

**VISVESVARAYA TECHNOLOGICAL UNIVERSITY  
JNANASANGAMA, BELGAVI-590018**



**SOFTWARE TESTING REPORT ON**  
**“GOOGLE EARTH AUTOMATION TESTING USING APPIUM TOOL”**

Submitted in partial fulfillment of the requirements for the 6<sup>th</sup> Semester  
**INFORMATION SCIENCE AND ENGINEERING DEPARTMENT**

**Submitted by**

**JAYA KUMAR**

**1BI18IS012**

**ROHAN RN**

**1BI18IS039**

**SATHVIK DN**

**1BI18IS046**

**Under the guidance of**

**Mrs. VEDASHREE T K**

Assistant Professor

Dept of ISE, BIT, Bangalore-04



**2021**

**DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING  
BANGALORE INSTITUTE OF TECHNOLOGY  
K. R. Road, V. V. Puram, Bengaluru-560004**

# TABLE OF CONTENTS

CHAPTER	TITLE	PAGE NO.
1	<b>INTRODUCTION</b>	<b>1</b>
	<b>1.1. Testing Levels</b>	<b>1</b>
	1.1.1. Unit Testing	2
	1.1.2. Integration Testing	2
	1.1.3. System Testing	2
	1.1.4. Acceptance Testing	2
	<b>1.2. Testing Types</b>	
	1.2.1. Member details	3
	1.2.2. Smoke and Sanity Testing	3
	1.2.3. Regression Testing	3
	1.2.4. Alpha Testing	4
	1.2.5. Beta Testing	4
	<b>1.3. TESTING STRATERGIES</b>	<b>5</b>
	1.3.1. STATIC TESTING	5
	1.3.2. DYNAMIC TESTING	5
2	<b>APPIUM TOOL</b>	<b>6</b>
	<b>2.1. About Appium</b>	<b>6</b>
	<b>2.2. Prerequisites to use Appium</b>	<b>7</b>
	<b>2.3. Appium Architecture</b>	<b>7</b>
	2.3.1. Workflow of Architecture	8
	<b>2.4. Components Used</b>	<b>9</b>
	2.4.1. Appium Python Client	9
	2.4.2. Appium Server	9
	2.4.3. Selenium J Standalone	10
	2.4.4. ABD Emulator	10
	<b>2.5. Advantages of Appium</b>	<b>10</b>
3	<b>IMPLEMENTATION</b>	<b>11</b>
	<b>3.1. Tool Setup</b>	<b>11</b>
	<b>3.2. Working on Android</b>	<b>16</b>
	<b>3.3. Program Code</b>	<b>17</b>

<b>4</b>	<b>TESTING</b>	<b>22</b>
	<b>4.1. Test Cases</b>	<b>22</b>
	<b>4.2. Test Report</b>	<b>23</b>
<b>5</b>	<b>SNAPSHOTS</b>	<b>24</b>
<b>6</b>	<b>CONCLUSION</b>	<b>29</b>
<b>7</b>	<b>REFERENCES</b>	<b>30</b>

---

---

## CHAPTER 1 INTRODUCTION TO SOFTWARE TESTING

Software testing involves the execution of a software component or system component to evaluate one or more properties of interest. In general, these properties indicate the extent to which the component or system under test:

- ✦ Meets the requirements that guided its design and development,
- ✦ Responds correctly to all kinds of inputs,
- ✦ Performs its functions within an acceptable time,
- ✦ It is sufficiently usable,
- ✦ It can be installed and run in its intended environment, ✦ Achieves the general result its stakeholders desire.

As the number of possible tests for even simple software components is practically infinite, all software testing uses some strategy to select tests that are feasible for the available time and resources. As a result, software testing typically (but not exclusively) attempts to execute a program or application with the intent of finding failures due to software faults. Software testing can provide objective, independent information about the quality of software and risk of its failure to users or sponsors.

Software developers can't test everything, but they can use combinatorial test design to identify the minimum number of tests needed to get the coverage they want. Combinatorial test design enables users to get greater test coverage with fewer tests. Whether they are looking for speed or test depth, they can use combinatorial test design methods to build structured variation into their test cases.

### 1.1 TESTING LEVELS

Broadly speaking, there are at least three levels of testing: unit testing, integration testing, and system testing. However, a fourth level, acceptance testing, may be included by developers. This may be in the form of operational acceptance testing or be simple end-user (beta) testing, testing to ensure the software meets functional expectations. Based on the ISTQB Certified Test Foundation Level syllabus, test levels includes those four levels, and the fourth level is named acceptance testing. Tests are frequently grouped into one of these levels by where they are added in the software development process, or by the level of specificity of the test.

### **1.1.1 Unit testing**

Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors.

Unit testing is a software development process that involves a synchronized application of a broad spectrum of defect prevention and detection strategies in order to reduce software development risks, time, and costs. It is performed by the software developer or engineer during the construction phase of the software development life cycle. Unit testing aims to eliminate construction errors before code is promoted to additional testing; this strategy is intended to increase the quality of the resulting software as well as the efficiency of the overall development process.

### **1.1.2 Integration testing**

Integration testing is any type of software testing that seeks to verify the interfaces between components against a software design. Software components may be integrated in an iterative way or all together ("big bang"). Normally the former is considered a better practice since it allows interface issues to be located more quickly and fixed.

Integration tests usually involve a lot of code, and produce traces that are larger than those produced by unit tests. This has an impact on the ease of localizing the fault when an integration test fails. To overcome this issue, it has been proposed to automatically cut the large tests in smaller pieces to improve fault localization.

### **1.1.3 System testing**

System testing tests a completely integrated system to verify that the system meets its requirements. For example, a system test might involve testing a login interface, then creating and editing an entry, plus sending or printing results, followed by summary processing or deletion (or archiving) of entries, then logoff.

### **1.1.4 Acceptance testing**

Commonly this level of Acceptance testing include the following four types:

- ✦ User acceptance testing
- ✦ Operational acceptance testing
- ✦ Contractual and regulatory acceptance testing
- ✦ Alpha and beta testing

## **1.2 TESTING TYPES:**

There are many types of testing that are based on various techniques and tactics.

### **1.2.1 Compatibility testing**

A common cause of software failure (real or perceived) is a lack of its compatibility with other application software, operating systems (or operating system versions, old or new), or target environments that differ greatly from the original (such as a terminal or GUI application intended to be run on the desktop now being required to become a Web application, which must render in a Web browser). For example, in the case of a lack of backward compatibility, this can occur because the programmers develop and test software only on the latest version of the target environment, which not all users may be running. This results in the unintended consequence that the latest work may not function on earlier versions of the target environment, or on older hardware that earlier versions of the target environment were capable of using. Sometimes such issues can be fixed by proactively abstracting operating system functionality into a separate program module or library.

### **1.2.2 Smoke and sanity testing**

Sanity testing determines whether it is reasonable to proceed with further testing. Smoke testing consists of minimal attempts to operate the software, designed to determine whether there are any basic problems that will prevent it from working at all. Such tests can be used as build verification test.

### **1.2.3 Regression testing**

Regression testing focuses on finding defects after a major code change has occurred. Specifically, it seeks to uncover software regressions, as degraded or lost features, including

old bugs that have come back. Such regressions occur whenever software functionality that was previously working correctly, stops working as intended. Typically, regressions occur as an unintended consequence of program changes, when the newly developed part of the software collides with the previously existing code. Regression testing is typically the largest test effort in commercial software development, due to checking numerous details in prior software features, and even new software can be developed while using some old test cases to test parts of the new design to ensure prior functionality is still supported.

Common methods of regression testing include re-running previous sets of test cases and checking whether previously fixed faults have re-emerged. The depth of testing depends on the phase in the release process and the risk of the added features.

### **1.2.4 Alpha testing**

Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing before the software goes to beta testing.

### **1.2.5 Beta testing**

Beta testing comes after alpha testing and can be considered a form of external user acceptance testing. Versions of the software, known as beta versions, are released to a limited audience outside of the programming team known as beta testers. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Beta versions can be made available to the open public to increase the feedback field to a maximal number of future users and to deliver value earlier, for an extended or even indefinite period of time (perpetual beta).



## **1.4 TESTING STRATEGIES**

The purpose of a test strategy is to provide a rational deduction from organizational, highlevel objectives to actual test activities to meet those objectives from a quality assurance perspective.

### **1.4.1 Static Testing**

It is also known as Verification in Software Testing. Verification is a static method of checking documents and files. Verification is the process, to ensure that whether we are building the product right i.e., to verify the requirements which we have and to verify whether we are developing the product accordingly or not. Activities involved here are Inspections, Reviews, Walkthroughs.

### **1.4.2 Dynamic Testing**

It is also known as Validation in Software Testing. Validation is a dynamic process of testing the real product. Validation is the process, whether we are building the right product i.e., to validate the product which we have developed is right or not. Activities involved in this is Testing the software application (Desktop application, Web application, Mobile Apps)

## **1.5 TESTING ARTIFACTS**

Test Artifacts are the deliverables that are given to the stakeholders of a software project. A software project which follows SDLC undergoes the different phases before delivering to the customer. In this process, there will be some deliverables in every phase. Some of the deliverables are provided before the testing phase commences and some are provided during the testing phase and the rest after the testing phase is completed. These documents are also known as Test Deliverables

- Test plan
- Test case
- Traceability matrix
- Test script
- Test suite
- Release Note
- Test data or Test Fixture

## CHAPTER 2

### APPIUM TOOL

#### 2.1 ABOUT APPIUM

Appium is an open-source automation mobile testing tool, which is used to test the application. It is developed and supported by Sauce Labs to automate native and hybrid mobile apps. It is a cross-platform mobile automation tool, which means that it allows the same test to be run on multiple platforms. Multiple devices can be easily tested by Appium in parallel.

In today's development area, the demand for mobile applications is high. Currently, people are converting their websites into mobile apps. Therefore, it is very important to know about mobile software automation testing technology and also stay connected with new technology. Appium is a mobile application testing tool that is currently trending in Mobile Automation Testing Technology.

Appium is used for automated testing of native, hybrid, and web applications. It supports automation test on the simulators (iOS) and emulators (Android) as well as physical devices (Android and iOS both). Previously, this tool mainly focused on IOS and Android applications that were limited to mobile application testing only. Few updates back, Appium declared that it would now support desktop application testing for windows as well.

Appium is very much similar to the Selenium Webdriver testing tool. So, if you already know Selenium Webdriver, Appium becomes very easy to learn. Appium has NO dependency on mobile device OS because it has a framework that converts the Selenium Webdriver commands to UIAutomator and UIAutomation commands for Android and iOS respectively, that depends on the device type rather than the OS type.

It supports several languages such as Java, PHP, Objective C, C#, Python, JavaScript with node.js, and Ruby, and many more that have Selenium client libraries. Selenium is the backend of Appium that provides control over the functionality of Selenium for testing needs.

## 2.2 PREREQUISITES TO USE APPIUM

- The latest version of Java, needed for Android Studio.
- Installation of Android Studio with SDK. This is needed since adb gets installed as part of the Android SDK. The adb utility is required to get the device list connected to the PC.
- Appium for Mac/Windows -1.21.0(+)
- Installation of the latest version of PyCharm.
- Connecting the Mobile device to the PC using USB cable and enabling the developer mode/USB debugging in the Android device.

## 2.3 APPIUM ARCHITECTURE

This is an HTTP server written in Node.js programming language that handles WebDriver sessions. The Appium server receives HTTP requests from the client libraries in JSON format. The requests are then handled in different ways, depending on the platform on which it is running on.

It follows the Client-Server Architecture. There are 3 main components included in it:

1. Appium Client
2. Appium Server
3. End device

### 1) Appium Client

The automation scripted code is what we call as Appium Client.

The code is scripted in any programming language like PHP, Java, Python, etc. This automation script holds the configuration details of the Mobile device and the application. Along with that, the logic/code to run the test cases of the application are scripted.

## 2) Appium Server

Appium server is written using Node.js programming language. It receives connection and command requests from the Appium client in JSON format and executes that command on mobile devices. The Server is necessary to be installed in the machine and is started before invoking the automation code.

The server interacts with various platforms such as iOS and Android. It creates a session to interact with end devices of mobile apps. It is an HTTP server written in Node.js programming language which reads the HTTP requests from the client libraries and sends these requests to the appropriate platform.

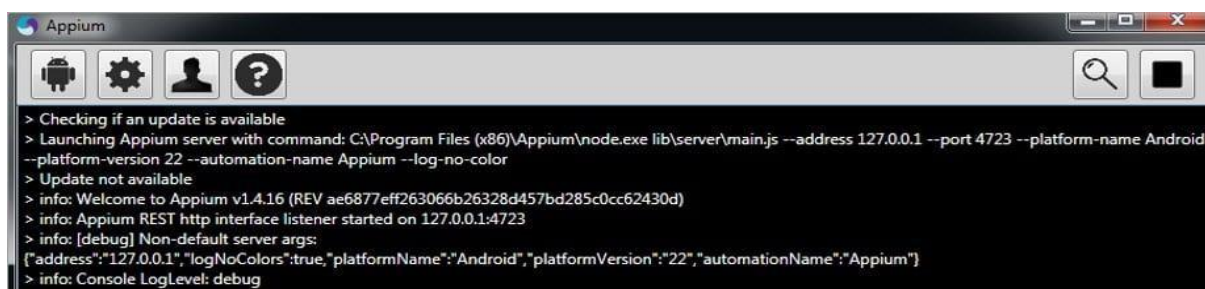


Fig.2.2. Running Appium Server

## 3) End Device

This is mostly a real-time mobile device or an emulator. The automation scripts are executed in the end device by the Appium server by the commands from the client.

### 2.3.1 Workflow of Appium Architecture

The Appium Client which holds the configuration details and the automation script for the test cases sends the commands to the server in JSON format. The automation script is converted into JSON format by in-built jar files in the client.

Appium Server then recognizes the command and establishes a connection with the corresponding end device. Once the connection is made, it triggers the execution of test cases in the end device. The End Device responds to the request in the form of HTTP to the Appium.

As and when the test cases are executed in the Mobile device/emulator, it populates the log of all the actions performed in the device/emulator.

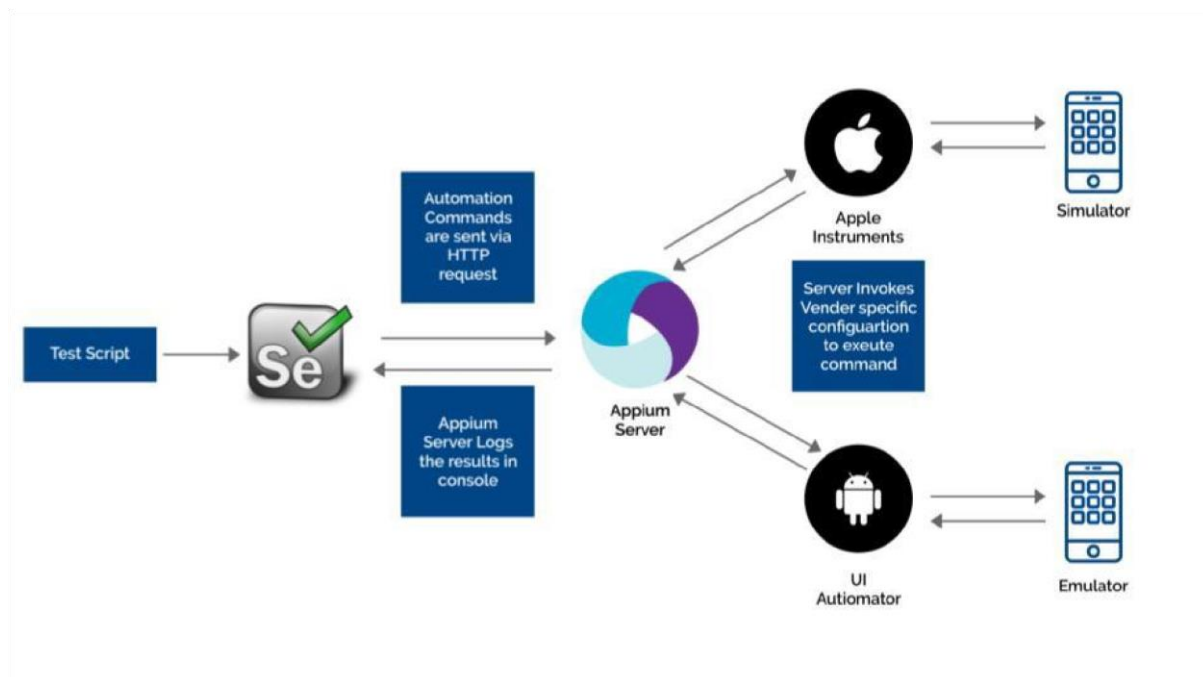


Fig.2.3. Workflow Architecture of Appium

## 2.4 COMPONENTS USED

### 2.4.1 Appium Python Client

The Appium Python Client is fully compliant with the Selenium 3.0 specification draft, with some helpers to make mobile testing in Python easier. The majority of the usage remains as it has been for Selenium 2 (WebDriver), and as the official Selenium Python bindings begins to implement the new specification that implementation will be used underneath, so test code can be written that is utilizable with both bindings.

To use the new functionality now, and to use the superset of functions, instead of including the Selenium webdriver module in your test code, use that from Appium instead.

### 2.4.2 Appium Server

Appium Server is the core component of the Appium architecture. It is written in Node.js and runs on the machine or in the cloud. The Appium Server receives requests from Appium client libraries via the JSON Wire Protocol and invokes the mobile driver (Android driver/iOS driver) to connect to the corresponding native testing automation frameworks to run client operations on the devices. The test results are then received and sent to the clients. The Appium Server, has the capacity to create multiple sessions to simultaneously run tests on multiple devices.

### 2.4.3 Selenium J Standalone

Server Selenium Standalone server is a java jar file used to start the Selenium server. It is a smart proxy server that allows Selenium tests to route commands to remote web browser instances. The aim is to provide an easy way to run tests in parallel on multiple machines. Selenium Jar is a group of API's rolled into one jar for different languages (Java, Python, C#, Javascript, etc.). The client jar can be acquired by tools like Maven or Gradle, basically opensource build automation systems. In case, the dependencies of the jar files are not available, one cannot perform testing in a specific programming language.

### 2.4.4 ABD Emulator

Android Debug Bridge (adb) is a versatile command-line tool that lets you communicate with a device. The adb command facilitates a variety of device actions, such as installing and debugging apps, and it provides access to a Unix shell that you can use to run a variety of commands on a device. adb is included in the Android SDK Platform-Tools package. It is a client-server program that includes three components: ➤ A client, which sends commands. The client runs on your development machine. You can invoke a client from a command-line terminal by issuing an adb command. ➤ A daemon (adbd), which runs commands on a device. The daemon runs as a background process on each device. ➤ A server, which manages communication between the client and the daemon. The server runs as a background process on your development machine.

## 2.5 ADVANTAGES OF APPIUM

- It is free and open-source
- These provide cross-platform solutions for native and hybrid apps
- It is compatible with JSON web-driver and Grid
- Testing based on cloud is supporting using testdriod
- Programming languages such a C#, Java, PHP, Python, Ruby are supported by appium
- App Automation is possible by using appium
- It enables to evaluate of cross-platform mobile apps without recompiling the code.
- It supports simulators, emulators and real devices simultaneously
- The testers can use the inspector for playback and record tool
- Supports JSON wire protocol

## CHAPTER 3

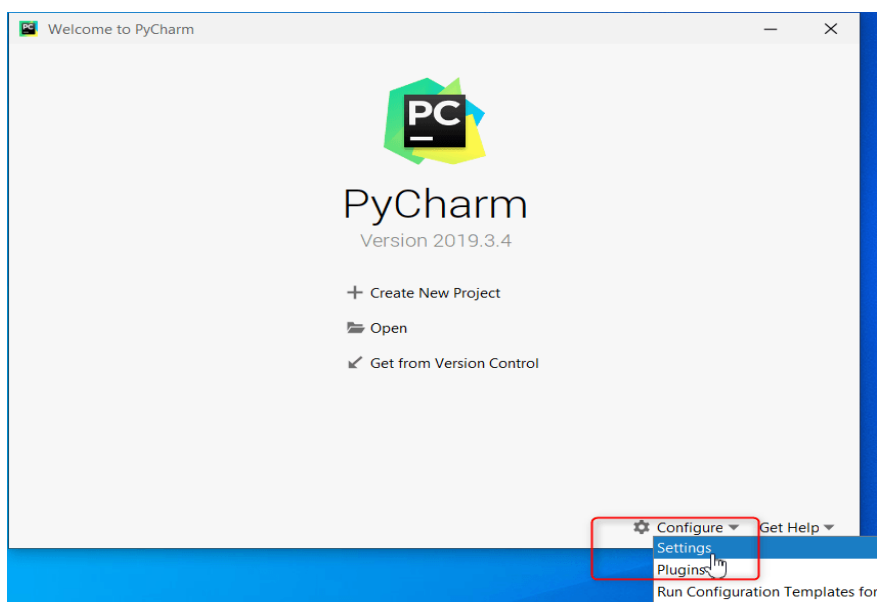
### IMPLEMENTATION

#### 3.1 TOOL SETUP

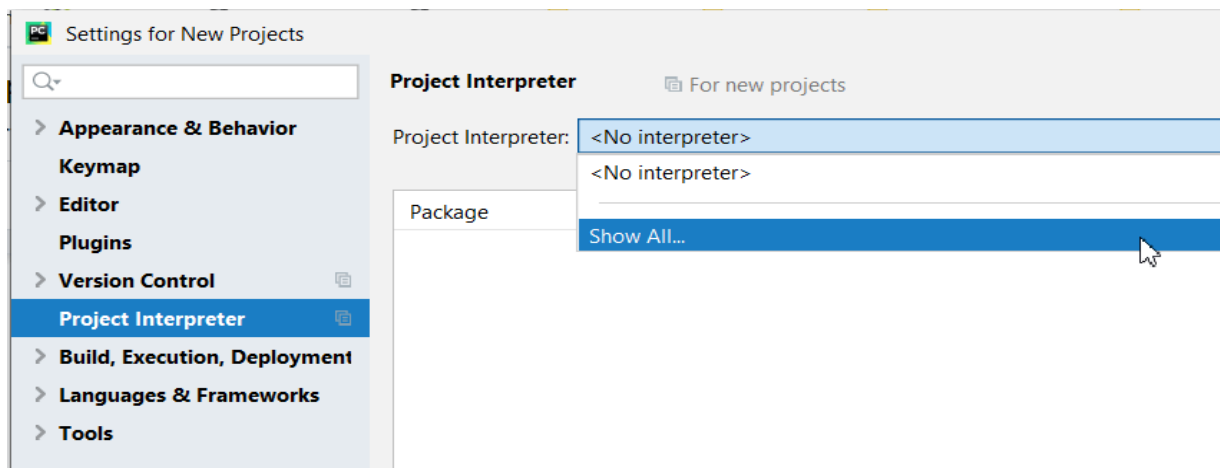
We need to configure the Python interpreter in the Pycharm, so that any project inherits the interpreter by default.

##### 2.6.1 Pycharm IDE setup

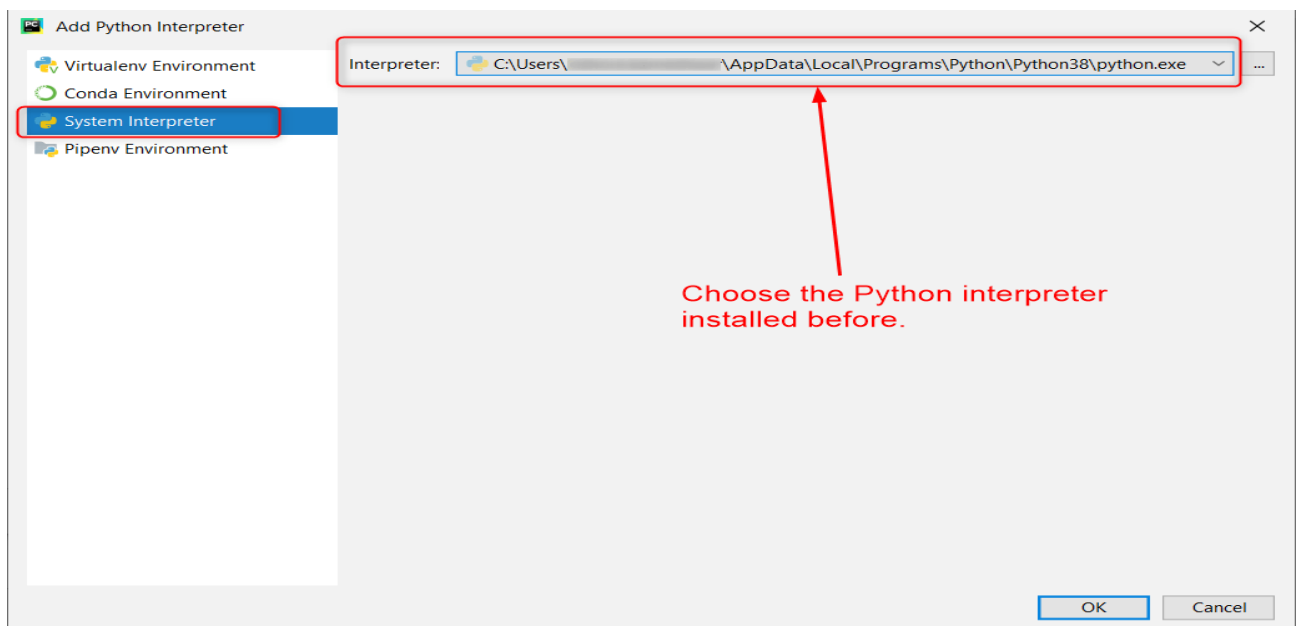
1. Launch the Pycharm
2. Click Configure > Settings as shown below.



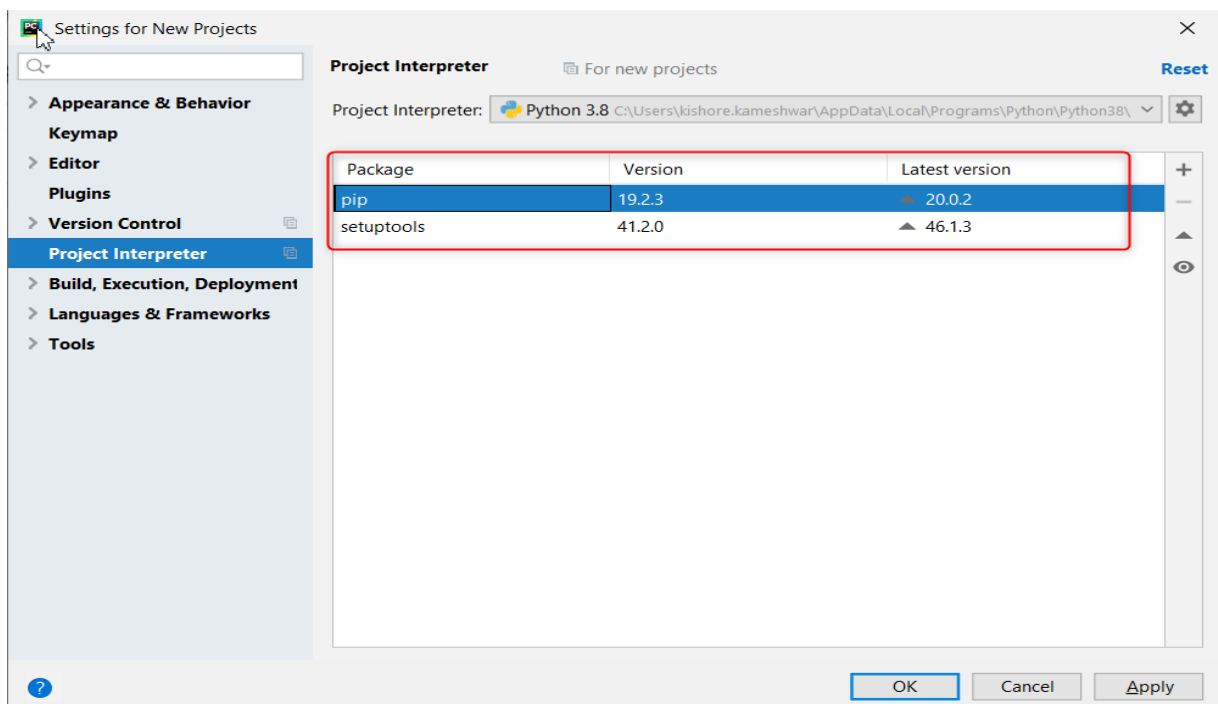
3. This will open up the Settings screen.



4. In the next screen click '+' and this will display all the interpreters installed in the system. Choose the System Interpreter left pane and select the interpreter path which you installed before in the Python Installation section and Click OK button.



5. In the next screen, click OK and Apply to finish the configuration. This screen displays the available packages in the selected Python interpreter.



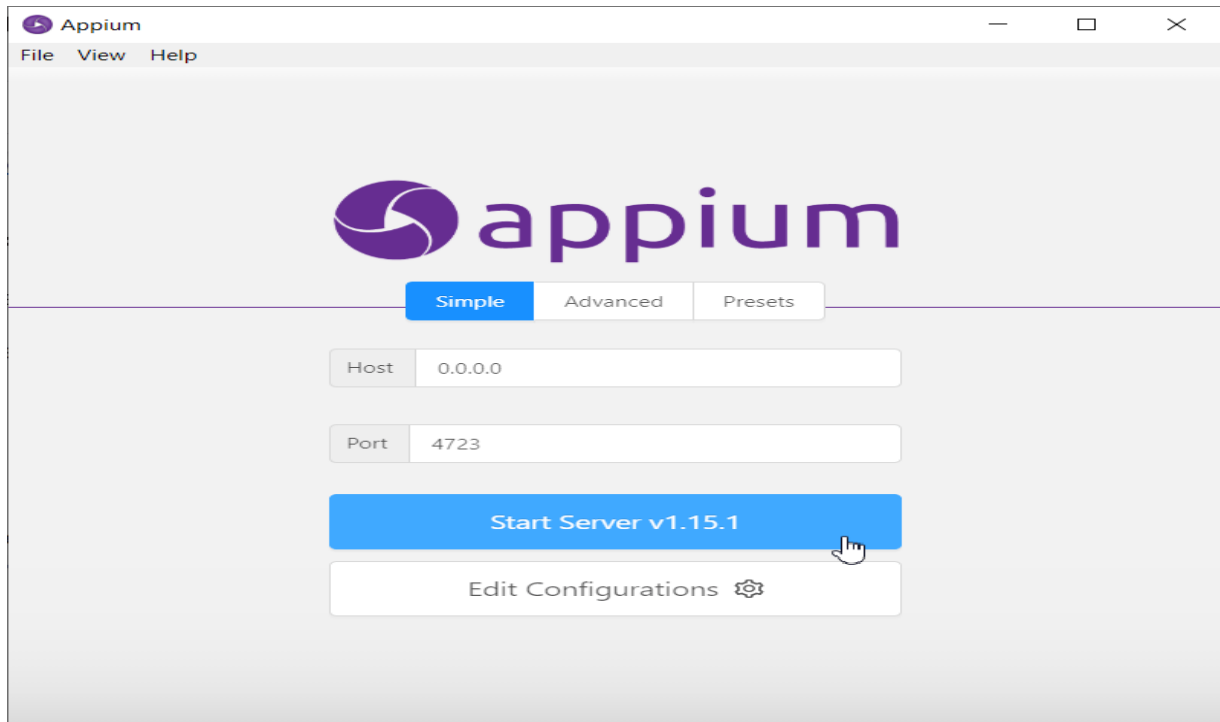
## 2.6.2 CREATING TEST USING APPIUM AND PYTHON



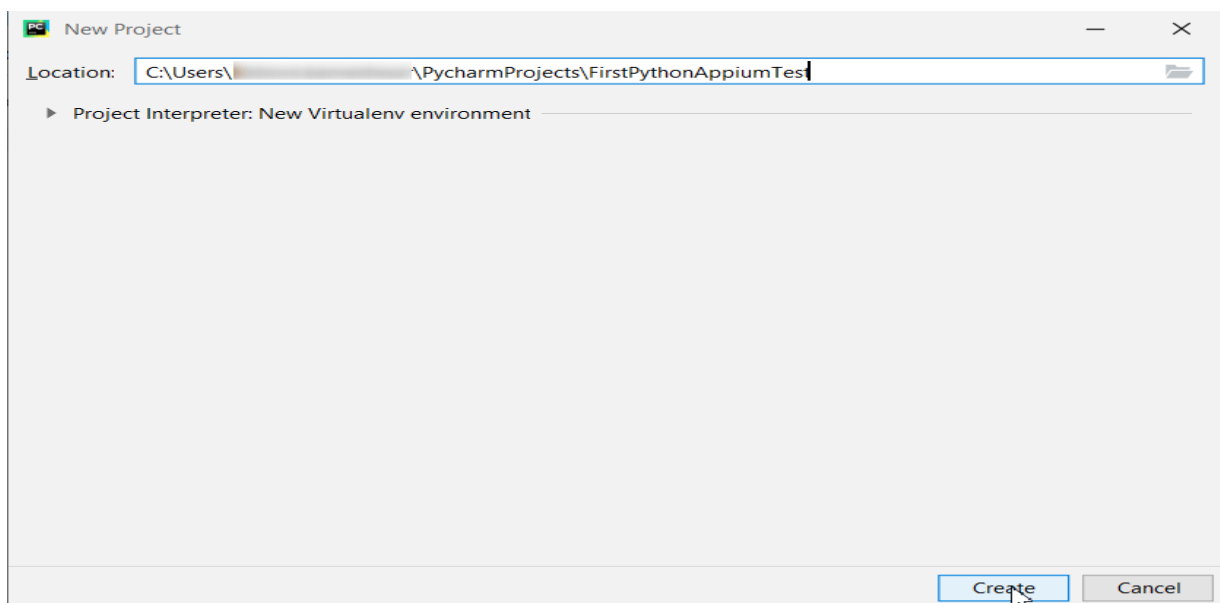
### Step 1. Start Appium Server

Launch the Appium Desktop and start the server.

By default it will run on localhost:4723. This example will consider the default

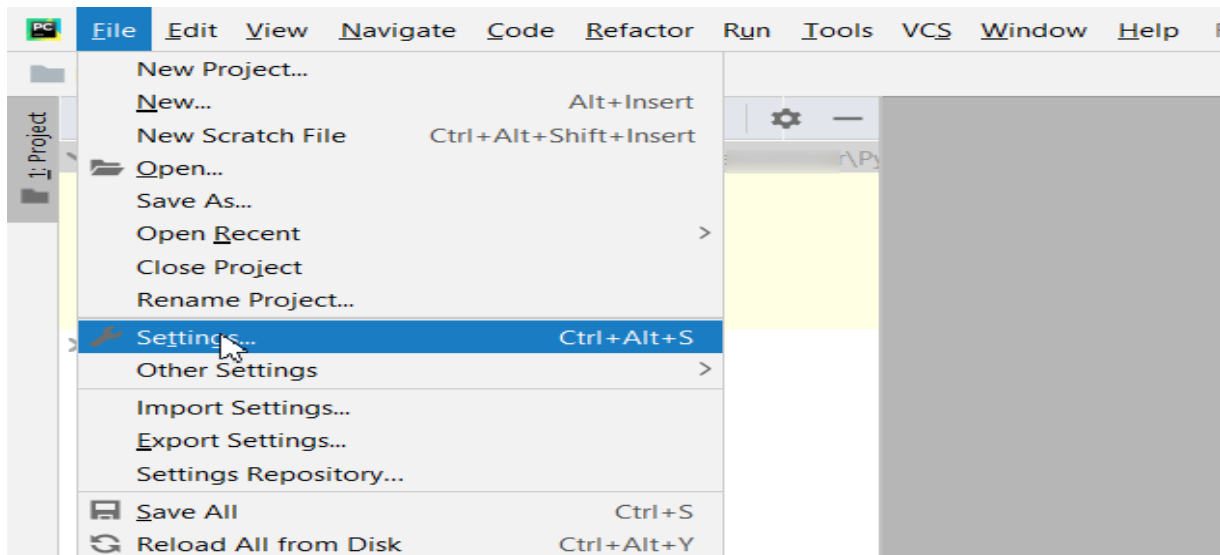


Step 2. Create a Python project in Pycharm, name the Project and click Create to create the project.



Step3. Set the Interpreter and add the Appium dependency for the project.

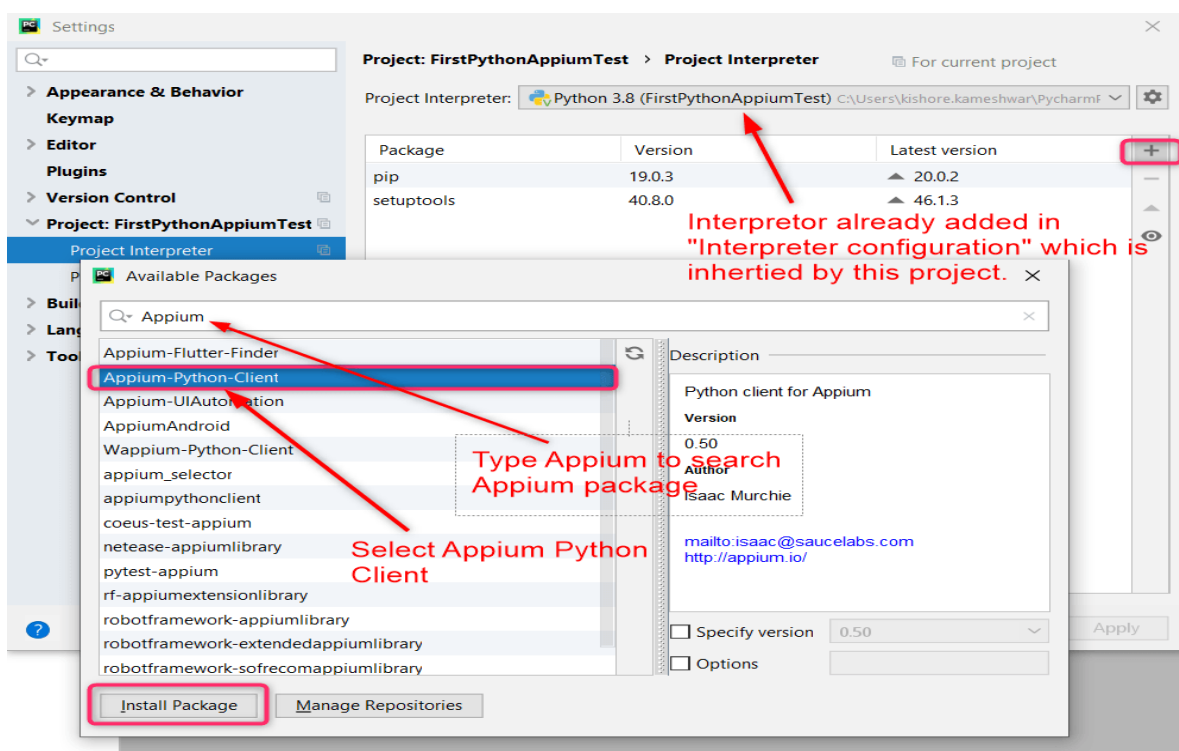
Select the newly created project and in the menu navigate to File > Settings



The interpreter for the project will be set automatically, See 'Interpreter Configuration' section above.

Click '+' and then search for Appium in Available Packages dialog box.

Select Appium-Python-Client and click Install Package to install Appium Python Client.

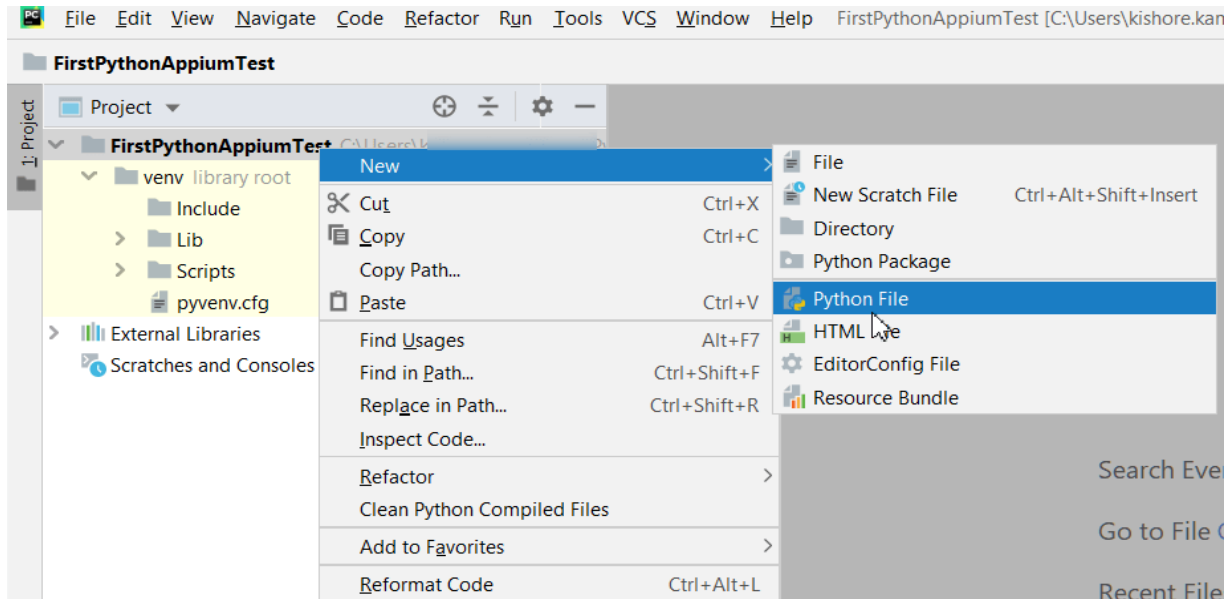


This will download the Appium Client dependency for this project.

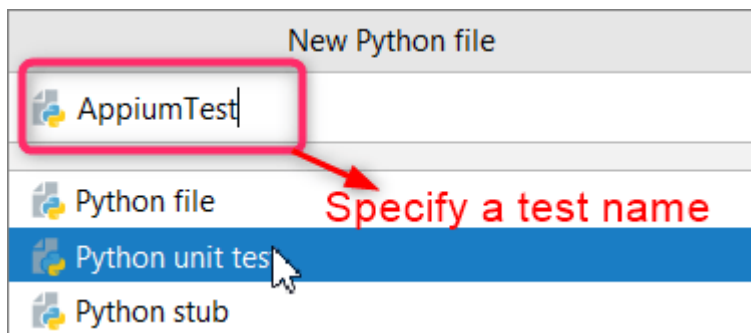
Click OK to save the project.

Step4.Creating a test in Python.

Select the project and then right-click. Click New > Python File



This will open a dialog box, Specify a test name and click the Python unit test.



This will create an initial Python unit test class.

We must now create an Appium Driver instance by passing the Desired Capabilities in the initialization function of the test. Basically the Appium Driver instance connects to the device and installs the application in the device.

## CHAPTER 3

### IMPLEMENTATION

While Appium can be used on both IOS and Android devices. In this project, we have used Appium to test an application on an Android device. Given below, is the working of Appium on an Android device.

#### 3.2 WORKING OF APPIUM ON ANDROID

Appium uses the UIAutomator framework to interact with the UI elements of the application on an Android device. The UIAutomator is a framework that is developed by the Android developers to test its user interface.

Here, Instead of UIAutomation API as in Apple, we have a UIAutomator/Selendroid and bootstrap.jar in place of the bootstrap.js file. It supports all Android versions that are greater than or equal to 17; for earlier versions of Android, Appium uses the Selendroid framework.

When we execute the test scripts, Appium sends the command in JSON format to the UIAutomator or Selendroid based on the Android version. Here, bootstrap.jar acts as a TCP server, which we can use to send the test command to execute the action on the Android device using UIAutomator/Selendroid.

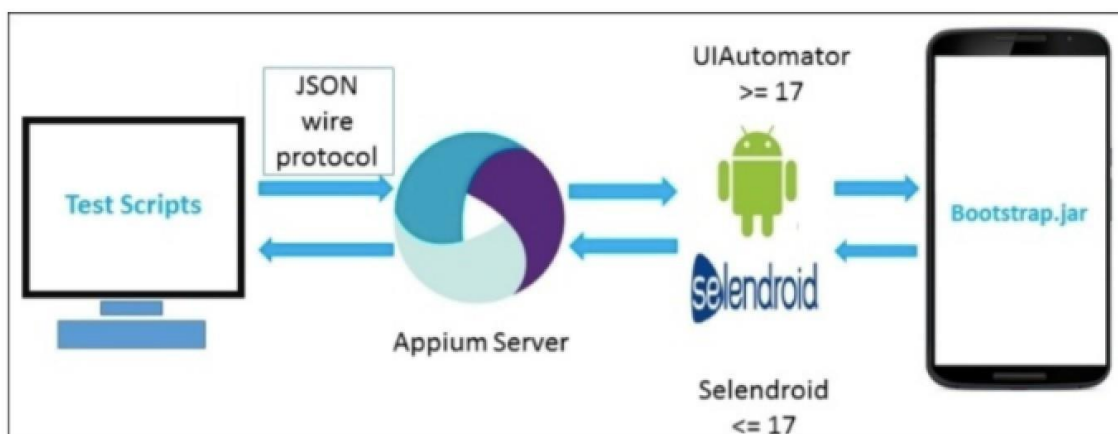


Fig.3.1. Working of Appium on Android

### 3.3 PROGRAM CODE

```
# This sample code uses the Appium python client

# pip install Appium-Python-Client

# Then you can paste this into a file and simply run with Python


from appium import webdriver

from appium.webdriver.common.touch_action import TouchAction

from time import sleep

#Test count

passcount = 0

totaltest = 0


#Setting Up Desired Capabilities of Appium for Automation

caps = { }

caps["platformName"] = "Android"

caps["platformVersion"] = "11"

caps["deviceName"] = "Pixel 4a"

caps["automationName"] = "Appium"

caps["appPackage"] = "com.google.earth"

caps["appActivity"] = "com.google.android.apps.earth.EarthActivity"

caps["ensureWebviewsHavePages"] = True


driver = webdriver.Remote("http://localhost:4723/wd/hub", caps)


driver.implicitly_wait(10)
```

#Test 1 (check if right package is selected )

```
totaltest += 1
```

```
pakkage =str(driver.current_package);
```

```
if(( pakkage == 'com.google.earth' ) ):
```

```
    passcount = passcount + 1
```

```
    print("\n test 1 : checking if correct pakkage is oppened \n test has passed \n")
```

#Test 2 (check if right activity is selected )

```
totaltest += 1
```

```
activity = str(driver.current_activity);
```

```
if( activity == 'com.google.android.apps.earth.EarthActivity' ) :
```

```
    passcount = passcount + 1
```

```
    print("\n test 2 : checking if correct activity is oppened \n test has passed \n")
```

#Test 3 (selecting an account )

```
totaltest += 1
```

```
e11 = driver.find_element_by_id("com.google.earth.id/og_apd_ring_view")
```

```
e11.click()
```

```
sleep(.5)
```

```
e12 = driver.find_element_by_accessibility_id("Use Sathvik D N sathvik.dn@gmail.com")
```

```
el2.click()
```

```
sleep(.5)
```

```
#checking if the profile is selected
```

```
el3 = driver.find_element_by_id("com.google.earth:id/og_apd_ring_view")
```

```
el3.click()
```

```
el = driver.find_element_by_id('com.google.earth:id/account_display_name')
```

```
if (el.text == "Sathvik D N"):
```

```
    passcount = passcount + 1
```

```
    print("\n test 3 : selecting an account \n test has passed \n")
```

```
el4 = driver.find_element_by_accessibility_id("Close")
```

```
el4.click()
```

```
#Test 4 (Testing I'm feeling lucky)
```

```
totaltest += 1
```

```
sleep(.5)
```

```
el5 = driver.find_element_by_accessibility_id("I'm Feeling Lucky")
```

```
el5.click()
```

```
TouchAction(driver) .press(x=588, y=2074) .move_to(x=571, y=1556) .release()  
.perform()
```

```
el7 = driver.find_element_by_id("com.google.earth:id/knowledge_card_title")
```

```
TouchAction(driver).tap(x=73, y=225).perform()
```

```
if (el7.text != "Kumano"):
```

```
    passcount = passcount + 1
```

```
    print("\n test 4 : Testing I'm feeling lucky \n test has passed \n")
```

```
#Test 5 ( Testing search )
```

```
totaltest += 1
```

```
el8 = driver.find_element_by_accessibility_id("Open navigation drawer")
```

```
el8.click()
```

```
el9=driver.find_element_by_xpath("/hierarchy/android.widget.FrameLayout/android.widget.LinearLayout/android.widget.FrameLayout/android.widget.LinearLayout/android.widget.FrameLayout/android.view.ViewGroup/androidx.drawerlayout.widget.DrawerLayout/android.widget.FrameLayout/android.widget.LinearLayout/android.widget.FrameLayout/android.widget.FrameLayout/android.support.v7.widget.RecyclerView/android.support.v7.widget.LinearLayoutCompat[1]/android.widget.CheckedTextView")
```

```
el9.click()
```

```
el10 = driver.find_element_by_id("com.google.earth:id/search_text_view")
```

```
el10.set_text("Mangalore")
```

```
driver.execute_script('mobile: performEditorAction', {'action': 'search'})
```

```
sleep(6)
```

```
el11 = driver.find_element_by_id("com.google.earth:id/knowledge_card_title")
```

```
if (el11.text == "Mangalore"):
```

```
    passcount = passcount + 1
```

```
    print("\n test 5 : Testing search \n test has passed \n")
```

```
sleep(3)
```



#Test 6 ( Testing for Invalid element id ))

totaltest += 1

el8 = driver.find\_element\_by\_accessibility\_id("Open navigation drawer")

el8.click()

el9 =

driver.find\_element\_by\_xpath("/hierarchy/android.widget.FrameLayout/android.widget.LinearLayout/android.widget.FrameLayout/android.widget.LinearLayout/android.widget.FrameLayout/android.view.ViewGroup/androidx.drawerlayout.widget.DrawerLayout/android.widget.FrameLayout/android.widget.LinearLayout/android.widget.FrameLayout/android.widget.FrameLayout/android.support.v7.widget.RecyclerView/android.support.v7.widget.LinearLayoutCompat[1]/android.widget.CheckedTextView")

el9.click()

el10 = driver.find\_element\_by\_id("com.google.earth:id/search\_text\_view")

el10.set\_text("Bangalore")

driver.execute\_script('mobile: performEditorAction', {'action': 'search'})

sleep(6)

try:

el11 = driver.find\_element\_by\_id("com.doogle.earth:id/knowledge\_card\_title") #wrong id

if (el11.text == "Bengaluru"):

passcount = passcount + 1

print("\n test 6 : Testing search \n test has passed \n")

sleep(3)

except:

print("\n test 6 : Testing search \n test has Failed \n")

```
print("\n total number of cases      : "+ str(totaltest))

print("\n total number of cases passed  : "+ str(passcount))

print("\n total number of cases failed  : "+ str(totaltest - passcount ))

driver.quit()
```

## CHAPTER 4

### TESTING

Software Testing is a method to check whether the actual software product matches expected requirements and to ensure that software product is Defect free. It involves execution of software/system components using manual or automated tools to evaluate one or more properties of interest. The purpose of software testing is to identify errors, gaps or missing requirements in contrast to actual requirements.

Some prefer saying Software testing definition as a White Box and Black Box Testing. In simple terms, Software Testing means the Verification of Application Under Test (AUT).

#### 4.1 TEST CASES

TEST CASE ID	TEST CASE DESCRIPTION	STEPS	ACTUAL OUTPUT	REMARKS
1	Testing app package	Use driver.current_package	App package found	TEST PASSED

2	Testing app activity	Use driver.current_activity	App activity found	TEST PASSED
3	Select an account	1) Select profile logo in top bar 2) Select your account	Google account of selected profile is displayed	TEST PASSED
4	Testing I'm feeling lucky	Click on 'I'm Feeling Lucky' button on Top bar	The result is random	TEST PASSED
5	Testing search	1). Click on search button in Top bar 2)Enter 'Empire Stste Building' 3) Get the location of the place	The address is match	TEST PASSED
6	Testing for invalid element id	Give a wrong id	should give an exception which will be handeled	TEST FAILED

Table.4.1. Test Case Table

## 4.2 TEST REPORT

A test report is an organized summary of testing objectives, activities, and results. It is created and used to help developers understand product quality and decide whether a product, feature, or a defect resolution is on track for release.

The Test Report of our project is shown below :

1. Number of Test Cases Executed : 06
2. Number of Test Cases Passed : 05
3. Number of Test Cases Failed : 01
4. Number of Defects Raised : 01

## CHAPTER 5

### SNAPSHOTS

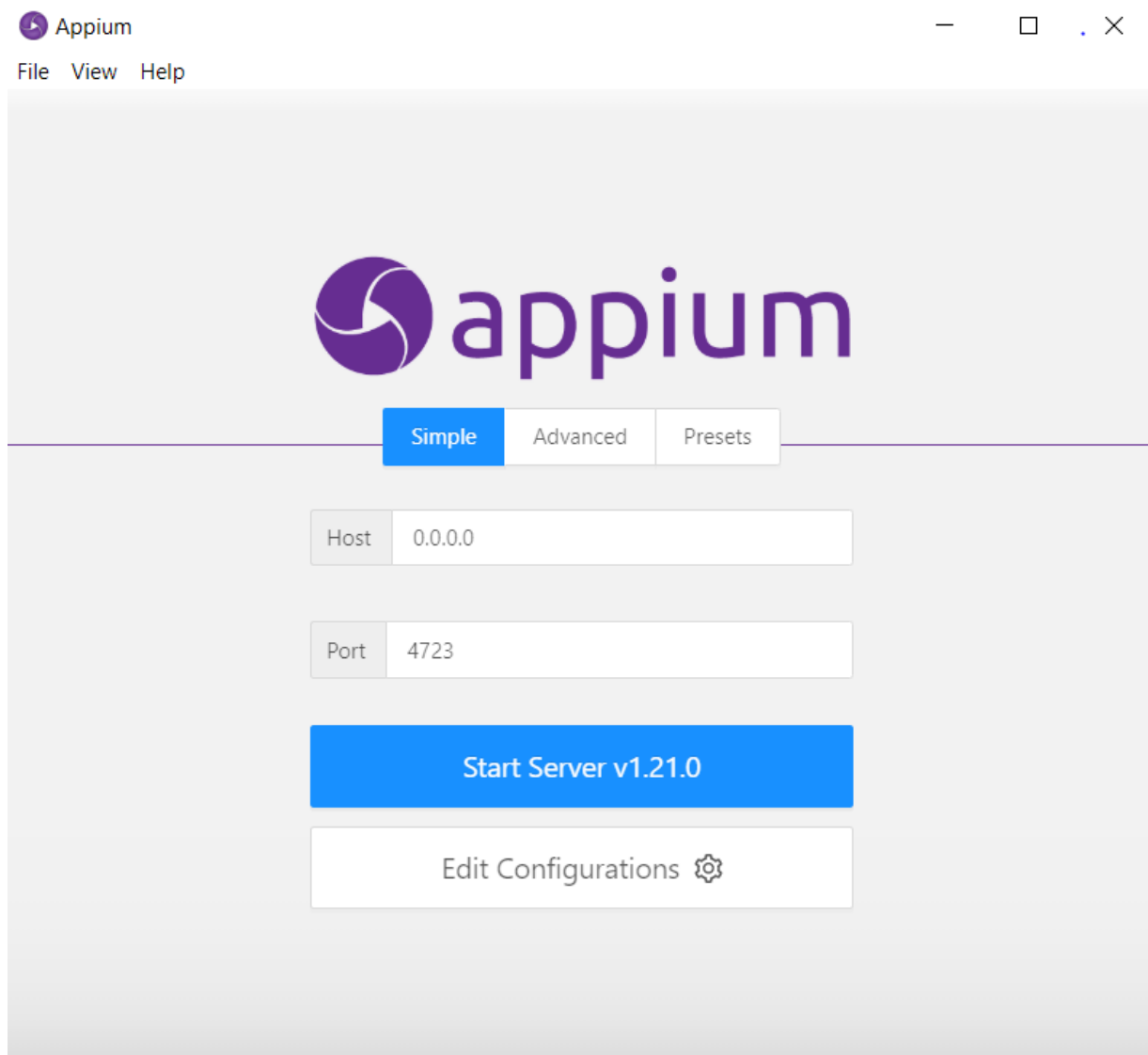


Fig.5.1. Launching Appium Server

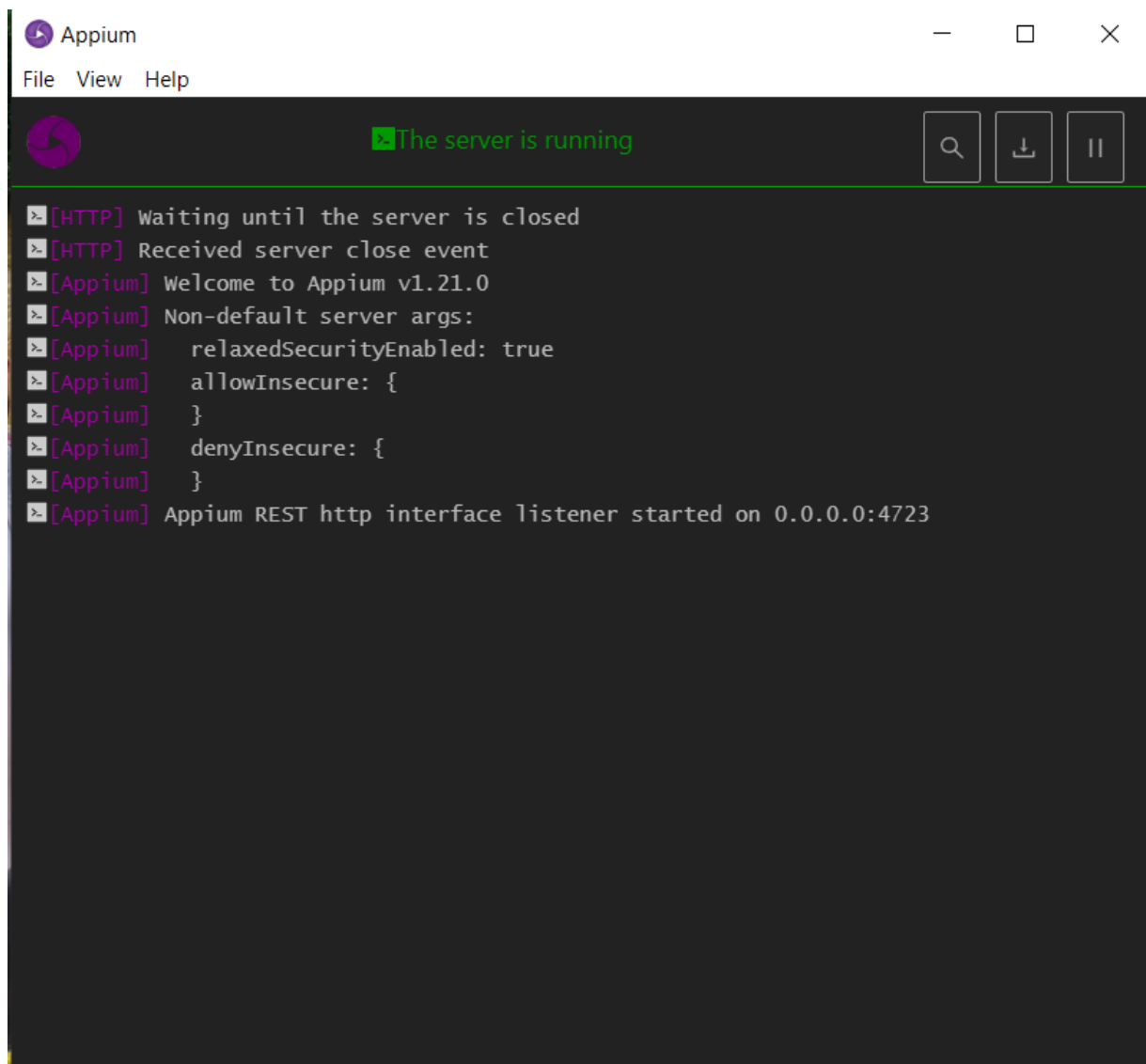


Fig.5.2. Appium Server running and go to start inspector session

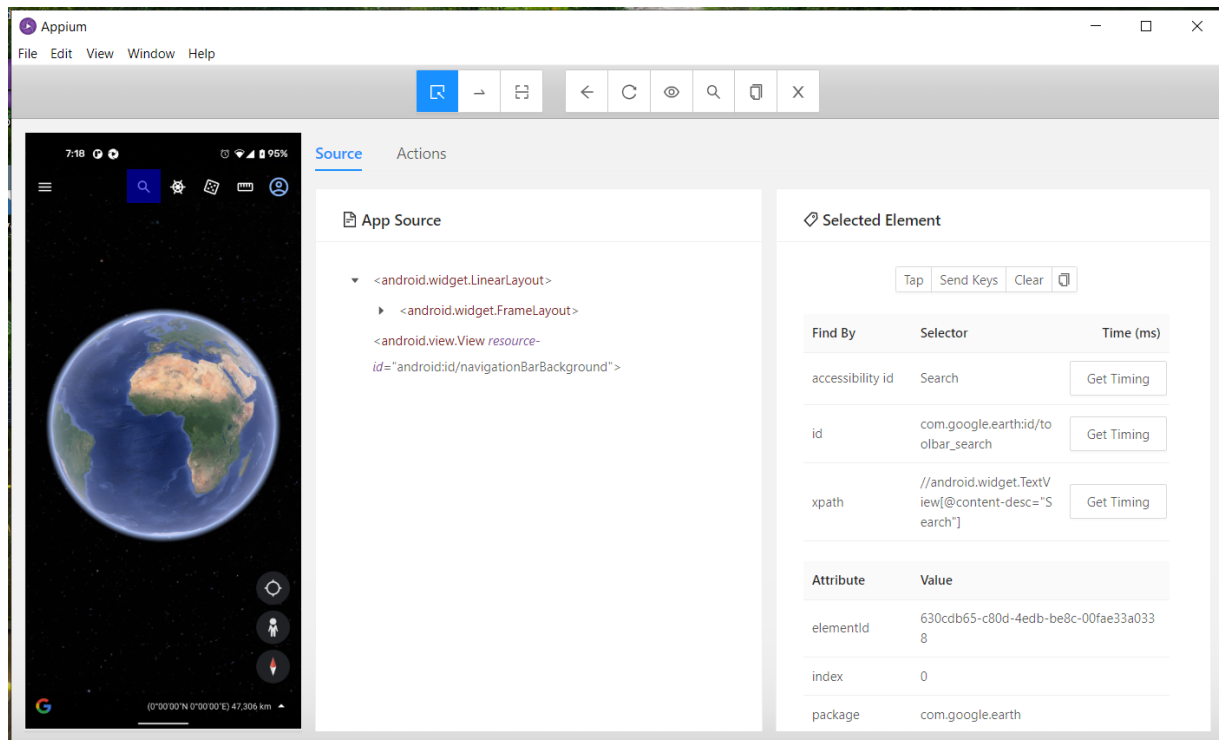


Fig.5.3 Creating a program to automate Google earth using Appium

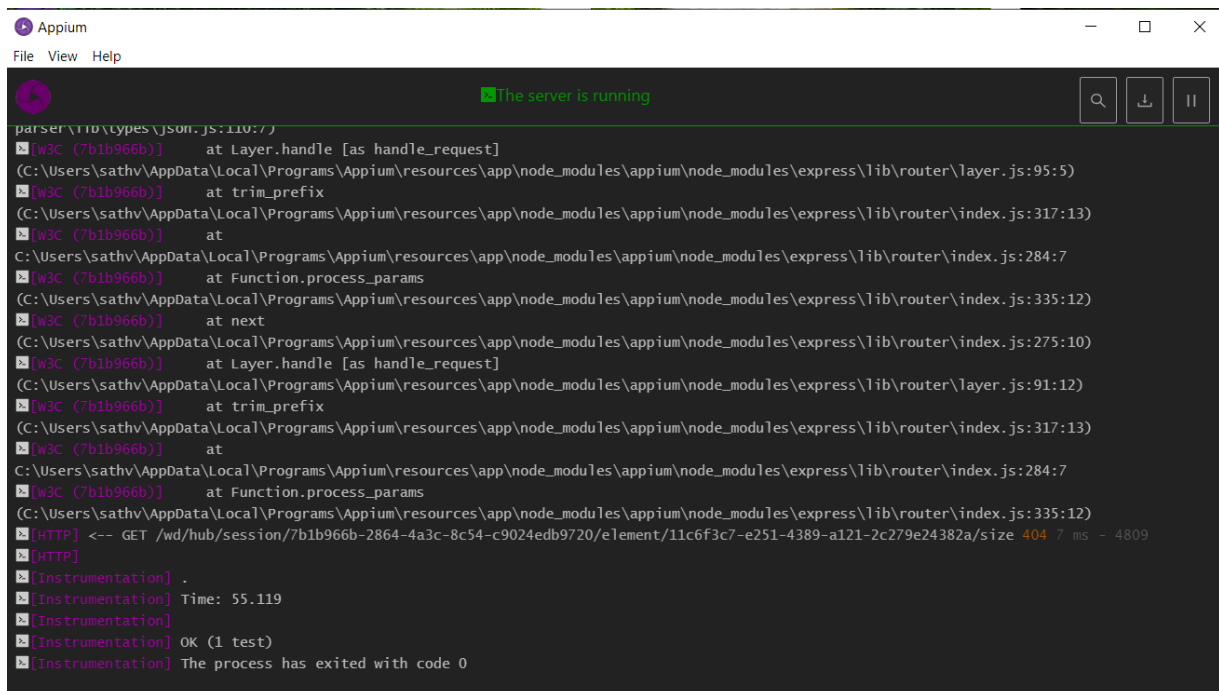
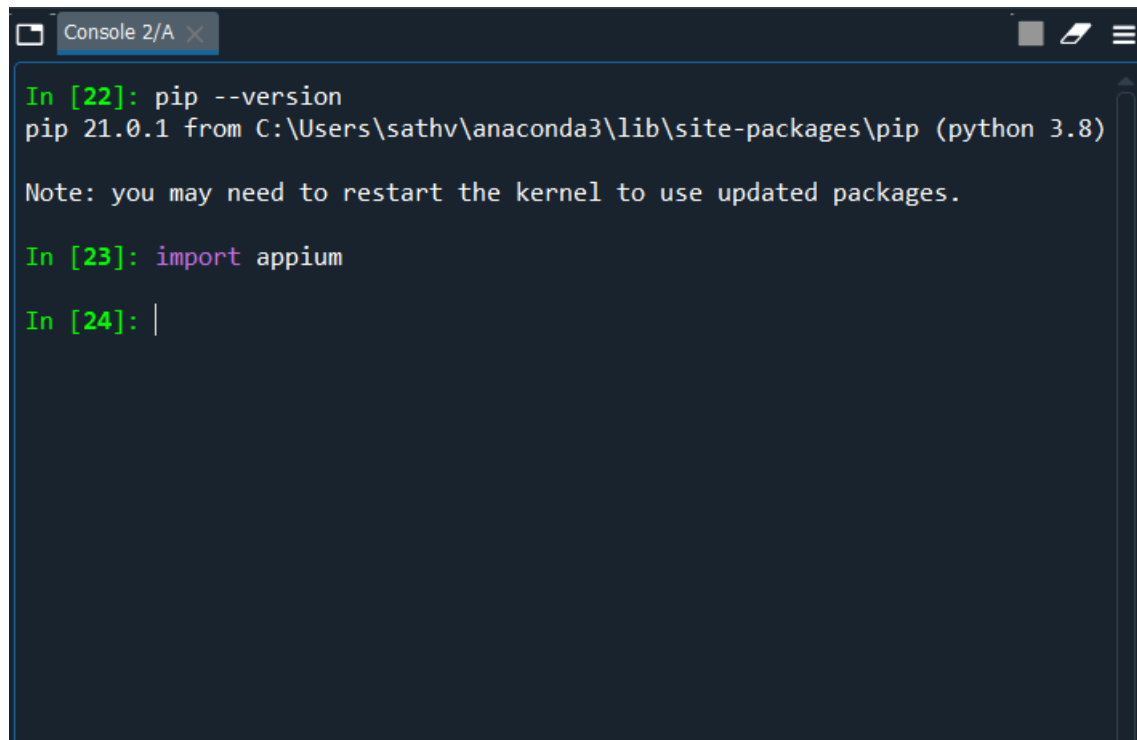


Fig.5.4. Appium inspector log after the session



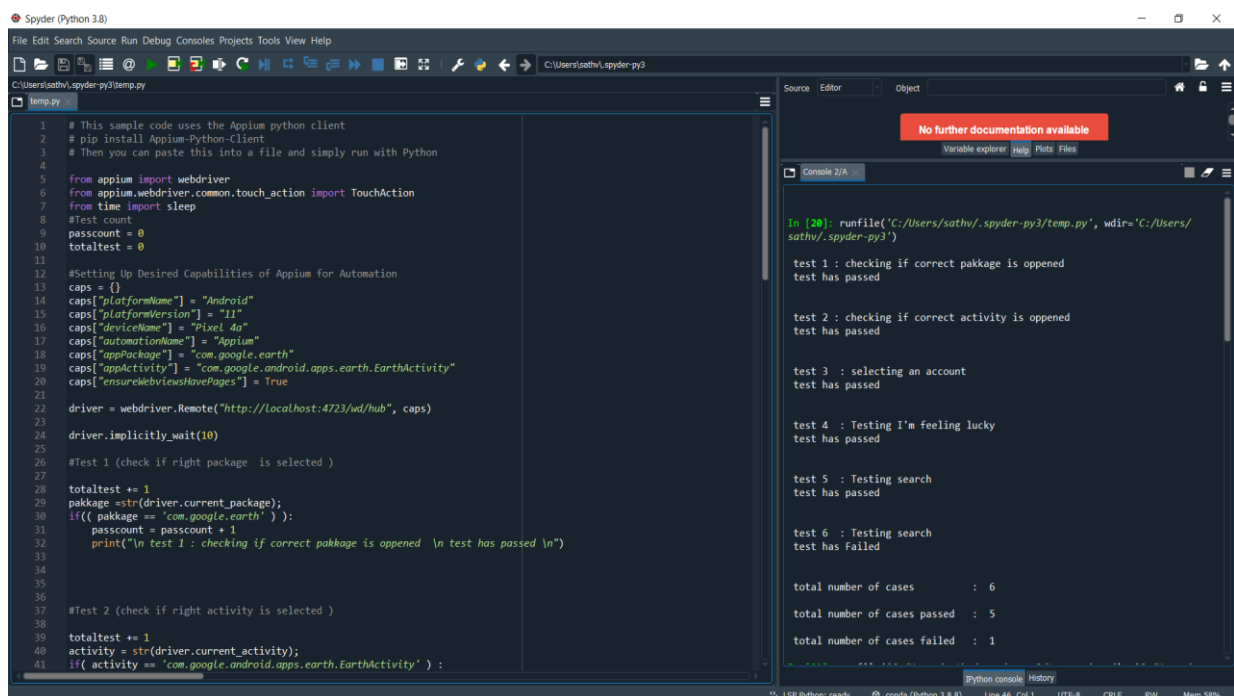
```
Console 2/A x
In [22]: pip --version
pip 21.0.1 from C:\Users\sathv\anaconda3\lib\site-packages\pip (python 3.8)

Note: you may need to restart the kernel to use updated packages.

In [23]: import appium

In [24]: |
```

Fig.5.5. Install pip and then import appium



```
1 # This sample code uses the Appium python client
2 # pip install Appium-Python-Client
3 # Then you can paste this into a file and simply run with Python
4
5 from appium import webdriver
6 from appium.webdriver.common.touch_action import TouchAction
7 from time import sleep
8 #Test count
9 passcount = 0
10 totaltest = 0
11
12 #Setting Up Desired Capabilities for Automation
13 caps = {}
14 caps["platformName"] = "Android"
15 caps["platformVersion"] = "11"
16 caps["deviceName"] = "Pixel 4a"
17 caps["automationName"] = "Appium"
18 caps["appPackage"] = "com.google.earth"
19 caps["appActivity"] = "com.google.android.apps.earth.EarthActivity"
20 caps["ensureWebViewsHavePages"] = True
21
22 driver = webdriver.Remote("http://localhost:4723/wd/hub", caps)
23
24 driver.implicitly_wait(10)
25
26 #Test 1 (check if right package is selected )
27
28 totaltest += 1
29 package = str(driver.current_package);
30 if(( package == 'com.google.earth' ) ):
31     passcount = passcount + 1
32     print("\n test 1 : checking if correct package is opened \n test has passed \n")
33
34
35
36
37 #Test 2 (check if right activity is selected )
38
39 totaltest += 1
40 activity = str(driver.current_activity);
41 if( activity == 'com.google.android.apps.earth.EarthActivity' ) :
```

Console 2/A

```
In [20]: runfile('C:/Users/sathv/.spyder-py3/temp.py', wdir='C:/Users/sathv/.spyder-py3')

test 1 : checking if correct package is opened
test has passed

test 2 : checking if correct activity is opened
test has passed

test 3 : selecting an account
test has passed

test 4 : Testing I'm feeling lucky
test has passed

test 5 : Testing search
test has passed

test 6 : Testing search
test has failed

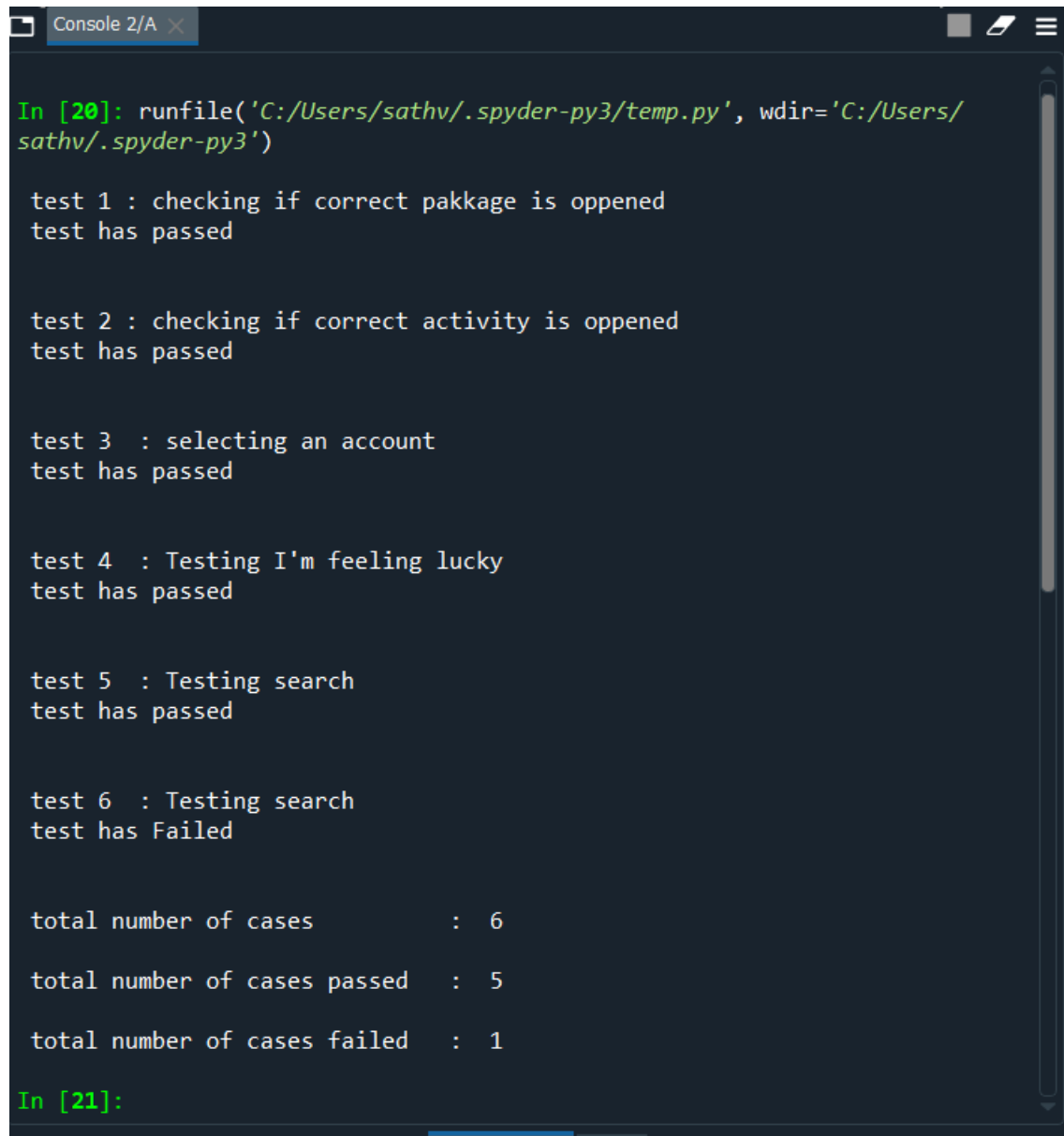
total number of cases      : 6
total number of cases passed : 5
total number of cases failed : 1
```

Fig.5.6. Run the code after all prerequisites are set



Fig.5.7. Appium Launching The Google earth



A screenshot of a Jupyter Notebook console window titled 'Console 2/A'. The window has a dark background with light-colored text. It shows the execution of a Python script using the 'runfile' function. The script performs six tests: test 1 (checking if correct package is opened), test 2 (checking if correct activity is opened), test 3 (selecting an account), test 4 (Testing I'm feeling lucky), test 5 (Testing search), and test 6 (Testing search). Tests 1 through 5 passed, while test 6 failed. A summary at the bottom shows 6 total cases, 5 passed, and 1 failed. The prompt 'In [21]:' is visible at the bottom.

```
In [20]: runfile('C:/Users/sathv/.spyder-py3/temp.py', wdir='C:/Users/sathv/.spyder-py3')

test 1 : checking if correct package is opened
test has passed

test 2 : checking if correct activity is opened
test has passed

test 3 : selecting an account
test has passed

test 4 : Testing I'm feeling lucky
test has passed

test 5 : Testing search
test has passed

test 6 : Testing search
test has Failed

total number of cases      : 6
total number of cases passed : 5
total number of cases failed : 1

In [21]:
```

Fig.5.8. Final output

## CHAPTER 6

## CONCLUSION

With plenty of mobile automation tools in the market, Appium comes with its own unique feature that drives Android/iOS systems to give user-friendly flexibility. It has its extension to work in native and hybrid mobile applications.

It can be written in any language and can be used for automated purposes without modifying anything. And Appium code can run in various devices, which are the best reliable and scalable choice for the automation of mobile.

It is a language-agnostic supports web driver application program interface, allows crossplatform tests. Applications are efficient, accurate, fast, and free from bugs, which saves a lot of time and cost. Keeping all this in mind learning appium will be worth it.

## CHAPTER 7

### REFERENCES

- <https://appium.io/>
- <https://www.javatpoint.com/appium>
- <https://en.wikipedia.org/wiki/Appium>
- <https://www.edureka.co/blog/appium-architecture/>