### Steps to create Automation Framework

Table of Contents

[Steps to create Automation Framework 1](#_Toc23755216)

[Java projects only. 2](#_Toc23755217)

[For Visual Studio (C#) projects. 2](#_Toc23755218)

[Create a New Solution – GIT – For .NET Core project 2](#_Toc23755219)

[Create New Solution – TFS 2](#_Toc23755220)

[Create new project 8](#_Toc23755221)

[Add Project Folders 9](#_Toc23755222)

[Install the following via Manage NuGet Packages 9](#_Toc23755223)

[Note on packages for .Net Core 9](#_Toc23755224)

[To be able to open the project in NUnit (optional) 10](#_Toc23755225)

[More on Using DOT NET Core 11](#_Toc23755226)

[Add Driver class to Utilities folder. 13](#_Toc23755227)

[Driver Class - Include the following methods (for example): 13](#_Toc23755228)

[Configuration Files 15](#_Toc23755229)

[App.config 15](#_Toc23755230)

[Configuration Manager 16](#_Toc23755231)

[.runsettings file 17](#_Toc23755232)

[RunSettings Class 18](#_Toc23755233)

[Create Hooks 19](#_Toc23755234)

[Adding .runsettings to WebHooks 20](#_Toc23755235)

[Create a BasePage 20](#_Toc23755236)

[Alternative, more flexible BasePage 22](#_Toc23755237)

[Pages 24](#_Toc23755238)

[If using PageFactory 24](#_Toc23755239)

[If not using PageFactory 24](#_Toc23755240)

[Feature Files 25](#_Toc23755241)

[Scenario Outline 27](#_Toc23755242)

[Step Definitions . 27](#_Toc23755243)

[Creating Steps and linking to Feature file 28](#_Toc23755244)

[More on Settings 30](#_Toc23755245)

[TestDataRespository 30](#_Toc23755246)

[Partial BasePage and Extra Methods 33](#_Toc23755247)

[Tips for individual BasePage Classes 34](#_Toc23755248)

## Java projects only.

Install JDK and add the path to the Path environment variable.

Install required packages via Maven. Specflow equivalent will be Cucumber

Add the path to the drivers for Chrome and IE (if necessary) to the Path environment variable.

# For Visual Studio (C#) projects.

## Create a New Solution – GIT – For .NET Core project

If you have not done so already, you will need to clone a GIT repository. See separate document **How to Clone a Git Repository**.

This will also contain the steps for creating the solution/project.

## Create New Solution – TFS

The notes below apply when using TFS as the source control.

A solution is a container for projects. Normally we only have 1 project per solution, but it can contain many

In Visual Studio, Select Team Explorer -> Source Control Explorer

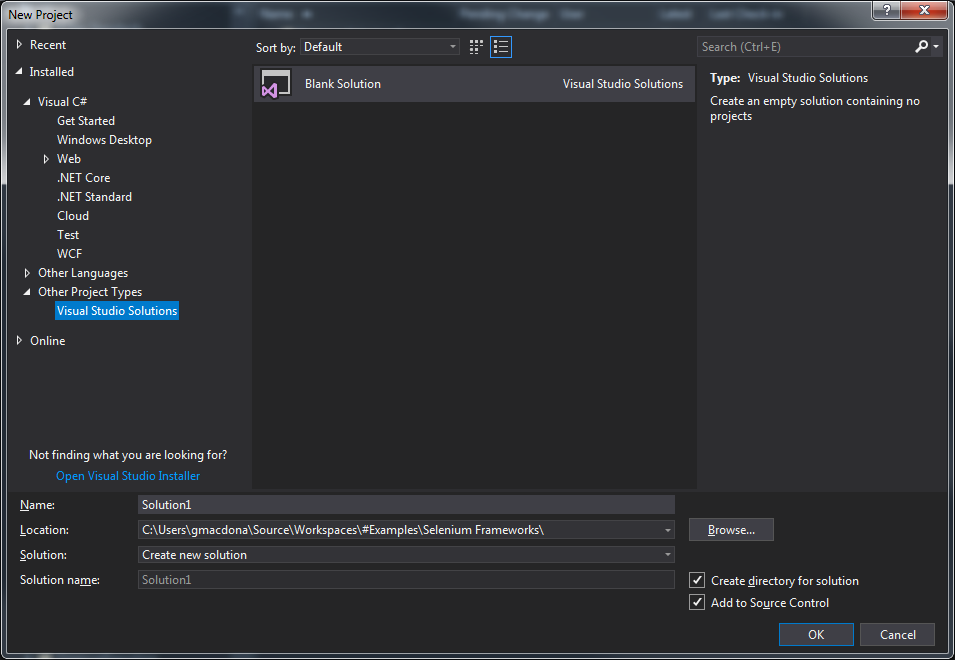
Select the folder where you will store your solution in source control.

Click on the Not mapped link.

Enter the folder on your local drive where you will be creating your solution and click on Map.

From main menu, select File-> New -> Project

Select Other Project Types 🡪 Visual Studio Solutions

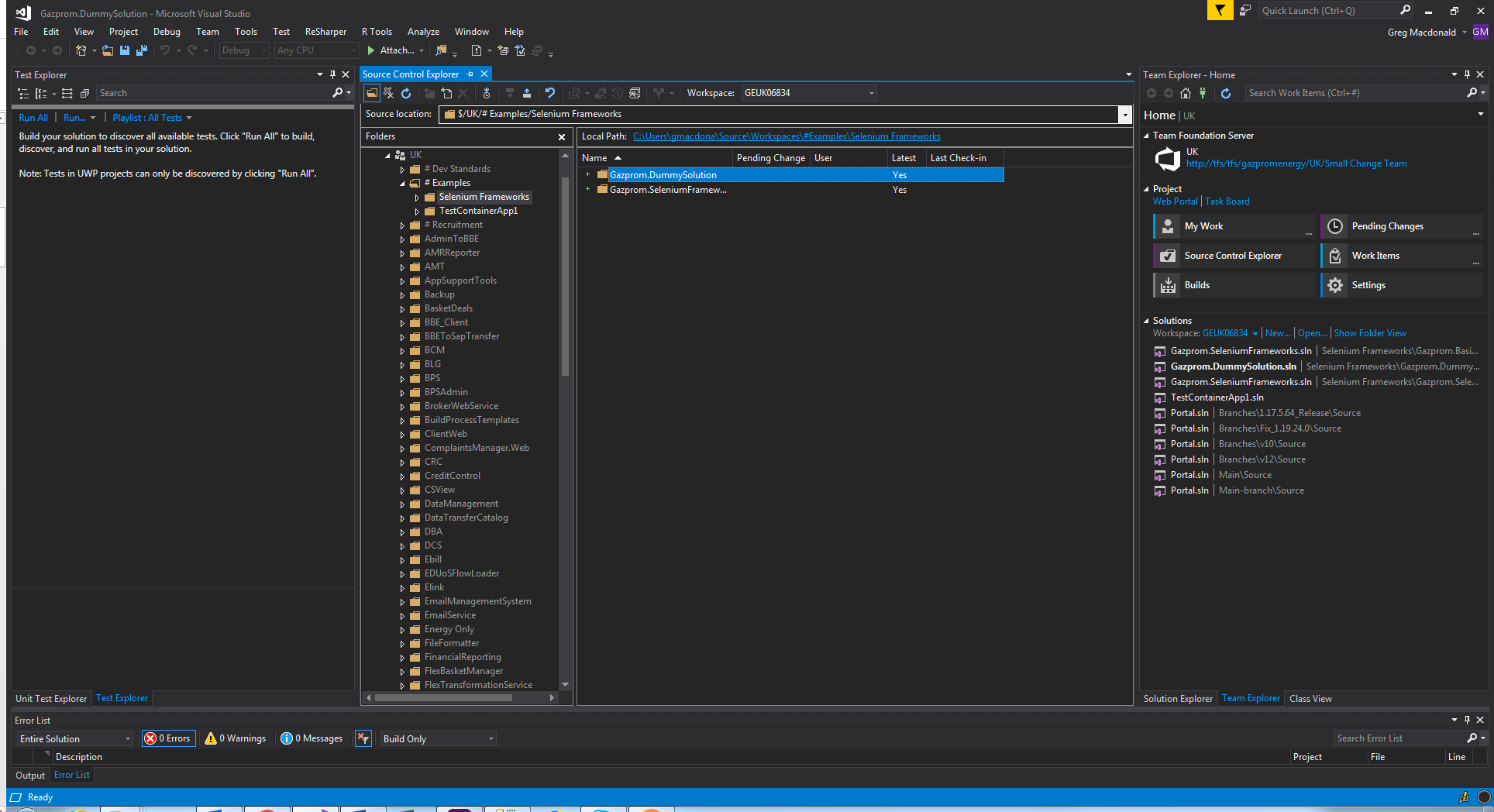


Give the solution a name in the format CompanyName.SolutionName

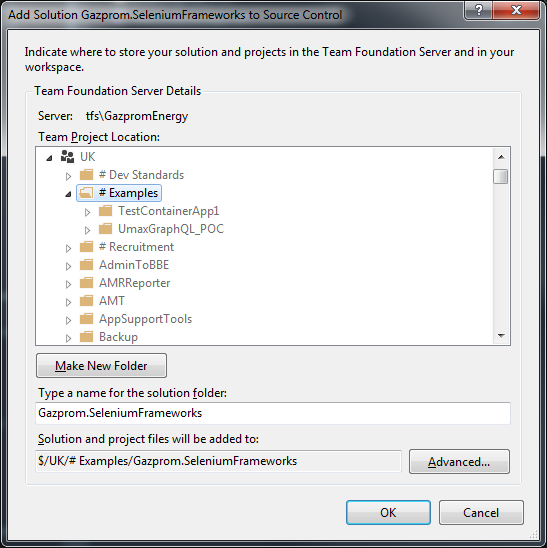
(Could use format CompanyName.ProjectArea.SolutionName)

Set the Location field to match where you have mapped the solution to and click on OK.

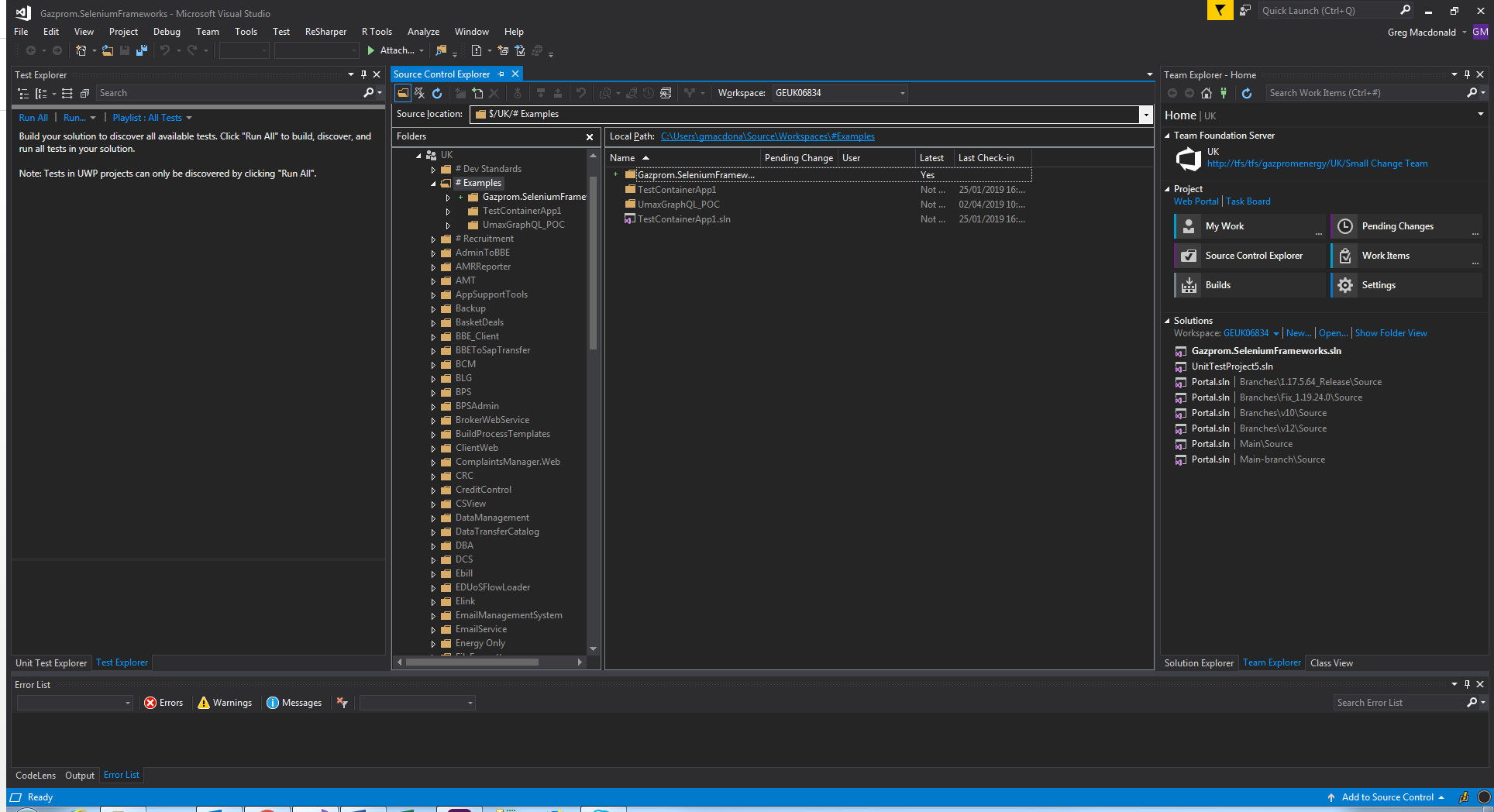
Your solution will now appear in Source Control Explorer



You may be presented with a screen where you are asked to select to store your solution in source control. Select an appropriate location (probably the same as the folder you mapped earlier) and click on OK.



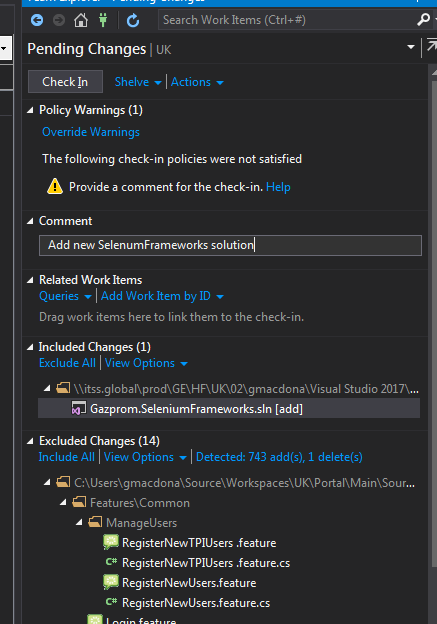
Solution is created in source control

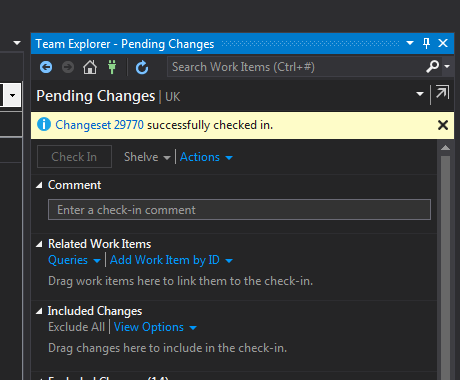


You will now need to check the solution into source control.

Go to Team Explorer->Pending Changes and select your solution. Right click and select Exclude Unselected to exclude any other pending changes that you may have.

Add a suitable comment and click on Check In





## Create new project

You may get a default project created with the same name as the solution. If so, you can rename it and use it. You will also need to change the folder in Source Control Explorer to avoid confusion

Check in your change via Team Explorer.

**However, it may be safer to delete it and create a new one following steps below**

Double click on the solution file (.sln) in Source Control explorer.

Go to Solution Explorer. Right click on your solution and select Add -> New Project

From the LHS, select Test and then select a project type from the list.

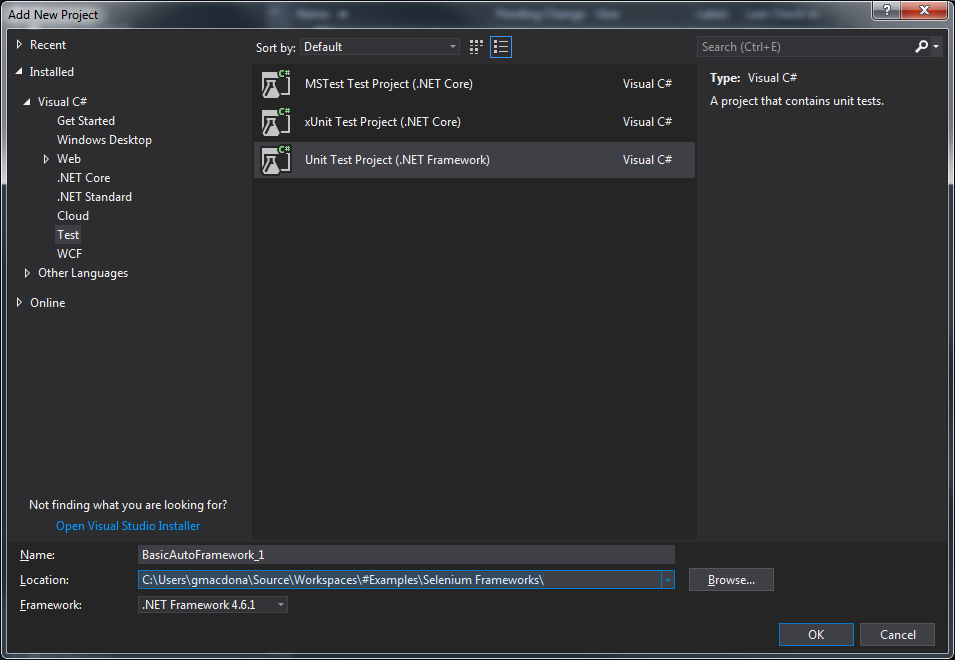
(MSTest Test Project (.NetCore) seems to be new one to use. However, it does NOT currently work with Specflow. Need to wait until Specflow 3.0 is released. So just use Unit Test Project (.NET Framework) until then)

Also see this link about issues with automatic updates

<https://specflow.org/2018/vs-integration-breaking-changes-affects-all-users/>

Give your project a name.

Set the Location field to match where you have mapped the solution to and click on OK.



**(Once complete, right click on the project and select Properties. Make sure Output Type is Class Library )**

## Add Project Folders

In Solution Explorer, right click on the project name

Click on Add->New Folder

Add new folders for Features, Pages, Steps and Utilities

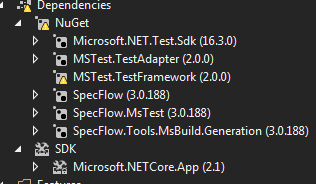
Also create a Shared subfolder in Steps. Your Hooks file will go here later.

## Install the following via Manage NuGet Packages

### Note on packages for DOT Net Core

For .NET Core projects, these are packages that you should install initially.

NOTE: There are later versions of the Specflow packages available, but these are the versions which seem to work. If you select later ones there is a chance they may not work. Use trial and error to see which ones are the latest versions that work for you.



When you create you feature files, make sure that the Custom Tool field in the project Properties is blank.

Right click on References (Dependencies for Core) and select Manage NuGet Packages.

(Note: for MSTest Test Project (.NetCore) projects, there is no longer a References folder. Use the Dependencies folder instead)

* DotNetSeleniumExtras.WaitHelpers \*
* DotNetSeleniumExtras.PageObjects \* (only if using Page Factory)
* Selenium.Support
* Selenium.WebDriver
* Selenium.WebDriver.ChromeDriver
* SpecFlow
* SpecFlow.MsTest
* MSTest.TestAdaptor
* MSTest.TestFramework (these last 2 should come already installed)

(Optional if using NUnit rather than MSTest)

* Nunit
* Nunit.Runners
* NUni3.TestAdapter

…

\* - Be careful using these as they are likely to be deprecated.

Save the source code for WaitHelpers on own machine (or Google Drive) as a precaution.

## To be able to open the project in NUnit (optional)

Select Project-Add Assembly to add the project.dll to NUnit.

For example:

C:\Users\user\Documents\Visual Studio 2015\Projects\AutomationFrameWork1\AutomationFrameWork1\bin\Debug\AutomationFrameWork1

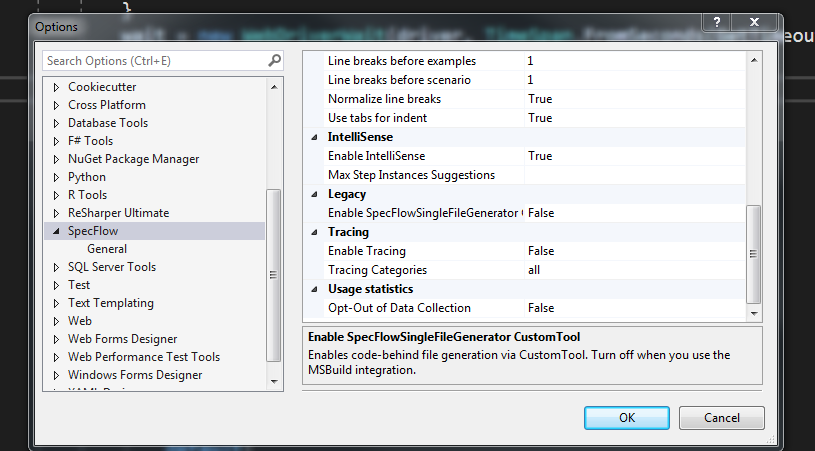
## More on Using DOT NET Core

If using DOT Net Core rather than DOT Net Framework 4.x:

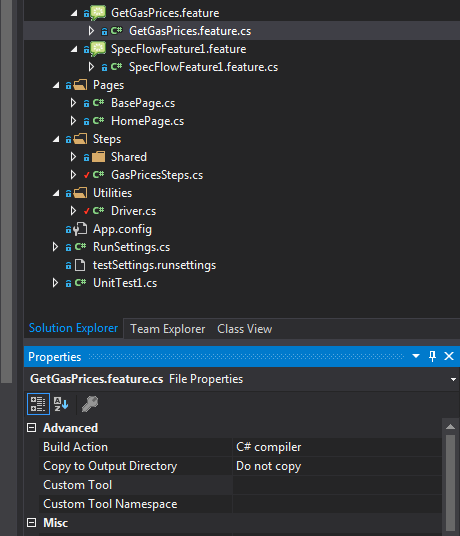
Got to Tools->Options-> SpecFlow

Scroll down to Legacy section

Set Enable SpecFlowSingleFileGenerator Custom Tool = False

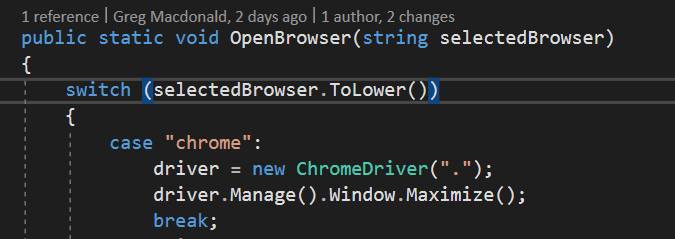


On each of your x.feature.cs files (that is the code generated for your feature file), right click and select Properties. Ensure that Custom Tool field is blank.



In your OpenBrowser() method in the Driver class, include a dot as the parameter for ChromeDriver().

This tells it to look in root to find chromedriver.exe



## Add Driver class to Utilities folder.

NOTE: Normally make the Driver class Public Static. All driver methods are also Public Static.

In the using section add:

using System;

using OpenQA.Selenium;

using OpenQA.Selenium.Chrome;

using OpenQA.Selenium.IE;

using OpenQA.Selenium.Support.UI;

At the top of the Driver class add:

public static IWebDriver driver;

public static WebDriverWait wait;

//This values come from .runsettings file, set via WebHooks (explained later)

public static string RootURl;

## Driver Class - Include the following methods (for example):

OpenBrowser(string selectedBrowser) –

This method needs to be included as a minimum. It should also maximise the window and create an instance of the WebDriverWait .

We will see alternative versions of this method later, when we come to Configuration files.

E.G.

public static void OpenBrowser(string selectedBrowser)

{

switch (selectedBrowser.ToLower())

{

case "chrome":

driver = new ChromeDriver();

driver.Manage().Window.Maximize();

break;

case "ie":

driver = new InternetExplorerDriver();

driver.Manage().Window.Maximize();

break;

default:

Debug.Print("unknown browser selected");

break;

}

wait = new WebDriverWait(driver, TimeSpan.FromSeconds(5));

}

public static void NavigateTo(string targetURL)

{

driver.Navigate().GoToUrl(targetURL);

}

public static void ShutDown()

{

driver.Quit();

}

Don’t call it Quit() as you could end up calling it in an endless loop if you call Driver.Quit() instead of driver.Quit() !

## Configuration Files

Some settings can change from one environment to another (the Base URL will definitely change). The best way to deal with this is via a configuration file.

There are 2 ways of doing this and the choice depends on how your Continuous Integration (CI) Tooling is set up. The old way (which may still be used where you are) is to have a file per environment and the CI tool will pick the appropriate file for the environment that it is running in. The new way is to have a single file and the CI tool will overwrite the settings in this file depending on the environment that it is running in.

The file used in the first method is App.config and the file used for the second is test.runsettings (this file can be called anything as long the file type is .runsettings.)

## App.config

The App.config file should already exist and have some specflow details in it. If not, right click on the project name and select Add - > New Item. From the LHS panel select General-> Application Configuration File. The name will probably default to App.config. Click on Add.

If you have had to create this manually (which seems to be the case for later versions of Specflow, then you will probably need to add the missing Specflow details. Although I suspect later versions do not actually require the code shown below, so realistically only need to add this for pre 3.0 versions when they have not been added automatically).

Add the following within the <configuration> </configuration> section:

<configSections>

<section name="specFlow" type="TechTalk.SpecFlow.Configuration.ConfigurationSectionHandler, TechTalk.SpecFlow" />

</configSections>

<specFlow>

<!-- For additional details on SpecFlow configuration options see http://go.specflow.org/doc-config -->

<!-- For additional details on SpecFlow configuration options see http://go.specflow.org/doc-config -->

<unitTestProvider name="MsTest" /></specFlow>

Here you can add configuration and constant values for such things as:

Base URL,

Browser

Default timeout for waits.

At the bottom of the <configuration> section, enter the following:

<appSettings>

<add key="DefaultTimeoutSeconds" value="10"/>

<add key="Browser" value="Chrome"/>

<add key="BaseURL" value ="http://porapp01qa/"/>

</appSettings>

Enter more key value pairs inside appSettings section as required

## Configuration Manager

In order to access App.config, you will need a Configuration Manager class. To use this you will need to do the following:

It's not only necessary to use the **namespace** System.Configuration. You have also to add the reference to the **assembly** System.Configuration.dll , by

1. Right-click on the *References / Dependencies*
2. Choose *Add Reference*
3. Find and add System.Configuration. (in Assemblies)

(It looks like you just need to add the **namespace** System.Configuration when using Core)

Now, to be able to get the values of the DefaultTimeoutSeconds key and you added earlier, write these methods in your Driver class:

public static string GetValueFromConfigKey(string key)

{

return ConfigurationManager.AppSettings[key];

}

public static int GetTimeoutSeconds()

{

var time = GetValueFromConfigKey("DefaultTimeoutSeconds");

return int.Parse(time);

}

public static string GetBrowser()

{

return GetValueFromConfigKey("Browser");

}

Now you can write a new version of OpenBrowser() that gets the browser and timeout seconds from App.config (leave the old one in place, that can be used later with runSettings. It is OK to have 2 version of the same method if the signatures are different. This is called overloading)

public static void OpenBrowser()

{

switch (GetBrowser().ToLower())

{

case "chrome":

driver = new ChromeDriver();

driver.Manage().Window.Maximize();

break;

case "ie":

driver = new InternetExplorerDriver();

driver.Manage().Window.Maximize();

break;

default:

Debug.Print(("unknown browser selected"));

break;

}

wait = new WebDriverWait(driver, TimeSpan.FromSeconds(GetTimeoutSeconds()));

}

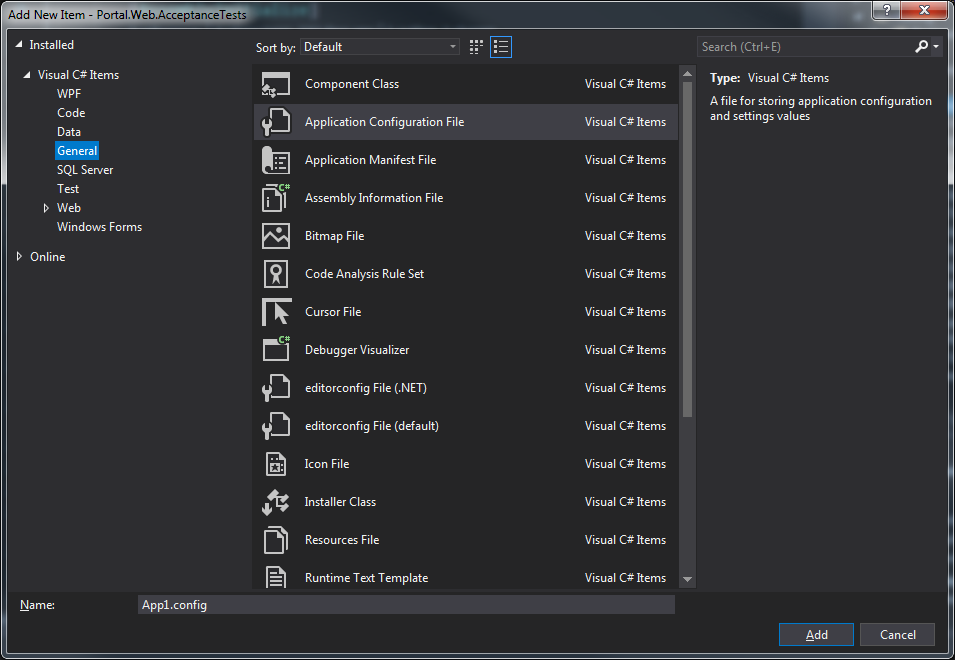
Alternatively, you could just use the previous version and pass in GetBrowser() as the parameter when you call it.

## .runsettings file

In Solution Explorer, at the project level, add a new item.

From LHS panel select General then select an Application Configuration File.

Rename the file so that it has an extension of .runsettings



E.G.

testSettings.runsettings

Change the <configuration> section into a <RunSettings> section

Within this section, add a <TestRunParameters> section

With this section add your various parameters as name value pairs.

E.G.

<?xml version="1.0" encoding="utf-8" ?>

<RunSettings>

<TestRunParameters>

<Parameter name="EnvironmentKey" value="POR QA 01" />

<Parameter name="WebRoot" value="http://porapp01qa/" />

<Parameter name="ApiRoot" value="http://porapp01qa:81/api/" />

<Parameter name="EnableChrome" value="true" />

<Parameter name="EnableIe" value="false" />

</TestRunParameters>

</RunSettings>

See this link for further information:

<https://docs.microsoft.com/en-us/visualstudio/test/configure-unit-tests-by-using-a-dot-runsettings-file?view=vs-2019>

## RunSettings Class

This is a wrapper class to take care of the interaction with the .runsettings file.

In Solution Explorer, at the project level, add a new item.

From LHS panel select Code then select Class.

Name the file RunSettings.cs

Add code similar to this example:

using System;

using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace BasicFrameworkOne

{

[TestClass]

public class RunSettings

{

//The TestContext variables here are only relevant when run on server.

//context is set by the framework itself. It is static so \_testContext needs to //be static too. testContext is passed to the non-static TestContext that //cannot use static variables.

//(Can't pass directly from context to TestContext it seems.)

[AssemblyInitialize]

public static void AssemblyInit(TestContext context)

{

\_testContext = context;

}

private static TestContext \_testContext;

public TestContext TestContext => \_testContext;

//The Test.Context.Propeties are set by TFS if tests run remotely on server.

//If run locally, they will be null and the default value will be taken //instead.

public string Browser => (TestContext.Properties["Browser"] ?? "Chrome").ToString();

public string EnvironmentKey => (TestContext.Properties["EnvironmentKey"] ?? "POR QA 01").ToString();

public string WebRoot => (TestContext.Properties["WebRoot"] ?? "http://porapp01qa/").ToString();

public string ApiRoot => (TestContext.Properties["ApiRoot"] ?? "http://porapp01qa:81/api/").ToString();

public bool EnableChrome => Convert.ToBoolean(TestContext.Properties["EnableChrome"] ?? "true");

public bool EnableIe => Convert.ToBoolean(TestContext.Properties["EnableIe"] ?? "false");

}

}

## Create Hooks

This class should go in Steps->Shared

A basic one would look like this:

using BasicFrameworkOne.Utilities; //location of Driver class

using TechTalk.SpecFlow;

namespace Gaz\_Practice\_2.Steps.Shared

{

[Binding]

public class Hooks

{

//Since RunSettings class is non-static and the Driver class is static,

//we need a static instance of RunSettings to pass details to Driver class

public static RunSettings \_runSettings = new RunSettings();

[BeforeScenario()]

public static void StartBrowsers()

{

Driver.RootURl = Driver.GetBaseURL();

Driver.OpenBrowser();

}

[AfterScenario]

public void ShutDown()

{

Driver.ShutDown();

}

}

}

If just creating tests and not Specflow scenarios, you will use these annotations instead (Google for more if required. I think there are part of Nunit):

[TestFixture]

[TearDown]

## Adding .runsettings to WebHooks

This will allow you to get such settings as the Browser from the.runsettings file instead of App.Config.

Change your WebHooks to this:

using BasicFrameworkOne.Utilities; //location of Driver class

using TechTalk.SpecFlow;

namespace BasicFrameworkOne.Steps.Shared

{

[Binding]

public class WebHooks

{

//Since RunSettings class is non-static and the Driver class is static,

//we need a static instance of RunSettings to pass details to Driver class

public static RunSettings \_runSettings = new RunSettings();

//Constructor

//Don't seem to need one. BeforeScenario is run before the constructor anyway.

[BeforeScenario()]

public static void StartBrowsers()

{

Driver.RootURl = \_runSettings.WebRoot;

Driver.OpenBrowser(\_runSettings.Browser);

}

[AfterFeature()]

public static void CloseBrowsers()

{

Driver.ShutDown();

}

}

}

## Create a BasePage

In the Pages folder, right click and select Add -> New Item. From LHS panel select Code then Class.

BasePage does not need to include much. Can have such things as GetClickableElement() and ClickOnElement(). You will probably add to this over time as it becomes apparent that certain methods are common enough to be here.

As a minimum, you will need a constructor and your IWebdriver and WebDriverWait, but some other methods for waiting for and clicking on elements would be useful too.

Note: You don’t have to set up baseDriver and baseWait. You could just reference Driver.driver and Driver.wait directly, this way is just a bit neater.

E.G.

using System;

using OpenQA.Selenium;

using OpenQA.Selenium.Support.UI; //Need this for WebDriverWait

using BasicFrameworkOne.Utilities; //Location of Driver class

//So it knows to take ExpectedConditions from here instead of (deprecated) OpenQA.Selenium

using ExpectedConditions = SeleniumExtras.WaitHelpers.ExpectedConditions;

namespace BasicFrameworkOne.Pages

{

public class BasePage

{

private IWebDriver baseDriver;

private WebDriverWait baseWait;

//Constructor

public BasePage()

{

baseDriver = Driver.driver;

baseWait = Driver.wait;

}

//Methods

/// <summary>

/// Returns URL to currently displayed page

/// </summary>

/// <returns>String</returns>

public string PageUrl()

{

return Driver.driver.Url;

}

/// <summary>

/// Waits for, then returns, a clickable element

/// </summary>

/// <param name="elementLocator">Used to locate the element, e.g. By.Id("xyz")</param>

/// <returns>IWebElement</returns>

public IWebElement GetClickableElement(By elementLocator)

{

return baseWait.Until(ExpectedConditions.ElementToBeClickable(elementLocator));

}

/// <summary>

/// Waits for given element to be clickable then clicks on it.

/// </summary>

/// <param name="elementLocator">Used to locate the element, e.g. By.Id("xyz")</param>

public void ClickOnElement(By elementLocator)

{

GetClickableElement(elementLocator).Click();

}

/// <summary>

/// Checks if an element is visible on the page. Visibility means that the element

/// is not only displayed but also has a height and width that is greater than 0.

/// </summary>

/// <param name="elementLocator">Used to locate the element, e.g. By.Id("xyz")</param>

/// <returns>boolean</returns>

public bool IsElementVisible(By elementLocator)

{

try

{

return baseWait.Until(ExpectedConditions.ElementIsVisible(elementLocator)).Displayed;

}

catch (Exception e)

{

}

}

}}

## Alternative, more flexible BasePage

In the BasePage above, you are tied to a single WebDriverWait and so a single timeout value.

The version below creates a new WebDriverWait for each method that requires one so that you can be flexible on the wait time.

int? waitSeconds = null

This parameter is nullable. If it does not contain a value then it gets replaced by webDriverTimeout in this line

int seconds = waitSeconds ?? webDriverTimeout;

This way you can supply your own timeout value when you call the method, or just accepted the default value by not supplying a value.

public class BasePage

{

//You will set the value of baseDriver to value in Driver Class

//using the constructor below. This will be inherited by all your pages

//It is just a bit neater the having to write Driver.driver

private IWebDriver baseDriver;

private int webDriverTimeout;

public BasePage()

{

baseDriver = Driver.driver;

webDriverTimeout = Driver.GetTimeoutSeconds();

}

//Methods

/// <summary>

/// Waits for, then returns, a clickable element

/// </summary>

/// <param name="elementLocator">Used to locate the element, e.g. By.Id("xyz")</param>

/// <returns>IWebElement</returns>

public IWebElement GetClickableElement(By elementLocator, int? waitSeconds = null)

{

int seconds = waitSeconds ?? webDriverTimeout;

WebDriverWait wait = new WebDriverWait(baseDriver, TimeSpan.FromSeconds(seconds));

return wait.Until(ExpectedConditions.ElementToBeClickable(elementLocator));

}

/// <summary>

/// Waits for given element to be clickable then clicks on it.

/// </summary>

/// <param name="elementLocator">Used to locate the element, e.g. By.Id("xyz")</param>

public void ClickOnElement(By elementLocator)

{

GetClickableElement(elementLocator).Click();

}

/// <summary>

/// Checks if an element is visible on the page. Visibility means that the element

/// is not only displayed but also has a height and width that is greater than 0.

/// </summary>

/// <param name="elementLocator">Used to locate the element, e.g. By.Id("xyz")</param>

/// <returns>boolean</returns>

public bool IsElementVisible(By elementLocator, int? waitSeconds=null)

{

int seconds = waitSeconds ?? webDriverTimeout;

WebDriverWait wait = new WebDriverWait(baseDriver, TimeSpan.FromSeconds(seconds));

try

{

return

wait.Until(ExpectedConditions.ElementIsVisible(elementLocator)).Displayed;

}

catch (Exception)

{

return false; //Element not found, so Displayed is false

}

}

}

## Pages

You are now ready create your first page object class, e.g. LoginPage.cs

### If using PageFactory

NOTE: PageFactory is being deprecated.

Use PageFactory to create and initialise the PageObject (unless not using new way of findby) in the BaseTest, e.g.

Pages.PageObject theLoginPage = new Pages.PageObject();

PageFactory.InitElements(Driver.driver, theLoginPage);

### If not using PageFactory

Just create pages without the PageFactory.InitElements(Driver.driver, theBasepage);

You will need to create locator Bys instead of WebElements

E.G.

private readonly By \_firstNameLocator = By.Id("firstName");

If you are inheriting from a BasePage, then it will just use the constructor from the BasePage, so you don’t need a constructor in the page unless you need something extra.

To inherit from BasePage, write the signature like this:

public class LoginPage : BasePage

Since PageFactory is being deprecated, the elements do not get initialised at page construction, so use element locators at the top of the page instead of elements.

E.G.

private readonly By \_passwordLocator = By.Id("Password");

These locator variables are of type By and can be passed into methods that accept that type.

If you access any of the methods from the Driver class, then you will need to include the namespace in your using section.

E.G,

using BasicFrameworkOne.Utilities;

Here is an example page object that uses methods from the Base page:

using OpenQA.Selenium;

using BasicSeleniumFramework.Utilities;

namespace BasicSeleniumFramework.Pages

{

public class LoginPage : BasePage

{

//Constructor

//It will just use the one from BasePage unless it needs something extra.

//WebElements

private readonly By \_emailLocator = By.Id("Email");

private readonly By \_passwordLocator = By.Id("Password");

private readonly By \_loginButtonLocator = By.Id("Login-Btn");

//Methods

public void GotoLoginPage(string rootURL)

{

Driver.NavigateTo(rootURL);

}

public void EnterEmail(string emailAddress)

{

IWebElement email = GetClickableElement(\_emailLocator);

email.SendKeys(emailAddress);

}

public void EnterPassword(string passwordText)

{

IWebElement password = GetClickableElement(\_passwordLocator);

password.SendKeys(passwordText);

}

public void ClickLoginButton()

{

ClickOnElement(\_loginButtonLocator);

}

}

}

## Feature Files

**NOTE**: To make sure Specflow generates its code-behind-files for feature files, go to App.config and make sure this line is included:

<unitTestProvider name="MsTest" /></specFlow>

(Not sure if this is required if you create project as MSTest Test Project rather than Unit Test Project)

Feature files need to have the .feature extension.

Right click on the Features folder and select Add->New Item.

On the panel on the LHS click on Visual C# Items. Specflow Feature File should now be one of the options on RHS. Select this. Give it a name and click on Add.

A feature file with a basic layout will be created. Amend this as required. More importantly, another file has been generated and is just below the feature file in the hierarchy. This is what links the feature file to your step definition files.

### Scenario Outline

If you are using input data as part of your feature, you will need to use Scenario Outline: and Examples: followed by a table. The parameters are enclosed in “<>” and must match the names in the table.

Here is an example of such a feature:

Feature: Login

*I want to login*

*@mytag*

Scenario Outline: Login

Given I am on the Login Page

When I enter email "*<Email>*" and password "*<Password>*"

Then I should be logged in successfully

Examples:

| *Email* | *Password* |

| cs@portal.com | Testing321|

## Step Definitions .

You are now ready create your first step definition class.

Right click on the Steps folder and select Add->New Item.

On the panel on the LHS, select Code. From the RHS select Class.

Give it a name, e.g. LoginSteps.cs, and click on Add.

You will need to include

using TechTalk.SpecFlow;

using BasicFrameworkOne.Pages; //namespace for your page objects.

You will also need

[Binding]

At the top of namespace, just below the first curly bracket.

Put your page declarations at the top of class, just below the first curly bracket.

E.G.

private readonly LoginPage \_theLoginPage;

You will need a constructor for the steps class that includes instantiating all page objects that it will use.

E.G.

public LoginSteps()

{

\_theLoginPage = new LoginPage();

}

## Creating Steps and linking to Feature file

Now you can start creating steps:

From the Feature File, right click on the first line that you want to create a step for

E.G.

Given I am on the Login Page

And select Go to Definition (F12)

Select Yes to the question about copying the step binding skeleton to the clipboard.

Go back to the step definition file and paste in the skeleton

E.G.

[Given(@"I am on the Login Page")]

public void GivenIAmOnTheLoginPage()

{

ScenarioContext.Current.Pending();

}

You will likely need to include

using TechTalk.SpecFlow;

using BasicFrameworkOne.Pages; //namespace for your page objects.

using BasicFrameworkOne.Utilities; //location of Driver class

Replace the body of the method with the necessary code, this should mostly be calls to the page methods.

E.G.

\_theLoginPage.GotoLoginPage();

(**Note** if you are using RunSettings to hold values such as your RootURL, then this method would likely be called like this:

\_theLoginPage.GotoLoginPage(Driver.RootURl);

)

Repeat the above for your other steps.

Create any other page objects and steps that your scenario may need.

Build the solution and you are ready to run your first scenario!

## More on Settings

**.runsettings – config file**

This defines what the parameters are, and initial values.

So if you want a new config value, add it here as well as RunSettings.cs.

These values can be overwritten by TFS and are only really used by the server.

E.G.

<Parameter name="WebRoot" value="http://porapp01qa/" />

**RunSettings.cs**

This will actually get the values.

Context (Set by framework -> \_testContext (static variable) -> TestContext (non-static variable)

TestContext.Properties["WebRoot"] -> WebRoot

OR, if this is null then

“http://porapp01qa/” -> WebRoot

Since RunSettings class is non-static and the Driver class is static, need a static instance of RunSettings to pass details to Driver class

**Context.cs**

This is the location from where the program properly interacts with the variables

You don’t really need it, you could just get the values direct from RunSettings.

Context seems to only be useful if you have additional things for it to wrap around, e.g. test data. So you would have runsettings and test data all within a context class.

RootUrl = RunSettings.WebRoot;

## TestDataRespository

This functionality will allow you to store test data (in this case users and their login details) within your solution to allow your tests to be data driven.

First add a Data folder and a Templates folder to your solution.

Then install the FileHelpers package by Marcos Meli via NuGet Manager.

You will then need a class that defines what your data will look like.

Create a new class something like this and save it in the Templates folder:

using FileHelpers;

namespace BasicFrameworkOne.Utilities

{

[DelimitedRecord(",")]

public class UserTemplate

{

public string UserKey { get; set; }

public string Username { get; set; }

public string Password { get; set; }

public string AccountNum { get; set; }

public string SiteRefNum { get; set; }

public string Lastname { get; set; }

public string NewPassword { get; set; }

public string GroupAccountNum { get; set; }

public string SiteRefNumElectricity { get; set; }

public string BasketIds { get; set; }

}

}

Create a .csv file to contain data that matches the template above.

(**NOTE**: the following are used in the Portal solution. They allow you to use .xlsx files instead of .csv files. However, I do not know where these reside and so I cannot include them in this framework. If you can find them, then you can use these instead of FileHelpers. You would use code like this to get your users

var users = \_xlsxService.ReadFromFile<TestUser>($"{\_dataFolder}Users.xlsx");

using GazpromEnergy.Shared.Integration.Interface.Services.Files;

using GazpromEnergy.Shared.Integration.Services.Excel;

)

In your Utilities folder, create a new class called TestDataRepository

In this class add a class level variable to hold an array of UserTemplate objects.

Then add ad method to read data from the file to the data repository and methods to get user details from the date repository:

class TestDataRepository

{

private UserTemplate[] userRecords;

public void ReadFileIntoDataRepository()

{

var engine = new FileHelperEngine<UserTemplate>();

var dataDirectoryPath = GetFilePath();

var dataFilePath = Path.Combine(dataDirectoryPath, "TestUsers.csv");

userRecords = engine.ReadFile(dataFilePath);

}

public UserTemplate GetUserDetails(string requiredUserKey)

{

ReadFileIntoDataRepository();

var selectedUser = userRecords.FirstOrDefault(user => user.UserKey == requiredUserKey);

if (selectedUser == null)

throw new ArgumentNullException(@"User", new Exception($"User {requiredUserKey} not found!"));

return selectedUser;

}

public UserTemplate GetUserDetailsByEmail(string requiredEmail)

{

ReadFileIntoDataRepository();

var selectedUser = userRecords.FirstOrDefault(user => user.Username == requiredEmail);

if (selectedUser == null)

throw new ArgumentNullException(@"User", new Exception($"User {requiredEmail} not found!"));

return selectedUser;

}

public string GetFilePath()

{

var dir = new DirectoryInfo(AppContext.BaseDirectory);

var solutionPath = dir.Parent.Parent.FullName;

var finalPath = $@"{solutionPath}\Data\";

return finalPath;

}

}

In your Feature, change the LoginSteps to login using an email or a userkey

*@mytag*

Scenario Outline: LoginTwo Email

Given I am on the Login Page

When I login as user with email "*<Email>*"

Then I should be logged in successfully

Examples:

| *Email* |

| cs@portal.com |

*@mytag*

Scenario Outline: LoginTwo Userkey

Given I am on the Login Page

When I login as user "*<UserKey>*"

Then I should be logged in successfully

Examples:

| *UserKey* |

| CsaUser |

In your Steps, create steps for amended feature to get the user details.

private readonly TestDataRepository \_theRepository;

public LoginSteps()

{

\_theLoginPage = new LoginPage();

\_theWelcomePage = new WelcomePage();

\_theRepository = new TestDataRepository();

}

[When(@"I login as user with email ""(.\*)""")]

public void WhenILoginAsUserWithEmail(string email)

{

var user = \_theRepository.GetUserDetailsByEmail(email);

\_theLoginPage.EnterEmail(user.Username);

\_theLoginPage.EnterPassword(user.Password);

\_theLoginPage.ClickLoginButton();

}

[When(@"I login as user ""(.\*)""")]

public void WhenILoginAsUser(string userKey)

{

var user = \_theRepository.GetUserDetails(userKey);

\_theLoginPage.EnterEmail(user.Username);

\_theLoginPage.EnterPassword(user.Password);

\_theLoginPage.ClickLoginButton();

}

## Partial BasePage and Extra Methods

If your BasePage file becomes large, it can become unwieldy. One solution for this is to split it into Partial Classes.

To do this, create a BasePage.cs page as normal.

Then make it a partial class, simply by adding partial before the word ‘class’.

public partial class BasePage

Your constructor will be in this class, it is not added to any of the other partial classes.

Create another class with a name of the form BasePage.NewClass.cs, e.g. BasePage.Clicks.cs

The class will also be called public partial class BasePage

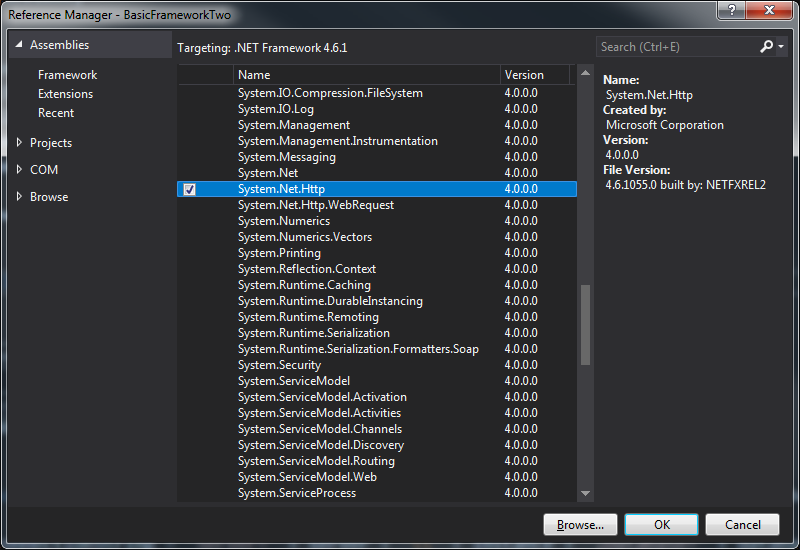
Add required methods to your partial BasePage classes. At compile time, they will all be in one BasePage class.

## Tips for individual BasePage Classes

When adding a class that uses System.Net.Http, you will need to add the reference.

From Solution Explorer, right click on References and click on Add Reference. In the Assemblies section, find System.Net.Http.

Tick it and click on OK.



When adding a class that uses System.IO.Compression, you will need to add the reference, in the same way as above. One oddity though is that you actually have to add System.IO.Compression.FileSystem.

For methods that read a PDF file, you will need to include this NuGet package.

(Right click on project, click on Manage Nuget Packages. Select Browse tab.)

