P&A | Test Automation Framework Guidelines

Owner	@ Hemanshu Chauhan
DL	@ chris.wraith, @ Vivek Upreti, @ Neeraj Verma, @ Nikhil Dass, @ Poulin, @ sruthy.pramod, @ rishabh.govindraj, @ shivani.gupta9, LBGPAEndeavourPlus_PBS_TEAM_EMEA@publicissapient.com, @ Saurabh Sameer Saim, @ Bhargava Nallini, @ sachin.mehta, @ akos.fenemore, @ Sammy Voong, @ Huiqian.Lin (Unlicensed), @ vivek.singh1, @ ed.scott (Unlicensed),

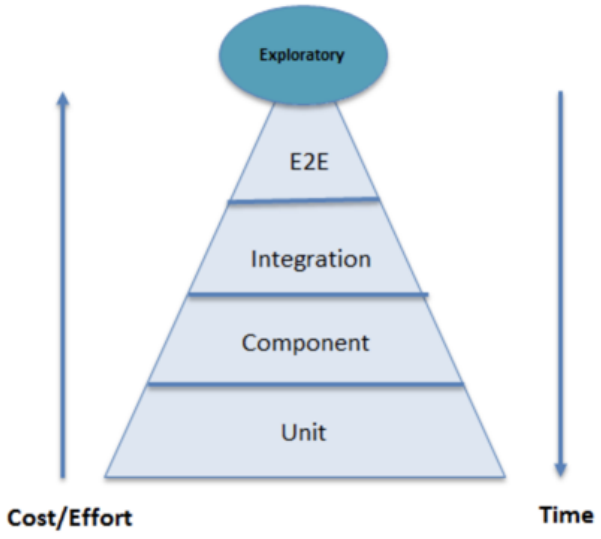
This document describes the guidelines that can help projects design an Automation Framework which is Robust, Maintainable and gives maximum ROI. Although there are endless possibilities to choose from different approaches, techniques, frameworks, tools and ways of code writing, this document provides an insight on best practices pertaining to the Test Automation Framework based on BDD methodology, relevant to Lloyds ecosystem.

This document provides guidelines for:

1. Test Pyramid Approach
2. Design
3. Configuration Options
4. Re-usable Libraries
5. Reports and Logs
6. Continuous Integration
7. Test Data
8. BDD Best Practices

1. Test Pyramid

The Test Pyramid is a concept in Software Testing that focuses on an essential point that you should have many more low-level unit tests than high level end-to-end tests running through a GUI. As, we traverse the pyramid in the upward direction, the Cost and Effort of defect increases rapidly. An ideal approach is the one where, Unit tests are 70%, E2E are 10% while the middle layers correspond to 20%. Any Automation Framework, Tool, Technology can only be prudent only if it's backed up by a robust Test Approach.



- **Unit** : Includes testing of the following:
  - Individual Classes, Functions, Subroutines, Methods and Properties
  - Input Validations/Expected Response
- **Component**: Includes testing of the following:
  - Business Logic
  - Contracts
  - Status Codes
  - Quality of Data
  - Headers
  - Query Parameters
- **Integration**: Includes testing of communication paths and interactions between APIs
- **E2E**: Includes testing of User Journey Paths
- **Exploratory**: Includes testing of AUT manually for uncovered/unexplored areas

2. Design

- Separate Tests from your Test Automation Code: To promote reusability across tests, the framework, and the tests, should be structurally separate within the automation code base. That means there should be one package that holds all of your framework code, and a separate package that holds the test code. The framework code should include methods that interrogate the application, and return what it finds. The test code should call a sequence of framework methods, and based on the information returned, pass or fail it.
- Separation of application locators from the test code so that the locators can be updated in the locator file independently on change. Example: Can be stored in external json,yaml files
- Test Data should be externalised and kept in Json/Yaml files
- Organize tests as modules/ functions, so that they are re-usable and easy to manage. Have application/ business logic in the separate class and call them from the test class
- Tests should start from the base state and ensure that it recovers and continues when there are intermittent test failures
- **Make the Framework Portable: The Framework and the scripts should be portable so that it can be downloaded, installed and executed on different platforms/configurations smoothly.**

3. Configuration Options

The framework should provide option to choose the configurations at run time so that it can be used as per the test execution requirement. Some of the configurations include:

- Ability to choose test execution environment such as Sandbox, SIT, LUAT, PROD
- Ability to test from Mock and Real Services
- Ability to choose the Browser (in case of UI)
- Ability to choose the set of Test(s) that need to be executed
- Ability to choose between Local and Jenkins environment

4. Re-Usable Libraries

Libraries helps in grouping of the application utilities and hide the complex implementation logic from the external world. It helps in code reusability and easy to maintain code.

- Build the library of utilities, business logic, external connections
- Build the library of generic functions of the framework

5. Reports and Logs

To troubleshoot and analyse the test execution smoothly, the Framework should have detailed Logs and Execution reports

- The logs should provide execution details with custom messages
- The reports (HTML) should provide the execution results in Gherkin Format with appropriate Pass/Fail results and details of failures (if any). In case of UI, Screenshots as well.

6. Continuous Integration

The design of the Framework should be in a way that the scripts can be executed in Local and Jenkins Box. Hence, the below components should be flexible to support the same

- Gulp Tasks/Commands
- Configuration Files
- Execution variables
- Sauce Labs Keys (in case of UI)

7. Test Data

Test data plays a vital role in a Test Automation Framework, Hence, it becomes imperative to have a well defined strategy for Creation and Maintenance of Test Date. Few things that should be considered are as follows:

- Test Data should be externalised: Test Data should be stored externally in Json/Yaml files (as appropriate) and not inside the Feature Files/Step Defs/Scripts etc. This gives better control on Data and make scripts Environment Agnostic

- Data Templating: Test data templates should be created on the basis of the Business Logics. This increases Test Data reusability
- Test Data creation libraries: Utilities/Libraries that support creation of Bulk data or data in various formats can be used to create data for testing.

8. BDD Best Practices

The Cardinal Rule of BDD is One Scenario, One Behaviour

Scenarios should be written in Domain Specific Language (DSL) like a user would describe them. Beware of scenarios that only describe clicking links and filling in form fields, or of steps that contain code or CSS selectors. This is just another variant of programming, but certainly not a feature description.

Declarative features are vivid, concise and contain highly maintainable steps.

8.1 Feature File

- In general, Feature Files should be placed inside a folder with the folder name as featureFiles (This can also be the name of the Journey)
- Feature File folder Name, Feature File Name and Tags should be in camel case
- Name of FF should depict a meaningful Functionality of an Application (Not the story number etc.)
- FF should contain a clear description of what functionality is to be tested through that Feature File
- Scenario and Scenario Outlines should clearly articulate the purpose of a specific scenario
- Each scenario should be clear and concise
- In case a scenario has a Pending Tag, it should have a TODO text containing JIRA ID along with a small description
- Feature File/Scenario should be tagged with Appropriate Tags (Like Sanity, Smoke, Regression etc.)
- In case a Scenario needs to be tested against different set of Inputs, Scenario Outlines should be used
- Every scenario should have a Priority/Severity Tag (p1, p2, p3, p4), this gives more control on Scenario execution
- Every scenario should have a Story Tag, this helps in maintaining Traceability between Stories and their corresponding scenarios
- If all the scenarios in a feature file, contain the same tag, the corresponding tag should be moved to the top of the file.
- Each Feature file must have the scenario number starting from 1. The scenario numbers should not be carried forward to another feature file.
- End of each file should have only 1 blank line between the Sceanrios
- The Description of every scenario should be unique
- A scenario should always start with a Given
- Avoid Conjunctive steps
- Mapping between a Scenario's step and step Definition should be One to One, else it can result in abrupt behaviors
- Test Data should be placed in the form of Placeholders, with their actual values stored in external files (Like Json), this results in Environment Agnostic Scenarios

8.2 Step Definitions

- The Step Definitions should be stored in a separate folder under the step_definitions folder
- In general, each page of the application, should represent a single step definition file
- All the Step Definitions of the corresponding page should be inside that file
- The Step Definition file name should be in Camel Case
- All the steps in the file should be unique
- File should not contain any unused step, if it's there it should be removed