

# HCP

How the unity extension to appium works, and how to write tests

## Table of Contents

[HCP Architecture](#)

[Code Structure](#)

[HCP Server Implementation](#)

[Appium Server](#)

[Installation](#)

[Building Changes / Updates](#)

[Appium Client](#)

[Installation](#)

[Building Changes / Updates](#)

[Running Tests](#)

[Android](#)

[iOS](#)

[Building Tests](#)

[Appium-dot-app Settings](#)

[General \(Required for all platforms\)](#)

[Android](#)

[iOS](#)

# HCP Architecture

HCP builds on Appium, and Appium builds on Selenium. Selenium provides what is called a WebDriver. Its purpose is to automate the testing of web applications build in javascript and HTML. It provides a RESTful API to GET/POST data, typically returning results that are used to validate the actions results.

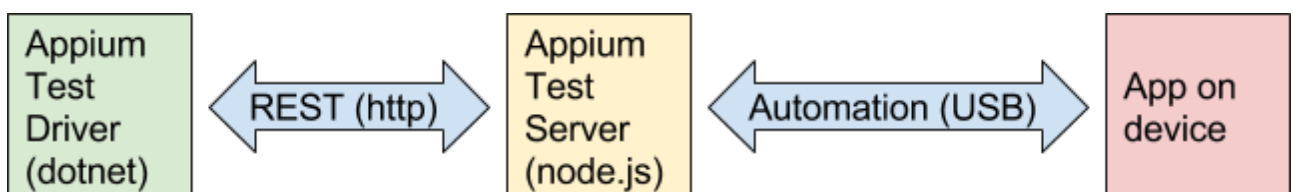
The Appium team saw the WebDriver as a useful framework to build enterprise App testing upon. Appium is very modular, and we currently have forked the following projects:

- Appium-android-bootstrap
- Appium-android-driver
- Appium-base-driver
- Appium-dot-app
- Appium-dot-exe
- Appium-dotnet-driver
- Appium-ios-driver
- Selenium

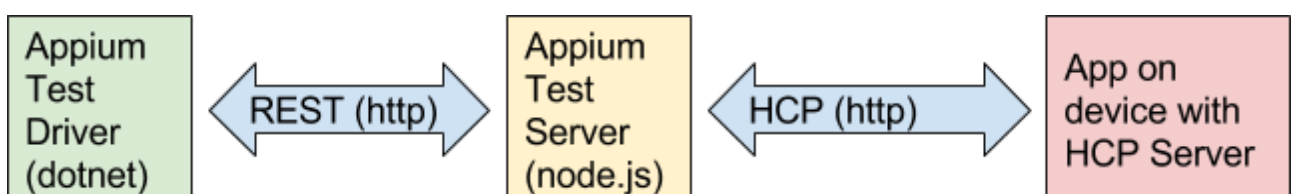
New dependencies are comprised of the following

- Appium-hcp
- Appium-hcp-driver
- Requisition

Although appium uses *driver* terminology. The **appium-dotnet-driver** is meant to write tests and it *drives* the appium driver (whether it being android or iOS). Building on this, **appium-android-driver** and **appium-ios-driver** drive automation frameworks provided by their respective platform developers. You cannot use them directly, but instead issue REST requests from a test driver. Thus, the flow of communication looks like this with vanilla Appium.



HCP communicates with any app that supports HCP. It works alongside automation through USB by sending REST commands over http between the appium test server and the app on device.



# Code Structure

The HCP implementation attempts to modify as little Appium code as possible. To facilitate this, HCP functions by inserting itself between Appium base classes and their Android/iOS implementations. This means that there isn't a separate implementation for both platforms, and they instead derive from a common implementation of the Hutch common protocol.

These changes to appium code is summarized by:

- Appium-base-driver - Adjusted the implementation so that REST endpoints can be appended to when constructing a driver. This is required for us to insert our HCP specific endpoints as appium didn't expect to inherit driver functionality.
- Appium-android-driver - We have this derive from a new HCP driver, rather than base. This means that every Android driver can also send HCP formatted commands over http to an android device.
- Appium-ios-driver - Same as above
- Appium-hcp-driver - This is a hcp driver implementation that receives requests from test cases, communicates with a http server in Unity, and responds to the test case appropriately. It uses the android driver as a base
- Appium-hcp - This is the module that actually contains the HTTPRequest code and is used by Appium-hcp-driver
- Appium-dotnet-driver - As with the android and ios driver, we insert the HCP into the dotnet device drivers. This allows commands to be directed to either the device or to the HCP server running on the device.
- Appium-dot-app - Somewhat hacky changes to force the app to use our appium implementation, support HCP, and record HCP compatible tests. This is only done for C# output.
- Selenium (dotnet)- Changed the base driver to set its Execute method to virtual. This allows us to override it and redirect HCP commands to HCP endpoints without major code duplication. This is likewise try for the base web element type.
- Selenim-objectice-c - A submobile of appium-dot-app. Same changes as above, but in a more hacky manner as its isolated to just the OSX app.

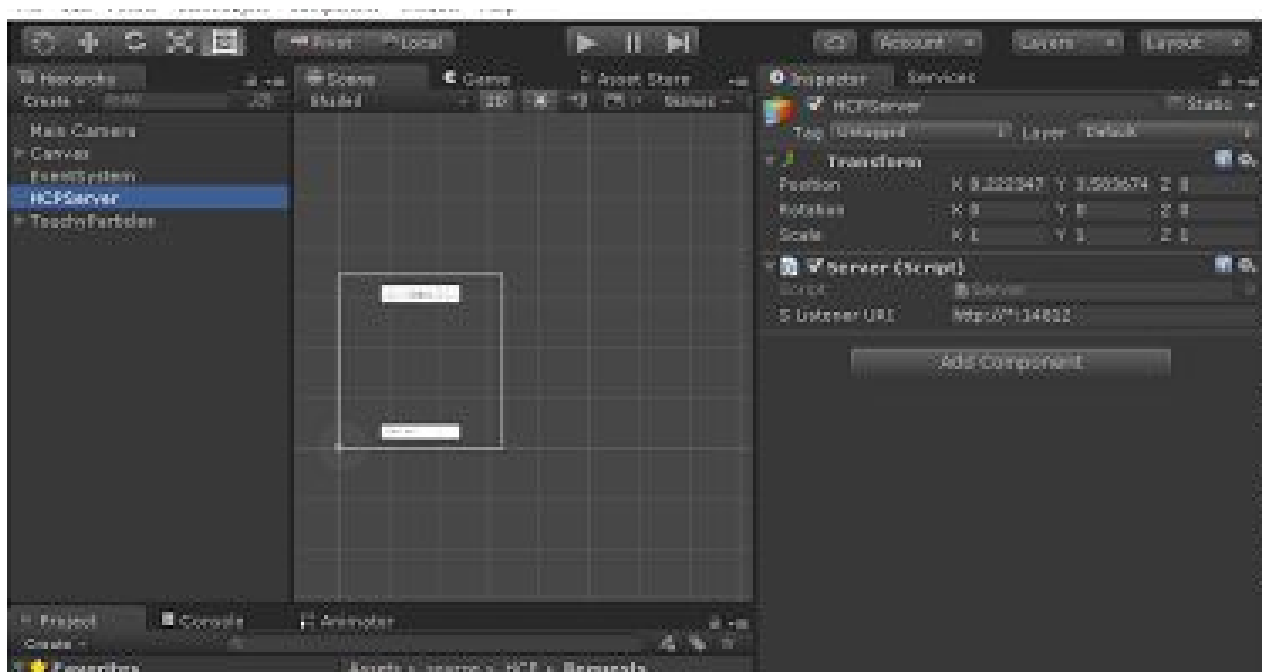
## HCP Server Implementation

An implementation of the HCP Server is provided for Unity. It is located in the Appium-Unity project. It listens for HCP requests and does its best to return the data it requires. The server must be added to an active gameobject, and all elements in the scene to be visible by HCP must have an HCP.Element component added to them. Currently, the system assumes that there is only one HCP.Element per gameobject, but this can be expanded to support multiple HCP.Element types (HCP.TextElement, HCP.ButtonElement, etc).

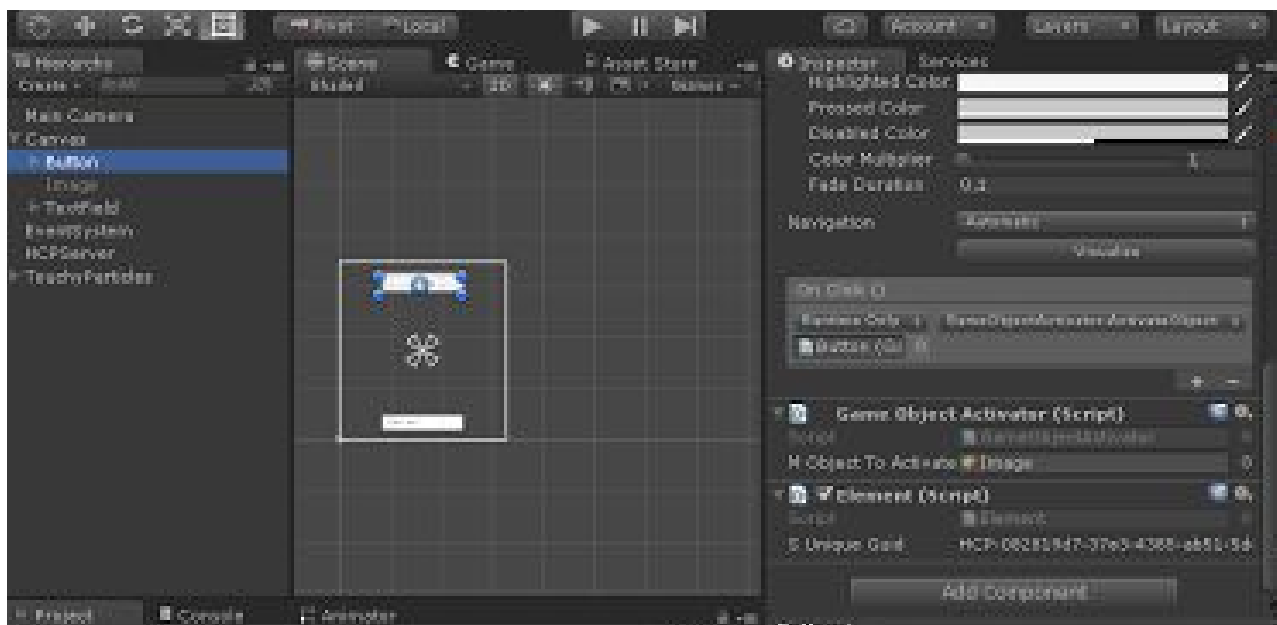
The reason for Elements is so that we can have a serializable id so that tests written looking for this ID will not fail in future builds. The id also self-describes as a HCP element which helps to figure out how and where to direct actions on buttons and text fields for client drivers

## Unity Setup

- Server Component: Use [http://\\*:14812](http://*:14812) as the listener URI. The \* will accept connections from any machine, and the port 14812 is the current default. You can certainly change these values, but there is currently no reason to.



- Element Component: You can either add this manually, or write an editor script to add them. If you do not add them, they will be added by the server at runtime. These runtime element id's will not be serialized, and tests which find by their id will fail in future runs.



# Appium Server

## Installation

The following installation instructions assume an OSX operating system. The project will still compile and run on Windows systems, but you will not be able to test on iOS devices.

- Retrieve the repo and submodules
  - <https://github.com/GoldenBear/Selenium-Unity-Driver>
- Ensure you have a recent version of node/npm installed
  - <https://nodejs.org/dist/v6.3.1/node-v6.3.1-x64.msi>
- Ensure you have a recent a working JDK
  - <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
- Bootstrap your system using
  - `$ sh bootstrap.sh`
- Use the supplied compile.sh script to compile the project
  - `$ sh compile.sh`
- Use the supplied run.sh script to start the server (should you wish to run the server in standalone mode)
  - `$ sh run.sh`

## Building Changes / Updates

If you wish to take the most recent HCP from the git repo, use the following steps

- Use the supplied update.sh script to update the project
  - `$ sh update.sh`
- Use the supplied clean.sh script to get ready to rebuild
  - `$ sh clean.sh`
- Use the supplied compile.sh script to build changes
  - `$ sh compile.sh`

## Debugging

If you wish to debug, you will need Chrome. You can easily debug:

- Use the supplied debug.sh script to build changes
  - `$ sh debug.sh`

# Appium Client

## Installation

The following installation instructions assume an OSX operating system. The project will still compile and run on Windows systems, but you will not be able to test on iOS devices.

- Retrieve the repo and submodules
  - <https://github.com/GoldenBear/Selenium-Unity-Driver>
- Use the supplied bootstrap\_app.sh script to prepare the xcode project
  - `$ sh bootstrap_app.sh`
- Open the `appium-dot-app/Appium.xcworkspace` project file
- Build the project (Xcode required)

## Building Changes / Updates

If you wish to take the most recent HCP from the git repo, use the following steps

- Use the supplied update.sh script to update the project
  - `$ sh update.sh`
- Open the `appium-dot-app/Appium.xcworkspace` project file
- Build the project (Xcode required)

# Running Tests

## General Setup

- The appium server must be running. Preferably on an OSX system so that you can launch iOS devices. Note that tests can be run on another computer. On my dev system, I have the appium server running on OSX and the tests run from Visual Studio in Windows.
- The tests must know where the appium server is, so you supply the URI of the appium server when constructing the driver. More details in the Building Tests section

## Android Setup

- Requires Android Studio
- Sample Capabilities List:
  - App = "mygame.apk"
  - AutoWebView = false
  - AutomationName = "anything"
  - BrowserName = ""; (Leave empty otherwise you test on browsers)
  - DeviceName = "anything";
  - FwkVersion = "1.0"; (Not really needed)
  - Platform = TestCapabilities.DevicePlatform.Android;
  - PlatformVersion = String.Empty; (Leave empty to attach to any Android version)
  - SupportsHCP = true;
  - HCPHost = "<http://127.0.0.1>" (Actual IP is not needed as ADB will forward)
  - HCPPort = 14812
- Generally hassle free from this point out.

## iOS Setup

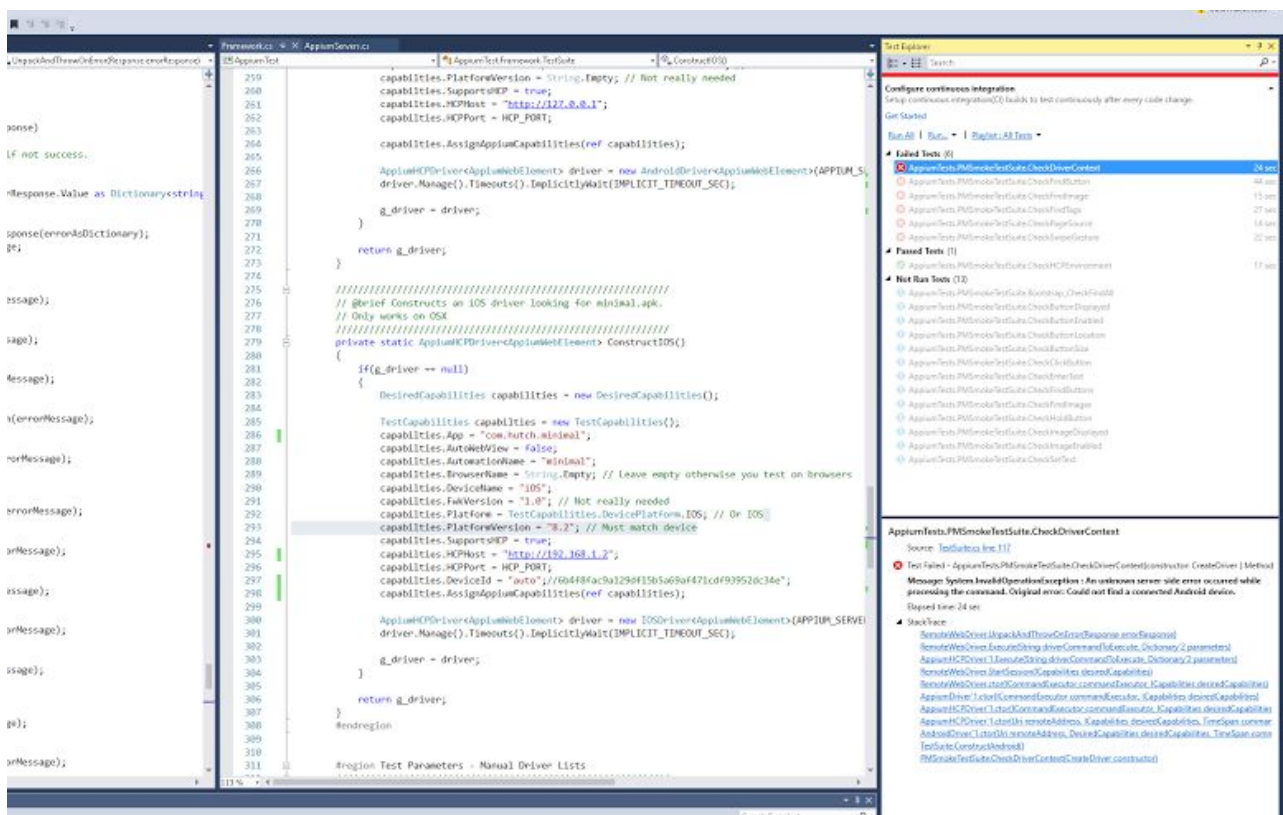
- Requires Xcode with working Instruments
- UIAutomation must be enabled on your iOS device. It is under Developer settings.
- Sample Capabilities List:
  - App = "mygame.ipa" (If the server stalls, preload and use bundle id, eg: com.hutch.game)
  - AutoWebView = false
  - AutomationName = "anything"
  - BrowserName = ""; (Leave empty otherwise you test on browsers)
  - DeviceName = "anything";
  - FwkVersion = "1.0"; (Not really needed)
  - Platform = TestCapabilities.DevicePlatform.Android;
  - PlatformVersion = "8.2"; (Must match OS version)
  - UDID = "auto" (or the actual udid to attach to a specific device, required if you want to have a usb hub with many devices)
  - SupportsHCP = true;

- HCPHost = "<http://192.168.1.12>" (Actual IP is needed until some auto forwarding is figured out)
- HCPPort = 14812
- iOS on appium is finicky. Troubleshooting tips below:
  - If the appium server hangs on waiting for the app to install, preload the app and set app (in capabilities) to the bundle id instead of the ipa filename.
  - If the appium server hangs on waiting for the app to install, you may wish to start up an appium server with the app being specified, rather than having the test suite set that up. See appium docs for reference on how to do this if you prefer an alternate workflow
  - If the app crashes on start due to appium, open Instruments from xCode (xCode -> Open Developer Tool -> Instruments) and try to start the app on the ios device. You will most likely get a legible error and be able to fix. It usually has to do with the following conditions not being met:
    - App is release and not debug mode
    - App doesn't have a proper provisioning profile

Really hacky workarounds include building the app from xCode and directly launching to the connected iOS device. This shouldn't be necessary, but will get you through this error if you are just getting things set up.

## Running from Visual Studio

The visual studio "Test Explorer" window will show all the tests available in your solution. This is the simplest way to build and run tests when they are initially written.





# Running from Console

Tests can be run from the command line using the xunit console runner. This is the preferred method for actual testing and supports output formatting used to review results, which is outlined later in this document.

## Sample Command:

h:\Projects\Appium-Unity\Project\hutch\AppiumTest>packages\xunit.runner.console.2.1.0\tools\xunit.console.exe bin\Debug\AppiumTest.dll -html results.html

```
Visual Studio Command Prompt (2016)
h:\Projects\Appium-Unity\Project\hutch\AppiumTest\packages\xunit.runner.console.2.1.0\tools\xunit.console.exe bin\Debug\AppiumTest.dll -html results.html
Starting: AppiumTest
Discovering: AppiumTest
Discovered: AppiumTest
Starting: AppiumTest
AppiumTests.FMSmokeTestSuite.CheckDriverContext(CreateDriver [Method = OpenQA.Selenium.Appium.HCP.AppiumHCPDriver`1[[OpenQA.Selenium.Appium.AppiumWebElement] ConstructIOS(), Target = null]] (FAIL)
OpenQA.Selenium.WebDriverException : Unexpected error. [Fiddler] The connection to '192.168.1.13' failed. <br />Error: ConnectionRefused (0x274d). <br />System.Net.Sockets.SocketException No connection could be made because the target machine actively refused it 192.168.1.13:4723
Stack Trace:
H:\Projects\Appium-Unity\Project\hutch\Selenium\dotnet\src\webdriver\Remote\RemoteWebDriver.cs (1317,0): at OpenQA.Selenium.Remote.RemoteWebDriver.UnpackAndThrowOnError(Response errorResponse)
H:\Projects\Appium-Unity\Project\hutch\Selenium\dotnet\src\webdriver\Remote\RemoteWebDriver.cs (1070,0): at OpenQA.Selenium.Remote.RemoteWebDriver.Execute(String driverCommandToExecute, Dictionary`2 parameters)
H:\Projects\Appium-Unity\Project\hutch\Appium-DotNet-Driver\Appium-DotNet-Driver\Appium\HCP\AppiumHCPDriver.cs (136,0): at OpenQA.Selenium.Appium.HCP.AppiumHCPDriver`1.Execute(String driverCommandToExecute, Dictionary`2 parameters)
H:\Projects\Appium-Unity\Project\hutch\Selenium\dotnet\src\webdriver\Remote\RemoteWebDriver.cs (1038,0): at OpenQA.Selenium.Remote.RemoteWebDriver.StartSession(ICapabilities desiredCapabilities)
H:\Projects\Appium-Unity\Project\hutch\Selenium\dotnet\src\webdriver\Remote\RemoteWebDriver.cs (118,0): at OpenQA.Selenium.Remote.RemoteWebDriver..ctor(ICommandExecutor commandExecutor, ICapabilities desiredCapabilities)
H:\Projects\Appium-Unity\Project\hutch\Appium-DotNet-Driver\Appium-DotNet-Driver\Appium\AppiumDriver.cs (46,0): at OpenQA.Selenium.Appium.AppiumDriver`1..ctor(ICommandExecutor commandExecutor, ICapabilities desiredCapabilities)
H:\Projects\Appium-Unity\Project\hutch\Appium-DotNet-Driver\Appium-DotNet-Driver\Appium\HCP\AppiumHCPDriver.cs (20,0): at OpenQA.Selenium.Appium.HCP.AppiumHCPDriver`1..ctor(ICommandExecutor commandExecutor, ICapabilities desiredCapabilities, TimeSpan commandTimeout)
H:\Projects\Appium-Unity\Project\hutch\Appium-DotNet-Driver\Appium-DotNet-Driver\Appium\HCP\AppiumHCPDriver.cs (92,0): at OpenQA.Selenium.Appium.HCP.AppiumHCPDriver`1..ctor(Uri remoteAddress, ICapabilities desiredCapabilities, TimeSpan commandTimeout)
H:\Projects\Appium-Unity\Project\hutch\Appium-DotNet-Driver\Appium-DotNet-Driver\Appium\IOS\IOSDriver.cs (107,0): at OpenQA.Selenium.Appium.IOS.IOSDriver`1..ctor(Uri remoteAddress, DesiredCapabilities desiredCapabilities, TimeSpan commandTimeout)
Framework.cs (301,0): at AppiumTest.Framework.TestSuite.ConstructIOS()
Tests\TestSuite.cs (146,0): at AppiumTest.FMSmokeTestSuite.CheckDriverContext(CreateDriver constructor)
AppiumTests.FMSmokeTestSuite.CheckDriverContext(CreateDriver [Method = OpenQA.Selenium.Appium.HCP.AppiumHCPDriver`1[[OpenQA.Selenium.Appium.AppiumWebElement] ConstructIOS(), Target = null]] (FAIL)
OpenQA.Selenium.WebDriverException : Unexpected error. [Fiddler] The connection to '192.168.1.13' failed. <br />Error: ConnectionRefused (0x274d). <br />System.Net.Sockets.SocketException No connection could be made because the target machine actively refused it 192.168.1.13:4723
Stack Trace:
H:\Projects\Appium-Unity\Project\hutch\Selenium\dotnet\src\webdriver\Remote\RemoteWebDriver.cs (1317,0): at OpenQA.Selenium.Remote.RemoteWebDriver.UnpackAndThrowOnError(Response errorResponse)
H:\Projects\Appium-Unity\Project\hutch\Selenium\dotnet\src\webdriver\Remote\RemoteWebDriver.cs (1070,0): at OpenQA.Selenium.Remote.RemoteWebDriver.Execute(String driverCommandToExecute, Dictionary`2 parameters)
H:\Projects\Appium-Unity\Project\hutch\Appium-DotNet-Driver\Appium-DotNet-Driver\Appium\HCP\AppiumHCPDriver.cs (136,0): at OpenQA.Selenium.Appium.HCP.AppiumHCPDriver`1.Execute(String driverCommandToExecute, Dictionary`2 parameters)
H:\Projects\Appium-Unity\Project\hutch\Selenium\dotnet\src\webdriver\Remote\RemoteWebDriver.cs (1038,0): at OpenQA.Selenium.Remote.RemoteWebDriver.StartSession(ICapabilities desiredCapabilities)
H:\Projects\Appium-Unity\Project\hutch\Selenium\dotnet\src\webdriver\Remote\RemoteWebDriver.cs (118,0): at OpenQA.Selenium.Remote.RemoteWebDriver..ctor(ICommandExecutor commandExecutor, ICapabilities desiredCapabilities)
H:\Projects\Appium-Unity\Project\hutch\Appium-DotNet-Driver\Appium-DotNet-Driver\Appium\AppiumDriver.cs (46,0): at OpenQA.Selenium.Appium.AppiumDriver`1..ctor(ICommandExecutor commandExecutor, ICapabilities desiredCapabilities)
H:\Projects\Appium-Unity\Project\hutch\Appium-DotNet-Driver\Appium-DotNet-Driver\Appium\HCP\AppiumHCPDriver.cs (20,0): at OpenQA.Selenium.Appium.HCP.AppiumHCPDriver`1..ctor(ICommandExecutor commandExecutor, ICapabilities desiredCapabilities, TimeSpan commandTimeout)
H:\Projects\Appium-Unity\Project\hutch\Appium-DotNet-Driver\Appium-DotNet-Driver\Appium\HCP\AppiumHCPDriver.cs (92,0): at OpenQA.Selenium.Appium.HCP.AppiumHCPDriver`1..ctor(Uri remoteAddress, ICapabilities desiredCapabilities, TimeSpan commandTimeout)
H:\Projects\Appium-Unity\Project\hutch\Appium-DotNet-Driver\Appium-DotNet-Driver\Appium\IOS\IOSDriver.cs (107,0): at OpenQA.Selenium.Appium.IOS.IOSDriver`1..ctor(Uri remoteAddress, DesiredCapabilities desiredCapabilities, TimeSpan commandTimeout)
Framework.cs (301,0): at AppiumTest.Framework.TestSuite.ConstructIOS()
Tests\TestSuite.cs (146,0): at AppiumTest.FMSmokeTestSuite.CheckDriverContext(CreateDriver constructor)
AppiumTests.FMSmokeTestSuite.CheckDriverContext(CreateDriver [Method = OpenQA.Selenium.Appium.HCP.AppiumHCPDriver`1[[OpenQA.Selenium.Appium.AppiumWebElement] ConstructIOS(), Target = null]] (FAIL)
OpenQA.Selenium.WebDriverException : Unexpected error. [Fiddler] The connection to '192.168.1.13' failed. <br />Error: ConnectionRefused (0x274d). <br />System.Net.Sockets.SocketException No connection could be made because the target machine actively refused it 192.168.1.13:4723
Stack Trace:
H:\Projects\Appium-Unity\Project\hutch\Selenium\dotnet\src\webdriver\Remote\RemoteWebDriver.cs (1317,0): at OpenQA.Selenium.Remote.RemoteWebDriver.UnpackAndThrowOnError(Response errorResponse)
H:\Projects\Appium-Unity\Project\hutch\Selenium\dotnet\src\webdriver\Remote\RemoteWebDriver.cs (1070,0): at OpenQA.Selenium.Remote.RemoteWebDriver.Execute(String driverCommandToExecute, Dictionary`2 parameters)
```

# Looking at the results

Inspecting the results of your tests should be self explanatory from within Visual Studio. Here, there test output can be seen from within the “Test Explorer” window as shown below

Test Explorer

Configure continuous integration  
Setup continuous integration (C) builds to test continuously after every code change.  
Get Started

Run All | Run... | Skip All Tests

Failed Tests (0)

AppiumTests.FMSmokeTestSuite.CheckDriverContext 24 sec

AppiumTests.FMSmokeTestSuite.CheckButton 44 sec

AppiumTests.FMSmokeTestSuite.CheckFindImage 15 sec

AppiumTests.FMSmokeTestSuite.CheckFindTags 27 sec

AppiumTests.FMSmokeTestSuite.CheckPageSource 14 sec

AppiumTests.FMSmokeTestSuite.CheckImageDisplayed 22 sec

Passed Tests (1)

AppiumTests.FMSmokeTestSuite.CheckDriverContext 17 sec

Not Run Tests (12)

AppiumTests.FMSmokeTestSuite.ButtonUp, CheckFind

AppiumTests.FMSmokeTestSuite.CheckButton Displayed

AppiumTests.FMSmokeTestSuite.CheckButton Disabled

AppiumTests.FMSmokeTestSuite.CheckButton Location

AppiumTests.FMSmokeTestSuite.CheckButton Size

AppiumTests.FMSmokeTestSuite.CheckClickButton

AppiumTests.FMSmokeTestSuite.CheckClickText

AppiumTests.FMSmokeTestSuite.CheckFindImage

AppiumTests.FMSmokeTestSuite.CheckFindImage

AppiumTests.FMSmokeTestSuite.CheckFindImage

AppiumTests.FMSmokeTestSuite.CheckImageDisplayed

AppiumTests.FMSmokeTestSuite.CheckImageNotDisplayed

AppiumTests.FMSmokeTestSuite.CheckSelfTest

AppiumTests.FMSmokeTestSuite.CheckDriverContext

Source: TestSuite line 111

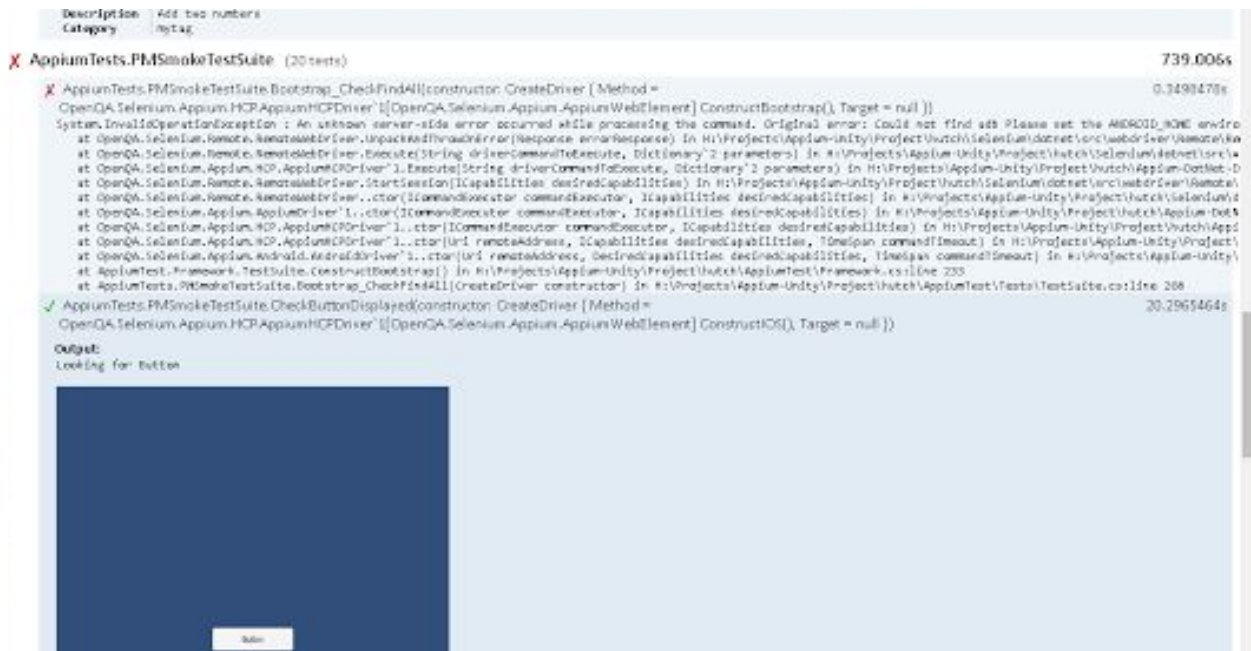
Test Failed - AppiumTests.FMSmokeTestSuite.CheckDriverContext(CreateDriver [Method = OpenQA.Selenium.Appium.HCP.AppiumHCPDriver`1[[OpenQA.Selenium.Appium.AppiumWebElement] ConstructIOS(), Target = null]] (FAIL)

Message: System.InvalidOperationException : An unknown server side error occurred while processing the command. Original error: Could not find a connected Android device.

Elapsed time: 24 sec

Stack Trace:  
RemoteWebDriver.UnpackAndThrowOnError(Response errorResponse)  
RemoteWebDriver.Execute(String driverCommandToExecute, Dictionary`2 parameters)  
AppiumHCPDriver`1.Execute(String driverCommandToExecute, Dictionary`2 parameters)  
RemoteWebDriver.StartSession(ICapabilities desiredCapabilities)  
RemoteWebDriver..ctor(ICommandExecutor commandExecutor, ICapabilities desiredCapabilities)  
AppiumDriver`1..ctor(ICommandExecutor commandExecutor, ICapabilities desiredCapabilities)  
AppiumHCPDriver`1..ctor(Uri remoteAddress, ICapabilities desiredCapabilities, TimeSpan commandTimeout)  
AppiumHCPDriver`1..ctor(Uri remoteAddress, DesiredCapabilities desiredCapabilities, TimeSpan commandTimeout)  
TestSuite.ConstructIOS()  
FMSmokeTestSuite.CheckDriverContext(CreateDriver constructor)

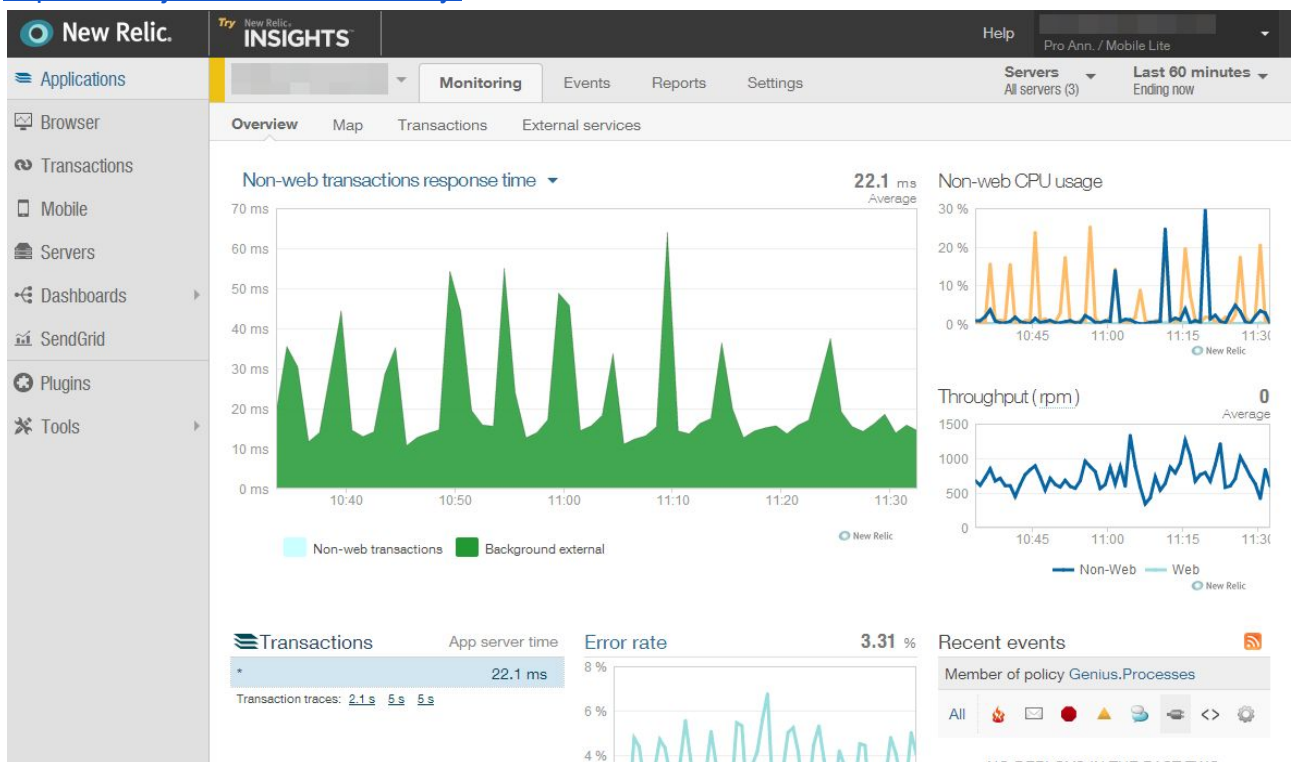
Running from the console allows you to output the results in html form. The test framework has been written with this in mind. A snippet of the html result is also shown below.



## Other considerations

You may wish to run a continuous build system which runs tests on submitted builds. I have reviewed all the options, and believe the TeamCity is your best option. It's fresher than Jenkins and supports xUnit natively. It seems easy to install and setup. I can send automated emails etc when builds fail.

<https://www.jetbrains.com/teamcity/>



# Building Tests

## General Workflow

The suggested workflow is to:

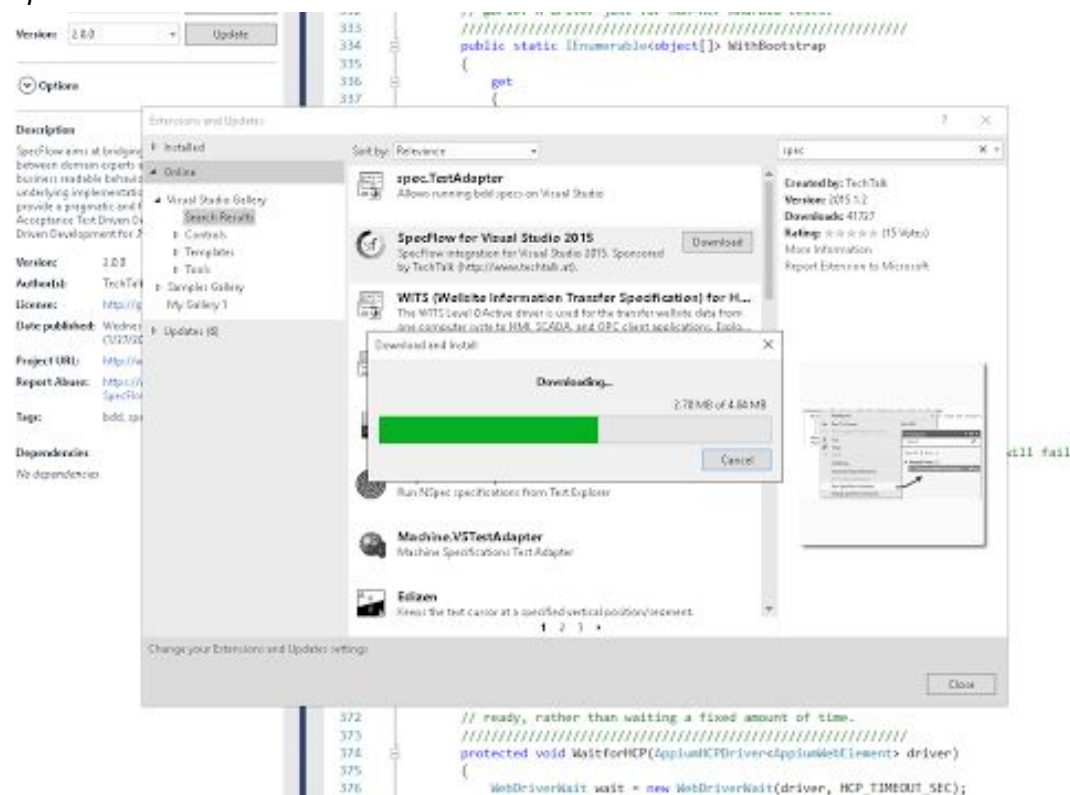
- Rough-out tests BDD language using the bundled SpecFlow
- Use Appium app to build our your steps, get element ideas, and generate some useful code to copy from when writing your tests.
- Fill out the generated test steps (from SpecFlow) with help from the supplied examples and the Appium App.

## Test Writing

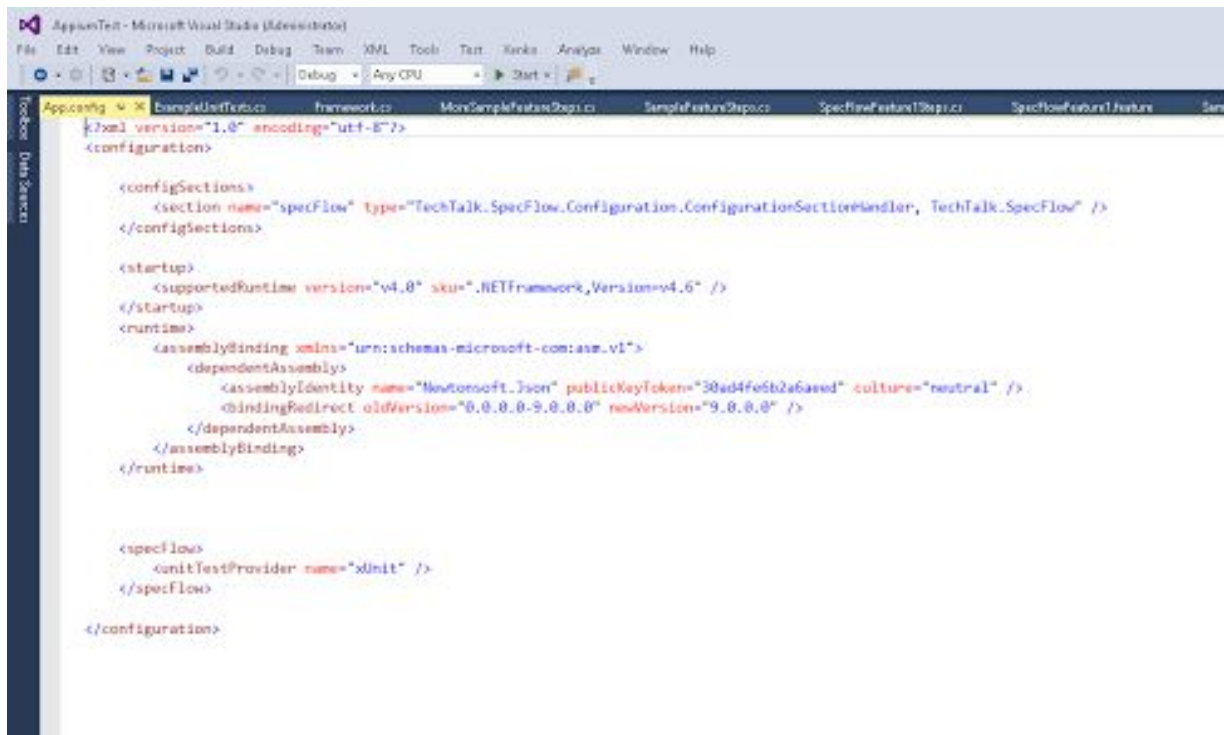
The suggested approach is to write tests on Windows using Visual Studio with the following installed:

- xUnit (nuget package)
- xUnit.VisualStudio.Runner (nuget package)
- xUnit.Console.Runner (nuget package)
- Custom appium-dotnet-driver (submodule within delivery)
- Custom specflow (submodule within delivery, prebuild under AppiumTest/SpecFlow/lib/)
- SpecFlow editor extension (not custom, get the VS2015 version from Tools->Extensions)

### *Specflow install screenshot for Visual Studio*



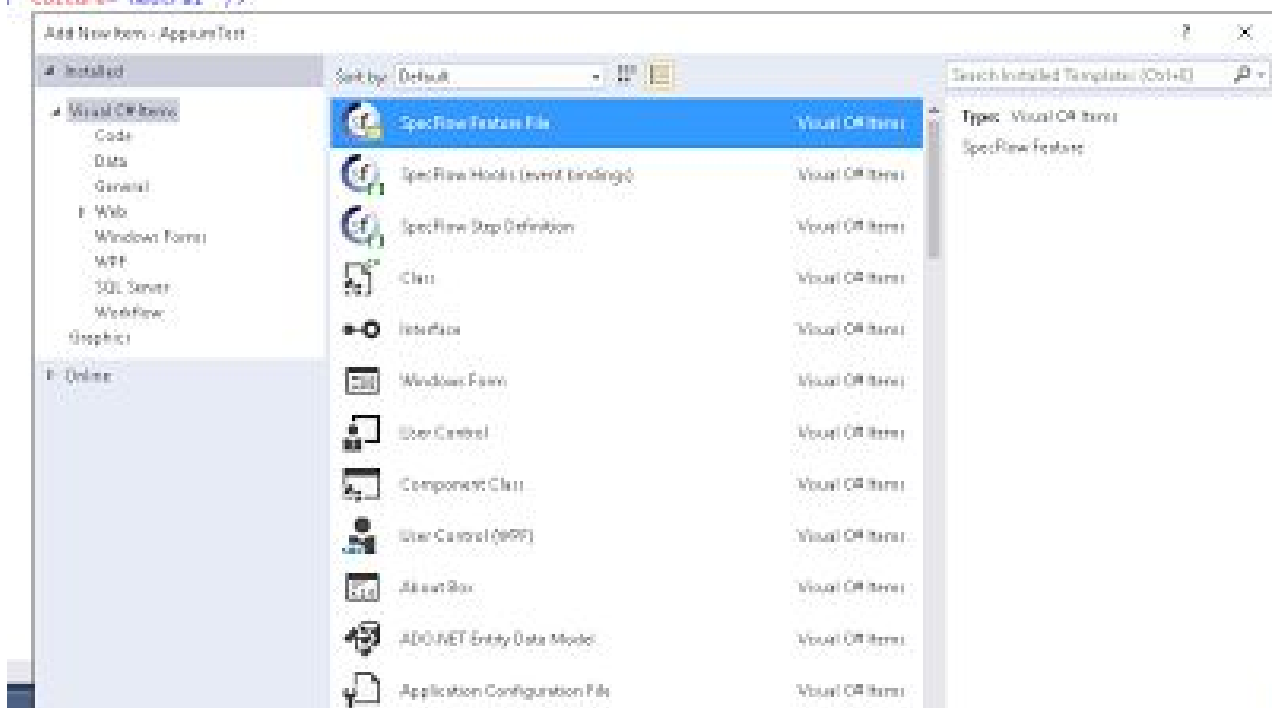
Once the above conditions are met, you will need to adjust to App.config file to look like the one supplied by the AppiumTest project. It tells SpecFlow that you want to generate with xUnit and not NUnit!. Screenshot below.



## Writing Scenarios and Features

You can write your tests using BDD if you like. This is done using SpecFlow. Right click in your solution view and choose to add a new file of type SpecFlow feature

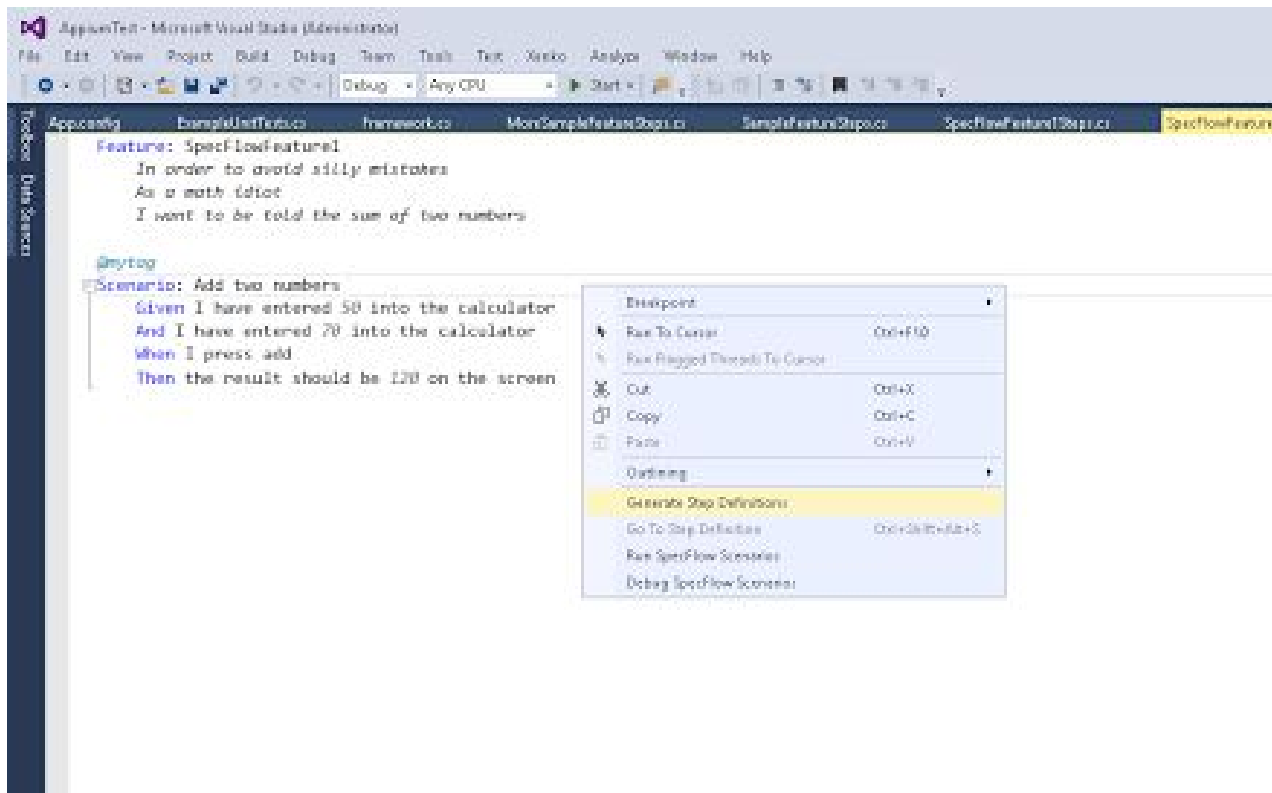
! " culture="neutral" />



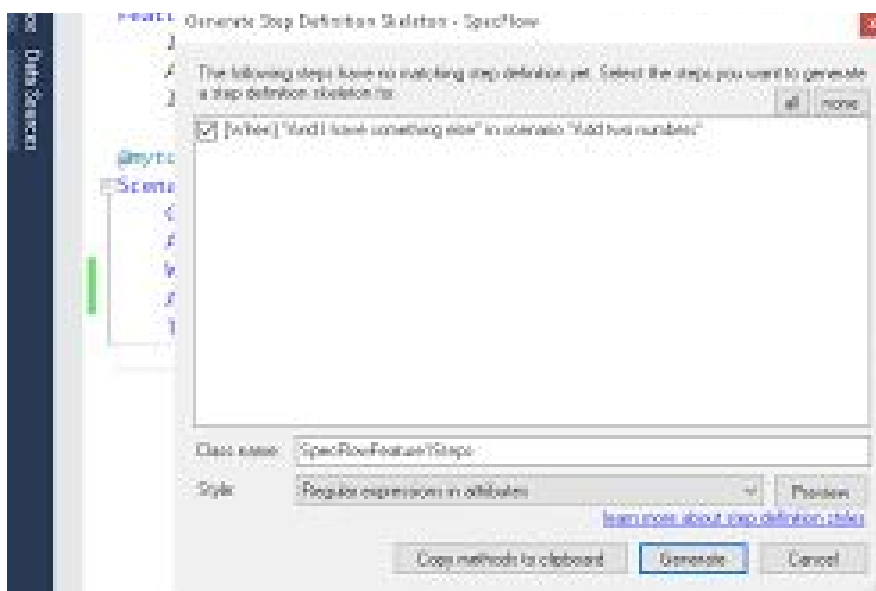


This will create a sample feature template that you can use to start writing your new test. Note that test steps can be reused across multiple tests. Just make sure that they have the same wording.

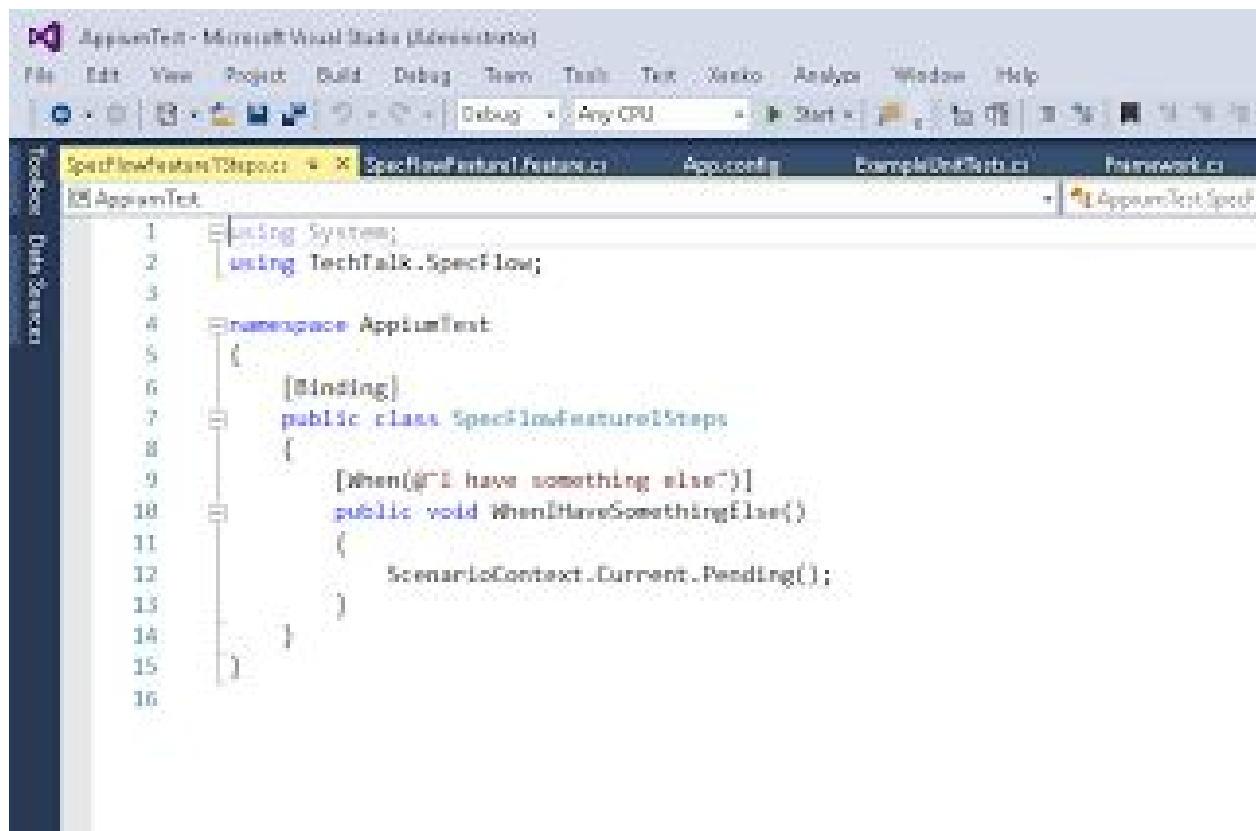
When your test is complete, you can then generate the step definitions. This creates C# files that you or another programmer can fill out. These generated functions will be automatically called when your test is run.



You will be presented with a dialog that has all the unwritten test steps presented. It will ask you to save these new step definitions to a new file. Do not be alarmed if there are fewer steps in this dialog than what has been written in the feature file, it just means that it is going to reuse an existing test step from another feature.



The result will be a new C# file that you must complete. For now, the test will fail with an “incomplete” message if it were run.



```
1 using System;
2 using TechTalk.SpecFlow;
3
4 namespace AppiumTest
5 {
6     [Binding]
7     public class SpecFlowFeature1Steps
8     {
9         [When(@"I have something else")]
10        public void WhenIHaveSomethingElse()
11        {
12            ScenarioContext.Current.Pending();
13        }
14    }
15 }
16
```

It's important to note that the AppiumTest project has two important hooks. Hooks are special methods that are called for each feature tests. In our case, we want to make sure that appium is shut down properly after each run. You can see this is implement in SpecHooks.cs of the AppiumTest project. You will want to do the same in any new projects.



```
29
30
31 [AfterScenario(Order = 100)]
32 public static void ReleaseDriver()
33 {
34     Testframework.ReleaseDriver();
35 }
36
37
```

The driver is created with a step in our AppiumTest sample. This is so that we can specific the driver capabilities ourselves. However, this is a bit clunky and it may be best to create a vanilla driver in a [BeforeScenario] hook and have the appium server specific the app and capabilities. This will allow you to have cleaner test cases with less Example data required.

## Supplied Framework

The modified appium-dotnet-driver supplies most of what you need to interface with vanilla appium and HCP. If you want to direct a command to HCP (Unity) then you set the driver to HCP mode prior to each request. For example:

```
driver.HCP().FindElementByName(...)
driver.HCP().FindElementById(...)
```

If you have a returned element, you do not need to worry about HCP mode as the element itself tracks that. In example:

```
var element = driver.HCP().FindElementByName("Button");
element.Click(); // This will direct the call to HCP because it's an HCP element
```

The AppiumTest project has some useful helpers that you may chose to use to incorporate into your test projects:

- WaitForHCP -> Wait for hcp to respond to an alive request. This will also tell you that unity is loaded and running
- Construct<...>Driver -> Create a driver for iOS or Android for out sample minimal app
- TakeScreenshot -> And example to take a screenshot, save it to disk, write that location to a the test log file
- Examples for how to:
  - Find an element (HCP or not)
  - Click an element
  - Touch an element
  - Hold touch on the screen
  - Input text using the keyboard
  - Getting visible status
  - Getting enabled status
  - Getting location and size of an element (top left point + width and height)
  - Swiping the screen

Important. The supplied framework uses xUnit which by default runs tests in different classes in parallel. This is not supported with Appium, thus you must manually turn it off, which is done at the start of Framework.cs

```
1  // -----
2  // <copyright file="Framework.cs">
3  // Copyright (c) Andrea Tino. All rights reserved.
4  // </copyright>
5  // -----
6
7
8  using Xunit;
9  [assembly: CollectionBehavior(MaxParallelThreads = 1)] // IMPORTANT!!!
10 // The above turns off parallel execution
11
12 namespace AppiumTest.Framework
13 {
```

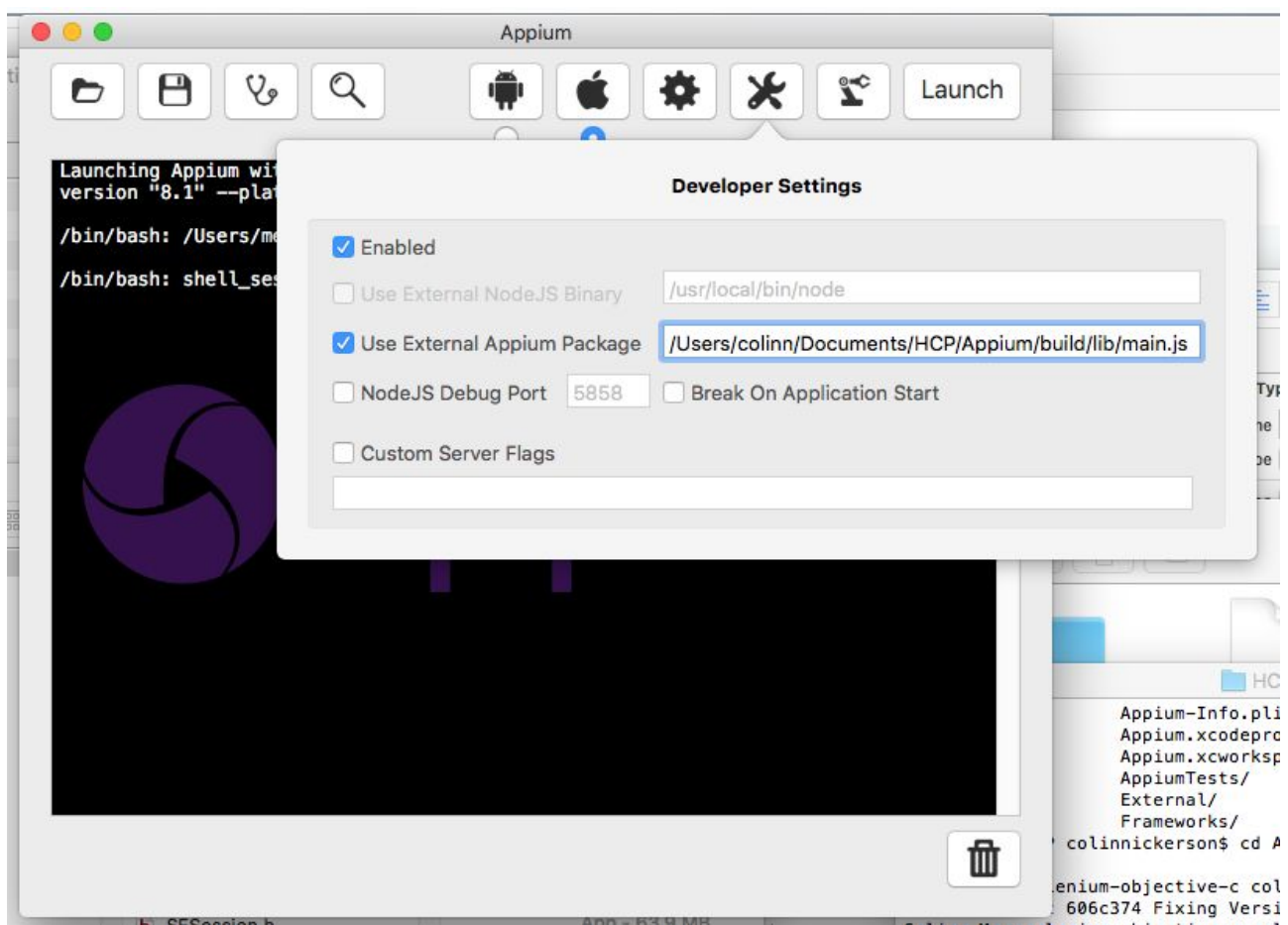
## Leveraging the Appium App

The Appium app has been updated to work with HCP. You can use it to retrieve element ids and record test steps to copy+paste into your main test project. In order to use it, you must use proper settings to point it to your app and our custom appium server.

### Appium-dot-app Settings






See the screenshots below for sample configurations

### *General (Required for all platforms)*





## Android



Launch

Android Settings

BasicAdvanced

Application

☐ App Path

Choose

☐ Package

▼

☐ Wait for Package

▼

☐ Launch Activity

▼

☐ Wait for Activity

▼

☐ Use Browser

Chrome

▼

☐ Full Reset☐ No Reset☒ Stop on Reset☐ Intent Action

android.intent.action.MAIN

☐ Intent Category

android.intent.category.LAUNCHER

☐ Intent Flags

0x10200000

☐ Intent Arguments☒ Use HCP

HCP Host

http://127.0.0.1

14812

Launch Device

☐ Launch AVD

▼

☐ Device Ready Timeout

5

s

☐ Arguments

Capabilities

Platform Name

Android

▼

Automation Name

Appium

▼

Platform Version

4.4 KitKat (API Level 19)

▼

☒ Device Name

Unnamed

☐ Language

en

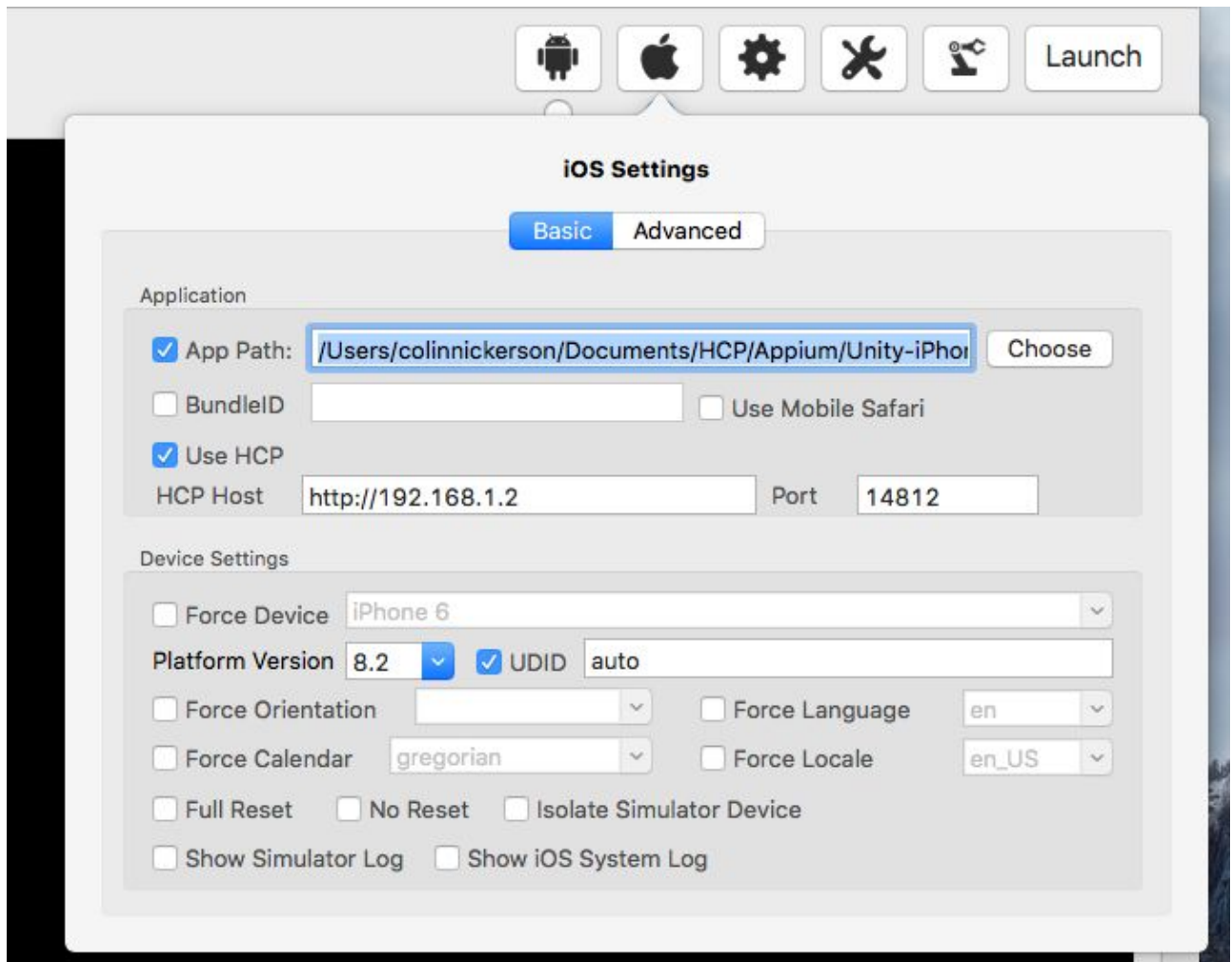
▼

☐ Locale

US

▼

## iOS



### Inspecting the UI

After you click the “Launch” button, you can open the inspector by clicking the button with the magnifying glass icon. This will load the app onto the device and request its pagesource. Pagesource is an XML doc showing the hierarchy of the app. We want to maintain compatibility with native UI controls, thus we manually merge the native pagesource with the HCP supplied pagesource built in request form unity.

The result is a navigable DOM that allows the QA tester to get element ID's and perform simple interactions with the app.

I see this as a good way to plan out your test cases, and to record code for simple commands that can be leveraged by the programmer ... I don't think it's going to be useful as a test case creator.

From the app, you can:

- Perform swipes
- Generic touches

- Button clicks
- Dismiss alerts
- Take screenshots