**Test Automation Framework SETUP**

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **JAVA** | **Install JRE** | https://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html |
| Verify the installed version | Open “cmd” and type “java -version” |
| **Install JDK** | https://www.oracle.com/technetwork/java/javase/downloads/jdk10-downloads-4416644.html |
| Configure JDK path in System Environment Variables | * Search in your computer for "Edit the system environment variables" * Go to "Environment Variables..." * Go to "System variables" and double click "Path" * Add the path according to the directory where you installed the JDK   + Example: Path: "C:\Program Files\Java\jdk-10.0.2\bin" |
| **Install Intellik IDEA – Coummunity Version** | https://www.jetbrains.com/idea/download/#section=windows |
| **Install Maven** | http://maven.apache.org/install.html |
| Configure Maven path in System Environment Variables | Add the path according to the directory where you installed Maven  *Example*: Path: "C:\Users\teodora.a.coldea\apache-maven-3.5.4\bin" |
| Verify if Maven is installed correctly | Open “cmd” and type “mvn -version” |
|  |  |  |
| **SourceTree** | **Install SourceTree** | https://www.sourcetreeapp.com/ |
|  | Clone project | * Go to InnerSource repository: <https://innersource.accenture.com/projects/GTAA/repos/selenium_restassured_java/browse> * Click on Clone icon from left side of the screen and press "Clone in Sourcetree" * Press "Clone" |
|  |  |  |
| **Configuration.properties file** | Open a notepad file and save it as “Configuration.properties” | In here we are adding all the sensitive information like passwords, hosts, etc |
|  | Create environment variables | In System variables, press "New" – Add “CONFIG\_DIR” and add the path to the Configuration file  E.g. C:\Work\Projects\BMW\Test Automation BMW Utils\Configuration.properties |
|  |  |  |
| **Create other environment variables** | RESULTS\_DIR | Will be used to store the path to the Results for each test |
|  | FILES\_DIR | Will be used to store the path to potential files that will be used in the project |
|  |  |  |
| **Chromedriver/IEDriver/Firefox** | Add paths to Chromedriver, IEDriver,  Firefox in:  **Run Edit Configurations -> VM options**  For class:  selenium.automation.psmg.steps.  RunFeatures | Update accordingly, see examples:  -Dwebdriver.chrome.driver="C:\\Work\\Projects\\BMW\\Test Automation BMW\\chromedriver.exe"  -Dwebdriver.gecko.driver="C:\\Program Files\\Mozilla Firefox\\firefox.exe"  -Dwebdriver.ie.driver="C:\\Work\\Projects\\BMW\\Test Automation BMW\\IEDriverServer.exe" |

**Cucumber Utils**

**RunFeatures Class**

@RunWith(Cucumber.class)

@CucumberOptions(

features = "make sure that the package name is written with "/" instead of ".""

, glue = { "name of the package where the steps are implemented" })

**Annotations:**

* **Cucumber Hooks: @Before, @After -** The methods implemented in @Before, @After, will run before every scenario.
* **Tags**
  + Example:

FeatureFile.feature

@Tag1

Scenario1

@Tag2

Scenario2

StepImplementation.java

@Before(“@Tag1, @Tag2”)

@After(“@Tag1, @Tag2”)

* + The methods implemented in @Before (“@Tag1,@Tag2”), @After(“@Tag1, @Tag2”), will run only before the scenario with “@Tag1” and “@Tag2”.

**HTML Report:**

Add in @CucumberOptions (RunFeatures.java):

plugin = {"html:target/cucumber-html-report"}

cucumber-html-report name of the test report target destination path

Green -> steps passed

Red -> step/test failed

Blue -> step skipped





**Test automation framework for both UI and Backend testing**

Test automation framework represents a set of assumptions, concepts and best practices that provide support for automated software testing. This framework will be used in order to cover the main testing types and multiple applications.

*Framework/test structure characteristics:*

* Creation of small, independent scripts that will represent sections, modules or functions in the current application
* Re-use the defined scripts in a hierarchical fashion to construct the particular test cases
* Build an abstractization layer in front of a component and provides modularity in the application design
* Use abstractization and encapsulation to improve the maintainability and scalability of the test suite
* Define re-usable utility functions
* Define separate and reusable test data sets

**Advantages:**

* High usability and re-usability
* Cross-browser support (Only for UI testing)
* Separation between test data, implementation code and UI identification
* Easy to maintain tests on long term
* Able to maintain simple and complex automation scenarios

**Used and integrated test automation patterns:**

|  |  |
| --- | --- |
| **Pattern** | **Description** |
| **Page Objects** | Represents the screens of the web app as a series of objects. |
| **Domain Driven Testing** | Getting all team members to speak the same language and  managing the life-cycle of a domain object. |
| **Loadable Components** | Providing a standard way of ensuring that pages are loaded and  providing hooks to make debugging the failure of a page to load  easier. |
| **Bot Style Tests** | Action-oriented abstraction over the raw Selenium APIs. |

**UI automation framework structure**

In defining the framework for UI automation, the main idea that is set up is script re-usability, building abstraction layers and use encapsulation for maintainable and scalable test suites.

In order to handle complex, low level and powerful tests for BMW applications, we are going to use PageObjects in order to structure the tests based on application and based on pages treated as objects for better integration and usage.

*At the end of this task the test framework will be set in place containing:*

* defining packages for specific applications/interfaces
* define packages for test step definition
* define packages and classes for base application (high level abstraction over the existing project packages)
* defining package and classes for page driver (browser settings, page, element, page errors – all being an abstraction over Selenium API)
* define packages and main classes for the most used functions in the system (setup application, report generators, error handlers,
* general until methods that can be used for multiple called methods)
* define configuration packages that will set up the tests and environments (environment config, browser setting, data config, CI integration)
* defining package for UI element identifiers (utility functions)
* test environment handlers to be able to run the tests on different environments (dev/test)
* define exceptions and error handling reusable methods

**Test Cases for Automation**

Gherkin is one of the most popular languages for writing formal behavior specifications – it captures behaviors as “Given-When-Then” scenarios. With the help of automation tools, scenarios can easily be turned into automated test cases. Anybody from engineers to product owners can write BDD scenarios, since they are just English phrases. BDD keeps developers focused on delivering precisely what the product owner wants. It also expedites testing.

As such, BDD pairs nicely with Agile Software Development.

**Gherkin terminology -> 10 key words**

* Given - a set of key pre-conditions for a scenario
* When - the key action a user will take (action)
* Then - the observable outcome (assertion)
* And - It can be used in association with Given, When, or Then
* But - Can be used in association with Then. It’s used to say something shouldn’t happen as an outcome
* Scenario - describe the scenario & give it a title
* Feature - Feature is used to give a title for the feature/piece of functionality
* Background - scenarios in a feature file may share common setup steps
* Scenario Outline/Examples - create a table and enter in values
* Step Table – all data is used in as one

**Feature files main terminology**

Useful information to take into consideration:

* Use the name of the actor in the feature files. In this context it will be used user.
* Implement general and reusable steps.
* Define a difference between the step name and the elements from the step.
* Matchers are short and clear.
* Matchers have at most two value parameters.
* Parameters have clear names.
* Focus the feature files to the outputs and not the inputs.
* Use Given Scenario to illustrate the existing scenario precondition.
* Given = past time.
* A step can contain multiple small elementary steps (click, focus, type). Focus on something that can be described as a clear user action. A step is a clear separated USER ACTION. That means not clicking an element but a series of possible actions(clicks, select, enter text)
* to complete a clear definable action (enter customer, end a process).
* Use tabular fields (|variable1|variable2| ...) when there is the need to add/verify multiple values on a specific page.
* Do not use generic steps like: "there should be an error returned" instead the message and the error needs to be defined.

**Why**: If feature files contain lots of elementary steps, then maintenance is on ALL of the feature files. BUT, maintenance should be in the step definitions.

*An example: "When the user clicks "Next" button"*

This looks OK, but what if an 'action' Next will be changed to 2 steps? Then it is needed to update ALL feature files with that additional step. If: "When the user finishes step x"

This means that no matter how many steps "Next" consists of, we can add all those to the step definition file. Only 1 place of maintenance.

**As a rule of thumb: describe how the application behaves, describe the action. Don't add in a feature file which element should be used, describe what the user is doing.**

**Feature files steps in details**

**Navigation**

* **Login** to the application

*Given the user is logged into the application*

* **Start a given process**. Use the complete name of the process/functionality (that is straight forward), just the name or an abbreviation that is widely used.

*Given the user opens "Dashboard" screen*

* **Navigate through pages - using the menu items**. Is expected not to use specific buttons.

*And the user navigates to "Parts" screen*

* **Access a resource URL for Web Services testing**

*Given the user access the resource url "/ExampleJSON"*

**Enter data**

To enter data in a screen several statements can be used, try to define a clear separated user action (the metaphor):

* **Enter data for a process**: describe what user actually is doing, based on the functionality described in

*Then the grid with the following fields is displayed:*

*|column1 |column2 |column3 |*

* **Fill in textboxes**: this is a last resort, only if no decent reusable user-action can be described for complex actions

*And the user fills "textBoxName" with "decimalValue"*

* **Radio buttons**

*When the user checks "radioButtonName1 or radioButtonName2" with "radioButtonName1"*

* **Checkboxes**

*When the user unchecks "checkboxName"*

* **Dropdowns**

*When the user selects a dropdown field "dropDownChoiceName"*

* **Parameters for Web Services testing**

*When the user provides parameter "username" as "testUsername"*

**Actions**

* **Confirm creation of an entity**

*And the user makes a “requestName” request*

* **Open a specific entity**

*When the user opens a “requestName” request*

**Verify data**

Usually "is present" should be used. If you want to see something is NOT present, use "is not present".

* **Verify that a certain text is present on the page.**

*Then the user "userName" is successfully logged in*

* **Verify a warning message is present**

*Then warning "warningMessage" is present*

* **Verify that a certain page or form is present**

*Then "pageName" is present*

*Then the request "requestName" is available in "pageName" screen*

* **Verify multiple results of a step**

*Then the "listName” list contains*

*|column1 |column2 |column3 |*

* **Verify status code retrieved**

*Then the status code should be "200"*

*Then the status code should be "400"*

**Test Cases Standard Examples**

**Scenario Examples**





|  |
| --- |
| **Technical Examples** |
| **Feature**: Testing a REST API  Users should be able to submit GET and POST requests to a web service  **Scenario**: Data Upload to a web service  When users upload data on a project  Then the server should handle it and return a success status  **Scenario**: Data retrieval from a web service  When users want to get information on the Cucumber project  Then the requested data is returned |

**Testing Recommendations**

Gherkin is great for testing APIs and web UIs. Gherkin excels for acceptance testing.

However, behavior specs would be overkill for unit tests, and it is also not a good choice for performance tests that focus on metrics and not pass/fail results.

**Writting Good Gherkin**

[**https://automationpanda.com/2017/01/30/bdd-101-writing-good-gherkin/**](https://automationpanda.com/2017/01/30/bdd-101-writing-good-gherkin/)