**Documentation for FlyBye project**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Version** |  |  |  |  |  | **Date Description** |  |  | **Author** |

|  |  |  |  |
| --- | --- | --- | --- |
| 1.0 | 10.05.2019 | Document Produced | Lucian Mihai  Cosmina Bocos  Elena Petricele |

Contents

[I. Project structure 3](#_Toc8656133)

[1. Appium Framework 4](#_Toc8656134)

[2. Mobile Framework 5](#_Toc8656135)

[2.1 api package 6](#_Toc8656136)

[2.2 core package 7](#_Toc8656137)

[II . Automation setup environment 10](#_Toc8656138)

[1. Setup prerequisites for running mobile automation on Android devices 10](#_Toc8656139)

[1.1    Java JDK 11](#_Toc8656140)

[1.2  Android SDK 13](#_Toc8656141)

[1.3  Node JS 14](#_Toc8656142)

[1.4  Appium server 15](#_Toc8656143)

[1.5  Intellij 15](#_Toc8656144)

[2. Setup prerequisites for running mobile automation on iOS devices 16](#_Toc8656145)

[2.1  Xcode 16](#_Toc8656146)

[2.2  Homebrew 16](#_Toc8656147)

[2.3   Install dependencies to make appium-xcuitest-driver to work: 16](#_Toc8656148)

[2.4   Install WebDriverAgent 17](#_Toc8656149)

[II. Preparing Android/iOS devices for automation testing 19](#_Toc8656150)

[1.  Android 20](#_Toc8656151)

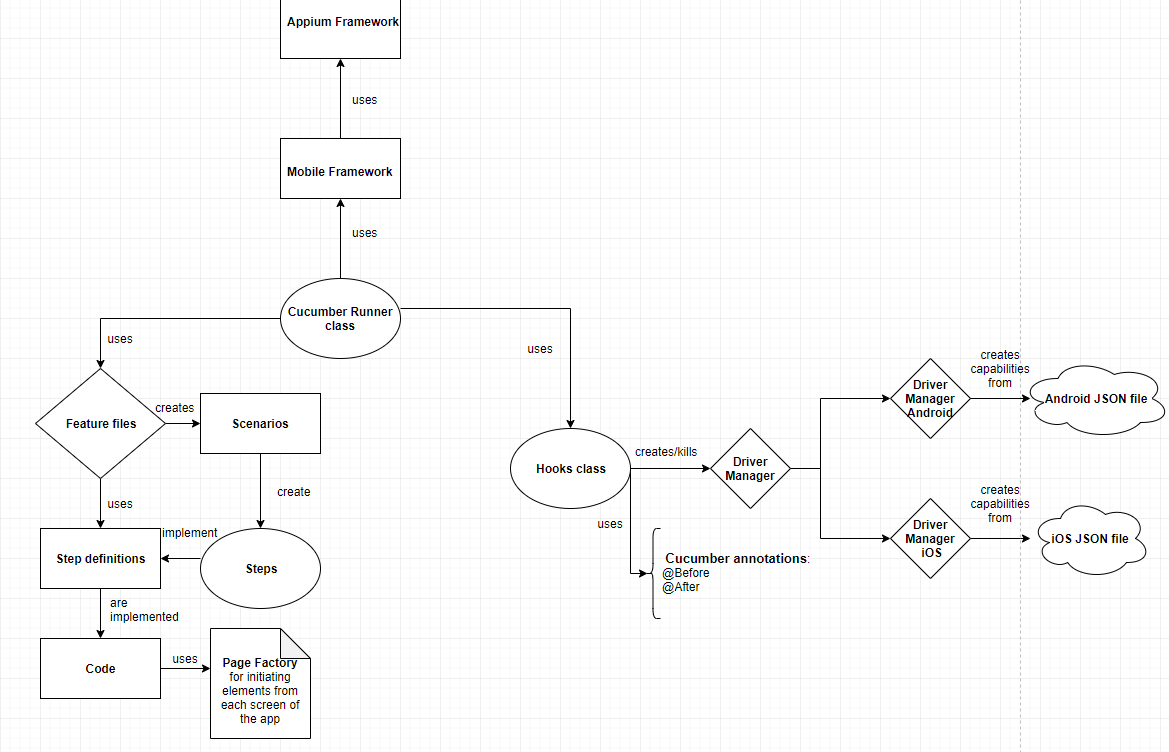
[2. iOS 21](#_Toc8656152)

# Project structure

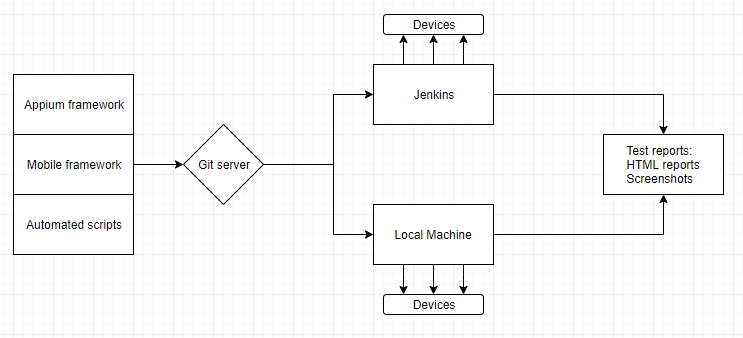
The FlyBye project is a Gradle project written in Java language, using JUnit library for assertions, Cucumber for running the tests and for the reporting part and Appium framework for running automation on mobile (cross platform: Android and iOS).

First we began to write and run automated scripts on our local machines. Further on, we have also integrated Jenkins for parallel run and CI.

The project can be structured under the below generic framework:



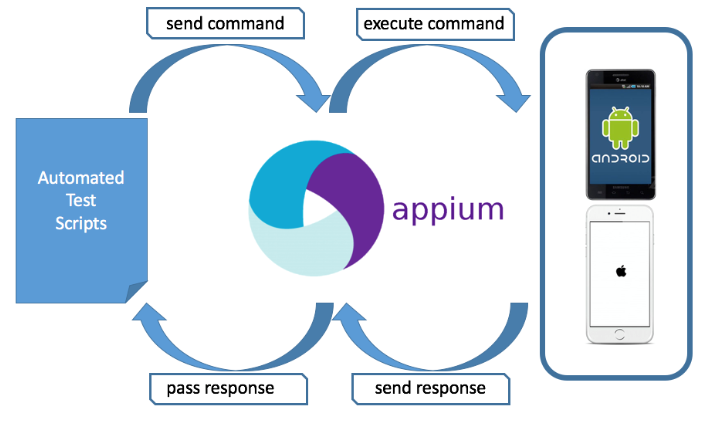
The framework can be used for running tests on local machine (but no parallel run can be executed) or it can be integrated with CI server - Jenkins (where parallel run is possible). Please see below figure:



## Appium Framework

Appium is a test automation framework for use with native, [hybrid](http://appium.io/docs/en/writing-running-appium/web/hybrid/) and mobile web apps.   
It drives iOS, Android and Windows apps using the WebDriver protocol. We have implemented this framework because it offers cross-platform automation that allows you to write tests against multiple platforms (iOS, Android, Windows), using the same API. This enables code reuse between iOS, Android and Windows test suites.

Appium works in a very simple way:

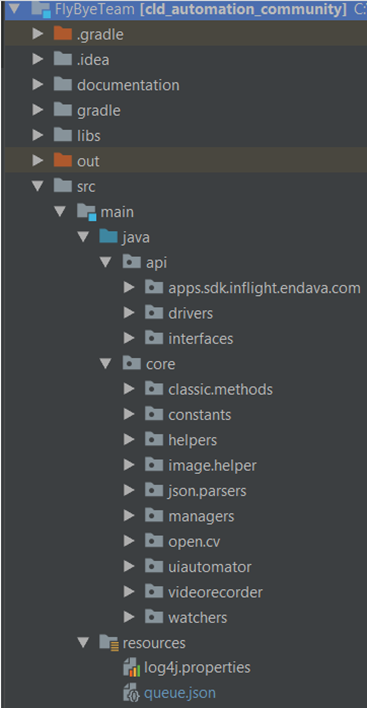
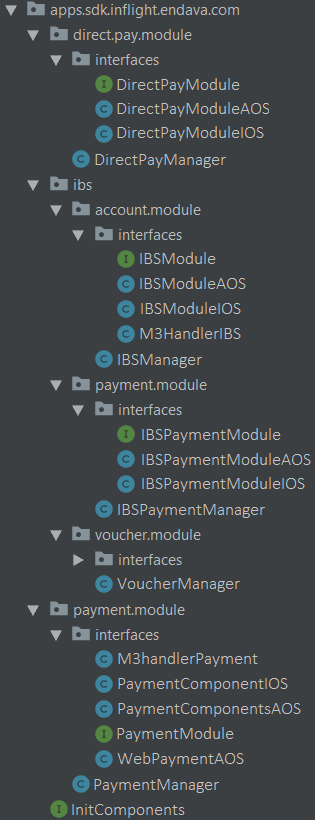
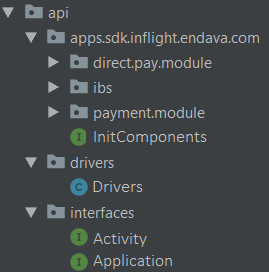


Through the automated test scripts, the desired commands are sent to the Appium server in form of JSON via HTTP, then the server invokes platform specific tools to execute the commands on the devices (real devices or emulators). The devices send back the responses to the Appium server and after that the server logs the results in the IDEA console.

## Mobile Framework

In the java folder from main, there are two main packages:

* **api** – containing packages for the app under testing, drivers and interfaces
* **core** – containing different packages which include helpful methods for interacting with the app



### 2.1 api package

→ **api.apps.sdk.inflight.endava.com** – contains the modules of the app structured under packages and an interface which initiates the modules

* Direct pay
* Ibs
* Payment

For example the **direct.pay.module** contains a package with *Interfaces* and a class for the *Manager*.

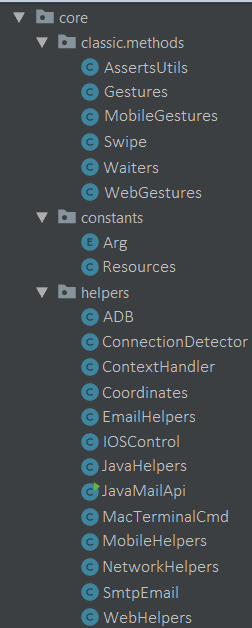
→ ***interfaces*** package:

* Interface **DirectPayModule** – contains methods for:
* fetchTariffList();
* fillUserData();
* purchseWithCard();
* Class **DirectPayModuleAOS** implements the interface, identifies the elements from the UI for both aOS and iOS, overrides methods from the interface, and adds new methods for clicking on the direct pay button and module
* Class **DirectPayModuleIOS** extends the DirectPayModuleAOS class and overrides the methods from the interface
* Class **DirectPayManager** contains methods for returning the directPayModule according to the OS version: getOS(), startModule(), fetchTariffs(), and creditCardFlow()

→ **api.drivers** – contains a single class where are instantiated drivers for Android, iOS, Chrome etc., and methods which return the drivers for mobile and web.

### 2.2 core package

→ **core.classic.methods** – contain classes and overload methods for interacting with the app

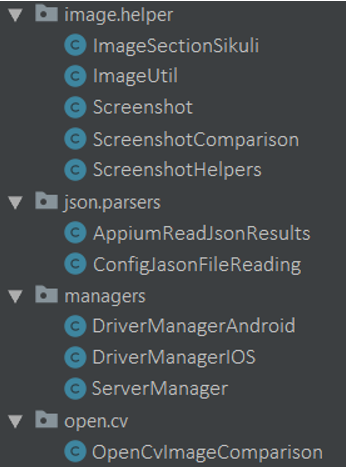


* Assertions: here are included methods which validate if an element is displayed, enabled, visible etc., methods for assertEquals, assertNotEquals etc.
* Gestures: contain methods for finding elements, clicking on web/mobile elements, send text etc.
* MobileGestures: contains methods mobile specific, like swipe, scroll, pinch screen, tap, drag and drop etc.
* Swipe: includes more specific methods for swipe, like: up, down, left, right, two fingers etc.
* Waiters: here are included methods which wait for an element to be visible, or displayed, or enabled, or to contain texts and so on

→ **core.constants** – here are included Enums and a class for Resources, which contains a getQueue() method

→ **core.helpers** – include classes for:

* ADB - adb commands → include methods which return the connected devices, install/uninstall the app, reboot device, push/pull/move/delete files etc.
* MacTerminalCmd - mac terminal commands
* NetworkHelpers - checking the network connectivity
* ContextHandler
* Coordinates - setters and getters for coordinates, which are used for screenshots comparison
* EmailHelpers
* IOSControl
* JavaHelpers – methods for typing random characters, convert string to int etc.
* JavaMailApi – contain methods which retrieve the sender, send date of the email etc.
* MobileHelpers – include methods for installing the app, uninstall, close, launch the app etc.
* SmtpEmail - interactions with the email – put the mail in inbox, remove emails, get email message etc.



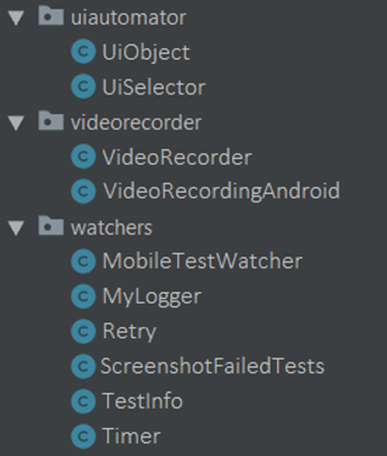
→ **core.image.helper** – contain classes and methods used for working with the images, like taking and saving screenshots, crop images, image comparison etc.

→ **core.json.parsers** – include classes for working with json files, setters and getters

→ **core.managers** – contain classes for Android and iOS driver managers where are methods for creating the capabilities used for communicating with the Appium server, create the server and the driver and method for kill and close the driver

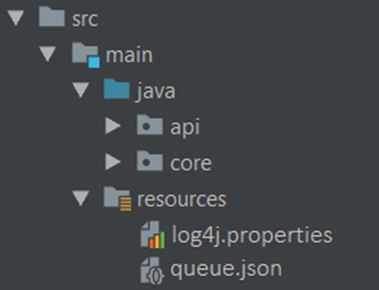
– server manager, which contains methods for getting the OS, validates if the OS is Mac or Windows, etc.

→ **core.open.cv** – include methods similar to the package image.helper, with the difference that they are using different libraries

→ **core.uiautomator** – contain classes with methods which return string values of locators like: resourceId, className, index etc., and methods which validate if web elements are checkable, clickable, focused, scrollable etc.

→ **core.videorecorder** – include classes with methods which record videos, pull video from device, remove videos etc.

→ **core.watchers** – contains classes and methods which keep a log of each passing and failing test, as well as capturing screenshots of failed tests

In the resources folder from main there are two files:

→ log4j.properties – used for logging messages at different levels

→ queue.json

# II . Automation setup environment

Before starting to use the Automation framework, a setup needs to be done. Please take into consideration that running the automation tests on an iOS device is not possible from the Windows machine, since the WebDriverAgent is necessary. WebDriverAgent is a WebDriver server implementation for iOS that can be used to remote control iOS devices. WebDriverAgent needs to be installed and setup to allow Appium to automate iOS devices.

## Setup prerequisites for running mobile automation on Android devices

Software required:

* 1. Java JDK
  2. Android SDK (Android Studio)
  3. Node.js
  4. Appium Server
  5. Intellij

### 1.1    Java JDK

* Go to <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
* Download the suitable version for your machine (ex: for 64-bit )

**On Windows:**

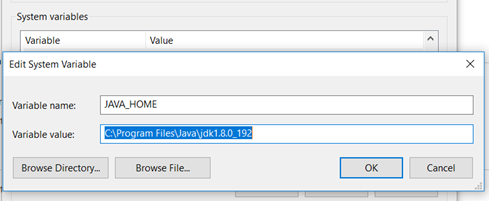
* + Install it in **C:\Java** if possible
  + Add **JAVA** as **System Variable:**

- Right click on This PC and select Properties.

- Click on Advanced system setting

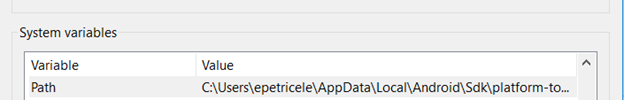
- Click on Environment Variables

- In the system variables create a new system variable named JAVA\_HOME and give the path to your Java folder (ex: C:\Program Files\Java\jdk1.8.0\_192)



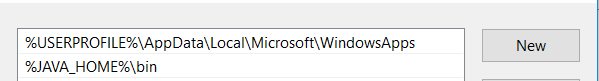
* + Add **JAVA** bin to **Path:**

- Click on Path from the System variables

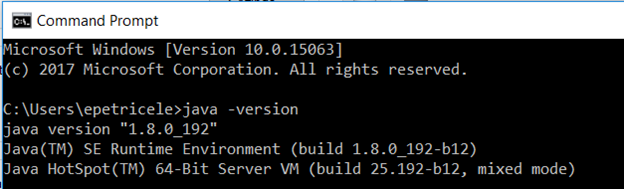


- Select Edit button

- Add a new line with %JAVA\_HOME%\bin



* + Open Command Prompt and type java -version or java -v



**On Mac:**

* + Create a .bash\_profile with JAVA\_HOME:
  + $ cd ~   -> this puts you in the home directory
  + Create the file with the following command $ nano .bash\_profile which opens the file in editing mode
  + Type in the file JAVA\_HOME variable &  edit the PATH variable:

• export JAVA\_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0\_71.jdk/Contents/Home

• export PATH=$JAVA\_HOME:$PATH

• Press CTRL+X. Confirm by typing the Y   key and hit Enter which will save and exit the editor

* Make changes to take effect by running the following command: $ source~/.bash\_profile

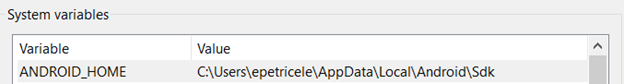
### 1.2  Android SDK

-        Download full Android Studio from: <https://developer.android.com/studio/>, install it and download all mandatory files (will be prompted on starting Android Studio)

-        To manipulate SDK manager from UI, start Android Studio à create a simple project and from IDEE click on SDK Manager button (upper right blue arrow next to emulator icon) – from there you can download latest tools and platform tools

**On Windows:**

-        **Add Android SDK as System Variable:** ANDROID\_HOME (C:\AndroidStudio\SDK) à this is an example; you need to put path to your SDK (by default it is installed in C**:\Users\yourUser\AppData\.....)**



 -        **Add Android tools and platform tools to Path:**

* Click on Path from the System variables
* Select Edit button
* Add 2 new lines as the below example:

C:\ae8bbde23a49d84fddfe3df827d7dd8c

**On Mac:**

-        Add in the .bash\_profile ANDROID\_HOME:

-        $ cd ~   -> this puts you in the home directory

-        Open the file with the following command $ open -e .bash\_profile

-        Type in the file ANDROID \_HOME variable & edit the PATH variable

• export ANDROID\_HOME=/Users/current\_user/Library/Android/sdk

• export PATH=$ANDROID\_HOME/tools:$ANDROID\_HOME/platform-tools:$PATH

-        Press CTRL+X. Confirm by typing the Y   key and hit Enter which will save and exit the editor

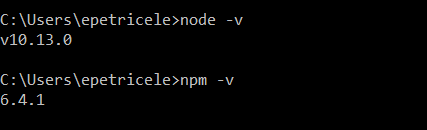
-        Make changes to take effect by running the following command: $ source~/.bash\_profile

### 1.3  Node JS

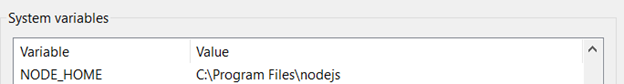
-        Installation of Node and NPM can be found here: <http://blog.teamtreehouse.com/install-node-js-npm-windows> (please verify in cmd that it is installed correctly and it is running)

**On Windows:**

-        Download location: C:\nodejs\node.exe



 -        Add node as System Variable: NODE\_HOME -> C:\nodejs

****

**On Mac:**

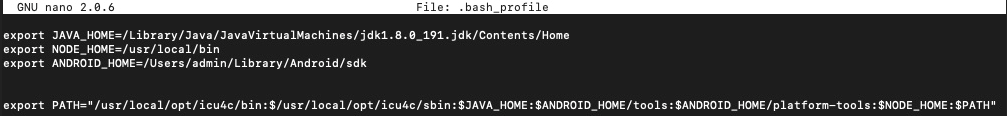
-        Installation of Node and NPM can be found here: <https://blog.teamtreehouse.com/install-node-js-npm-mac>

-        Verify that the installation was succeeded by entering $ npm -v and $ node -v commands in the terminal and it will display the version.

-        Add Node JS in the .bash\_profile as NODE\_HOME:

-        Type in the file NODE \_HOME variable & edit the PATH variable

After adding all the variables, your .bash\_profile should look something like this:



### 1.4  Appium server

-        Before starting, type in cmd/terminal : **npm info Appium** and hit enter

-        If you scroll up you will see Appium server versions

-        Latest version is always the one which is in development (new fixes are added but also might introduce other bugs) – follow Appium on GitHub for more info: <https://github.com/appium/appium> mostly on issues to see latest bugs

-        When you decided which Appium server you want to use open again cmd and type: **npm install -g** [**appium@1.9.0**](mailto:appium@1.9.0)(to download a specific version) or **npm install -g appium@beta** (to download latest beta version) and hit enter.

-        Download location will be by default: **C:\Users\yourUser\AppData\Roaming\npm\node\_modules\appium\build\lib\main.js** on Windows

### 1.5  Intellij

-        Download Intellij community edition: <https://www.jetbrains.com/idea/download>

-        This IDE it is used for writing automated scripts so it is highly recommended to be the one used for running the scripts.

## 2. Setup prerequisites for running mobile automation on iOS devices

Software required:

* 1. XCode
  2. Homebrew
  3. Dependencies to make appium-xcuitest-driver to work:
  4. WebDriverAgent

### 2.1  Xcode

-      Install XCode: <https://developer.apple.com/download/>

### 2.2  Homebrew

-      If you haven’t installed it with Node JS, then you should install it now

-      Go to: <https://brew.sh/> and follow the instructions on how to install it

### 2.3   Install dependencies to make appium-xcuitest-driver to work:

-        Open terminal and type:

* **brew install ideviceinstaller**  (to add libmobiledevice)

-        Libimobiledevice

* **brew uninstall --force libimobiledevice**
* **brew install --HEAD libimobiledevice**

-        Ios-webkit-debug-proxy

* **brew uninstall --force ios-webkit-debug-proxy**
* **brew install ios-webkit-debug-proxy**

-        In order to make WebDriverAgent to work we need to have Carthage:

* **brew install carthage**

-        For iOS 10 ideviceinstaller does not work so we need ios-deploy:

* **npm install –g ios-deploy** - To be able to install applications on your phone like on android with ADB,  
  here on iOS we have ios-deploy commands. Basically you need ios-deploy installed and after that you only need that device is connected via USB to your mac and just pass the commands.

-        For real device use, and to make Xcode user friendly we need to install xcpretty:

* **gem instal xcpretty**

### 2.4   Install WebDriverAgent

-        Now we have all dependencies downloaded and we need to configure appium-xcuitest-driver in order to be able to run test on real iOS devices. Go to terminal and change directory to the one where WebDriverAgent is  installed and run the script to download all necessary frameworks:

    Ex: In terminal type:

* cd /usr/local/lib/node\_modules/appium/node\_modules/appium\_xcuitest\_driver/WebDriverAgent

-        Then run the command:

* ./Scripts/bootstrap.sh

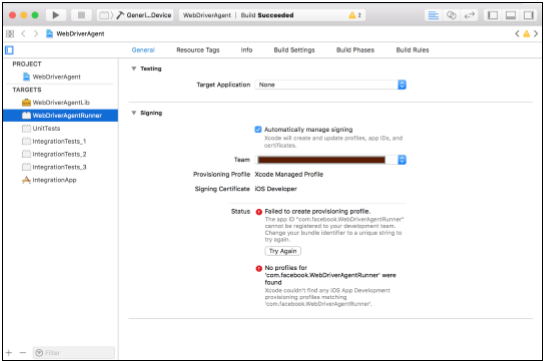
-   Once all dependencies are downloaded from WebDriverAgent folder open WebDriverAgent.xcodeproj (a small project will appear).

-  **BEFORE MAKING ANY CHANGE** be sure that a valid certificate is installed on our Mac Machine (preferably the one which was used to sign the application that you want to automate).

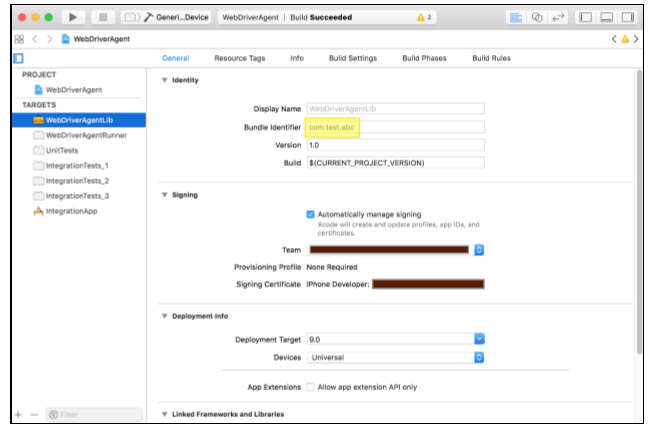
-           On WebDriverAgent Xcode project go to **WebDriverAgentLib** and from **General** tab on **Signing Debug** and **Signing release** select the Team of your certificate from above (hit build and save to save changes and to see that it   is working). After select **WebDriverAgentRunner** and in **General** tab on **Signing debug** and **release** select **Provisioning profile** for your certificate (hit build and save).

**Note:**

-        If an error appears saying that Xcode failed to create 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.



-           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).

 -          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.

 -           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).*

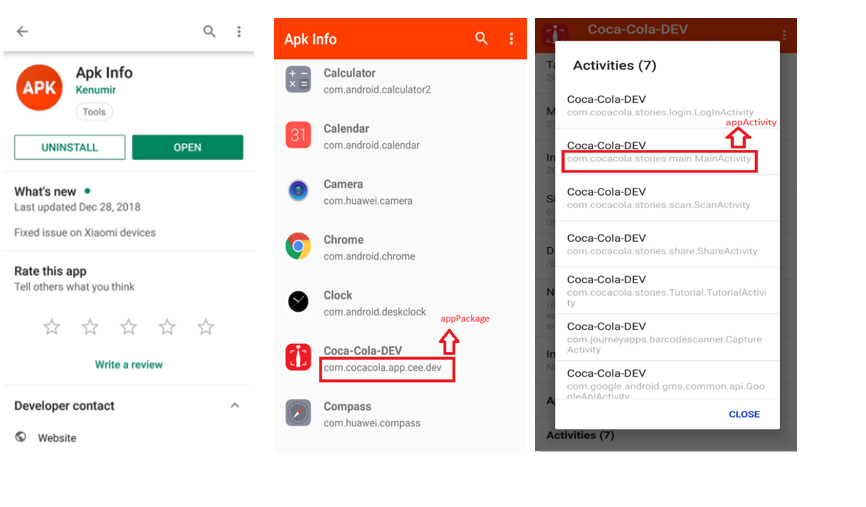
# Preparing Android/iOS devices for automation testing

          After the Automation setup environment is all configured on the machines, some settings must be done also on the devices on which we want to run the automated tests. These configurations are necessary so that the framework knows what app we want to launch, how to launch it, on what device and so on.

         Because the tests are ran on cross-platform, settings must be done both for Android and iOS devices.

## 1.  Android

* + On the device you are trying to run the testes, unable **USB debugging** (follow the steps from the link: <https://www.embarcadero.com/starthere/xe5/mobdevsetup/android/en/enabling_usb_debugging_on_an_android_device.html>)
  + Install the APK app from Google Play which will provide the appPackage and appActivity name of any app which is installed on your mobile device.
  + **Step 1:** Download “APK Info” app from Google Play Store on your android mobile.
  + **Step 2:** Once you have successfully installed APK Info app, open it and check that it lists down all the apps that you have on your phone. Then search for “Coca-Cola”
  + **Step 3:** Check the images below

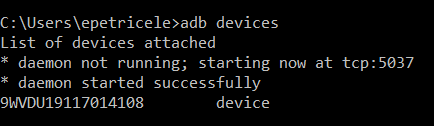


               The appPackage and appActivity will be used in the Json file like in the below example:



Device id must be changed with the UDID that mobile device is using:

* + Run in cmd the following command: adb devices



## iOS

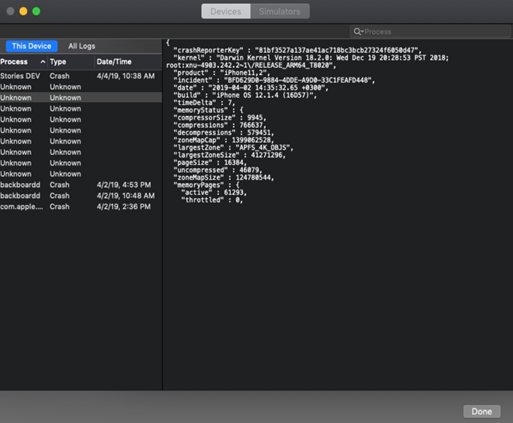
* + On the device you are trying to run the testes, enabling Developer Mode on iPhone:
  + Open the Xcode app on your Mac.
  + You will need to agree to the terms of the software and license agreements when you open Xcode for the first time. This will install software components and finalize the Xcode installation process
  + Plug your iPhone into your Mac
  + Open the Settings app on your iPhone
  + Scroll down and tap Developer
  + Tick the Enable UI Automation label

Seeing this option in your Settings means you have enabled developer mode on your iPhone. You can now start demoing apps, checking logs and playing with other developer settings on your device.

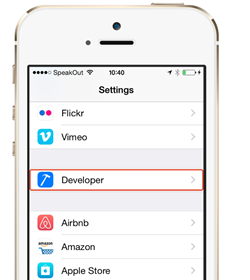
On your mac – > open Xcode –>   from top bar select Window –> select Devices and Simulators and something like this will appear:



If you click on View Device Logs, logger will appear. Here you can see real time logging of your device and applications.



After you finished with installing Xcode, verifying that Xcode recognized your iPhone and logging works, go to your phone OS settings and verify that developer options appeared. It should look like this:



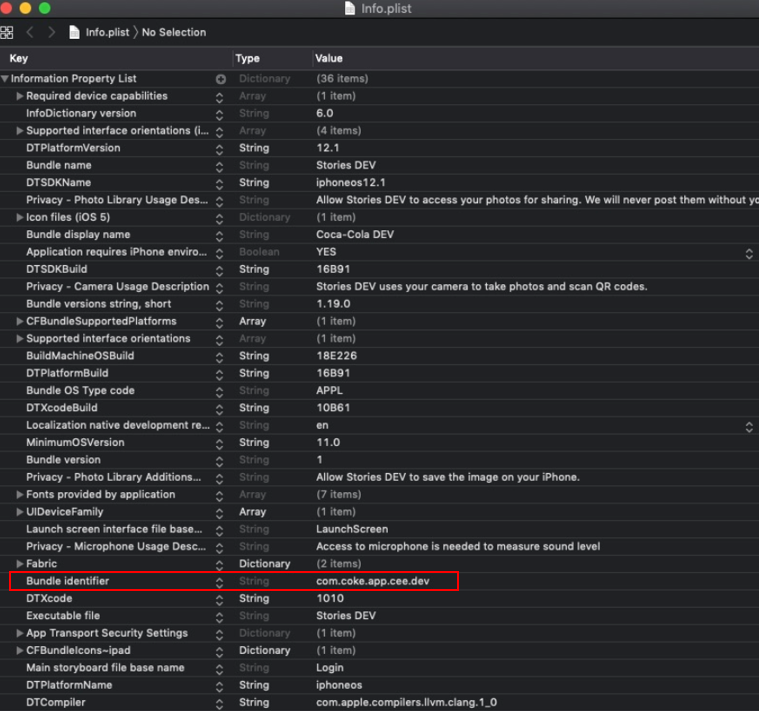
To be able to install applications on your phone, on iOS  ios-deploy commands should be used.

Basically you need ios-deploy installed and after that you only need that device to be connected via USB to your mac and just pass the commands. Command to install ios-deploy:npm install -g ios-deploy

An example of how to install an ipa file to your connected phone:

* + In terminal type Ios-deploy –b path\_to\_your\_application/application.ipa  (and now your app will be installed on your phone).

On iOS devices, the apps can be identified by the bundle identifier. In order to find the bundle id of the app, click right on the ipa file (from mac)→ Open With → Archive Utility → open the Payload folder → right click on the app → Show Package Contents → Info.plist:

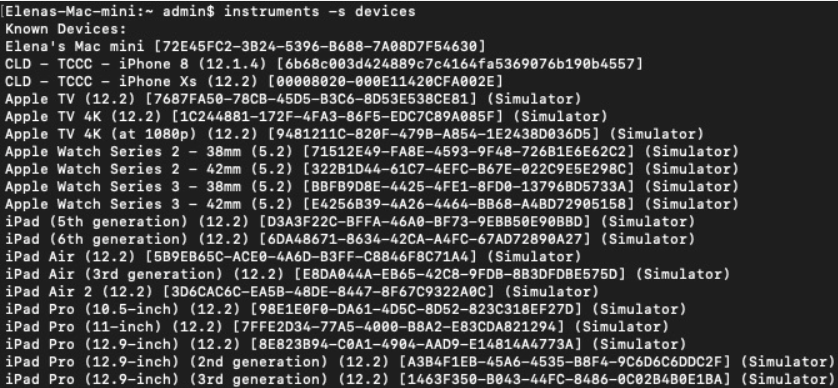


The bundle id must be used in the Json file like in the below example, so that the app can be recognized and launched on the device on which the tests are ran.



Device id can be found by writing the following command in the terminal:

* + instruments -s devices (the Simulators are also displayed in the list)



**OBS:**

In Json files it is very important to provide the location where the node.exe was installed and the appium server. The paths should be changed according to the path they were installed on your machine:

* + 0945a22383a7f563d28786a99aba868f- for Windows
  + c295931f5573c0093ac2af3e1f225d8c- for Mac

Furthermore, appiumServerPort, systemPort and wdaLocalPort should be changed for every device that you are running the testes (if several tests are run in parallel) since sometimes there can be a port conflict if different ports aren't used.

f83b7337b05a0dba667a054111e0df52

d0abf12f13ee2d9875081e7e13689526

Another important aspect here is that the userMail, userPassword, name and prefixEmail should be changed with the credentials for the email address that you are running the tests.