Xebia
SOFTWARE DEVELOPMENT DONE RIGHT

# Code Review & Best Practice for BDD Automation Scripting for SALT

**Xebia India,** 612 Park Centra Sec 30, Gurgaon 122002, P +91 (0124)4700 200, F+91 (0124) 4700 222

## Purpose

The purpose of this document is to provide a clear set of standards to be applied to assets developed within the Travelex Automated GUI Test Framework. This document should be understood by any engineer prior to working on automation.

The goal of this document is to provide clear standards for various aspects of the automation framework. This document will help ensure consistency across the code, resulting in increased usability and maintainability of the developed code

## Feature Files should actually represent feature of the application, not a Jira story or Portions of an App

Each feature file should be named as ***<feature_of _application>***. Feature and all scenarios for that feature should be inside that feature file. If feature file has too many scenarios then feature file can be divided as ***<featureName_regression>.***feature: containing all cases needed to run for regression testing, ***<featureName_smoke***>.feature : containing all cases for that feature to be run for health check-up of the application. ***<featureName_functional>.***feature: containing all cases related to acceptance story of that feature

Currently, in the SALT project features are named as <storyname>.feature.

Ex: NZ LCTR Daily Report – Admin.feature, MarketingOptions.feature, Smoke.feature

## Avoid Inconsistencies with Domain Language

You'll get the most benefit out of using BDD when your BA's are involved. To that end, make sure you use their domain language when you write stories. The best course of action is to have them involved in writing the stories. If possible try to write scenarios in Declarative style than Imperative style

**Ex.** Taking example of a below Imperative style scenario:

```
Scenario Outline: Validate First Available Date and Last Available Date in Basket page
for Home Delivery Fullfillment Type
Given I navigate country '<Country>' from admin
When I navigate to SALT Settings from Country tab
And I save new Order Processing Time '<OrderProcessingTime>'
And I click on sales funnel default link
And I enter maximum future order sale time for Home Delivery
'<MaxFutureOrderSaleTime>'
And I click on save button on sales funnel default
```

**In Declarative style:**

```
Scenario Outline: Validate First Available Date and Last Available Date in Basket page
for Home Delivery Fullfillment Type
Given I am on '<Country>' admin page
When I set new Order Processing Time '<OrderProcessingTime>' in setting page
And I set maximum future order sale time < MaxFutureOrderSaleTime > for Home Delivery
```

## Organize Your Features and Scenarios with the Same Thought You Give to Organizing Your Code

One useful way to organize things is by how fast they run. Use 2-3 levels of granularity for this:

- **Smoke Test** : scenarios that verify only basic functionality of relevant feature and run very fast, e.g. under 30/60 of a second
- **Acceptance**: scenarios that verify acceptance criteria of relevant feature are slower but not painfully so, maybe under 120 seconds
- **Regression**: scenarios that verify effect on whole application for relevant feature and take a real long time to run eg. under 600 Sec

You can do this separation several different ways (and even some combination):

- Put them in separate feature files
- Put them in separate subdirectories
- Tag them

## Use Tags

Tags are a great way to organize your features and scenarios in nonfunctional ways. You could use @Regression, @Smoke, @Acceptance, @featurename and @sprint<no>. Using tags let you keep a feature's scenarios together structurally, but run them separately. It also makes it easy to move features/scenarios between groups, and to have a given feature's scenarios split between groups.
The advantage of separating them this way is that you can selectively run scenarios at different times and/or frequencies, i.e. run @Smoke scenarios more often, or run really @Acceptance/@Sprint<no> scenarios overnight on a schedule.
Tagging has uses beyond separating scenarios into groups based on how fast they are:

- When they should be run: on @checkin, @hourly, @daily
- What external dependencies they have: @local, @database, @network
- Level: @functional, @system, @smoke
- NotRequiredToRun: @ignore: this can be used to disable feature and/or scenario that are not ready to be run yet.

## Don't Get Carried Away with Backgrounds (Stick to Givens)

Larger the background, greater is the load of understanding for each scenario. Scenarios that contain all the details are self-contained and as such, can be more understandable at a glance.

## Make Scenarios Independent and Deterministic

There shouldn't be any sort of coupling between scenarios. The main source of such coupling is state that persists between scenarios. This can be accidental, or worse, by design. For example one scenario could step through adding a record to a database, and subsequent scenarios depend on the existence of that record. This may work, but will create a problem if the order in which scenarios runs changes, or they are run in parallel. Scenarios need to be completely independent.
Each time a scenario runs, it should run the same, giving identical results.

## Write Scenarios for the Non-Happy-Path Cases As Well

Happy path tests are easy; edge cases and failure scenarios take more thought and work. As of now feature team (InSprint team) not writing any edge cases.

## Be DRY: Refactor and Reuse Step Definitions

Especially look for the opportunity to make reusable step definitions that are not feature specific. As a project proceeds, you should be accumulating a library of step definitions. Ideally, you will end up with step definitions that can be used across projects.

## Revisit, Refactor, and Improve Your Scenarios and Steps

Look for opportunities to generalize your steps and reuse them. You want to accumulate a reusable library of steps so that writing additional features takes less and less effort over time.

## Refactor Language and Steps to Reflect Better Understanding of Domain

This is an extension of the previous point; as your understanding of the domain and your customer's language/terminology improves, update the language used in your scenarios.

## Use Compound Steps to Build Up Your Language

Compound steps (calling steps from steps) can help make your features more concise while still keeping your steps general—just don't get too carried away. For example:

```
[Given(@"the user (.*) exists")]
    public void GivenTheUserExists(string name)
    {
        // ...
    }

    [Given(@"I log in as (.*)")]
    public void GivenILogInAs(string name)
    {
        // ...
    }

    [Given(@"(.*) is logged in")]
    public void GivenIsLoggedIn(string name)
    {
        Given(string.Format("the user {0} exists", name));
        Given(string.Format("I log in as {0}", name));
    }
```

## Avoid Using Conjunctive Steps

Each step should do *one* thing. You should not generally have step patterns containing "and." For example:

`Given A and B` should be split into two steps:

```
Given A

And B
```

## Do not write application logic inside steps, application logics should be written inside method of relevant page.

Webdriver instance should not be used in steps files. Driver should be used only in page level. Steps should only use pages and call the action methods. Ex. In current framework

```
[Given(@"Login to salt admin")]
        public void GivenLoginToSaltAdmin()
        {
            driver = WebdriverFactory.getDriver();
            Assert.IsNotNull(driver);

            String url = ConfigurationManager.AppSettings["Admin"];
            string userID = ConfigurationManager.AppSettings["RGT_UserId"];
            string passport = ConfigurationManager.AppSettings["RGT_Password"];
            string domain = ConfigurationManager.AppSettings["RGT_Domain"];


            log.Info("URL : " + url);

            try
            {
                driver.Url = url;
                adminLoginPage = new AdminLoginPage(driver).Load();
                Assert.AreNotEqual(adminLoginPage, null);
                adminHomePage = adminLoginPage.LoginToSalt(userID, passport, domain);
            }
            catch (WebDriverTimeoutException e)
            {
                log.Info(e.StackTrace);
                driver.Navigate().Refresh();
                adminHomePage = new AdminHomePage(driver).Load();
            }
            Assert.IsNotNull(adminHomePage);
            ScenarioContext.Current["adminHomePage"] = adminHomePage;

        }
```

In above example all logic to invoke the admin page should be inside invoke() method of adminLoginPage . And above step should be like below:

```
      [Given(@"I loged in Salt Admin application")]
              public void GivenLoginToSaltAdmin()
              {
                    AdminLoginPage.Invoke();

              }
```

## Use custom assert library, for proper messaging on success and failure of assertion

```
Ex. String success = "login button found on page"

        String failure = "login button does not found"

SaltAssert.assertTrue(AdminLoginPage.isLoginButtonExists(),success, failure)
```

User assertion inside step definition for only 'Then' clause

## Use Regular expressions when creating steps.

1. Extract words and values from text in your steps (Capturing-Group). Ex: I have 3 statements and I need to create a step for only first 2 statements.

```
Given A Bird in the hand is worth two in the bush

Given A frog in the pound is worth three in the tree

Given a toad in hole is worth four in the Oven
```

**Sol:**

```
[Given(@"A (bird | frog) in the (hand|pond) is worth (one|two|three) in the
(bush|tree)")]

Public void GivenAPhrase(String animal, String location, String qty, String
whereCouldItBe)

{

……

}
```

2. Supporting variations in sentence construction (Non-Capturing-group). Ex : I have 2 statements which have some variation in sentence and I need to create a step for them

```
Given A Bird in the hand is worth two in the bush

Given A frog in the pound is worth three in a tree
```

**Sol:**

```
[Given(@"A (bird | frog) in the (hand|pond) is worth (one|two|three) in
(?:a|the) (bush|tree)")]

Public void GivenAPhrase(String animal, String location, String qty, String
whereCouldItBe)

{

……

}
```

## Use Properties file for configuring automation environment setup and write utility method to read this property file and initialize the automation run time environment setup.

It will help to setup automation environment easily. Ex. Some Attributes in "automation_config.properties"

```
Admin_URL: "http://admin.rgt.salt.travelex.com/Login.aspx"
RGT_UserId:= “SALTAdministrator”
RGT_Password: Canada123"
RGT_Domain :"emea.travelex.net”
UK_URL: "http://www.travelex.co.uk.rgt.salt/gb/Purchase"
Browser: "Chrome”

Wait_time :8

Implicit_wait: 12

ScreenShot_path : “C:\\Salt_Result_SreenShots”

Chrome_exe_path : <path of your chrome exe>

Capture_screen_shot: false
```

## Page Object Implementation guidelines:

A class object that represents a web page - commonly referred to as ‘Page Object’. It contains group of forms, components and controls to mimic UI Web. It contains two type of functions , one type of function are to mimic high and low level of functionality that a user can perform on the UI page, and second type of functions are for self-test verification.

1. Each page of the app should be its own PageObject
2. Dialog windows should be included in their source PageObject, not one of their own.
3. Page aspects that cross multiple pages should be their own PageObject and not duplicated across multiple PageObjects (ex. Top Navigation and Footer)
4. If function can be used in more than one page then put it in CommonUtiltyFunctionPage.