

RSTest

Documentation and Reference Resource

(Proof of concept, Revision 4. Using Selenium Webdriver) **Authored by Kyle Taylor, 2014**



# Preface

This Document is intended to act as a reference and training resource for development and maintenance of the functional testing suite for the RealStew platform.

This document is relevant to Revision 4 of the Selenium Testing Repository

The aim here is to provide a space for the collection and storage of experience/knowledge pertaining to automated web testing in RealStew so that it may be passed to new employees or as reference material outlining coding conventions, design decisions and other important information.

This material should be kept up-to-date and open to be edited by any employee working with the software directly, hopefully becoming a conduit for a collective experience and drive for improvement.

This is **NOT** documentation for the Selenium API. That is widely available and supported by a sizeable community, any queries regarding selenium should be directed through their public channels. Nor is it the be all and end all of design within the company, only a collection of design patterns and agreed upon best practices.





# Contents

[Preface](#h.8b1reoqnyejz)

[Contents](#h.62yh3jn3n5rr)

[Design Notes](#h.6abowlmtdjyk)

[The UI Map](#h.x550pj4evrvi)

[Selecting the best Locator](#h.2om4b1xos7v1)

[Introduction by Example: MasterPage Class](#h.31o16ji5h4no)

[Login(IWebDriver,String,String)](#h.uttsx2kmjri)

[IsLoggedIn(IWebDriver)](#h.29py851j8m93)

[The Art of Waiting:](#h.kqh39uql8of6)

[Waiting for Ajax](#h.v9114uxr8c5x)

[Waiting To Click](#h.2oblddois7a0)

[Click, Check, Repeat](#h.g4mioh4ajc2t)

# 

# 

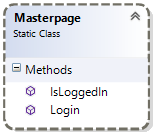
# 





# Design Notes

This section is an introduction to the design decisions implemented in the current revision, specifically the use of a page object model of representing objects on the page as individual factory classes.



As shown in the above figure, the “Masterpage” class contains two methods, both would accept a WebDriver as an argument and either carry out an operation (possibly demanding more parameters) or return a result.

This pattern tidies code into manageable classes. The “Masterpage” class will contain all functionality present in or depended on by all/most other classes and represents the HTML Masterpage of RealStew.

Accordingly all functionality relating to the contact book would logically be stored in a static class “Contactbook”. This would include: adding a contact, deleting a contact, searching for a contact and any navigation inside the contact book element(i.e. tabs or indices).

The decision to have WebDrivers passed in as a parameter as opposed to as a local variable within a class (and returned in the function) is due to the fact that, as a parameter, we have significant flexibility in writing tests and far fewer memory allocation calls. This decision, in turn, lead to the conversion from public classes to static classes.





## The UI Map

The UI map was a design decision made in an attempted to simplify the identification of often used UI elements from the website. The maintenance cost is lowered significantly by storing all the common identifiers in one class as public static variables. The optimal unique identifier is always the one stored in the UI map.

An example use is sending a click event to the userBar element (the button used to bring up the login form):

|  |
| --- |
| driver.findElement(UIMap.NavigationBar.UserBar).Click(); |

The type of all identifiers is a ‘public static By’ where the means of location is varied depending on what the best option is for that element. For instance, say one element could be identified by its ID attribute, but another has no ID and instead can be selected by a CSS selector. The first would be stored using By.Id(() method, while the second would be By.CssSelector().

This way the above code is applicable to both elements by changing variable of in the UI map. But the glaring problem with this model is the fact that this design will quickly get out of hand with the number of elements, along with similarly named elements conflicting with each other.

To help remedy this issue, we use inner-classes to separate elements into logical categories. Ideally this would reflect the nomenclature of our page object classes, but that would not provide us with much naming flexibility, so it is instead separated into a much more specific variation on the same design. The downside to this is that traversing the classes can become quite burdensome. For example”

|  |
| --- |
| UIMap.NavigationBar.UserControlMenu.ApplicationManagementMenu.AppSubscriptions; |

The justification of this model is hopefully in the improved readability of the above example when compared with an a single selector, for example :

|  |
| --- |
| By.CssSelector("#pageerr > table > tbody > tr > td:nth-child(1) > div:nth-child(1) > div.crmTabs > div:nth-child(7)"); |

Another reason for the decision is because having the identifiers all accessed from a single location, when inevitable change affects the way we locate a element, the level of code reuse lets us easily change it once for all tests

It is worth noticing that the variable location in the UIMap gives a hint to what elements must be accessible before the final element will be visible. Looking at the first example, we must be able to see the **Navigation bar**, then open the **User Control Menu**, then the **Application Management Menu**, then click on **AppSubscriptions.**





### Selecting the best Locator

When adding an element to the UIMap, it is important that three factors be considered when selecting the method in which we find the element:

- Reliability

- Speed

- Uniqueness

Reliability refers mainly to the difference between locating an element based on its relative position or by a unique identifier. The website is prone to change, the format is far more volatile than the element ID, thus we should always prefer the By.id and By.Name over either the By.CssSelector, By.XPath or By.DOM. We should also keep in mind that the By.CssSelector is prefered to the By.XPath, because xpath relies on the implementation of xpath in the browser, this is vendor specific and may act in an unexpected way in cross-browser tests.

Speed mainly concerns By.CssSelector and By.XPath, By.CssSelector is marginally faster than By.XPath. But one should also be aware that By.ID and By.Name are faster than either of the above.

Uniqueness is in reference to locators such as By.TagName, By.Identifier and By.Name. These locators are prone to producing multiple results and while only the first found will be used, unless we are explicitly searching for multiple elements or the first of a list of elements, we should prefer By.Id.

* Structure-based locators: locators that rely on the structure of the page to find elements.
  + XPath
  + DOM
  + CSS
* Attributes-based locators: locators that relies on the attributes of the elements to locate them
  + Identifier
  + Id
  + Name
  + Link
  + CSS

Generally we prefer By.Id when ever we can use it, but that isnt always an option. In which case By.CssSelector is the most common in our code.



\





# Introduction by Example: MasterPage Class

Like most classes in the library, the Masterpage class is a public factory (static) class. It acts as a container for a number of functions that do not logically belong to any single class. An example of this is the **Login** function. This function is called by many functions within other classes and so would be inappropriate (and repetitive) to store elsewhere. The Masterpage class should ideally mirror the concept of an HTML template page and will include functions relating to the navigation bar and app slider.

|  |  |
| --- | --- |
| **Method** | **Description** |
| [**public static void** Login(**IWebDriver**, **string**, **string**)](#h.uttsx2kmjri) | Function takes a webdriver and logs in to realStew using either the details passed to it or default details in the function declaration |
| [**public static void** isLoggedIn(**IWebDriver**)](#h.29py851j8m93) | Function takes a webdriver, searches for the userBar element and returns a bool based on the contents of the title attribute. False if empty, otherwise true |



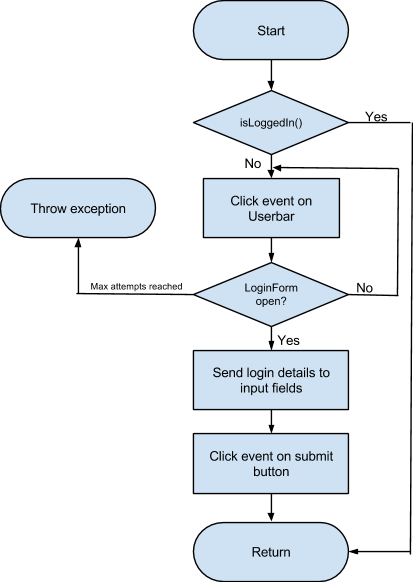


### Login(IWebDriver,String,String)

|  |
| --- |
| **public static void** Login(**IWebDriver** driver, **string** userName=**"defaultLogin@example.com"**, **string** password = **"defaultPsw"**) |

The **Login** function offers the ability to login using provided details or, if omitted, the default details contained in the function definition. When using this function, the developer should be aware that this function requires access to the navigation bar and will not run if the driver is already logged in to an account.

It is not advisable to use this function to check if the driver is logged in, as the “**IsLoggedIn**” function is provided to do that. If this **Login** attempted when the user is already in a logged in state, no action will be carried out.



**Design Notes:**

This function will fail if the userBar element isn’t visible on function call.

Function estimated average runtime ~ 1second

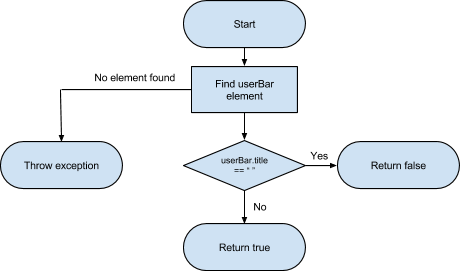




### IsLoggedIn(IWebDriver)

The **IsLoggedIn** function provides returns a boolean if the driver is logged in to realstew. This function reads the userBar ‘title’ attribute against an empty string and will return the negated result of the comparison.

|  |
| --- |
| **public static bool** IsLoggedIn(**IWebDriver** driver) |



Function Estimated Avg. run time: 22.2ms





# The Art of Waiting:

The first and most common issue one is bound to experience with the Selenium tool is that it runs too quickly. While being one of the slower automated testing tools (due to the fact that it renders the webpage in a fully featured browser), it still runs far too quickly for a dynamic website like RealStew. It follows then, that it will inevitably attempt to find elements before the page has reacted to an event (onclick) or for an animation is completed.

Due to the webdrivers inability to tell when exactly the page has finished reacting or if ajax has completed its jobs, we must be careful to purposefully slow our tests down and avoid any unfortunate ‘NoSuchElement’ exceptions.

Ideally we would only wait the minimum amount of time possible, but it is becomes increasingly difficult to build in code redundancies for the purpose of increased stability. In this section we will store information relating to the avoidance of errors caused by inadequate waiting precautions in our program.

**NOTE**: Thread.Sleep() is both undesirable (because it may wait longer than necessary) but also wholly inaccurate (may not wait long enough). Please avoid using this to wait for elements

### Waiting for Ajax

The bane of automated testing, Ajax will update items with no regard for our fantastic program or how it feels about the whole matter. With this in mind, it is important that we check ajax **OFTEN**. Fortunately, in the file CustomConditions.cs, there are a number of custom functions used to improve the way we wait; one of which is called WaitForAjax.

WaitForAjax first checks if the driver passed is capable of executing javascript. Those that are implement the IJavaScriptExecutor Interface. The interface enforces the implementation of a function ExecuteScript(script), we can now inject the following line of JS “return jQuery.active == 0”.

If this returns a true value, then we return. Otherwise we rest for 50 ms and query again. This will loop until the condition is met and we return to the code that called the function or if we meet the timeOut parameter, in which case, we write “WaitForAjax timeout, hoping for the best…” to the console and return.

There is no exception thrown for a timeout because the element may be present and we would have thrown an exception for nothing, but the message is presented so the user knows where in the website there may be heavy ajax activity, and design accordingly.

An example use of WaitForAjax:

|  |
| --- |
| Contactbook.OpenContactbook(driver);  CustomConditions.WaitForAjax(driver, 3000); |

**NOTE:** The second parameter (timeOut) is given in milliseconds, a long timeout is best to be safe, but usually a FaitForAjax call should terminate before 2 seconds.





### Waiting To Click

Occasionally, waiting for Ajax is inappropriate given the situation. Instead we use the Wait.Until function provided by selenium. This function takes a single argument which is a conditional function. Most of these are found in the file OpenQA.Selenium.Support.UI.ExpectedConditions (e.g. ElementIsVisible) but alternatives are provided through the CustomConditions.cs file. These include ElementIsClickable, which waits until a given element is both visible and enabled before returning.

When making use of the WebDriverWait class provided by selenium, it is useful to know that you can disable timeout exceptions. One reason for doing this is if we plan repeat a number of steps if an element is not present.

|  |
| --- |
| WebDriverWait wait = new WebDriverWait(driver, TimeSpan.FromSeconds(5));  wait.PollingInterval = TimeSpan.FromMilliseconds(100);  wait.IgnoreExceptionTypes(typeof(WebDriverTimeoutException)); |

There are others included in the file, but these are in a volatile state as it is unclear if they are necessary or not and will likely be deleted.

### 

### Click, Check, Repeat

The act of firing an event (onclick) then checking if a condition has been fulfilled (the appearance of a certain expected element). If the element is not present, then we attempted the event again. Once we have attempted this action a certain number of times, we throw an exception (E.g. “MaxAttemptsReached”).

This technique is effective on static websites, as Selenium knows to wait for a pageload event to complete before searching for elements, but on a dynamic website (taking into account numerous animations and the limitation of variable connection speeds) we need to add certain steps to ensure that we give the element enough time to become available on the page before looking for it.

**Basic Example:**

|  |
| --- |
| do  {  if (attemptCounter > 3) throw new Exception("Max login attempts reached");  attemptCounter++;  userBar.Click();  } while (wait.Until(ExpectedConditions.ElementIsVisible(UIMap.LoginForm.LoginBody)) == null); |

In the above example, you should take note of the wait.Until condition, this will wait for a specified amount of time, regularly looking for the element until either it is found or the time runs out. In this example ‘WebdriverTimeoutException’ is ignored so the function should return ‘null’ if no element is found.

