



CPA Automation User guide

|  |  |
| --- | --- |
| Client | CPA |
| Version | 0.1 |
| Status | Client Review |
| Created by | paul.mateos@cpaaustralia.com.au |
| Date | 20/05/2015 |

# Table of Contents

[Table of Contents 2](#_Toc419966294)

[1 Document Control 3](#_Toc419966295)

[1.1 Contact Details 3](#_Toc419966296)

[1.2 Document History 3](#_Toc419966297)

[1.3 Audience 3](#_Toc419966298)

[2 Overview 4](#_Toc419966299)

[2.1 Implementation 4](#_Toc419966300)

[2.2 Demonstration 4](#_Toc419966301)

[2.3 Architecture 4](#_Toc419966302)

[3 Selenium WebDriver 5](#_Toc419966303)

[4 The Planit Framework 7](#_Toc419966304)

[4.1 Page Object Pattern 7](#_Toc419966305)

[4.2 ActionWords 7](#_Toc419966306)

[4.3 Property.Settings 8](#_Toc419966307)

[5 Lab Center (Lab Manager) 10](#_Toc419966308)

[6 Build Definitions 11](#_Toc419966309)

[6.1 CustomLabTemplate.xaml 11](#_Toc419966310)

[6.2 DefaultTemplate.xaml 12](#_Toc419966311)

[6.3 CustomTemplate.xaml 13](#_Toc419966312)

[6.4 Running Build Definitions 13](#_Toc419966313)

[7 MSTest and MTM 18](#_Toc419966314)

[7.1 Running tests from VS 18](#_Toc419966315)

[7.2 Associating Tests with Work Items 18](#_Toc419966316)

[7.3 Managing Tests in MTM 19](#_Toc419966317)

[7.4 Running Automated Tests from MTM 20](#_Toc419966318)

[7.5 Test Runs 21](#_Toc419966319)

[8 SpecFlow 22](#_Toc419966320)

[8.1 Installation 22](#_Toc419966321)

[8.2 Feature files 22](#_Toc419966322)

[8.3 Steps 23](#_Toc419966323)

[8.4 Wiring in Unit Test Attributes 23](#_Toc419966324)

# Document Control

## Contact Details

If you have any queries regarding the information in this document, please contact:

|  |  |
| --- | --- |
| Contact | Paul Mateos |
| Phone | 0438593423 |
| Email | pmateos@planit.com.au |

## Document History

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Author | Comments |
| 19/5/2015 | 0.1 | Paul Mateos | Draft |
|  |  |  |  |
|  |  |  |  |

## Audience

|  |  |  |
| --- | --- | --- |
| Name | Company | Role |
| Adam Gernon | CPA | Portfolio Delivery Manager |
| Benjamin Logan | CPA | Development Team Leader |
| Ray Lynch | CPA | Test Team Leader |
| Philip Ross | CPA | Project Manager - Digital |
| Alex Edwards | Planit | Director of Technical Testing Services |
| Brodie Court | Planit | Account Director |

# Overview

## Implementation

Planit have implemented a test automation framework for the purposes of automated UI testing. The tools that have been chosen to demonstrate the capabilities are;

* Selenium
* MSTest
* TFSBuild (with custom build activities)
* MTM
* SpecFlow

## Demonstration

A demonstration of the framework is to be presented to highlight the solutions ability to;

* Create automated tests through Visual Studios using C# Selenium and SpecFlow.
* Managing MTM test suites and running them through Lab Center environments
* Creating, Maintaining and running Build Definitions
* Running tests from a command line
* Running tests using SpecFlow

## Architecture

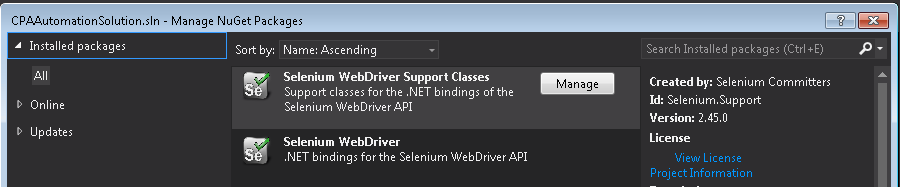
There are several key components which have been integrated to form Planit’s solutions. Installation and configuration of these components have been documented.

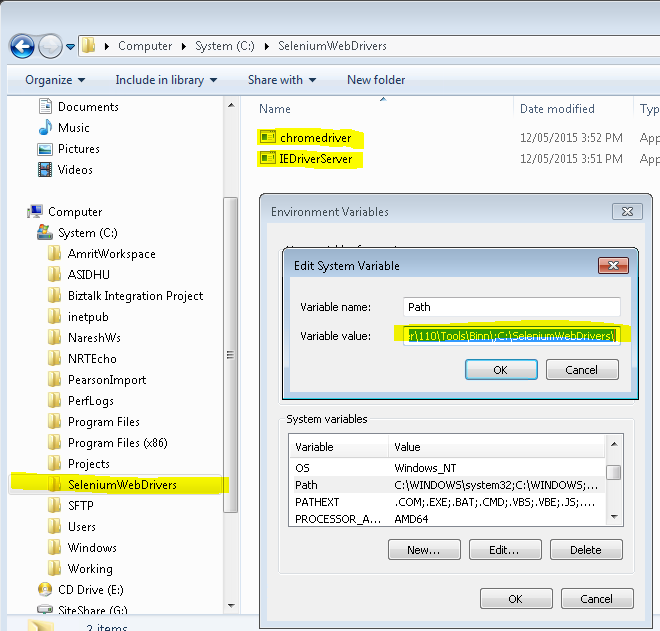
# Selenium WebDriver

This is the test automation framework that is being used to create the automated tests. To use selenium in this solution, there are a few components that need to be installed.

Using the NuGet package manager, locate the Selenium WebDriver and Selenium WebDriver Supporting Classes packages. Install these into your VS project.

From the Selenium Website (<http://www.seleniumhq.org/>) download the appropriate Browser WebDrivers. Both IE Drivers and Chrome Drivers should be available for download. These files should be stored in a publicly accessible location or from a location within the local and remote host machines where the tests will be executed from. This location also needs to be added to the ***PATH*** variable.





**There are known issues with IE11. Documentation has been provided by Selenium and can be referenced here.**

**At this stage there is no new IE WebDriver that has been released to remedy the issues being found. This article provides workarounds that may work in different circumstances.**

[**https://code.google.com/p/selenium/issues/detail?id=6511**](https://code.google.com/p/selenium/issues/detail?id=6511)

[**https://code.google.com/p/selenium/wiki/InternetExplorerDriver#Required\_Configuration**](https://code.google.com/p/selenium/wiki/InternetExplorerDriver#Required_Configuration)

# The Planit Framework

## Page Object Pattern

Within your web app's UI there are areas that your tests interact with. A Page Object simply models these as objects within the test code. This reduces the amount of duplicated code and means that if the UI changes, the fix need only be applied in one place.

This can be thought of as facing in two directions simultaneously. Facing towards the developer of a test, they represent the **services** offered by a particular page. Facing away from the developer, they should be the only thing that has a deep knowledge of the structure of the HTML of a page (or part of a page) It's simplest to think of the methods on a Page Object as offering the "services" that a page offers rather than exposing the details and mechanics of the page.

A common Planit framework example would be;

TestClass\_DoATest.cs: This contains the test method that will be executed by the automation tool.

Public Void DoATest()

{

Homepage homepage = new HomePage();

homepage.ClickbuttonX();

}

These are stored within the Tests folder in the project.

Homepage.cs: This is the page model class that contains the services available for the home page.

Public Void ClickbuttonX()

{

IWebElement element = driver.FindElement(By.Id(“buttonID”));

Elrmrnt.Click();

}

These are stored within the Pages folder in the project.

We also provide a Utilities folder which will contain classes for;

* Maintaining common methods such as GET, CLICK, SET object interactions
* Table or Grid interactions
* Other customised interactions

## ActionWords

A key component to the Planit framework is the ActionWords . This is an approach and an implementation of tools and libraries which have been developed to simplify keyword driven and data driven test automation, and to enable the test automation to be extended to business users and non-technical testers via a simple GUI provided by an Excel Add-In. This is not currently implemented and out of scope during this initial engagement.

## Property.Settings

The use of a project settings file enables the parametrisation of variables that can be used internally by the project and its tests or it can also be used externally to integrate with third party tools such as build servers.

In this engagement, we are using the settings file to control the test environment, browser and wait times.

### Test Environments

At this stage we have hardcoded the test environment URLs within this test class. This can be moved out to a flat file, database connection or even within the property file itself.

public static TestEnvironment GetEnvironment()

{

switch (Properties.Settings.Default.Environment)

{

case EnvironmentType.PreProd:

return new TestEnvironment("http://preprod.cpaaustralia.com.au");

case EnvironmentType.SIT\_BLACK:

return new TestEnvironment("http://black.ext.test.cpaaustralia.com.au/");

case EnvironmentType.SIT\_WHITE:

We have implemented an EnvironmentType and BrowserType enum to limit the list of TestEnvironments and Browsers available.

public enum EnvironmentType

{

[Description("PreProd")]

PreProd = 0,

[Description("SIT\_BLACK")]

SIT\_BLACK = 1,

[Description("SIT\_WHITE")]

SIT\_WHITE = 2,

public enum BrowserType

{

[Description("Firefox")]

Firefox = 0,

[Description("IE")]

IE = 1,

[Description("Chrome")]

Chrome = 2,

}

### BaseTest

A base class is used and inherited by other test classes. This base class contains the initialize and cleanup for each test that is run. The initialise instantiates the type of selenium browser driver to use and navigates to the test environment page.

[TestInitialize]

public static void Initialize()

{

var options = new InternetExplorerOptions()

{

IntroduceInstabilityByIgnoringProtectedModeSettings = true

};

switch(Properties.Settings.Default.Browser)

{

case BrowserType.Firefox:

driver = new FirefoxDriver();

break;

case BrowserType.IE:

driver = new InternetExplorerDriver(options);

break;

case BrowserType.Chrome:

driver = new ChromeDriver();

break;

default:

throw new ArgumentException("Browser Type Invalid");

}

driver.Navigate().GoToUrl(TestEnvironment.GetEnvironment().Url);

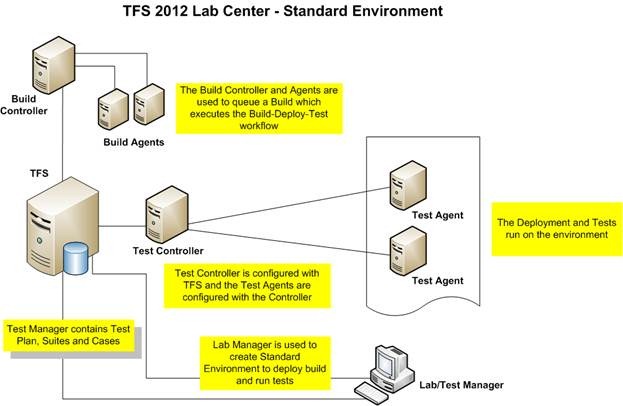
driver.Manage().Window.Maximize();

}

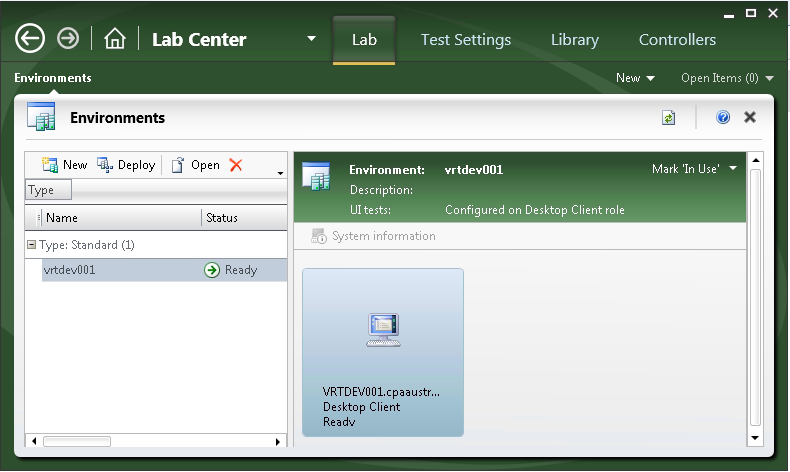
The TestCleanUp method ensures that the browser is closed after each test and also ensured that any remaining unwanted processes are also closed.

# Lab Center (Lab Manager)

Lab Center is used to configure the test environments that the automated tests will run on. These environments are machines (can be VMs) that have the Test Agent Installed on them. A Test Controller is a service that is installed on another machine that manages the Test Agents.

[](http://www.google.com.au/url?sa=i&rct=j&q=&esrc=s&frm=1&source=images&cd=&cad=rja&uact=8&ved=0CAcQjRw&url=http://blogs.msdn.com/b/mvpawardprogram/archive/2012/09/04/automated-build-deploy-test-using-tfs-2012.aspx&ei=PN5aVbWsD-jAmAXDz4CIDA&bvm=bv.93564037,d.dGY&psig=AFQjCNEw8db54glJPM_OrdCfqjXIPsfRIQ&ust=1432104887481247)

Within the Lab Center, the testing environment can be configured.



A more detailed guide can be found here. <https://msdn.microsoft.com/en-us/library/dd936084.aspx>

# Build Definitions

Team Foundation Build uses build definitions for building a solution or set of solutions in a distributed environment. Build definitions provide reports, enable testing, and provide other custom settings which are covered in the section below.

<https://msdn.microsoft.com/en-us/library/ms181709(v=vs.90).aspx>

For this solution build definitions are used to trigger CI tests and can also be used as the starting point to run automated tests through VS or a command line.

There are two types of build definitions we are using in this solution. They **MUST** be run together to successfully execute automated tests. The order in which they are run will depend on certain scenarios that will be described below.

Browser variable of “Chrome”. Two child builds will then be kicked off to run. “PVTAutomatedTestBuild” and “RegressionAutomatedTestBuild”

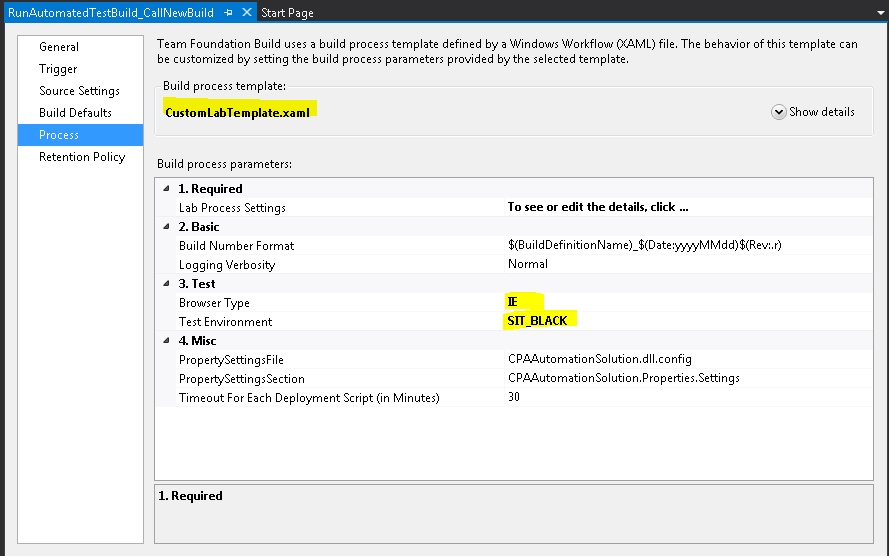
## CustomLabTemplate.xaml

This is derived from LabDefaultTemplate.xaml which is an out of the box template which orchestrates the windows workflow task related to executing automated tests in Lab Manager.

We have customised this file to accept arguments for a) Test Browser Type, b) Test Environment.

This build definition needs to be configures so that the dependant build definition is assigned within the Lab Process Settings.

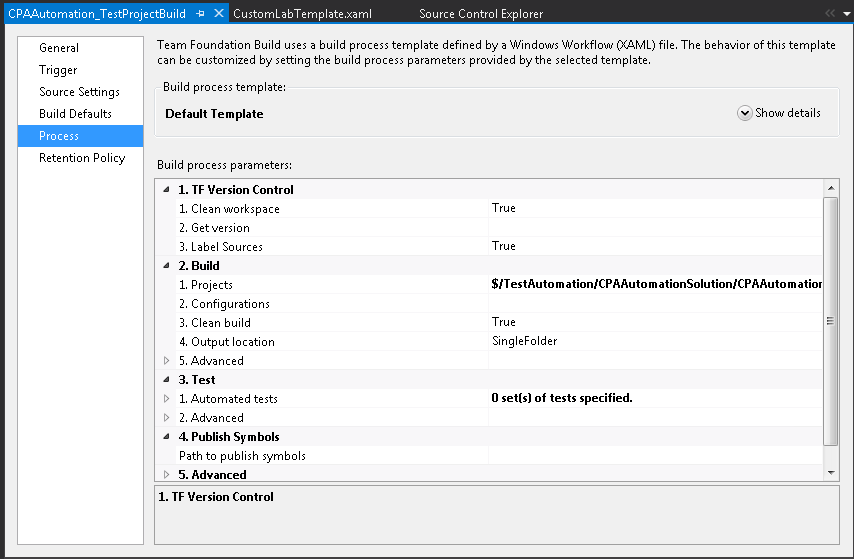
The dependant build will create the test script container(dll).



The purpose of this build definition is to kick off a dependant test project build and once successful, run automated tests within MTM against a Lab Environment.

## DefaultTemplate.xaml

This build definition is an out of the box definition used to compile a test project containing the automated test scripts and deploy the container to a drop location. It will be run as part of the CustomLabTemplate.xaml definition.



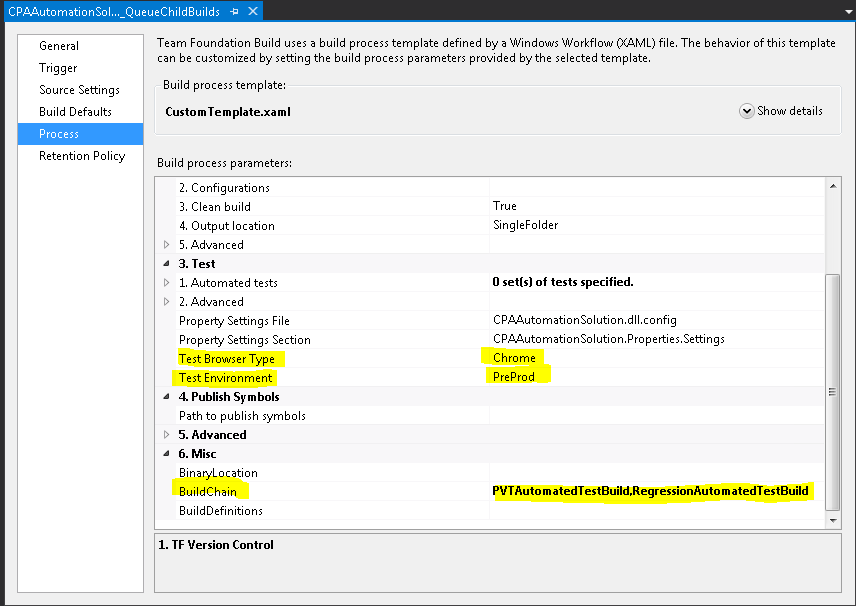
## CustomTemplate.xaml

***This definition is not being used for this engagement but has been made available for future planning.***

This is derived from the DefaultTemplate.

We have customised this file to accept arguments for a) Test Browser Type, b) Test Environment c) Child builds that are to be run after this build is completed

The purpose of this definition is to build the test project containing all the automated test scripts and related files into a container and have it available for MSTest to run. The build definition then updates the project.config file with the arguments that have been passed into the build definition.



The image above describes the arguments being passed into the definition. After the build has completed, the project.settings file will be updated with an Environment variable of “PreProd” and a Browser variable of “Chome”

## Running Build Definitions

The solution has been developed to be flexible enough to handle future requirements regarding application deployment. The process we will be demonstrating will allow a user to create/use a build definition that will run tests that have been organised within a specific MTM test suite.

This build will begin by kicking off a test project build which will build all the automated test case scripts into a container(dll). This container will be deployed to a location and made available for MTM to use so that it can run the required automated tests which are associated to the test cases in the MTM test suites.

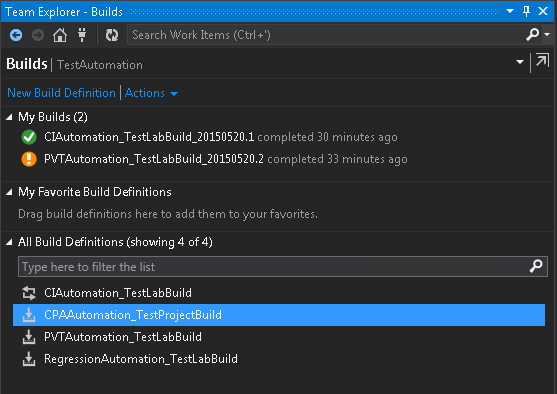
### Available Build Definitions;

* + These are used to run test cases that have been organised within the Regression Tests test suite in MTM
    - CIAutomation\_TestLabBuild
    - PVTAutomation\_TestLabBuild
    - RegressionAutomation\_TestLabBuild
  + This is use to build the test project containing all the automated test scripts and deploy them to a drop location. This is run from within one of the TestLabBuilds above.
    - CPAAustralia\_TestProjectBuild

### Ways to Run

#### From Visual Studios

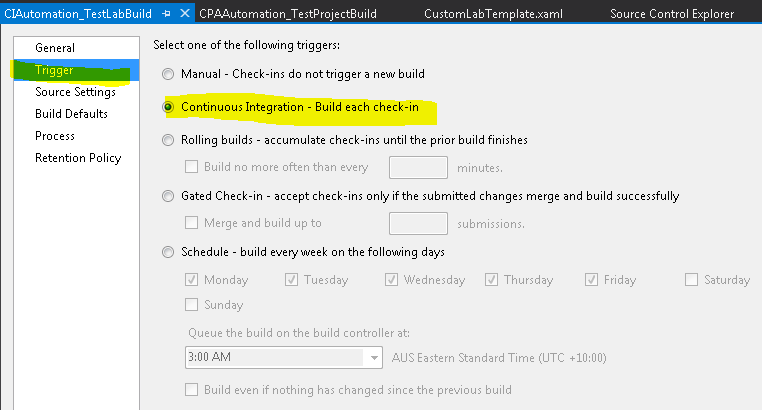
Build definitions can be viewed through the Build Page within a TFS Team Project.



They can be run simply by right clicking on the desired definition and clicking on “Queue New build…”

#### From a Project Checkin

The build definition can be configures so that it is queued when a project is checked into source control.



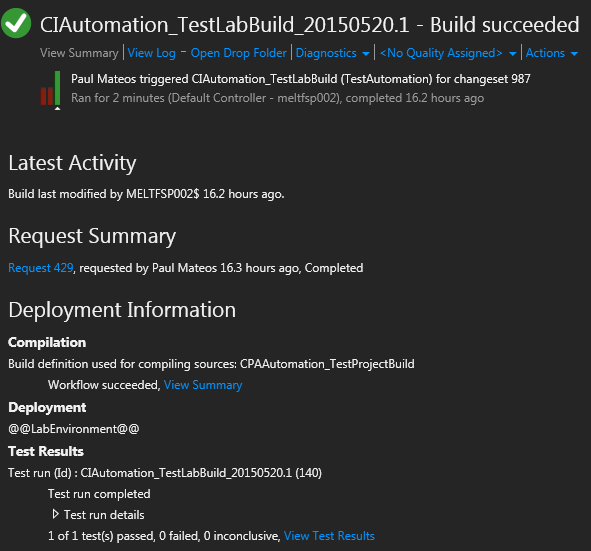
#### From a Command Line

TFSBuild is a utility that is available from the Visual Studios Command console.

### When Builds Pass and Fail

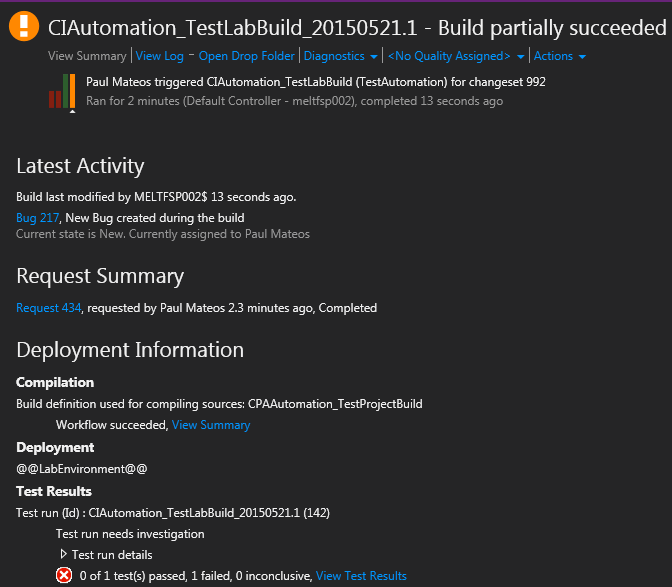
A build which has passed will display a status of “Build secceeded”. The result of a successful build should be;

* The test project has been deployed to a drop location ready to be used for test execution.
* The properties.config file has been updated with the browser and environment arguments passed in through the build definition
* The required test suite has been run and the results of the test run are accessible through TFS/MTM



A build can fail for several reasons. Some of these may be;

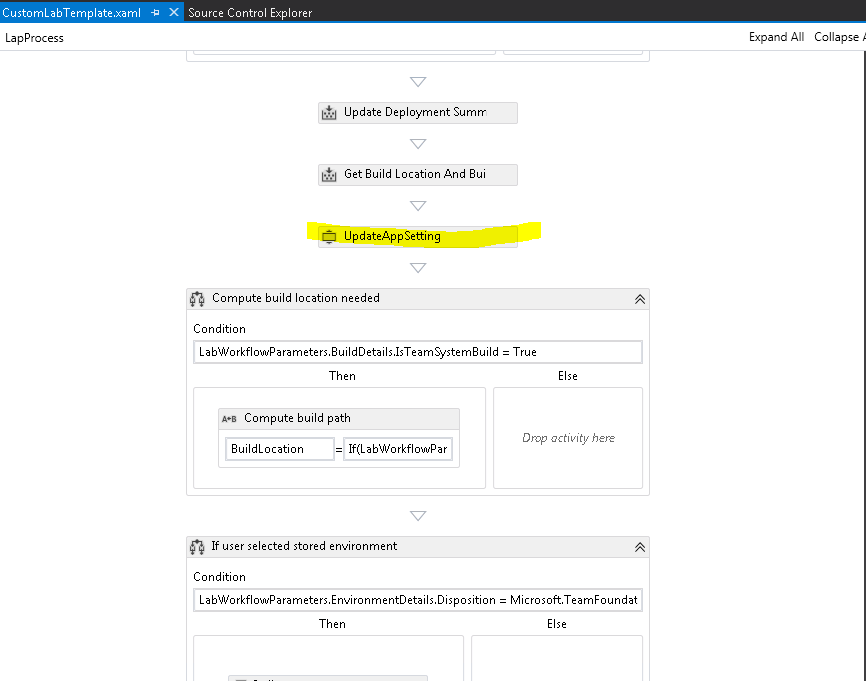
* The test project has failed to build due to issues with the project code or libraries and resource files are missing.
* Automated tests have failed which are reported within the build mussary report.



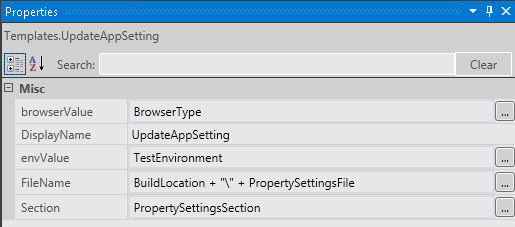
## Customising Build Activities and Templates

To create/maintain build activities and templates, there is a separate project called BuildProcessSource. This project is a Workflow Activity Library project which contains a Code Activity C# Item.

There are two xampl files within this project. One is the CustomTemplate.xaml and the other is CustomLabTemplate.xaml. They can be edited either via code or via the workflow designer.

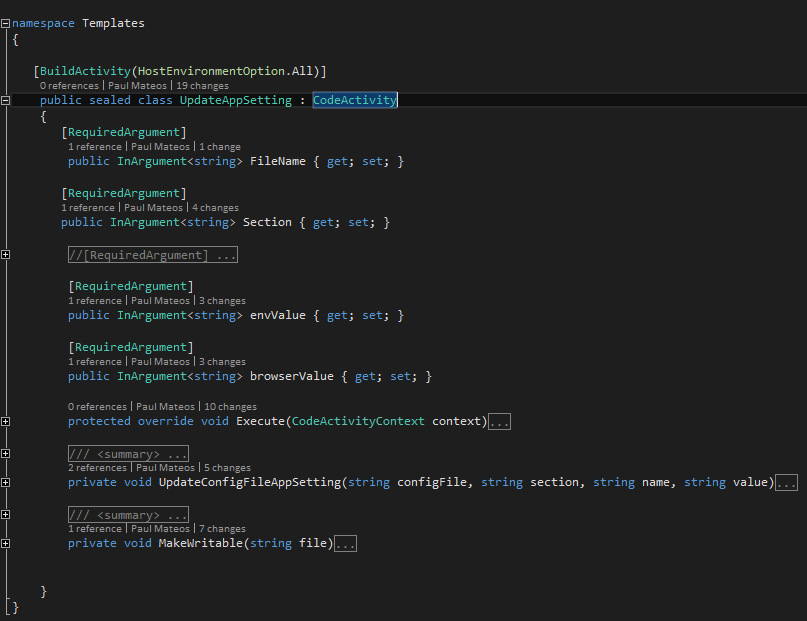


The image above shows the CustomeLabTemplate in designer view. Highlighted is the custom code activity that has been created to update the process.config file during the build process.

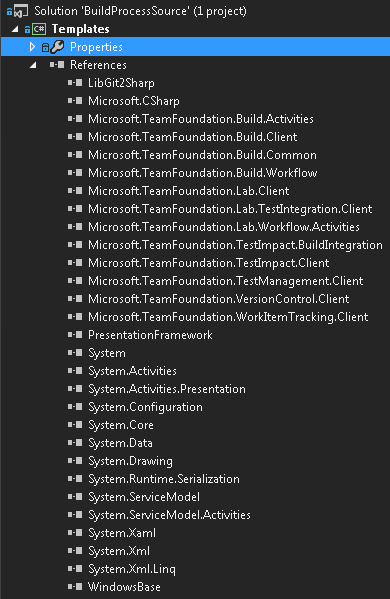


Above is an image of the UpdateAppSettings workflow activity and the arguments that are needed to be passed in.

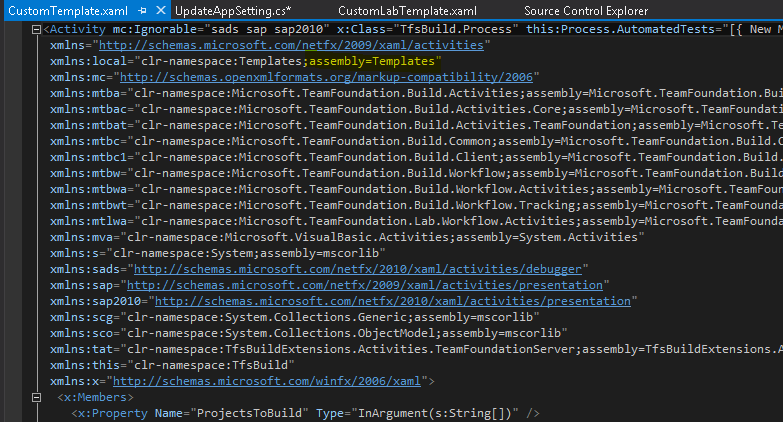
All workflow items can be added from the ToolBox. The toolbox requires the DLL file that is created when the BuildProcesSource project is compiled.



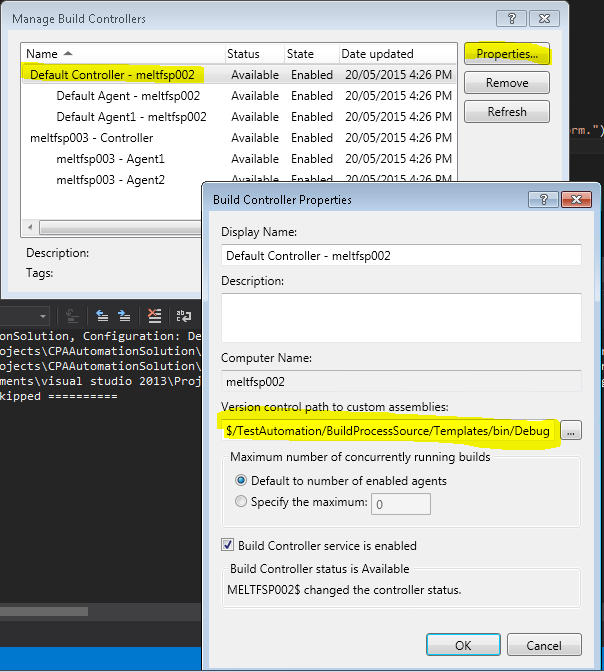
This is the code activity that is used to update the property.config file.



The required libraries have to be added to the reference.



**GOTCHA:** This highlighted code above has to be added after working in the workflow designer for this solution to build correctly. For some reason, the designer removed this crucial bit of code.



When you have completed working on the xaml templates and code activities, the project needs to be compiled and checked in to TFS **(including the debug\bin folder in this case)**. The above image shows how to make the new templates accessible to the build definitions.

In a real world scenario, the project Dll and Pdb file along with the xaml files need to be copied out of the project bin folder and saved to a location where they can be used throughout the organisation. Probably in a location where other customised build processes are stored.

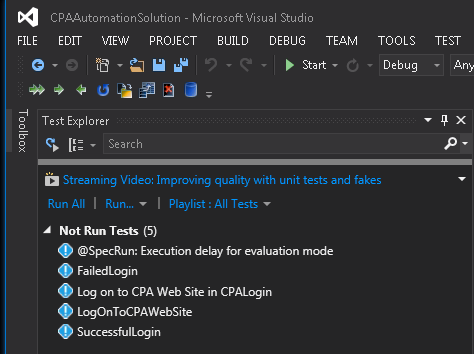
# MSTest and MTM

Essentially, the solution is using MSTest as the execution tool that runs unit tests. Selenium is wired within the standard Microsoft unit test template so that the tests are recognised by MSTest as regular Microsoft unit tests. Having done this, all the tools available throughout the TFS suite which manage unit tests can be utilised. MSTest can also be run independently as a standalone application.

A Visual Studios test project will compile into a container that MSTest then uses. It is this dll that will also be used to run test from MTM.

## Running tests from VS

Once a test project is compiled, VS should recognise the tests as Microsoft unit tests. The text Explorer window should display a list of tests that have been discovered for the project.



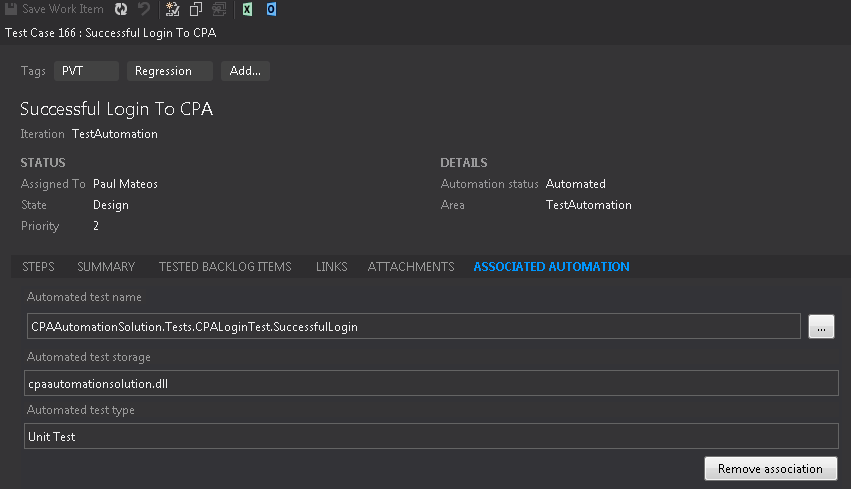
Simply by right clicking on any of or a selection of the above tests and choosing “Run Selected Tests” will run the test locally.

“Debug Selected Tests” will run the tests in debug mode which is used when there are issues found with the test case.

## Associating Tests with Work Items

To run unit tests from MTM they must be associated to a test case work item within TFS. This can either be an existing or new work item with a work item type as “Test Case”. Associations can only be made through VS by opening the work item and navigating to the “ASSOCIATED AUTOMATION” tab. Clicking on the  button for “Automated test name”.

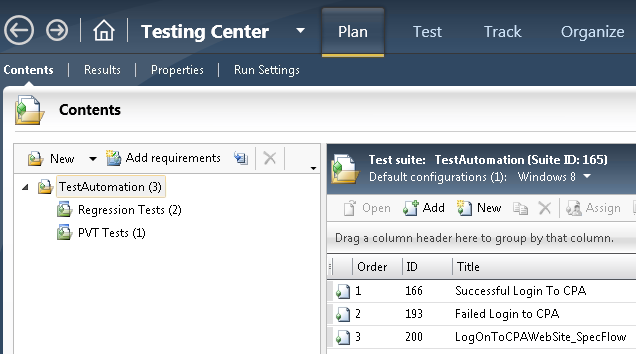
Select the unit test you would like to associate with this work item.



This test case can now be used within MTM.

## Managing Tests in MTM

Managing automated tests in MTM is exactly the same as managing manual test cases. The same process applied in regards to adding test cases to test suites.



The easiest way to locate test cases that have been associated to automation test scripts is to use the “Automation State” value. This should be set to “Automated” while manual test cases that have no association should be set to “Not Automated”.

At this point, automated tests cannot be executed until Lab Center is configured and a build definition is executed to build the unit test container.

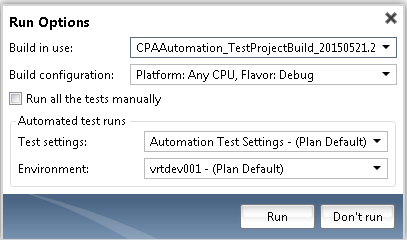
## Running Automated Tests from MTM

TFS Test Work Items can only be run as automated tests if they have been associated to Unit Test Scripts. They are identified by having an Automation State of “Automated”. You will need to add and manage these test cases the same way as you would manual test cases. Create/use a test plan within MTM and add the test cases that you would like to run.

***Note: If you want to run test suites in MTM though the build definition process, you will need to configure a build definition to use the test suite.***

Once your test suite and test cases is ready to run;

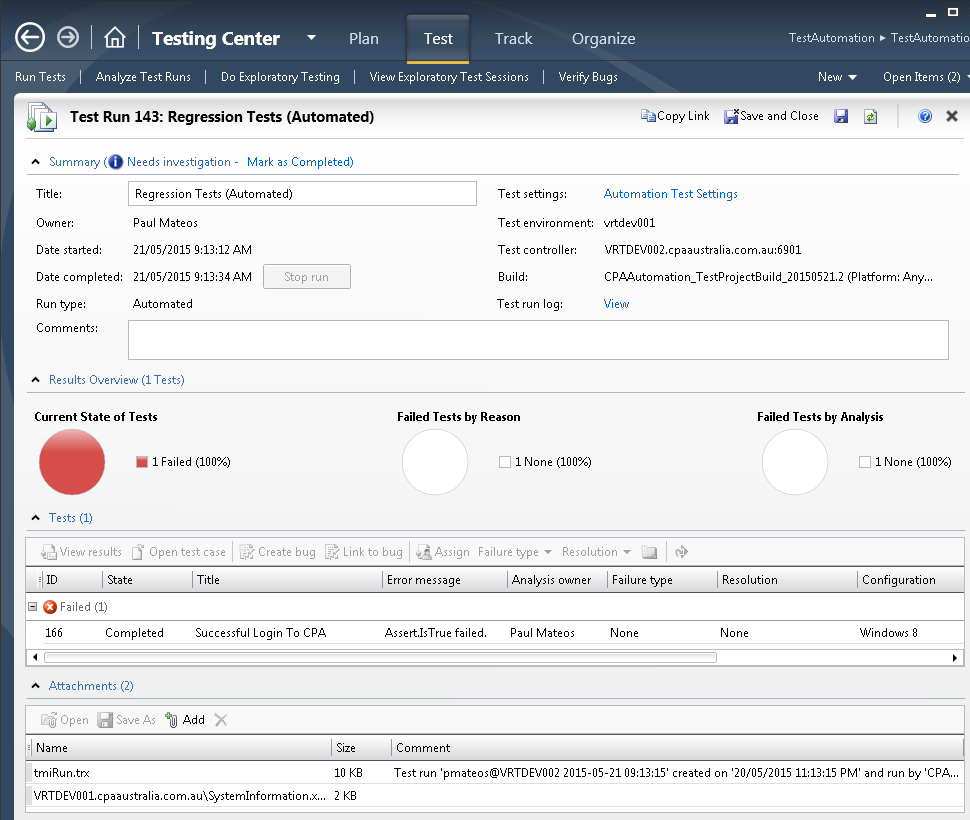
1. Navigate to the Test tab within MTM
2. Right click on the suite or test case that you want to execute.
3. An option should now be presented to you “Run with options”



1. Build in use: represents the most recent build that has been deployed containing the unit test scripts. You can run the tests against the most recent build of the test project or a previous build perhaps related to a version of a code release.
2. Build configuration should be left as default
3. Test settings: There should be a test setting that has been set up within Lab Center for you to use.
4. Environment: should be the Test Agent you want the test to run on. This can be a VM or physical machine.
5. Clicking Run will start running the test pm the Environment selected.
6. All test results will be displayed in the test run screen.

## Test Runs

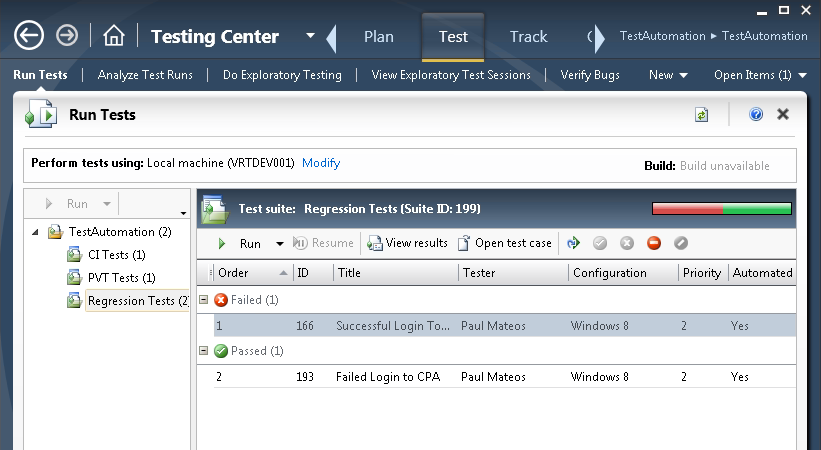
Once a test has completed execution. You will be able to view test results in the Test Run page.



Summary information will provide the information of the execution. When, where and with what configurations and build.

Result overview will give an overview of the tests that have passed and failed and some messages regarding the execution results.

The Attachments section will contain any information regarding the target test agent and also a trx file that can be viewed through visual studios. Information such as screenshots, screen video recordings etc are stored in this file.

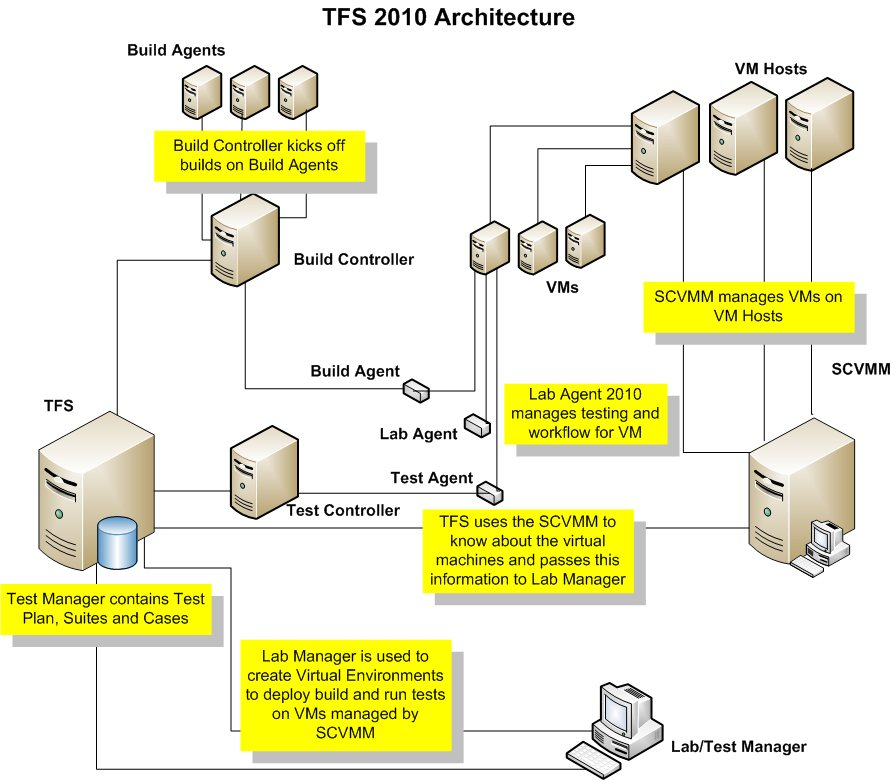


The overall test result can then be viewed through the Run Tests page in MTM.

# Solution Architecture

## TFS ALM

The solution has been build within the a TFS ALM infrastructure.



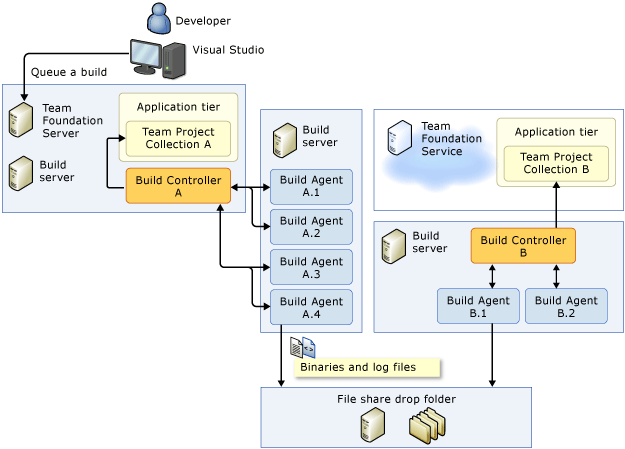
The components specific to Test Automation are;

**Test Controller** - The test controller distributes tests and manages test runs by communicating with the test agents on each test machine. In addition, test controllers enable you to use diagnostic data adapters to collect data or impact the test machine when the tests are run.

**Test Agents** - Each test agent can perform tasks such as installing software, running tests, and collecting test data. They are installed on VMs or physical machines and are managed from within MTM through the Lab Center UI.

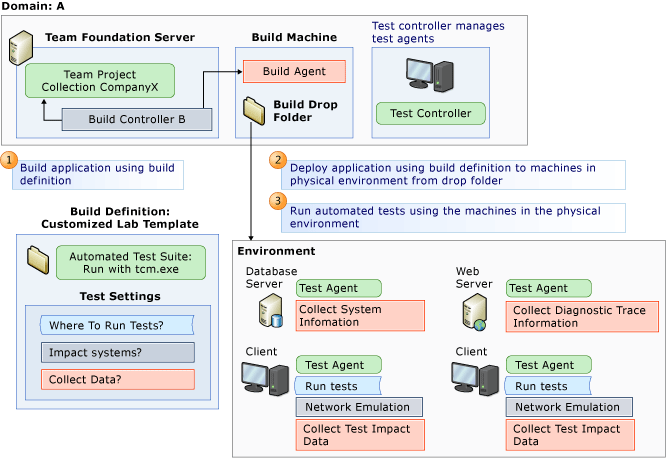
## Build and Deployment.

Building the automation solution is like building any visual studios project. The test cases are created/modified and the code change is checked in to TFS. A new build is queued either through a CI or manual process. The build server will call upon a build agent to complete building the project and will eventually deploy the build binaries and other related files to a drop folder.



## Test Execution

MTM will determine which automated tests will be run through associations with test case work items and automated test scripts. The test will be run againsts environments that have been configured through MTM Lab Center.

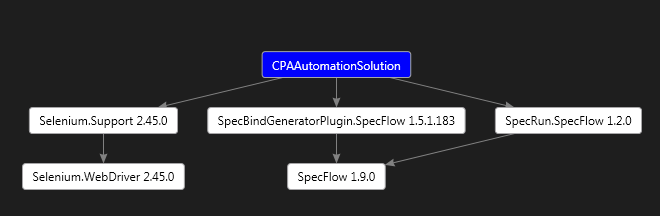


# SpecFlow

SpecFlow aims at **bridging the communication gap between domain experts and developers** by binding **business readable behavior specifications and examples** to the underlying implementation.

## Installation

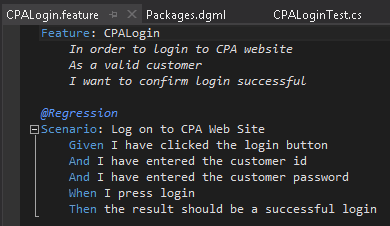
There are some packages that need to be installed in VS.



These packages are available through NuGet Package Manager.

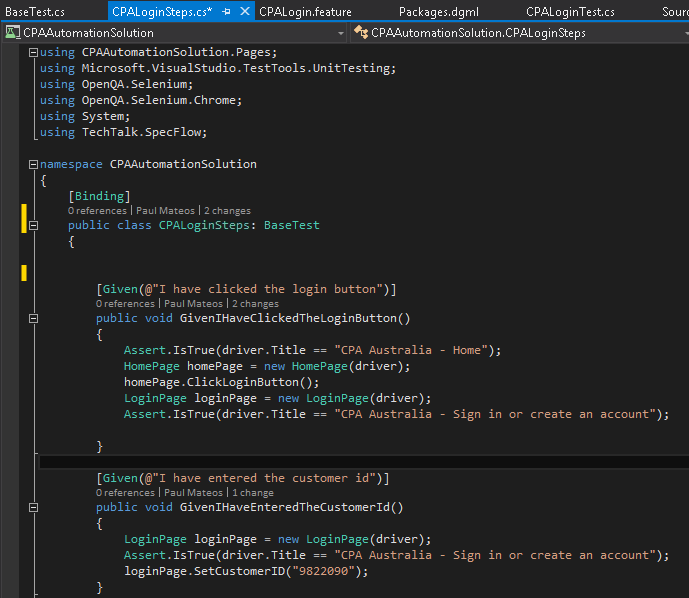
Once installed you should be able to start creating new Feature and Step files.

## Feature files



This is a feature file that we have created that will integrate within the Planit framework. It will reuse the page model methods that are also being used by our unit tests. Right click on any of the scenario lines to navigate to the bound step method.

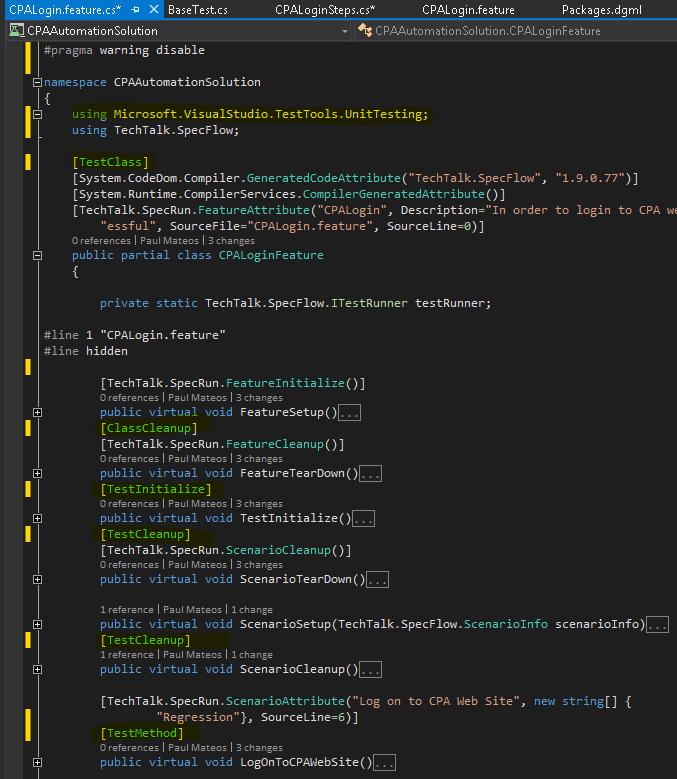
## Steps



You can see from the image above that each line in the feature file is related to a GIVEN, WHEN, THEN SpecFlow attribute. Within these methods is the Selenium code referencing the page model methods.

## Wiring in Unit Test Attributes

To enable the execution of SpecFlow tests through MTM and this Build Definitions, Unit Test attributes must be added to the feature file code.



Once the project is compiles, MSTest should recognize this class as a standard Unit Test class. It will then be available for you to associate with a TFS Test Case work item and run from MTM.