Mastek™

Swift - Selenium User Guide

Table of Contents

1 INTRODUCTION 3

1.1 Objectives 3

1.2 Overview 3

Management's requirements 3

Manual testing team's requirements 3

Automation tester's approach 3

2 FRAMEWORK STRUCTURE 4

2.1 Library Architecture Framework 4

2.2 Keyword Driven Framework 4

2.3 Data Driven Framework 4

3 FRAMEWORK INSTALLATION 5

3.1 Creating Folder Structure 5

3.2 Creating Java Project 5

4 FRAMEWORK DETAILS 12

4.1 Flowchart 12

4.2 Folder Structure 12

4.3 Key Components 12

Src 12

Common Resources 12

Resources 15

Input 16

Verification 17

Results 17

Templates 18

5. Execution 18

5.1 Interactive Mode 18

5.2 Batch Mode 18

AMENDMENT HISTORY 19

1 INTRODUCTION

Mastek has successfully developed a highly configurable Hybrid Automation framework: "Swift" using Selenium tool that offers quick turnaround for Automation Suite Implementation. The Automation regression suite can be created or modified using tester friendly keywords for modules, screens, and verifications, thereby ensuring a more effective and efficient Test automation process and better ROI through the complete application life cycle.

1.1 Objectives

The objective of this document is to provide an insight of the Mastek's Swift (hybrid) automation framework using Selenium tool.

1.2 Overview

A test automation framework is a set of functions, assumptions, concepts, and practices that provide support for automated software testing. Functions and concepts are developed for common test automation activities like UI data input, verification and reporting.

The framework caters to the goals and requirements of the management, the testing team and, of course, the automation testers.

Management's requirements

- Reuse the framework across the organization
- Reduction in total testing efforts, improvement in quality of deliverable
- Have a quick turnaround time while releasing newer versions
- Determine status of test execution by looking at the test results

Manual testing team's requirements

- Control test execution process
- Select verifications/checkpoints
- Understand how to run the tests and analyse the failures
- Execute script with minimal support from the automation development team

Automation tester's approach

- Develop and maintain scripts with minimum effort
- Reuse scripts across multiple projects
- Recover from script failure(s)

2 FRAMEWORK STRUCTURE

Swift is a hybrid framework that derives benefits from the following automation framework methodologies.

2.1 Library Architecture Framework

Framework contains rich and robust library of generic functions for data input, verification and reporting. Library functions can be called from anywhere within the framework.

2.2 Keyword Driven Framework

Development of the functionality as step-by-step instruction set captured in external spreadsheets. These spreadsheets use keyword names that are relevant to the business users.

E.g. RegisterUser, EnterUserName, Address details, Save, VerifyUser etc.

The automated driver script interprets the instruction set to simulate user actions. This approach makes the framework independent of the application under test.

2.3 Data Driven Framework

Different data sets used for same functionality to increase regression test coverage. Test case input values will be read from a data file provided by business users. Involves externalization of input data and verification data files provided by business users. Test case expected output values can be read from either input data or external file provided by business users.

The data files can be in any of the following formats: CSV, XLS, XML, etc.

3 FRAMEWORK INSTALLATION

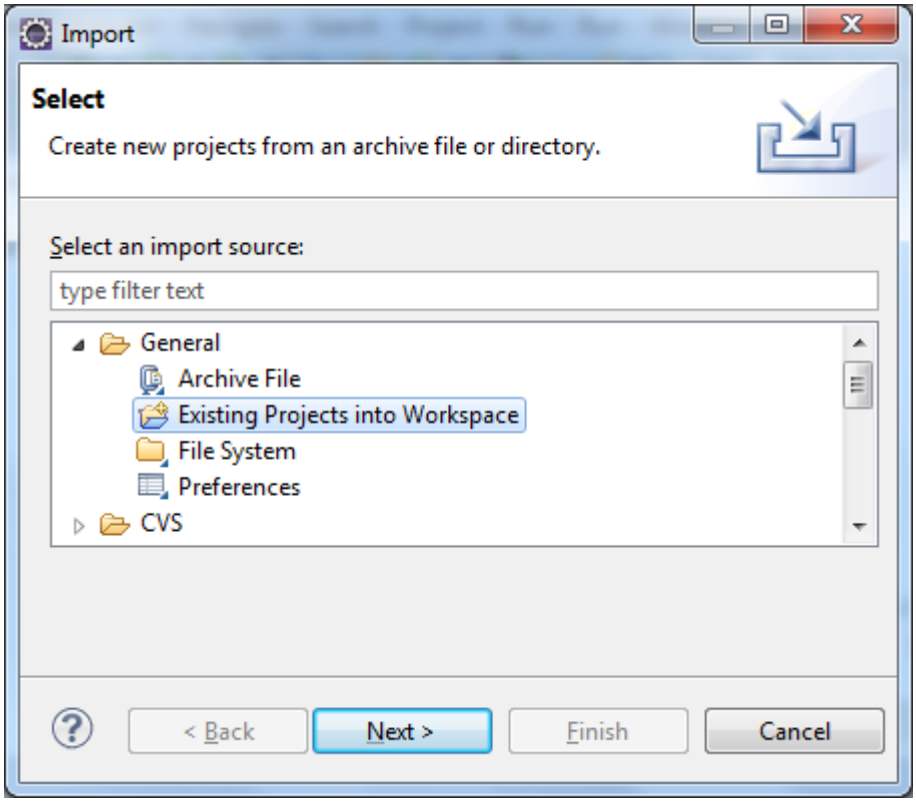
3.1 Creating Folder Structure

SwiftLite framework is provided as Java project containing requisite files, libraries, folders, etc.

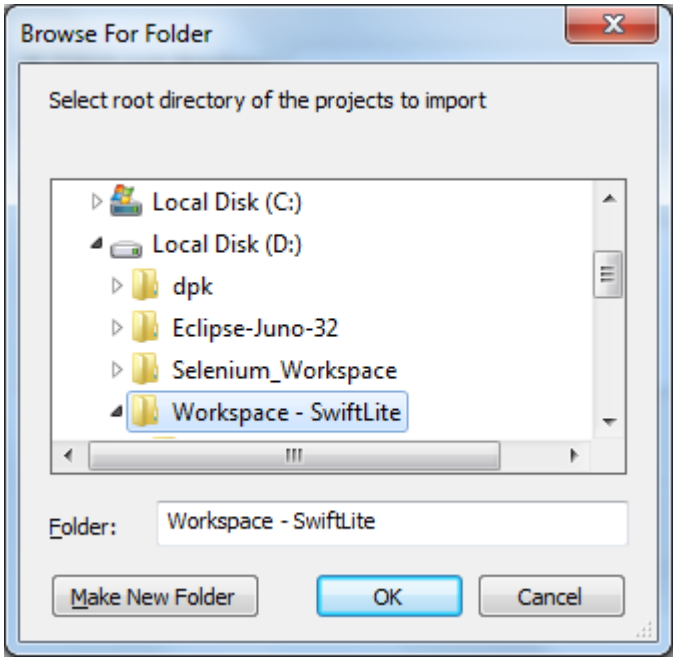
Now create a workspace folder (let's say 'Workspace - SeleniumAutomation') in root drive and download the entire SwiftLite folder as a project into the created workspace.

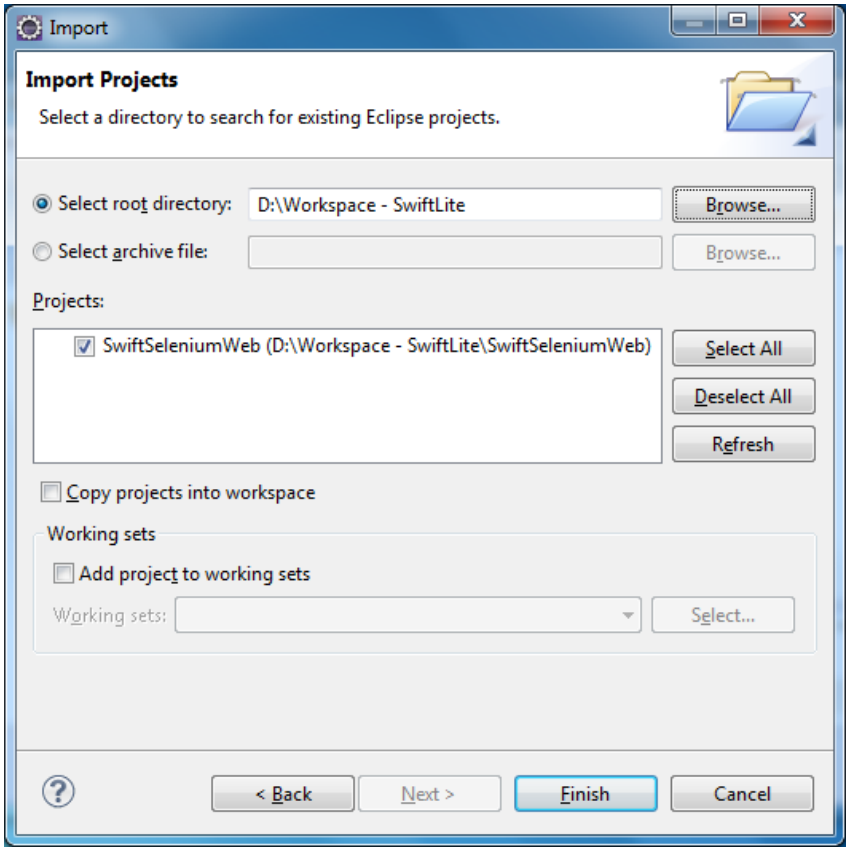
3.2 Creating Java Project

Once the folders are extracted, open Eclipse and select the above create workspace. Now import the above project (File >> Import) into the workspace as shown in the screenshots below:



Select the option as shown above and click 'Next'. Then browse to the workspace and click 'OK'.

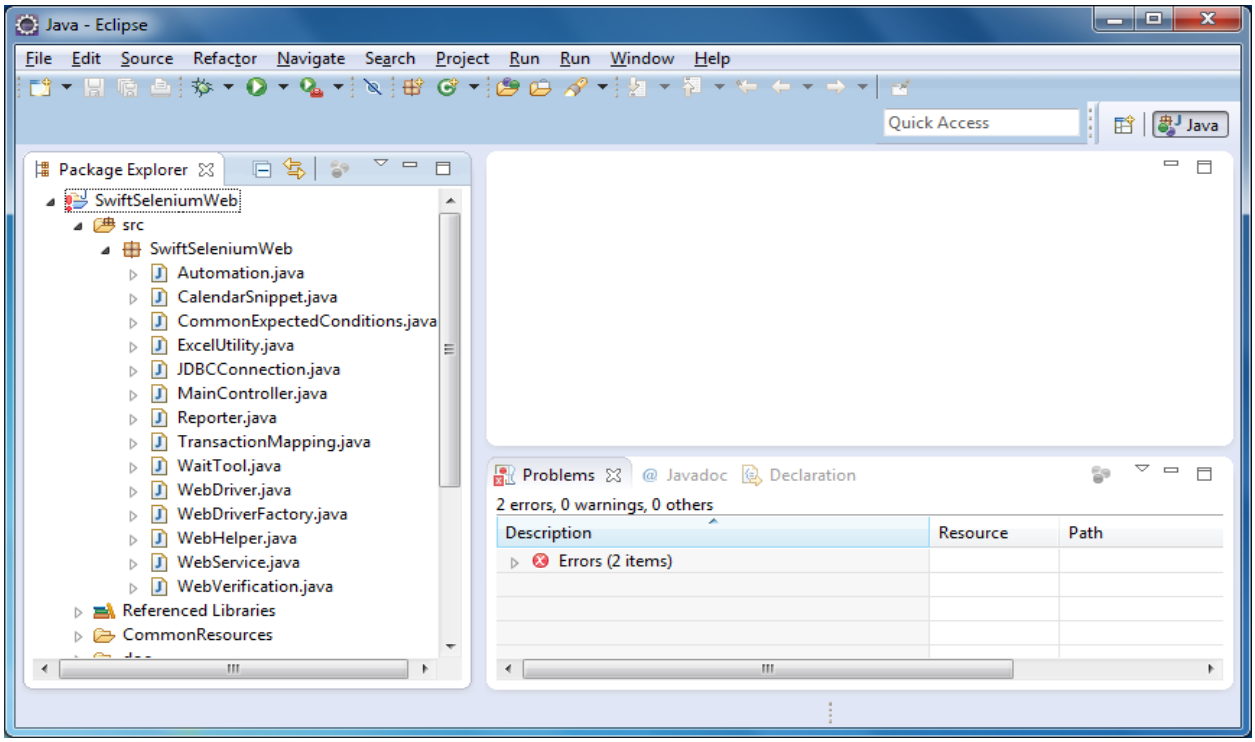


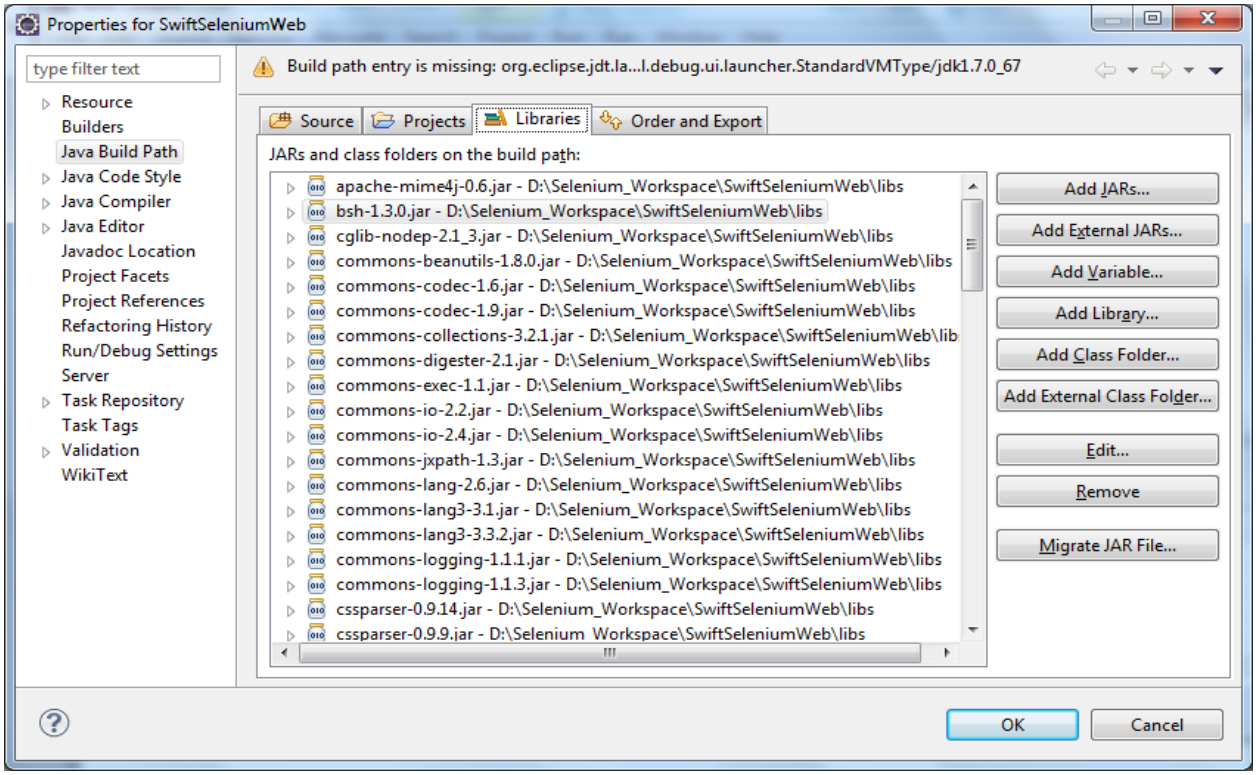


Click ‘Finish’.

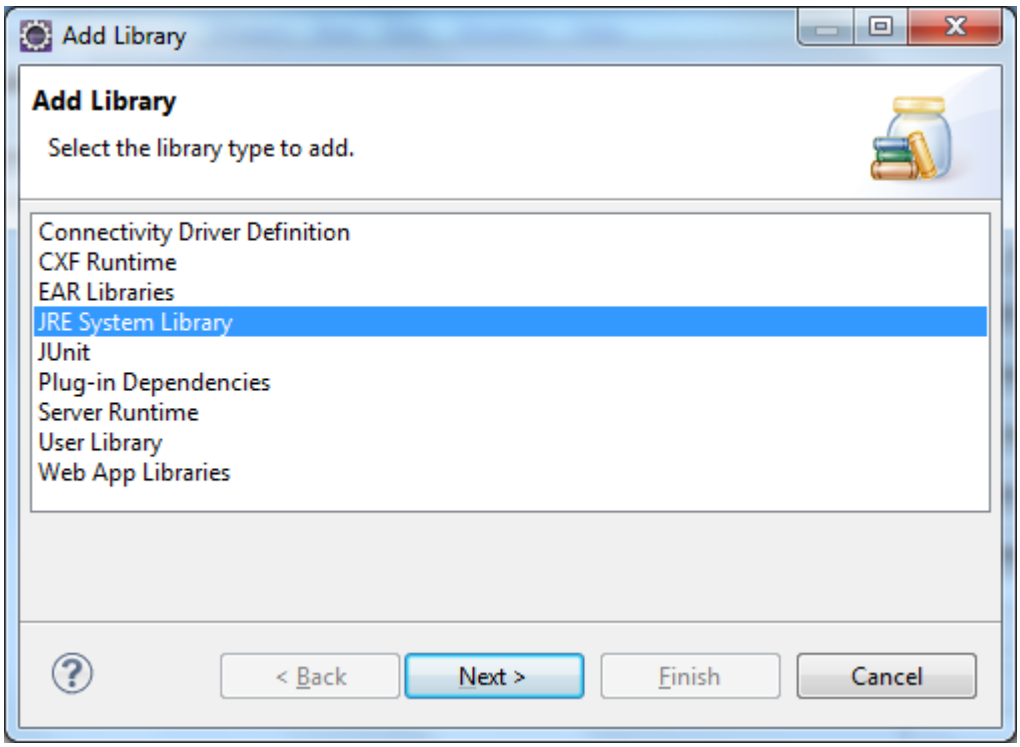
Once the project gets imported, the console will show a few errors. This is because the incorrect referenced library path saved from source build location.

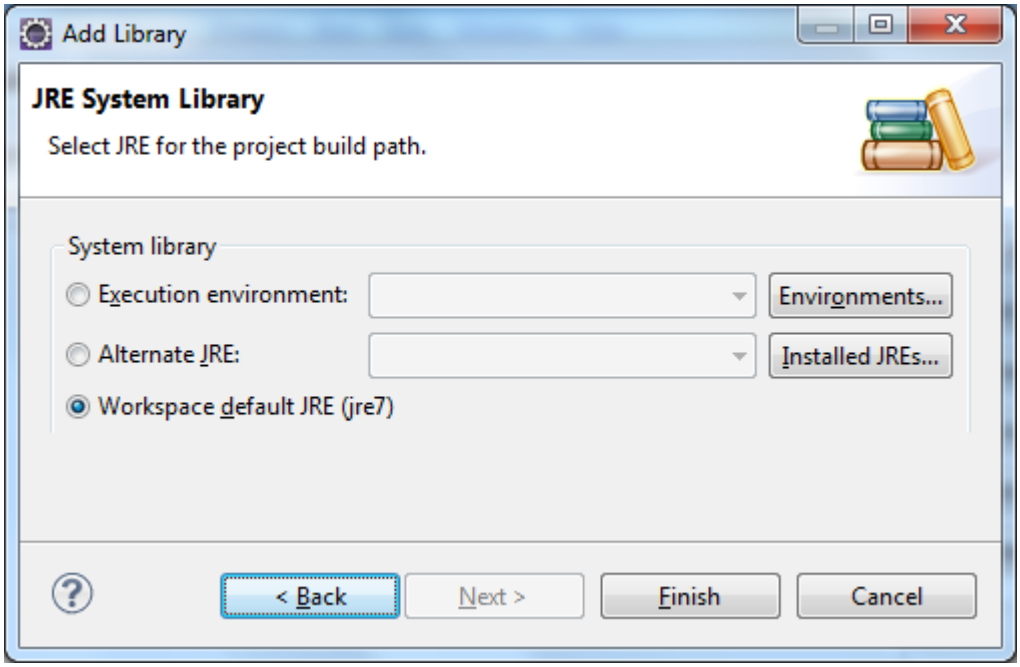
So change the referenced library path by clicking on the 'SwiftSeleniumWeb' project and select Build Path >> Configure Build Path >> Libraries tab as shown in the screenshot below:



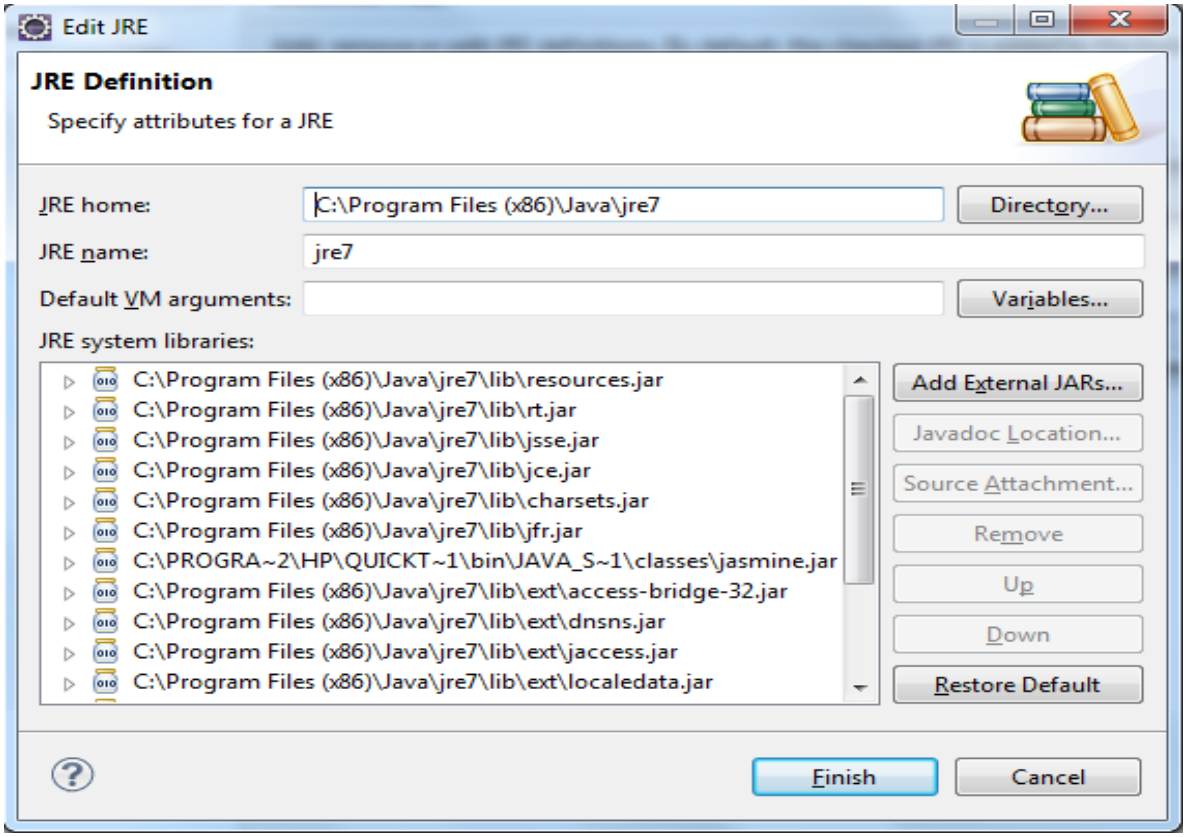


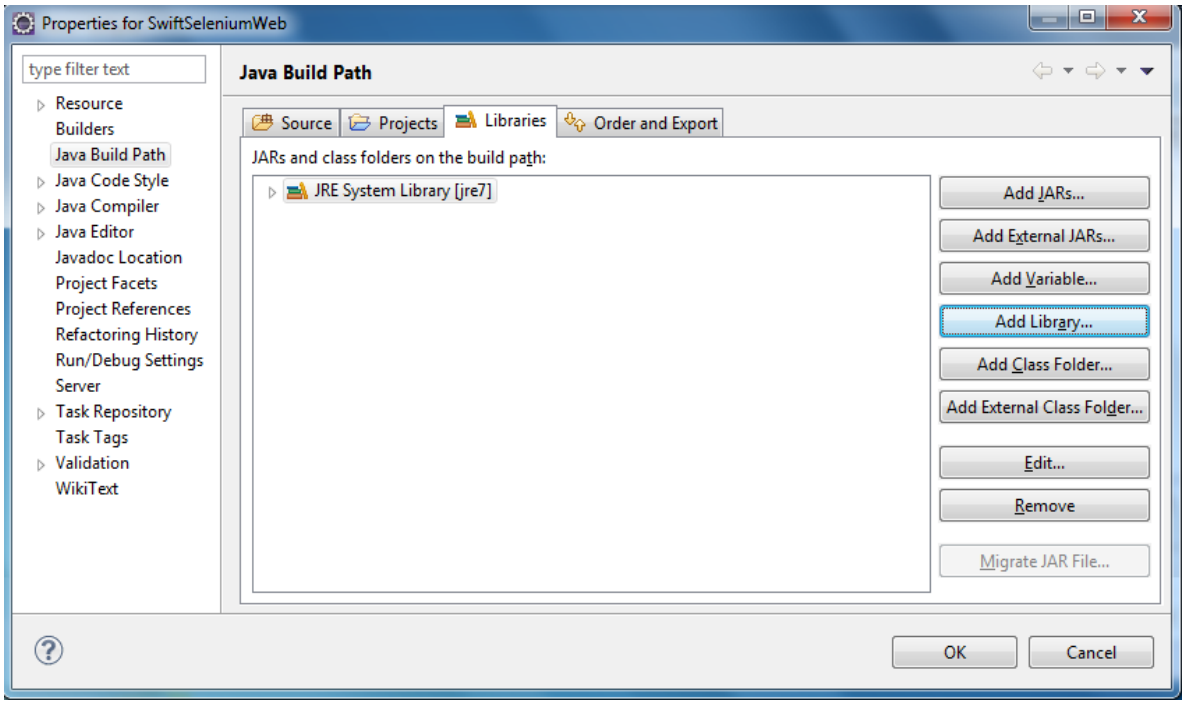
Select all the libraries (Ctrl A) and click on the ‘Remove’ button. Now click on ‘Add Library’ button:



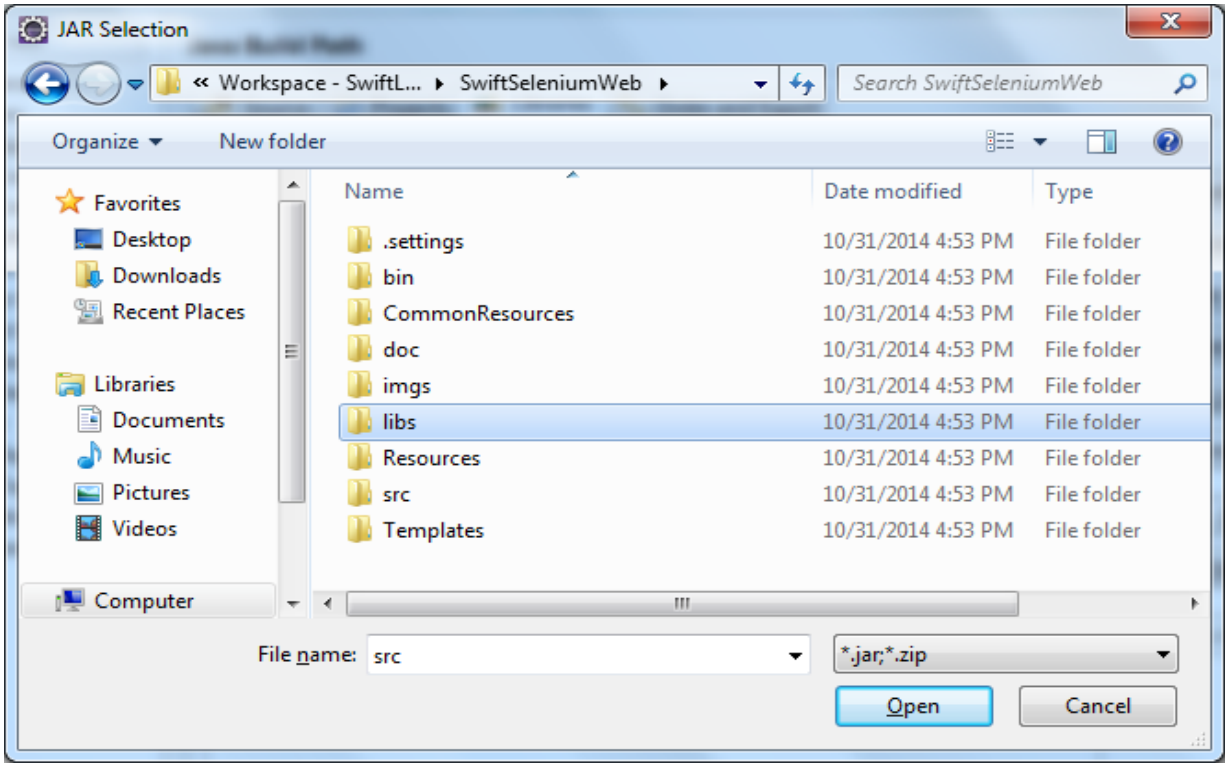


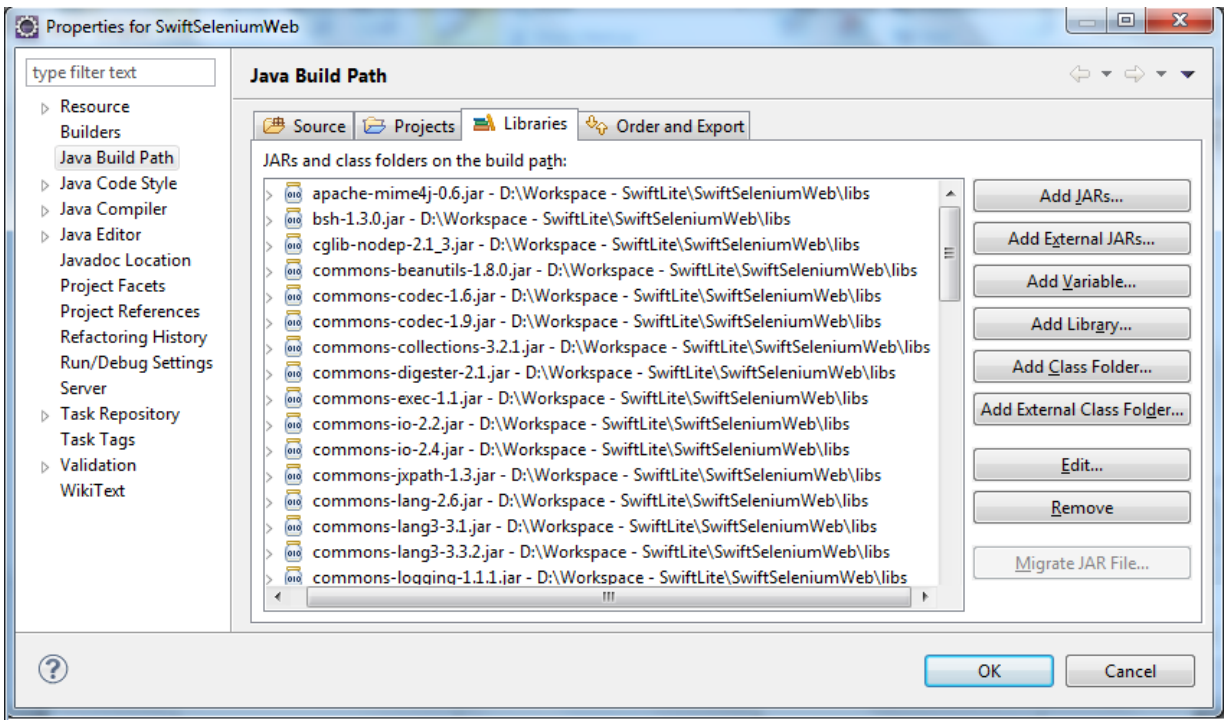
Select ‘Aletrnate JRE’ option and click on ‘Installed JREs’ button and slect the JRE installed on your machine.



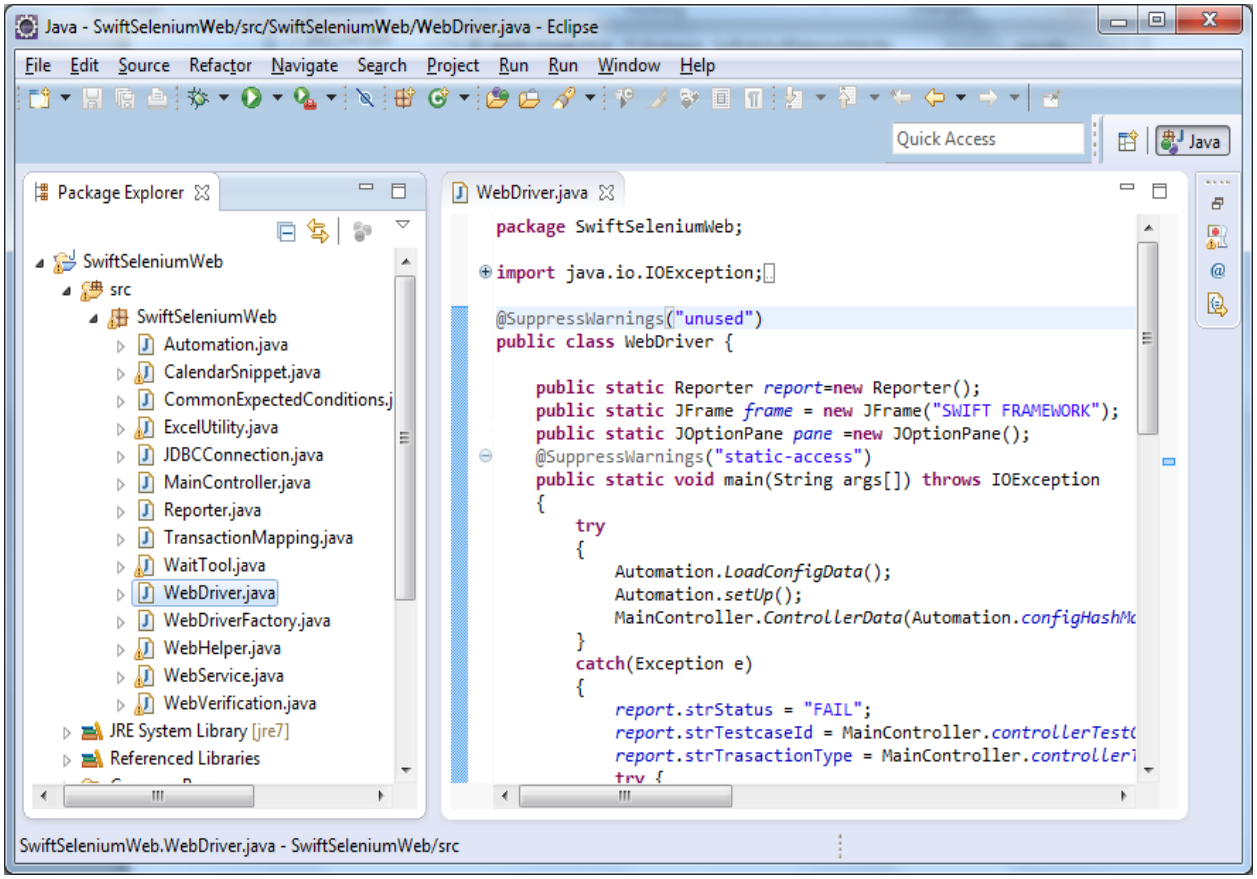


Now click on the 'Add External JARs' button and browse to the 'libs' folder within the 'SwiftSeleniumWeb' project and select all the files in that folder and click 'OK'.





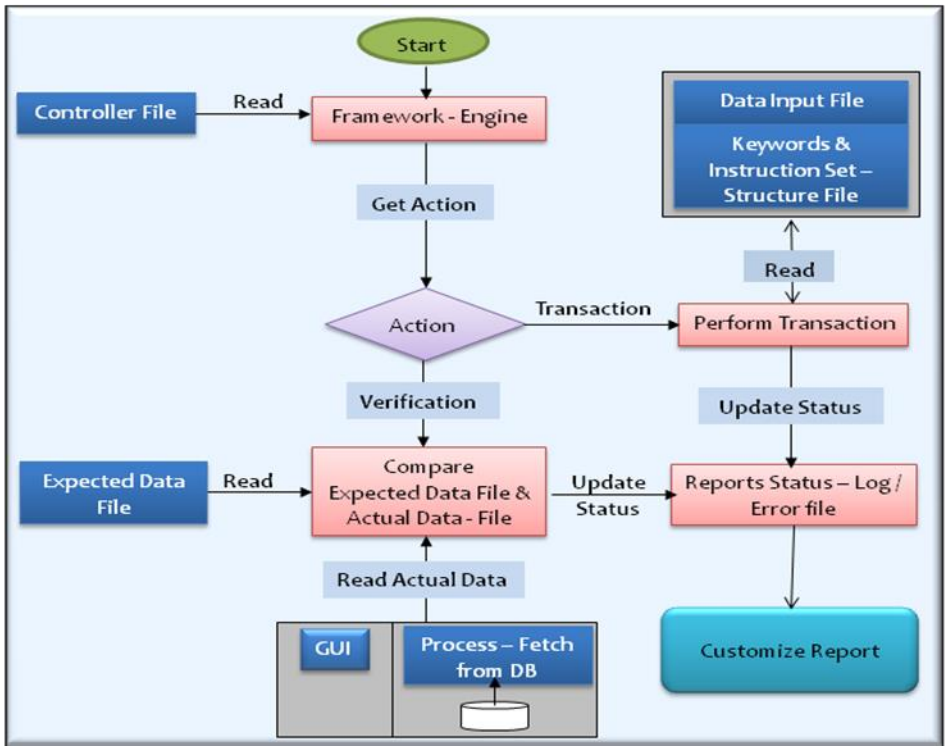
Now click on the 'OK' button and all the error messages will disappear and only a few warnings may remain, which can be ignored. The build is ready to use. Execution starts with the file 'WebDriver.java'.


Java - SwiftSeleniumWeb/src/SwiftSeleniumWeb/WebDriver.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Run Window Help
Quick Access
Java
Package Explorer
SwiftSeleniumWeb
src
SwiftSeleniumWeb
Automation.java
CalendarSnippet.java
CommonExpectedConditions.j
ExcelUtility.java
JDBCConnection.java
MainController.java
Reporter.java
TransactionMapping.java
WaitTool.java
WebDriver.java
WebDriverFactory.java
WebHelper.java
WebService.java
WebVerification.java
JRE System Library [jre7]
Referenced Libraries
WebDriver.java
package SwiftSeleniumWeb;
import java.io.IOException;
@SuppressWarnings("unused")
public class WebDriver {
public static Reporter report=new Reporter();
public static JFrame frame = new JFrame("SWIFT FRAMEWORK");
public static JOptionPane pane =new JOptionPane();
@SuppressWarnings("static-access")
public static void main(String args[]) throws IOException
{
try
{
Automation.LoadConfigData();
Automation.setUp();
MainController.ControllerData(Automation.configHashM
}
catch(Exception e)
{
report.strStatus = "FAIL";
report.strTestcaseId = MainController.controllerTest
report.strTrasactionType = MainController.controller
SwiftSeleniumWeb.WebDriver.java - SwiftSeleniumWeb/src

4 FRAMEWORK DETAILS

4.1 Flowchart

The following schematic represents the functional flow of the automation framework



4.2 Folder Structure

Swift framework uses a specific folder structure for arranging the key components of the automation suite. The folder hierarchy aids the framework in attaining the expected functions. The following section explains the main components of the automation suite folder structure.

4.3 Key Components

Src

This folder contains the java code files which form the machinery of the Swift framework. **WebDriver.java** is the test execution file from where the execution starts. This is also the file to keep the custom code for any complex transactions.

Common Resources

This folder stores all the driver spreadsheets and configuration files.

1. Config.xls is the configuration file that captures commonly used values for variables/constants. This allows adding and changing the variables, without the need to change the code or scripts. It also helps in creating a common location for most common variables as well as keeping the code independent of machines/servers. The file paths are stored in different environment variables, which will then be called via the automation scripts. These paths may need to be changed depending on the workspace folder path directory.

2. MainController.xls is the driver spreadsheet that controls the flow of execution. Each row in the Main Controller represents a test case or Scenario and each column defines corresponding transactions to complement the test case actions or steps. The flow of execution is from left to right and then top to bottom.

Each of these transaction keywords perform a set of specific actions defined in Transaction Input File (described later).

Each row in the Main Controller represents a test case or Scenario and each column defines corresponding transactions to complement the test case actions or steps. The flow of execution is from left to right and then top to bottom.

GroupName	TestCaseID	Test_Description	Execute				
Smoke	TC1	Registration	Y	**START**	Register	MercuryLogoff	MercuryLogin
Smoke	TC2	FlightBooking	Y		MercuryLogin	FlightBooking	TicketBooking
Regression	SC2	Masteknet Search	N		MastekSearch	VerifySearchResults	
Regression	SC3	Check flight records	Y		MercuryLogoff	MercuryLogin	FlightBooking
Integration	PC5	Verify travel data	N		MercuryLogin	FlightBooking	VerifyTravelEntry
Integration	TC5	Web Service	Y		Shakespeare_WS	**PAUSE**	

Column Description:

1. **GroupName**: This is type or category of the test case, or scenarios, like Smoke, Deployment etc. This determines the grouping of the results at the time of execution.

2. **TestCaseID**: This is the Test case or Scenario identifier helpful to identify the input or verification for respective Transaction Type to be specified in the same row later. This is required during execution as well as when results are being written. The Test Case ID should be unique and should not be repeated.

3. **TestDescription**: This is just an informatory field and defines the purpose of the test case.

4. **ExecuteFlag**: This column value determines whether a test case should be executed or not. A value of **'Y'** means that the corresponding row (test case) should be executed and a value of **'N'** means that the corresponding row is to be completely ignored during execution.

5. **User defined keywords**: These keywords can start from the 6th column i.e. column 'F'. Each of these keywords performs a specific action, which is defined in the Transaction Input File.

Apart from these, there are two tags:

1. **START**

2. **PAUSE**

The execution starts from the START tag and continues (horizontally i.e. row-wise) till the PAUSE tag is reached. These tags can be placed anywhere beginning from the 6th column i.e. column 'F' onwards. The execution begins from the first START tag and stops at the first PAUSE tag. So if a test case is to be executed, it must have the Execute Flag as 'Y' and should fall within the START and PAUSE tags.

Since the entire flow of execution is maintained in a spreadsheet in the form of user defined keywords, a new test case execution can be easily configured, or an existing one amended, if the user has the knowledge of the required transaction keywords.

3. Transaction_Input_Files.xls determines what action a transaction keyword used in the Main Controller spreadsheet would perform. When the framework encounters a transaction keyword in the Main Controller, it searches for its associated file/action in the Transaction Input File

Created	TransactionCo	TransactionType	DirPath	InputSheet
RP	Login	LoginMercury	Login	Login.xls
RP		SelectFlight	FlightReservation	SelectFlight.xls
RP		BookFlight	FlightReservation	BookFlight.xls
RP		Confirmation	FlightReservation	
RP		Register,UpdateRegistration	FlightReservation	Register.xls
RP		VerifyFlightBooking		
DR		VerifyItenary		
DR		EmpSearch	MastekEmpSearch	EmpSearch.xls
DR		VerifyEmpSearch		

Keywords are associated with one of the following actions

1. Data Input
2. Verification (Not included in 'Swift-Lite' build)

A transaction keyword lets framework know what needs to be done as part of invoking that transaction type. Keywords are associated with one of the following actions, apart from also containing steps to be performed for navigating through the application:

1. **Data Input**: In this case, the transaction keyword points to **Input Datasheets** (described later) which contain data/values to be entered on a particular screen or page of the application. Field level verifications can also be done using these sheets.

   Example: A keyword 'CreateStudent' will contain instructions for creating a new student and also contain all the requisite input values (data) to be entered in the process of student creation.

2. **Verification**: In this case, the transaction keyword starts with prefix "Verify..." points to Verification Datasheets (described later). These keywords will invoke comparison of web table data based on the expected sheets and actual application values. These keywords will also publish the comparison results for the execution.

   Example: The keyword 'VerifyAllStudentDetails' will invoke verification of all student details captured from application and compare them with expected sheets stored in the verification folder. The results of this comparison will then be published in summary and detailed results for users to validate/authenticate the run results.

   ***Note:** This feature is not included in 'Swift-Lite' build.*

There are 4 columns in the Transaction Input File:

1. **Created By**: This column is not functional for documentation use only.
2. **Transaction Code**: This column is not required to be edited or entered by end users/SME. Internally used by automation framework only.
3. **Transaction Type**: This is same as the transaction keyword in the Main Controller file. Each transaction type should be unique, but can be associated with multiple datasheets.
4. **Dir Path**: This defines the directory path of the datasheets associated with a particular transaction type. This column lets the framework know which sheet has to be considered for data input actions and screen navigations. In case of verification transaction types, these fields may remain blank if there are no data entry or screen navigations happening for a particular transaction keyword.
5. **Input Sheet**: This column will contain the name of the spreadsheet associated with the transaction keyword. The value in this field will form the basis of data input or screen navigations for the application as well as providing specific selection/input criteria.

4. VerificationTableList.xls holds the information of the path and names of templates, expected results and actual results. All keywords with verification associated are listed in this sheet. These keywords invoke comparison of data from the expected sheets and actual application values. These keywords also publish the comparison results for the execution.

***Note:** This feature is not included in 'Swift-Lite' build.*

TransactionType	VerificationType	TemplateAddition	TemplateSheet	ExpectedDataAdditior	ExpectedDataSheet
VerifyBillingEnquiry	VerifyFlightDetails	BID	TC_POLINQ_014_1.xls	BID	VerifyFlightDetails.xls
VerifyFlightDetails	VerifyFlightDetails	FlightBooking	Flight_Confirmation.xls	SearchResults	VerifyFlightDetails_Expected.xls
VerifyCruise	VerifyCruise	FlightBooking	Cruises.xls	SearchResults	VerifyCruise_Expected.xls
VerifySearchResults	VerifySearchResultsU	FlightBooking	SearchResultsUniform.xls	SearchResults	VerifySearchResults_Expected.xls
VerifyItenary	Verify Flight Itenary	FlightBooking	CruiseItinerary.xls	FlightBooking	CruiseItinerary_ExpectedValues.xls

Resources

This folder contains 3 types of data and hence is sub-divided into 3 folders:

Input

This folder contains the input data sheets which co-relate to a particular screen or a group of screens in the application. You can further create sub-folders here that separate different modules of an application.

The input files define the execution flow on a particular screen or group of screens. These files contain navigation steps and expected values for input fields during the navigation. Each input sheet has 2 worksheets – 'Structure' and 'Values'. When a transaction input file that contains keywords and path invokes an input file, Selenium will look at the 'Structure' sheet for the specific input scenario and follow the navigation steps. 'Values' sheet contains field values that are to be considered/ inserted in specific instances.

Structure sheet:

ExecuteFlag	Action	LogicalName	ControlName	ControlType	ControlID
Y	NC	Register	REGISTER	WebLink	LinkText
Y	I	FirstName	firstName	WebEdit	Name
Y	I	LastName	lastName	WebEdit	Name
Y	I	Phone_No	phone	WebEdit	Id
Y	I	eMail	userName	WebEdit	Id
Y	I	Pin_code	postalCode	WebEdit	Name
Y	I	Country	country	WebList	Name
Y	I	UserName	email	WebEdit	Name
Y	I	Password	password	WebEdit	Name
Y	I	ConfirmPwd	confirmPassword	WebEdit	Name
Y	NC		register	WebButton	Name
Y	V	VerifyText	//p/font/b	WebElement	XPath

Values Sheet:

TestCaseID	TransactionType	FirstName	LastName	Phone_No	eMail	Pin_code	Password	ConfirmPwd	UserName	VerifyText
TC1	Register	Deepak	Sharma	987654321	DeepakS@abc.com		123pass	123pass	Test	Dear Deepak
TC2	Register	Imran	Khan	123456789	ImranK@abc.com	223344	pass123	pass123	Test1	Dear Imran

Object identification is done using the data in columns **'ControlId'** and '**ControlName'**. So if control id is "LinkText" and control name is "REGISTER", the framework looks for an object on the screen whose "LinkText" is "REGISTER".

What type of action is to be taken on the identified object is defined by the column '**ControlType'**. So if control type is "WebLink", a click operation is performed and for "WebEdit", text is entered.

'**LogicalName'** column forms the link between the Structure and the Values sheet. What data is entered for which control is determined by the logical name. Hence, Logical name should be kept as similar as the object's screen name so that data entry becomes easy.

'**Action'** specifies the action to be carried out. Action 'NC' stands for navigation control meaning the values from other columns in that row specify where the application is supposed to navigate. No data needs to be specified for a 'NC' action and this action will always be carried out once specified.

Action 'I' stands for Input, where input value for the parameter is specified in the 'Values' sheet. Every 'I' in the structure sheet must have a value specified or the framework will skip it and not perform any operation for that control.

Action 'V' is for field level verification i.e. verification of the value in a particular control like a label, a web edit etc. The actual value mis captured from the screen and is compared with the expected value specified in the 'Values' sheet.

'**ExecuteFlag'** column specifies if the row needs to be considered or not. 'Y' means the row needs to be considered and 'N' means that the row should be skipped from execution.

Verification

This folder stores the expected values sheet for table verification as per the test criteria in the folder specified in the verification table list for a given transaction type. This is the path where the actual data file created during execution will be stored for comparison.

***Note:** This feature is not included in 'Swift-Lite' build.*

Results

This folder stores the execution results. There are 2 result files generated at the end of execution – Report.csv and DetailedResults.csv. The Report file contains one row per transaction of a test case and shows which transaction in a test case passed or failed. The 'DetailedReport' file contains specific verification results for both field level verification and table verification.

Report file:

GroupName	Iteration	TestCaseID	TransactionType	TestCaseDesription	StartDate	EndDate	Status
Smoke	Cycle3	TC1	Shakespeare_WS_1	Web Service	7/31/2014 19:35	7/31/2014 19:35	PASS
Smoke	Cycle3	TC1	Shakespeare_WS_1	Web Service	7/31/2014 19:35	7/31/2014 19:35	PASS
Regression	Cycle3	TC2	CreateProduct	Products Menu	9/12/2014 10:48	9/12/2014 10:49	FAIL
Regression	Cycle3	TC2	CreateProduct	Products Menu	9/15/2014 19:36	9/15/2014 19:36	FAIL
Regression	Cycle3	TC2	CreateProduct	Products Menu	9/15/2014 19:39	9/15/2014 19:39	PASS
Regression	Cycle3	TC2	**PAUSE**		9/15/2014 19:39	9/15/2014 19:39	

Detailed Results file:

Iteration	TestCaseID	TransactionType	CurrentDate	RowType	Status	PassCount	FailCount	Comment
Cycle3	SC1_TC1	AddEntry	10/10/2014 9:37	Field: VerifyHeader	FAIL	0	1	FAIL|Imran|Deepak
Cycle3	SC1_TC1	UpdateEntry	10/10/2014 9:37	Field: VerifyHeader	PASS	1	0	VIP Database Test Web Application

Templates

This folder contains templates which act as a blue print for a web table that needs to be verified for its data contents. The template sheet holds information for identification of the table or the fields inside the table. Once the table is identified on the screen, data is extracted from it and stored as actual data and is then compared with expected data.

Below are the screenshot of a uniform table verification template.

TableType	TableID	TableIDType	StartRow	EndRow	ColumnName	Row	Column
Uniform	//div[@id='StaDiv']/div[2]/table	XPath	1		Status		0
					TransEffDate		1
					CancelType		2
					CancelMethod		3
					NoticeAmount		4
					CancelReason		5

The 'TableType' can be Uniform or Non-Uniform. 'TableIDType' and 'TableId' together identify the web table. 'Column' number determine which columns need to be considered for data extraction. 'ColumnName' is like the logical name in the input data sheets and is user defined matching the screen column names and act as a link with the column names in the expected data sheets. 'StartRow' determines which row to start with and 'EndRow' should be entered if certain rows from the end need not be extracted.

Once the actual data sheet is created in the 'Verification' folder, as stated in previous section, its compared with the expected data sheet present in the same folder and results are published in the 'Results' folder as explained above.

***Note:** This feature is not included in 'Swift-Lite' build.*

5. EXECUTION

There are two modes of execution: Interactive and Batch mode. The Config.xls parameter 'USERINTERACTION' determines the mode of execution.

5.1 Interactive Mode

If 'USERINTERACTION' is set to "TRUE", execution happens in interactive mode. Alerts are displayed on screen in case of any issues and execution does not proceed unless a response is provided to the alerts.

5.2 Batch Mode

If 'USERINTERACTION' is set to "FALSE", execution happens in non-interactive mode. In case of any issues during execution, the error message is logged in the result files and the execution proceeds with the next test case. This mode is not recommended till the automation suite has reached some maturity or stability.

AMENDMENT HISTORY

Version No.	Date	Amendment History	Remarks
1.0	07/11/2014	Baseline	Baseline version