**Framework User Guide**

# Web UI Automation

To automate Web UI scenarios, follow the below steps:

1. Creation of Feature File

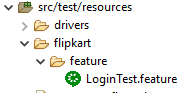
2. Creation of Page Class

3. Creation of Step Definition

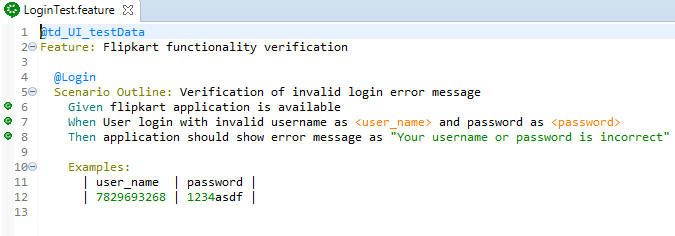
4. Run script from “RunCucumberTest” class

## Creation of Feature File

1. Create application folder under *‘src/test/resources’* (here flipkart).
2. Under application folder create another folder *‘feature’*. This folder will contain all the feature files.



1. Create feature file with Feature, Scenario and Scenario Outline (application specific).
2. (optional) Test data is loaded per feature. To enable the same, user need to mention feature name (should be the primary key from DB) prefixed with ‘@td\_’ above ‘Feature:’ key word. For example: @td\_UI\_Flipkart\_Login
3. Browser can be open and close on the need base.
4. To open a browser, use generic statement *“Given user launches the browser”* at the beginning of the scenario.
5. To close a browser, use generic statement *“And user closes the browser”* at the end of scenario.
6. Provide respective tags like @Smoke, @Regression etc. on Scenario / Scenario Outline.
7. Limit the actions in each step to one single UI page or one API/end-point. If a step is performing actions on more than one page, split the step into multiple steps. This will help with maintaining step definitions at page or API level.

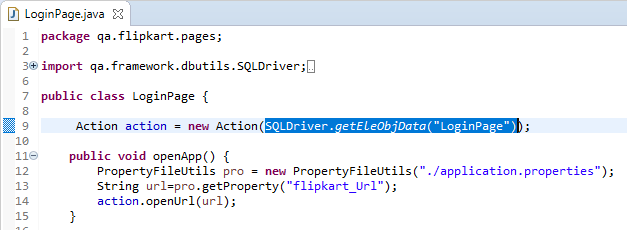


## Creation of Page Class

1. Create Page package under ‘src/test/java’ in format “qa.<application name>.pages”. This will contain all the page classes in the application.



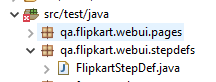
1. Since framework is Page Object Model (POM) inspired, each web application page has its own class.
2. To load the web element attributes from the DB, respective page name should be provided in the following method: SQLDriver.getEleObjData("page name").

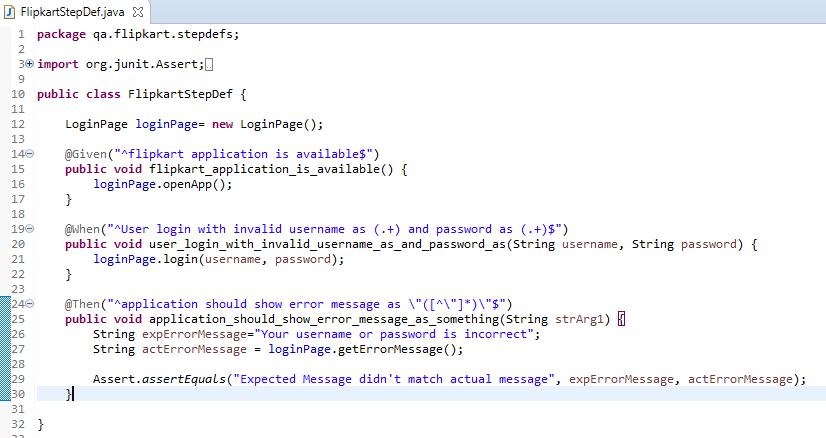


1. Create an instance of ‘qa.framework.utils.Action’ class. Action class will be used for all the web actions performed on web page like openUrl(), click(), sendkeys() etc. The constructor of Action takes [List](eclipse-javadoc:%E2%98%82=LEAF/src%5C/main%5C/java=/optional=/true=/=/maven.pomderived=/true=/%3Cqa.framework.dbutils%7BSQLDriver.java%E2%98%83SQLDriver~getEleObjData~QString;%E2%98%82java.util.List)<[DBRowTO](eclipse-javadoc:%E2%98%82=LEAF/src%5C/main%5C/java=/optional=/true=/=/maven.pomderived=/true=/%3Cqa.framework.dbutils%7BSQLDriver.java%E2%98%83SQLDriver~getEleObjData~QString;%E2%98%82qa.framework.dbutils.DBRowTO" \o "in qa.framework.dbutils)>, which is return by SQLDriver.getEleObjData("page name") method (shown in above image).
2. Page class have all the methods interacting with web page like fillSignUpForm(), getSuccessMessage() etc.
3. It is recommended to have only the actions in page methods and all verification steps should ideally be part of step definition methods.

## Creation of Step Definition

1. Create Step Definition package under ‘src/test/java’ in format qa.<application name>.stepdefs. This will contain all the gherkins step definition in an application.

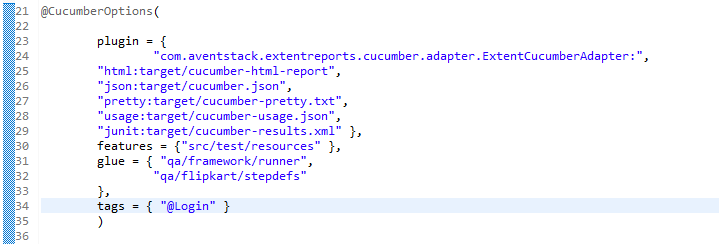


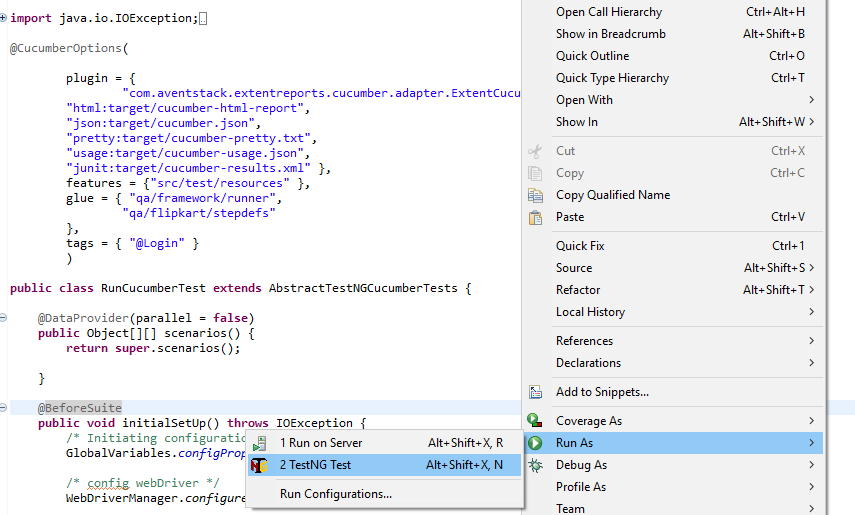
1. Create step definition class suffixed ‘StepDef’. This class will contain gherkin step definitions. 
2. Step definition normally calls page methods specified in page class and all the verifications are done here. This way one can clearly identify what steps are followed, and what verifications are done.
3. Recommendation is to create step definition classes at page level for UI and one step definition class for each API/ end point.
4. If user want to at additional message to Extent report can use qa.framework.utils.Reporter class.



## Run script from “RunCucumberTest”

1. Open RunCucumberTest class available in ‘src/test/java’.
2. Under @CucumberOptions ‘glue’ section mention step definition package (with dot {.} replaced with forward-slash {/}).
3. Under @CucumberOptions ‘tag’ mention scenario tags which user want to execute.



1. To run Right click on class > Run As > TestNG

Note: Do not change anything in plugin, features section.

# API Automation

Follow the below steps for API automation:

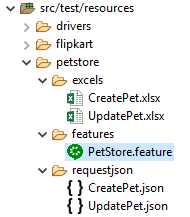
1. Creation of Feature File

2. Creation of Step Definition

3. Run script from “RunCucumberTest” class

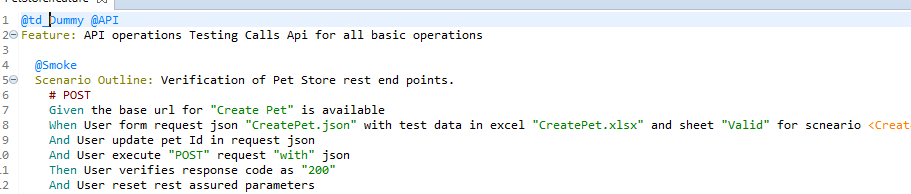
## Creation of Feature File

1. Create application folder under *‘src/test/resources’* (here petstore).
2. Under application folder create following sub-folders: *‘features’, ‘excels’ and ‘requestJson’*. These folders will contain feature files, excel and request json template respectively.



1. Create feature file with Feature, Scenario and Scenario Outline (application specific).
2. Provide *‘@API’* tag above ‘Feature:’ key word. This tag enables framework to take care of pre-requisites. In case of API, resets ‘RestAssured’ static variables.
3. (optional) Test data is loaded per feature. Mention the feature name (primary key from DB) prefixed with ‘@td\_’ above ‘Feature’ key word.

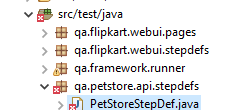
For example: @td\_API\_PetStore\_DummyTestData

1. Provide respective tags like @Smoke, @Regression etc. on Scenario / Scenario Outline. 
2. Explaining above scenario:

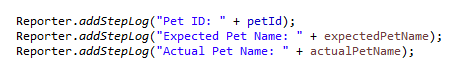
* Given is responsible for setting base URI and base path.
* When is responsible for creating request json dynamically using test data from workbook (here CreatePet.xlsx) and JSON template (here CreatePet.json).
* ‘And User execute “POST” request “with” json’, this statement executes different HTTP request like GET, POST, PUT etc. If request requires json, we can mention ‘*“with” json*’ else ‘*“without” json’*
* Then, request code is verified in this step (here 200).
* ‘And User resets rest assured parameters’ this is a **mandatory** step to be written after one endpoint is executed, verified and before we are setting base URI for another endpoint.

## Creation of Step Definition

1. Create Step Definition package under ‘src/test/java’ in format qa.<application name>.api.stepdefs. This will contain all the gherkins step definitions of the application.

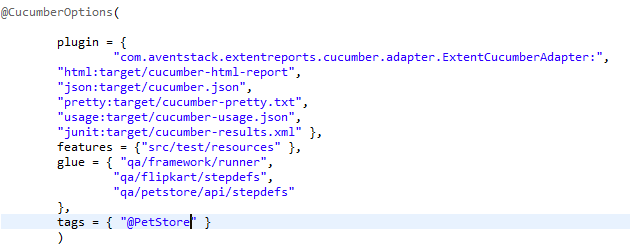


1. Create step definition class suffixed ‘StepDef’. This class will contain gherkin step definitions. 
2. qa.framework.restutils.RestApiUtils class has all the necessary functions required to create http requests and execute the same.
3. If user wants to add additional message to Extent report, use qa.framework.utils.Reporter class.



## Run script from “RunCucumberTest” class

1. Open RunCucumberTest class available in ‘src/test/java’.
2. Under @CucumberOptions ‘glue’ section mention step definition package (with dot {.} replaced with forward-slash {/}).
3. Under @CucumberOptions ‘tag’ mention scenario tags which user want to execute.



d. To run Right click on class > Run As > TestNG

# LeanFT Automation Test:

Mainframe (Green Screen) automation is achieved with the combination of IBM PCOMM and LeanFT tools. IBM PCOMM is a mainframe emulator and LeanFT is a functional test automation tool. Installation and configuration step are available in *‘InstallationGuide v0.2’* document.

**Follow the below steps for API automation:**

1. Creation of Feature File

2. Identify Object using LeanFT Spy

3. Creation of different mainframe screens.

4. Creation of Step Definition

5. Enable Mainframe feature

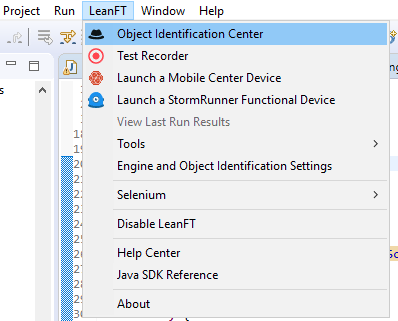
6. Run script from “RunCucumberTest” class

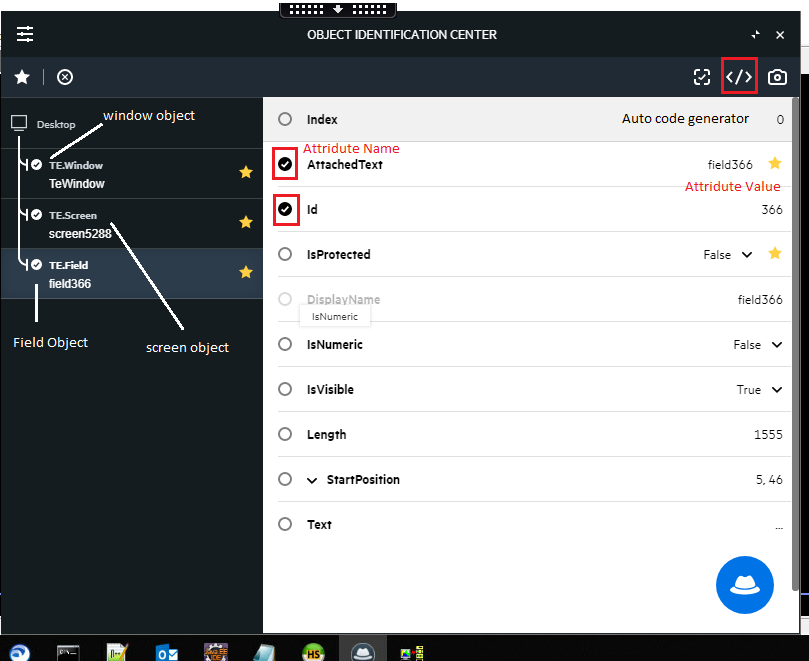
## Creation of Feature File

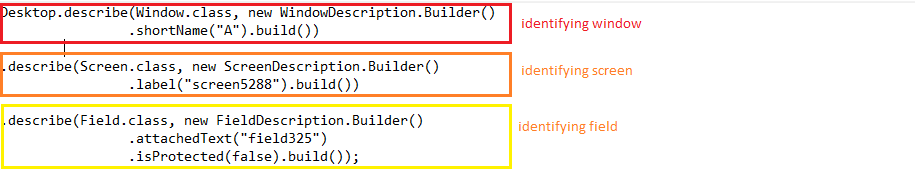
1. Create application folder under *‘src/test/resources’*.
2. Under application folder create following sub-folders: *‘features’.*
3. Create feature file with Feature, Scenario and Scenario Outline (application specific).
4. Provide *‘@MAN’* tag above ‘Feature:’ key word. This tag enables framework to take care of pre-requisites. In case of mainframe, starting LeanFT server.
5. (optional) Test data is loaded per feature. Mention the feature name (primary key from DB) prefixed with ‘@td\_’ above ‘Feature’ key word. For example: ‘@td\_MAN\_MainframeApplication\_DummyTestData’. Provide respective tags like @Smoke, @Regression etc. on Scenario / Scenario Outline.

## Identify Object using LeanFT Spy

1. Open Eclipse > LeanFT > Object Identification Centre



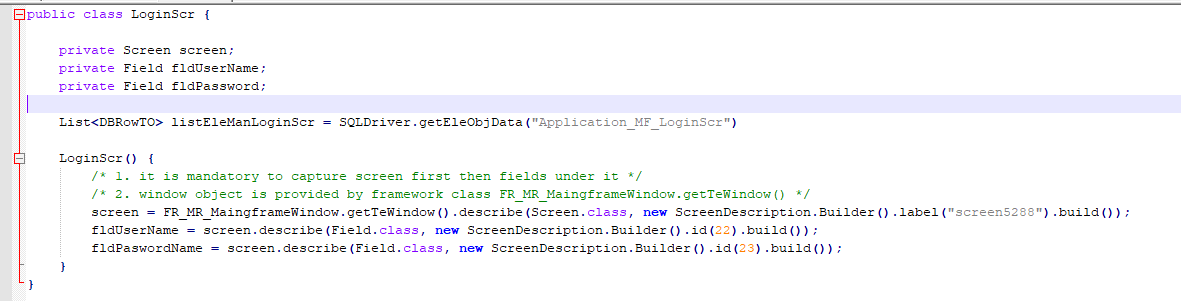
1. Click on ‘Click to Start Spying’ > Select a field in Mainframe screen. 
2. Above image illustrate the hierarchy in which object are maintained in mainframe application, Window > Screen > Field. Once user spies any object the respective attributes are shown in right panel like Attached Text, id etc. ‘Object Identification Centre’ also provide auto code generator option which provide descriptive code to identify any object. (shown in below image)



Note: this hierarchy should be maintained while writing code.

## Creation of different mainframe screens

1. Create Screen package under ‘src/test/java’ in format qa.<application name>.man.screen. This will contain all the mainframe screens.
2. Create a class (say LoginScr). And capture screen and fields under a constructor.



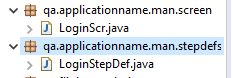
Note: 1. ‘Screen’ should be captured first and then the fields under it.

2. ‘Window’ element is provided by ‘FR\_MR\_MaingframeWindow.getTeWindow()’ class/method of framework.

1. Also screen class should not only contain the respective elements but also method to perform on it (like POM in Selenium).
2. ‘LeanFTAction’ class is provided to perform different actions like setText(), getText(), sendTeKeys() etc.

## Creation of Step Definition

1. Create Step Definition package under ‘src/test/java’ in format qa.<application name>.man.stepdefs. This will contain all the gherkins step definitions of the application.



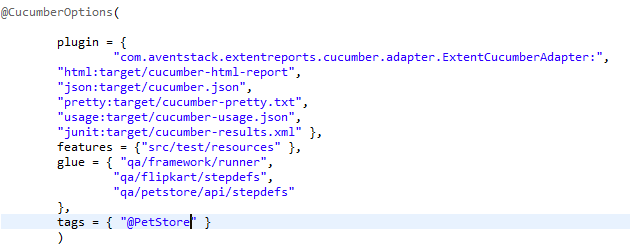
1. Create the step definition (application specific) and call the necessary method from different screen classes.

## Enable Mainframe feature

1. Open config.properties file and make ‘mainframe’ as ‘ture’
2. Specify ‘.ws’ file name against ‘mainframe\_ws’ (normally available in ‘src/test/resources > mf > launcher > <name>.ws’).
3. Specify ‘.kmp’ file name against ‘mainframe\_kmp’ (normally available in ‘src/test/resources > mf > launcher > <name >.kmp’).

## Run script from “RunCucumberTest” class

1. Open RunCucumberTest class available in ‘src/test/java’.
2. Under @CucumberOptions ‘glue’ section mention step definition package (with dot {.} replaced with forward-slash {/}).
3. Under @CucumberOptions ‘tag’ mention scenario tags which user want to execute.



1. To run Right click on class > Run As > TestNG

# Framework Configuration

There are several configuration variables available in framework for customizing as per requirements. Refer to *config.properties* file in the framework.

These properties are described below:

## Test Data - DB Configurations

Following variables provide configuration to load specific test data from data base.

1. db\_url: Data base URL – Host Name or IP (e.g. 10.0.xx.xx)
2. db\_port: Data base port (e.g. 8080)
3. db\_name: Data base name (e.g. lti\_qa\_db)
4. db\_username: user name with permission to access DB.
5. db\_password: user password.
6. client: client name. This is only used when scripts are created for multiple clients/ configurations. Usually applicable in case of product regression.
7. environment: environment name. This is essential to load test data based on environment specified. (e.g. QA, DEV, INIT etc.)
8. language: Used when handling web pages that support multiple languages (localization). This property provides flexibility to load web elements based on the specified language. English is the default language selected. For other languages, user need to specify here. E.g. Spanish, French etc.

## Browser Configurations

1. browser: user need to specify browser name (e.g. chrome, ie, firefox, edge) on which to execute the tests
2. browserVersion: (for remote execution only) user should specify browser version matching with node registered in Selenium hub (e.g. ANY).
3. platform: (for remote execution only) user should specify operation system name here matching with node registered in Selenium hub (e.g. ANY, WIN8).
4. remote: This property accepts Boolean value. If true, framework will look for Selenium hub to connect to.
5. huburl: Selenium hub url should be provided here (e.g. http://198.xx.xx.10:xxxx/wd/hub). This should only be provided if remote is *true*.
6. launchNewBrowserPerScenario: This property accepts Boolean value. If *true*, web browser will launch per scenario. Opening and closing of browser will be taken care by framework. However, if user do not want to open/close browser per scenario, he/she can chose to keep this setting as *false*. In such case user explicitly need to manage opening and closing the browser per scenario. *(recommended true)*
7. scrCapPerAction: If true, screenshot will be taken per action. This property accepts Boolean value. It is recommended to keep this setting false otherwise report generated will be huge and will take time to load.

## Mainframe Configuration

1. mainframe: Mainframe functionality can be enabled from here like launch IBM PCOMM, LeanFT server etc. This property accepts Boolean value.
2. mainframe\_ws: Mention ‘.ws’ file name here normally available in ‘src/test/resources > mf > launcher > <name>.ws’.
3. mainframe\_kmp: Mention ‘.kmp’ file name here normally available in ‘src/test/resources > mf > launcher > <name >.kmp’.

## Jira Auto Update Configuration

1. enableJiraUpdate: if user want to update the test execution status to Test Ids in JIRA, user can choose to make this setting as *true*. This property accepts Boolean value.
2. autoUpdateJiraRecords: if true, test execution status will be auto updated to Jira Test ids once execution is complete. If false, then user can see execution result in semi-colon separated file in “target/cucumber-reports/Spark/ExecutionReportxxxx.txt”. User can review the result and update the same in Jira from JiraManualUpdate.java class.
3. uploadReportZip: if true, zip of Spark report folder will be uploaded and attached to all the Test Execution ids specified. This property accepts Boolean value.
4. authentication: user need to provide authentication to access Jira (e.g. Basic xxxxxxxx)
5. executionIds: this property take comma separated execution ids of Jira. Accepts multiple JIRA test execution IDs.

# DB Details

Framework use MySQL DB to store web element locators and test data. Web elements are stored page wise and test data is stored feature wise.

**Web Element**: The hierarchy followed to store web elements is Workstreams (if any) > Applications > Pages

And table Value\_Type to store locater type.

Below are the SQL queries to create different tables:

Workstream (if any):

CREATE TABLE ‘Workstreams’(

‘Workstream\_Name’ VARCHAR(50) NOT NULL,

‘Workstream\_ID’ VARCHAR(50) NOT NULL,

‘Last Updated’ TIMESTAMP NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

PRIMARY KEY (‘Workstream\_ID’)

);

|  |  |  |
| --- | --- | --- |
| Workstream\_Name | Workstream\_ID | Last Updated |
| E-commerce | ECOM | <auto updated> |

Application:

CREATE TABLE ‘Applications’(

‘Workstream\_ID’ VARCHAR(50) NOT NULL,

‘Application’ VARCHAR(50) NOT NULL,

‘App\_ID’ VARCHAR(50) NOT NULL,

‘Last Updated’ TIMESTAMP NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

PRIMARY KEY (‘App\_ID’),

UNIQUE INDEX ‘Application’ (‘Application’,’Workstream\_ID’),

INDEX ‘Application\_Workstream\_FK’ (‘Worskstream\_ID’), CONSTRAINT ‘Application\_Workstream\_FK’

FOREIGH KEY (‘Workstream\_ID’) REFERENCES ‘Workstreams’ (‘Workstream\_ID’)

);

|  |  |  |  |
| --- | --- | --- | --- |
| Workstream\_ID | Application | App\_ID | Last Updated |
| ECOM | Flipkart | ECOM\_Flipkart | <auto updated> |

Pages:

CREATE TABLE ‘Pages’(

‘App\_ID VARCHAR(50) NOT NULL,

‘Page\_Name’ VARCHAR(50) NOT NULL,

‘Page\_ID’ VARCHAR(50) NOT NULL,

‘Last Updated’ TIMESTAMP NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

PRIMARY KEY (‘Page\_ID’),

INDEX ‘Pages\_Applications\_FK’ (‘App\_ID’),

CONSTRAINT ‘Pages\_Applications\_FK’ FOREIGN KEY (‘App\_ID’) REFERENCES ‘Applications’ (‘App\_ID’) ON UPDATE NO ACTION ON DELETE NO ACTION

);

|  |  |  |  |
| --- | --- | --- | --- |
| App\_ID | Page\_Name | Page\_ID | Last Updated |
| ECOM\_Flipkart | LoginPage | ECOM\_Flipkart\_LoginPage | <auto updated> |

Elements:

CREATE TABLE ‘Elements’(

‘Page\_ID’ VARCHAR(250) NOT NULL,

‘Element\_Key’ VARCHAR(250) NOT NULL,

‘Element\_Value’ VARCHAR(500) NULL DEFAULT NULL,

‘French’ VARCHAR(250) NULL DEFAULT NULL,

‘Spanish’ VARCHAR(250) NULL DEFAULT NULL,

‘Value\_Type\_ID’ INT(250) NULL DEFAULT NULL,

‘Last Updated’ TIMESTAMP NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

PRIMARY KEY (‘Element\_Key’,’Page\_ID’),

INDEX ‘Element\_Value\_Type\_FK’ (‘Value\_Type\_ID’),

INDEX ‘Elements\_Pages\_FK’ (‘Page\_ID’),

CONSTRAINT ‘Element\_Pages\_FK’ FOREIGN KEY (‘Page\_ID’) REFERENCES ‘Pages’ (‘Page\_ID’),

CONSTRAINT ‘Element\_Value\_Type\_FK’ FOREIGN KEY (‘Value\_Type\_ID’) REFERENCES ‘Value\_Type’ (‘Value\_Type\_ID’)

);

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Page\_ID | Element\_Key | Element\_Value | French | Spanish | Value\_Type\_ID | Last Updated |
| ECOM\_Flipkart\_LoginPage | txtUserName | Login | Null | Null | 1 | <auto updated> |

Value\_Type:

CREATE TABLE ‘Value\_Type’ (

‘Value\_Type\_ID’ INT (11) NOT NULL,

‘Value\_Type’ VARCHAR (250) NULL DEFAULT NULL,

‘Last Updated’ TIMESTAMP NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

PRIMARY KEY (‘Value\_Type\_ID’)

);

|  |  |  |
| --- | --- | --- |
| Value\_Type\_ID | Value\_Type | Last Updated |
| 1 | Id | <auto updated> |
| 2 | Name | <auto updated> |
| 3 | Classname | <auto updated> |
| 4 | Linktext | <auto updated> |
| 5 | Partiallinktext | <auto updated> |
| 6 | Xpath | <auto updated> |
| 7 | Tagname | <auto updated> |
| 8 | Cssselector | <auto updated> |
| 9 | ng-model | <auto updated> |
| 10 | ng-bind | <auto updated> |
| 11 | Exactbinding | <auto updated> |
| 12 | Buttontext | <auto updated> |
| 13 | Partialbuttiontext | <auto updated> |
| 14 | Repeater | <auto updated> |
| 15 | Value | <auto updated> |