

# MINI PROJECT



## MANUAL, GUI AND API TESTING

Team One SDET SEPT, 2021

## CONTENTS

Chapter	Contents	Page number
Chapter 1	Introduction	3
Chapter 2	Flow of the project	6
Chapter 3	Testing in Selenium 3.1 Cross Browser Testing 3.2 TestNG Framework 3.3 POM & DDT 3.4 Extent Reports	7
Chapter 4	API Testing 4.1 Introduction 4.2 Setup & Working with API 4.3 POSTMAN 4.4 Authentication & Authorization	19
Chapter 5	Karate Framework 5.1 Introduction 5.1.1 Installation 5.2 Structure of Karate 5.3 DDT and Correlation in Karate 5.4 Different Runners 5.5 Reports 5.6 Cucumber Reporting	27
Chapter 6	Conclusion	31

## Chapter 01- Introduction

The Mini-Project for Team One (GEMS SDET Sept 2021) contains GUI Testing and API Testing. This project report consists of detailed steps performed while testing the AUT (Application under Test), that is the Shopizer (Locally Hosted).

Shopizer is an open source E-commerce Bags Website. It is a Java based project, using MySQL database and HTML components. Support is provided for different category of bags. It is freely available.

The testing is conducted following the problem statement wherein a variety of tools are implemented including TestNG, Extent, Maven. The project includes the implementation of Page Object Model (POM) and Data-Driven Testing(DDT), Karate (For Api). For the Mini-Project various modules from the locally hosted E-commerce Website are considered for extracting test cases and scenarios for the testing. These modules constitute the most integral parts of the functioning and were selected based on that. API Implementation is carried out using Postman Tool as well as Karate Framework. The document contains all necessary screenshots and references. The entire process was carried out using the concepts of Scrum from Agile methodology and maintained a central tracker as well.

Github link of shopizer : <https://github.com/RameshMF/shopizer>

Documentation link of Shopizer: <https://shopizer-ecommerce.github.io/documentation/#/starting>

Swagger documentation link of Shopizer: <http://demo.shopizer.com:8080/swagger-ui.html>



## Mini Project Problem Statement

Application under Test	Shopizer Application
Start Date	26 <sup>th</sup> November 2021
End Date	17 <sup>th</sup> December 2021
Final Expected Outcome	<ul style="list-style-type: none"><li>• Selenium scripts</li><li>• Logs, appropriate screenshots as applicable</li><li>• Follow scrum methodology.</li><li>• Use appropriate templates for Test Case writing, presentation, etc.</li><li>• Proper naming convention and all best Practices followed.</li><li>• Present the learnings, challenges faced, process followed, demo of script execution along with report generation.</li><li>• Record a Demo in video format along with the explanation.</li><li>• Logs using log4j</li><li>• Rest API Testing</li></ul>
Deliverable's location	<ul style="list-style-type: none"><li>• GitHub</li></ul>
POC	Mentors: Group1 – Varsha Bakshi Group2 – Sunil Bansal Technical help: Respective trainer
Team Members	Isha Agrawal, Chinmay Sawant, Avinee Nemade, Sakshi Yedatkar, Aditya Jadhav, Aditya Dhaygude, Divyansh Sen, Ayan Shaikh



**Problem Statement:**

- Install Shopizer application and explore the AUT (Application under Test).

Ref Link: <http://demo.shopizer.com:8080>

**GUI Automation** (using TestNG) –

- Identify 50 Test Cases from different modules.
- Identify the test cases to be automate.
- Using Maven as dependency and build management tool, we will create maven project.
- Use Selenium and java for automation.
- Use POM structure and data driven testing using csv/excel files.
- Use TestNG for defining tests.
- Log4j will be used for logging.
- Explore the RESTful API feature of the Application Under Test.
- Analyze on which resources should be tested
- REST API Testing with Karate.
- Make collection of requests along with proper validation.
- Based on request collection, reports will be generated.

**Unique Features: -**

- Testing mouse hover feature.
  - Verifying the discounted price.
  - Checking for broken links.
  - Verifying whether the page is translated from English to French and vice versa.
-



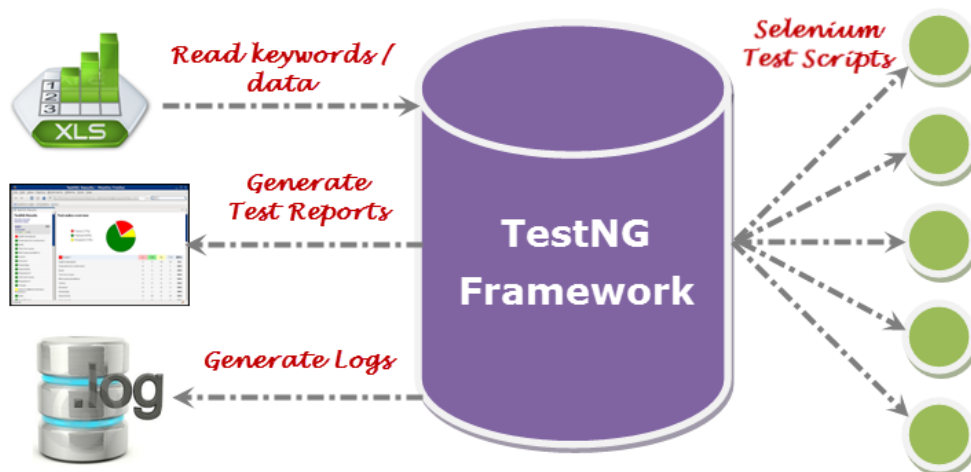
## Chapter- 03 Testing in Selenium

**3.1 Cross Browser testing** is a type of non-functional testing that lets you check whether your website works as intended when accessed through:

- **Different Browser-OS combinations** i.e., on popular browsers like Firefox, Chrome, Edge, Safari—on any of the popular operating systems like Windows, macOS, iOS and Android.
- **Different devices** i.e., users can view and interact with your website on popular devices—smartphones, tablets, desktops, and laptops etc.
- **Assistive Tools** i.e., the website is compatible with assistive technologies like screen readers for individuals who are differently abled.

### 3.2 TestNG Framework

It is an open-source automated testing framework, where **NG** of Test**NG** means **N**ext **G**eneration. TestNG is like JUnit, but it is much more powerful than JUnit but still, it's inspired by JUnit. It is designed to be better than JUnit, especially when testing integrated classes.



## Features Of TestNG

Below is the list of features of TestNG:

- Supports annotations.
- TestNG uses more Java and OO features.
- Supports testing integrated classes (e.g., by default, no need to create a new test class instance for every test method).
- Separates compile-time test code from run-time configuration/data info.
- Flexible runtime configuration.
- Introduces 'test groups. Once you have compiled your tests, you can just ask TestNG to run all the "front-end" tests, or "fast", "slow", "database" tests, etc.
- Supports Dependent test methods, parallel testing, load testing, and partial failure.
- Flexible plug-in API.
- Support for multi-threaded testing.

## Installation Through Marketplace

Given below are the steps to install TestNG from Marketplace:

**Step 1:** Start Eclipse.

**Step 2:** Go to the Help Section.

**Step 3:** Click **Eclipse Marketplace** in the Help Section.

**Step 4:** The Eclipse Marketplace window opens. Enter TestNG in the **Find** option and click on the search button.

**Step 5:** Click on the **Install button** as shown below.

**Step 6:** Another new window will open, do not change anything. Just click on the **Confirm button**.



**Step 7:** Click on the **Next button** and the License Agreement dialog box will get opened. Click on “I accept the terms of the license agreement” and then click on the **Finish button**.

**Step 8:** When a Security warning is received, click on the OK button.

**Step 9:** Please wait for the installation to be completed.

**Step 10:** Eclipse will prompt for a restart, click the **Yes button**. If not, we should restart Eclipse for the changes that we have done.

**Step 11:** Once the restart is completed, we can verify if TestNG was successfully installed or not. To verify, click on Windows, then on Preferences and see if TestNG is included in the Preferences list or not.

## **Annotations:**

- **@BeforeMethod:** This will be executed before every @test annotated method.
- **@AfterMethod:** This will be executed after every @test annotated method.
- **@BeforeClass:** This will be executed before first @Test method execution. It will be executed one only time throughout the test case.
- **@AfterClass:** This will be executed after all test methods in the current class have been run
- **@BeforeTest:** This will be executed before the first @Test annotated method. It can be executed multiple times before the test case.
- **@AfterTest:** A method with this annotation will be executed when all @Test annotated methods complete the execution of those classes inside the <test> tag in the TestNG.xml file.
- **@BeforeSuite:** It will run only once, before all tests in the suite are executed.
- **@AfterSuite:** A method with this annotation will run once after the execution of all tests in the suite is complete.

- **@BeforeGroups:** This method will run before the first test run of that specific group.
- **@AfterGroups:** This method will run after all test methods of that group complete their execution.

### 3.3 POM and DDT

**Page Object Model (POM)** is a design pattern, popularly used in test automation that creates Object Repository for web UI elements. The advantage of the model is that it reduces code duplication and improves test maintenance.

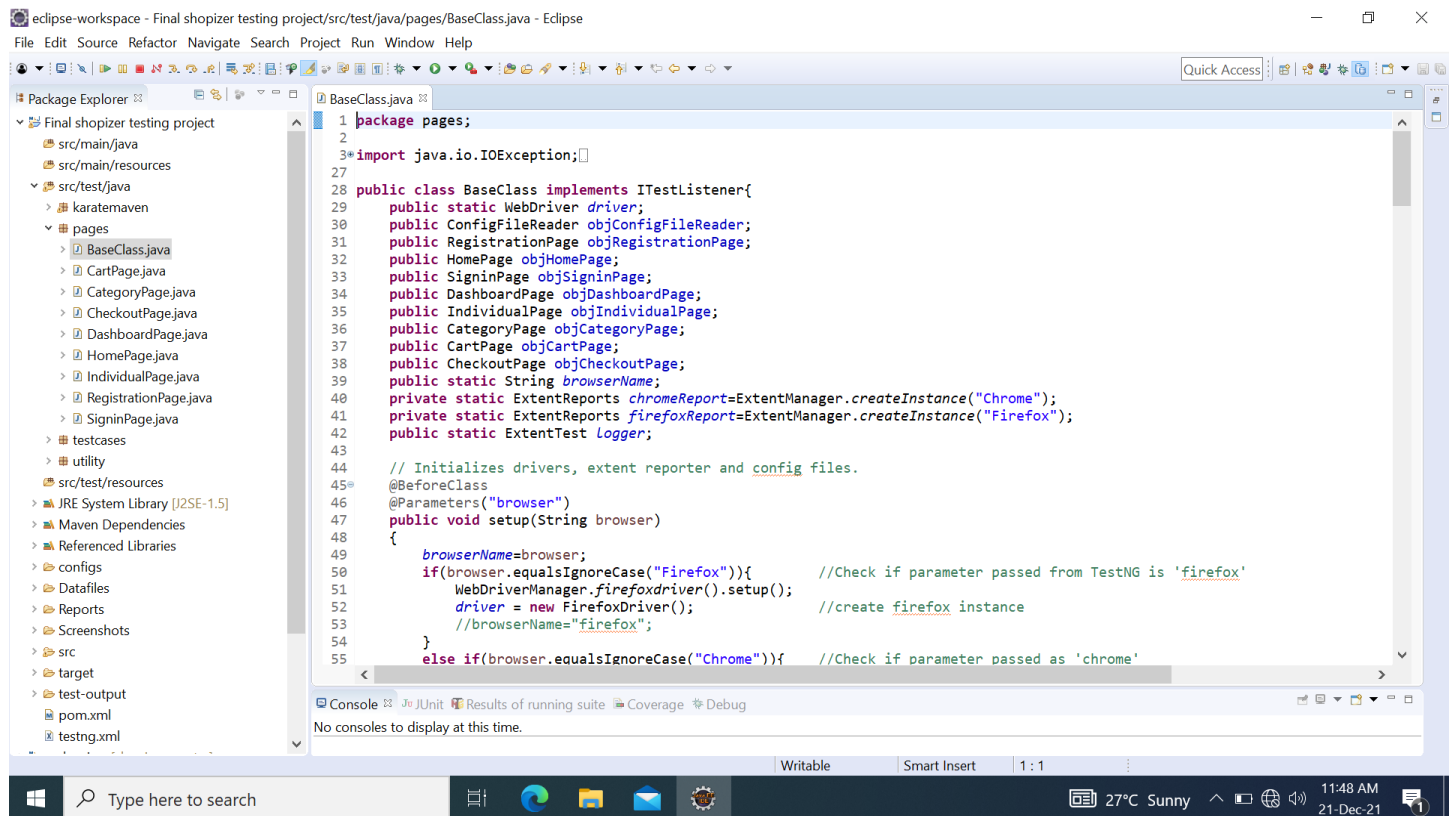
Under this model, for each web page in the application, there should be a corresponding Page Class. This Page class will identify the Web Elements of that web page and contains Page methods which perform operations on those Web Elements. Name of these methods should be given as per the task they are performing.

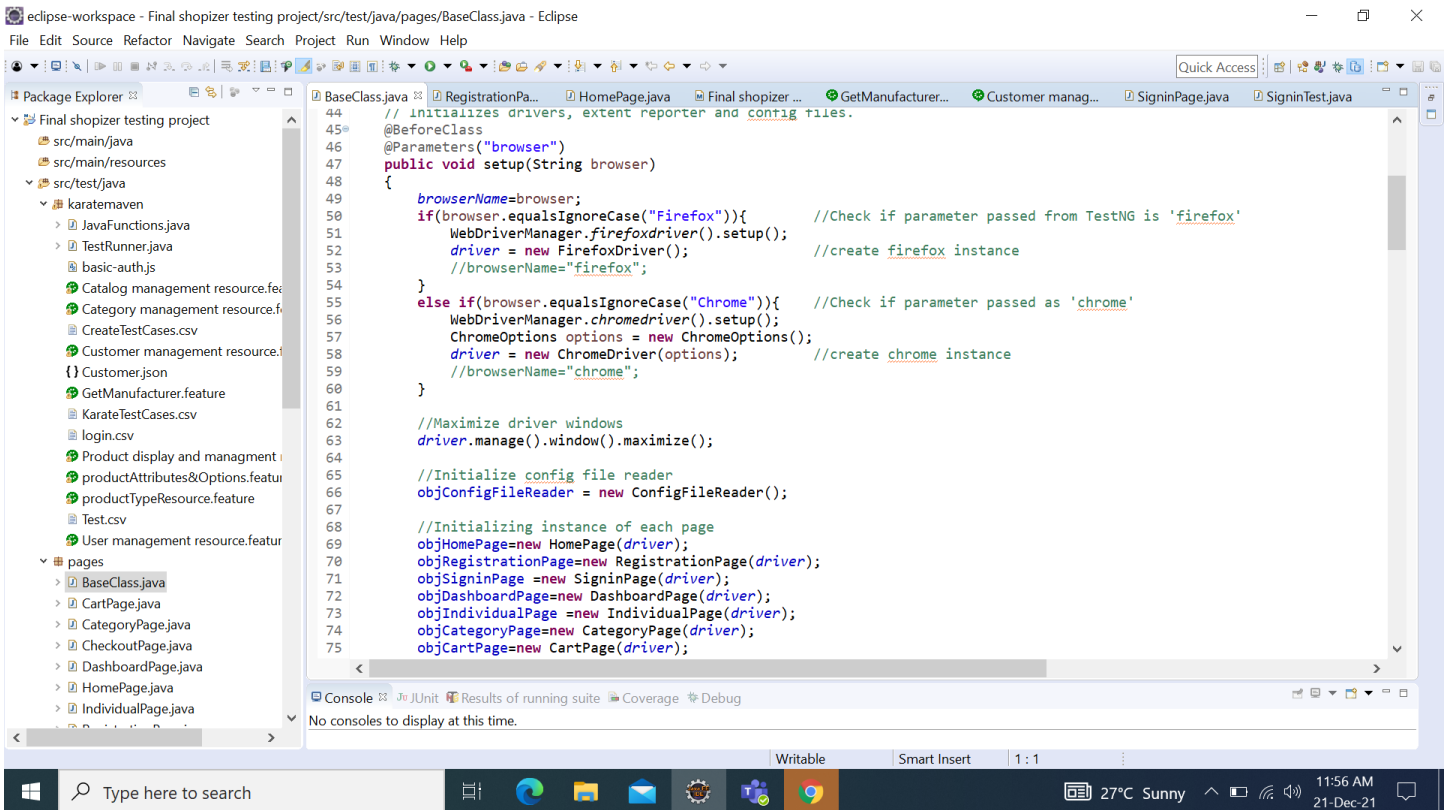
To implement POM, we have created six pages like base class, home page, Helper class, Registration page, Registration test page, Config file reader, TestNG test and config properties file for the execution of test cases in selenium.

Below are the snapshots of the scripts.

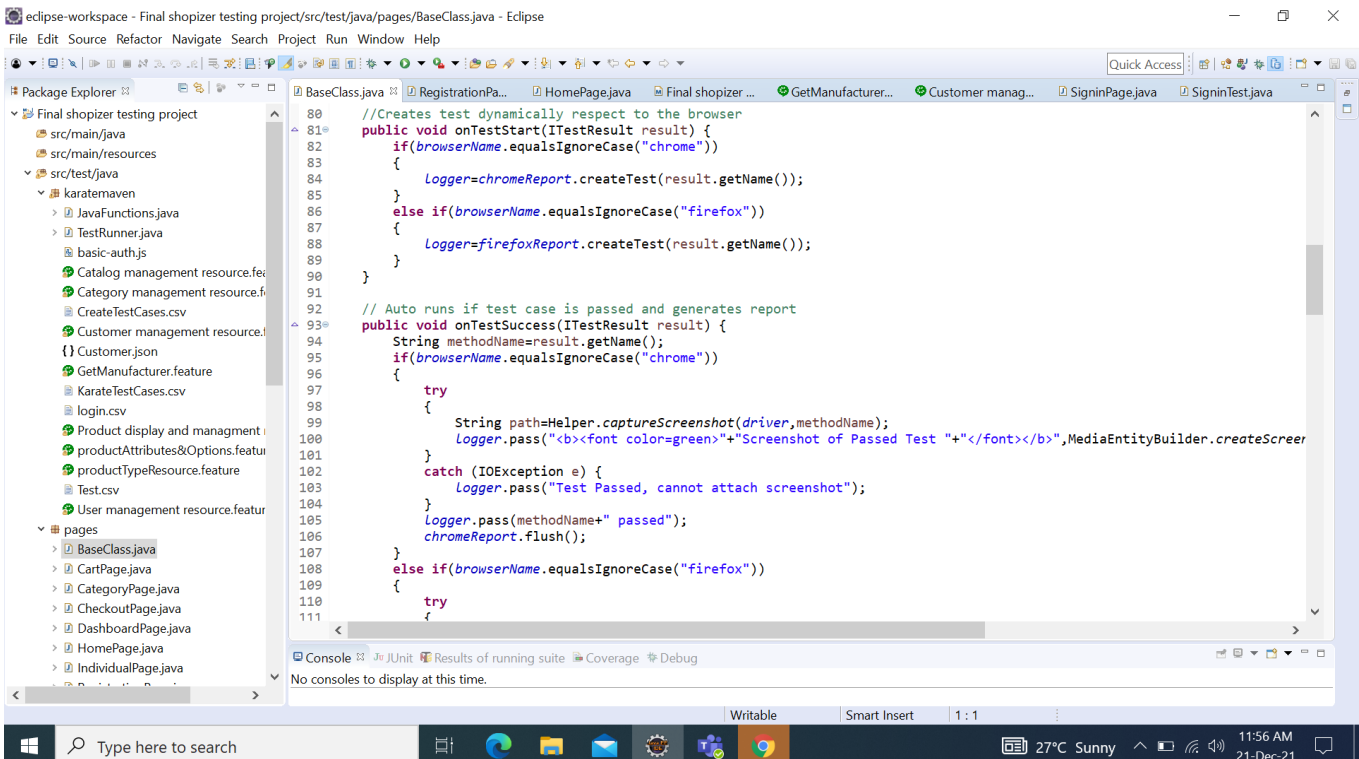
## Base

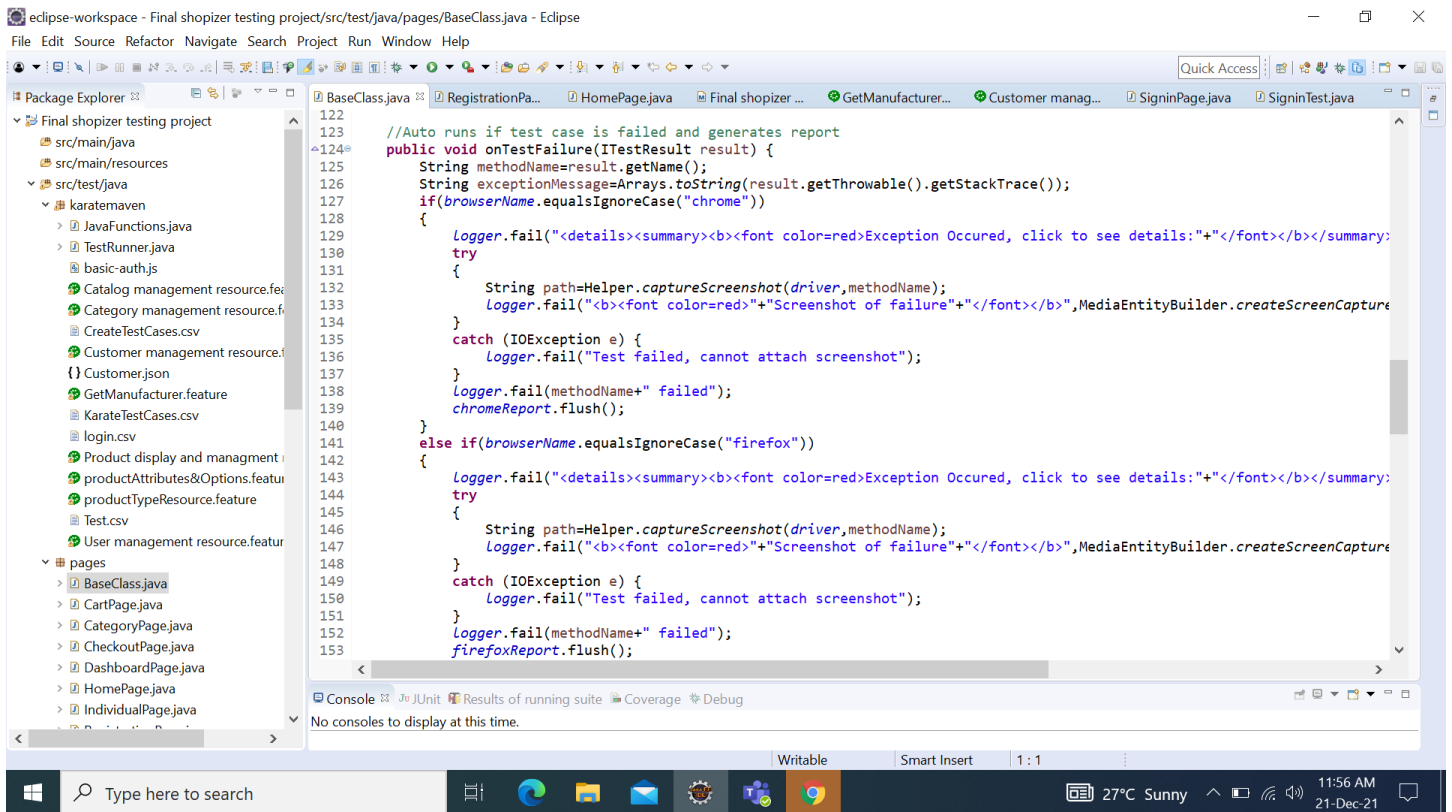
Class:





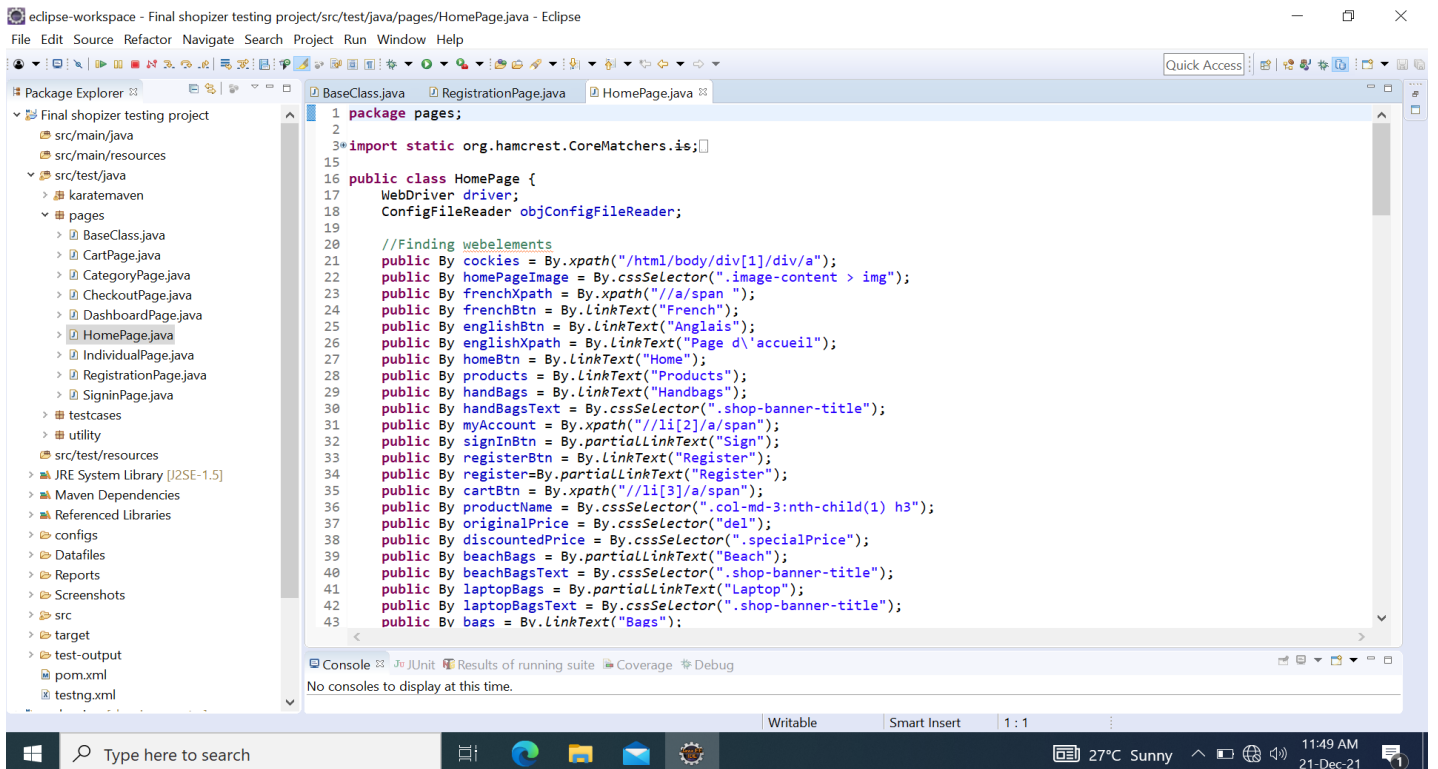
## ITestListener implementation in Base class:





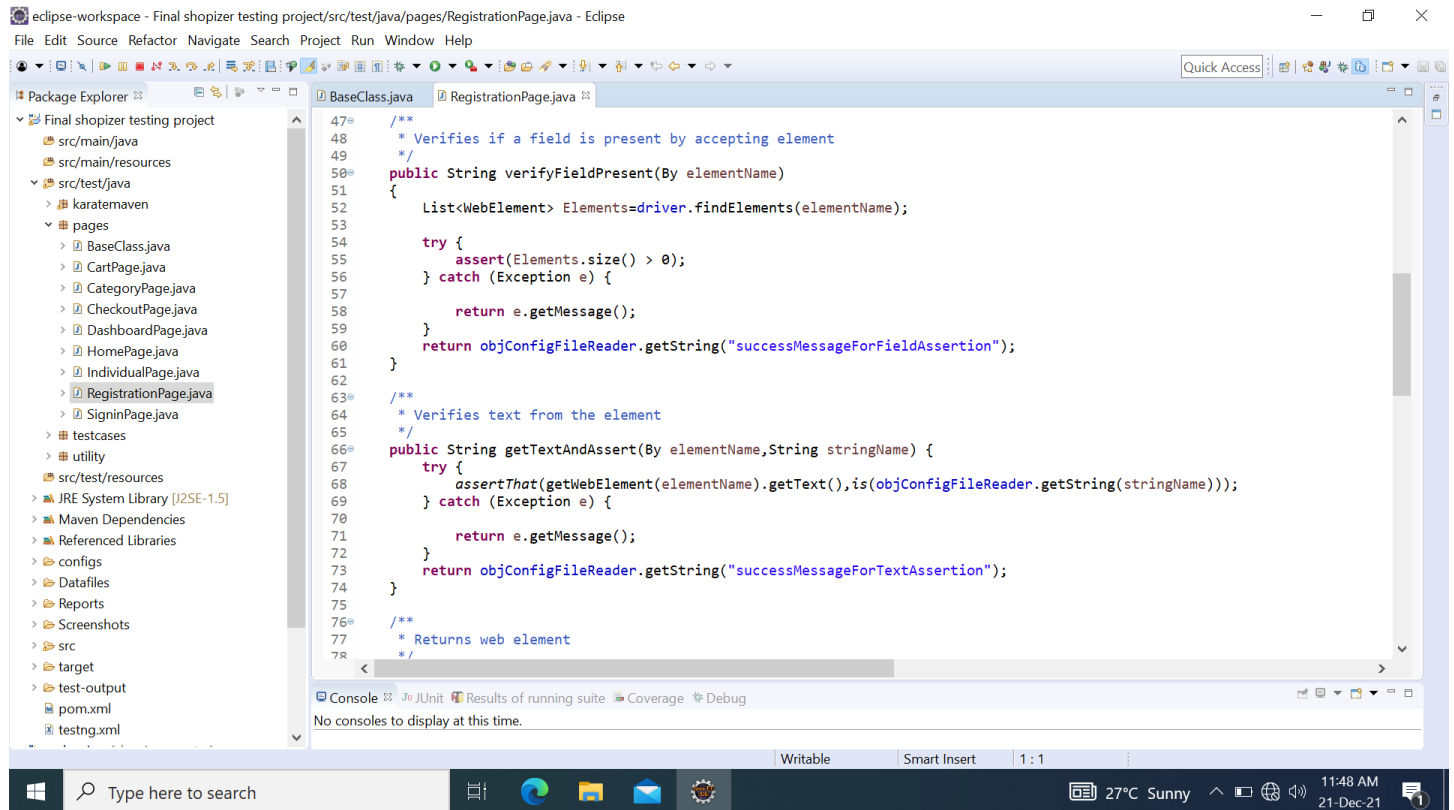
Home

page:



December 20, 2021

# Registration Page:



## POM.XML:

```
<dependency>
  <groupId>io.github.bonigarcia</groupId>
  <artifactId>webdrivermanager</artifactId>
  <version>5.0.3</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.testng/testng -->
<dependency>
  <groupId>org.testng</groupId>
  <artifactId>testng</artifactId>
  <version>6.11</version>
  <scope>test</scope>
</dependency>

<!-- https://mvnrepository.com/artifact/com.aventstack/extentreports -->
<dependency>
  <groupId>com.aventstack</groupId>
  <artifactId>extentreports</artifactId>
  <version>4.0.0</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.poi/poi -->
<dependency>
  <groupId>org.apache.poi</groupId>
  <artifactId>poi</artifactId>
  <version>3.17</version>
</dependency>

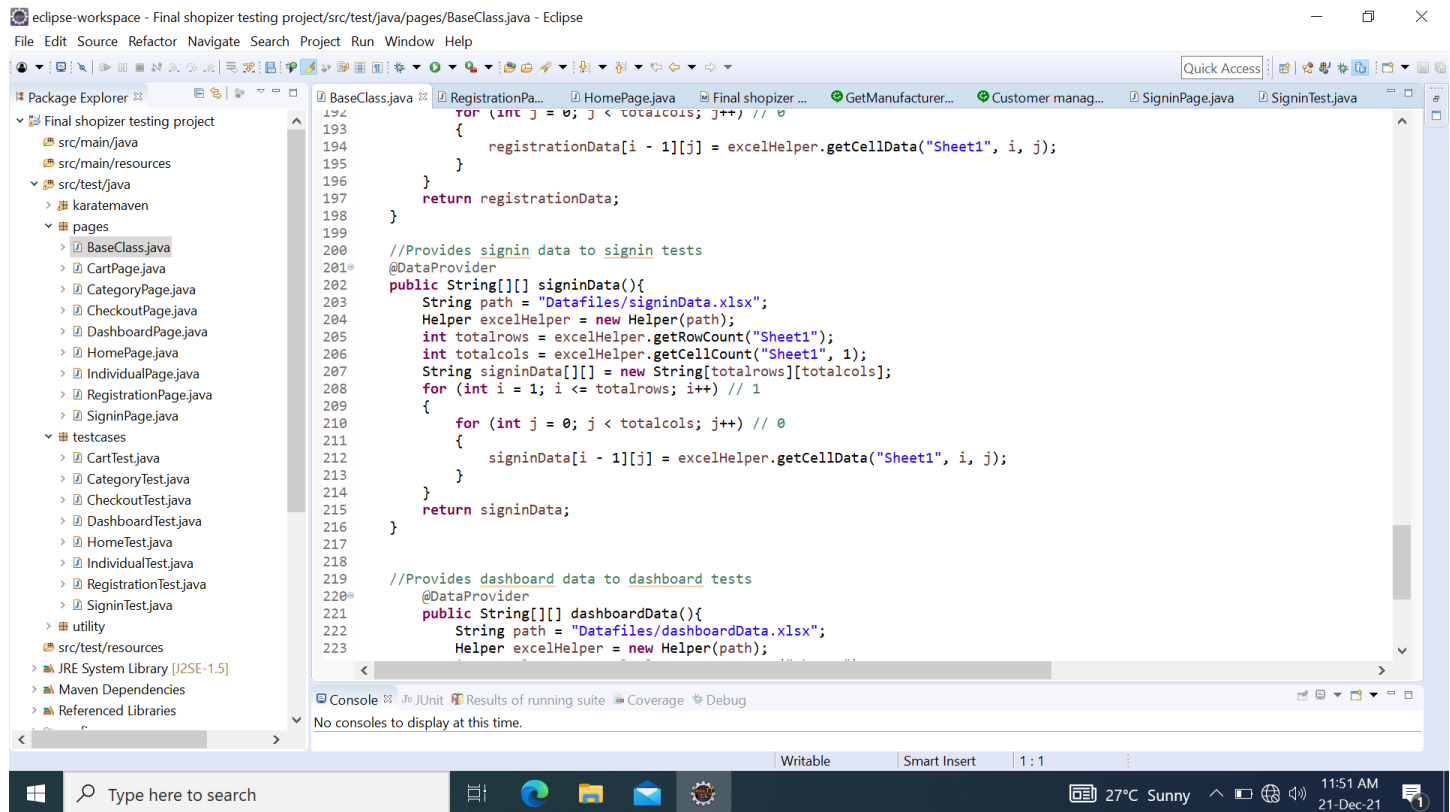
<!-- karate dependency -->
<dependency>
  <groupId>com.intuit.karate</groupId>
  <artifactId>karate-junit4</artifactId>
  <version>1.2.0.RC1</version>
  <scope>test</scope>
</dependency>
```

## DDT

Data-driven testing is creation of test scripts where test data and/or output values are read from data files instead of using the same hard-coded values each time the test runs. This way, testers can test how the application handles various inputs effectively.

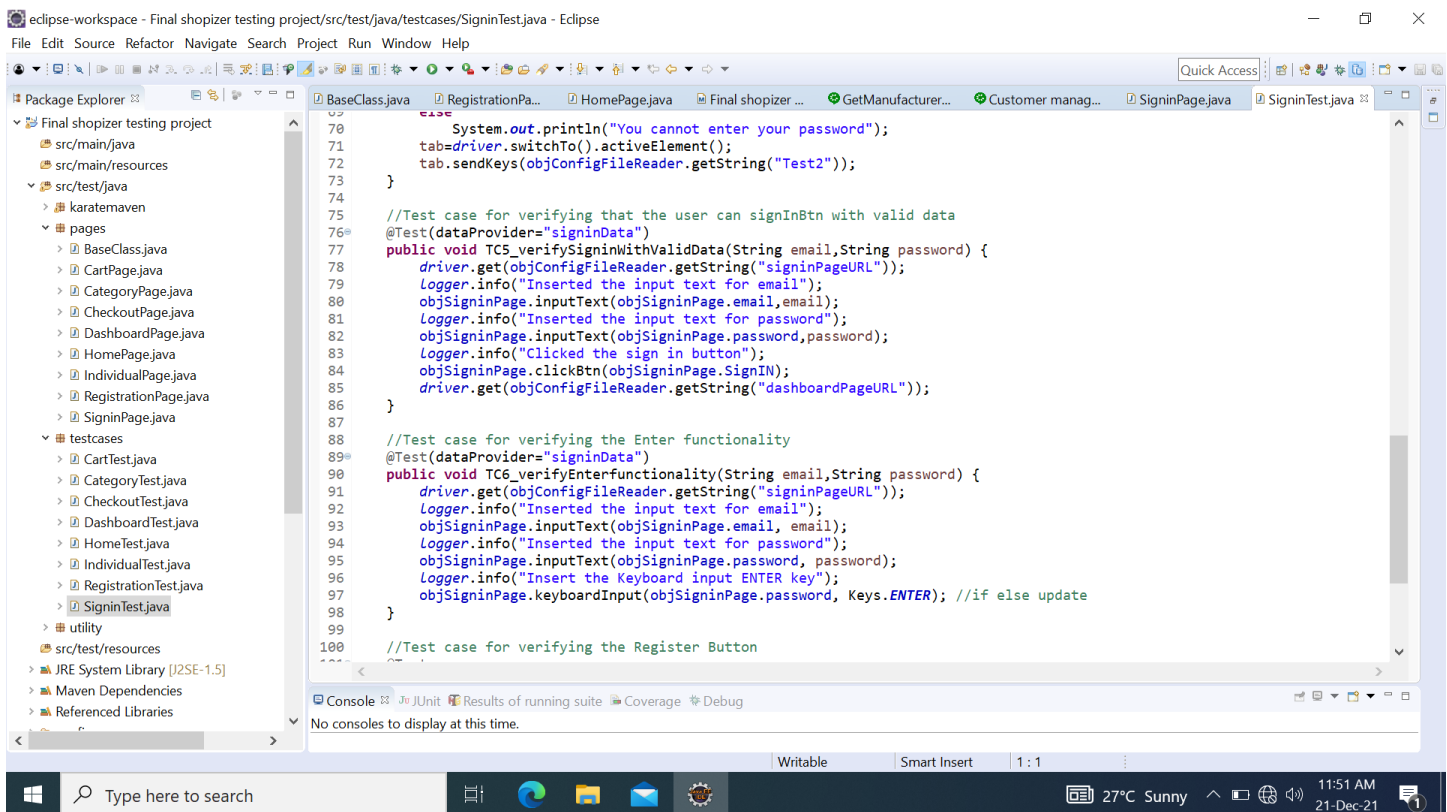
We have used `FileReader` and `CSVReader` classes to read data from a CSV file that is needed for the script to run.

## DDT Screenshots:



`CSV_PATH` is a `String` object that is used to store the path of the CSV file that the script needs to read the comma separated values from.





Reading data from a csv file using FileReader and CSVReader classes.

## 3.4 Extent Reports

This method will generate two separate reports for tests.

### 1) Dependency used for ExtentReport:

```
<!-- https://mvnrepository.com/artifact/com.aventstack/extentreports -->
<dependency>
    <groupId>com.aventstack</groupId>
    <artifactId>extentreports</artifactId>
    <version>4.0.0</version>
</dependency>
```

December 20, 2021

**2)Extent Manager – We made a java class for creating an instance of the report.**

```

BaseClass.java  ExtentManager.java
1 package utility;
2
3 import com.aventstack.extentreports.ExtentReports;
4
5
6
7 public class ExtentManager {
8     private static ExtentReports report;
9
10    public static ExtentReports createInstance(String browserName)
11    {
12        ExtentHtmlReporter htmlReporter=new ExtentHtmlReporter("./Reports/Shopizer_"+bro
13        htmlReporter.config().setEncoding("utf-8");
14        htmlReporter.config().setDocumentTitle("Shopizer Test Result");
15        htmlReporter.config().setReportName("Automation Tests");
16        htmlReporter.config().setTheme(Theme.STANDARD);
17
18        report=new ExtentReports();
19        report.setSystemInfo("Team", "1");
20        report.setSystemInfo("AUT", "Shopizer");
21        report.attachReporter(htmlReporter);
22        return report;
23    }
24 }
25

```

### 3) Base class for ExtentReport:

#### A) Implemented ITestListener in base class:

```

public class BaseClass implements ITestListener{
    //Creating two separate Report instances for Chrome and Firefox
    private static ExtentReports chromeReport=ExtentManager.createInstance("Chrome");
    private static ExtentReports firefoxReport=ExtentManager.createInstance("Firefox");
    //Creating Common test object for both the Reports
    public static ExtentTest logger;
}

```

#### B) OnTestStart Method:

```

/**
 * Creates test dynamically respect to the browser
 */
public void onTestStart(ITestResult result) {
    if(browserName.equalsIgnoreCase("chrome"))
    {
        logger=chromeReport.createTest(result.getName());
    }
    else if(browserName.equalsIgnoreCase("firefox"))
    {
        logger=firefoxReport.createTest(result.getName());
    }
}

```

December 20, 2021

#### C) OnTestFailure Method:

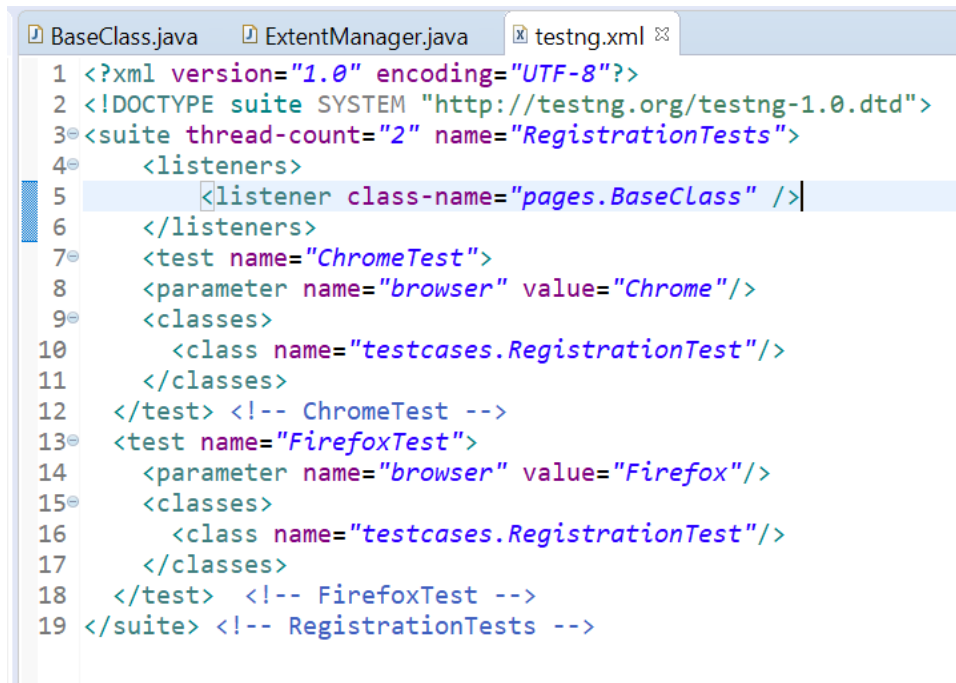
```

public void onTestFailure(ITestResult result) {
    String methodName=result.getName();
    String exceptionMessage=Arrays.toString(result.getThrowable().getStackTrace());
    if(browserName.equalsIgnoreCase("chrome"))
    {
        Logger.fail("<details><summary><b><font color=red>Exception Occured, click to see details:</font></b></summary>"+e>
        try
        {
            String path=Helper.captureScreenshot(driver,methodName);
            Logger.fail("<b><font color=red>"+Screenshot of failure"+</font></b>"+MediaEntityBuilder.createScreenCaptureFrom
        }
        catch (IOException e) {
            Logger.fail("Test failed, cannot attach screenshot");
        }
        Logger.fail(methodName+" failed");
        chromeReport.flush();
    }
    else if(browserName.equalsIgnoreCase("firefox"))
    {
        Logger.fail("<details><summary><b><font color=red>Exception Occured, click to see details:</font></b></summary>"+e>
        try
        {
            String path=Helper.captureScreenshot(driver,methodName);
            Logger.fail("<b><font color=red>"+Screenshot of failure"+</font></b>"+MediaEntityBuilder.createScreenCaptureFrom
        }
        catch (IOException e) {
            Logger.fail("Test failed, cannot attach screenshot");
        }
        Logger.fail(methodName+" failed");
        firefoxReport.flush();
    }
}
}

```

Similarly We implemented OnTestSuccess and OnTestSkip method.

#### D) Declare Listener class in TestNG Class:



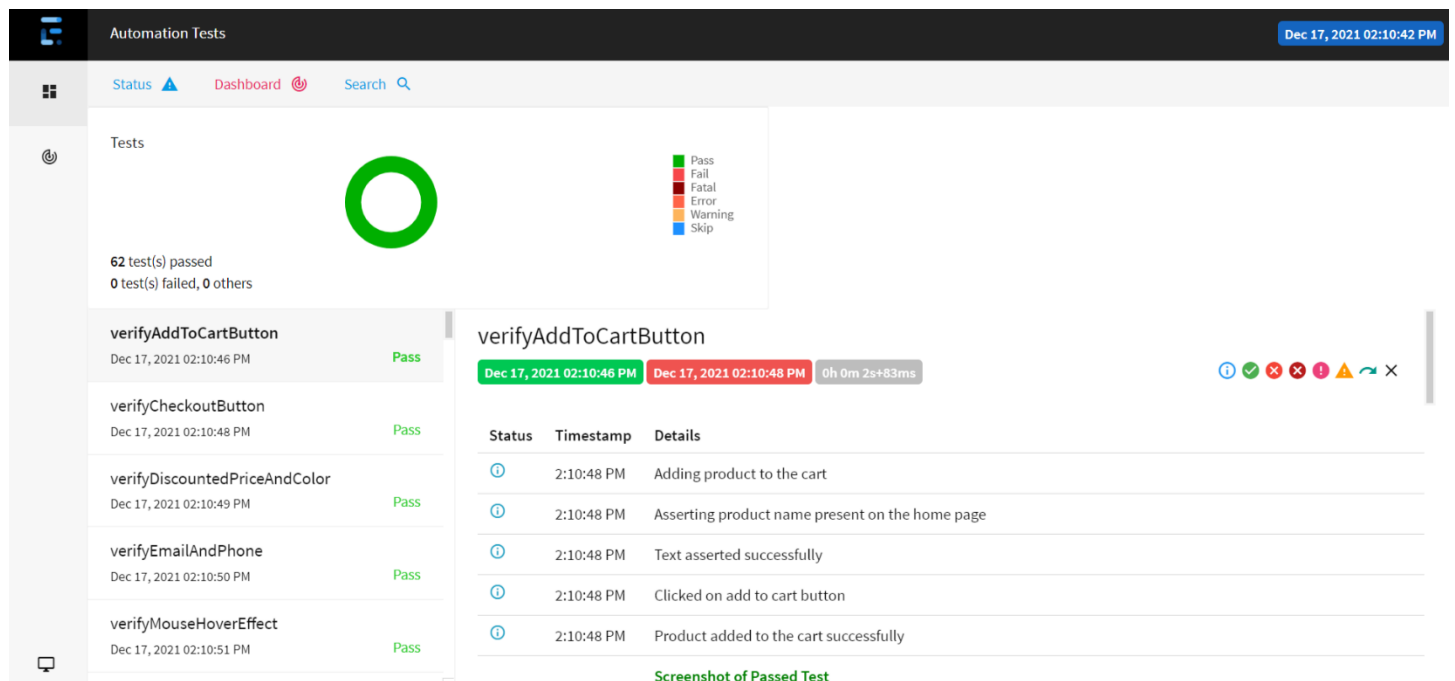
```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
3 <suite thread-count="2" name="RegistrationTests">
4   <listeners>
5     <listener class-name="pages.BaseClass" />
6   </listeners>
7   <test name="ChromeTest">
8     <parameter name="browser" value="Chrome"/>
9     <classes>
10      <class name="testcases.RegistrationTest"/>
11    </classes>
12  </test> <!-- ChromeTest -->
13  <test name="FirefoxTest">
14    <parameter name="browser" value="Firefox"/>
15    <classes>
16      <class name="testcases.RegistrationTest"/>
17    </classes>
18  </test> <!-- FirefoxTest -->
19 </suite> <!-- RegistrationTests -->

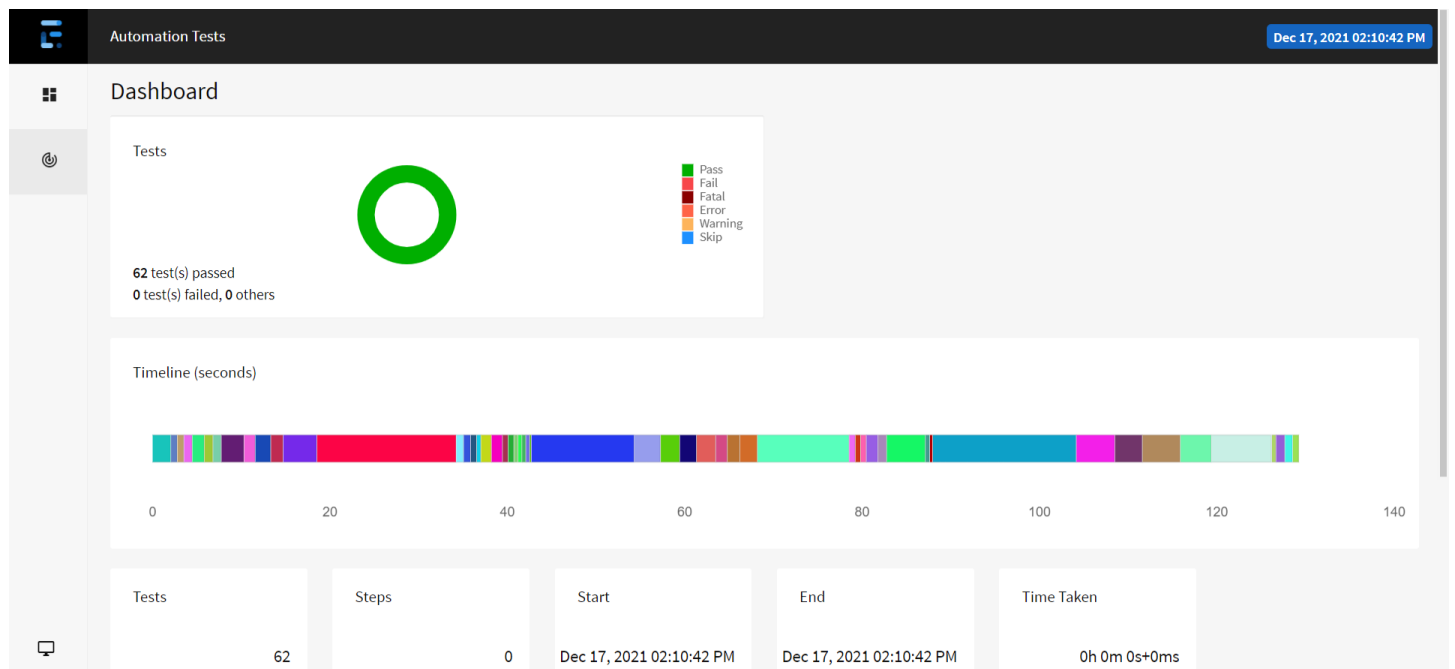
```

ExtentReports is an open-source reporting library used in selenium test automation. Extent reports become the first choice of Selenium Automation Testers, even though Selenium comes with inbuilt reports using frameworks like JUnit and TestNG. With extent reports, you can offer a more extensive and insightful

perspective on the execution of your automation scripts. We used ITestListener for extent reports.



## Sample Extent Report



## Chapter- 04 API Testing

### 4.1 Introduction

#### RESTful API

A RESTful API is an architectural style for an application program interface (API) that uses HTTP requests to access and use data. That data can be used to GET, PUT, POST and DELETE data types, which refers to the reading, updating, creating, and deleting of operations concerning resources.

An API for a website is code that allows two software programs to communicate with each other. The API spells out the proper way for a developer to write a program requesting services from an operating system or other application.

For the concerned AUT only a limited number of endpoints were available over the internet for free API access and only GET and POST methods would work without a paid subscription.

Below is the list of a few API resources used in API Testing phase of our project:

- Catalogue Management resource
- Category Management resource
- Customer Management resource
- Manufacturer/Brand management resource
- User management resource
- Product Type resource
- Product display and management resource
- Product attribute options resource

## 4.2 Setup and Working with API

### A) Software requirements:

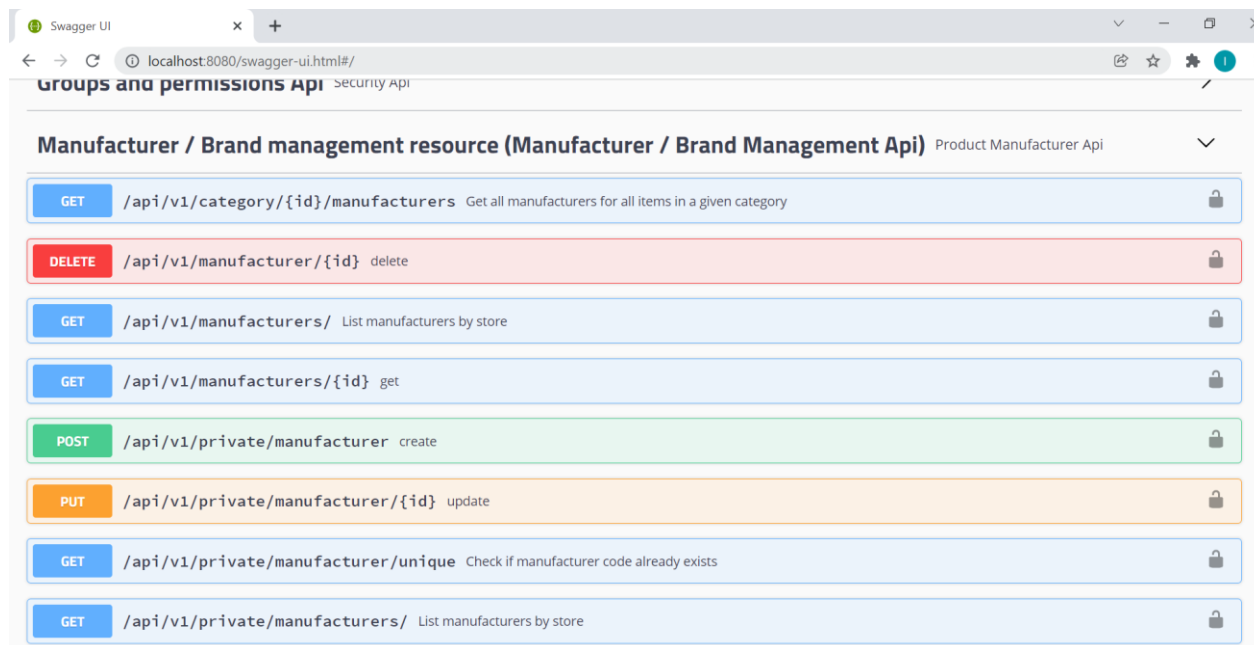
Postman - To work with API and handle requests.

Karate - To work with API and handle requests, correlation, and DDT

### B) Setting up API server:

We used Swagger UI for getting the resources for automation purpose.

### C) Swagger Resource Screenshot:



December 20, 2021

## 4.3 POSTMAN

### Overview

Postman is an API (application programming interface) development tool which helps to build, test, and modify APIs. Almost any functionality that could be needed by any developer is encapsulated in this tool.

It can make various types of HTTP requests (GET, POST, PUT, PATCH), saving environments for later use, converting the API to code for various languages (like JavaScript, Python).

While working with Postman, the first thing we did was to create a team workspace but that was limited only to three team members, then we created a collection to start with the requests.

<http://localhost:8080/api/v1/private/manufacturer>

the successful POST request of the above URL will work fine and will create a new item and give 201 Created response.

The screenshot shows a REST client interface with the following details:

- Overview:** Overview, POST Create a new ma..., +, No Environment
- Path:** shopizer\_Manufacturer / Brand management resour... / Create a new manufacture ...
- Buttons:** Save, Send
- Method:** POST
- URL:** http://localhost:8080/api/v1/private/manufacture...
- Body:**

```
{
  "code": "68690", "name": "bags", "visible": true
}
```
- Response:**

```
{
  "id": 450,
  "code": "68690",
  "order": 0,
  "descriptions": []
}
```
- Metadata:** 201 Created, 1316 ms, 727 B, Save Response

**1) POST Request:** - A POST request is used to send data to the server, for example, customer information, file upload, etc. using HTML forms. This method can make changes in database. Mainly used to add new entries to database.

A) Here we are adding the product for an unique code with a name of the product.

B) The URL used for this operation:

Localhost/api/v1/private/manufacture



The screenshot shows a REST client interface with the following details:

- Overview:** Overview, POST Create a new ma..., +, No Environment
- Path:** shopizer\_Manufacturer / Brand management resour... / Create a new manufacture ...
- Buttons:** Save, Send
- Request Method:** POST
- Request URL:** http://localhost:8080/api/v1/private/manufacturer
- Request Body:**

```
{
  "code": "68690", "name": "bags", "visible": true
}
```
- Response Body:**

```
{
  "id": 450,
  "code": "68690",
  "order": 0,
  "descriptions": []
}
```
- Response Status:** 201 Created, 1316 ms, 727 B
- Response Format:** JSON

**2) GET Request:** - The GET method is used to retrieve information from the given server using a given URI. Requests using GET should only retrieve data and should have no other effect on the data.

A) Here we are retrieving the details of the product that we have already added to the cart in the previous request.

B) The URL used for this operation:

Localhost/api/v1/manufacturers/450

B) The details of the product as a json object are returned.

shopizer\_Manufacturer / Brand management re... / Get single manufacturer ID by ...

Save

GET http://localhost:8080/api/v1/manufacturers/450 Send

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL

This request does not have a body

Body Cookies Headers (9) Test Results (1/1) 200 OK 64 ms 466 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 450,
3   "code": "68690",
4   "order": 0,
5   "description": null
6 }
```

**3) PUT Request:** - Replaces all the current representations of the target resource with the uploaded content. With this method, we can update existing data.

A) Here while updating, all the items are updated in the database and only 200 OK response is printed.

B) The URL used for this operation:

Localhost/api/v1/private/manufacturer/450

C) Here we are using the same id we used for POST and GET methods.

shopizer\_Manufacturer / Brand management resource / Update

Save

Send

PUT http://localhost:8080/api/v1/private/manufacturer/450

Params Authorization Headers (11) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2   "code": "okeey", "name": "bags", "visible": true
3 }
```

Body Cookies (1) Headers (15) Test Results 200 OK 431 ms 628 B Save Response

Pretty Raw Preview Visualize Text

```
1
```

**4) DELETE Request:** -Removes all the current representations of the target resource given by URI.

A) Here we are deleting the item, and nothing is printed in the console and it gets deleted from the database.

B) The URL used for this operation:

Localhost/api/v1/manufacturer/450

C) here we have used the same Id we used for GET, PUT, POST.

shopizer\_Manufacturer / Brand management reso... / Delete a single Manufacture...

Save

DELETE

http://localhost:8080/api/v1/manufacturer/450

Send

Params Authorization Headers (6) **Body** Pre-request Script Tests Settings Cookies

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

This request does not have a body

Body

Cookies

Headers (8)

Test Results (1/1)

200 OK 53 ms 371 B

Save Response

Pretty

Raw

Preview

Visualize

Text

1 |

## Chapter – 05 Karate Framework

### 5.1 – Introduction:

Karate is an open-source general-purpose test-automation framework that can script calls to HTTP end-points and assert that the JSON or XML responses are as expected. Karate also has support for service-virtualization where it can bring up "mock" (or stub) servers which can substitute for web-services that need to participate in an integration-test. Karate's capabilities include being able to run tests in parallel, HTML reports and compatibility with Continuous Integration tools.

#### 5.1.1 – Installation Steps:

Prerequisites for Environment Setup:

- 1) Download and install JDK and JRE from <http://www.oracle.com/technetwork/java/javase/downloads/index.html> as Cucumber supports Java for execution.
- 2) Download Eclipse from <https://eclipse.org/downloads/> as it contains the base workspace.
- 3) Download Maven from <https://maven.apache.org/download.cgi> and set its environment variable.
- 4) Configure Cucumber in Maven:
  - A) Create a Maven Project.
  - B) Open pom.xml.

C) Add dependency for Karate Framework:

```
<dependency>
  <groupId>com.intuit.karate</groupId>
  <artifactId>karate-junit4</artifactId>
  <version>1.2.0.RC1</version>
  <scope>test</scope>
</dependency>
```

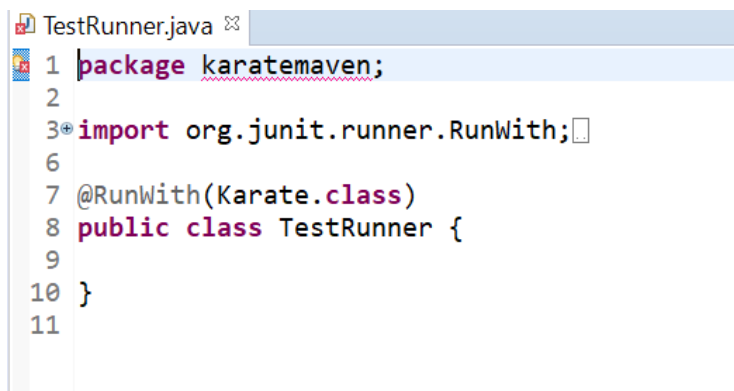
Save the POM.XML for future use.

## 5.2– Structure of Karate:

- 1) Test Runner.java -
- 2) Basic-Auth.js
- 3) CustomerManagement.feature file

Below are the Description and screenshots of the above scripts:

1) TestRunner.java: Test runner class also acts as an interlink between feature files. Right click on src/test/Java folder --> New --> Package --> Give Package name as testRunner. Right click on testRunner package --> New --> Class --> Give class name as any class name. Modify your testRunner class file as below.

A screenshot of a code editor showing the TestRunner.java file. The code is as follows:

```
1 package karatemaven;  
2  
3 import org.junit.runner.RunWith;  
6  
7 @RunWith(Karate.class)  
8 public class TestRunner {  
9  
10 }  
11
```

2) Basic-Auth.js: This feature file is used to set headers (admin-headers. Feature). This feature file gets the token after admin user login is performed via karate-config.js. Then assigns the token along with the Base64 encoded basic auth to the headers calling headers.js. The Base64 user and password are being input as maven arguments and read via karate-config variables.

```
basic-auth.js
1 function fn(creds) {
2   var temp = creds.username + ':' + creds.password;
3   var Base64 = Java.type('java.util.Base64');
4   var encoded = Base64.getEncoder().encodeToString(temp.toString().getBytes());
5   return 'Basic ' + encoded;
6 }
7
```

### 3)CustomerManagement.feature file:

A Feature File is starting point to the Karate tests. This is a file where we will describe our tests in Descriptive language. Every feature file should have dot feature extension. As an example, => "filename.feature".

#### **Feature File Folder Structure:**

Below screenshot shows Karate Feature File Structure:  
Feature Files== Where we keep all feature files.  
test.feature , testnew.feature == 2 sample feature files.

# All feature files in Karate should have the extension. feature in its name. If. feature is not present, then Karate won't recognize it as a feature file.

# We need to add a keyword called Feature. Without this, you will see compile errors in your script.

# Each scenario in a feature file should also start with a keyword called Scenario. A single feature file can have multiple scenarios.

# Scenarios are written starting with - Given, When, Then etc.

Right click on FeatureFiles folder --> New --> File --> Give file name as testing1.feature

```

1=Feature: Shopizer Manufacturer / Brand management resource
2
3=Background:
4  * url 'http://localhost:8080/' #globally declaring the url
5
6 @getcall1
7=Scenario: Get all manufacturers for all items in a given category with category id 1 #start of a feature file
8  Given path 'api/v1/category/1/manufacturers/' #declaring a path obtained from Swagger.UI
9  #accessing username and password fro, basic-auth.js file
10  * header Authorization = call read('basic-auth.js') { username: 'admin@shopizer.com', password: 'password'
11
12  When method GET #calling the GET method
13  Then status 200 #Response status
14  Then print response #Printing the response
15

```

### 5.3– DDT and Correlation in Karate:

Data-driven testing is the creation of test scripts where test data and/or output values are read from data files instead of using the same hard-coded values each time the test runs. This way, testers can test how the application handles various inputs effectively.

We have used FileReader and CSVReader classes to read data from a CSV file that is needed for the script to run. We have executed DDT and correlation in Karate script to get the desired output for the test cases.

The following steps are done for DDT and correlations:

- 1)Java Function file
- 2) Csv file
- 3) JSON File

**Below are the scripts for DDT and correlation:**

Java Function file: This Java file is used for setting and getting the ID of the user stored in the database. It is mainly used for Correlation and is declared in the feature file.



```

basic-auth.js  GetManufacturer.feature  JavaFunctions.java
1 package karate;
2
3 public class JavaFunctions {
4     static int id; //declaring static variable to use
5
6     public static void setId(int id)
7     {
8         JavaFunctions.id=id; //Setting the ID
9     }
10    public static int getId()
11    {
12        return id; //Getting the ID
13    }
14 }
15

```

CSV File: CSV file is used to store the Ids, names, code which are used for DDT.

```

basic-auth.js  GetManufacturer.feature  JavaFunctions.java  CreateTestCases.csv
1 code, name
2 sunshine,bags
3 day,bags
4 night,bags
5 sunny,bags

```

Code for DDT:

```

4
5 @postcall1
6 Scenario: Create a new manufacturer item #start of a feature file
7 Scenario Outline: create a user from given details.
8     Given url 'http://localhost:8080/api/v1/private/manufacturer' #path obtained from Swagger.UI
9
10    #Accessing username and password from basic-auth.js file
11    * header Authorization = call read('basic-auth.js') { username: 'admin@shopizer.com', password: 'password'
12        And header Content-Type = 'application/json'
13        And request {"code":<code>, "name":<name>} #accessing code and name from CSV file
14
15    When method POST #Declaring POST method
16    Then status 201 #201 status is used for creation
17    Then print response #printing the response
18
19 Examples:
20 | read('CreateTestCases.csv') | #Reading the CSV file
21

```

## Implementation of correlation:

```
basic-auth.js  GetManufacturer.feature  JavaFunctions.java  CreateTestCases.csv  *Customer management resource.feature  x
1 Feature: Shopizer Customer management resource
2
3 Background:
4 * url 'http://localhost:8080'
5 * def NonExistID = 1
6 * def helper = Java.type('karatemaven.JavaFunctions') #Accessing Java class in Feature file
7 * def ID = helper.getId(); #Getting ID from Java file
8 * def customerPayload = read('Customer.json') #reading Customer payload daata from JS
9
10 #Create a Customer
11 Scenario: TC_04:Create a customer with valid authentication
12   Given path '/api/v1/private/customer'
13   * header Authorization = call read('basic-auth.js') { username: 'admin@shopizer.com', password: 'password'
14   And request customerPayload[0] #Accessing payload data from JSON file
15   When method POST
16   Then status 200
17   Then print response
18   * helper.setId(response.id); #Setting the created ID
19
20 #Get a Customer
21 Scenario: TC_08:Get a customer using id with valid authentication
22   Given path '/api/v1/private/customer/'+ID #Accessing ID from Java File
23   * header Authorization = call read('basic-auth.js') { username: 'admin@shopizer.com', password: 'password'
24   When method GET
25   Then status 200
26   Then print response |
27
```

## JSON File:

```
basic-auth.js  GetManufacturer.feature  JavaFunctions.java  CreateTestCases.csv  Customer management resourc...  {}Customer.json  x
1 [{
2   "billing": {
3     "address": "123 street",
4     "billingAddress": false,
5     "city": "Toronto",
6     "company": "2018-12-12",
7     "country": "AL",
8     "firstName": "abc",
9     "lastName": "123",
10    "phone": "8888888888",
11    "postalCode": "416510",
12    "zone": "QC"
13  },
14  "delivery": {
15    "address": "123 street",
16    "billingAddress": false,
17    "city": "Toronto",
18    "company": "2018-12-12",
19    "country": "AL",
20    "firstName": "abc",
21    "lastName": "123",
22    "phone": "8882288888",
23    "postalCode": "364240",
24    "zone": "QC"
25  },
26  "emailAddress": "tester",
27  "firstName": "abc",
28  "groups": [
29    {
30      "name": "string",
31      "type": "string"
32    }
33  ],
34  "lastName": "123",
35  "password": "test",
36  "provider": null,
```

## 5.4 - Different Runners

Created a new java class named `ParallelRunner.java` for parallel execution:

```
import org.junit.Test;

import com.intuit.karate.KarateOptions;
import com.intuit.karate.Results;
import com.intuit.karate.Runner;

@SuppressWarnings("deprecation")
@KarateOptions(tags= {"@smoke"}) //Run scenarios with tag "Smoke" Parallely
public class ParallelRunner {
    @Test
    public void testParallel() {
        Results results = Runner.parallel(getClass(),5);
    }
}
```

## 5.5 – Karate Reports

Report for Customer Management resource.feature file:

The screenshot displays a Karate test report for the file `karatemaven.Customer management resource.feature`. The report is organized into a sidebar and a main content area.

**Sidebar Summary:**

- Scenarios: 27 (Passed), 8 (Failed)
- Timestamp: 2021-12-17 06:13:09 PM
- Scenarios List:
  - [1:11] TC\_01: Get list of customers with valid authentication (Passed)
  - [2:19] TC\_02: Get list of customers with invalid authentication (Failed)
  - [3:28] TC\_03: Get list of customers with valid authentication but limited authorization (Failed)
  - [4:37] TC\_04: Create a customer with valid authentication (Passed)

**Main Content Area:**

**Scenario: [1:11] TC\_01: Get list of customers with valid authentication** (ms: 251)

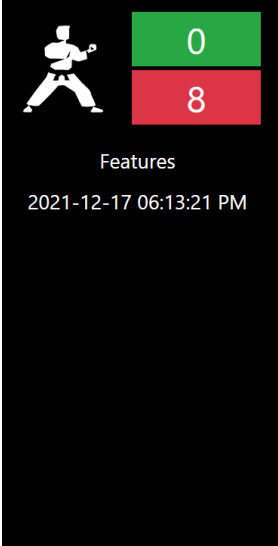
Step	Step Description	Duration (ms)
>>	Background:	
12	Given path '/api/v1/private/customers'	0
13	* header Authorization = call read('basic-auth.js') { username: 'admin@shopizer.com', password: 'password' }	3
14	When method GET	239
15	Then status 200	0
16	Then print response	1
17	Then match response.customers == '#array'	1

**Scenario: [2:19] TC\_02: Get list of customers with invalid authentication** (ms: 116)

Step	Step Description	Duration (ms)
>>	Background:	
20	Given path '/api/v1/private/customers'	0
21	* header Authorization = call read('basic-auth.js') { username: 'admin', password: 'password' }	2
22	And header Content-Type = 'application/json'	0
23	When method GET	110
24	Then status 401	0
25	Then print response	0
26	And match response.error == 'Unauthorized'	1

**Scenario: [3:28] TC\_03: Get list of customers with valid authentication but limited authorization** (ms: 115)

## Report of the summary:



Tags   Timeline					
Feature	Title	Passed	Failed	Scenarios	Time (ms)
karatemaven/GetManufacturer.feature	Shopizer Manufacturer / Brand management resource	15	2	17	1469
karatemaven/Category management resource.feature	Shopizer category management resources	11	1	12	1875
karatemaven/Catalog management resource.feature	Shopizer catalog management resources	10	4	14	3220
karatemaven/productAttributes&Options.feature	Shopizer product attributes-options management resources	20	2	22	4194
karatemaven/productTypeResource.feature	Product Type Resource	8	1	9	1710
karatemaven/Customer management resource.feature	Shopizer Customer management resource	27	8	35	5156
karatemaven/User management resource.feature	Shopizer User management resources	9	1	10	2042
karatemaven/Product display and managment resource.feature	Shopizer Product Display and Management Resource	9	1	10	1238

## 5.6 Cucumber Reporting:

1) Add dependency of Cucumber-Reporting to POM.XML:

```
<dependency>
  <groupId>net.masterthought</groupId>
  <artifactId>cucumber-reporting</artifactId>
  <version>5.6.1</version>
  <scope>test</scope>
</dependency>
```

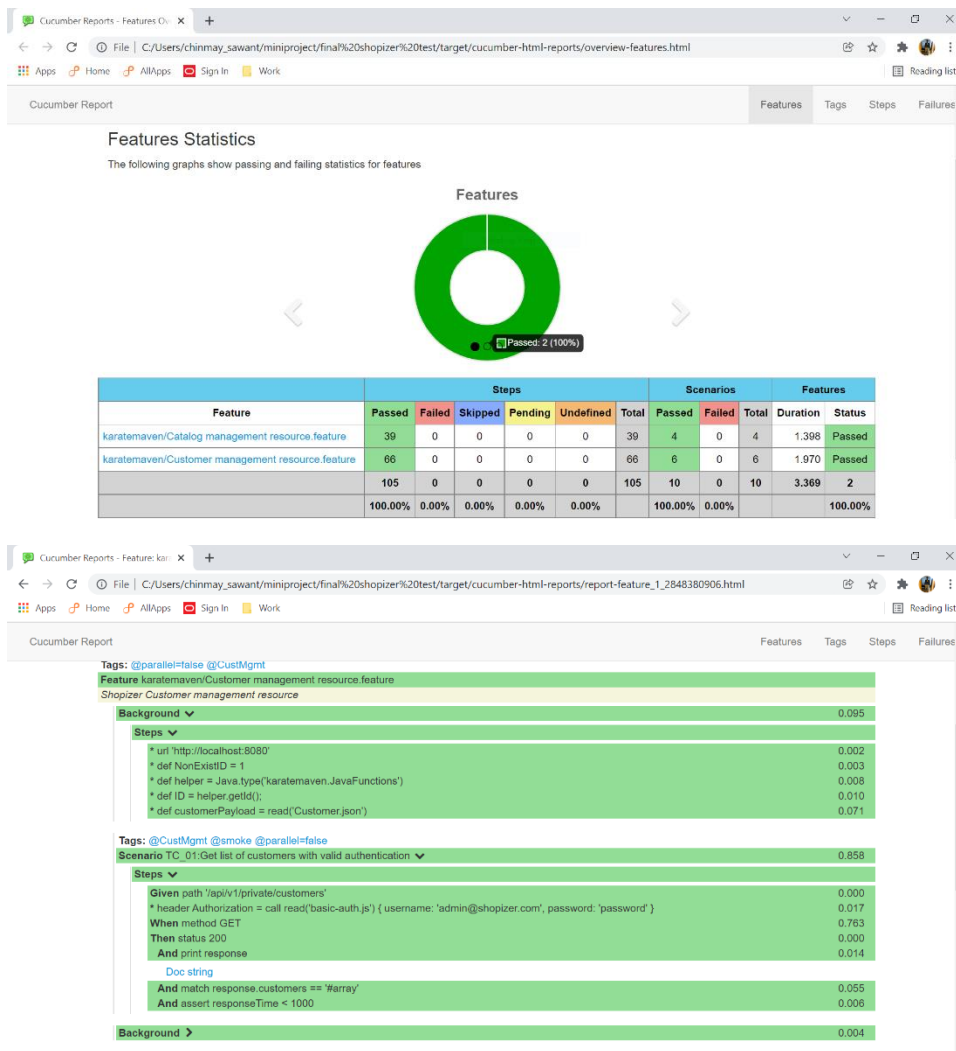
2) Created a java class named RunnerWithCucumberReport for Cucumber Reporting:

```

1 package karatemaven;
2
3 import com.intuit.karate.Results;
4
5 public class RunnerWithCucumberReport {
6     @Test
7     public void testParallel() {
8         Results results = Runner.path("classpath:karatemaven")
9             .tags("@smoke") //This will only run scenarios with tag "Smoke"
10            .outputCucumberJson(true)
11            .parallel(5); //This will execute the scenarios parallelly
12        generateReport(results.getReportDir());
13        assertTrue(results.getErrorMessages(), results.getFailCount() == 0);
14    }
15
16    /*This Method will generate Cucumber Report
17     */
18    public static void generateReport(String karateOutputPath) {
19        Collection<File> jsonFiles = FileUtils.listFiles(new File(karateOutputPath), new String[] {"json"}, true);
20        List<String> jsonPaths = new ArrayList<String>(jsonFiles.size());
21        jsonFiles.forEach(file -> jsonPaths.add(file.getAbsolutePath()));
22        Configuration config = new Configuration(new File("target"), "demo");
23        ReportBuilder reportBuilder = new ReportBuilder(jsonPaths, config);
24        reportBuilder.generateReports();
25    }
26 }
27
28

```

### 3) Screenshots of Cucumber Reports:



## **Chapter – 06 Conclusion**

Throughout the lifecycle of the project SCRUM methodology and central tracker was strictly followed enabling us to complete the project on time. The project includes Manual Testing, GUI Testing and API Testing on Shopizer(a locally hosted AUT). The testing was performed using all the tools and concepts mentioned in the problem statement, during the life cycle of the project and all the primary goals were timely achieved. Cross-browser testing, TestNG, DDT, POM and Extent Reports are all a part of the project. Postman tool was used for implementing API Testing with the available API of the AUT and Karate was an additional challenge for us to perform the automation more effectively.