

# A Beginner's Guide To Android App Automation Using



---

# Table of Contents

Introduction	1.1
Introduction to Mobile Test Automation	1.2
What is Appium	1.3
Appium 1.6.0	1.4
System Installation	1.5
Installing Android	1.5.1
Installing Appium via app	1.5.2
Installing App on Emulator	1.5.3
Install Appium via npm	1.5.4
Writing First Android Test	1.6
Java - Gradle Project	1.6.1
Understanding Desired Capabilities	1.6.2
Desired Capabilities for Android	1.6.3
Exploring UiAutomatorViewer	1.7
Exploring Appium Inspector	1.8
Remote Debugging on Android via Chrome	1.9
Starting Appium Server	1.10
Executing Test on Real Devices	1.11
Working with Genymotion Emulator	1.12
Automating Gestures	1.13
Tips (With Code Snippets)	1.14
Running multiple Appium Server for parallel execution	1.15
References	1.16

# Introduction

Appium is an Open Source tool for testing native or hybrid mobile apps. It uses WebDriver JSON Wire protocol to drive both iOS and Android apps.

## Why this book ?

There is a decent documentation on the Appium site but somewhere it lacked to produce a step by step guide, which would help you get rolling from scratch. There are bits and pieces which are fairly technical in nature and dev centric.

This book is an effort in the direction of consolidating all the knowledge and resources around appium. It will help software developers and testers successfully use Appium to automate the Android application.

This book will take you through the journey of understanding appium. It would also help you understand how the different pieces come together to build a test automation framework for Android App testing.

***Book also contains lot of code snippets which will help you in your daily automation stuff to get you started with Appium and Java.***

## What I will learn from this book ?

By the end of this book, you would be knowing

- How to **install and set up Appium**
- How to **configure device or Emulator via Desired Capabilities**
- How to **write your first appium** test in Java
- How to **find locators using UIAutomatorViewer and Appium Inspector**
- How to **debug hybrid android app via Chrome browser**
- How to **execute appium test on real Android devices**
- How to **automate gestures** like tap, touch etc...
- How to **run multiple appium server for parallel execution**
- How to **run appium test on GenyMotion Emulator**

This book also contains a git link to an android project which could be ported for any other android application automation with basic alteration.

This is a java project which has been created using **IntelliJ IDEa Community Edition**. POM File manages the dependency of **Selenium**. Project is using **TestNG** annotation. We have also bundled the respective android mobile application under the apps folder for ease.

## Want us to help you with Mobile Test Automation

At [TestVagrant](#), we help Startups with building robust test automation solution be it **Mobile**, **Web** or **Services**. The core idea behind [TestVagrant](#) is to bring an

- **affordable**,
- **niche**
- **rapid**

solution to startups and enterprise.

Please visit [TestVagrant](#) site to request for free POC.

## Queries regarding Appium

In case of any doubts and clarifications about Appium feel free to reach out to [nishant@testvagrant.com](mailto:nishant@testvagrant.com)

Happy Reading!!!

# Introduction to Mobile Test Automation

Before the mobile application development boom, we have explored and tried every bit of web application testing and kind of conquered that space. Wondering what makes mobile app test automation challenging?

Couple of things, which I would want to highlight here:

- Never ending screen size and form factors
- Apps hosted in a sandboxed environment and very limited inter process communication
- Network Challenges
- Usability aspect of mobile and the complexity in terms of navigation
- Lack of easy to use API

While in this book we are not going to discuss all of these but look into one aspect of the challenge which is "*lack of easy to use api*". Appium is very popular because of api's which are similar to that of Selenium.

*All the instructions mentioned here after are tested on **Mac OSX version (10.10)**.*

# What is Appium

**Appium** is an *open source* test automation tool for mobile applications. It allows you to test all the three types of mobile applications: *native*, *hybrid* and *mobile web*.

It also allows you to run the automated tests on actual devices, emulators and simulators.

Today when every mobile app is made in at least two platform iOS and Android, you for sure need a tool, which allows testing cross platform. Having two different frameworks for the same app increases the cost of the product and time to maintain it as well.

The basic philosophy of Appium is that you should be able to reuse code between iOS and Android, and that's why when you see the API they are same across iOS and android. Another important thing to highlight here is that unlike Calabash, Appium doesn't modify your app or need you to even recompile the app.

Appium let's you choose the language you want to write your test in. It doesn't dictate the language or framework to be used.

## Architecture

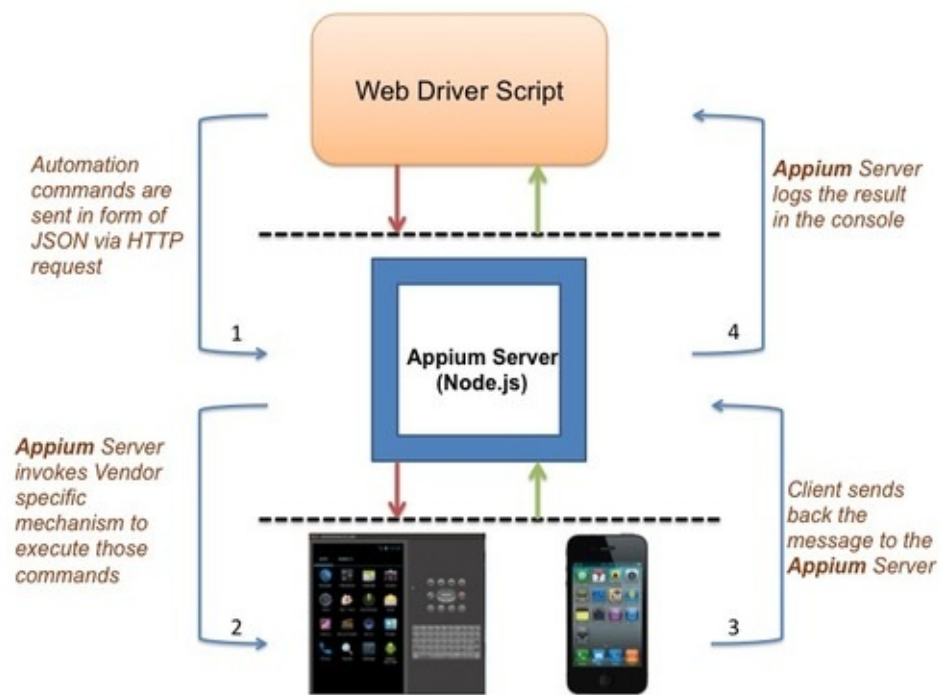
So how does Appium do all this? Let's try to understand what happens behind the scenes.

When you download the Appium app you are basically downloading the server. The server is written in Node.js and implements the Selenium WebDriver. It allows one to use available WebDriver client to fire your tests. And then your mobile app acts precisely like a web app where the DOM is represented by View hierarchy.

So this server basically exposes REST api which performs the following actions:

- 1) Receives connection from client
- 2) Listen command
- 3) Execute command
- 4) Respond back the command execution status

In terms of architecture diagram, below is how Appium can be explained.



# What is new in Appium 1.6.0

What's new in Appium 1.6 ?

- **UiAutomator 2**

With release of appium 1.6.0 , appium has started using **UiAutomator 2**

Till now appium was primarily dependent on Google's UiAutomator framework as the primary way of automating native Android apps.

To use UiAutomator 2, you can specify

```
automationName: uiautomator2
```

in your desired capabilities and try out the new driver.

Prerequisites: This module should support from Android 5.0 (API Level 20) and above

For more info, check out the wiki article here: <https://github.com/appium/appium-uiautomator2-server/wiki>

- **XCUITest**

Appium has added support for Apple's new XCUITest framework. When you specify a **platformVersion** of **10** or higher in Capabilities Builder, Appium automatically uses the XCUITest automation backend.

If you want to run your iOS test on version 9.3, then you need to specify

```
automationName: XCUITest
```

in your desired capabilities.



# Installing Appium

Continuing from the last chapter, which was more to explain what Appium is about and the explanation of architecture. This chapter will detail how to write your first test and execute that.

Before installing Appium, you need to get few other things in ecosystem:

- Android SDK
- Install Appium
- IDE of your choice (Eclipse or IntelliJ IDEA)

I am assuming that **JDK installation** is already there and you have the latest version of java installed.

# Installing Android

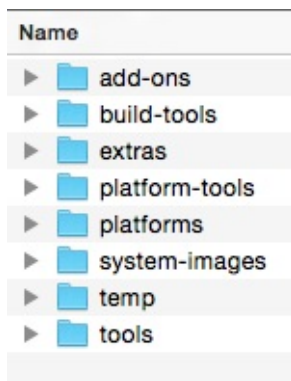
## Android Requirements

- Android SDK API  $\geq 17$  (Additional features require 18/19)
- Appium supports Android on OS X, Linux and Windows.

**Android SDK** needs to be downloaded from the following location.

```
http://developer.android.com/sdk/index.html
```

Once you download the file and unzip, you would see the below structure in the sdk folder.



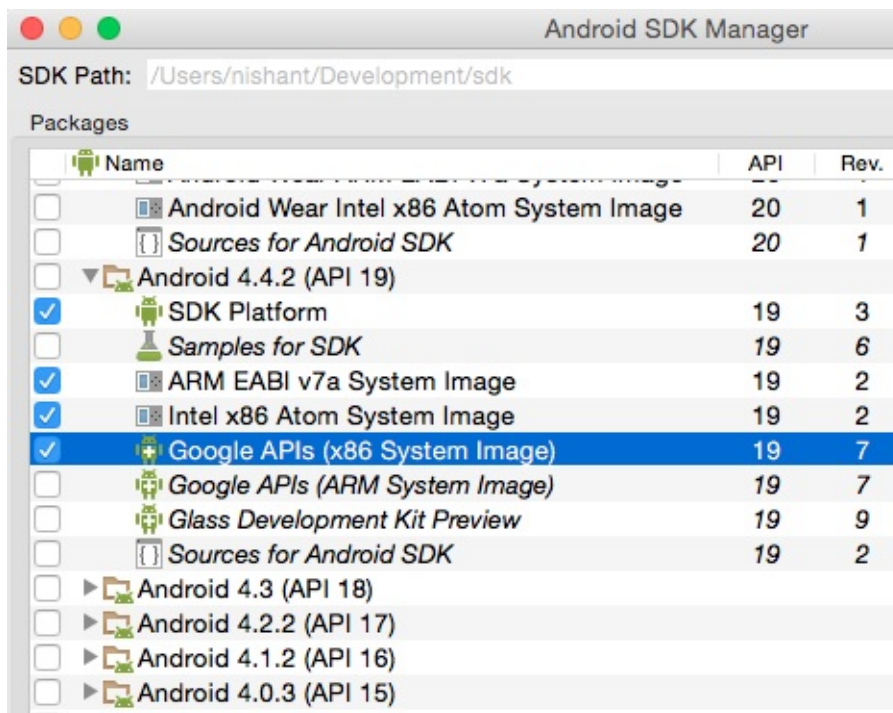
Post unzipping, update the system variable to include sdk as well. For mac users below are the command snapshot (change the path and folder name accordingly as per your machine)

```
export ANDROID_HOME=/Users/Steve/Development/SDK
export PATH =${PATH}:${ANDROID_HOME}/tools:${ANDROID_HOME}/platform-tools
```

Once this is done, we need to launch the SDK manager to download relevant files to create an emulator and run the virtual device. In terminal, type in

```
android sdk
```

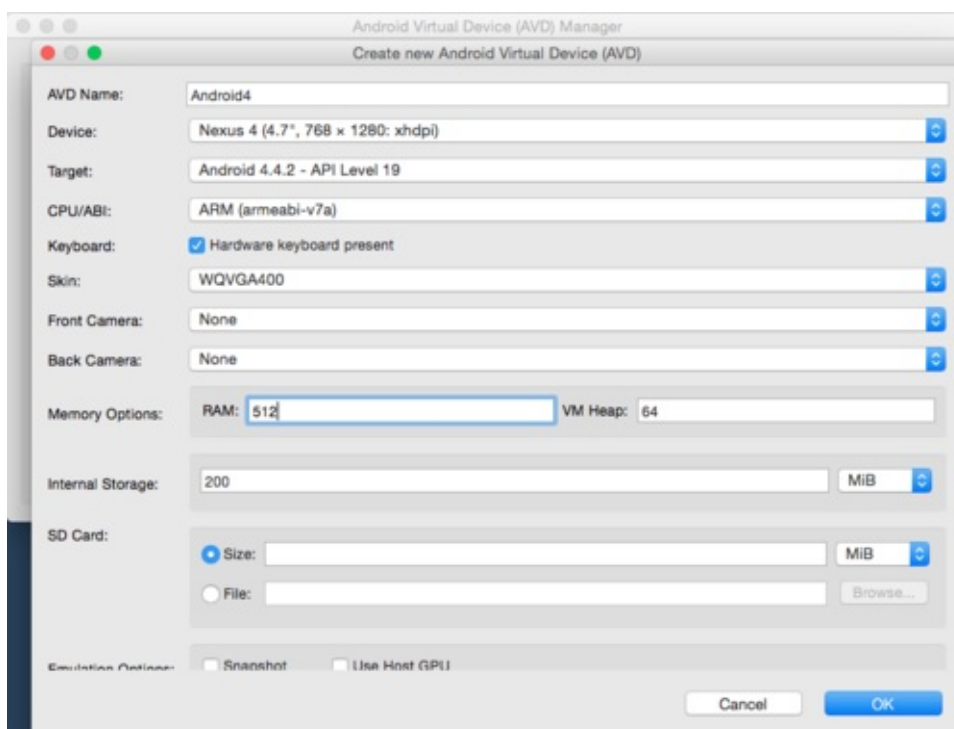
This would open the SDK manager for you to download the relevant files. In the SDK manager select the below files as shown below. This will help you create a virtual device running Android 4.4.2.



Select the above files and choose to install, once done close the SDK manager. In terminal then type

```
android avd
```

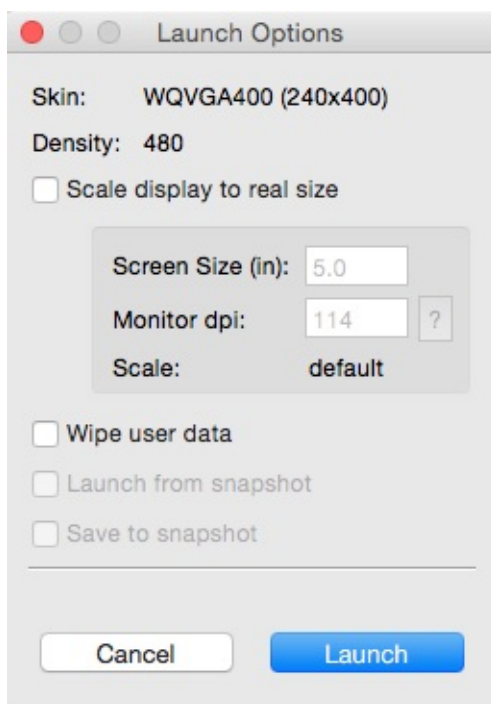
This would open the Android Virtual Device Manager, which would help you create the virtual devices. Click on **Create** on the pop up opened and follow the below snapshot for the values. Select the **RAM** size as per your machine configuration.



Once you have selected the above options, click on OK. Once done, it will show up in the AVD manager as below.



Select the AVD name and click on Start on the right. This would launch the pop up with few options, you may choose as you want.



"Wipe user data" would wipe any previous app installation you have done and would launch a plain vanilla emulator.

Once done click on Launch, this will launch the emulator.

You can use the command "`adb devices`" to see if the adb is detecting the emulator. This basically completes the android SDK installation part.

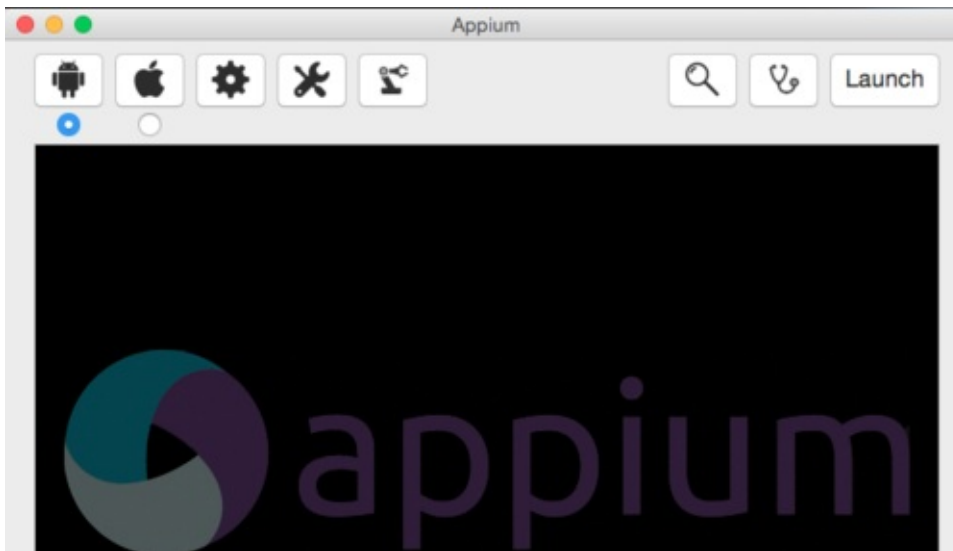


# Installing Appium

Download the Appium installer from the below mentioned location. Once done extract it and install the app.

```
https://github.com/appium/appium/releases
```

You can select the Android icon from the top and click on Launch.



Once the Appium server is running you will see some logs in the app screen or alternately hit the below url to check the status.

```
http://localhost:4723/wd/hub/status
```

Appium can also be run via npm install. For this you would need node.js and npm version (0.10 or greater). In order to verify if the appium is installed correctly and it's dependencies run the command

```
appium-doctor
```

This command can be supplied with flag `--android` to verify all the dependencies are set up properly.

The appium clients are simple extension to the WebDriver client.



# Installing App on Emulator

Once your emulator is ready and you have the android path set up properly, the next action would be to install an app on the Emulator which would become your application under test.

The first step is to run the emulator on which you want to install the app.

```
emulator -avd <AVD_Name>
```

This would launch the emulator and the command would be running in the terminal.

There are couple of ways to install the app:

- install via adb command
- drag and drop

## Install via adb command:

- Download the apk and put it in some folder
- Run the following command

```
adb install path/to/your/app.apk
```

when more than one emulator is open, run the following command

```
adb -s serial-number-of-device install path/to/your/app.apk
```

## Drag and Drop

- You can alternately drag the apk file and drop on to the emulator.



# Install Appium via Source

One of the alternative way to install Appium is to install via npm (Node JS Package Manager).

But before that you need to have below items installed

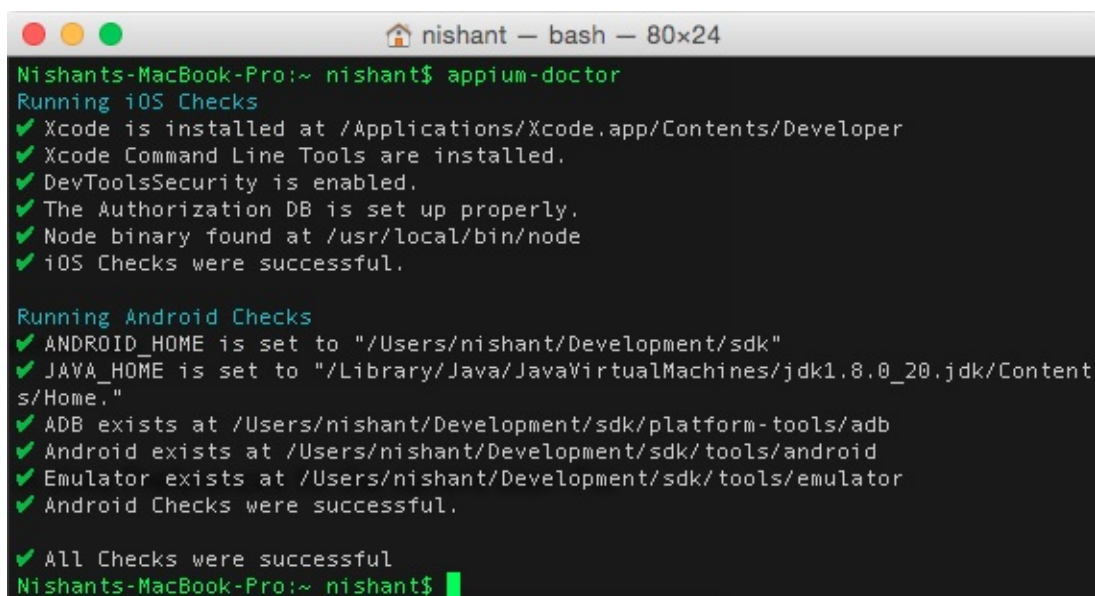
- npm (version .10 or greater)
- node.js

Brew will be very handy here to install the above stuff. It's an amazing package manager for OS X and installing packages and updating it will be breeze once you install brew.

Once the node.js and npm are installed, you can use the below command to check if all the appium dependencies are met, run the below command:

```
appium-doctor
```

It would show up a result like this if everything is okay:

A terminal window titled 'nishant — bash — 80x24' showing the output of the 'appium-doctor' command. The output is as follows:

```
Nishants-MacBook-Pro:~ nishant$ appium-doctor
Running iOS Checks
✓ Xcode is installed at /Applications/Xcode.app/Contents/Developer
✓ Xcode Command Line Tools are installed.
✓ DevToolsSecurity is enabled.
✓ The Authorization DB is set up properly.
✓ Node binary found at /usr/local/bin/node
✓ iOS Checks were successful.

Running Android Checks
✓ ANDROID_HOME is set to "/Users/nishant/Development/sdk"
✓ JAVA_HOME is set to "/Library/Java/JavaVirtualMachines/jdk1.8.0_20.jdk/Contents/Home."
✓ ADB exists at /Users/nishant/Development/sdk/platform-tools/adb
✓ Android exists at /Users/nishant/Development/sdk/tools/android
✓ Emulator exists at /Users/nishant/Development/sdk/tools/emulator
✓ Android Checks were successful.

✓ All Checks were successful
Nishants-MacBook-Pro:~ nishant$
```

Once done run the below command to install the appium:

```
npm install -g appium
```

Once this is done, start the appium server by the following command:

```
appium &
```



# Write your first Android Test

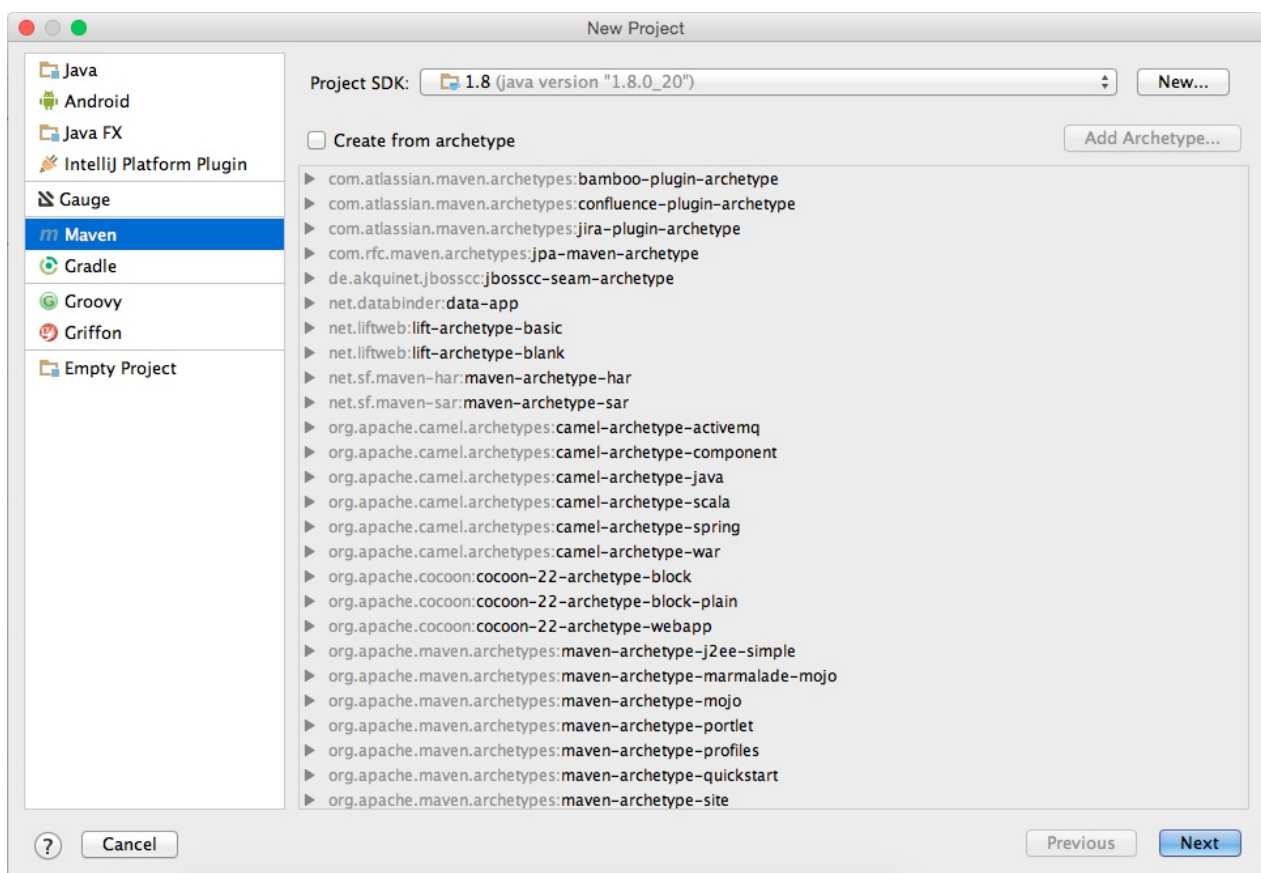
We are using IntelliJ Idea 13 CE for this example.

Create a new maven project and update the pom file with below mentioned Selenium dependency. I am assuming you are familiar with Java project creation in Eclipse or IntelliJ. Let me include a brief step by step guide to create a project.

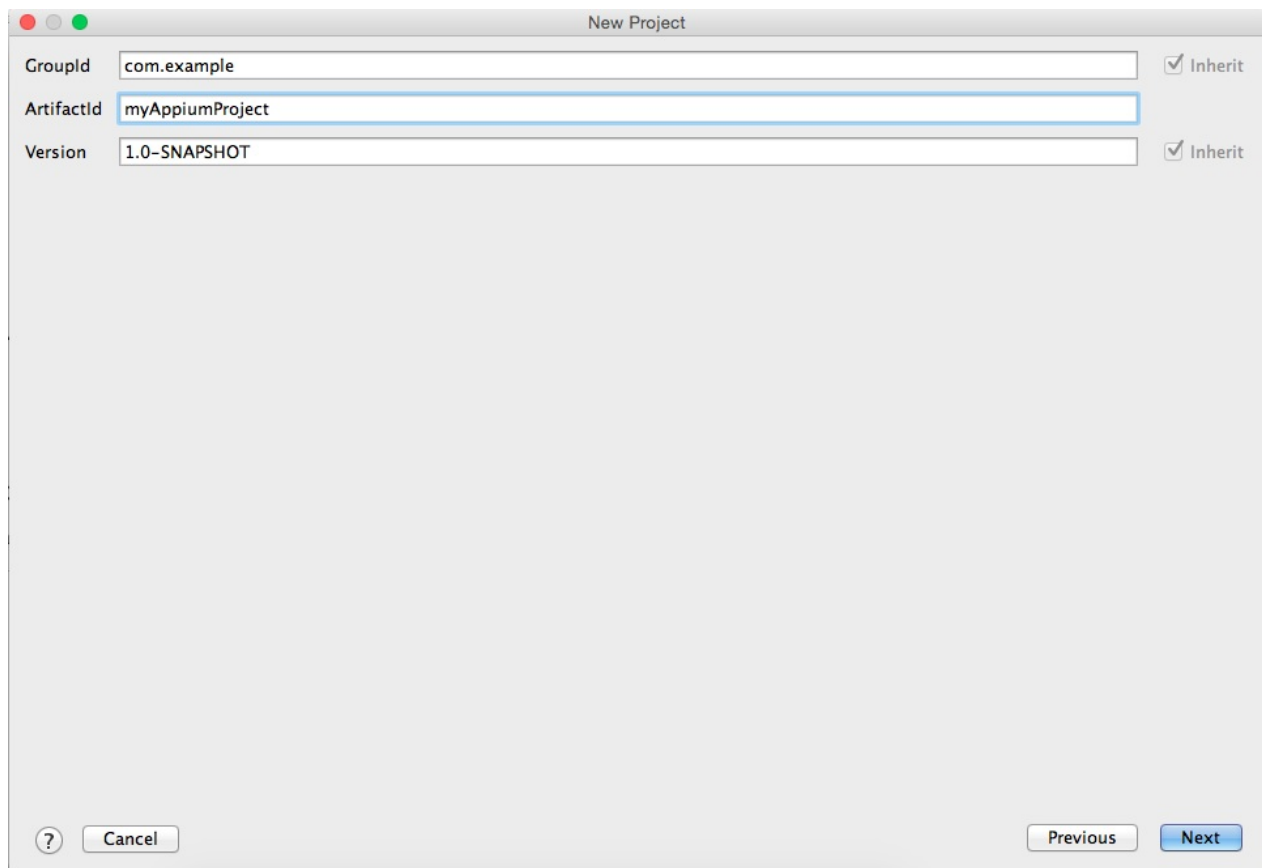
## Creating a Maven Project:

You can choose to create a Maven project in IntelliJ IDEA. This is the recommended way when using the Community Edition.

- Select New Project
- In the **New Project** window, select **Maven** on the left side of the screen as shown below. If the SDK is already defined in IntelliJ then it will pick up from there, else click New and select the installation folder of the desired JDK.



- Click **Next**
- Give a Maven **GroupID**, **ArtifactID**, and a **Version** for the project, or use the defaults.



New Project

GroupId  ☒ Inherit

ArtifactId  ☒ Inherit

Version  ☒ Inherit

? Cancel Previous Next

- Click **Next**
- Enter Project name "AppiumDemo" and Project Location "Your Desired Location"
- Click on **Finish**.

Once the project is created, copy the below dependencies in the pom.xml file.

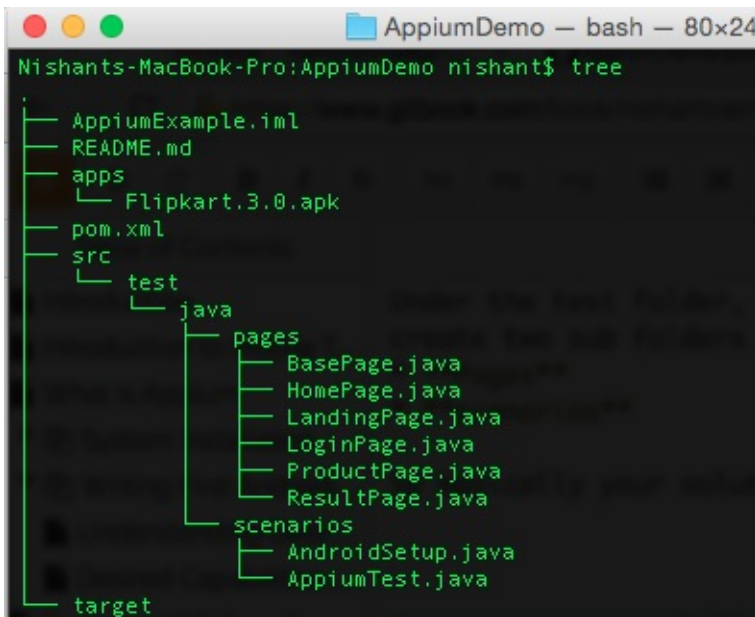
```
<dependencies>
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>2.46.0</version>
  </dependency>
  <dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>6.1.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>io.appium</groupId>
    <artifactId>java-client</artifactId>
    <version>3.0.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Once the project is created, create folder called **apps** and put the .apk file there which is your application under test.

Under the test folder, create a folder called "java", and create two sub folders

- **pages**
- **scenarios**

Sp basically your solution would look like below:



## Creating the Appium Set up

Add a Java Class file under scenarios folder and name it AndroidSetup. Add the below content to the class file.

```

protected AndroidDriver driver;

protected void prepareAndroidForAppium() throws MalformedURLException {
    File appDir = new File("/Users/nishant/Development/AppiumDemo/apps");
    File app = new File(appDir, "Flipkart.3.0.apk");
    DesiredCapabilities capabilities = new DesiredCapabilities();
    capabilities.setCapability("device", "Android");

    //mandatory capabilities
    capabilities.setCapability("deviceName", "Android");
    capabilities.setCapability("platformName", "Android");

    //other caps
    capabilities.setCapability("app", app.getAbsolutePath());
    driver = new AndroidDriver(new URL("http://127.0.0.1:4723/wd/hub"), capabilities);
}
  
```

## What do these steps do ?

- installs the apk file (application) on the connected device,
- start the application
- inform the appium server which kind of session we are interested in (via Desired Capabilities)
- give you the driver object which is of `AndroidDriver` type

Once you get the driver, from there on it's more like the Selenium usage of the appium driver and familiar method calls.

## What next from here?

The next logical step from here is to create Page class files for different pages in the app and then create an according test class file. However to test the code just written, you can write a small piece of code as below:

```
@Test
public void helloTest()
{
    String app_package_name = "com.flipkart.android:id/";
    By userId = By.id(app_package_name + "user_id");
    By password = By.id(app_package_name + "user_password");
    By showPassword = By.id(app_package_name + "show_password");
    By login_Button = By.id(app_package_name + "btn_login");

    driver.findElement(userId).sendKeys("someone@testvagrant.com");
    driver.findElement(password).sendKeys("testvagrant123");
    driver.findElement(showPassword).click();
    driver.findElement(login_Button).click();
}
```

I am sure you would have question that why we are using "**app\_package\_name**", which is explained in the chapter "**Exploring UiAutomatorViewer**".

Post this set up method you can write your normal automation test (Selenium types). Sample code would look like:

```
driver.findElement(userId).sendKeys("someone@testvagrant.com");
driver.findElement(password).sendKeys("password");
driver.findElement(showPassword).click();
driver.findElement(login_Button).click();
```

We have created a project in github to help you with the basic framework setup using Appium. Below is the github link.

```
https://github.com/testvagrant/AppiumDemo
```

The above project uses Flipkart android application, which is bundled along with and is present in the "apps" folder. The code is tested on this version of Flipkart app and works well to show the concepts.

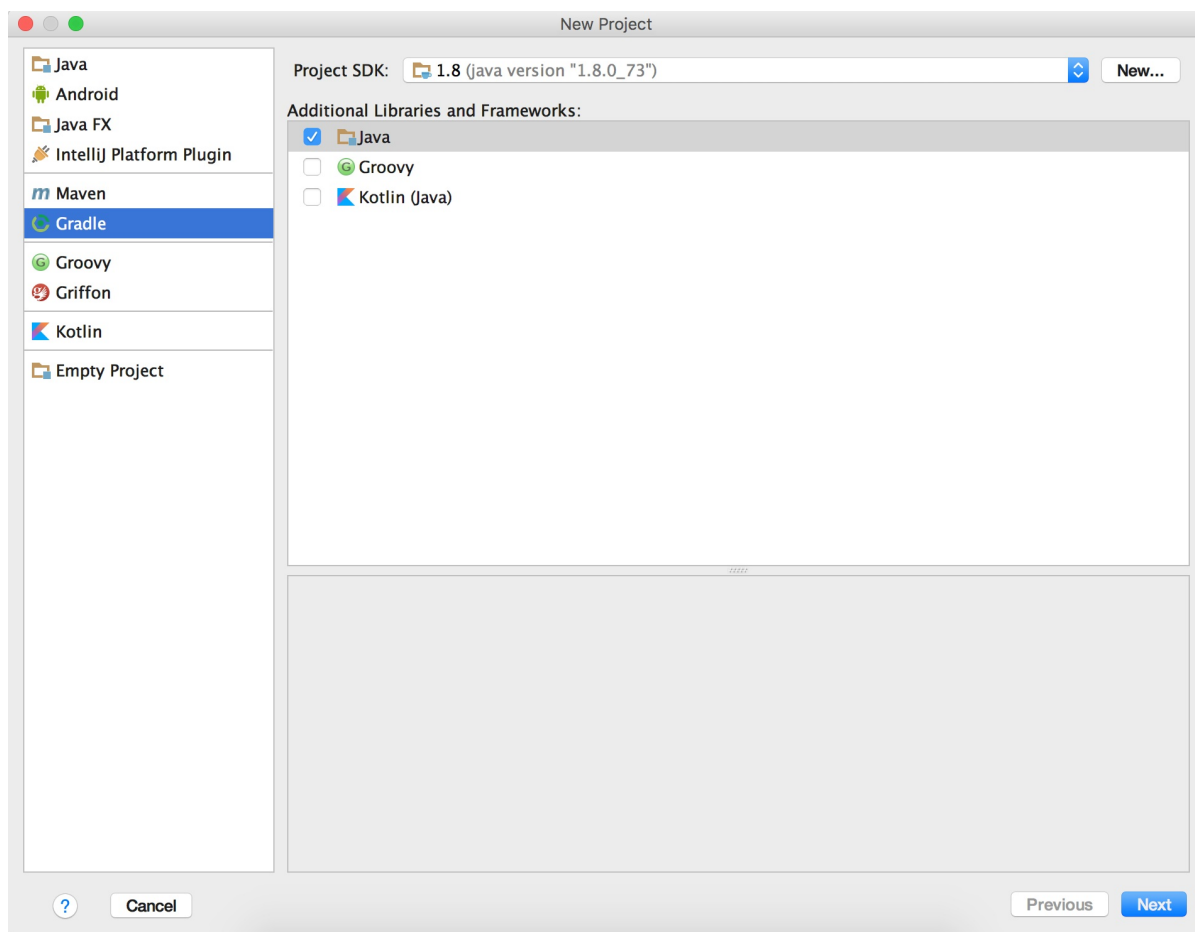
In the coming chapters we will understand **Desired Capabilities** in much details.

# Cucumber-JVM-Appium - Gradle Project

## Creating Gradle Project

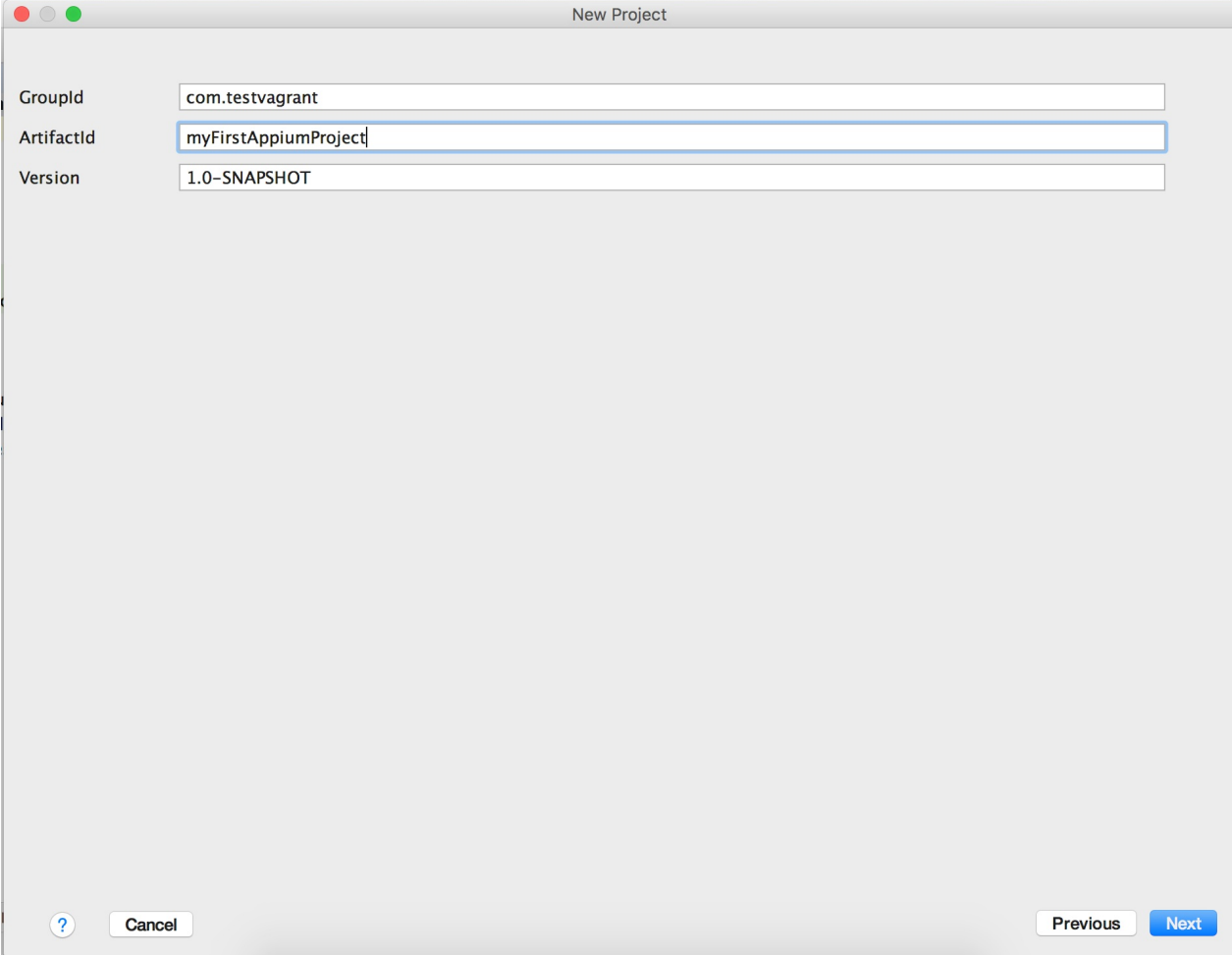
You can choose to create a gradle project. Below steps will guide you for the same.

- Select **New Project**
- In the New Project window, select **Gradle** on the left side of the screen as shown below. If the SDK is already defined in IntelliJ then it will pick up from there, else click New and select the installation folder of the desired JDK.



- Click **Next**
- Give a **GroupID**, **ArtifactID**, and a **Version** for the project. Click Next.



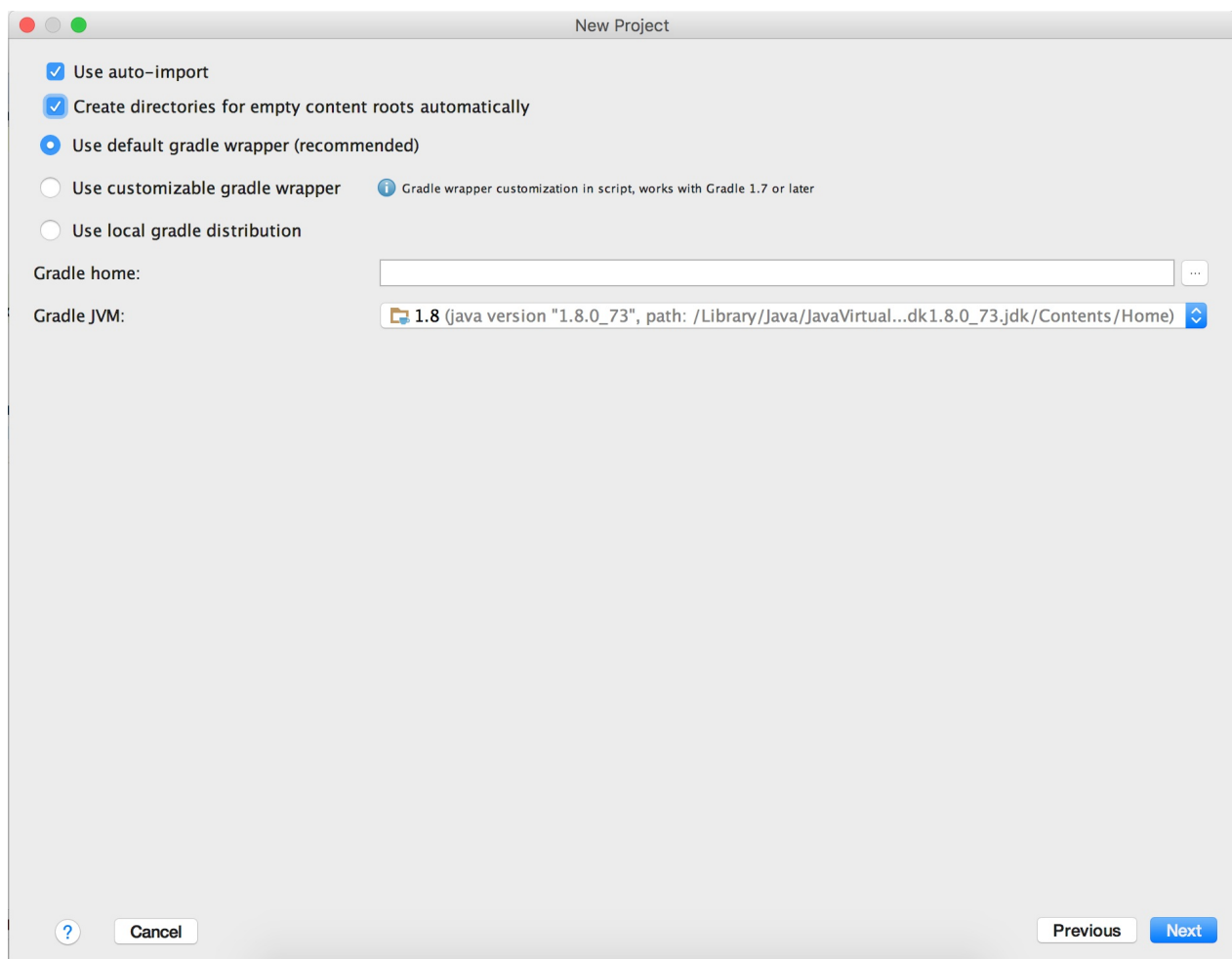


The screenshot shows a 'New Project' dialog box with the following fields and values:

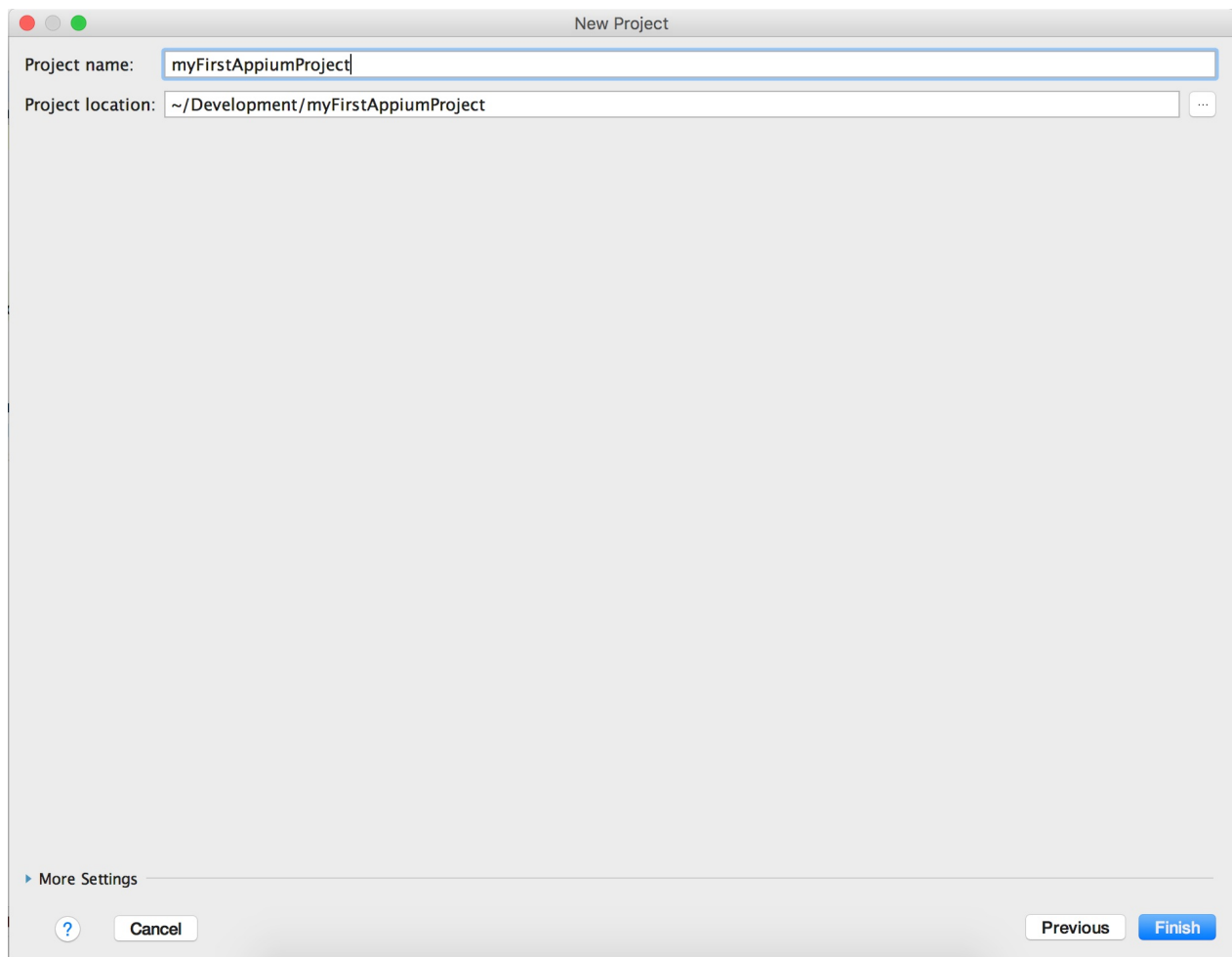
Field	Value
GroupId	com.testvagrant
ArtifactId	myFirstAppiumProject
Version	1.0-SNAPSHOT

At the bottom of the dialog, there are four buttons: a help button (question mark), a 'Cancel' button, a 'Previous' button, and a 'Next' button.

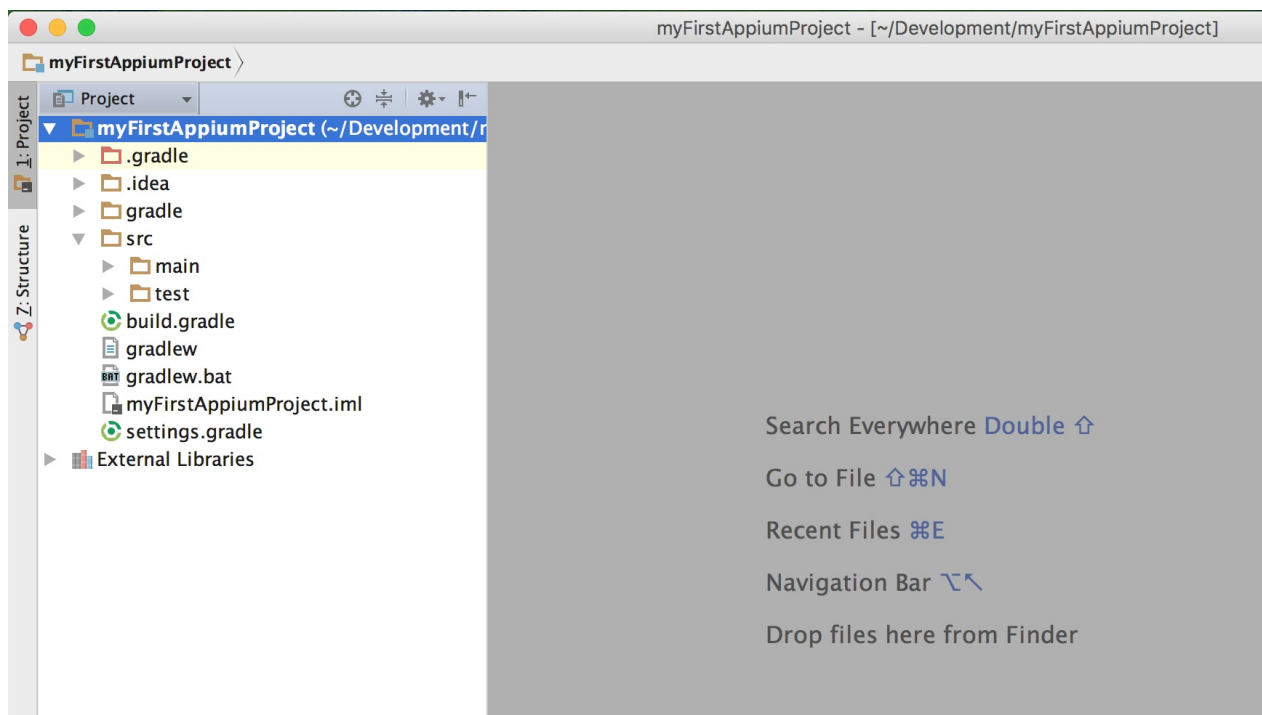
- Select the option of **"Use auto-import"** and **"Create directories for empty content..."**. Click Next



- Lastly enter a **project name** and **project location** and click on **Finish**.



It should create a project with following structure.



## Adding dependencies to build.gradle

Open build.gradle file and copy the below two lines

```
compile 'io.appium:java-client:4.0.0'
```

```
``testCompile 'info.cukes:cucumber-java:1.2.4',  
              'info.cukes:cucumber-junit:1.2.4'
```

## Adding app folder

Once the project is created, created folder called **apps** and put the **.apk** file there which is your application under test.

# Understanding Desired Capabilities

Before we start writing further test code to automate and do other stuff, we need to understand why we are using Desired Capabilities.

Desired Capabilities are present in the below library and hence it needs to be imported.

```
org.openqa.selenium.remote.DesiredCapabilities
```

Desired Capabilities is a way of telling the Appium Server which kind of session we are interested in. So it's a hash of Key and Value to let us have more control on the server during automation. So some of the key info which we will be using are :

```
automationName    Which automation engine to use    Appium (default) or Selendroid
platformName      Which mobile OS platform to use     iOS, Android, or FirefoxOS
```

Similarly, we can specify *platformVersion* which will be the version of OS running on the device, could be "8.3" or "5.1".

Below is one such table which contains the various Desired Capabilities which would be set at the server level and customized to either Android or iOS.

Capability	Description	Values
automationName	Which automation engine to use	Appium (default) or Selendroid
platformName	Which mobile OS platform to use	iOS, Android, or FirefoxOS
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...
app	The absolute local path or remote http URL to an .ipa or .apk file, or a .zip containing one of these. 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	/abs/path/to/my.apk or <a href="http://myapp.com/app.ipa">http://myapp.com/app.ipa</a>

	appActivity capabilities (see below). Incompatible with browserName.	
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
autoLaunch	Whether to have Appium install and launch the app automatically. Default true	true, false
language	(Sim/Emu-only) Language to set for the iOS Simulator	e.g. fr
locale	(Sim/Emu-only) Locale to set for the iOS Simulator	e.g. fr_CA
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

## Desired Capabilities for Android

Since **Appium** caters to both Android and iOS, there are different set of desired capabilities for Android and iOS.

This section will list all the desired capabilities associated with Android. Majority of them are optional but you can choose to use them as it suits your needs.

Capability	Description	
appActivity	Activity name for the Android activity you want to launch from your package	MainActivity,
appPackage	Java package of the Android app you want to run	com.example
appWaitActivity	Activity name for the Android activity you want to wait for	SplashActivit
appWaitPackage	Java package of the Android app you want to wait for	com.example
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/
enablePerformanceLogging	(Chrome and webview only) Enable Chromedriver's performance logging (default false)	true, false
androidDeviceReadyTimeout	Timeout in seconds used to wait for a device to become ready after booting	e.g., 30
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.
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
	How long to wait in milliseconds for	

avdReadyTimeout	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
keystorePassword	Password for custom keystore	e.g., foo
keyAlias	Alias for key	e.g., android
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/
specialChromedriverSessionArgs	Custom arguments passed directly to chromedriver in chromeOptions capability. Passed as object which properties depend on a specific webdriver.	e.g., {'android', 'opera_beta_
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.i
intentCategory	Intent category which will be used to start activity (default android.intent.category.LAUNCHER)	e.g. android.i android.inten
intentFlags	Flags that will be used to start activity (default 0x10200000)	e.g. 0x10200
optionalIntentArguments	Additional intent arguments that will be used to start activity. See Intent arguments	e.g. --esn , --
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 used alone. Default false	true or false



If you notice above some of the capabilities like *appWaitActivity* , *avd*, *androidDeviceReadyTimeout* are very handy and would be recommended to make use of in your automation suite.

# How to get locators via UiAutomatorViewer

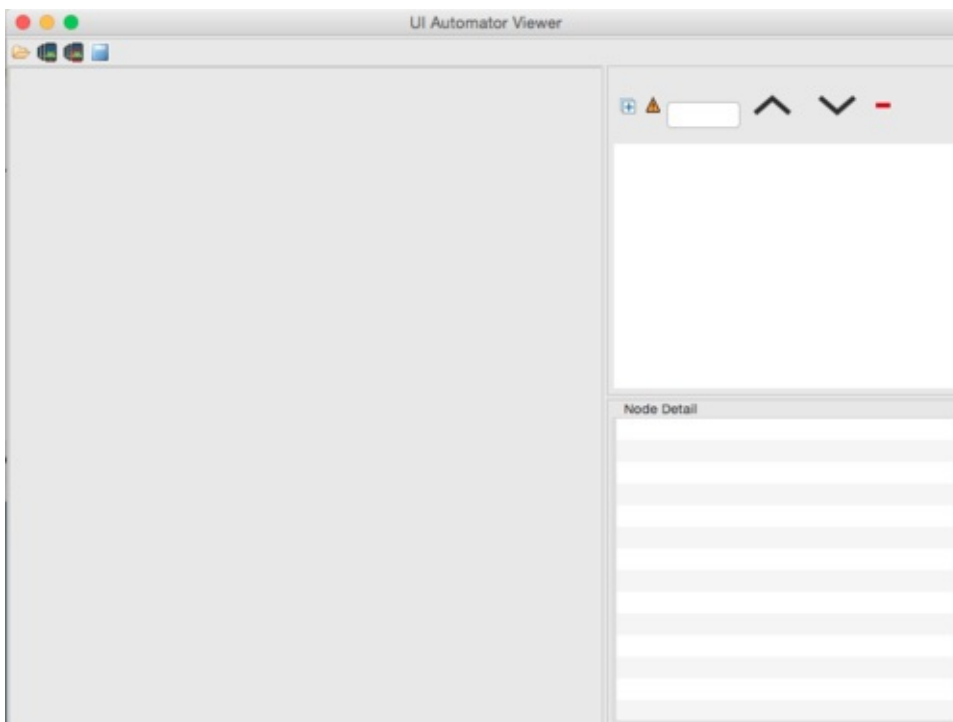
**UiAutomatorviewer** comes packaged with Android sdk and is present under “*tools*” folder. It’s a tool, which lets you inspect the UI of an application in order to find the layout hierarchy, and view the properties associated with the controls.

While designing your UI automation suite, this tool is very helpful as it exposes the Id and other attributes of an element, which is needed for writing scripts.

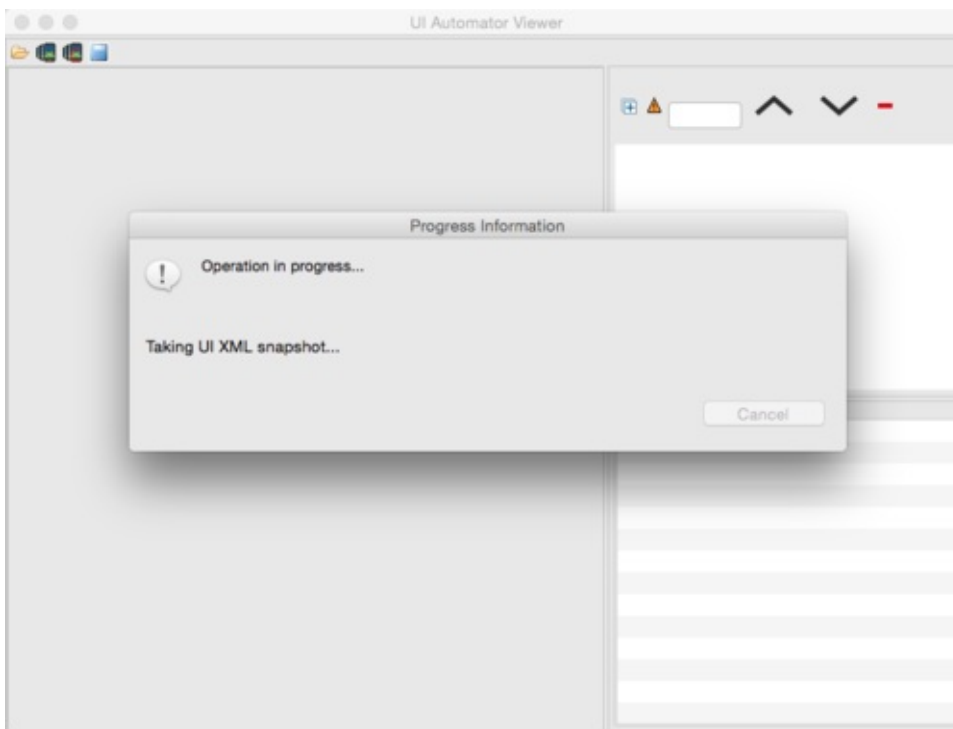
Once the android sdk path is set on the machine, open the terminal and type

```
uiautomatorviewer
```

and this would launch the UI inspection tool. Below is how the tool looks like when no app is monitored.



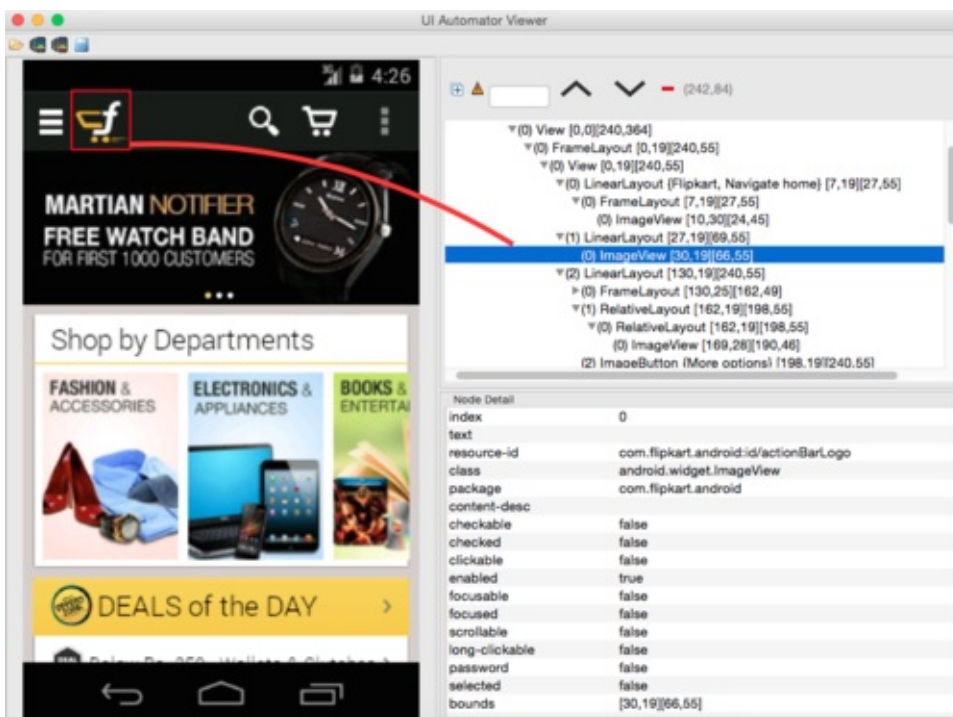
Clicking on the devices icon from left takes a dump of UI XML snapshot of the screen opened in the device. So you will be presented with an intermediate screen like this:



Once it finishes loading, this is how the tool looks like; we have highlighted the logo of Flipkart and the respective attribute details of the logo.

The left side of the tool shows how the device looks like and the right side is divided in two parts:

- Upper half to show the UI XML snapshot and the nodes structure
- Lower half shows the details of the node selected with all the attributes.



So if you carefully look below, you would see the resource-id, which roughly translates to id of the HTML. And then there are other details like what is the class name, package it belongs to, whether it is enabled, whether it is selected etc.

Node Detail	
index	0
text	
resource-id	com.flipkart.android:id/actionBarLogo
class	android.widget.ImageView
package	com.flipkart.android
content-desc	
checkable	false
checked	false
clickable	false
enabled	true
focusable	false
focused	false
scrollable	false
long-clickable	false
password	false
selected	false
bounds	[30,19][66,55]

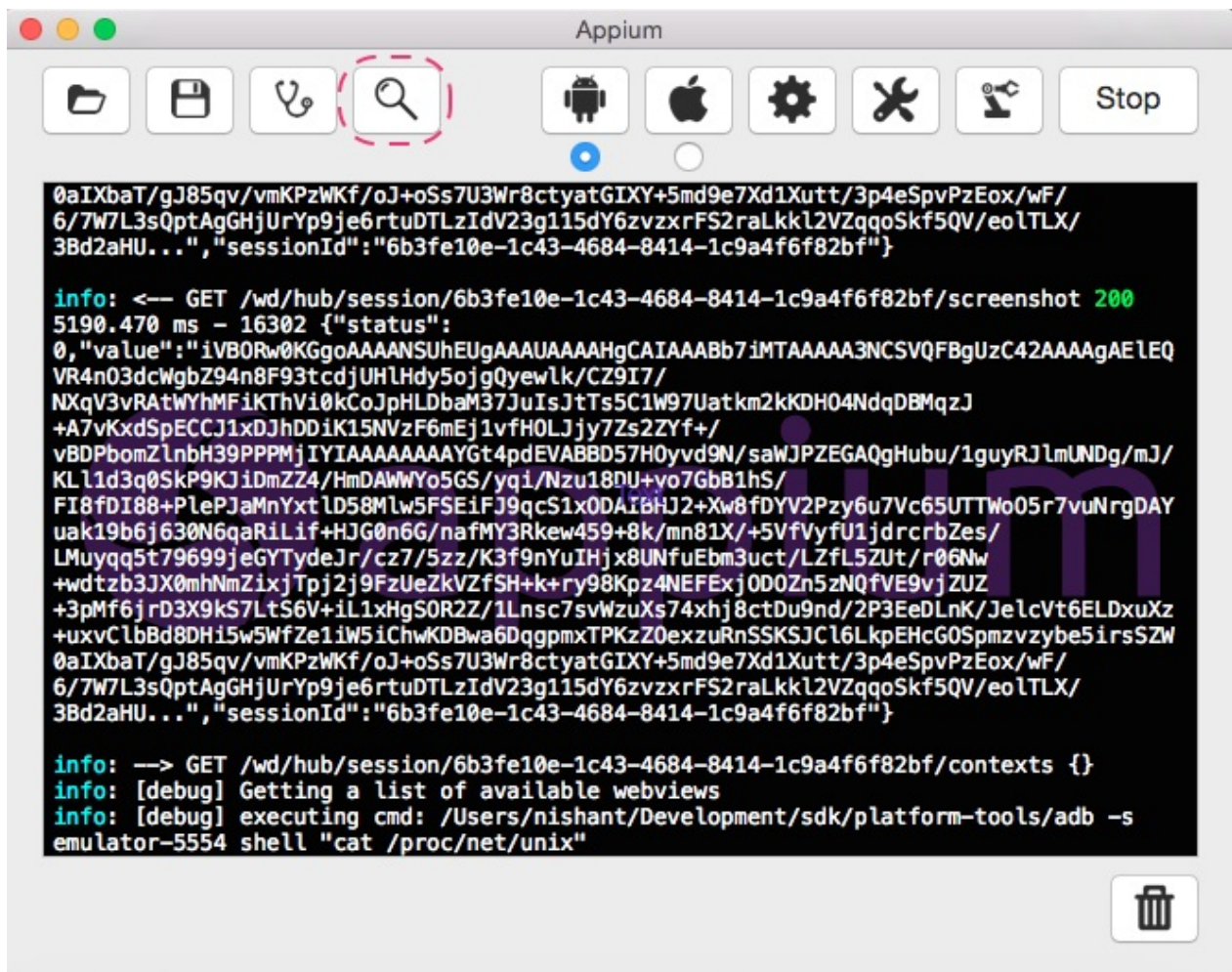
Apart from this all the UI elements on the left can be clicked upon and the right side changes dynamically to show the details of the element selected.

This detail pretty much about the UiAutomatorviewer.

# How to get locators via Appium Inspector

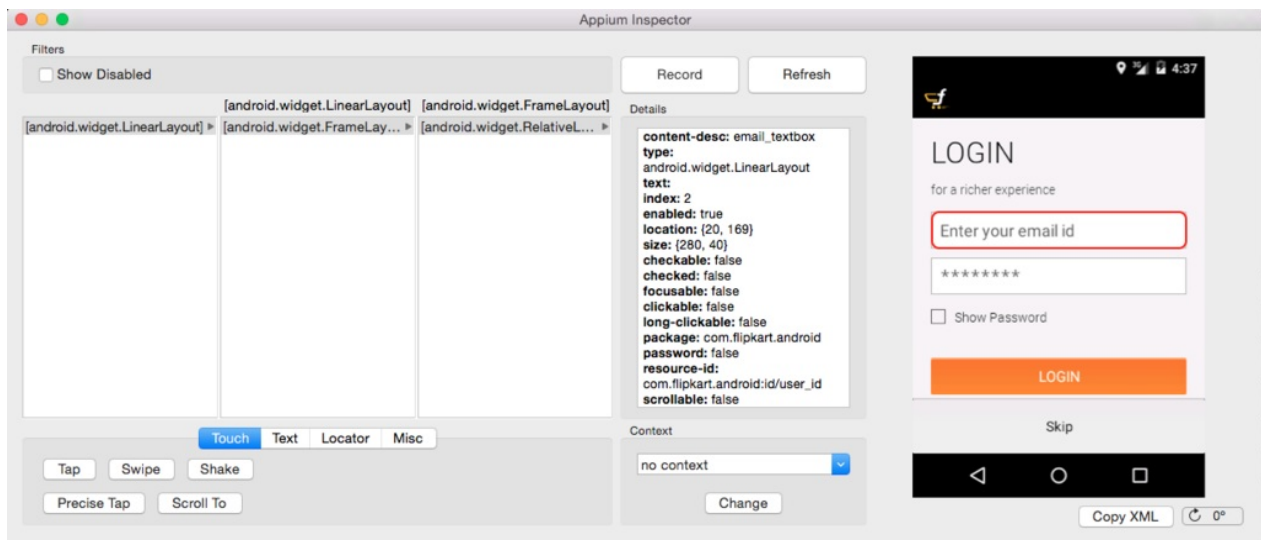
In the previous chapter we saw **UiAutomatorViewer** which comes bundled with Android SDK. Appium has built something like that which makes our work of finding locators very easy.

When you start the Appium app, you would notice the icons at the top (image below), I have highlighted the Appium Inspector in red.

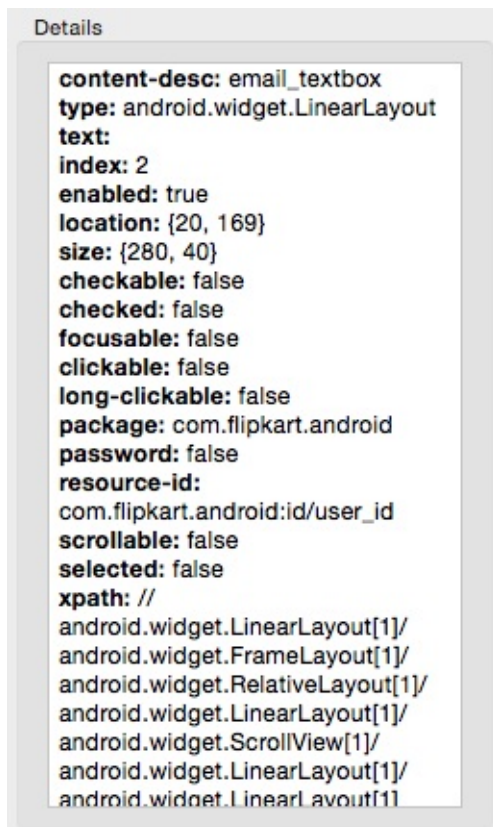


So if you notice that the appium server is running and the log shows that it's connected to the emulator running locally.

When you click on the circled icon (Appium Inspector), it will launch a new UI as shown below with the application UI state captured.



The panel on the extreme right is clickable and you can click on the element you want to find details about. So if you see the panel with title "**Details**", there are bunch of attributes listed for the highlighted element.



If you notice the attribute "**resource-id**", it's made of the package name "**com.flipkart.android**" and the value of id attribute.

This **package name** is present for all the elements and hence we can externalize it. So to construct an element identifier for automation, we are going to use it as shown below:

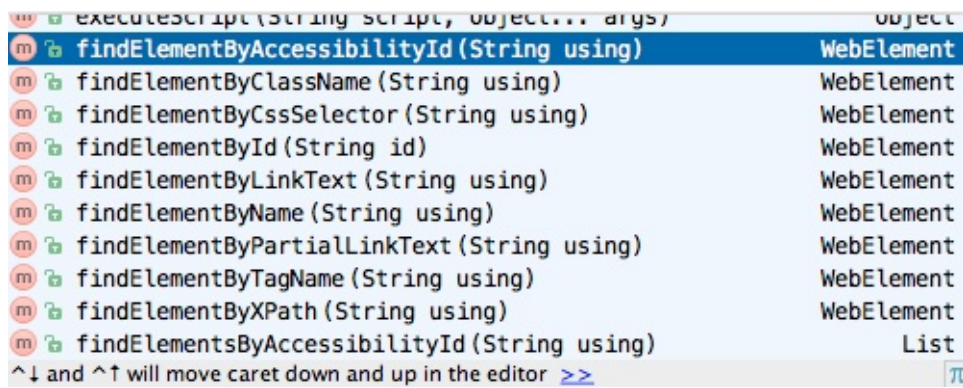
```
String app_package_name = "com.flipkart.android:id/";
By userId = By.id(app_package_name + "user_id");
```



So the code would look like:

```
driver.findElement(userId).sendKeys("someone@testvagrant.com");
```

Also you can use other options like xpath, index etc. Below is the snapshot of all the options you have for identifying the element.

A screenshot of the Appium Inspector interface showing the 'find element' dropdown menu. The menu lists various methods for locating web elements, each with a magnifying glass icon and a red circle icon. The methods are: findElementByAccessibilityId, findElementByClassName, findElementByCssSelector, findElementById, findElementByLinkText, findElementByName, findElementByPartialLinkText, findElementByTagName, findElementByXPath, and findElementsByAccessibilityId. The return types are listed on the right: WebElement for most methods and List for findElementsByAccessibilityId. At the bottom, there is a note: '^↓ and ^↑ will move caret down and up in the editor' followed by a blue double arrow icon.

	Object
findElementByAccessibilityId (String using)	WebElement
findElementByClassName (String using)	WebElement
findElementByCssSelector (String using)	WebElement
findElementById (String id)	WebElement
findElementByLinkText (String using)	WebElement
findElementByName (String using)	WebElement
findElementByPartialLinkText (String using)	WebElement
findElementByTagName (String using)	WebElement
findElementByXPath (String using)	WebElement
findElementsByAccessibilityId (String using)	List

^↓ and ^↑ will move caret down and up in the editor >>

# How to debug hybrid android app via Chrome browser

In the previous chapter we saw **UIAutomatorviewer** and how to use that tool to look for elements and locators.

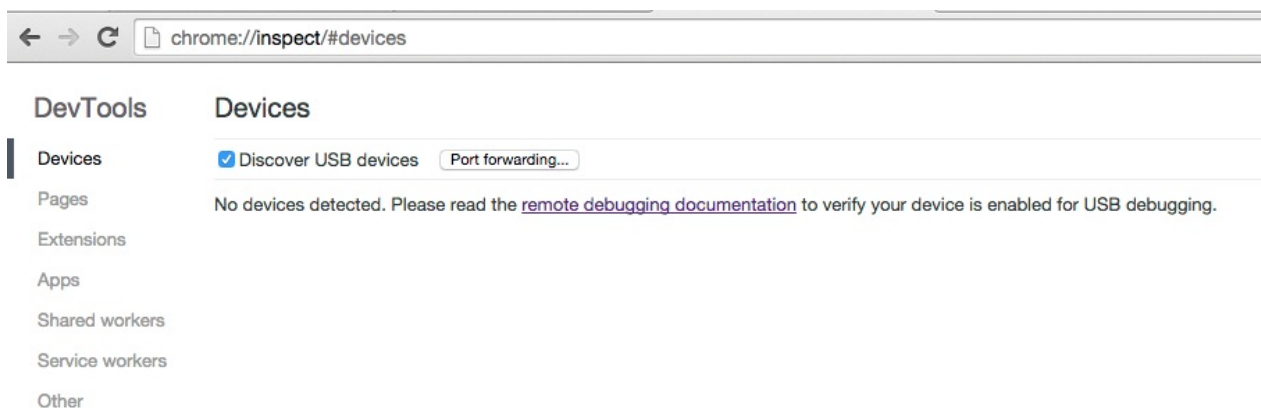
However, when you are working on a hybrid application (developed on a platform like [Phonegap](#) ), we can use remote debugging.

To use Remote debugging, we need to have:

- Chrome (version 32) or later installed on your machine
- Android Device
  - Should be running Android 4.4+
  - USB Debugging enabled
- USB cable

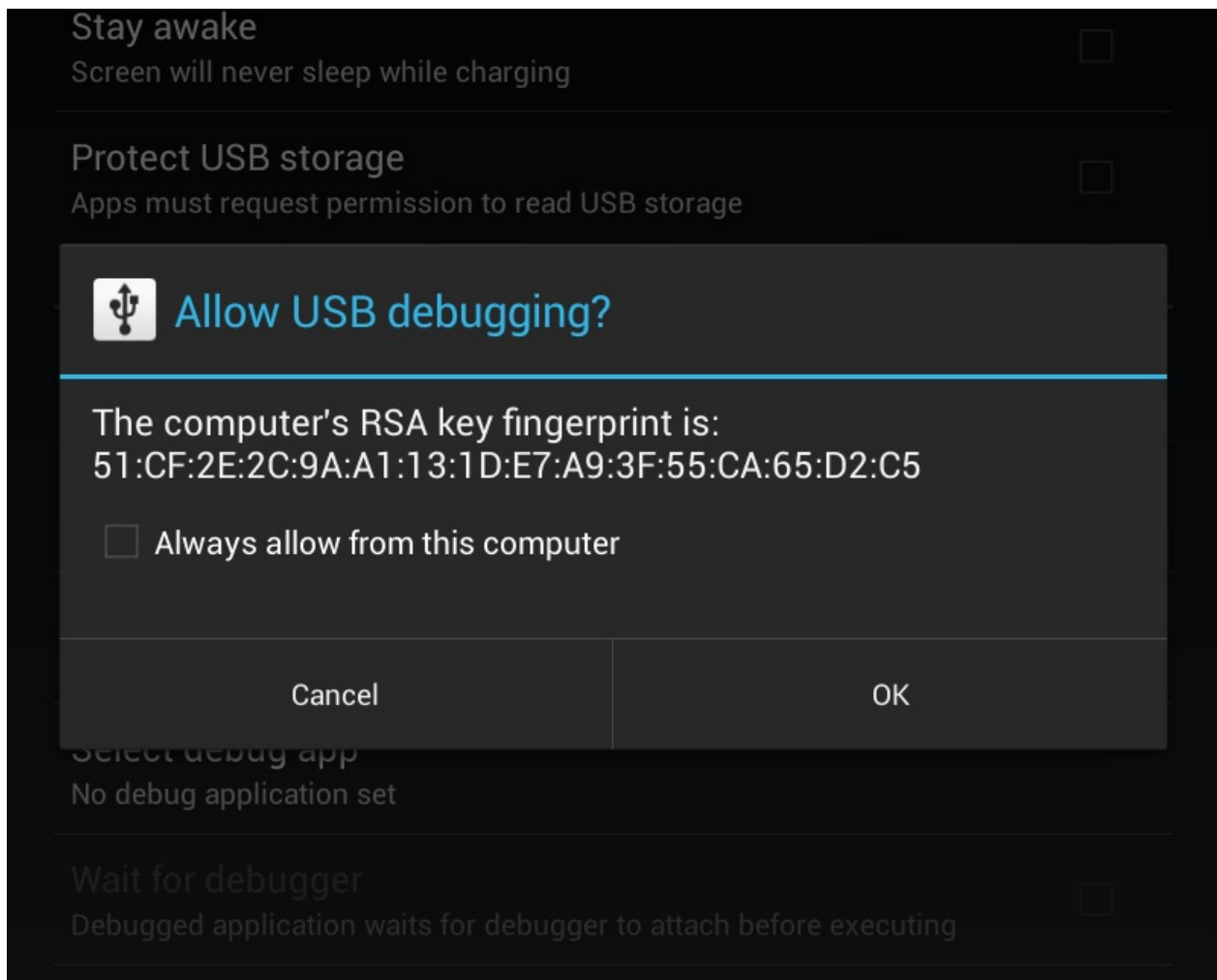
Once the basic things are in place, connect your device to machine using USB cable.

Launch Chrome and open a tab with target "*chrome://inspect*". This is how it would look like.

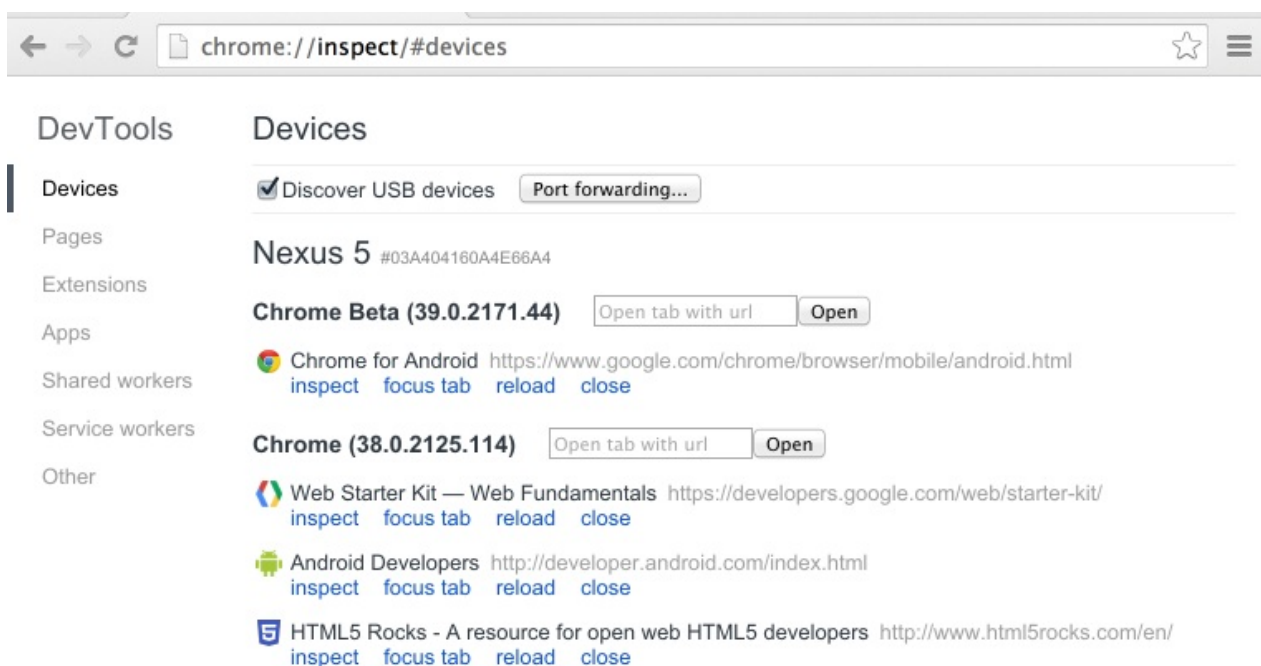


If you notice your phone, you would have got an alert saying "Allow USB Debugging".

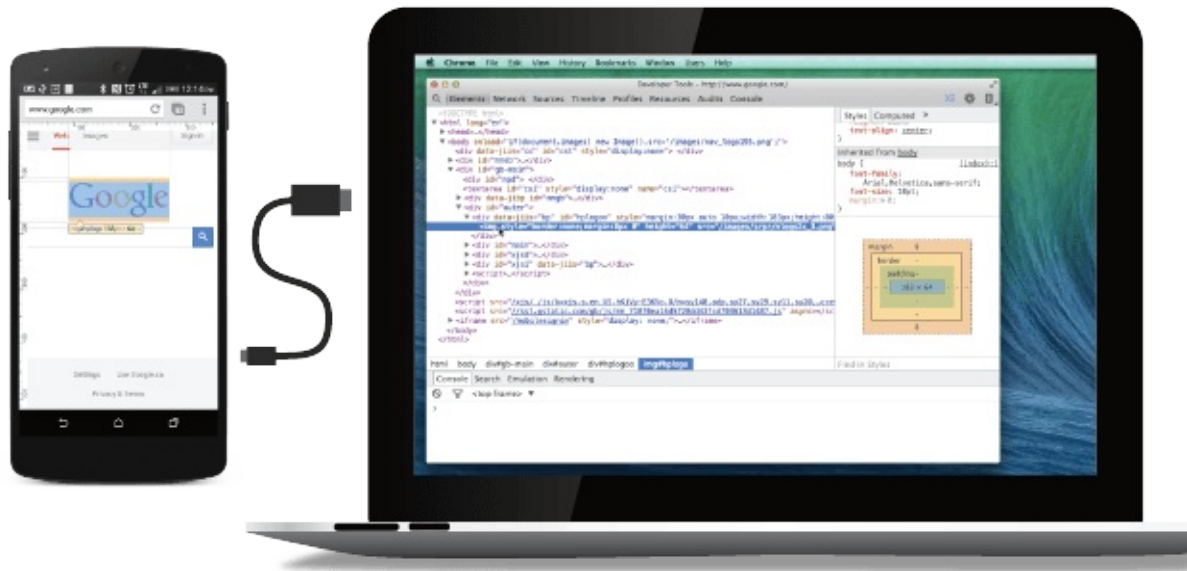




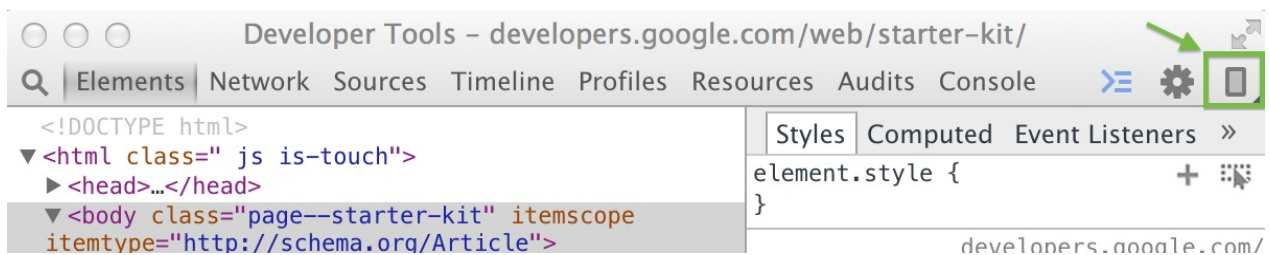
Once you click OK, your device would get showed up in browser tab. Below is how it would show up:



Once you click on **inspect**, you can see the locators as you have always done in web for Selenium.



You can also use Screen casting option by clicking the **ScreenCast** icon in the upper right corner of your remote debugging DevTools window.



***This method of finding locator is faster than using UIAutomatorviewer and would be highly recommended.***

## How to start appium server via Java code

Appium has recently made some changes and now you can actually write a couple line of code and start appium server.

Appium has introduced a class called **AppiumDriverLocalService** which will help you do this. There are two ways you can write code:

1. If you have appium installed via npm then try the below lines of code.

```
service = AppiumDriverLocalService.buildDefaultService();
service.start();
```

For stopping the appium server use:

```
service.stop();
```

2. If you don't have appium installed via npm then try the below lines of code. For your local machine you might want to change the path accordingly. In the below example, I am showing if the OS is mac then use the installed Appium to start the server. If the OS is windows use the npm installed appium. This is just for the demonstration, ideally you should have common strategy whether it is Mac or Windows.

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

```
if (osName.contains("Mac")) {
    service = AppiumDriverLocalService.buildService(new AppiumServiceBuilder()
        .usingDriverExecutable(new File("/Applications/Appium.app/Contents/Re
sources/node/bin/node"))
        .withAppiumJS(new File("/Applications/Appium.app/Contents/Resources/n
ode_modules/appium/bin/appium.js"))
        .withIPAddress("127.0.0.1")
        .usingPort(port)
        .withLogFile(new File("target/"+deviceUnderExecution+".log"))));
}
else if (osName.contains("Windows")) {
    service = AppiumDriverLocalService.buildService(new AppiumServiceBuilder()
        .usingPort(port)
        .withLogFile(new File("target/"+deviceUnderExecution+".log")));
}
else {
    Assert.fail("Unspecified OS found, Appium can't run");
}

System.out.println("- - - - - Starting Appium Server- - - - -");
service.start();
```

In the code above, I have the port as parameter for parallel runs on device.

# How to execute test on real android devices

This chapter would help you understand how to execute the test on real Android Devices, after all the biggest advantage of Appium is that you can run the same test on local devices.

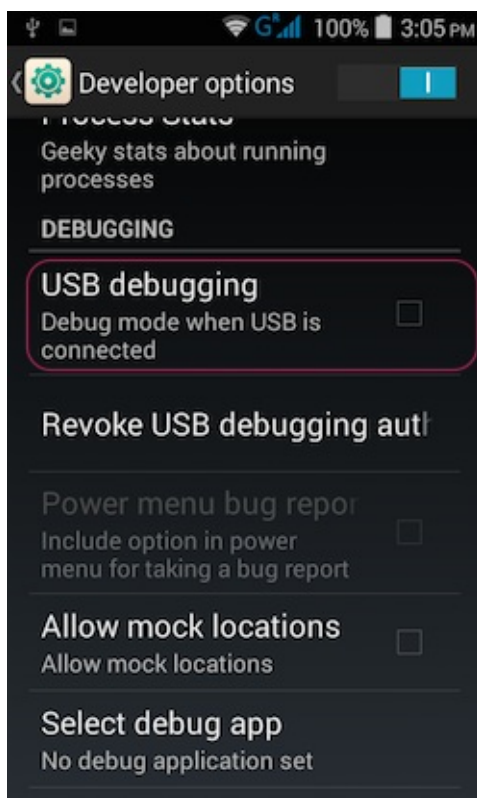
For test to run on devices, we need to make sure :

- **USB Debugging** is enabled
- **ADB** lists your devices into the connected devices
- Changing the **Desired capability** as per the hardware

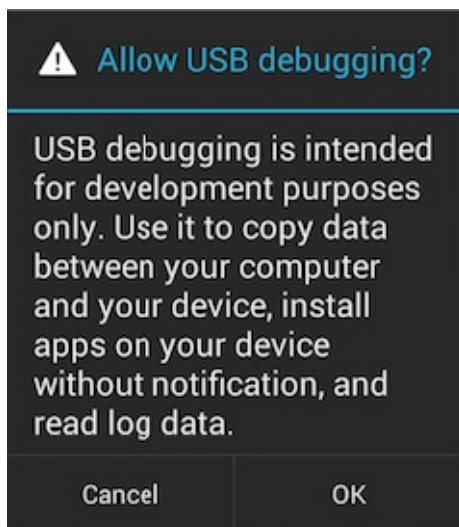
## Enabling USB Debugging:

By default, Android devices do not have USB Debugging enabled, these are under Developer Options. To turn them on,

- Navigate to Settings app on phone
- Scroll down and click on the Developer Options
- Turn on the Developer Options and click the USB Debugging.



On the pop up (shown below), click on Ok.



Some devices do not have “*Developer Options*” and hence the way to enable Debugging mode is to launch the Settings screen. Once done, tap “*About Phone*” and then scroll to the bottom and tap on “*Build Number*” 7 times (Yes 7 times)!

Once done, you will now be able to enable/disable it whenever you desire by going to

```
Settings -> Developer Options -> Debugging -> USB debugging
```

Once the above set ups are done, launch *Terminal* (or Command Prompt) and type in

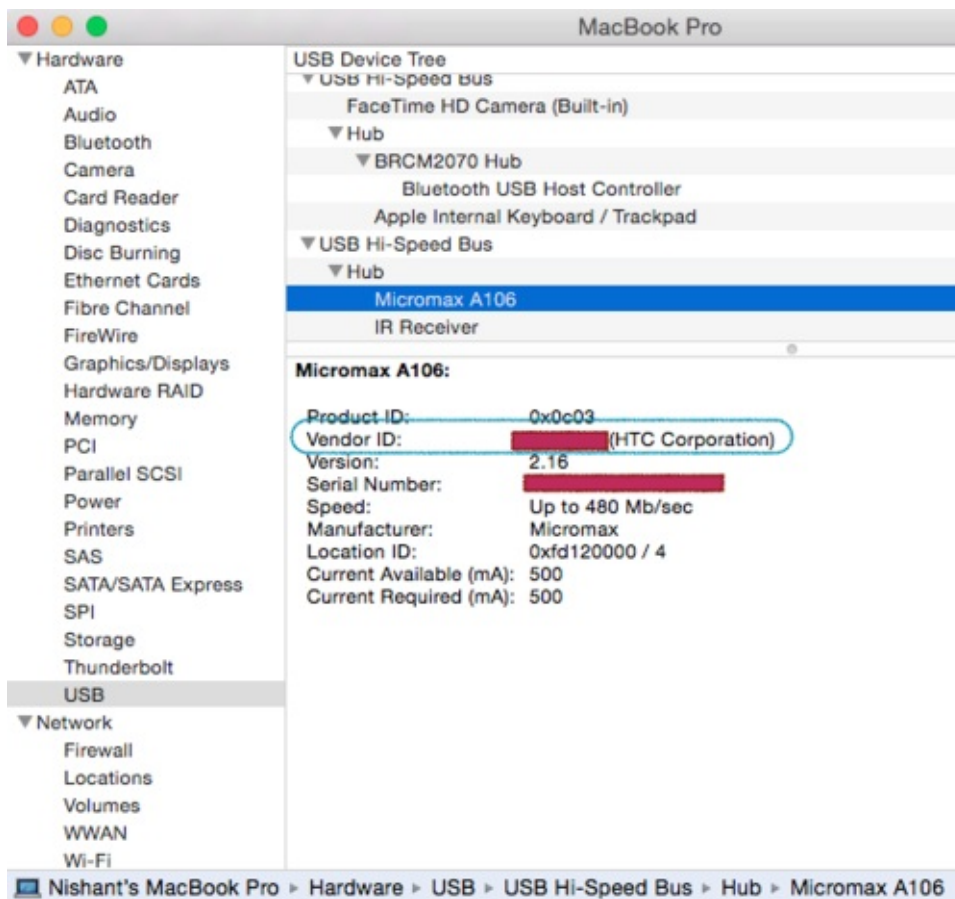
```
adb devices
```

On a happy path, it would show the below result.

```
Nishants-MacBook-Pro:~ nishant$ adb devices
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
List of devices attached
0123456789ABCDEF    device
emulator-5554      device
```

There are times when even after starting USB mode on your device, adb wouldn't list the device here. To fix that, below are some of the steps you need to follow:

- Open the USB manager on your laptop.
- Use the vendor id (highlighted in blue below) and update in the **adb\_usb.ini** file.
- Steps for MAC OS X to open the USB Manager:
  - Click on the Apple icon on top left of the screen
  - Click on “*About This Mac*”
  - On the pop up, click on the System Report
  - Under Hardware section, click on USB
  - You would notice the device connected there, click on the device.

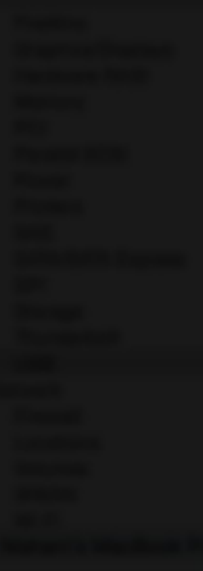


Once you have copied the **vendor ID**, we need to update the file below using the following command:

```
vim ~/.android/adb_usb.ini
```

This is how a sample file would look like after updating the Vendor ID.

```
# ANDROID 3RD PARTY USB VENDOR ID LIST -- DO NOT EDIT.  
# USE 'android update adb' TO GENERATE.  
# 1 USB VENDOR ID PER LINE.  
0x0a01  
0x0b02  
  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~
```



Once the above changes are done, run the following commands:

```
adb kill-server
adb start-server
adb devices
```

This would finally show up the devices as connected. Once the device shows up as online, we are good to run the test.

Start the **Appium** and in that launch the **Android Server**.

In case you have any android version mentioned in your code, you can remove it and have only the one, which are mandatory. Below is a sample snippet I have used and works flawlessly.

```
File appDir = new File("/Users/Steve/Development/SampleApps");
File app = new File(appDir, "Flipkart.apk");
DesiredCapabilities capabilities = new DesiredCapabilities();
capabilities.setCapability("device", "Android");
//mandatory capabilities
capabilities.setCapability("deviceName", "Android");
capabilities.setCapability("platformName", "Android");
capabilities.setCapability("udid", Properties.udid);

//other caps
capabilities.setCapability("app", app.getAbsolutePath());
driver = new RemoteWebDriver(new URL("http://127.0.0.1:4723/wd/hub"), capabilities);
```



# How to run appium test on GenyMotion Emulator

## Introduction

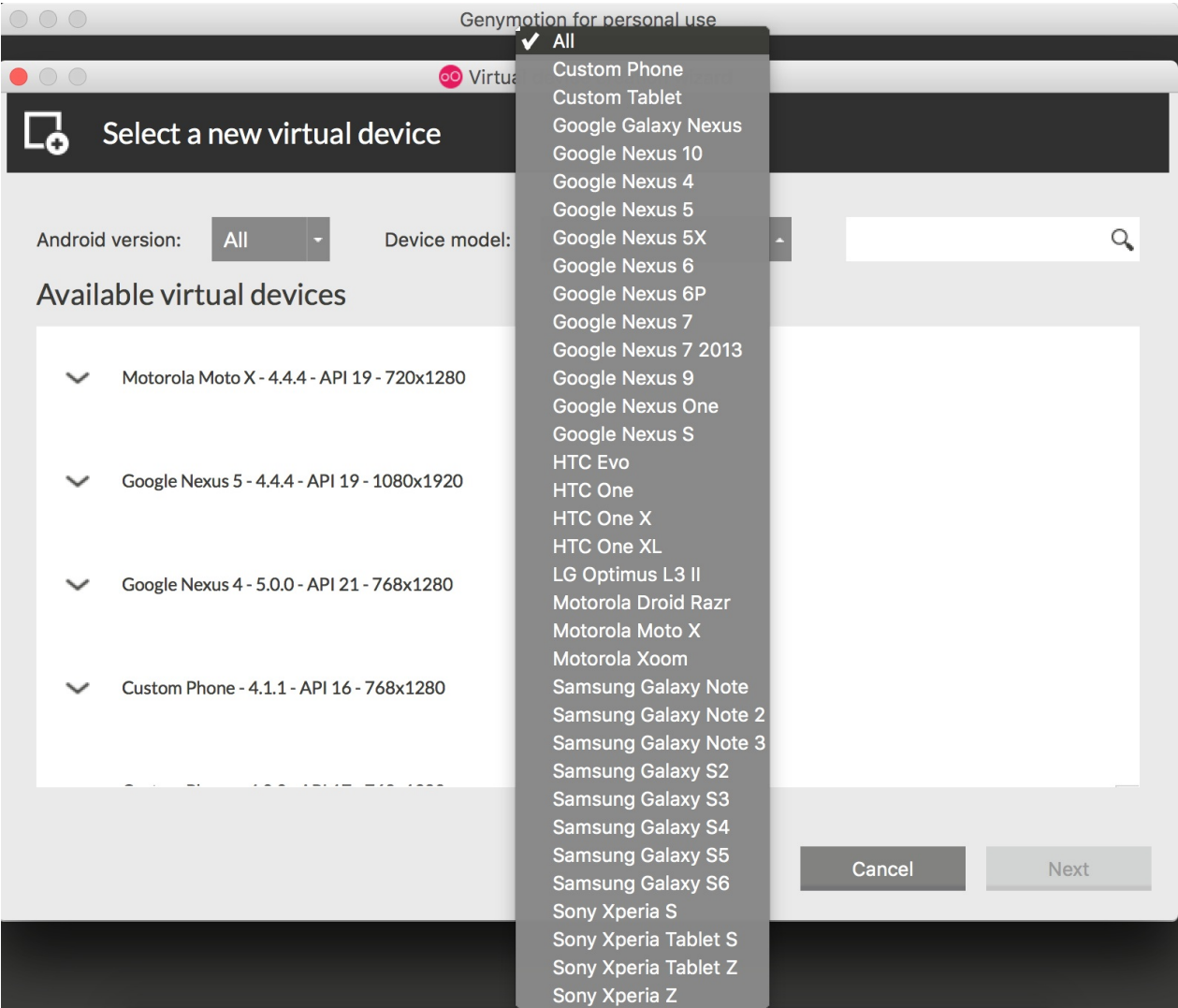
A **Genymotion emulator** is a virtual device which runs on your system and like a typical android emulator. It's an emulator based on **Virtualbox**. It can emulate a bunch of android devices and support a wide variety of API levels. Since the Virtualbox is cross platform compatible, it allows Genymotion to be used on any platform be it windows, Linux, Mac. Genymotion allows you to create custom device image as well as standard device image which is easy to download and get started. It's an emulator using x86 architecture virtualization, making it much more efficient.

Some of the features of Genymotion which stands out are:

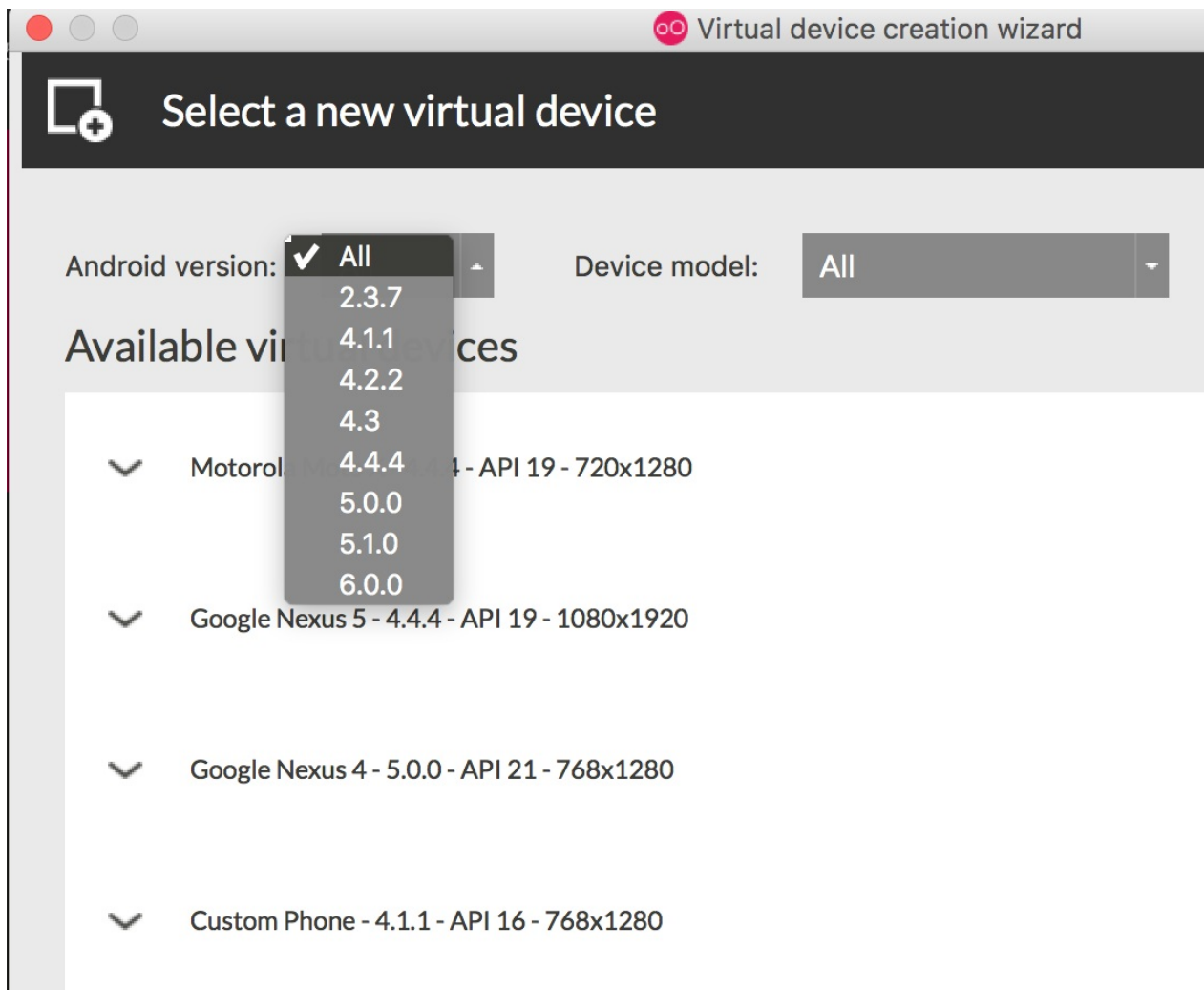
- **Networking**: emulates WiFi connection
- **GPS**: allows coordinate configuration
- **Battery**: allows configuring battery levels
- **Display**: full screen display
- Genymotion shell which allows you to interact with your VM using a command line.
- **Performance**: very fast compared to android emulator

Genymotion has a free version and license one as well. Even in free version you will see a wide range of devices supported and the android version. Download and install the Genymotion app. Once done you should be able to log in with your registered credentials and launch the Genymotion app. This will open up a window shown below and give you option to choose device or android version.

List of Device Image available



List of Android Version supported



Once you download the required device image, if the app depends on Google Play, you need to do 2 things:

- install one apk "**com.android.vending-x.x.xx.apk**"
- Flash it with "**gapps-lp-YYYYMMDD-signed**"

## Running appium test on Genymotion emulator

To execute test on Genymotion emulator, you need to pass the emulator udid to the device config file. When you run "adb devices" command it will show you the udid of the running emulator. It would be typically like "192.168.57.101:5555". You need to pass this value against the udid in the device config file. This is not mandatory if you are running just one instance of emulator and appium server. But if you are attempting parallel run it will make sense to specify udid and bind it with one of the appium server.

## How to automate gestures

**Gestures** play an important role in how your app is being used. With lot of unique gesture supported on mobile devices, automation has its own challenge. Some of the common gestures are:

```
single tap
double tap
flick (left or right)
pull down to refresh
long press
```

Appium handles these gestures using **TouchActions** api they have created. It's more like Actions class in Selenium. Apart from that they have also enabled JSON wire protocol server extensions.

We can pass in coordinates as parameters and specify the action we want. Sample code would look like as:

```
JavascriptExecutor js = (JavascriptExecutor) driver;
HashMap<String, Double> swipeObject = new HashMap<String, Double>();
swipeObject.put("startX", 0.01);
swipeObject.put("startY", 0.5);
swipeObject.put("endX", 0.9);
swipeObject.put("endY", 0.6);
swipeObject.put("duration", 3.0);
js.executeScript("mobile: swipe", swipeObject);
```

When we enter the coordinates in decimal, it actually specifies the percentage. So in above example it means, 1% from x and 50% from y coordinates. *Duration* basically specifies how long it will tap and is in seconds.

Some of the mobile methods to be used are:

```
mobile: tap
mobile: flick
mobile: swipe
mobile: scroll
mobile: shake
```

Prefix "*mobile:*" allows us to route these requests to the appropriate endpoint.

You might also want to explore *TouchActions* class provided by Selenium. It implements actions for touch devices and basically built upon the *Actions* class.

We have added some test in the git project to show how to use the gestures in automation.

The project would help you learn how to automate android app using Appium.

Appium let's you do a scroll to the element text but sometimes it might not work depending on how app is and CSS structure are. You can write your own parallel code to perform scroll and here is the code snippet for the same.

```
JavaScriptExecutor js = (JavaScriptExecutor) driver;  
    HashMap<String, String> scrollObject = new HashMap<String, String>();  
    scrollObject.put("direction", "down");  
    js.executeScript("mobile: scroll", scrollObject);
```

## Tips (With Code Snippets)

**Appium** is pretty evolving and hence there are lot of new things it supports and hence this book needs constant update. Here in this chapter, I intend to share some appium code snippets which is not necessarily explained as a separate chapter.

### \* Changing context while testing an app

Most of the time when you are testing an app, you will find that there is a specific page in app which is a Webview and your normal code is not working. So in those situations we need to change the application context to **"WEBVIEW"** or **"NATIVE"** accordingly. Below is a code snippet will do the same and change the context to Webview.

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

A similar code can be used with Native as parameter to change the context to Native app.

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

### \* Starting Appium Server via code

Generally when you write an automation script you should try to write a code which runs in an un-attended mode. Starting Appium Server and killing it when the test is run is one thing which we should be handling with in the framework. Below code snippet will help you do the same. So if you notice the below code, we are using the installed appium to start the server and then the "port" on which we start is parameterized. Also we are generating an appium log file so as to capture the appium log which is again parameterized based on the device under execution.

```
service = AppiumDriverLocalService.buildService(new AppiumServiceBuilder()  
    .usingDriverExecutable(new File("/Applications/Appium.app/Contents  
/Resources/node/bin/node"))  
    .withAppiumJS(new File("/Applications/Appium.app/Contents/Resource  
s/node_modules/appium/bin/appium.js"))  
    .withIPAddress("127.0.0.1")  
    .usingPort(port)  
    .withLogFile(new File("target/"+deviceUnderExecution+".log")));
```

# Running multiple Appium Server for parallel execution

In the journey of mobile automation it would eventually happen that you will have substantial scenarios automated. Also the desire to have faster feedback would push the need for running **test in parallel**.

So how do you run the test in parallel ? One of the approach I would suggest here is to have multiple appium server running. Steps to do it:

- Connect couple of android devices to the system
- Run the below command and note down the device id.

```
adb devices
```

- Run the below command and replace the **udid** with actual device udid and port with values from (4723)

```
appium -U udid -p port
```

- For second device update the **udid** and increment the port value by 10

Appium gives a way to do the same via code by giving AppiumServiceBuilder class. Below is a sample code which takes input param as **port** and udid. In Appium 1.5.3 release they have move the udid to **GeneralServerFlag** as **Robot\_Address**.

```
def service = AppiumDriverLocalService.buildService(new AppiumServiceBuilder()  
.usingDriverExecutable(new  
File("/Applications/Appium.app/Contents/Resources/node/bin/node")) .withAppiumJS(new  
File("/Applications/Appium.app/Contents/Resources/node_modules/appium/bin/appium.js"))  
.withIPAddress("127.0.0.1") .usingPort(port as int)  
.withArgument(GeneralServerFlag.ROBOT_ADDRESS, udid as String)  
.withArgument(AndroidServerFlag.BOOTSTRAP_PORT_NUMBER, ((port as int) + 2) as String)  
.withArgument(SESSION_OVERRIDE) .withLogFile(new File("build/${device}.log")));
```

So the above command helps you create the appium service which then you can start as

```
service.start
```

The next work would be to create a mapping of tag and the devices on which you want to run the test. This could be done via a properties file.



# References

- Official Site & Tutorials
  - <http://appium.io/>
  - <https://docs.saucelabs.com/tutorials/appium/>
- Android SDK
  - <https://developer.android.com/sdk/index.html?hl=i>
- Github Project Link
  - <https://github.com/appium/appium>
  - <https://github.com/testvagrant/AppiumDemo>
- Resources
  - <http://testvagrant.com/#!/blogs/c152q>
- Appium Discussion Group
  - <https://discuss.appium.io/>
- Selenium
  - <https://code.google.com/p/selenium/wiki/JsonWireProtocol>
  - <http://www.seleniumhq.org/>
- UI Automation Documentation
  - [https://developer.apple.com/library/ios/documentation/DeveloperTools/Reference/UIAutomationRef/\\_index.html](https://developer.apple.com/library/ios/documentation/DeveloperTools/Reference/UIAutomationRef/_index.html)
- UI Testing Android \*[http://developer.android.com/tools/testing/testing\\_ui.html](http://developer.android.com/tools/testing/testing_ui.html)