

# **APPIUM DOCUMENTATION FOR WINDOWS USERS**

1: What is Appium?	2
2: How to setup Appium?	3
3: Setting up connection with Appium	8
4: Writing first test in C#	16
5: Using advanced options	17

# 1: What is Appium?

Appium is an open-source tool for automating native, mobile web, and hybrid applications on iOS mobile, Android mobile, and Windows desktop platforms. **Native applications** are those written using the iOS, Android, or Windows SDKs. **Mobile web apps** are web applications accessed using a mobile browser (Appium supports Safari on iOS and Chrome or the built-in 'Browser' application on Android). **Hybrid applications** have a wrapper around a "webview" -- a native control that enables interaction with web content. Appium was designed to meet mobile automation needs according to a philosophy outlined by the following four tenets:

1. You should not have to recompile your app or modify it in any way in order to automate it,
2. You shouldn't be locked into a specific language or framework to write and run your tests,
3. A mobile automation framework shouldn't reinvent the wheel when it comes to automation APIs,
4. A mobile automation framework should be open source, in spirit and practice as well as in name.

Appium is at its heart a web server that exposes a REST (**Representational state transfer**) API. It receives connections from a client, listens for commands, executes those commands on a mobile device, and responds with an HTTP response representing the result of the command execution. The fact that we have a client/server architecture opens up a lot of possibilities: we can write our test code in any language that has a http client API, but it is easier to use one of the Appium client libraries. There are client libraries (in Java, Ruby, Python, PHP, JavaScript, and C#) which support Appium's extensions to the WebDriver protocol. When using Appium, you want to use these client libraries instead of your regular WebDriver client. We can put the server on a different machine than our tests are running on. Also we can write test code and rely on a cloud service like Sauce Labs to receive and interpret the commands.

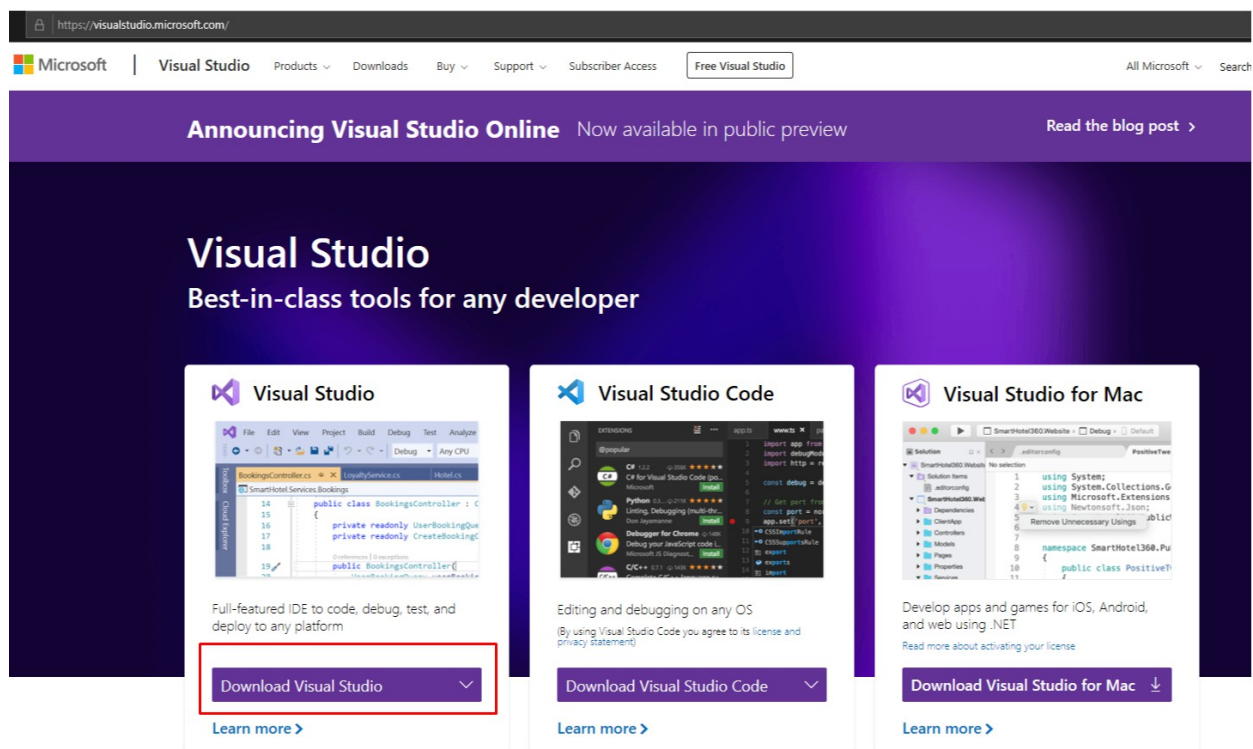
## 2: How to setup Appium in C# programming language?

In this chapter user will learn how to setup Appium in C# programming language.

To do this user must do several steps:

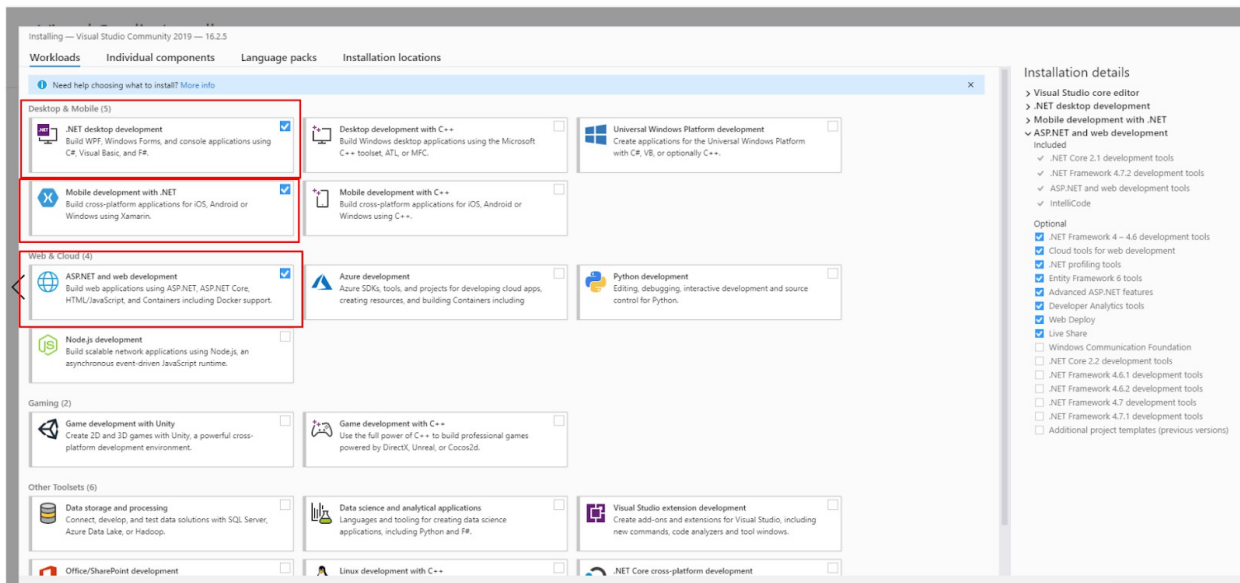
1. Install Visual Studio,
2. Install Java jdk 1.8,
3. Set Java home and Java path,
4. Install Node.js,
5. Install Appium desktop,
6. Set Android home and path.

To install Visual Studio user must go to next link: <https://visualstudio.microsoft.com/> (picture 1). There is option to download visual studio (Community 2019).



Picture 1: Visual studio

After download is completed, start installation. Select only option for picture 2.



Picture 2: Option that must be selected when installing Visual Studio

After installation was finished, we can go to next step, and that is installing Java jdk 1.8. Go to next link:

<https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>, and there download Windows x64, that is marked on picture 3.

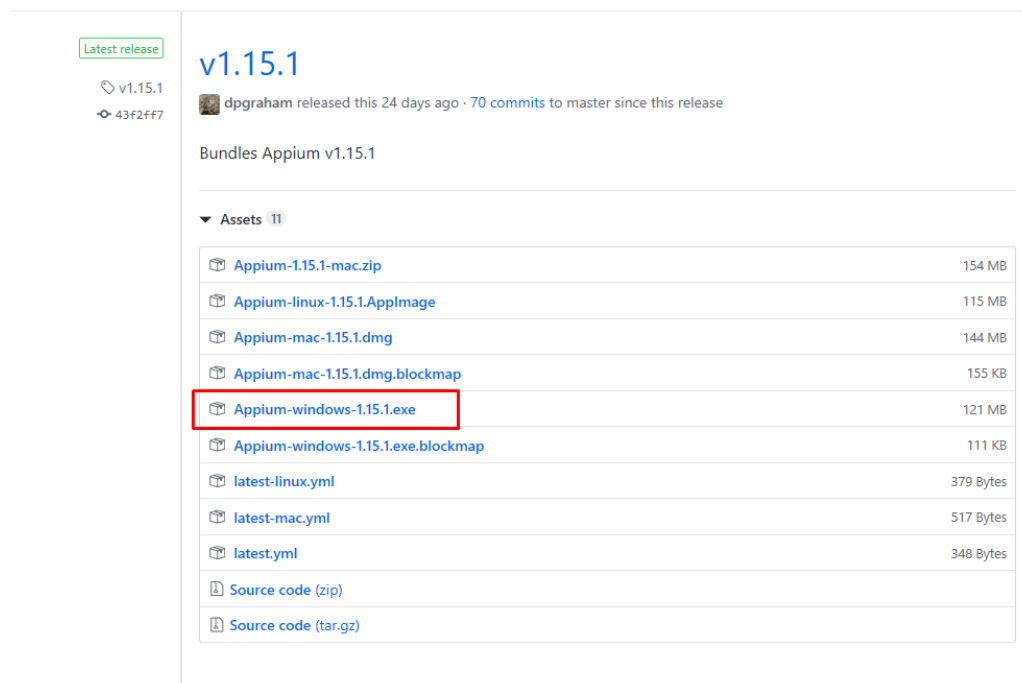
Java SE Development Kit 8u231		
You must accept the <b>Oracle Technology Network License Agreement for Oracle Java SE</b> to download this software.		
Thank you for accepting the Oracle Technology Network License Agreement for Oracle Java SE; you may now download this software.		
Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	72.9 MB	<a href="#">jdk-8u231-linux-arm32-vfp-hflt.tar.gz</a>
Linux ARM 64 Hard Float ABI	69.8 MB	<a href="#">jdk-8u231-linux-arm64-vfp-hflt.tar.gz</a>
Linux x86	170.93 MB	<a href="#">jdk-8u231-linux-i586.rpm</a>
Linux x86	185.75 MB	<a href="#">jdk-8u231-linux-i586.tar.gz</a>
Linux x64	170.32 MB	<a href="#">jdk-8u231-linux-x64.rpm</a>
Linux x64	185.16 MB	<a href="#">jdk-8u231-linux-x64.tar.gz</a>
Mac OS X x64	253.4 MB	<a href="#">jdk-8u231-macosx-x64.dmg</a>
Solaris SPARC 64-bit (SVR4 package)	132.98 MB	<a href="#">jdk-8u231-solaris-sparcv9.tar.Z</a>
Solaris SPARC 64-bit	94.16 MB	<a href="#">jdk-8u231-solaris-sparcv9.tar.gz</a>
Solaris x64 (SVR4 package)	133.73 MB	<a href="#">jdk-8u231-solaris-x64.tar.Z</a>
Solaris x64	91.96 MB	<a href="#">jdk-8u231-solaris-x64.tar.gz</a>
Windows x86	200.22 MB	<a href="#">jdk-8u231-windows-i586.exe</a>
Windows x64	210.18 MB	<a href="#">jdk-8u231-windows-x64.exe</a>

Picture 3: Java jdk 1.8 download

After download is finished start installing jdk. Next step after installation is finished is to add Java home and Java path. Next link will explain everything user must do: <https://javatutorial.net/set-java-home-windows-10>.

Next step is to Node.js, and we can do that by going to next link: <https://nodejs.org/en/download/>. After download and installation is finished, we can go further.

Go to next link: <https://github.com/appium/appium-desktop/releases/tag/v1.15.1>, so you can download and install Appium. That is for most recent update, for now (date of writing documentation - 06.19.2019.). Download file that is marked on photo 4.

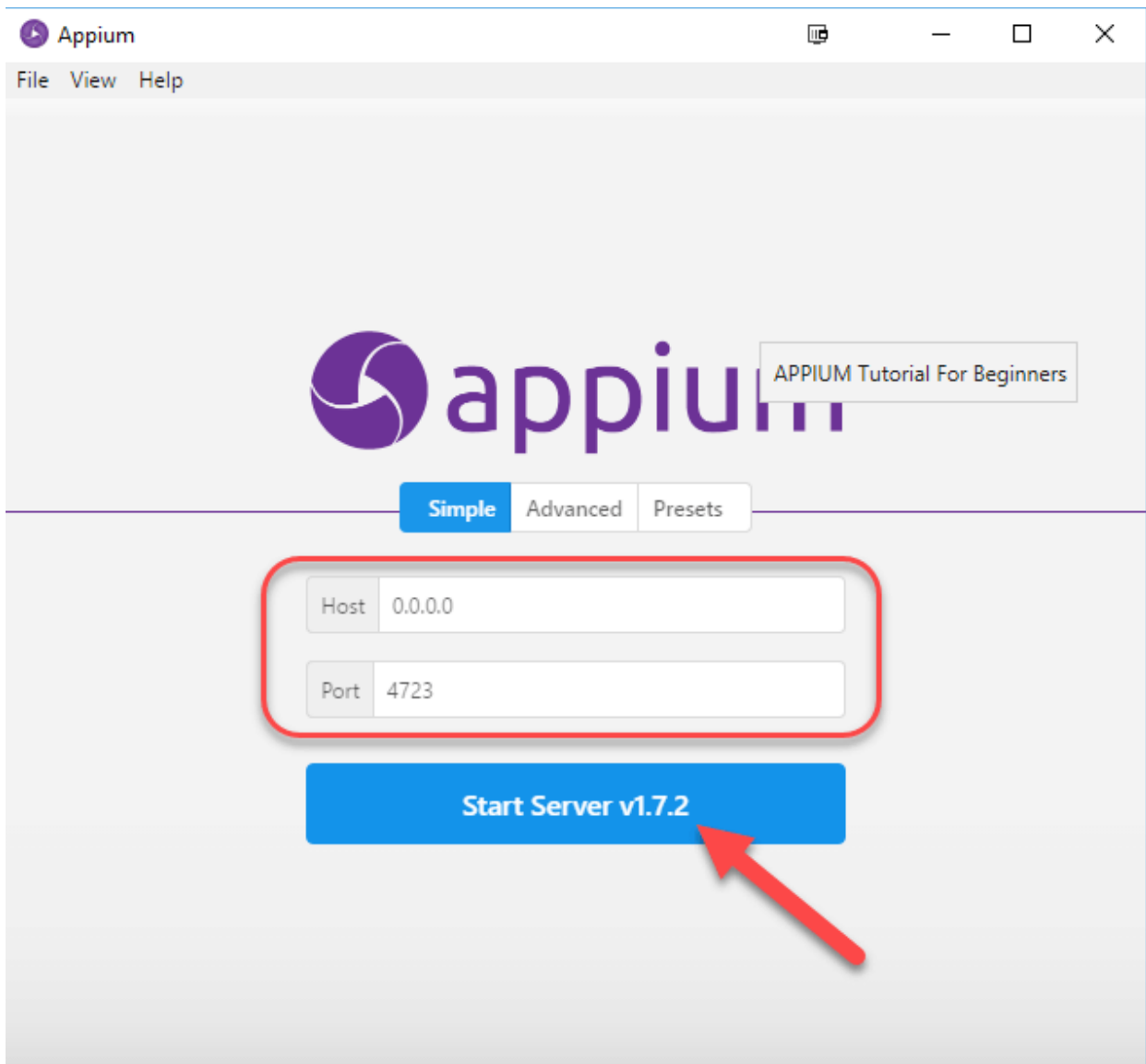


Picture 4: Download Appium

Last thing we need to do is to set Android home and path. Next link will give full guide for doing that:

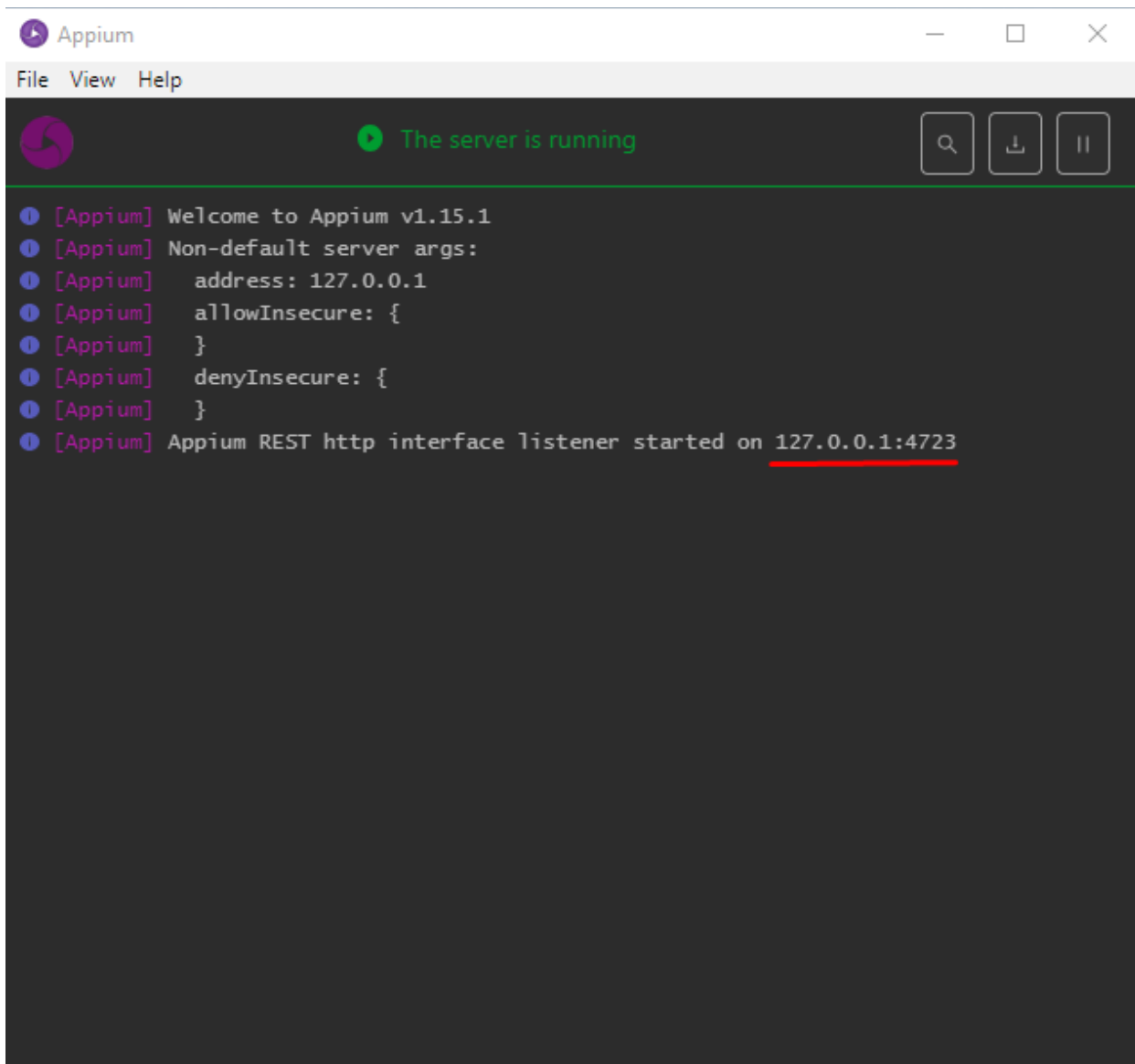
<http://www.automationtestinghub.com/setup-android-environment-variables/>.

Now we are able to start appium. Important to do before starting Appium is to connect our device with computer with USB, and to enable. To do this go to Setting on your ANDROID device, find developer option, and there find option "USB debugging" and turn it on. After that we can start Appium by opening location of Appium and finding Appium.exe file. Clicking on it will open new window (Picture 5).



*Picture 5: Opening Appium*

On this screen for now we will use Simple tab, where we need to enter Host: 127.0.0.1 and Port 4723. Clicking on Start Server (Blue button will have information about version of appium), will open new screen (picture 6).



Picture 6: Starting appium (127.0.0.1:4732)

If we did all things till now correctly Appium will be started. But to actually do something with it, we need to do much more. First we need to create Tests (That is why we install C#).

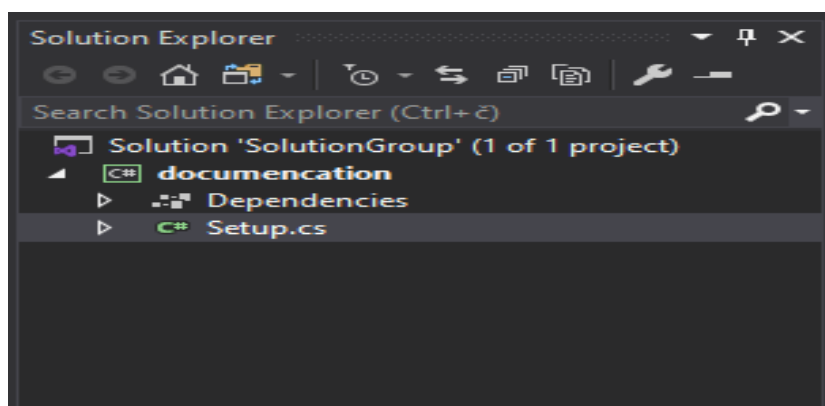


### 3: Creating first Test

To create test we need some programming language where we can write then. For this documentation we decide to use C#, but user can use Java, Ruby, Python.

Open C#, and start new project, where project type must be class library (there are others, like console application...). Add name and solution name (solution name is important because we will use it later to add new project to same solution). Also add location where project will be saved.

Rename class name, by right clicking on class, and then on option rename (i put Setup, because in this class i will setup appium specifications).



Picture 7: Rename Class name

If you do not see Solution explorer on your screen, click on View, and after that on solution explorer (shortcut: Ctrl + Alt + L).

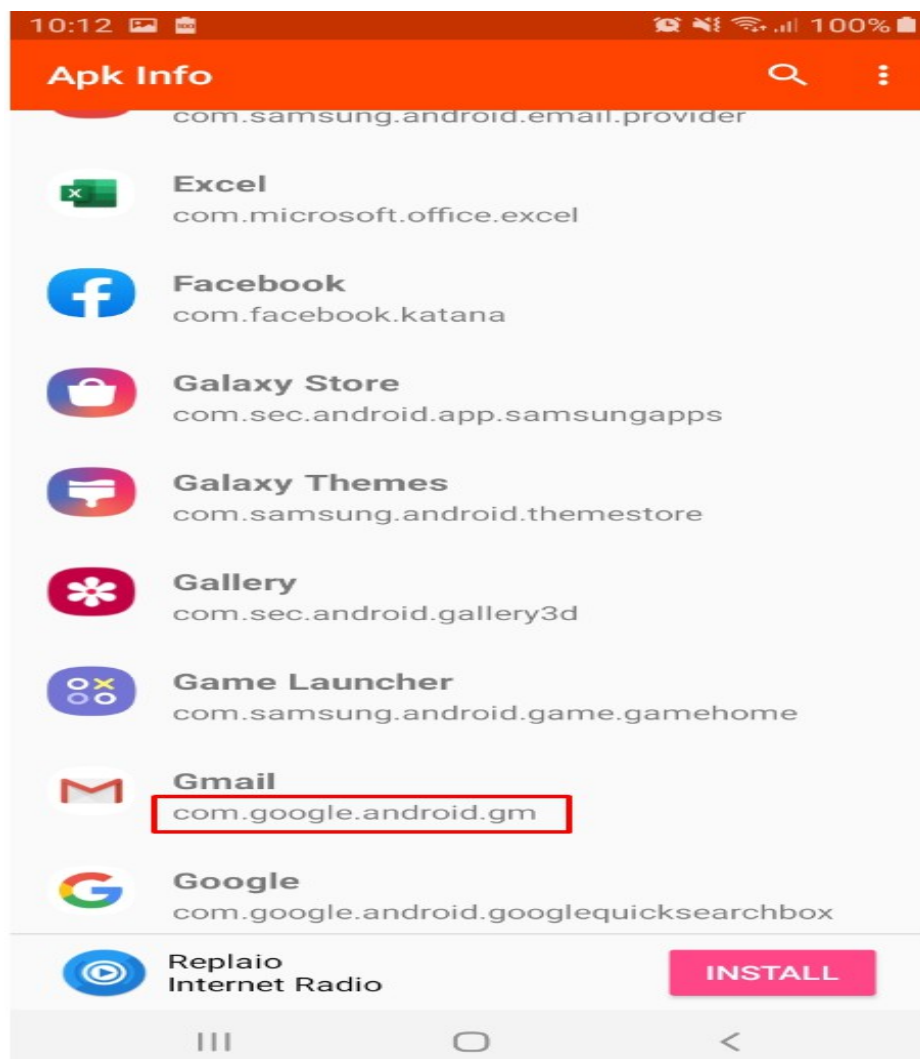
In Setup class enter from next picture:

```
namespace IndimaAndroid
{
    4 references
    public class InitialSetUp
    {
        public string androidVersion = "";
        public string deviceName = "";
        public string platformName = "";
        public string appPackage = "";
        public string appActivity = "";
        public string automationName = "";
    }
}
```

Picture 8: Setup variables

On picture 8 we have androidVersion (version of phone you want to use in testing), deviceName (random name i personally use name of Phone: Samsung Galaxy S7 or something like that, reason for this is that i can later start testing on more devices, and it is nice know what device user is using currently), platformName (In this case it will be Android), appActivity and appPackage (will show how to find it), and automationName (here use “UiAutomator2”, it is driver we will use).

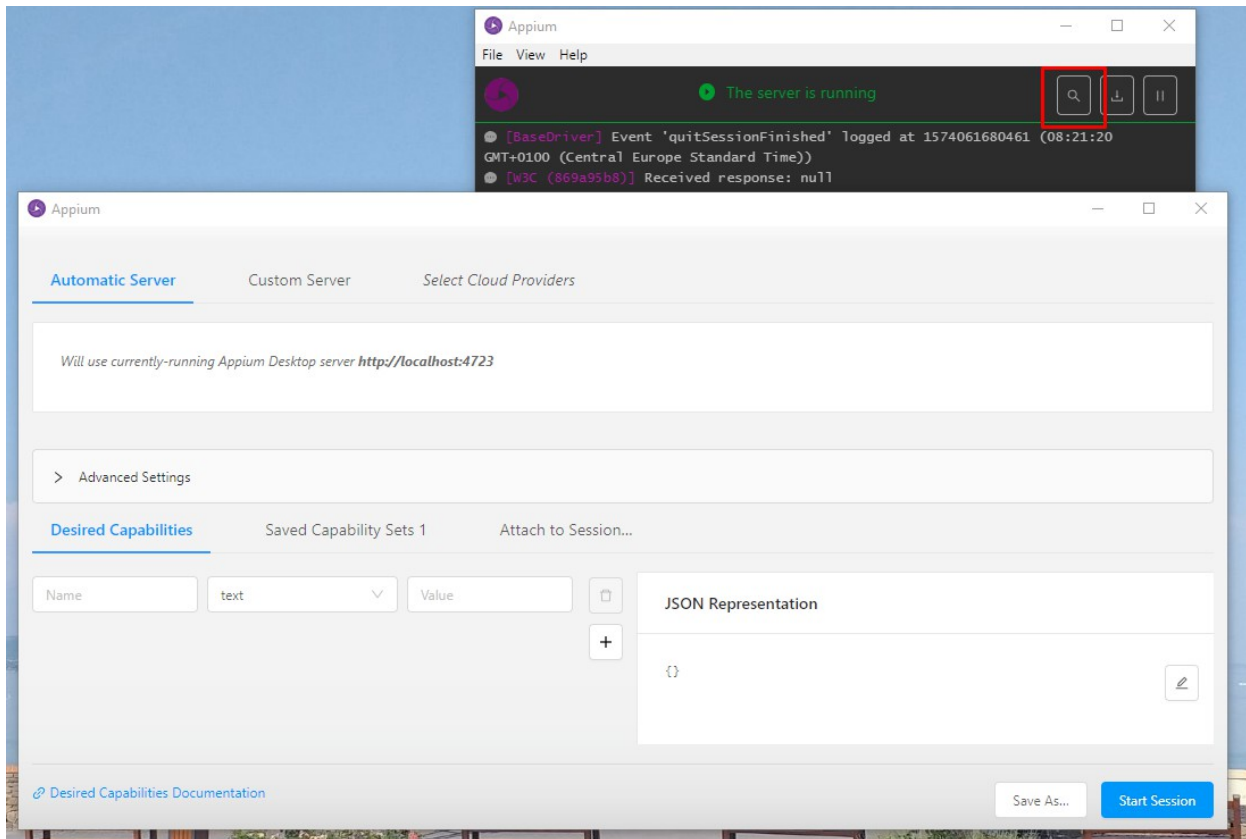
For appActivity and appPackage user must download “Apk Info” from play store. When Apk Info is installed open it and find application you want to test. For this example will use Gmail. On picture 9 we can see Gmail application and under it, there is label (com.google.android.gm), that label is appPackage.



Picture 9: Getting appPackage

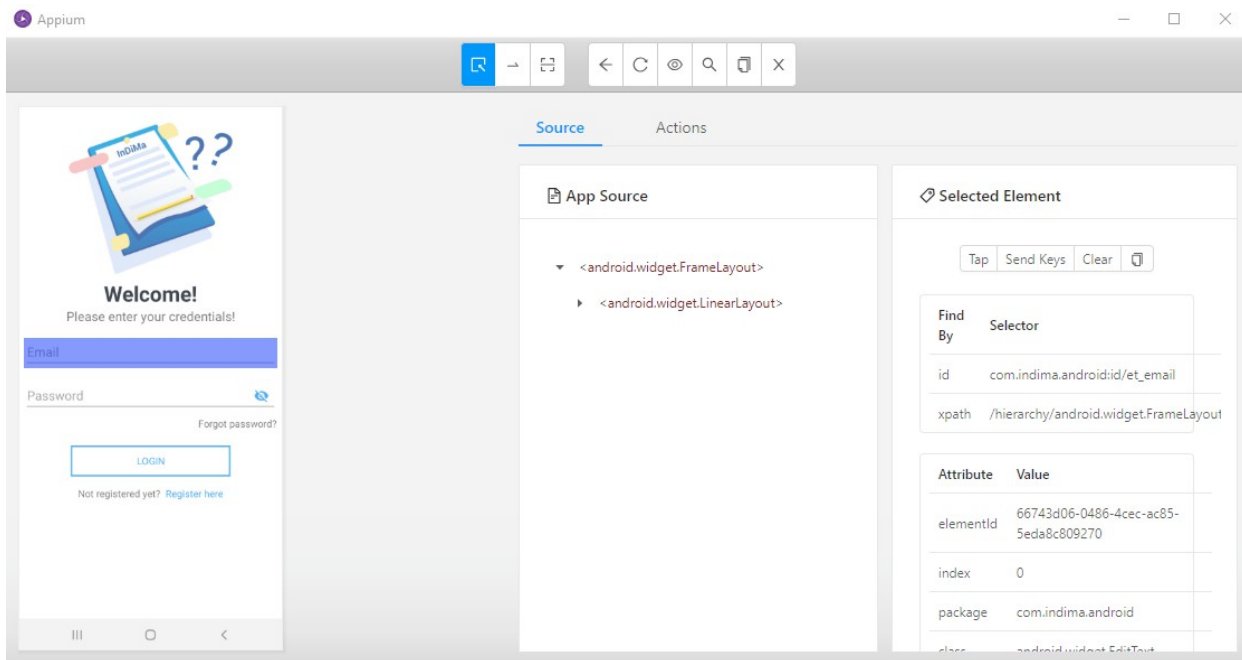
After this click on Gmail then find Activities. For smaller project user will get small list, Gmail have 47 activities. On my project appActivity that i used is SplashScreenActivity, activity that is started when user start application. Here i am not sure, try to find it or switch to another application. Now when we have all desired data, we can also start Appium Inspector in almost same way.

Open Appium and click on search icon. It will open Appium inspector (picture 10).



Picture 10: Appium Inspector (Setup)

Under desired Capabilities in field name enter platformVersion and in field value enter version of your phone (on picture 9 it was android version). Then click on plus sign, and add rest from picture 9, except automationName. Click on save As and save it. After that every time you start Appium inspector you will have configuration saved in Saved Capability Sets tab. Now click on Start Session, and wait couple of seconds, it will start new screen, and also will turn on application on your phone (only if appActivity that you provided is first one activity that will happen when user open application).

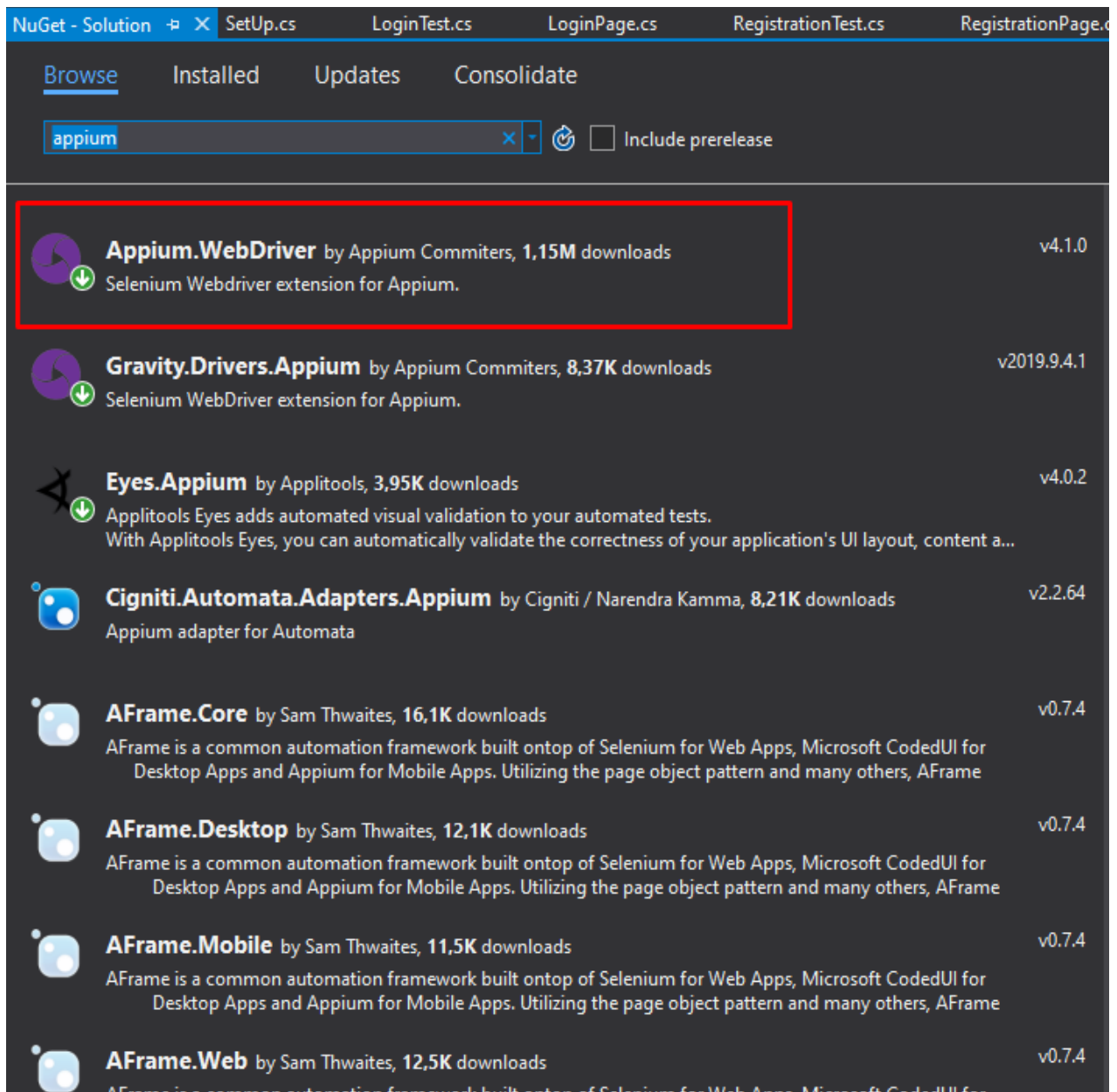


Picture 11: Appium Inspector (Started)

Here user can click on any element on screen (left side, where application screen will be displayed), find element by search tree (middle part of screen), or get specifics for desired element (right side of screen). On the right side of the screen, there are id, xPath, class, package, all can be used to interact with that specific element. In addition, user can tap on element (on phone or using Appium inspector, he can send key (send input to field: email, password...)), clear some field (email, password...). All this is an important part of automation testing, because we need some way to interact with desired elements.

Now we can go back to C# and make new class in your project. Rename it (i use LoginPage if i will test Login process, or RegistrationPage if process is registration).

Before we start writing in new class right click on Solution, and then on Manage NuGet Package for Solution. It will open new windows, click on a browse tab, and in search field enter Appium.WebDriver (picture 12), then Selenium.WebDriver, Castle.Core, Newtonsoft.Json, DotNetSeleniumExtras.PageObjects, DotNetSeleniumExtras.WaitHelpers, NUnit, NUnit3TestAdapter (install latest versions).



Picture 12: NuGet packages

Now we can write in our class LoginPage (picture 13).

```
1 using OpenQA.Selenium;
2 using OpenQA.Selenium.Appium.Android;
3 using OpenQA.Selenium.Appium;
4 using System.Collections.Generic;
5 using SeleniumExtras.PageObjects;
6
7
8 namespace IndimaAndroid.PageObjects
9 {
10     3 references
11     class LoginPage
12     {
13         1 reference
14         public LoginPage(AndroidDriver<AppiumWebElement> androidDriver)
15         {
16             SeleniumExtras.PageObjects.PageFactory.InitElements(androidDriver, this);
17         }
18         // Ovde je bio problem, proveriti kako da se zameni PageFactory
19
20         // Locators
21
22         [FindsBy(How = How.ClassName, Using = "android.widget.ImageView")]
23         12 references | 1/12 passing
24         public IWebElement splashImage { get; set; }
25     }
26 }
```

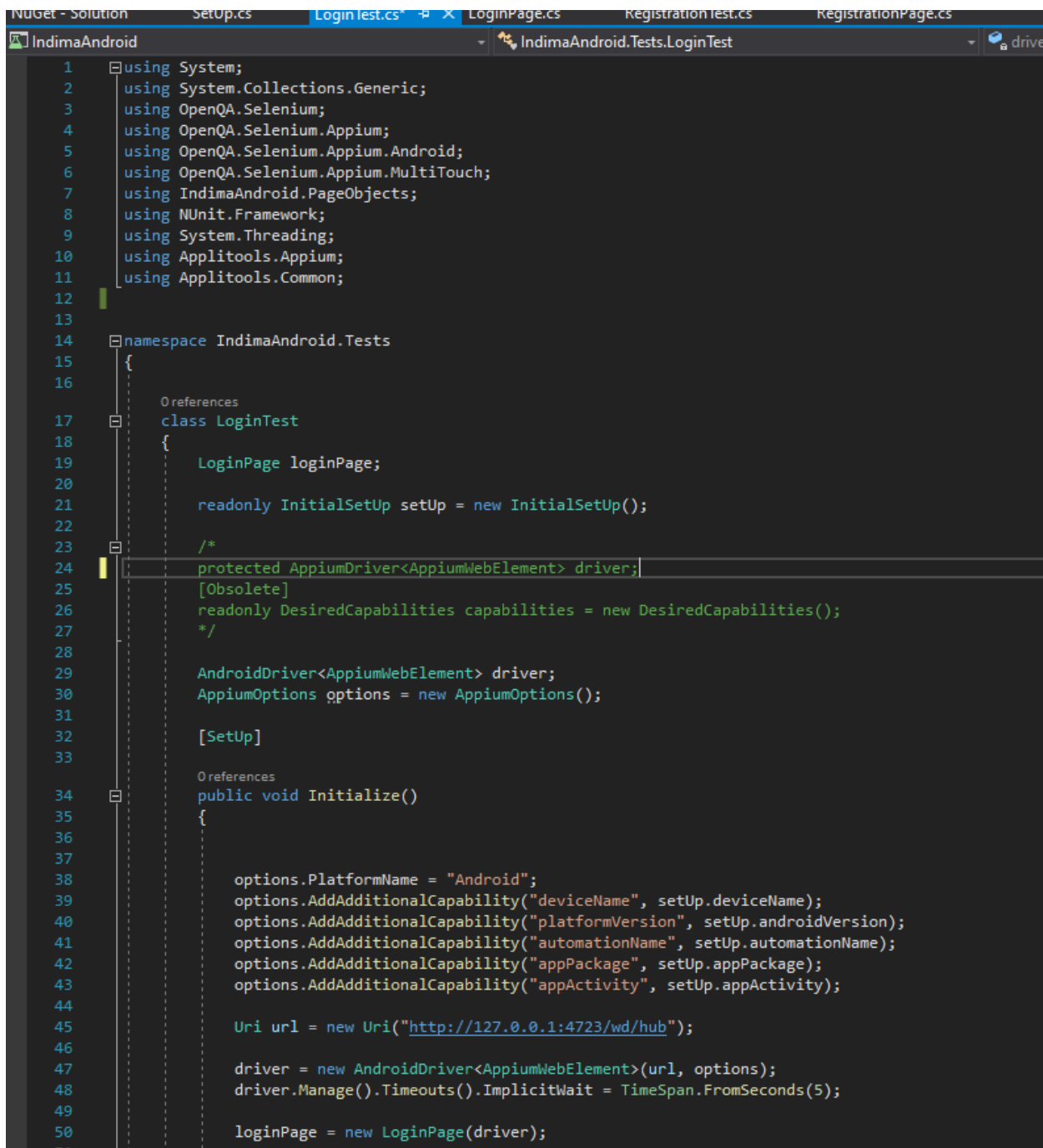
Picture 13. LoginPage code

Here we imported some libraries from NuGet packages, by statement **USING**.

To use Selenium we imported OpenQA.Selenium, to user Appium for android we import OpenQA.Selenium.Appium.Android and OpenQA.Selenium.Appium. System.Collections.Generic is here so we can use lists, and SeleniumExtras.PageObjects is here so we can create factory of objects of AppiumWebElement (consider this as object repository for AppiumWebElement).

We are finding element by statement [FindsBy(How = How.**Locator**, Using = "**ValueOfLocator**")], where locator is Id, ClassName, Xpath or any that is available, and ValueOfLocator is just value Id, ClassName, Xpath. We talked about how to find value for Locator, using Appium Inspector. Now we can create a new class, where we will write our first test.

Create new class, like we created last one, with some name, usually it is like the last one, but Page is replaced with Test. In order to make first test we need to setup configuration so program can communicate with Appium. We will do it by typing code from picture 14.



```
1  using System;
2  using System.Collections.Generic;
3  using OpenQA.Selenium;
4  using OpenQA.Selenium.Appium;
5  using OpenQA.Selenium.Appium.Android;
6  using OpenQA.Selenium.Appium.MultiTouch;
7  using IndimaAndroid.PageObjects;
8  using NUnit.Framework;
9  using System.Threading;
10 using Applitools.Appium;
11 using Applitools.Common;
12
13
14 namespace IndimaAndroid.Tests
15 {
16     0 references
17     class LoginTest
18     {
19         LoginPage loginPage;
20
21         readonly InitialSetUp setUp = new InitialSetUp();
22
23         0 references
24         [Obsolete]
25         protected AppiumDriver<AppiumWebElement> driver;
26         readonly DesiredCapabilities capabilities = new DesiredCapabilities();
27         */
28
29         AndroidDriver<AppiumWebElement> driver;
30         AppiumOptions options = new AppiumOptions();
31
32         [SetUp]
33
34         0 references
35         public void Initialize()
36         {
37
38             options.PlatformName = "Android";
39             options.AddAdditionalCapability("deviceName", setUp.deviceName);
40             options.AddAdditionalCapability("platformVersion", setUp.androidVersion);
41             options.AddAdditionalCapability("automationName", setUp.automationName);
42             options.AddAdditionalCapability("appPackage", setUp.appPackage);
43             options.AddAdditionalCapability("appActivity", setUp.appActivity);
44
45             Uri url = new Uri("http://127.0.0.1:4723/wd/hub");
46
47             driver = new AndroidDriver<AppiumWebElement>(url, options);
48             driver.Manage().Timeouts().ImplicitWait = TimeSpan.FromSeconds(5);
49
50             loginPage = new LoginPage(driver);
51         }
52     }
53 }
```

Picture 14. Setting up configuration so we can use Appium from our program

Using statements like using AppliTools, are here but will not be explained, because it is a separate tool, for more information visit AppliTools site.

Using OpenQA.Selenium.Appium.MultiTouch is imported here so we can scroll page in any direction, and later, when we get into details will be showed how to use it.

Using IndimaAndroid.PageObjects i namespace from LoginPage class,

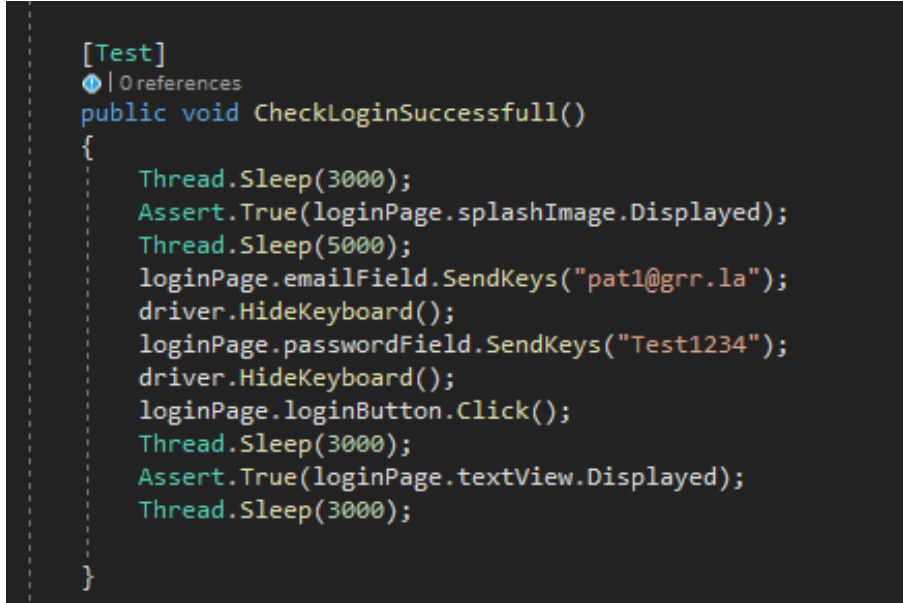
Using NUnit.Framework is package that we are using to implement tests and finally last System.Threading is here so we can delay action in our test, this will be showed soon.

Now first we need to create object of LoginPage, and that we did in line 19, after that we need to make object with what we will make connection with Appium, line 21. Next in line 29 and 30, we create object of AndroidDriver<AppiumWebDriver> with name driver, and AppiumOptions for adding desired capabilities of program. In line 32 we have special notation so program know it is SetUp function [SetUp]. In public void Initialize() we add desired capabilities to object options. If you look closer you will notice that we use variables from Setup class. Then we define Url for connection between our program and appium, this is the same port and host like that one we used to start Appium. Then we initialize driver, with that url and options (desired capabilities). Last two steps we need to do it to set driver timeout, so the driver will always wait for some defined period of time before next action (in this case it is 5 seconds), and to initialize loginPage object, with driver itself. Now we can write our first test and start it. Next chapter will explain how to write it.



## 4: Writing first test in C#

So now we finished all setup, and we can start writing test cases. To do that, we will use code from picture 15.

A screenshot of a code editor showing a C# test method. The code is written in a dark-themed editor with syntax highlighting. The method is named 'CheckLoginSuccessful()' and is decorated with the '[Test]' attribute. It includes several steps: waiting for 3000ms, asserting that 'loginPage.splashImage' is displayed, waiting for 5000ms, sending the email 'pat1@grr.la' to 'loginPage.emailField', hiding the keyboard, sending the password 'Test1234' to 'loginPage.passwordField', hiding the keyboard again, clicking 'loginPage.loginButton', waiting for 3000ms, asserting that 'loginPage.textView' is displayed, and finally waiting for 3000ms. The code is enclosed in curly braces.

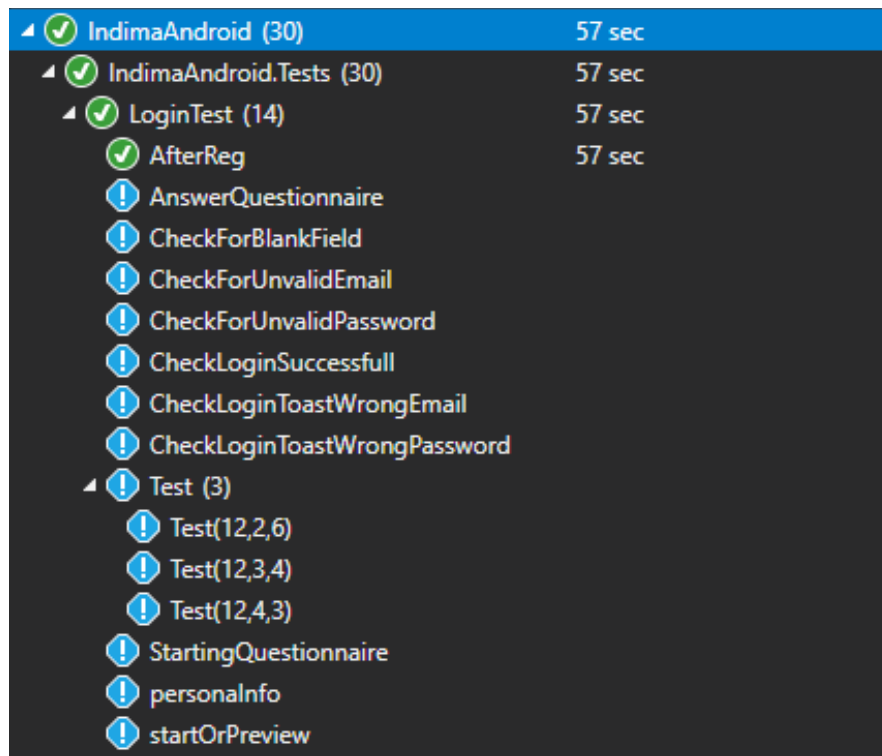
```
[Test]
0 references
public void CheckLoginSuccessful()
{
    Thread.Sleep(3000);
    Assert.True(loginPage.splashImage.Displayed);
    Thread.Sleep(5000);
    loginPage.emailField.SendKeys("pat1@grr.la");
    driver.HideKeyboard();
    loginPage.passwordField.SendKeys("Test1234");
    driver.HideKeyboard();
    loginPage.loginButton.Click();
    Thread.Sleep(3000);
    Assert.True(loginPage.textView.Displayed);
    Thread.Sleep(3000);
}
```

Picture 15: First simple test

We again use special notation [Test] to declare that function is test. Name of function need to be consistent with what test will do. In this case it is Successful Login process. All elements were added previously to class LoginPage, with FindsBy logic.

Here we use Thread.Sleep from System.Threading library so when test will start, it will wait 3 (3000 ms) second before Assert statement is used. We use Assert statement to confirm that some element is displayed on screen. Be careful with Thread.Sleep because it can make a problem where they do not exist. Later replace for this method will be shown, but for now let it be like this. Then we send some email address to loginPage object, and variable emailField with command SendKeys(). After that driver.HideKeyboard() is called to minimize keyboard, because Appium can work only with element that are visible on screen, and sometimes keyboard cover that element. So here we make test that send some text to email and password field, then click on Login button. We know that user exist in application, so we expect to see home page of application, and we Assert that.

On picture 16 user will see how to start test. Big thing to notice is that Appium must be started, before user can run any test (on the same host, and port). When user start test, the application will be started on phone, user can see all changes, and if the test is positive, there will be a green check mark next to test, if test is not started there will be a blue exclamation point mark, and if test failed, there will be a red x mark.



*Picture 16: List of all tests, with status in circles*

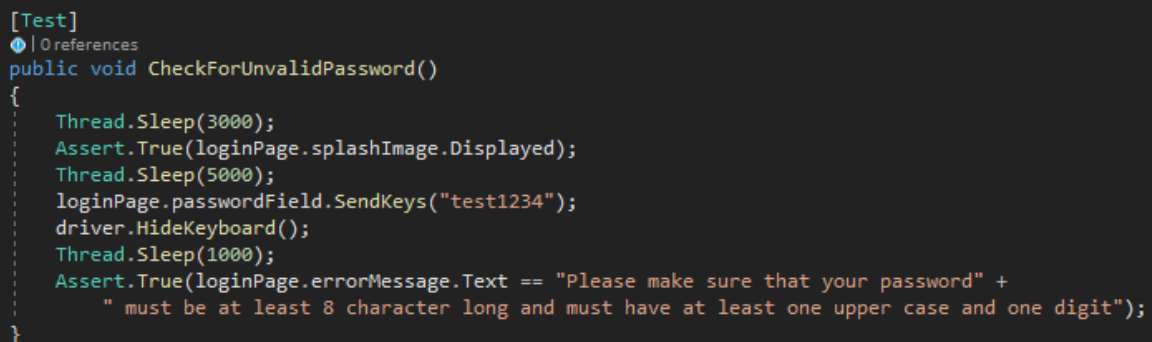
So user must right click on any test he wants to start, and then run command.

Important to do before writing any test is to come up with strategy, what test must do, how to do it, find element that will help us do it, write good and often reusable code. So for me it is an art of different skills: coding, observation, planning...

So this is just a simple test and in the next section we will talk about some more advanced skills.

## 5: Using advanced options

Now let's get started with more tests. First of all, when writing tests, we need to decide what is our goal, and how can we implement it. For example if we are trying to check some field input for invalid input, there can be different strategy for it. First off all, application can return error message at different stage of our interaction. Some application are doing inplace validation, so while user is providing input, some message will be visible (good or bad state of input). Then there is on message when user click on specific event (user click on login button and only then message will be displayed, not before that). Message can be in form of Toast message, or message under specific field. To write test we must know how application is working. When we know that we can Assert if desired state occurs. Sometimes there can be problem with getting Toast messages, so we need to find some other way, and let our mind surprise us. So let's try to write our first invalid email or password test (picture 17).



```
[Test]
public void CheckForInvalidPassword()
{
    Thread.Sleep(3000);
    Assert.True(loginPage.splashImage.Displayed);
    Thread.Sleep(5000);
    loginPage.passwordField.SendKeys("test1234");
    driver.HideKeyboard();
    Thread.Sleep(1000);
    Assert.True(loginPage.errorMessage.Text == "Please make sure that your password" +
        " must be at least 8 character long and must have at least one upper case and one digit");
}
```

Picture 17: Test for invalid Password

In this test we know that password validation is done in place, and what will validation message display. So we can find exactly that element of validation message, and assert it. If Assert return true, test will exit and state will be successful.

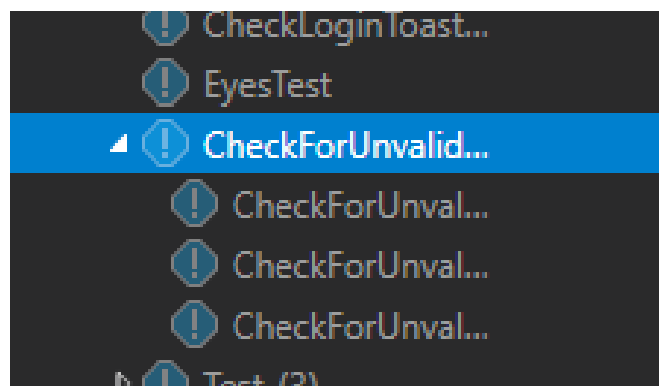
Now if we need more test, for invalid password, we can do it in two ways. First is to create [TestCase] strategy, showed on picture 18. Second is to read invalid password from xml file, and that will showed later.

```

[TestCase("test1234")]
[TestCase("Test123")]
[TestCase("Test 1234")]
| 0 references
public void CheckForUnvalidPassword(string password)
{
    Thread.Sleep(3000);
    Assert.True(loginPage.splashImage.Displayed);
    Thread.Sleep(5000);
    loginPage.passwordField.SendKeys(password);
    driver.HideKeyboard();
    Thread.Sleep(1000);
    Assert.True(loginPage.errorMessage.Text == "Please make sure that your password" +
        " must be at least 8 character long and must have at least one upper case and one digit");
}

```

Picture 18: Providing test with multiple test scenarios



Picture 19: Tests grouped under one main test (using [TestCase()] notation)

On picture 19 it is showed how it looks like on test explorer, when we use [TestCase] notation where we have multiple tests under one. Also we can run it separately or all together. Important to notice is that message that you use to assert, must match perfectly with message that will be displayed when input is invalid. That means that we need to take care if there is some character that must be escaped in string (/, or "...). To escape characters we use \. For example in next message we escaped two times " character: **"//android.view.View[@content-desc=\"22 October 1986\"]"**. One was before number 22, and one was after =. So be carefull about this. Also watch that your string do not contain additional white spaces, because it will give error, and you will lose half hour to figure it out. Trust me it is better to focus for one minute, then to lose half hour trying to figure it out. I think Click(), SendKey(), Enabled, Displayed, Clear() method are easy to understand, so let's talk about Assert. We can Assert if something is True, to confirm that element exist, or we can Assert if something is not displayed on screen. For this we can use False method or we can put "!" right after (), something like this **Assert.True(!element.Displayed())**. This will give error if element is displayed on screen. We also need to include one special method at the end of program, so Appium can close connection when test is done. That method is TearDown, and it is displayed on picture 20.

```

[TearDown]
0 references
public void TearDown()
{
    driverReg.Quit();
}

```

Picture 20: TearDown method

We do not need to call this method, program is doing it on its own, but we must implement it.

Next thing we can talk about is how to replace `Thread.Sleep()` method with some more elegant solutions. We used **`driver.Manage().Timeouts().ImplicitWait = TimeSpan.FromSeconds(5)`** method in our Initialize method. This method will wait maximum for 5 second in this case, can be more if we define it (`FromSeconds(x)`, where `x` is number of seconds), for some action to happens. So if we are waiting to some element to be displayed, Appium can do it for us if we use this method. This method is generally best solution, but there is some others. For example, if we define our wait interval to be 10 second, because response will be in that interval for 95% of our application, what to with 5% of it. To define 30 seconds? Or to find other was? Well here is good to know that we can define new wait interval, that we can use in case like this. To do this we must define time interval we want to wait like this: **`public System.TimeSpan time = new TimeSpan(300000000)`**, where this large number here is actually number of ticks. Let's take that that number is about 30 seconds, +- one or two seconds. So now we can define our wait variable like this:

**`WebDriverWait wait = new WebDriverWait(driverReg, timeReg)`**, where `timeReg` is time, and `driverReg` is driver we initialize earlier, and that we use for making connection to Appium. Now we must use **`SeleniumExtras.WaitHelpers`** package (download it from NuGet, and include it), and use next function:

**`wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.ElementIsVisible(By.Id("com.indima.android:id/tv_register_here")))`**, where we just want to wait till element that have ID = **`"com.indima.android:id/tv_register_here"`** is displayed. So now we can combine this two ways, and create really reliable tests. `Thread.Sleep()` have problem that it will wait for defined moment of time, before we can go further. But this new methods have defined maximum time to wait, so if they find element faster, they will go further, no matter how long timer is.

Now when we can make reliable test, next thing to talk about is using list to store elements in it. If we have multiple elements of same Id, but different index, we can put that elements in list, and later access then by index notation.

```
[FindsBy(How = How.ClassName, Using = "android.view.View")]
1 reference | 0/1 passing
public IList<IWebElement> date1 { get; set; }
// Element[0] = 1...
```

Picture 21. Making list of elements with same id, class or some other strategy

Now we can call list from our test like this: **registrationPage.date1[15 - 1].Click()**. Here we use [15 - 1], because first element in list is [0]. So if we want date to be 15, we need to use [14]. Upper notation ([15 - 1]) is here just for clarity. Good thing about list is that if list do not find element, it will return empty list. That allow us to control flow of our test, without being forced to use Assert and to fail test. Example: if we have test, where on some action there can be one of two element, but we don't know which one, we can define them as list, and if list is empty, we then know that that element did not occur (picture 22).

```
List<IWebElement> elementTask = new List<IWebElement>();
elementTask.AddRange(driver.FindElements(By.Id("com.indima.android:id/iv_task_status")));

List<IWebElement> elementList1 = new List<IWebElement>();
elementList1.AddRange(driver.FindElements(By.Id("com.indima.android:id/cv_questionnaire")));

if (elementTask.Count > 0)
{
    var output = @"C:\Users\kamenarac\Desktop\XML\TTTTT.txt";
    System.IO.File.AppendAllText(output, ":" + elementTask.Count + "\n");

    //If the count is greater than 0 your element exists.
    elementTask[0].Click();
    //On account doc3, that task is finished, so clicking on it will not open anything

    Screenshot screenshot1 = driver.GetScreenshot();

    Screenshots(screenshot1, @"C:\Users\kamenarac\Desktop\XML\task.jpg");

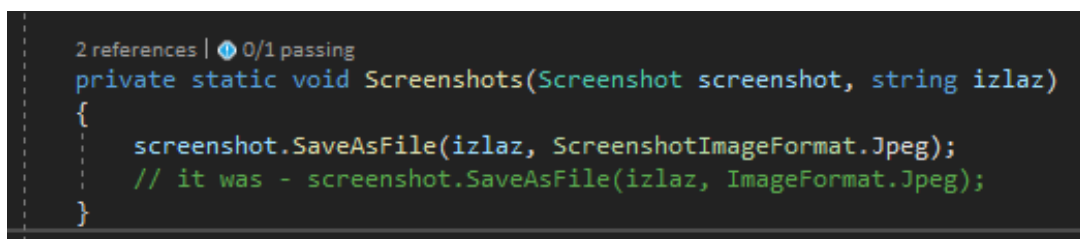
    Thread.Sleep(3000);
}
else
{
    elementList1[0].Click();
    Thread.Sleep(3000);

    Screenshots(driver.GetScreenshot(), @"C:\Users\kamenarac\Desktop\XML\123.jpg");
    Thread.Sleep(5000);
}
```

Picture 22: Using list of elements

Here code is a bit messy, but point is that we define two list, one contain one elements, and second contain some different one. With IF statement we check if list have elements in it, then we click on first one in list, and take screenshot (will be explained soon). If first list is empty, we know our program must have displayed elements from second list, so we just click on first one in second list and take screenshot. **Controlling flow, simple as that.** Here user just must know how application will behave, and make test accordingly to that behavior.

Now we can talk about taking screenshot of our application. It can be very powerful tool, because we can generate screenshots of important states in application. Example: If we have decision to make we can take screenshot before that decision, and after it, so we can verify that decision is right.



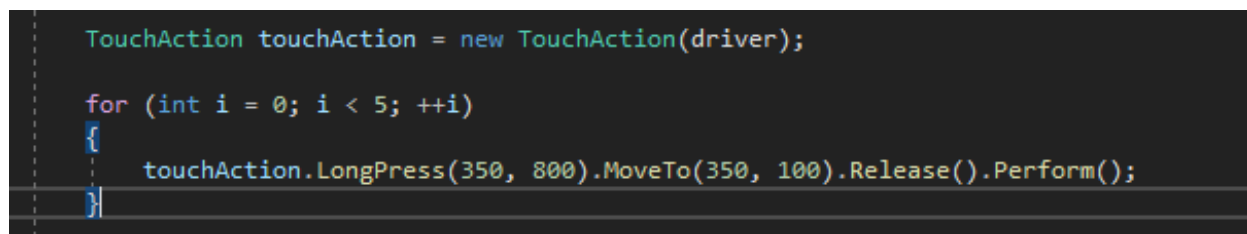
```
2 references | 0/1 passing
private static void Screenshots(Screenshot screenshot, string izlaz)
{
    screenshot.SaveAsFile(izlaz, ScreenshotImageFormat.Jpeg);
    // it was - screenshot.SaveAsFile(izlaz, ImageFormat.Jpeg);
}
```

Picture 23: Define Screenshots method

Here we define screenshot method, that will take screenshot of screen, and path of where we want to save that screenshot. Inside method we call SaveAsFile method that take that path, and format of image. To take screenshot in our test we call define: **Screenshot screenshot1 = driver.GetScreenshot()**, and then pass screenshot1 to Screenshots method, and as second parameter we provide path. Easiest way to provide path is like this:

**@ "C:\Users\kamenarac\Desktop\XML\task.jpg"**, where @ will help us, so we do not need to use double \\ because that is not portable, and at then end of path is file extension (here extension is **.jpg**).

Let's talk now about how to scroll of swipe screen. It is really simple (picture 24).



```
TouchAction touchAction = new TouchAction(driver);

for (int i = 0; i < 5; ++i)
{
    touchAction.LongPress(350, 800).MoveTo(350, 100).Release().Perform();
}
```

Picture 24: Scroll option

Here we must include next library: **OpenQA.Selenium.Appium.MultiTouch**. We define and initialize TouchAction touchAction = new TouchAction(driver). If we need to scroll multiple time, we do it by using for loop, and define how much time we need it. Then we

use LongPress method, that takes x and y coordinates (width, height), where we need to press, then we use MoveTo method that also take x and y coordinates. After that we use Release, and finally we use Perform. Simple as that.

Big thing to notice is that if you scroll down you must go from big number to small number, because 0 0 coordinates are in top left corner. It will take you time to remember what coordinates to use first, to scroll in direction to want, because it is a bit counterintuitive.

To control flow of test, user can also reset application. It can be done with driver.ResetApp() method. This way user can start application again at any point he desire.