Automated Testing Framework and Suite Documentation

The Automated Testing Framework and Suite is designed to remotely run end-to-end tests of the CaseWare Monitor Software Solution, in order to further reduce the testing effort required during each development cycle. It allows for early detection of defects and quicker fixes resulting in less modified code. The framework combines several tools to allow for the remote deployment of the lastest build of Monitor, the automated running of tests, and for the tests to directly interact with the browser in the same way a user would.

Layers

There are three layers in the testing framework: the **Test Steps**, the **Page Object Models**, and the **Logic Layer**. The Test Steps layer contains all the tests written in Gherkin organized by application feature (e.g. Login, Permissions, Roles). The Page Object Model (POM) layer contains all the web element information in static classes; every button/textfield/etc. is mapped to class that represents the page in the web application. This layer also contains all actions to interact with these elements. The third layer is the Logic layer and it contains the actual Selenium calls used by the POMs; it also contains some Monitor API calls that are used to setup tests more quickly than through the UI. By using these layers the code can be reused across different classes which reduces the overall complexity and allows for much easier maintenance of the code-base.

Test Steps

Gherkin Test Cases

All the test cases - called *Scenarios* in **Gherkin** - are initially written in Gherkin feature files. The naming convention for these feature files is FeatureName.feature Gherkin allows for the tests to be written in plain-text english, allowing for the test suite to be maintained and modified by someone with little or no programming experience. Each test case contains about 3-12 test steps.

There are two types of test cases: single test cases and test cases that differ by one step or value. The former are denoted by *Scenario* followed by the test name and the test steps. The latter type is denoted by *Scenario Outline* followed by the test name and the test steps, but also a table of *Examples*; this table hold values to be substituted into the test steps. This will be explained further below.

An example of a **feature file** is shown here:

```
Feature: Engine Configuration

Using the first or third analytic engine server for all tests that require only
one engine
Using the second analytic engine server for all tests that require engine that
does not have engines

# -----------------
# IDEA engine tests
# -----------------

@EngineConfig @success @active
    Scenario Outline: View Engine Configuration page while logged in as allowed
license types
    Given I am logged in as a <usertype>
    And I am on the Home page
    Then I should be able to navigate to the Engine Configuration page

        Examples:
            | usertype                        |
            | system administrator            |
            | technical administrator         |


@EngineConfig @success @active
Scenario: View Engine Configuration page while logged in as BPA with granted
permissions
    Given I am logged in as a Business Process Administrator
    And I have permissions to Engine Configuration page
    Then I should be able to navigate to the Engine Configuration page
```

Scenario Outlines

Scenario Outlines allow for the same test to be exectued with different users or entering differnt inputs into a text-field, without having to duplicate the entire test; this greatly reduces the total lines of the test suite and makes it easier to modify the tests when there is a change to the application. The test steps contain *variables* within angled brackes (e.g. <variable name>). These variables are referenced in the first row of the Examples tables below the Scenario Outline. Each test-run can have multiple variables, represented as a new column in the table. Each row in the table corresponds to a test-run. An example for a Scenario Outline with multiple varibales is shown below:

```
Scenario Outline: feeding a suckler cow
    Given the cow weighs <weight> kg
    When we calculate the feeding requirements
    Then the energy should be <energy> MJ
    And the protein should be <protein> kg

    Examples:
        | weight | energy | protein |
        |    450 |  26500 |     215 |
        |    500 |  29500 |     245 |
        |    575 |  31500 |     255 |
        |    600 |  37000 |     305 |
```

Tags

Each test case is preceded by *tags* that organize the tests into useful categories, especially for secondary features that span multiple pages such as help. These tags allow for selective running of tests. Additionally, these tags allow for multiple test steps with the same wording but on different pages. For example, when pressing save to add a new user is not the same action as pressing save to add a new workflow but the test step would be the same. This is done through *Scoping* and will be explained further in C# Test Steps. The @~*testing* tag is used for test cases currently in development.

Keywords

All Gherkin tests have the same format: the setup, the actions, and the results. Each of these sections have their own keywords: *Given*, *When*, and *Then*. Things such as users that are used and their permissions are setup up using the Monitor APIs in the *Given* statements. The actions that make up the test case, such as making sure a user can navigate to a page of the application after permissions were granted, are written with the *When* keyword. And the statements involving verifying that the expected result is true are denoted by the *Then* keyword. Only the first test step of each sections needs the keyword, all the preceding steps can start with the *And* keyword.

Gherkin Documentation

Further information on Gherkin can be found in the Gherkin documentation found at https://cucumber.io/docs/reference.

C# Test Steps

Each test step, written in Gherkin, must be implemented in code. This allows the steps to be reused across feature files. This allows the new test cases to be added by someone without the need to understand much programming. These step definitions hold all the logic of how a user would interact with the application. For example, a test step for logging into the application user inferes that the user clicks on the username field, enters the keys for their username, clicks on the password field, enters these keys, and clicks on the Login button. Thus, one test step can hide complexity from the test writer and simplify the test suite reducing the possibility of human errors being introduced when adding new test cases, perhaps due to forgetting to click on a field before entering the password. This step definition is shown below:

```
[Given(@"I Login as Admin")]
internal static void WhenILoginAsAdmin()
{
    const string username = "administrator";
    Utilities.LoggedInAs = username;
    LoginPage.LoginAs(username).WithPassword("Caseware45");
}
```

The test definition for a test case needs have the correct keyword (Given, Then, When) and exact string match for a given Gherkin test case.

These test definition methods should be static since this allows for them to be called from other test steps. This allows for certain test steps like filling out a username to be reused in other test cases tht do not explicitly require the username to be filled out. This results in lesss complicated test and reduced maintenance of the framework.

Scoping

Scoping is done in the attribute statement after the string match as shown below. Multiple scopes are allowed. The following test would only be accessible from a test case in the *Users* feature and with the tag *@user_list*

```
[When(@"I delete the user"), Scope(Tag = "user_list"), Scope(Feature = "Users")]
```

Scoping Rule

These are the rules from the **SpecFlow** documentation on Scoping at https://github.com/techtalk/SpecFlow/wiki/Scoped-bindings. SpecFlow is explain further below in this section.

```
Scope can be defined at the method or class level.

If multiple criteria (e.g. both tag and feature) are specified in the same
[Scope] attribute, they are combined with AND, i.e. all criteria need to match.

If multiple [Scope] attributes are defined for the same method or class, the
attributes are combined with OR, i.e. at least one of the [Scope] attributes
needs to match.

If a step can be matched to both a step definition without a [Scope] attribute
as well as a step definition with a [Scope] attribute, the step definition with
the [Scope] attribute is used (no ambiguity).

If a step matches several scoped step definitions, the one with the most
restrictions is used. For example, if the first step definition contains
[Scope(Tag = "myTag")] and the second contains [Scope(Tag = "myTag", Feature =
"myFeature")] the second step definition (the more specific one) is used if it
matches the step.

If you have multiple scoped step definition with the same number of restrictions
that match the step, you will get an ambiguous step binding error. For example,
if you have a step definition containing [Scope(Tag = "myTag", Scenario =
"myScenario")] and another containing [Scope(Tag = "myTag2", Scenario =
"myScenario")], you will receive an ambiguous step binding error if the
myScenario has both the "myTag1" and "myTag2" tags.
```

There may be issues with ambiguous test steps so scoping test definitions by Tag are recommended and additionally by feature. It is only advisable to first scope by tag since it is more easily controlled; any test cases that require this simple need a tag added, as opposed to all the test cases within a feature file potentially creating a conflict between ambiguous test cases.

Bindings

The test step classes contain the code definition of the Gherkin steps; they are denoted by the *Binding* attribute. All test step definitions pertaining to a specific feature are held together in one of these bindings. These classes call on the Page Object Models across the application to perfom actions such filling out the information for adding a new user. These test steps also include Assert statements that verify that the test results are correct. The class structure is shown below with some example methods. The naming convention for the static test steps class is FeatureNameSteps.

```
[Binding]   // denotes that this class contains step definitions
internal static class EngineConfigSteps
{
    [Given(@"I am on the Engine Configuration page")]
    [When(@"I go to the Engine Configuration page")]
    [Then(@"I should be able to navigate to the Engine Configuration page")]
    internal static void GivenIAmOnTheEngineConfigPage()
    {
        var result = EngineConfigPage.ShouldGoTo();
        Assert.True(result, "Was not able to go to Engine Configuration page");
    }

    [Then(@"I should not be able to navigate to the Engine Configuration page")]
    internal static void ThenIShouldNotBeAbleToGoToEnginConfigPage()
    {
        var result = EngineConfigPage.ShouldNotGoTo();
        Assert.True(result, "Was able to go to Engine Configuration page");
    }

    #endregion Navigate


    // all test set-up steps are defined in 'Givens'
    #region Given

    [Given(@"the (.*) engine on the first analytic engine server is configured
with the default configuration")]
    internal static void GivenEngineIsConfiguredWithDefaultFirst(string engine)
    {
        Utilities.Monitor.ReturnDataEngineToDefault(engine,
Utilities.Monitor.AnalyticEngineServerMachine1);    // monitor APIs for setup
    }

    #endregion Given


    // test actions (user actions) are defined in 'Whens'
    #region When

    [When(@"I select the (.*) engine on the first analytic engine server"),
Scope(Feature = "Engine Configuration")]
    internal static void WhenISelectTheEngineFirst(string engine)
    {
        const int index = 0;
        WhenISelectEngine_Logic(engine, index);
    }
    #endregion When


    // all test results should be defined in 'Thens'
    #region Then

    [Then(@"the target path should be (.*)"), Scope(Feature = "Engine
Configuration")]
    internal static void ThenTheTargetPathShouldBe(string path)
    {
        var expected = path.ToLower().Trim();
        var actual = EngineConfigPage.GetTargetPath().ToLower().Trim();

        var result = actual == expected;
        Assert.True(result, $"target path was not correct\n\tactual:
{actual}\n\texpected: {expected}\n");
    }

    #endregion Then


    // all logic that is only used on this page that is only to be used by these
steps should be included here
    #region Private

    /// <summary>
    /// Collapses all but first, second, or third (0,1,2)
    /// analytic engine server machine and then selects
    /// specified engine
    /// </summary>
    /// <param name="engine"></param>
    /// <param name="index"></param>
    private static void WhenISelectEngine_Logic(string engine, int index)
    {
        Navbar.Refresh();
        // save current engine used in test for context
        Utilities.SavedString = engine.ToLower().Trim();

        var indices = new List<int> {0, 1, 2};
        indices.RemoveAt(index);

        if (Utilities.Localhost) indices = new List<int>{0};

        // toggles all other severs so the relevent one is expanded
        foreach (var i in indices)
        {
            EngineConfigPage.ToggleServerExpandedByIndex(i);
        }

        EngineConfigPage.SelectEngine(Utilities.SavedString);
    }

    #endregion Private
}
```

Here the test definitions call on POMs to interact with the web application since most test require navigating multiple pages. They also interact with the Common class for often-used actions/steps across the application such as selecting yes/no on an error message. Navigating through the application is done with the FeaturePage.ShouldGoTo() methods, which use the Navbar class to interact with the navbar.

Scoping

Bindings can be scoped in the same way individual test cases can be scoped, in the attribute declaration. Below is a binding scoped to a feature:

```
[Binding, Scope(Feature = "Engine Configuration")]
```

xUnit

the xUnit testing framework is used for reporting and *Asserts* within the testing framework. It allows for quicker debugging and more clear error messages as to why a test failed. These Asserts should be used only in the *Then* test definitions, but can be used elsewhere where a test case might be considered failed if for example initial navigating to page failed.

The format for a *Then* test definition is as such:

```
[Then(@"the engine should have the Disabled icon"), Scope(Feature = "Engine
Configuration")]
internal static void ThenDisabledIcon()
{
    var engine = Utilities.SavedString;
    var result = EngineConfigPage.HasDisabledIcon(engine);
    Assert.True(result, $"{engine} did not have Disabled icon");
}
```

Assert.True is used on a result boolean so that the test cases are more readable and the reporting is consistent.

SpecFlow

To use Gherkin with C# in Visual Studio we use *SpecFlow*. SpecFlow uses the feature files written in Gherkin and autogenerates a C# file that Visual Studio uses to run the tests. This file is regenerated everytime the project is built. The generated report file is an HTML document and shows many details such as tests re-run, tests that failed once but passed on subsequent runs (labelled randomly failed), and the reasons for the failure (Assert message if this caused the test failure).

SpecFlow Documentation

More information on SpecFlow can be found in its documentation at: http://specflow.org/documentation/Using-Gherkin-Language-in-SpecFlow/

Page Object Models

Page Object Class

One level below the C# Test Steps, each page within the application is to be mapped into a static class named FeatureNamePage. They are mapped as a Page Object Models, where an object represents the entire webpage; every web element and every action (i.e. clicking a button) used by test cases on this page is defined here. The structure for the POM class is shown below with some example methods. The class is divided into the following regions: Element IDs, Navigation, Buttons. After Buttons region new regions can be added for more page-specific functionality.

```
internal static class EngineConfigPage
{
    // all element IDs for buttons or text-fields should be declared here
    #region Element IDs

    private static readonly By EnableButtonBy = By.Id("EnableEngineButton-
btnEl");

    #endregion Element IDs  // note 2 new lines between regions


    #region Navigation

    /// <summary>
    /// Navigates through the navbar to the engine configuration page
    /// </summary>
    internal static void GoTo()
    {
        Common.WaitForLoadingScreen();

        const string correctPage = "Engine Configuration";
        var currentPage = Common.CurrentPage();
        if(currentPage == correctPage) return;

        Navbar.ConfigureTabButton();

        // if you have already gone to any page in Analytics section then it
will load
        Common.WaitOneSecond();

        // if this page is correct, return
        currentPage = Common.CurrentPage();
        if(currentPage != correctPage)
        {
            Navbar.Anaytics();
            Navbar.EngineConfiguration();
        }

        Common.Wait(2); // this is wait helps
        Common.WaitForLoadingScreen();

        var result = Common.CorrectCurrentPage(correctPage);    // checks if
current page is desired page
        Assert.True(result, $"Not on {correctPage} page");
    }

    /// <summary>
    /// Attempts to go to the engine configuration page, returns true if able to
do so
    /// </summary>
    internal static bool ShouldGoTo()
    {
        try
        {
            GoTo();         // expected NoSuchElementException
            return true;   // failure
        }
        catch(Exception ex)
        {
            Common.ExceptionMessage(ex);
            return false;    // success
        }
    }

    /// <summary>
    /// Attempts to go to the engine configuration page, returns true if unable
to
    /// </summary>
    internal static bool ShouldNotGoTo()
    {
        try
        {
            GoTo();         // expected NoSuchElementException
            return false;   // failure
        }
        catch(NoSuchElementException ex)
        {
            Common.ExceptionMessage(ex);
            return true;    // success
        }
    }

    #endregion Navigation


    #region Buttons

    /// <summary>
    /// Presses Enable engine button
    /// </summary>
    internal static void EnableButton()
    {
        var by = EnableButtonBy;
        try
        {
            Common.ClickButton(by);
        }
        catch(NoSuchElementException ex)
        {
            throw new NoSuchElementException("\nCould not find Enable engine
configuration button\nInner message:\n{ex.Message}\n", ex);
        }
    }

    #endregion Buttons
}
```

After the Buttons region, new regions can be added such as for interacting with the text-fields. All regions are seperated by two new lines for easier readability.

Selenium

Selenium is a testing framework for web applications that allows for automated testing of a web page as if a real user was interacting with it. It works with multiple browsers and with SpecFlow it can be used to run the test across multiple browsers concurrently.

Selenium is used in this framework to do all the user actions in the browser such as pressing buttons or entering values into a textfield. Selenium is only present in to lower levels of the framework and should not be used in the test definitions. Its usage will be covered more thoroughly in the Logic Layer section.

A useful feature of Selenium is that an element's values, css style, or text can be acquired using the GetAttribute method.

```
v = elem.GetAttribute("value");
```

Common Methods

Many common browser commands such as ClickButton and EnterField are in Common.cs where these functions have try-catch statements to improve debugging, maintenance, and readability of the framework. These functions should be reused as often as possible. There are certain functions that were created to find users by name in the gridview, or checking if a checkbox is enabled also in the Common class. These functions can be used directly or modified slightly for different gridview layouts.

Selenium Documentation

If more information on Selenium is required, it can be found at the Selenium documentation at: http://www.seleniumhq.org/docs/.

Mapping

Mapping the elements in Page Object Models is amongst the most time-consuming part of build the framework and can prove especially challenging when elements do not have static IDs. When the ID is unavailable to use to find a web element two other methods are recommended: use CSS selector, or xpath. Xpath is more complicated harder to read so it less preferred.

Adding element IDs as class variables allows them to be reused throughout the class, speeding up this process.

Logic Layer

The Logic Layer of the Automated Testing Framework consists of the Selenium Webdriver, Monitor APIs, and the Utlities Class. the Selenium Webdriver is responsible for interacting with the browser for user actions. The Monitor APIs allow for rapid configuration of Monitor and setup for tests. The Utilities class is responsible for test context and keeping track of users for each test.

This framework is designed to run multi-threaded to reduce the total run-time of the test suite. This introduced many challenges, mostly due to two tests using the same users and objects within monitor. This was solved by isolating the memory of each testing thread, and creating users with random names so that they would not conflict between threads.

Selenium

Selenium 2 (aka Selenium WebDriver) allows for automated testing of a web-based application. This improves the repeatability of tests and the speed at which these tests are executed. Selenium works by interacting directly with the browser in the same way a user would; it sends actual mouse-clicks and keystrokes.

To send requests to the browser selenium uses the Driver class. A Driver object stores all the possible commands that can be sent to a webpage, and is the bridge between Visual Studio and the browsers.

Common Selenium Commands

The common Selenium methods used are stored in the Common.cs class. An attempt is made to keep all of the selenium calls in this class, so that any changes to the browser interaction can be easily propegated throughout the test suite. It is advisable to spend some time reading through the available methods here as they have been designed to speed the up development and reduce the maintenance of the framework.

FindElement Method

One of the most common methods used is the FindElement. This method is able to find a web element (IWebElement object) such as a button or test-field based of its element ID or if needed css-selector (or xpath, but this is to be avoided). The method definition is shown below:

```
internal static IWebElement FindElement(By by, string elementName = null)
{
    try
    {
        var wait = Utilities.Localhost ? 5 : 10;
        WaitForElementToAppear(by, wait);
        var element = Driver.Instance.FindElement(by);
        return element;
    }
    catch(NoSuchElementException ex)
    {
        throw new NoSuchElementException($"Could not find element:
{elementName}'\n", ex);
    }
}
```

The method is wrapped in a try-catch block in order to improve the debugging and test reporting. The message is clear and concise so that the cause of a test failure can be quickly deduced.

Selenium uses the By object to pass the request information to Driver; it is used to specify if the element is to be found by its ID or its css-class. Below is an example of getting an element by its ID and by css-selector

```
var list = Common.FindElement(By.Id("GridViewEngineConfiguration"));
var button = Common.FindElement(By.CssSelector("img.x-tool-img.x-tool-help"));
```

The Common class is used here but it is also possible to call the Driver class directly. This should be avoided, since the Common implementation will ensure the webpage has finished loading.

Note: generally the By's should be kept as class properties in the Page Object Models.

```
Driver.Instance.FindElelent(By.CssSelector("h1"))
```

The FindElement method will only return the first instance of a web element matching the ID or css-selector; a list of all elements that match a given ID or css-selector (such as rows in a table [gridview]) can be retrived using the FindElements method:

```
// this returns all the rows in a gridview
var list = gridview.FindElements(By.CssSelector("tr.x-grid-data-row"));
```

The above example also shows that you can perform the FindElement and FindElements method on a IWebElement class to reduce the breadth of the search to only its children elements.

The test suite uses the C# version of Selenium with the SpecFlow plugin as explained in Test Steps section.

Utilities Class

This class stores the methods called before and after each test-run as well as all the methods called before and after each scenario. Each test-thread has its own instance of this class and uses it to store test context such as the user the test is logged in as and which users have been created so that they can all be deleted upon test completion.

Context

The class stores the following properties:

```
internal static Monitor Monitor { get; set; }
internal static bool Localhost { get; set; }   // from Monitor

internal static string PageUrlHttp = Monitor.PageUrlHttp;
internal static string PageUrlHttps = Monitor.PageUrlHttps;
internal static string PageUrlLocal = Monitor.PageUrlLocal;

/// <summary>
/// Use this string to save test specific text
/// </summary>
internal static string SavedString { get; set; }
internal static string LoggedInAs;

// stack for the Generic Users
internal static Stack<string> GenericUsers;

// keeps track of users, including those not not used in step cases
internal static Stack<string> UsersCreated;

/// <summary>
/// Stack of current users, cleared after each test
/// </summary>
internal static string CurrentUser
{
    get => _currentUserStack.Peek();
    set { _currentUserStack.Push(value);
            UsersCreated.Push(value); }
}
// keeps track of how many users are being tracked in current test case
private static Stack<string> _currentUserStack;

/// <summary>
/// Stack of current roles, cleared after each test
/// </summary>
internal static string CurrentRole
{
    get => _rolesCreatedStack.Peek();
    set => _rolesCreatedStack.Push(value);
}
private static Stack<string> _rolesCreatedStack;

// removes last name in stack if username change is not saved
internal static void DontSaveUsernameChange()
{
    _currentUserStack.Pop();
}

/// <summary>
/// Returns a copy of the current users stack
/// </summary>
internal static Stack<string> CopyCurrentUsers()
{
    return new Stack<string>(_currentUserStack.Reverse());
}

/// <summary>
/// Returns a copy of the current roles stack
/// </summary>
internal static Stack<string> CopyCurrentRoles()
{
    return new Stack<string>(_rolesCreatedStack.Reverse());
}
```

The *Monitor* object is used to access the Monitor APIs. For example creating a user is done as such:

```
// this method returns the username, which is bpa with a hex string appended
var bpaUsername = Monitor.CreateUser("bpa", license:
LicenseType.BusinessProcessAdministrators)
```

Whenever any user is created in a test, the username must be added the *UsersCreated* stack:

```
UsersCreated.push(bpaUsername)
```

This is critical to ensure correct clean-up of each test case as many user in the gridviews will cause tests to fail since they cannot find specific username if they are not in view.

The *Localhost* property is set since some tests cannot be run in a local development environment such as help tests since the help documents are not available in the Visual Studio Solutions. Its value is set manually in the Monitor.cs file.

The *SavedString* property is a special property used to temporarily store some test context such as a role name to see if it was not changed or to store the name of an analytic engine.

The *GenericUsers* and *UsersCreated* stacks store all the users created for each test-run. Generic users are the simulated users that log in and run the test. These are the stacks used to track the users for clean-up.

The *CurrentUser CurrentRole* stacks are used to track which users are relevant for a given test. They will usually only store one user/role at a time. The DontSaveUsernameChange is used to remove the last entry in the CurrentUser stack, since the name is added when the Username field is changed but should be removed when changes are reverted.

All of these properties are cleared at the start of each test case.

Monitor APIs

Each testing thread runs a before-test-run method that sets up the API access to Monitor. There is also a before-scenario method that clears all the data from the last test and initializes a new Driver instance. Creating a new driver instance opens a new browser window.

Monitor Class

The Monitor class holds all the API commands available in the test setup and clean-up. These API calls should not be present in the Page Object Model or the test step definitions. These commands allow for users and other monitor objects to be created at the start of a scenario; this allows for each test to have unique users/resources and still executing all the tests in a reasonable amount of time.

This file stores all the variables used for the test-run including the machine names for all the servers. It holds the variable for Localhost to modify which tests get run.

Test Controller

The APIs are integrated in a similar way that the Integration Test Suite used to access the APIs, with three layers. The parent class to the Monitor class is the AutomatedTestController class, which in turn is a child of the AutomatedTestControllerBase class. AutomatedTestControllerBase has all the low-level API calls and is where all the commands are sent to the server to be executed.

Models and Services

The test controller uses the models and services to organize the available APIs. The models and services are based off the same ones for the Integration Test Suite but they have been refined.

The Services are the collection of all services offered in Monitor, that is all the APIs. New services could be added, as well as APIs if new features are added.

The Models store the format of certain objects in Monitor in order to be able to create result sets or setup the API access at the start of each testrun.