Des McCarter (@BJSS)

des.mccarter@bjss.com

Abstract

A guide for creating robust and reliable test frameworks based on simple testing and development principles within an enterprise environment. This framework (Maxwell) referred to in this document was initially implemented against the HS2 Notice Production engine (by BJSS). This document outlines the reasoning behind its development and can be used as an example on approaching and implementing web based test automation.

MAXWELL test framework REFERENCE AND USAGE GUIDE

A general guide to web based automation testing

Contents

[Introduction 2](#_Toc489267853)

[Framework location 2](#_Toc489267854)

[A typical framework implementation approach 4](#_Toc489267855)

[The *traditional* page factory approach 4](#_Toc489267856)

[Example of the traditional page factory approach - ToolsQA 4](#_Toc489267857)

[Defining pages 5](#_Toc489267858)

[Using page factory definitions (page objects) in tests 5](#_Toc489267859)

[Maxwell Framework approach 7](#_Toc489267860)

[Drawbacks to traditional page factory/management approach 7](#_Toc489267861)

[Page management in Maxwell 7](#_Toc489267862)

[Using page expressions 8](#_Toc489267863)

[Switching between browsers 9](#_Toc489267864)

[Test flow management 10](#_Toc489267865)

[Reasoning behind test flow development 10](#_Toc489267866)

[Test flow in Maxwell – actionfactory 11](#_Toc489267867)

[Defining test flows 12](#_Toc489267868)

[Implementing test flows 14](#_Toc489267869)

[Executing test flows 16](#_Toc489267870)

[Another example using BDD (Specflow) 17](#_Toc489267871)

[K2 specific test and library workarounds 19](#_Toc489267872)

[Introduction 19](#_Toc489267873)

[K2 API 19](#_Toc489267874)

[Handling page crashes 19](#_Toc489267875)

[References 20](#_Toc489267876)

[Framework dependencies 20](#_Toc489267877)

[Software Requirements 20](#_Toc489267878)

[Framework Installation and test set-up 20](#_Toc489267879)

[Installation 20](#_Toc489267880)

[Browser configurations and extras 21](#_Toc489267881)

[Enable “Prompt for username and password” 21](#_Toc489267882)

[Clear login auth cache credentials 22](#_Toc489267883)

[Disable pop-up blocker 24](#_Toc489267884)

[Hs2NpsK2Api Set-up 25](#_Toc489267885)

Named in honour of James Clerk Maxwell



# Introduction

This document should be used both as a technical reference guide to using the Maxwell test framework and also as a general approach to developing an enterprise test framework similar to Maxwell. Information here includes the pitfalls found when developing tests around a highly volatile application such as K2 ([www.k2.org](http://www.k2.org)).

K2 is a .NET application framework which allows developers to develop an application with minimal code. Maxwell was initially developed against an application required by HS2 and was found problematic to test given its intolerant nature both from a functional and performance perspective (both in manual and automated testing). This document includes a mention of these pitfalls as well as their solutions.

The Maxwell framework was initially developed to functionally test a product developed at HS2 by BJSS. The core of the framework comprises of two bespoke libraries: *pageengine* and *actionengine*. Pageengine manages the physical interactions between web based tests and the web page’s themselves. Actionengine was developed to manage the execution flow, usability and re-usability of the tests cases / workflow steps / actions.

## Framework location

The framework can be downloaded from GIT from the following location:

<https://github.com/desmccarter/maxwell>

The complete steps for its installation can be found in the Reference section at the end of this document.

The framework also comes along with sample test cases (based on Google.com and Expedia web pages, the latter for its richness in page elements) and can be found in the actionsamples project: <https://github.com/desmccarter/maxwell/tree/master/all/actionsamples> (bbc/news, google/simplesearch and expedia) folder. Maxwell is flexible, which means that tests can be implemented within any off the shelf framework. With this in mind (and to show this), all examples are either in Speflow (e.g. expedia/bddfeatures) or NUNIT (google/simplesearch/tests) format.

# A typical framework implementation approach

A typical tactical approach normally taken when developing a web based front end framework is to a. use (or build) a page factory library for retrieving page information (i.e. information on the pages being tested) and b. include test execution flow/steps within code (or scripts).

Another approach (which is poor, clumbersome and extremely difficult to manage) is to manage page interactions DIRECTLY in the test itself along with the test flows. The latter I have seen at various organisations (Waitrose for example) and I will only advise as route NOT to take and must be avoided (especially within an enterprise development environment) at all costs IF you require tests to be stable and manageable.

## The *traditional* page factory approach

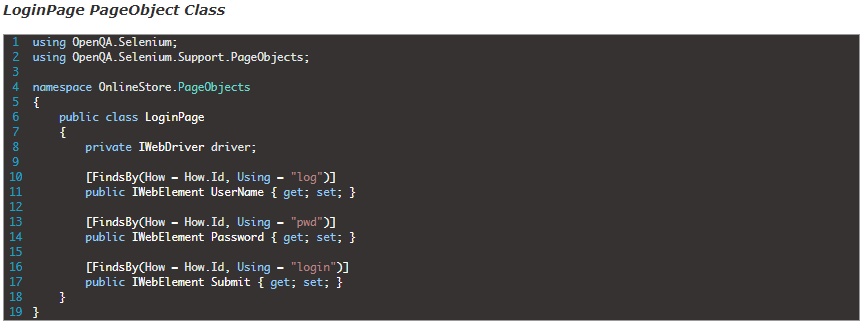
The former is the approach taken in the development of Maxwell but has a slight difference to the traditional page factory implementations. The main difference is the approach taken in how pages (and their elements) are defined. Most page factories define pages in code, where pages are represented by classes. Elements of these pages are represented by properties (or getters/setters) of the class. Using Maxwell, all pages and their elements are defined within an XML file (pages.xml).

## Example of the traditional page factory approach - ToolsQA

Libraries such as the one developed by Tools QA (<http://toolsqa.com/selenium-webdriver/page-object-pattern-model-page-factory/>) use the traditional class approach for defining pages and their elements. This general approach is absolutely fine but most require coders to define and describe pages in code in the form of classes (per page). A page would typically be represented by a class with their buttons, inputs, text boxes etc represented by getters/setters (or properties in .NET).

### Defining pages

Example traditional page representation in code (Tools QA - from <http://toolsqa.com/selenium-webdriver/c-sharp/pagefactory-in-c/>)



The image above illustrates how pages are typically defined and represented. This example is used by ToolsQA, where the page itself (under test) is represented by a class, named LoginPage. This class has properties (getters and setters) which represent each element to be interacted with on the Login page: *login*, *password* and *log*.

### Using page factory definitions (page objects) in tests

Example Tools QA / traditional page object usage

The image below shows how the previously defined page object is used based on the Tools QA page factory. The steps used are as follows:

1. ***Define the driver object (the object which dictates the browser to be used in the test (Firefox in this case).***



1. ***Set the URL property of the driver object to the page under test.***



1. ***Initialise the driver with the page(s) definition by calling InitElements() for each page.***

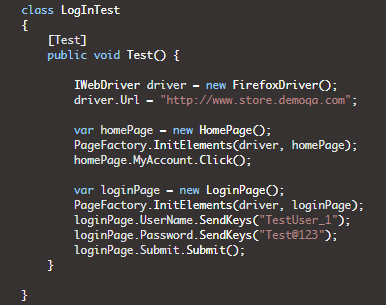


… and then finally …

1. Start interacting (clicking, sending text etc) with the page.



A typical class (in this case, NUNIT based) would look like this:



# Maxwell Framework approach

## Drawbacks to traditional page factory/management approach

Whilst these separation of concerns makes sense (the concerns in this case being page and test execution flow management), in the real world, *HOW* this is implemented is one of the keys to having suitably reliable set of tests. This increasingly becomes important as the number of test cases developed becomes a significant count and even more so if the application being tested is enterprise.

For example, if the xpath location of a certain button has changed then you will need to update the class and rebuild the tests etc to reflect the change. Also, if the xpath location of a certain elements vary between each environment (this has been known to happen) then you will need to branch the tests in source *control PER ENVIRONMENT*!.

## Page management in Maxwell

Pages under test are managed in Maxwell by a. expressing/representing their detail within an XML file and b. using this representation in code. This differs from the typical approach by a. how pages are defined and expressed and b. how pages and drivers are initialised.

Example page expression – representing Expedia.co.uk

The image below shows an example of how pages are defined and expressed in Maxwell. All pages are defined within one XML file. The page under test itself is defined within its own ***Page XML element*** and each page element (i.e. each element on the page which is to be interacted with in the test, e.g. buttons, text boxes etc) are defined as ***Element XML elements*** within the Page XML element.



In the simplest case, each page definition has a name …



… and a URL associated with that name …



The page *Name* element is used within the test to retrieve an instance of that page whereas the URL is used by pageengine to navigate to that page.

### Using page expressions

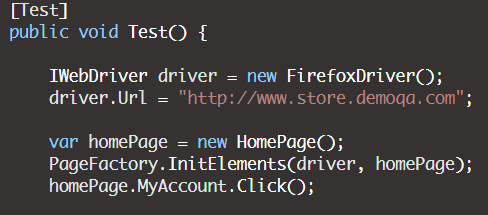
Page expressions are used by a. first getting an instance of this page by calling the OpenPage method (which also implicitly navigates to that page using the selected browser) b. applying actions to the page elements, such as clicking them, reading text, sending text etc. An example of this is shown below using the page ExpediaPage and ExpediaDeparturePage expressions shown earlier:



Note also that any Selenium based functionality is extracted out from the test. For a test you simply want to a. open the page and b. interact with its elements. This is opposed to the normal approach which is to define additional information, such as declaring an instance of the web driver object and initialising an instance of the defined page against it (along with declaring the page URL):

Example of the *“normal”* approach of interacting with

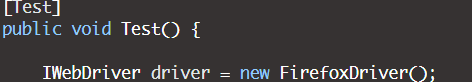
page elements using a traditional page factory



### Switching between browsers

Using the traditional approach, browser drivers are typically defined and initialised within the test itself:

Defining browser (selenium driver) information using typical approach



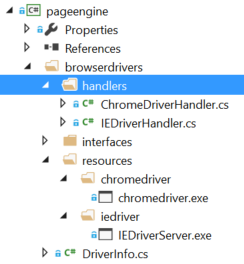
So for every test, in theory, a driver instance will need to be created. In the Maxwell framework browsers are defined (and pointed to) via the tests App.Config file:

Defining browser (selenium driver) information using Maxwell



In this case, all tests will use the browser defined by the ***driver.wrapper.class key*** within the applications App.Config file. In the case shown above the IEDriverHandler class is used. This class handles the initialisation of the specified selenium driver, in this case Internet Explorer (reflected by the *IE* bit of the *IEDriverHandler* class name). Driver wrappers are located within the pageengine project under the browserdrivers/handlers folder:

Browser driver handlers – location in the pageengine project



## Test flow management

### Reasoning behind test flow development

In most cases when dealing with automation it is often necessary to define a formal way in which tests themselves are developed. This ensures that tests are a. better maintained b. quick to automate c. easy to understand d. adaptable and e. re-usable (where nedbe).

**Maintainability** – able to adapt to significant changes with minimal disturbance. For example, a large change in product functionality requiring a significant change to sets of tests. But if these tests are modularised then significant changes in the product being tested should result in minimal changes of the tests.

**Quick to Automate** – The styles or patterns used in tests/automation allows for a. tests to look and appear similar b. newcomers to easily understand how the tests can be automated / are automated.

**Easy to understand** – This leads slightly from the “Quick to automate”, where viewing the implemented tests gives an idea to what is actually being tested. This also ensures quick peer development.

**Adaptable** – any significant changes, including architectural, can easily be adapted to. For example, the product being tested may require changes of API calls from the SOAP protocol to JSON. If the tests DIRECTLY and EXPLICITLY hit these functions in code (and especially if there are a significant number of tests) and a change needs to be done to switch these SOAP calls to JSON then there is significant work required to change the tests. If these calls, however, were made via some sort of façade or services, then you will only need to amend these services and the code actually implemented in the tests should not change (or have very minimal change).

**Re-usable** – For example, if performance testing becomes part of your portfolio then it would probably save time automating performance tests if the components / libraries developed to set-up data for the functional tests can be used (and vice versa).

### Test flow in Maxwell – actionfactory

All test flows are implemented using the **actionfactory** library. This library allows for a. all test flows to be defined in XML b. all test steps (i.e. steps within the flow) to be implemented in a modular fashion, in code (C# .NET).

### Defining test flows

#### Example test flow definition

Here is an example test flow. It contains a project title ***(<Project>),*** author ***(<Author>***) a description ***(<Description>)*** and two test steps (two ***<Action>*** tags). All elements are contained within the root element ***Actions***:

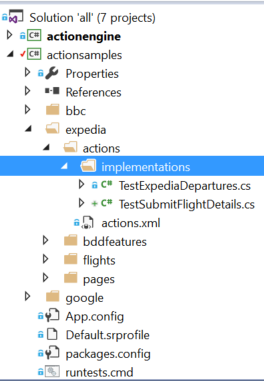


Each action (i.e. a test step) has a unique name and description which describes what the test does. The ACTUAL **test for** **each action is identified by the <Test> tag**. **Each test tag has a** **Name element**, the value of which identifies the test class. For example, in the image above there are two test names: *SubmitFlightDetails* and *ExpediaDepartures*. These refer to class (C# .NET) implementations which have their actual names prefixed with Test.

### Implementing test flows

The SubmitFlightDetails test mentioned previously has its implementation in a class named ***Test****SubmitFlightDetails.* This class can be found under the *expedia/actions/implementations* folder of the *actionsamples* project.

TestExpediaDepatures and TestSubmitFlightDetails



The test code for TestExpediaDepatures looks like this (implemented using the pageengine library:

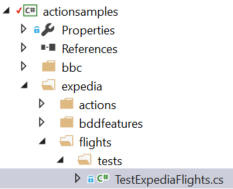
Test code implementation (C# .NET) of TestSubmitFlightDetails



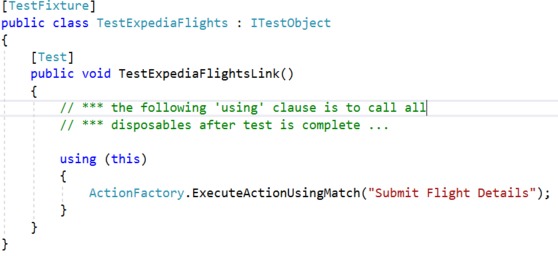
### Executing test flows

All test flows are defined (as previously shown) as a set of actions, with each action triggering a specific test. The action itself is triggered using the ***ActionFactory.ExecuteActionUsingMatch()*** method, which takes a string value which matches exactly with the actions <Match> element:

Location of code example using ActionFactory.ExecuteActionUsingMatch()



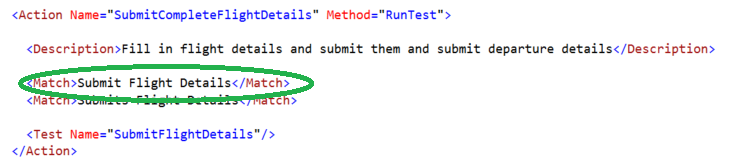
... and the code itself (TestExpediaFlights – NUNIT Example) …



The above code shows an **example of triggering the set of actions for *“Submit Flight Details”***. **The string passed into this method** **MUST match with the *<Match*>** element of the action (<Action>) XML tag, highlighted in green below:

The string parameter passed into ExecuteActionUsingMatch() must

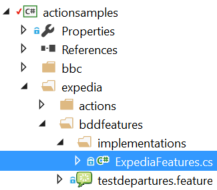
Match exactly with the text contained in the <Match> element.



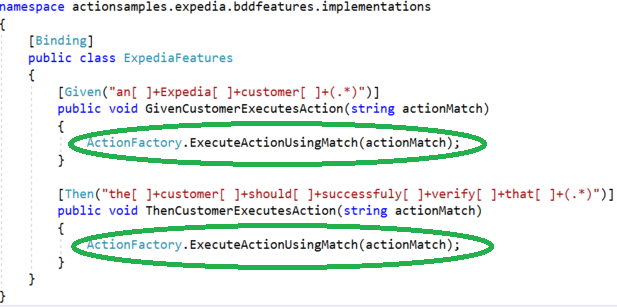
### Another example using BDD (Specflow)

Another example of triggering actions but this time using using Specflow. In most cases any other functional framework can be used (for example Fitnesse):

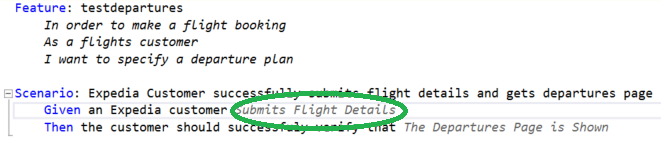
Location of Specflow sample test cases



Specflow sample code triggering ExecuteActionUsingMatch()



The BDD which triggers the above specflow matches



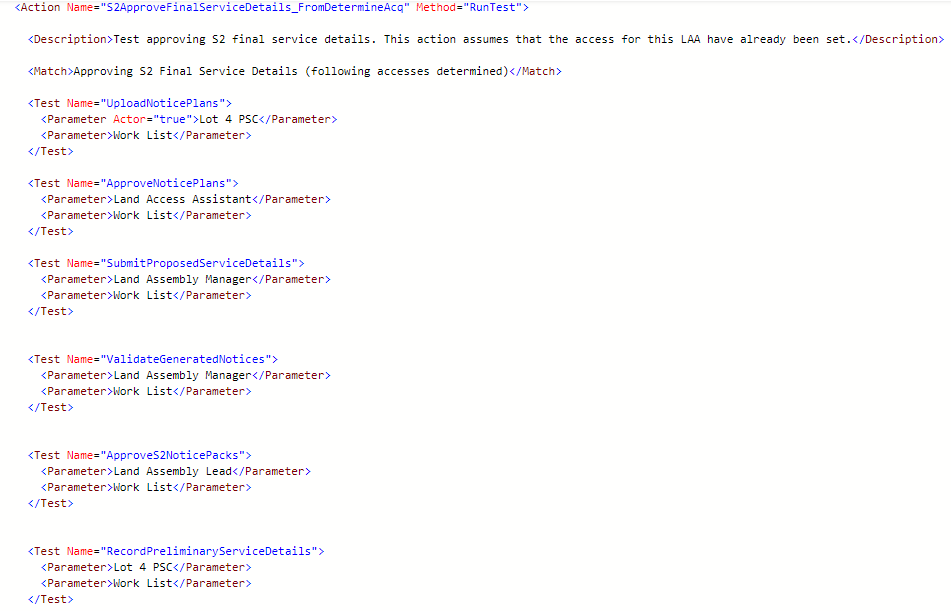
Note that the text highlighted in green above matches the *<Match>* element of the *SubmitFlightdetails* action. This means that this BDD is configurable to call other actions, by replacing the text highlighted in green (“Submit Flight Details”) with a <Match> of another action.

### Assigning multiple test steps to an Action

In the previous action we have exactly one test being called on a specific action. Actions can also trigger multiple tests in sequence. This defines the test flow of the action.

Sample action triggering multiple test steps in sequence

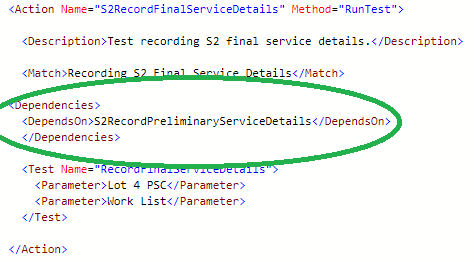
(taken from HS2 Notice Production tests)



### Triggering dependent actions

If actions are dependent on a previous action being triggered, then use the <Dependencies> tag. This tag contains multiple <DependsOn> tags, the text of which should refer to the name of the action. Using this tag, actionfactory will trigger the dependent actions in the same order in which they are defined:

Triggering a dependent action

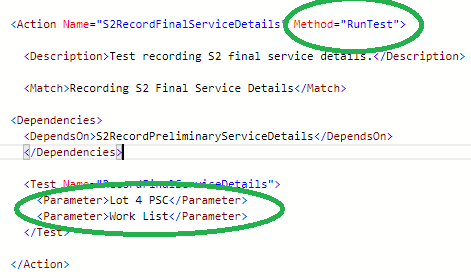


In the above case, calling the S2RecordFinalServiceDetails action will first trigger the S2RecordPreliminaryServiceDetails action first. Failure on the dependent action (exception) will cause the parent action not to be triggered.

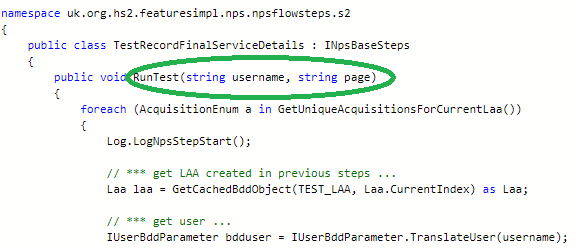
### Using parameters in tests

Test XML elements can also take parameters, depending on whether the test itself takes parameters. All tests must have a method that is defined in the Method attribute of the Action element. In the example used below (taken from the HS2 Notice Production tests) the method is named RunTest and arguments are passed into the actual test:

Defining test method and parameters within an action: RunTest()

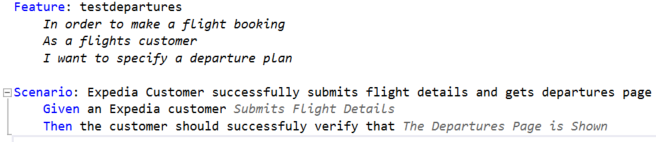


Implementation of test method: RunTest(param1, param2)



### Sharing and passing data between test steps

Most test steps require state information in order to carry out specific test actions. A test flow may include two or more tests, where test steps within the test flow depend on information on previous test steps. An example of this can be shown in the Expedia (Specflow based) test case, the BDD of which looks like this:



The BDD shown above trigger two actions: “Submits Flight Details” and “The Departures Page is Shown” highlighted below.



The first action fires up the Expedia home page but the second action requires the previous page to have information submitted on flight details (name, age, number of children etc). Here’s what the test for the “Submit Flight Details” looks like:



This test code opens the expedia.co.uk homepage, clicks on the Flights link, enters flight details and hits the search button:

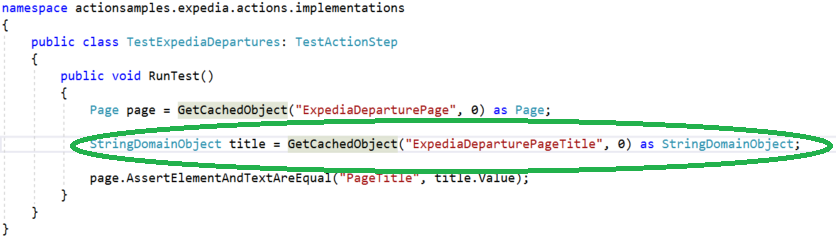


After the search button is clicked ( *page.DoubleClick(“SearchButton”)* ) a second page is rendered, the Departure page, the heading of which reflects the name of the airport being proposed as a destination.

In this case the departure aiport is “Melbourne Victoria Airport”. The second test case /action needs to know which destination was selected in the previous step/action. The first test case therefore needs to save or cache this information so that the second (and subsequent steps if necessary) can use it. **This is done (cached/saved) by calling the *CacheObject()* method**:



This step stores the title of the departures page in the cache, saving it as a string named *ExpidiaDeparturePageTile*. The second test then calls the the GetCachedObject() method to retrieve this string and make an assertion on it:



# K2 specific test and library workarounds

## Introduction

This section specifically details the issues discovered around automating tests against dynamic web pages. These are pages that contain elements that are updated regularly in real time. In simple cases most selenium interactions can check the document.

## K2 API

## Handling page crashes

# References

## Framework dependencies

### Software Requirements

|  |  |  |
| --- | --- | --- |
| **Software** | **Version** | **Installation** |
| **Visual Studio 2015 Community Edition** | 14.0.25431.01 Update 3 | https://www.visualstudio.com/downloads/ |
| **Specflow** | 2.2.0 | NUGET |
| **Specflow.Runner** | 1.6.0 | NUGET |
| **SpecRun.Specflow** | 1.6.0 | NUGET |
| **Selenium.WebDriver** | 3.4.0 | NUGET |
| **WebDriver.ChromeDriver.win32** | 2.30.0 | NUGET |
| **Selenium.Support** | 3.4.0 | NUGET |
| **NUNIT** | 3.7.1 | NUGET |

## Framework Installation and test set-up

### Installation

#### Visual Studio Community Edition 2015+

Download and install Visual Studio Community Edition from the URL specified within Software Requirements.

#### NUGET Packages

Following the installation of Visual Studio, install all packages using the NUGET package installer: *Tools -> NuGet Package Manager -> Manage NuGet Packages for Solution …*

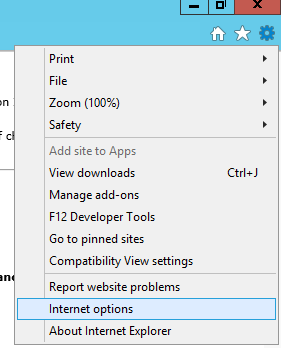
#### GIT Client

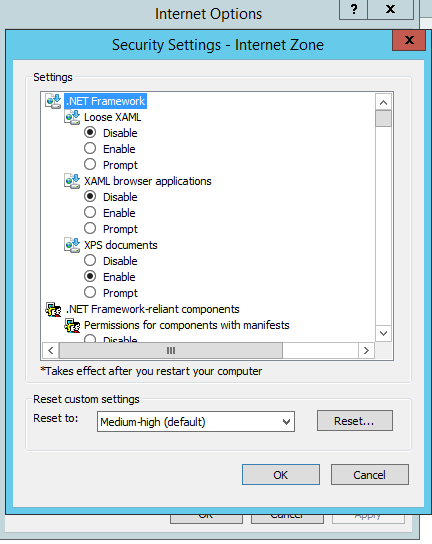
The Maxwell Framework is source controlled using GIT, which means you will need to install the GIT client locally on your test development machine. The (windows) client itself can be downloaded from: <https://git-scm.com/download/win>.

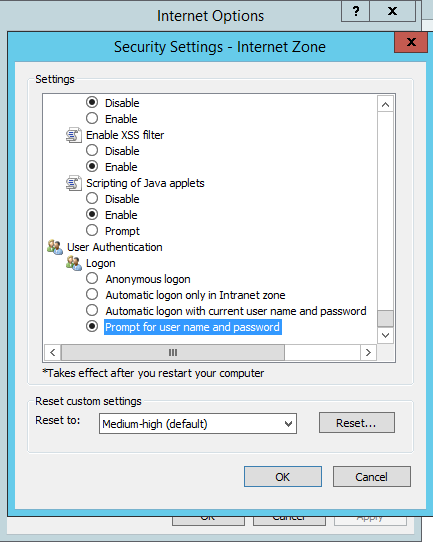
## Browser configurations and extras

### Enable “Prompt for username and password”

Open Internet Explorer Browser and go into Internet Options:





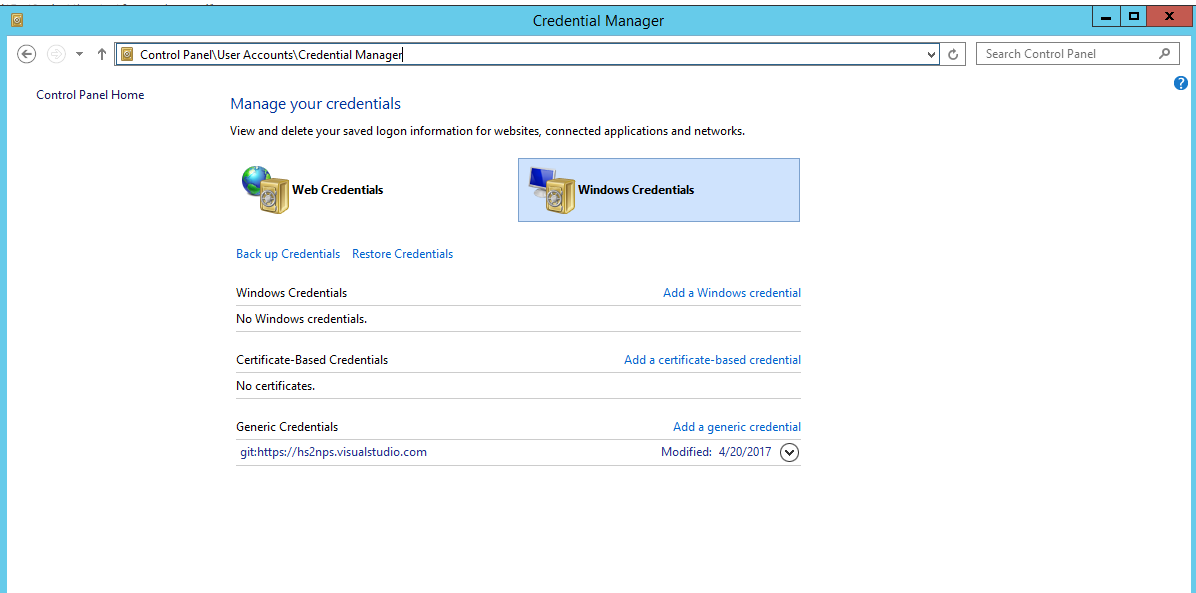


Repeat for ALL other zones.

### Clear login auth cache credentials

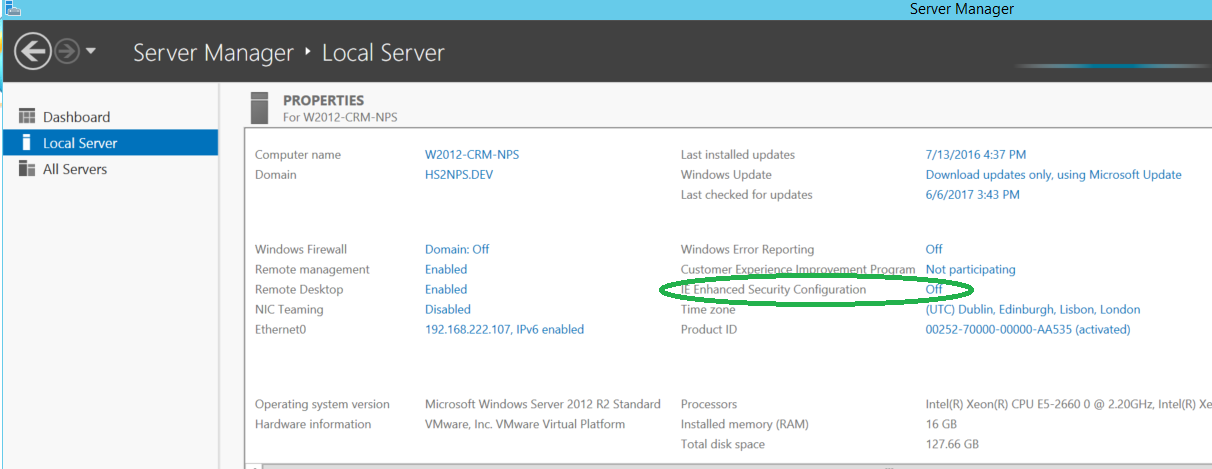
This allows for the web security login cache to NOT have default login credentials, which is problematic when using the selenium alert functionality.

Control Panel\User Accounts\Credential Manager

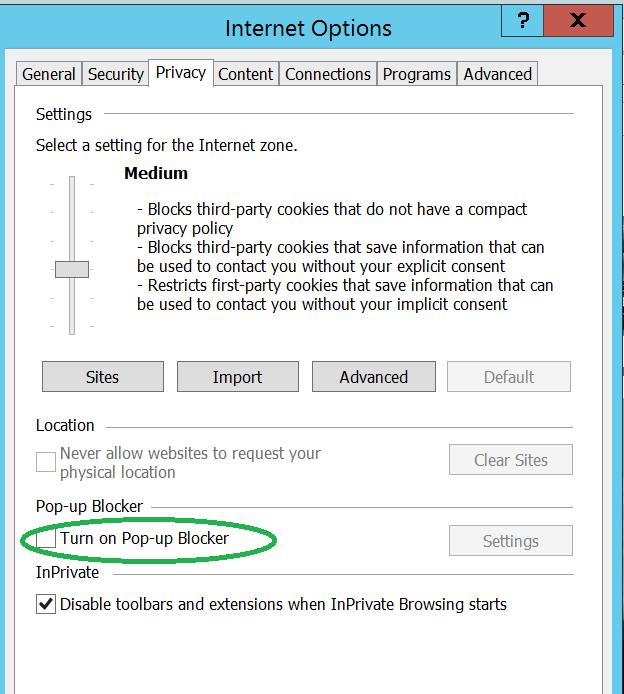


Disable windows enhanced security





### Disable pop-up blocker



### Hs2NpsK2Api Set-up

The Hs2K2Api is a set of functionality which was used specifically for testing the HS2 Notice Production system and is separate from Maxwell.

The Hs2NpsK2Api library contains a set of services which are used to directly interact with the API layer of NPS (i.e. the layer behind the its web front-end). This helps with the speed of execution of the tests by avoiding Selenium based calls against NPS for setting up data typically used by specific scenarios.

**GIT Repo/Location**: https://hs2nps.visualstudio.com/NoticeProduction/\_git/Hs2NpsK2Api

**Clone Command**: git clone <https://hs2nps.visualstudio.com/NoticeProduction/_git/Hs2NpsK2Api>