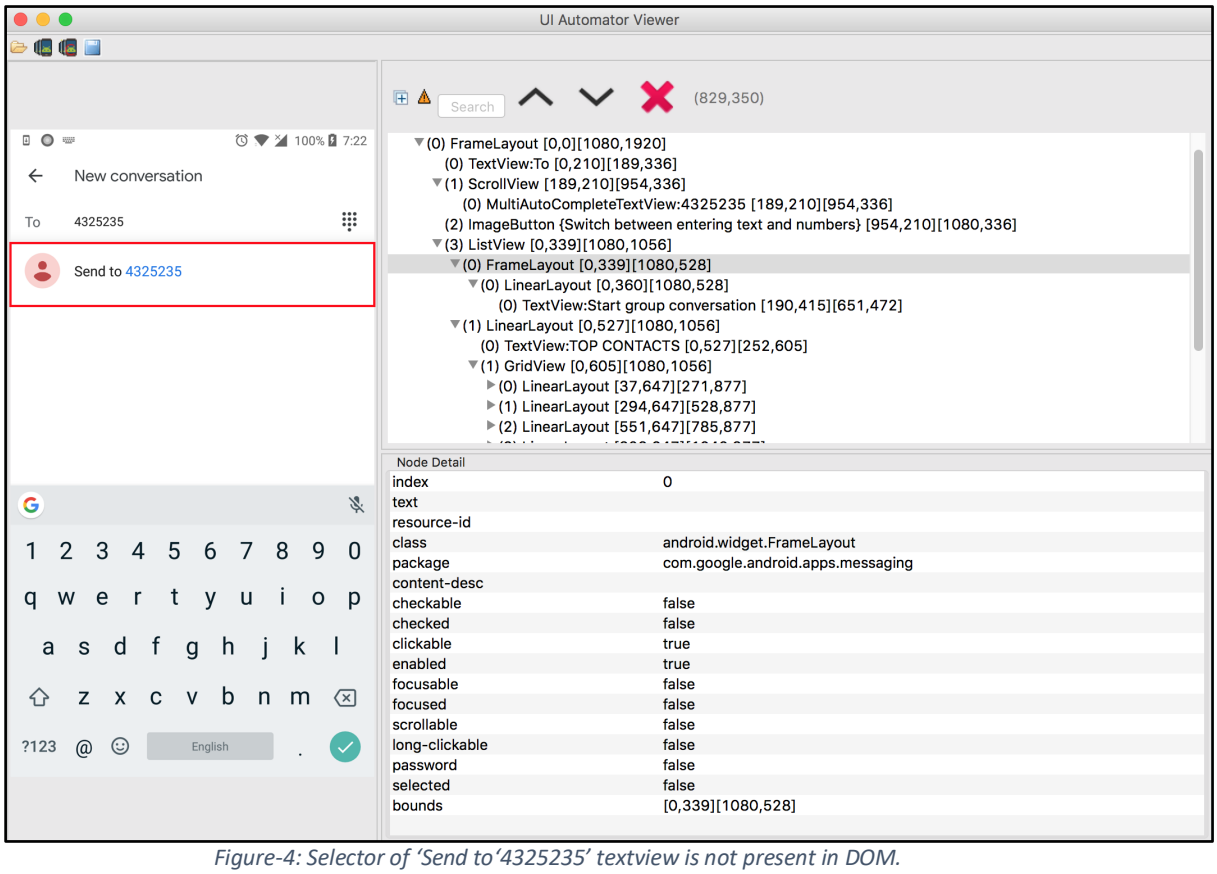

Figure-4: Selector of 'Send to'4325235' textview is not present in DOM.

So, we need to look for a workaround. If you look closely at this screen then you might figure out that you just have to press the 'correct' (green checkmark) icon on the keyboard and you don't need to press the 'Send to 432-5235' textview.

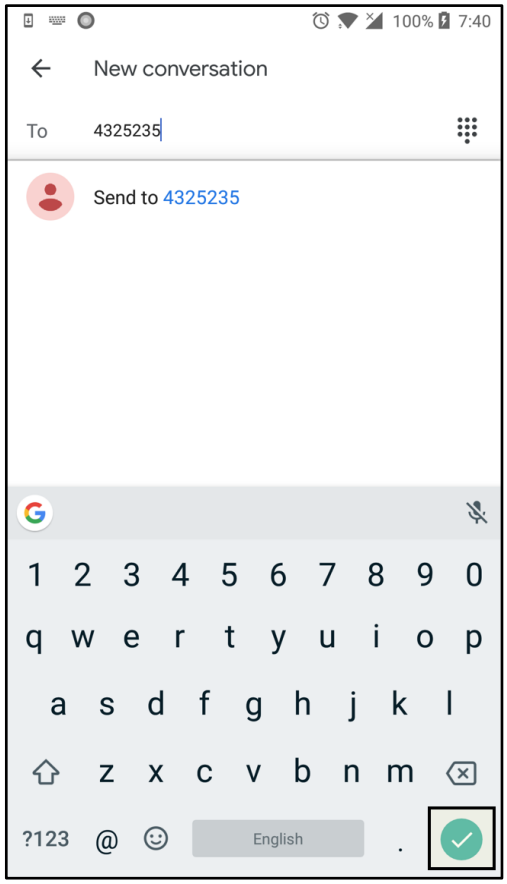

Figure-5: Use Right Icon instead of 'Send to 4325235' textview.

We can click on the icon from the soft keyboard:

```
driver.pressKey(new KeyEvent().withKey(AndroidKey.ENTER));
```

2) The 2nd issue is there is no unique id assigned to the sent message textview. So we need to look at another locator strategy. What about using the accessibility id locator strategy? Unfortunately that would only work if the id is static and here it keeps changing, so that strategy won't work either.

So the XPath locator strategy remains. It's actually good practice to use the ***text*** attribute in XPath but if you see the below screenshot you can find that there is no text assigned to the sent message textview. So we will need to use XPath as follows:

```
@AndroidFindBy(xpath =
"//android.support.v7.widget.RecyclerView/android.widget.Fram
eLayout")
List<AndroidElement> sentMessageLayout;
```

This game of finding the right locator strategy to use is very common

in test automation. You'll explore the elements and then go through a process of trying to find what the best locator strategy is in order to reach those elements.

You may be wondering why we have taken the List of `AndroidElement` ? This is because the problem with the single `AndroidElement` is if you have multiple messages sent to the same mobile number it will always take the first element but we need the last element of the sent message. Please see *Figure-7: FrameLayout presents for each sent message.*

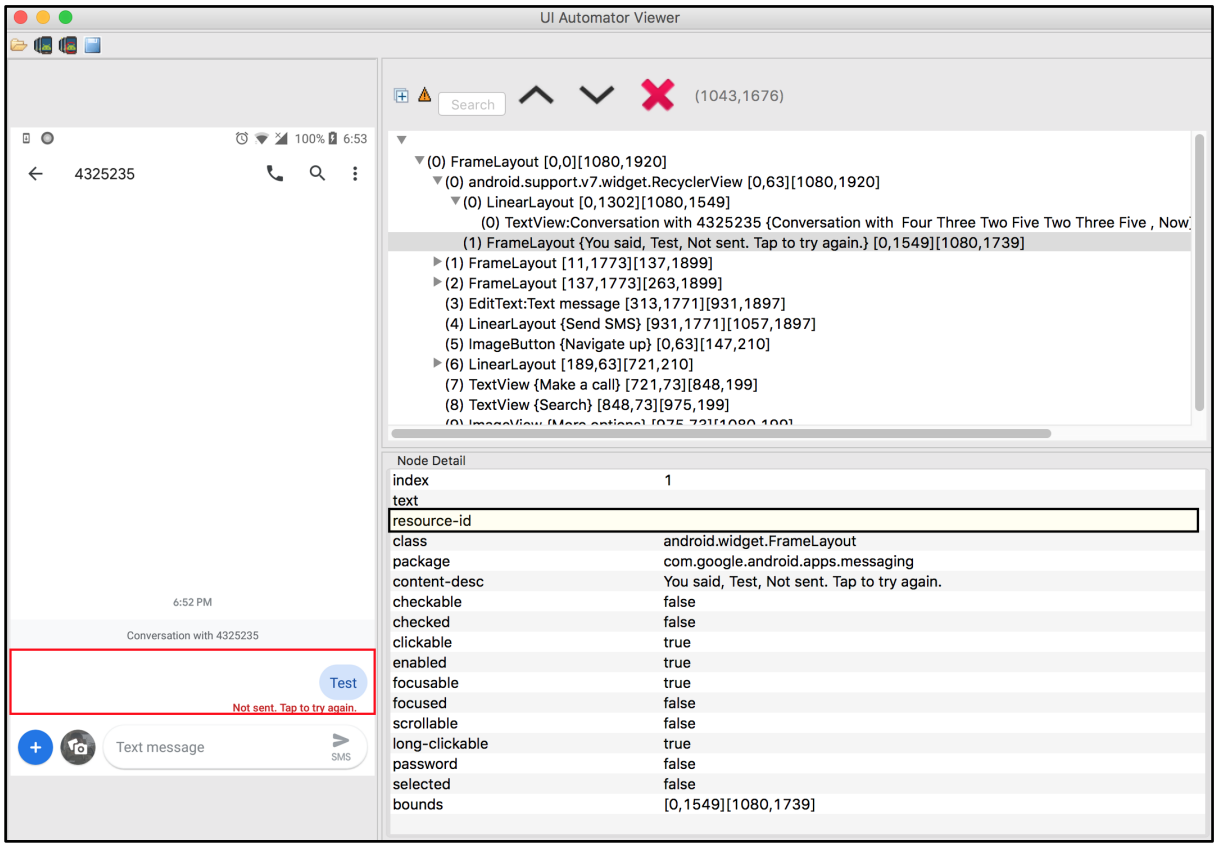

Figure-6: Id is not assigned to sent message textview.

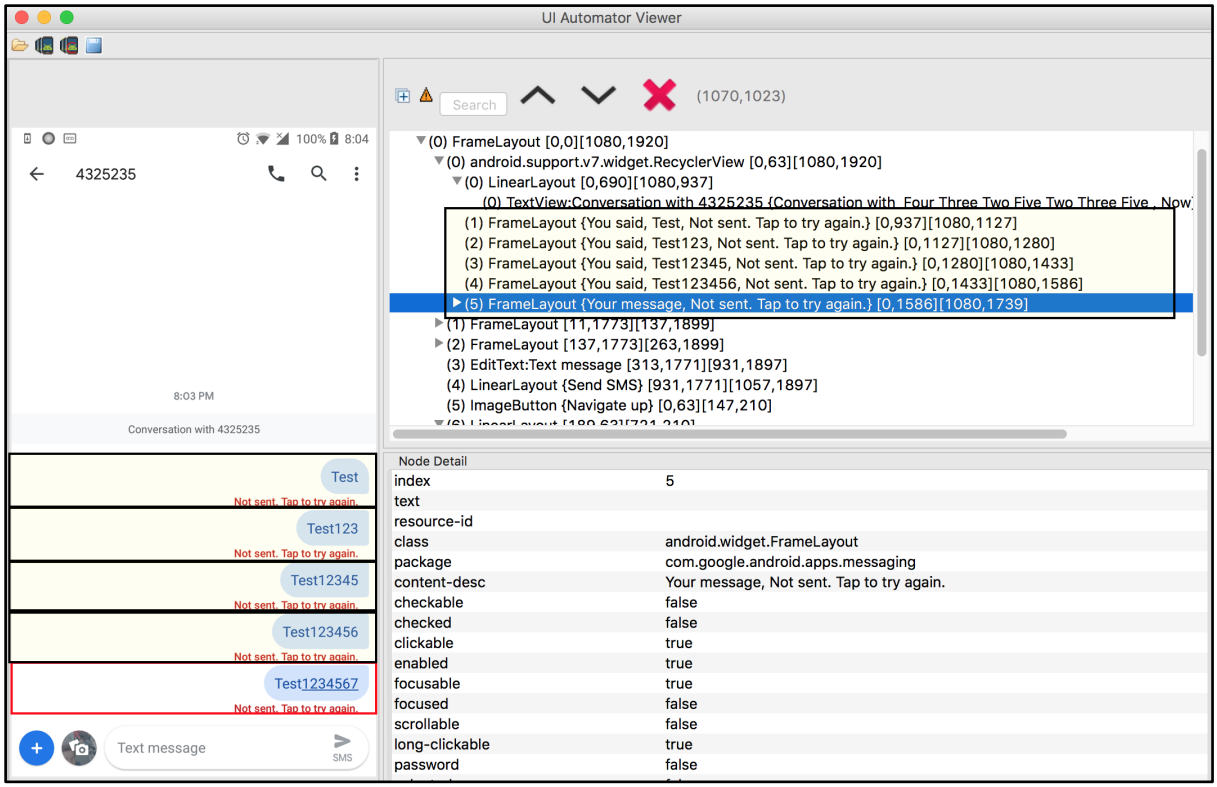

Figure-7: FrameLayout presents for each sent message.

Now that we have all the unique locators which we wanted, we can start creating our actions in the PO classes.

Create action methods in PO classes

We need to create action methods on Page Object classes which will tap on buttons, fill the text fields and assert text values on the app screen.

For example, on the Message(Dashboard) screen we need a method which will tap on the 'Start chat' button. We already have the selector of the 'Start button' so we can create our method:

```
public class MessagesPO extends BasePO {
    public MessagesPO(AppiumDriver driver) {
        super(driver);
    }

    @AndroidFindBy(id =
"com.google.android.apps.messaging:id/start_new_conversation_
button")
    AndroidElement startChatButton;
    public NewConversationPO tapOnStartChatButton() {
        startChatButton.click();
        return new NewConversationPO(driver);
    }
```

```
}
```

Why does our `tapOnStartChatButton()` method return a new object of `NewConversationPO`? The answer is that whenever any method is responsible to change the screen we can return the object of the subsequent screen's PO class so that we don't need to create the object of that PO class separately while writing test case.

Here, the method `tapOnStartChatButton()` will tap on 'Start chat' button which will navigate to the New Conversation screen so we are returning the object of `NewConversationPO` class.

This practice is not mandatory but it is good to have.

NOTE: *All PO class should extend the BasePO class as it contains the logic which initializes the page factory and other utility classes. You can look into the chapter titled: "Developing Test Automation Framework for Appium using Page Object Modeling(POM)" for more details.*

We have created the below table which gives the mapping between screen names and corresponding method names.

Screen Name	Method Name
Messages	**tapOnStartChatButton():** It will tap on Start chat button.
New conversation	**typeAndSubmitContactNumber(String contactNo):** It will type the contact no. and submit it for conversation.
Conversation	**typeInSMSTextField(String text):** This method will type into message textfield. **tapOnSMSButton():** This method will tap on the SMS button. **isConversationScreenDisplayed():** This method is used to verification purpose, it will return true if conversation screen appears. **getLastSentMessage():** It will return the AndroidElement for last sent message. **getLastSentMessageText():** This method is used to get the last sent message text. **isMessageSent():**It will verify that message is sent or not. **isLastSentMessageContains(String subString):** This method will check whether last sent message contains the passed subString or not.

Create the test case and use action methods from PO classes

This is the final and easiest, yet most powerful, step of automation test case

writing. Here you have to organize all the methods from your PO classes in order to make the complete automation test case and validate the result using assertions.

Assertions are fundamental in test case writing - without them you're just doing automation, and not automated testing. Ideally, you would put as many assertions as you can. Assertions can be thought of as checkpoints. At the end of any action method you will have some expected results, and to measure those expected results you have to put assertion statements in the code.

In our example we are using the TestNG assertions.

Assertions simply compares the value of expected and actual values. If the expected and actual values are equal then the assertion 'passes' and we continue with code execution. If it fails (the actual result does not match the expected result) the user defined message is thrown.

In our example we have 2 assertions:

1) **Verify that the message has been sent to the user:**
   Here we want to check that the application has sent the message and is being displayed on the conversation screen.

```
Assert.assertTrue(conversationPO.isMessageSent(), "Message:'"
+ messageText + "' is not being sent!");
```

The 2nd parameter is the error message we want to throw in case of failure of the assertion. A good error message should give sufficient information about what it is checking, so that your diagnostics becomes easier.

2) **Verify that sent message is as per expectations:**
   Checking that the message is sent is one thing … but was the *right* message sent? As discussed earlier there is no unique id assigned to the sent message text view, so we can not get the text of the last message. We can however get the whole FrameLayout text.

   So logically we can have an assertion which will check that the expected text is present on the last sent message or not. In our example, we have put the timestamp in milliseconds in a text message and we will verify that given timestamp text value is present on last sent message or not.

```
Assert.assertTrue(conversationPO.isLastSentMessageContains
(timestamp), "Last sent message is different than expected!,
Original message is: '" +
conversationPO.getLastSentMessageText() + "', while the
expected substring is: " + timestamp + "'");
```

After organizing the action methods from the PO and adding our assertions we have our complete test case which will send the message to a particular contact number.

```
public class TestCases extends BaseTest {

    @Test
    public void verifyUserCanSendMessage() {
        final String phoneNo = "00011122233";
        final String timestamp = System.currentTimeMillis() +
"";
        final String messageText = "Hello, there. Current
time is: " + timestamp;

        MessagesPO messagesPO = new
MessagesPO(androidDriver);

        NewConversationPO newConversationPO =
messagesPO.tapOnStartChatButton();

        ConversationPO conversationPO =
newConversationPO.typeAndSubmitContactNumber(phoneNo);
        Assert.assertTrue(conversationPO.
isConversationScreenDisplayed(), "Conversation screen didn't
appear!");
        conversationPO.typeInSMSTextField(messageText);
        conversationPO.tapOnSMSButton();

        Assert.assertTrue(conversationPO.isMessageSent(),
"Message:'" + messageText + "' is not being sent!");

Assert.assertTrue(conversationPO.isLastSentMessageContains
(timestamp), "Last sent message is different than expected!,
Original message is: '" +
conversationPO.getLastSentMessageText() + "', while the
expected substring is: " + timestamp + "'");
    }

    @BeforeTest
    @Override
    public void setUpPage() throws MalformedURLException {
```

```
        androidDriver = new AndroidDriver(new
URL(APPIUM_SERVER_URL), getDesiredCapabilitiesForAndroid());
    }

}
```

The full code is available on github here.

As we said at the outset, this app may have changed by the time this guide was published. A new UI could render some of our test cases invalid. We have however tried to lay out a methodical and disciplined approach to tackling any test automation project. You should be to apply this approach for any mobile application.

Chapter-15: Test Automation Design Patterns You Should Know

Design patterns are used extensively when programming and they generally offer a reusable solution to a known occurring problem. In many respects, they introduce a set of best practices into your code and usually result in more flexible and maintainable code.

Strictly speaking, design patterns are optional. There are many ways to code a solution. Your organization may enforce certain patterns precisely because of maintainability. Although optional, knowledge of various design patterns and knowing when to use them will improve your skills in test automation design. And that's what we're going to do in this chapter.

We already looked into one the best test automation design patterns - the Page Object Model - in the **Developing a test automation framework using appium** chapter**.** But there are many other framework patterns out there used by automation teams, and that's what we will explore in this chapter. Be forewarned though, this is quite a technical chapter. A suggested approach may be to skim through the content and then revisit it a second time in more detail.

1) Page Object Model(Pattern)

The Page Object Model is a widely used object design pattern for structuring automation test code. Here, pages in the app are represented as Classes, and various UI elements of that pages are defined as variables. We already have gone through this technique in detail previously, so here we will discuss the abstract structure and explore how it can be implemented in a slightly different way or in a different programming language.

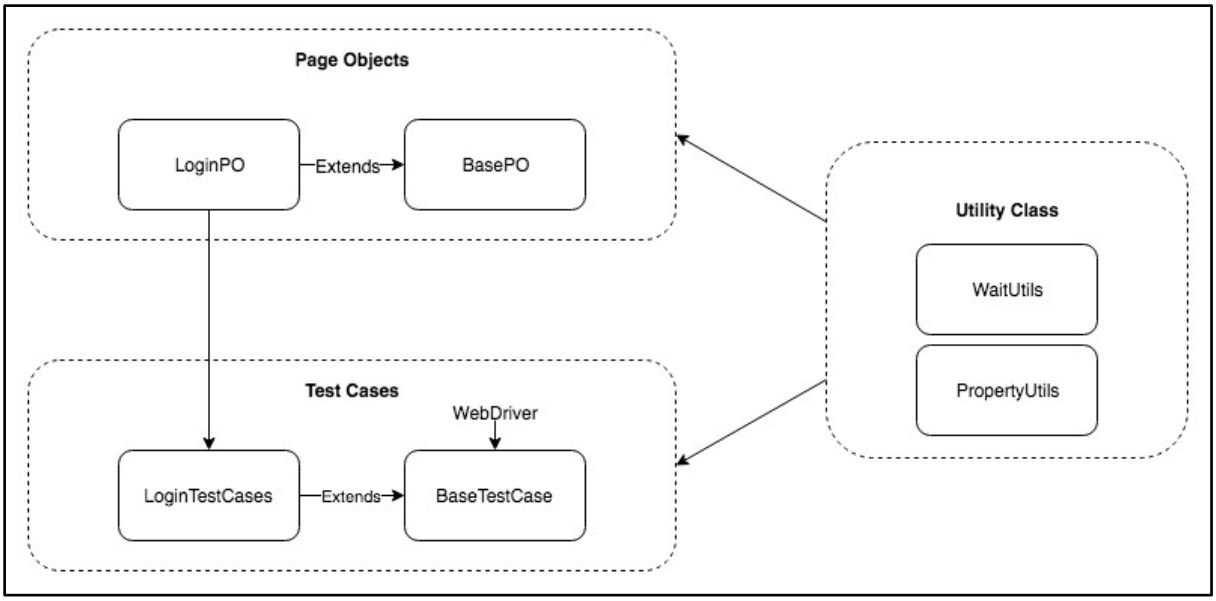

Figure-1: Page Object Pattern.

We use the Page Factory class to initialize the mobile(web for web application) elements that are defined in Page Object(PO) classes.

PO classes containing the mobile elements needs to be initialized using the Page Factory before it can be used, and this can be achieved by simply calling the initElements function of PageFactory. In that method you need to initialize the `AppiumFieldDecorator` class by passing the Appium Driver and Implicit wait duration objects.

```
PageFactory.initElements(new AppiumFieldDecorator(driver,
Duration.ofSeconds(IMPLICIT_WAIT)), this);
```

Or you can put the code in the constructor of the `BasePO` class.

Page Factory will initialize every MobileElement(AndroidElement or IOSElement) variable with a reference to a relevant element on the actual mobile screen and this is achieved by using **@FindBy** annotations. This annotation allows us to not only retrieve the mobile element, but also information such as the locator identifying strategy name and the locator value for retrieving it:

```
@iOSFindBy(accessibility = "Toolbar Done Button")
IOSElement doneButtonOnKeyboard;
```

Whenever the above code is used, the driver object will find it on the current mobile screen and simulate the action.

With this pattern, you will need to design a Page Object class according to the particular screen you wish to automate. For example, for the Login screen you

can create a `LoginPO` class and can put all the UI element locators as variables iin the `LoginPO` class. And don't forget that every PO class will extend the BasePO class where we are calling the PageFactory initElements method in the constructor.

In the Page Object Pattern all the locators will be reside in the relevant Page Object Classes such as `LoginPO`, `RegisterPO`, `DashboardPO` etc.. and the method from that PO classes gets used by the Test Classes - and this is the main advantage to using the Page Object pattern as locators and tests are residing in different places. So whenever any UI locator is changed you just need to apply changes on the particular Page Object classes to fix the automation script. This is the primary reason why the Page Object pattern is so widely used in automation projects.

2) Factory Design Pattern

In the factory design pattern we have a super class with multiple subclasses and based on some input, we need to return a particular subclass. It is often used when a class cannot anticipate the type of objects it needs to create beforehand. Here, instantiation of a class is done from the factory class. So when we need to create the object based on particular input this pattern is used. So how does it relate to automation test design?

This design pattern is best suited to when you are working with Android and iOS and both having the same accessibility id on iOS and content-desc in Android. So here the Factory class will create the relevant driver object (either Android or iOS) and always returns a newly created object or re-initialized one, so you don't have to check the platform every time.

Example:

We have implemented a factory class that creates a Driver object based on a specific input(platform name). This factory is very simple but it is perfect fit for our purpose.

You can design the factory class and it's methods for more complex applications.

```
public class DriverFactory {

    public AppiumDriver getDriver(String platformType) throws MalformedURLException {
        if (platformType == "Android") {
            DesiredCapabilities desiredCapabilities = new DesiredCapabilities();
            desiredCapabilities.setCapability(MobileCapabilityType.AUTOMATION_NAME, AutomationName.IOS_XCUI_TEST);
            desiredCapabilities.setCapability(MobileCapabilityType.DEVICE_NAME,  value: "John's iPhone");
            desiredCapabilities.setCapability(MobileCapabilityType.PLATFORM_NAME,  value: "iOS");
            desiredCapabilities.setCapability(MobileCapabilityType.PLATFORM_VERSION,  value: "12.2");
            desiredCapabilities.setCapability(MobileCapabilityType.APP,  value: "/Users/abc/projectA/src/test/resources/sampleApp.ipa");
            desiredCapabilities.setCapability(MobileCapabilityType.FULL_RESET,  value: false);
            desiredCapabilities.setCapability(MobileCapabilityType.NO_RESET,  value: true);
            return new AndroidDriver(new URL( spec: "http://127.0.0.1:4723/wd/hub"), desiredCapabilities);
        } else if (platformType == "iOS") {
            DesiredCapabilities desiredCapabilities = new DesiredCapabilities();
            desiredCapabilities.setCapability(MobileCapabilityType.AUTOMATION_NAME, AutomationName.IOS_XCUI_TEST);
            desiredCapabilities.setCapability(MobileCapabilityType.DEVICE_NAME,  value: "John's iPhone");
            desiredCapabilities.setCapability(MobileCapabilityType.PLATFORM_NAME,  value: "iOS");
            desiredCapabilities.setCapability(MobileCapabilityType.PLATFORM_VERSION,  value: "12.2");
            desiredCapabilities.setCapability(MobileCapabilityType.APP,  value: "/Users/abc/projectA/src/test/resources/sampleApp.ipa");
            desiredCapabilities.setCapability(MobileCapabilityType.FULL_RESET,  value: false);
            desiredCapabilities.setCapability(MobileCapabilityType.NO_RESET,  value: true);
            return new IOSDriver(new URL( spec: "http://127.0.0.1:4723/wd/hub"), desiredCapabilities);
        } else
            return null;
    }
}
```

Figure-2: DriverFactory Class.

As you can see above, we have created one method `getDriver(String platfromType)` on the `DriverFactory` class. So according to the platform type this method will return the particular AppiumDriver. This factory is used to instantiate AppiumDriver in tests based on the external parameter.

```
AndroidDriver driver = new
DriverFactory().getDriver("Android");
driver.findElement(By.id("username")).sendKeys("john");
driver.findElement(By.id("password")).sendKeys("abc123");
driver.findElement(By.id("login")).click();
MobileElement profileIcon = (MobileElement)
driver.findElement(By.id("profileIcon"));
Assert.assertTrue(profileIcon.isDisplayed(), "Login was not
successful.");
```

3) Facade Pattern

The Facade pattern provides a simple interface to deal with complex code.

In the facade pattern, as applied to test automation, we design the facade class which has methods that combine actions executed on different pages.

It is best to understand by looking at a practical example:

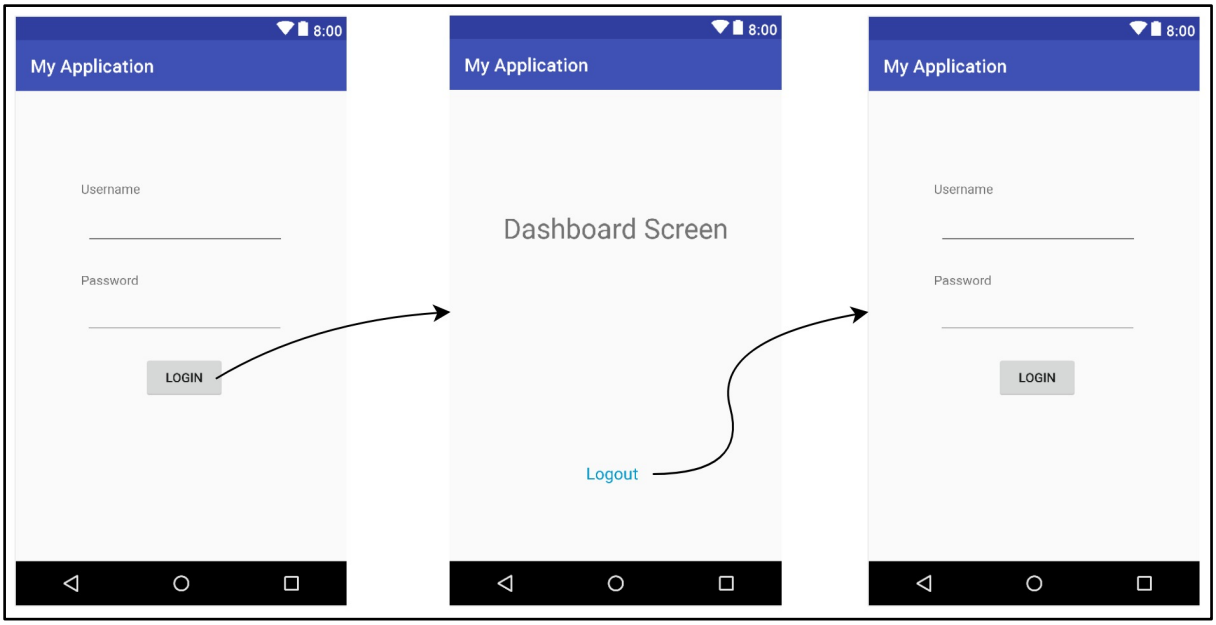

Figure-3: Sample Login Application.

Here we are going to automate a simple workflow using the facade pattern, like first login to the application and move to the dashboard page, and from there logout.

The Facade pattern is just an extension of the Page Object pattern, so basically in order to automate the above scenario we need to create page objects for different screens, so here we need to create 2 PO classes:

1) LoginPO
2) DashboardPO

Now we need to create one additional class, LoginFacade, which contains the objects of PO classes and it also contains the business logic using those objects. So the advantage of facade is you don't have to deal with the PO classes individually in your test script, you just need to use the facade class.

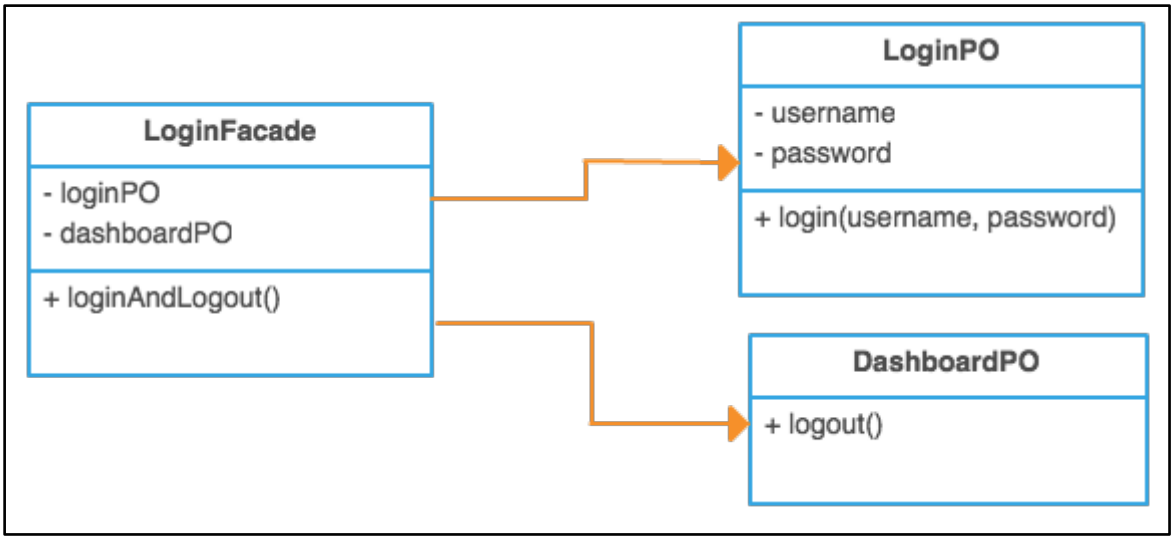

Figure-4: Facade Pattern.

LoginPO - It contains the method of login to the application.

DashboardPO - It contains a `logout()` method.

LoginFacade - It has LoginPO and DashboardPO objects defined and method `loginAndLogout()` also added, so you don't have to call all PO methods, you just have to deal with Facade class methods which are internally calling PO methods. This type of automation framework is really useful when you are dealing with complex app and you have many POs defined.

LoginFacade.java

```
public class LoginFacade {

    private AppiumDriver driver;
    private LoginPO loginPO;
    private DashboardPO dashboardPO;

    public LoginPO getLoginPO() {
        if (loginPO == null) {
            loginPO = new LoginPO(driver);
            return loginPO;
        } else
            return loginPO;
    }
```

```
    public DashboardPO getDashboardPO() {
        if (dashboardPO == null) {
            dashboardPO = new DashboardPO(driver);
            return dashboardPO;
        } else
            return dashboardPO;
    }

    public void loginAndLogout(String username, String
password) {
        getLoginPO().setUsernameTextField(username);
        getLoginPO().setPasswordTextField(password);
        getLoginPO().tapOnLoginButton();
        getDashboardPO().tapOnLogoutTextView();
    }
}
```

So as you can see in above example you only have to use the `loginAndLogout()` method in order to do login and logout without depending on other PO classes.

Now in case any workflow changes in your test cases you just need to change it in one place, and if you want to add some additional business logic, you can directly add them to the facade class.

For large and complex applications, If you don't use facade pattern then it's totally fine, but you may face some complexity in your automation framework and your code may ultimately become unwieldy.

You can find the complete project on our github project.

4) Singleton Pattern

A Singleton class means only one instance of it can exist at any time.

But why would you need this?

Well it is very useful in a case when you need to use the same object across the whole framework. A Singleton class returns the same instance every time you try

to instantiate an instance of the class. Think of it providing global access to a single object, for example, the log file object.

Creating a singleton class consists of:

1) Making the constructor of the Class private.
2) Make a static reference of the class, as we want to make this available globally.
3) Make a static method which returns an object of type class and it should also check whether class is already instantiated once or not - if it's not instantiated then it should instantiate that otherwise it can return reference of the class directly.

Sample of SingletonClass:

```
public class SingletonClass {
    public static SingletonClass singletonClass;

    private SingletonClass() {
        System.out.println("Singleton Class object
created.");
    }

    public static SingletonClass getSingletonClass() {
        if (singletonClass == null) {
            singletonClass = new SingletonClass();
        }
        return singletonClass;
    }

    public static void main(String[] args) {
        SingletonClass sc1 =
SingletonClass.getSingletonClass();
        SingletonClass sc2 =
SingletonClass.getSingletonClass();
    }
}
```

Output:

```
Singleton Class object created.
```

In above example we have created a Singleton class and we have defined two objects which are instantiating the SingeltonClass two times but as you can see in the output, the SingletonClass instantiates only once, after that it will re-use the created instance.

Now let's discuss how we can leverage this in Automation.

The Singleton pattern in automation can help us in many ways including:

1) We can ensure a single driver instance is used throughout our test cases.
2) Loading test data or other files just once rather than loading them repeatedly.

So whenever you feel that particular objects should only be instantiated once, you need to use the Singleton pattern. For example if a properties file in java is loaded once you don't want to load it again every time, consuming memory and resources. Using the singleton pattern you can do just that.

In the following example we will create a Singleton class to create the AppiumDriver(for iOS) only once.

SingletonAppiumDriver:

```
public class SingletonAppiumDriver {

    public static SingletonAppiumDriver
singletonAppiumDriver;
    private AppiumDriver appiumDriver;
    public String appiumURL =
"http://127.0.0.1:4723/wd/hub";

    private SingletonAppiumDriver() {
        appiumDriver = new IOSDriver(new URL(appiumURL),
getDesiredCapabilitiesForIOS());
    }

    private DesiredCapabilities
getDesiredCapabilitiesForIOS() throws
MalformedURLException {
       // set desired capabilities for iOS
        return desiredCapabilities;
    }
```

```
    public static SingletonAppiumDriver
getSingletonAppiumDriver() throws MalformedURLException {
        if (singletonAppiumDriver == null)
            singletonAppiumDriver = new
SingletonAppiumDriver();
        return singletonAppiumDriver;
    }

    public AppiumDriver getAppiumDriver() {
        return appiumDriver;
    }
}
```

Using this Singleton class we can create test cases which will reuse the appiumdriver object instead of creating a new one every time.

IOSTestCase:

```
public class IOSTestCase {
    @Test
    public void sampleTestCase1() throws
MalformedURLException {
        int a = 9;
        int b = 1;

        SingletonAppiumDriver singletonAppiumDriver =
SingletonAppiumDriver.getSingletonAppiumDriver();
        AppiumDriver driver =
singletonAppiumDriver.getAppiumDriver();

        driver.findElement(By.id("IntegerA")).sendKeys(a +
"");
        driver.findElement(By.id("IntegerB")).sendKeys(b +
"");

driver.findElement(By.id("ComputeSumButton")).click();
        String answer =
driver.findElement(By.id("Answer")).getText();
        Assert.assertEquals(answer, a + b + "", "Expected
and Actual Result didn't match!");
```

```
    }

    @Test
    public void sampleTestCase2() throws
MalformedURLException {
        int a = 1;
        int b = 1;

        SingletonAppiumDriver singletonAppiumDriver =
SingletonAppiumDriver.getSingletonAppiumDriver();
        AppiumDriver driver =
singletonAppiumDriver.getAppiumDriver();

        driver.findElement(By.id("IntegerA")).sendKeys(a +
"");
        driver.findElement(By.id("IntegerB")).sendKeys(b +
"");

driver.findElement(By.id("ComputeSumButton")).click();
        String answer =
driver.findElement(By.id("Answer")).getText();
        Assert.assertEquals(answer, a + b + "", "Expected
and Actual Result didn't match!");
    }
}
```

You can see the full project on our github project.

5) Fluent Page Object Model Pattern

As you already know, the Page Object model is the the best framework to use on automation projects. However, it be simplified and made even more readable using the Fluent Page Object Model.

In the fluent page object pattern every method which is responsible to perform an action returns “this” in order to implement chaining methods for the business logic of the test.

But please note that doesn’t mean we never return the other screen PO class. Please refer to this example:.

LoginPO.java

```
public class LoginPO extends BasePO {
...
...
    public LoginPO setUsernameTextField(String username) {
        usernameTextField.sendKeys(username);
        return this;
    }

    public LoginPO setPasswordTextField(String password) {
        passwordTextField.sendKeys(password);
        return this;
    }

    public DashboardPO tapOnLoginButton() {
        loginButton.click();
        return new DashboardPO(driver);
    }
...
...
}
```

TestCases.java

```
public class TestCases extends BaseTest {

    @Test
    public void testUserCanLoginAndLogout() {
        String username = "pratik";
        String password = "test123";

        LoginPO loginPO = new LoginPO(driver);

        loginPO.setUsernameTextField(username).
        setPasswordTextField(password).
        tapOnLoginButton().
        tapOnLogoutTextView();

        Assert.assertTrue(loginPO.isLoginPageDisplayed(),
"Login Page did not appear after logout");
    }
```

```
}
```

In above example you can see that methods `setUsernameTextField` and `setPasswordTextField` are returning "this" while method `tapOnLoginButton` is returning the `DashboardPO`. So we are not forcing every method to return "this" object of it's own class since it is not a practical approach.

One this is implemented, the chained method call is far more elegant:

```
loginPO.setUsernameTextField(username).
        setPasswordTextField(password).
        tapOnLoginButton().
        tapOnLogoutTextView();
```

You can find this example on our github page.

For those that are interested, there are many other design patterns. Most of these are used extensively in software development, but arguably less so in test case automation:

1) Observer Design Pattern
2) Observer Design Pattern via Events and Delegates
3) IoC Container and Page Objects
4) Strategy Design Pattern
5) Advanced Strategy Design Pattern

You can learn more about these patterns on: https://dzone.com/articles/design-patterns-in-automation-testing

As we said on the onset, using design patterns is optional from a purely technical perspective and there are many ways to implement a solution. The more complex your environment, the more likely you are to benefit from the rigor imposed by design patterns. Also, many people new to test automation are intimidated at using design patterns. Our recommendation is to get comfortable with appium and test automation first, and then slowly expand your knowledge by incorporating design patterns.

Chapter 16 - Industry Viewpoints

In this chapter we have the privilege of getting some thoughts on the state of QA from industry leaders. It is a great way of concluding this book and leaving you with different perspectives.

Our panel of experts include:

- Patrik Patel
- Paul Grizzaffi
- Mush Honda
- Joe Colantonio

Pratik Patel

Pratik is one of the primary authors of this book. He shares his thoughts on the mobile application testing market:

> The mobile application testing service market is gaining traction as companies are recognizing the importance of automated testing as an essential requirement to improve product quality in an increasingly time-sensitive environment.
>
> The rapid advancement of technology is having a significant impact in the world of automated testing, and companies have more choices then ever before. The strides made in machine learning, data science and artificial intelligence are making their way into automation testing tools.
>
> And it isn't just technology advancement - QA and automation testing is becoming a fundamental part of DevOps processes and Agile methodologies. Companies are introducing automation earlier and more frequently. Testing early and testing often has a much lower cost than defects discovered later in the cycle.
>
> Moving beyond automated tests, exciting areas in testing include proactive defect prediction. Technologies such as machine learning, reinforcement

learning, neural networks, cognitive computing, robotics process automation and bot programming are giving rise to an exciting advancement in testing. Technology can analyze source code, user behavior, app structure etc and predict high-risk areas that should be tested, and even predict where defects may be occuring.

Even Appium is benefiting from these technologies. We already see more AI libraries helping with locators and classification.

Regardless of the technology used, the reality is that automation is a critical part of a testing portfolio in order for companies to stay competitive. Whether it is to improve quality or lower the time to market, automation is the strategy to enable this. The modern tester should be very comfortable with technology. Automation is the future, and test engineering will become more prominent.

About Pratik: Pratik is an automation engineer with significant experience working in the QA Automation Industry and has extensive Selenium and Appium experience. He has worked on giant automation test frameworks(at Cybage Software Pvt. Ltd., Gandhinagar) which was capable of executing 4500 UI test cases parallely - all built on Selenium, TestNG, Docker and Gradle. Recently he has worked as an automation expert for a cloud based mobile automation platform. Moreover he has experience working with Espresso, XCUITest, Katalon Studio, RanoRex Studio, Appium Studio and cloud base automation platforms such as Kobiton, Perfecto Mobile, Saucelabs, Browserstack and AWS Device Farm.

Paul Grizzaffi

There are two points that I'd like to see the automation discipline embrace. The first is to evaluate automation endeavors with business lenses. That is to say, let's not automate just because we "are agile" or because we are "doing Scrum". We should only automate when there is value. If there is no value in automating for every work item, then don't automate for every work item. If there is a higher value in creating scripts to generate data than there is to create traditional smoke test scripts, then build the data generation scripts first. I challenge the discipline to be more responsible in how we spend our automation dollars.

Secondly, I challenge the tool vendors to focus more on reducing the effort

of automation maintenance as opposed to reducing the time it takes to create an automation script. While reducing the speed of automation creation is certainly valuable, historically, I find that the cost of maintenance eclipses the cost of creation. In my experience, tool vendors focus on the speed and the ease of creation while largely ignoring the challenge of maintenance. Fortunately, I'm seeing that change with some vendors, but I'd like to see "lowering the cost of maintenance" on the sales sheets for more vendors.

About Paul: As a Principal Automation Architect at Magenic, Paul Grizzaffi is following his passion of providing technology solutions to testing and QA organizations, including automation assessments, implementations, and through activities benefiting the broader testing community. An accomplished keynote speaker and writer, Paul has spoken at both local and national conferences and meetings. He is an advisor to Software Test Professionals and STPCon, as well as a member of the Industry Advisory Board of the Advanced Research Center for Software Testing and Quality Assurance (STQA) at UT Dallas where he is a frequent guest lecturer. Paul enjoys sharing his experiences and learning from other testing professionals; his mostly cogent thoughts can be read on his blog at https://responsibleautomation.wordpress.com/.

Mush Honda

Test Automation should be viewed as an enablement tool for testers; tools such as appium help testers automate the mundane, repetitive checks that must be performed. However, the automated checks done are only as good as the tester who creates these tests; therefore, it is very important to ensure that testers are continuously evolving and learning new methods of testing, as well as performing a deep-dive into the domain of the application under test.

It is very important that testers learn and apply automation tools as part of their testing strategy, since the market demands are simple: Deliver faster, with quality! With the rapid adoption of Continuous Delivery, it is very important that testers apply automation tools to supplement the business assurance testing that is being performed; again, testers should not simply focus on being experts with tools only, they should also be learning to becoming better testers (by learning domains, new testing concepts, etc).

> The testing industry is at a very exciting stage, where it has great support from AI/ML based tools, as well as acknowledgement from the business teams that have realized the important role that testers (and testing) play: we help deliver software with HIGH CONFIDENCE, whether it is for an enterprise's digital transformation or a startup's MVP into the consumer's hand!

About Mush: Mush is a leading expert in the testing industry known for his practice leadership, solutions development and cross-industry expertise. During his career tenure, he has worked with applications in insurance, healthcare, speech analytics and financial services. Further, he has a proven track record of creating, modifying, and innovating on test solutions and bringing them successfully to market. Mush has been featured in several leading industry publications as a thought leader on topics in the QA industry. Mush Honda is the Vice President of Testing at KMS Technology, Inc. He is a driven IT leader with over 15 years of experience in software testing and practice management

Index

---THE END---