# Overview of Engine and How it Works

The framework engine functions are all contained in the package:

C:\Projects\testSeleniumFramework\src\main\java\com\automatedTest\engine

### Action

All methods and objects of the Action class are static.

The Action class contains all the action methods which perform the actions as entered into the “Action” column of an html script (test script.) E.g. click, setparameter, navigate etc.

By the time the engine gets to the action class, if the action is one which acts on an element, the element will already have been located. Each action method has the same parameter interface:

public static Result actionmethod(WebElement currentElement, String inputvalue) {

}

They all return a Result object. Result is a class within the engine package. It consists of three strings:

**errorMessage:** If a step fails should be populated with an appropriate useful error message giving the reason for the failure.

**outcome:** The outcome of the step. Should be set to “PASS” or “FAIL”.

**output:** If the action returns any output – which is returned to the Output column of the test results, the output string should be populated with the value of the output. In the test script, if a script value name is given in the Result Name column of the test script, the script variable will be populated with the output returned.

The Action class contains a Result object actionResult. Each action method uses this Result object as the object to return. Therefore at the start of each action method the “resetActionResult” is called, which initialises the three strings. errorMessage and output are initialised to blank and outcome is initialised to “PASS”.

The action is then performed and if it fails you must specifically set actionResult.outcome to “FAIL” and actionResult.errorMessage to an appropriate error message using the actionResult.setOutcome and actionResult.setErrorMessage methods.

See an existing method for examples of how to create a new action. At the top of the Action.java are a list of instructions for creating a new action method. Note all action methods should have a name which is all lowercase.

### Test

All methods and objects of the Action class are static.

The Test class is where the main processing of the engine takes place. It is the class which contains the driver object for a test (so you will see Test.getDriver() all over the place in the code, anywhere the engine needs to perform anything on the driver object.) It also contains a load of String fields which store information about the test, needed by the engine.

The Test class contains the actOnElement method which does the following:

* Sets the current “wait” time for the step if required. This is the time the engine will keep looking for the element on the page specified in the locator columns (or if switching windows etc, it will keep looking for the window you are switching to until the wait time expires.) If no wait time is passed in will leave as the default wait time. If 0 is passed in will set to zero. If any other integer is passed in will increase the default wait time by this amount. E.g. if the default wait time is 20 seconds and the step has set “5”, then the wait time for that step will be increased to 25 seconds.
* For action, inputvalue, locator type, locator text and innertext, the actOnElement method will then evaluate any “formulae” which have been entered into these columns. E.g. $myvalue .toLowerCase().
* If a locator type has been passed in, the method will then locate the element. As mentioned above, it will do this for the timeout wait. If it does not find the element in this time, it will deem the element “not found”. For all actions other than the “doesnotexist” action, this is a step failure and the step will be failed and processing will not continue for the step. See separate section about how elements are located for more details on this.
* Will call the Action method for the action passed in (see section on the Action class for further information.) The Action method called will return the Result object which tells us whether the step passed or failed, returns and output and any error message. If a script variable was defined in the Result Name column for that step, the script variable will be populated with the output value returned from the step. The result is then returned to the script object (either AllTestFixture or CalledTest both of which inherit from StepBase) which returns this to the result html file.

### StepBase

The StepBase class is the class which represents the actual script being run. Unlike the Test and Action classes, it is not a static class as when a test is run a new object is created each time. However it does have a couple of static objects which are static so that the engine can get their values throughout the life of the test. Otherwise need to pass a StepBase object everywhere:

currentTestStatus: this stores the current test status. Is used by the Action method “callhtml” to determine whether an html script just called passed or failed.

currentResource: This stores the current html file which is being run and is needed so that the java class which runs the test (i.e. AllTestFixture or CalledTest) knows what the current script name is.

The StepBase class contains method “performAction”. This is the method which is called by each row of the Concordion html scripts. In summary this method does the following:

* Keeps a note of whether the script passes/fails/does not complete so the correct PASS/FAIL status can be returned.
* Looks at the rowswitch to determine whether the step should be run or not. A row can be switched off by commenting it out (with //) or by putting in a formula which evaluates to false.
* If the row is to be run, will call the action Test.actOnElement.
* Handles parameter passing after each step – basically will clear them down before any step which is not calling another script or method, setting up parameters or getting parameters.
* Has a @before method “setUpTest” which clears everything down which it needs to in the Test class for a new test including opening up a browser. It also has an @after method “tearDown” which closes the browser and determines whether the script passed or failed or fell over (so failed). Note that if the test is a called script (i.e. one which is called via the callhtml or drivehtml action), then the setup and teardown method do slightly different things as for a called test, stuff must not all be re-initialised and closed down.

The StepBase class has two classes which inherit from it:

**AllTestFixture:** For any html script which is the “initial calling test” – i.e.a script which is “launched” from the frontend.jar file, an “AllTestFixture” object is the Concordion class which runs through the test script. AllTestFixture is located in the “Test” portion of the project, i.e. in:

\testSeleniumFramework\src\test\java\com\automatedTest

**CalledTest:** This is what is used for any html script which is called or driven via the “callhtml” or “drivehtml” actions. The only difference between CalledTest and AllTestFixture is that CalledTest overwrites the value of testType to “CALLEDTEST” instead of “INITIALCALLINGTEST” so that the engine knows that this is a called test - which it needs to do so it knows not to initialise everything when the test starts and knows not to close the browser and shut everything down when the test ends. Only an “INITIALCALLINGTEST” should do this. The CalledTest class is in the main section of the project:

\testSeleniumFramework\src\main\java\com\automatedTest\engine

### AppProperties

This class handles any retrieving anything from any properties files in the Test/Resources folder. Particularly the TestConfig.properties file which contains the browser which should be used for the test and urls to use etc. It also has a method which gets the password from the User.xlsx file used for the tests. Still to be decided how this works. May just store usernames and passwords in TestConfig.properties. . This properties file is located at C:\Projects \testSeleniumFramework\src\test\resources

### Runner and RunnerList

The Runner class is the class from which any called method classes need to inherit from. The RunnerList class basically stores a list of all the methods which have been written (either utility methods such as ExecuteSql or application methods) and defines their “name” i.e. what the tester should put in the test script when calling that method. E.g. to call the run method of the TurnIIPOn class, the tester should pass in “turn\_iip\_on” as the input value for the “callmethod” action.

# How the Engine Locates WebElements

This is all done in the Test class.

As the engine needs to cater for very generic actions performed in step by step processing, the task of locating objects and ensuring the scripts run through as robustly as possible is quite a complex process. The main issue is that Internet Explorer browser does not wait for a page to re-synch after an action which causes a page to reload. Therefore when writing test scripts, the tester must try to ensure that whenever they perform an action which causes the page to reload (typically clicking a button/link etc) then the next step should ideally be performed on a webelement that is “new” on the reloaded page. So for example if you have a search field and button which always appear on the page, and you perform a search, performing an action on the search field or its button will not make the script wait for the page to load as these elements already appear on the page. To try to cater for this, the “click” Action method calls a “waitForPageToLoad” method – which is only performed if the browser is IE. This effectively waits for the page to go “stale”. However the tester should still ensure they perform an action on a new object if possible as between them the “waitForPageToLoad” and looking for a new object should be good enough to ensure the script waits before performing the next action.

There are three columns which locate objects. Locator type, locator text and innertext. There is a separate document on how to actually use these locators to identify objects – which is not covered here except where needed to explain how the engine works.

The Locator Type and Locator Text columns are used by the engine to locate objects using standard selenium locator types. The Locator Type is basically a Selenium Locator type e.g. by xpath, id, name etc. The Locator Text is the actual input the Locator Type requires. E.g. for the “id” locator, the locator text should be the Id. For the “xpath” locator should be an xpath query etc.

The third column “innertext” adds another dimension to locating objects. The “innertext” is any visible text on the page for the element. So essentially any text in between the element tags or any of its child tags. E.g the innertext for the following:

<a href=/mylink>Hello There Universe</a>

Is “Hello There Universe”.

If this “innertext” is passed in, the engine will basically first check for any webelements located via the locator type and text. It will then go through each “potential” element and looks for one with the innertext passed in. The innertext can be a regular expression such as “Trees and the environment.\*” so will look for potentials with innertext starting with “Trees and the environment”. You could argue, why do we need this innertext column? Can’t you just use an xpath query with [starts-with(text(),' Trees and the environment ']? Well yes you can, but I have found that sometimes the text() function does not work as expected. It does not strip out white space characters and so will not work. Therefore, as well as adding flexibility, the innertext column is actually needed to locate objects in some cases. It also makes the scripts easier to read.

The engine will actually strip out whitespace etc to ensure that when working with innertext everything works as expected. In the engine anywhere you need to get the “innertext” use the method in the Action class: Action.getElementInnerText.

The engine will only look for an element for the “currentTimeOutWait” in the Test class. This is determined by the default timeout wait (timeOutWait in the Test class) and anything in the “Wait” column of the script for that step. If the wait column is blank will wait for the default timeout wait. If 0 is entered currentTimeOutWait will be set to zero (to be used when you know a page has loaded so if the element does not exist, will not delay the test for no reason). If “the wait columns is set to an integer greater than zero , the currentTimeOutWait will be set to the default timeout wait plus the integer value entered in the wait column.

The way the engine locates objects and handles the timeout wait is all done manually in various methods. The browser driver does not have the implicit timeout wait set using

manage().timeouts().implicitlyWait(timeInSecs, TimeUnit.SECONDS)

**Indeed, the above must never be used anywhere in the framework.**

The reason for this is that this can only be done once for the life of the driver object. So certainly cannot be used to control the timeout wait, which might be different from step to step. If you don’t set it, it defaults to zero which is crucial for the framework to work correctly.

There are a number of methods which are used to locate objects:

##### boolean isElementThere(By by)

For the currentTimeOutWait will perform a Selenium findElement(by) to locate the element using the Locator Type and Locator Text passed in. It does this by staying in a loop which waits every second until an element matching the Locator Type and Locator Text appears – up to the timeout wait. If such an element is found before the timeout wait runs out, returns true. Otherwise returns false. This is essentially a method which handles the currentTimeOutWait for each step explicitly.

##### int getElementCount(By by)

Calls isElementThere() method to establish if any elements with the Locator Type and Locator Text exist (which looks for the element for the timeoutwait). If the element does exist, then it uses a Selenium findElements(by) to get a count of the number elements matching the locator type and locator text passed in.

##### WebElement findElementByInnertext(By by, String byinnertext)

This is quite a complex method which for the currentTimeOutWait will:

* Get a count of elements which match the Locator Type and Locator Text passed in and ensure the number is not rising (does this to cater for fact page may be loading – and if have not waited, may get a StaleElementReferenceException)
* Once established there are elements which match the Locator Type and Locator Text and the number is not rising, will then search through these elements until it comes across one which has the innertext passed in. If no element matching the innertext is found, it will go through the whole process again (if not exceeded the timeout wait.)

The whole process of locating an element is best illustrated by the diagram below. The process basically has to cater for the fact a page may still be loading and an element may be changing its state, properties, innertext etc:

Get currentTimeOutWait and establish the end time

Has the innertext column been filled in?

Call method findElementByInnerText

Call method getElementCount.

If count > 0, use Selenium findElementBy method to get the element

No

Yes

Was an element found?

End: Element was not found.

No

Establish whether the element is not stale and is in the correct state:

Check the element has not gone stale by checking its enabled and displayed status. This will throw an error if the element has gone stale.

If the action is one which requires the element to be displayed and enabled (e.g. set, select, click etc) then if the element is not displayed and enabled, the element is deemed to not be in the correct state.

Element in correct state and not stale?

End: Element was found

Yes

Is current time > end time or has loop been gone through 4 times?

Yes

End: Element was not found

Yes

No

No

# Timeout Waits

There are a number of timeout waits used in the framework to ensure the tests run as efficiently and smoothly as possible. These are all in the Test class and in there is documented what each one does.

**timeOutWait** = the default timeout to locate an element. This has been set to 10 seconds at time of writing. This is fairly low but coupled with the pageLoadAfterActionTimeOutWait will give any page load around 15 seconds. We want to keep it fairly low otherwise tests which are failing for whatever reason and cannot locate elements they need to will take forever to run as they will wait for the timeout to locate that object. Note that this timeoutWait is implemented by the isElementThere method rather than using implicit timeouts or explicit timeouts. We needed to do this to ensure complete flexibility in locating elements and mainly because setting an implicit timeout wait causes problems as can only be done once against a driver object. So in the framework **you must NEVER user a manage().timeouts().implicitlyWait** to set the implicit timeoutwait. We need this to be the default of zero. Also an explicit (waitForElementPresent) action cannot be done against elements where we are using a “root element.” So this is why we manually created this wait in the isElementThere method (which is basically a loop which waits for the element to appear every second up to the timeout wait limit.)

**currentTimeOutWait** – the current time out wait to locate an element (for the step) – this may vary from step to step depending on what has been entered in the “Wait” column of the script.

**pageLoadTimeOutWait** – this is what is used to manage the driver page load time out wait using manage().timeouts().pageLoadTimeout. If this is not set, then it defaults to waiting indefinitely. This is usually fine but sometimes on the Internet Explorer driver a page says it is still loading and stays in the state forever. Therefore we had to put in this timeout wait and set it to a low value. For certain actions such as navigate and click, we have had to increase it to genuinely allow for a page to low but after the action is completed it is always set back to the value of pageLoadTimeOutWait.

**pageLoadAfterActionTimeOutWait** – after a click, a page is usually reloaded. For the click action, the page load timeout wait is increased to pageLoadAfterActionTimeOutWait (using manage().timeouts().pageLoadTimeout) so that browsers which do wait for a page to load have a time to load which is longer than the default (e.g. Firefox browser). For Internet Explorer, the method waitForPageToLoadIE in the Action class is called after a click to mimic as best as possible waiting for the page to reload. Basically it will wait for the page to go stale. After some clicks, a page is not reloaded, so this is why this has been set to a fairly low value of 5 seconds. However for that step the tester can always pass in 0 as the inputvalue and in this instance the engine will not wait for the page to load. It should be noted that just because we have put in processing to wait for a page to load, the “best practice” of after any action which causes a page to load, making sure the next step acts on an a new object on the page to ensure we wait for the page to load, should still happen to ensure the script are as robust as possible.

### Why have these timeout waits not been added to the properties file test/resources/ TestConfig.properties?

The “tweaking” of allowing a page to load and the test running most efficiently is best done by the maintainers of the framework. To ensure tests are as robust as possible we want all testers to be running with the same timeout waits rather than locally changing them.

# Script Variables

Please note the term “script variable” and “script value” are used throughout the documentation but they mean the same thing.

Basically some actions return output (e.g. gettext, getattribute etc) and to enable a tester to be able to save this value and use it later in the script we have the script variables. If the tester wants to save the value, they should define a name for the script variable in the Result Name column. This name must begin with “$” and the second character should not be a number. So $myvalue2 is fine but $2myvalue is not allowed.

The script variables are stored in the Test class in the scriptVariables hashmap object. Whenever a script variable is saved, the javascript engines are also updated:

**jsScriptEngine:** Stores the value as a String

**jsNumberScriptEngine:** Stores the value as a number if it is numeric, otherwise as a String.

Basically we store these values in the script engines above so that when a script entry has a formula in it such as

$myvalue + “case”

(which basically concatenates “case” onto the end of whatever is in $myvalue) – all we have to do is evaluate the js script engine to evaluate the value of the formula (all formulae should be valid javascript notation – this is documented elsewhere.)

The jsNumberScriptEngine supports formulae where you need the script to treat numeric fields as numbers rather than strings. You would do this using the action “checknumericexpressiontrue” and “evaluatenumber”.

So every time a step has a script variable name in the Result Name column, a new script variable will be created with the value of the output from that action – or if the script variable exists already in that test script its current value will be overwritten with the value of the output from this action. Also the two js Engines described above are updated.

There is slight complexity in all this that because we can call other html scripts using the “callhtml” action and also the “drivehtml” action, and we want all these variables to be local to each test script in the call stack, we need to store the three sets of data (scriptVariables plus the two js engines) in hashmaps for a particular call. This is done in hashmaps:

callStackScriptVariableInstances

callStackScriptEngines

callStackNumberScriptEngines

Every time a new method or script is called the Test.callStackInstance is incremented and method setCallStackInstance sets up the new objects for use at that level in the call stack. Once a called test is finished, the callStackIntstance is decreased by one and the script and engine data that was stored for it is cleared down.

# How Test are Run

This is all located in:

C:\Projects\testSeleniumFramework\src\test\java\com\automatedTest

And

C:\Projects\testSeleniumFramework\src\test\java\org\concordion\internal

Concordion html scripts usually need an accompanying java class to run. In the Framework this has been changed so that we only need one java class – the AllTestFixture class to run each script. To get this to work the Framework uses local class

C:\Projects \testSeleniumFramework\src\test\java\org\concordion\internal\ ClassNameBasedSpecificationLocator.java

Which takes precedence over the Concordion class with the same name and basically looks up the html resource to run by looking at the current value of AllTestFixture.currentResource.

To call a test the maven command needs to therefore pass in the name of the html script or scripts you want to run:

The FrontEnd. jar basically sets up the correct maven call.

For example to call test script \SmokeTests\SmokeTest01.html and for the results to go to the C:\Selenium Results folder, you would need the maven command:

mvn test exec:java -Dconcordion.output.dir="C:\SeleniumResults" -Dexec.args="SmokeTests\SmokeTest01.html".

To change the results folder, choose you change what you set for –Dconcordion.output.dir.

To call more than one test, comma separate the list of tests.

To call a folder of test, pass in the folder name.

All test scripts and folders are relative to the automatedTest folder.

You can also override Application Properties such as the browser by passing in –Dbrowser=xxxxxx.

But on the whole we don’t need to know what maven command to run as the FrontEnd editor does all this for us. The project pom file has been configured so that this all works.

If you want to see the command generated from the FrontEnd editor just look in C:\Projects\testSeleniumFramework\framework\runSelectedTests.bat

# Calling html scripts from another html script

This is done in two ways, using the callhtml Action method or the drivehtml Action method. In both instances it is the CalledTest class (located in the main Engine package) which is actually used rather than the AllTestFixture class. This is exactly the same as the AllTestFixture class except that its constructor method sets the value of testType to “CALLEDTEST”. This is because the engine needs to know if the current html script it is running is the INITIALCALLINGTEST (i.e. the main test) or a called test.

The below diagram summarises how this works for the callhtml method. The drivehtml script is similar but has to also build a result file for each row of the data which is processing. The below diagram also gives an indication of how a test run works.

FrontEnd editor launches a set of test scripts (a “test set”)

No

END:

Test set run complete

Test scripts still left to run in the set?

Yes

Sets AllTestFixture.currentResource and ClassNameBasedSpecificationLocator.currentResource to the current html script name and runs the AllTestFixture class as a Concordion test.

Before running through the script will run the AllTestFixture.setUpTest method which:

* Initialises and clears down all Test class objects
* Sets the Test.callingTestLocation (the path from com/thomsonreuters/selenium/automatedTest) and Test.callingTestName
* Moves any existing test result for the script into the previousRuns folder for that test.
* Starts the driver instance (opens the browser)

The script will be processed like a normal Concordion test. For the framework this involves going through each row and calling the AllTestFixture.performAction method which works out if the line is to be processed according to the rowswitch value and then calls Test.actOnElement.

If the action is a “callhtml” action:

1. Creates a timestamped copy of the test script and moves it to the \_Calls folder for that script (in the src files).
2. Sets Test.calledTest to the test being called’s name.
3. Sets CalledTest.currentResource to the test being called
4. Saves the callStackInstance, callingTestLocation and callingTestName
5. Calls Test.setCallStackInstance to increment the callStackInstance by one and set the Script Variable Objects for that level in the stack
6. Runs the called script through the CalledTest class.
7. The CalledTest.setUpTest method knows that this is a “CALLEDTEST” so does not do all the initialisation that the AllTestFixture does.
8. Runs through each row of the script. If the called script also calls a html script via the callhtml action, then steps 1 – 8 are repeated.
9. Once a called test is finished, the CalledTest.tearDown method is called which does not close down the test because it knows it is a “CALLEDTEST”. Also the called test result file is updated to remove the breadcrumbs and set the heading to make it clear it is a “called test”. The timestamped copy of the called test is deleted from the src files. The callingTestLocation and callingTestName and callStackInstance of the Test class are returned back to what they were saved as before the call.

When the initial calling test finishes the tearDown method of the AllTestFixture class is run. This updates the summary result file (ResultFolder\com\automatedTest\TestResultSummary.txt) and builds all the test results. It updates the result file of the just ended test to include a link to the Call Log for that test. Then all the driver instances for the test are closed.

# Result Reporting

Whenever tests are run through the Framework, the result summary file which is located at :

[Result Folder]\com\automatedTest\TestResultSummary.txt.

This basically holds each test html script that has been run (not any called tests, just the “INITIALCALLINGTEST”.) If the test already exist in the file will overwrite the result with the latest run result.

At the end of every test the buildTestResults method of the test java class (i.e. AllTestFixture which inherits from StepBase) is called. This reads through the TestResultSummary file and creates the summary result for each test “folder” (e.g. smoketests or demo etc). It does this using the velocity templates in C:\Projects\testSeleniumFramework\src\main\resources\templates\ testresultsummary.vm. These methods build up the breadcrumbs which allow navigation through the results.

**NOTE: for the breadcrumbs to appear in the actual test script results for each level of folder, every time a new folder is created for the test scripts which should be created under:**

**C:\Projects\testSeleniumFramework\src\test\specs\com\automatedTest or any of its subfolders, you must add a dummy html file with the same name as the folder under that folder. E.g. the folder demo has a dummy demo.html file.**

The classes that support this are C:\Projects\testSeleniumFramework\src\main\java\com\automatedTest\extensions

TestSetTestResult.java and TestSetTestResultSet.java and ProduceTestSetResults.java

(NOTE: They are in the extensions package, but should probably be in the Engine package or a different package called Reporting or something. . but have not had time to move these.)

LinkingExtension.java is an extension which has been written to enable a link to appear on the result of a callhtml or drivehtml action. Basically for every event during the test run where a result is reported, it checks Test.calledTestLinkPath to see if it is populated. This indicates the step just run is a callhtml or drivehtml action and contains the link path to the called/driven html test to attach to the result.