NIMBAL

Test Automation Made Easy

USER MANUAL OF WEB IDE

Accelerate, Improve and Plan success with an economical code.

https://www.nimbal.io/

INDEX

INTRODUCTION

Testing is the method of evaluating and verifying that a software program product or utility does what it is meant to do. The advantages of testing is finding and getting rid of bugs, decreasing development fees, and enhancing performance.

Involvement of testers in requirement critiques and consumer tale refinement- Involving testers at some stage in the requirement segment guarantees identity of a number of the requirement defects even earlier than their implementation. It considerably reduces the solving cost. Also, the tester profits considerable task perception at this stage, it turns to facilitate him in the execution segment of the task.

PRE-REQUISITE

1. Knowledge of XPath
2. Basics of Testing concepts
3. Access to Nimbal Web IDE i.e., its URL, such as nimbal-webide.getskills.co.nz
4. Your browser should have the following programs installed
    1. Selector Hub : it is a helper browser extension, which gives us prebuilt XPath.
    2. ChroPath: it is an alternative extension for XPath.

CREATING TEST CASES

Let us create a new feature page and test the following cases using some scenarios/cases

Follow the steps below to create a new file

1. Click on the file's  icon.
2. Navigate to src > java > feature.
3. A dialog box will appear and will ask for the name of the new file enter any name with login. feature extension

   For example- login. feature for login related feature test scenarios as shown in the below screenshot

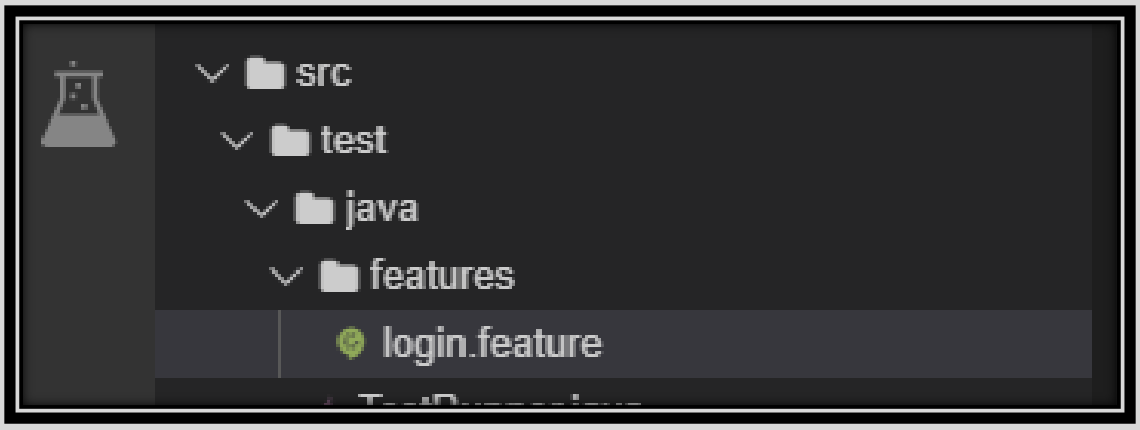

Figure 1: Path to the file

CODES AND TEST CASES

Setting up the project configuration

We need to add some properties in the following files in Web IDE

1. **config.dev.properties** : Located at **/home/project/src/test/resources/env/config.dev.properties**
   Add the following values as shown below in Figure 4
   a. app.gmail = gmail.com

      instead of gmail, you can replace it with any constant value of your concern. For example, app.website = www.website.com

   b. web.browser=chromeheadless
   Specified which browser to the user for running the test. other values possible are firefox

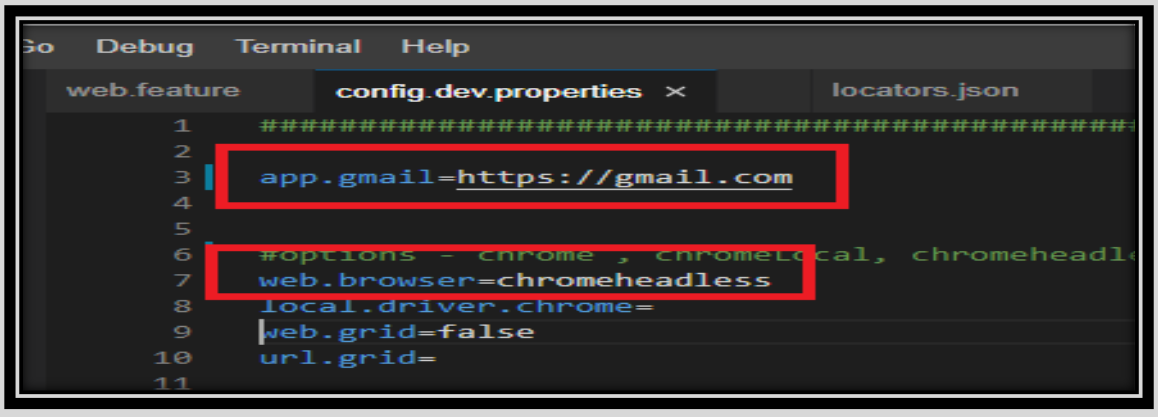

Figure 2: config.dev.properties

2. **locators.json** : Located at **/home/project/src/test/resources/locators.json** .We will use this file to add XPath key pair values, while key will be user understandable keyword and value will be an XPath, which is used to locate an HTML element within a webpage. For example, XPath for

inputting email id in gmail.com page will be **//input[@id='identifierId']** as shown below in Figure 2.

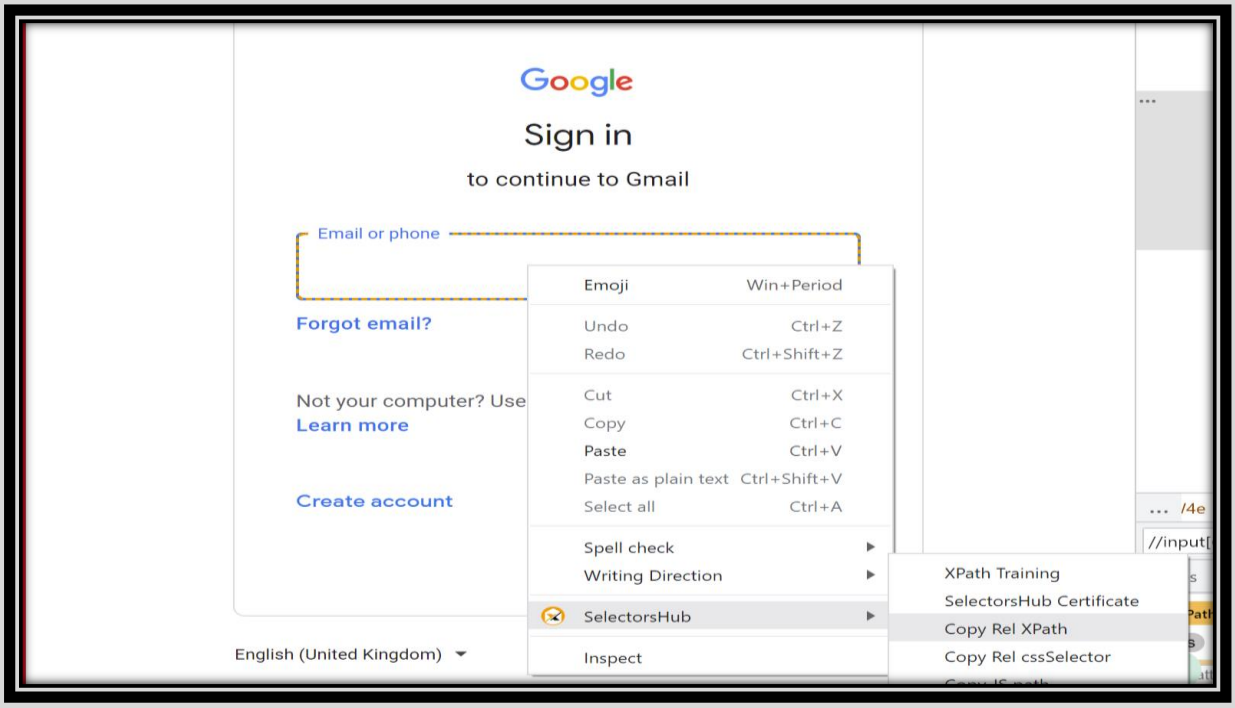

Figure 3: illustrates how to choose the XPath through Selectors Hub

Therefore, the locators.json file will look something like the below after making changes

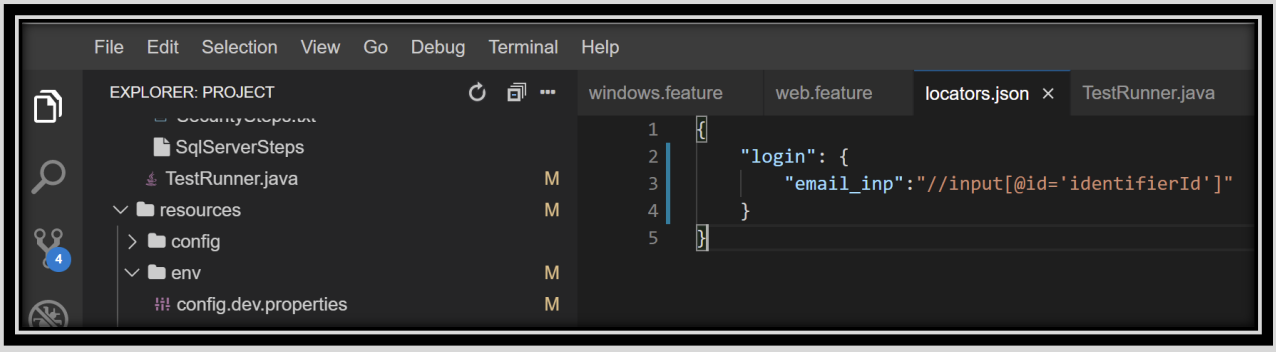

Figure 4: locator.json file after adding XPath for Gmail input location from Gmail.com

3. **TestRunner.java**: Located at **/home/project/src/test/java/TestRunner.java** it is a file used to indicate which tests need to be run using tags. A tag can be placed on a scenario/test or a feature. It usually starts **"@"** keyword.

For example,

1. Figure 6 depicts the functioning of the **TestRunner.java** file and the tag provided in the tags section written at line number 14 as
   tags**={**"**@gmailLoginNegativePath**"}

   Later we will replace this tag to run the tests we want to execute.

2. If tags are empty then the TestRunner.java file will run all the feature files present in the project as shown below
   tags={“”}

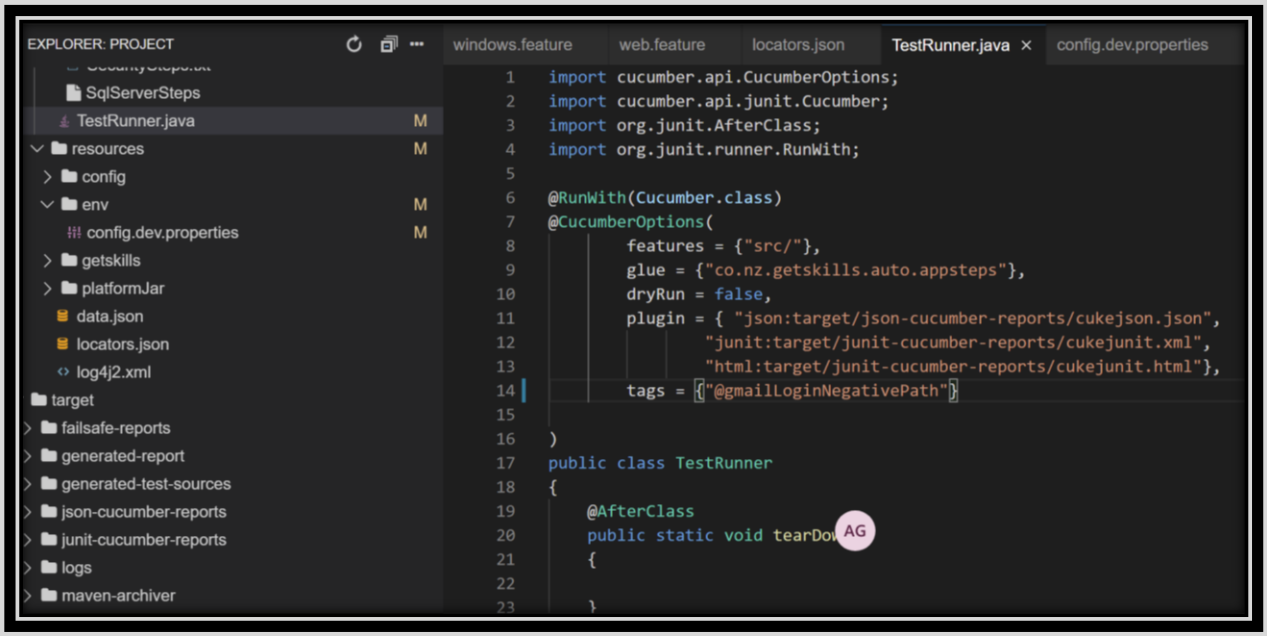

Figure 5: TestRunner.java file

Ways to run the test.feature file

1. Right-click on the TestRunner.java file and click Run. The results will be shown in the terminal.

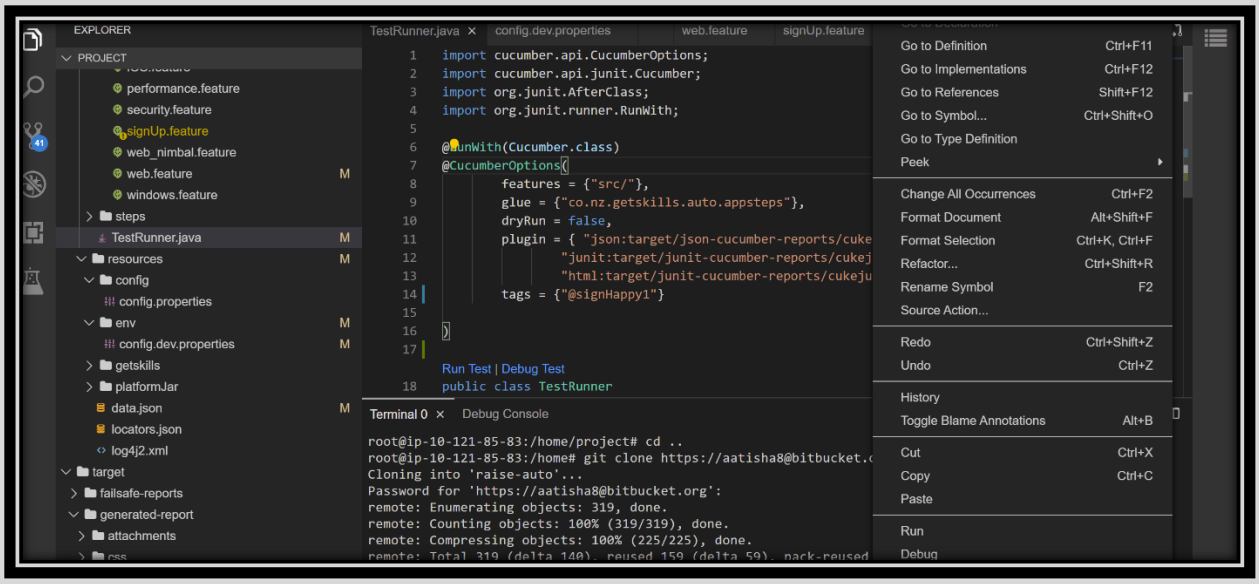

Figure 6:Running a test case

2. In the terminal, enter **mvn install** command and it will show the results in the terminal.

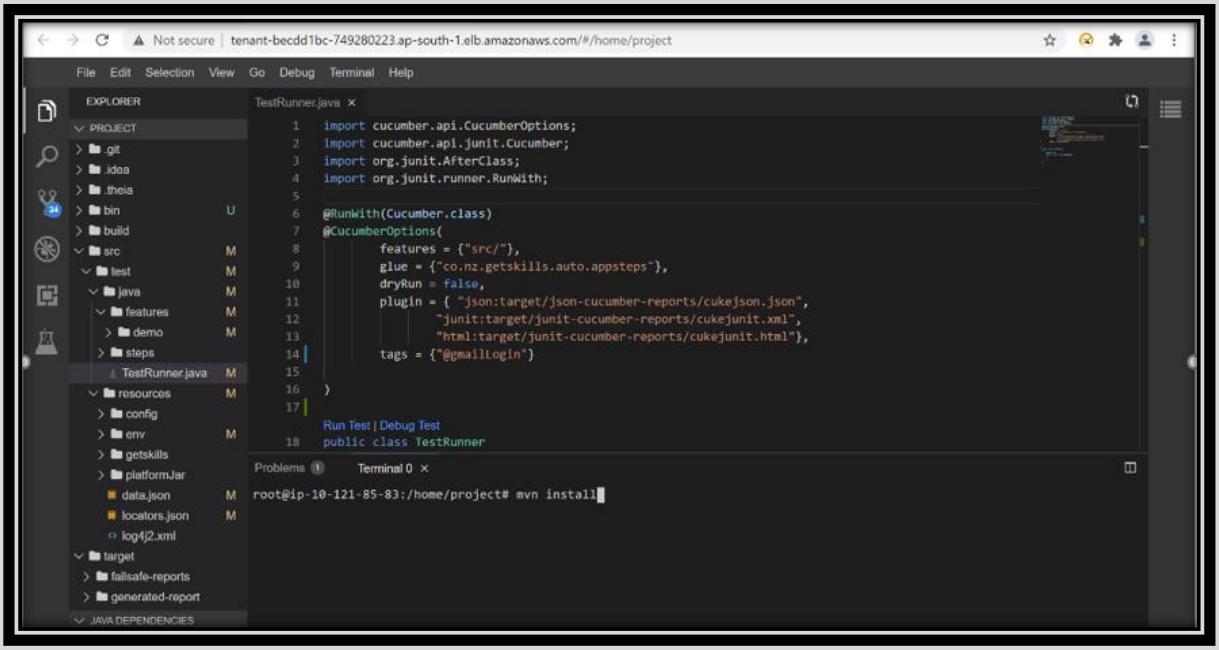

Figure 7:TestRunner.java

Run 1: Running Happy Path Scenario

Following is the code is written for the Negative path as shown in code snippet 1. Happy path test is a well-defined test case using known input, which executes without exception and produces an expected output.

```
Feature: Login in the app

@gmailLoginHappyPath
Scenario: Login in the Gmail- Happy path
Given I open gmail.com
And I fill input email_inp with abcd123
And I fill input password_inp with 123@abcd
And I click element login_btn
And I can see the text "Compose"
```

Code Snippet 1: Happy Path for Gmail Login

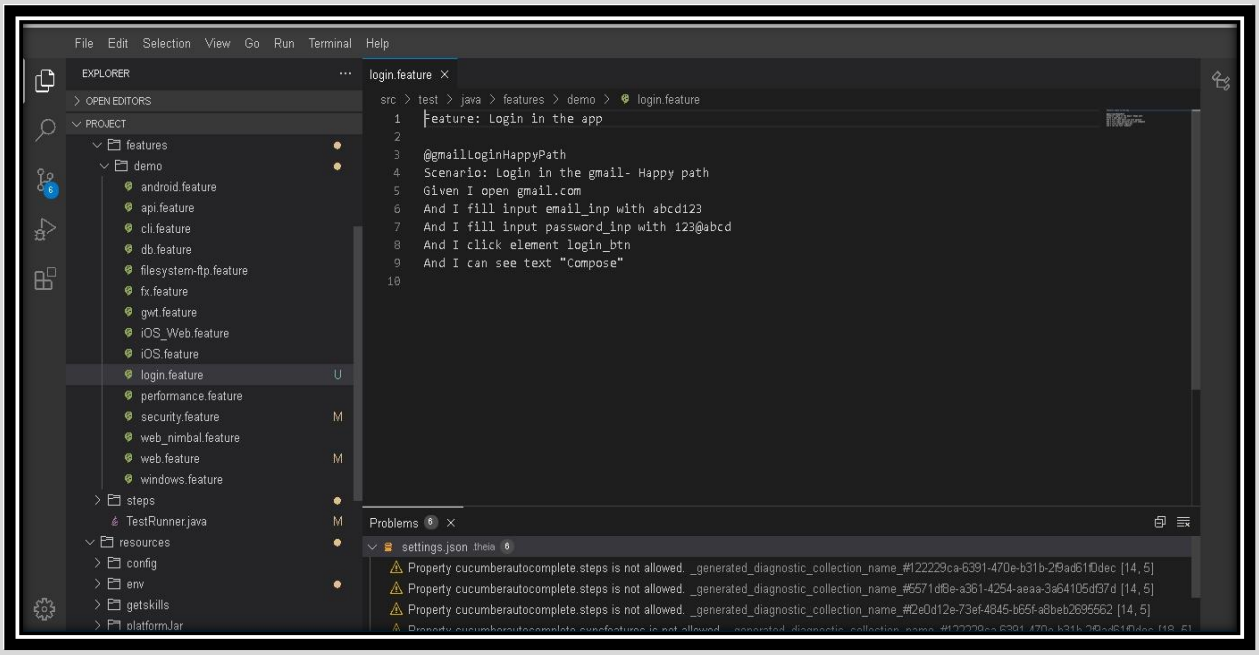

Figure 8: Reference image in Web IDE Happy Path

Explanation of the Code

1. Title of the Feature, always start with Feature Keyword
2. The keyword of the whole case (No need to write the complete code for testing just enter the keyword)
3. Description of the code
4. The user opens the URL gmail.com
5. Then users enter the data as `abcd123` in the email field (fill input is a keyword for giving input)
6. Then users enter the data as `123@abcd` in the password field (fill input is a keyword for giving input)
7. Then by clicking the `login_btn` (Button named Login)
8. User login successful can view the compose button

Run 2: Running Negative Path Scenario

Following is the code is written for the Negative path as shown in code snippet 2. Negative testing ensures that your application can gracefully handle invalid input or unexpected user behavior.

```
@gmailLoginFeature
Feature: Login in the app

@gmailLoginNegativePath
Scenario: Login in the Gmail- Negative path
Given I open gmail.com
And I fill input email_inp with abcd123
And I fill input password_inp with 123@abcd
And I click element login_btn
And I check the message "Wrong password. Try again or click ‘Forgot password’ to
 reset it."
```

Code Snippet 2: Negative Path for Gmail Login

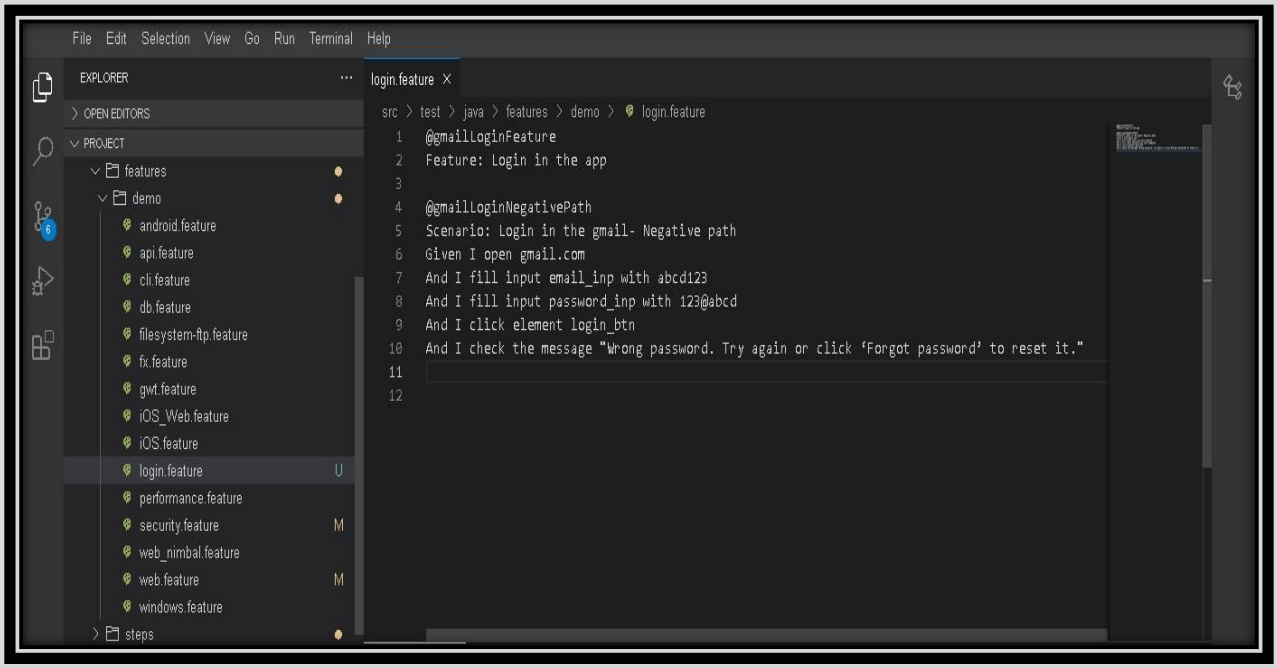

Figure 9: Reference image in Web IDE for Negative path

Explanation of the Code

1. The keyword of the whole case (No need to write the complete code for testing just enter the keyword)
2. Title
3. The keyword of case 2(No need to write the complete code for testing just enter the keyword)
4. Description of the code
5. The user opens the URL gmail.com
6. Then users enter the data as abcd123 in the email field (fill input is a keyword for giving input)
7. Then users enter the data as 123@abcd in the password field (fill input is a keyword for giving input)
8. Then by clicking the login_btn (Button named Login) User login successful, can view the message "Wrong password. Try again or click 'Forgot password' to reset it."

Run 3: Running the whole feature

Following is the complete code for our feature. Which can be executed by the adding tag **@gmailLoginFeature** in the **TestRunner.java** file.

```
@gmailLoginFeature

Feature: Login in the app

@gmailLoginHappyPath
Scenario: Login in the Gmail- Happy path
Given I open gmail.com
And I fill input email_inp with abcd123
And I fill the input password with 123@abcd
And I click element login_btn
And I can see the text "Compose"

@gmailLoginNegativePath
Scenario: Login in the Gmail- Negative path
Given I open gmail.com
And I fill input email_inp with abcd123
And I fill input password_inp with 123@abcd
And I click element login_btn
And I check the message "Wrong password. Try again or click 'Forgot password' to
 reset it."
```

Code Snippet 3: complete feature file code

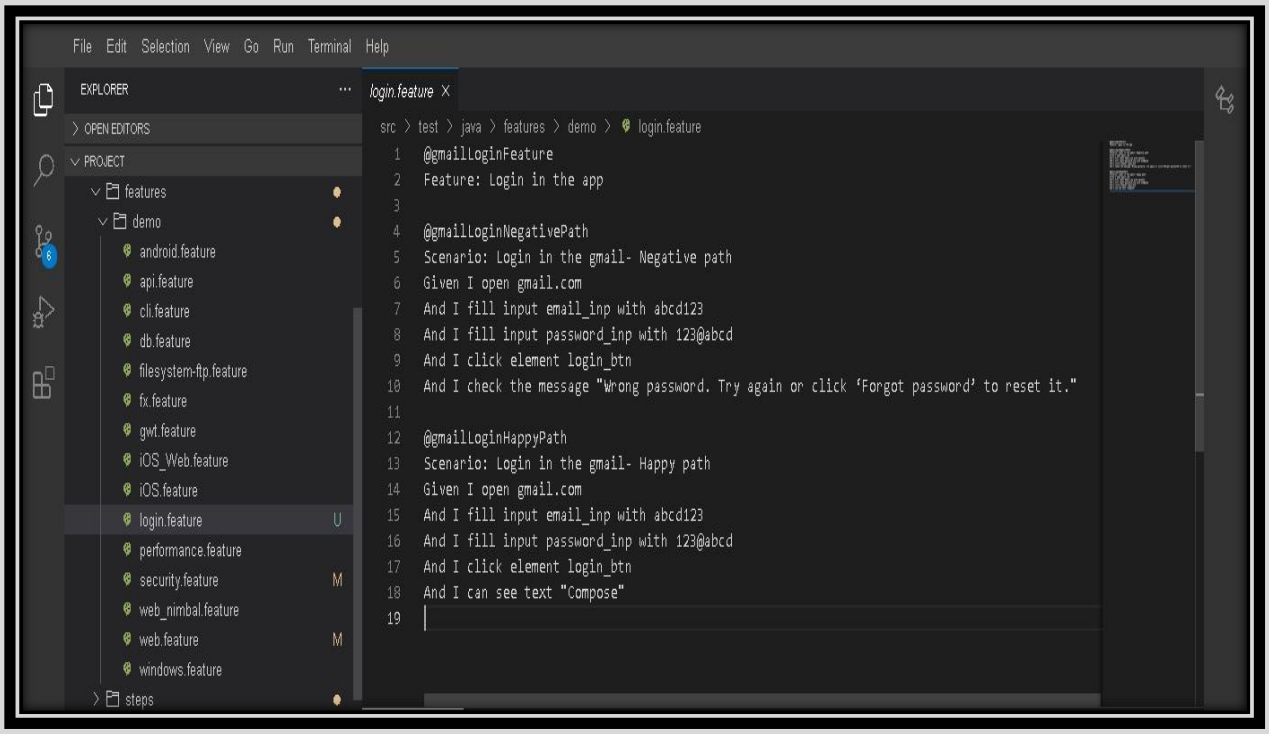

Figure 10: Reference image in Web IDE for complete code

RUNNING TEST USING TAGS

There are two components to understand how to run tests (features/scenarios) by tags

1. Tags:

Various tags are used in feature files. We can identify a tag as a group of features or a group of scenarios or a group of both features and tags. Depending on a tester.

For example, in login.feature file above files we have used the following tags

- **@gmailLoginFeature** – To execute complete feature
- **@gmailLoginHappyPath** – To execute the happy path scenario
- **@gmailLoginNegativePath** – To execute the negative path scenario

2. TestRunner.java file:

It is used for running the test with the help of tags. Every feature and scenario has a tag associated with it. We will insert this tag in the TestRunner.java file and run the program to get the desired outcome of the test using reports.

Following is the way to utilize the TestRunner.java file to run the tests.

1. Click on the file's  icon. Navigate to src > java > TestRunner.java

TestRunner.java

```
1   import cucumber.api.CucumberOptions;
2   import cucumber.api.junit.Cucumber;
3   import org.junit.AfterClass;
4   import org.junit.runner.RunWith;
5
6   @RunWith(Cucumber.class)
7   @CucumberOptions(
8           features = {"src/"},
9           glue = {"co.nz.getskills.auto.appsteps"},
10          dryRun = false,
11          plugin = { "json:target/json-cucumber-reports/cukejson.json",
12                  "junit:target/junit-cucumber-reports/cukejunit.xml",
13                  "html:target/junit-cucumber-reports/cukejunit.html"},
14          tags = {"@AddYourTagHere"}
15
16  )
17  public class TestRunner
18  {
19      @AfterClass
20      public static void tearDown()
21      {
22
23      }
24  }
```

Figure 11: Test case Runner

2. In any pre-created tag can be used for the test by taking any Tag from an existing code then writing it in the place of **"@AddYourTagHere"**

For example tags = {"@gmailLoginFeature"}

Here the tag @gmailLoginFeature is used.

TestRunner.java

```
1   import cucumber.api.CucumberOptions;
2   import cucumber.api.junit.Cucumber;
3   import org.junit.AfterClass;
4   import org.junit.runner.RunWith;
5
6   @RunWith(Cucumber.class)
7   @CucumberOptions(
8           features = {"src/"},
9           glue = {"co.nz.getskills.auto.appsteps"},
10          dryRun = false,
11          plugin = { "json:target/json-cucumber-reports/cukejson.json",
12                  "junit:target/junit-cucumber-reports/cukejunit.xml",
13                  "html:target/junit-cucumber-reports/cukejunit.html"},
14          tags = {"@AddYourTagHere"}
15
16  )
17  public class TestRunner
18  {
19      @AfterClass
20      public static void tearDown()
21      {
22
23      }
24  }
```

Figure 13: Generated Report -JSON

REPORTS

The report is generated each time when the user runs a tag using the maven command or by running right click on the TestRunner.java file.

1. The report generated can be seen under the targets folder that is located at /home/project/target.

2. The json reports are found at /home/project/target/json-cucumber-reports.

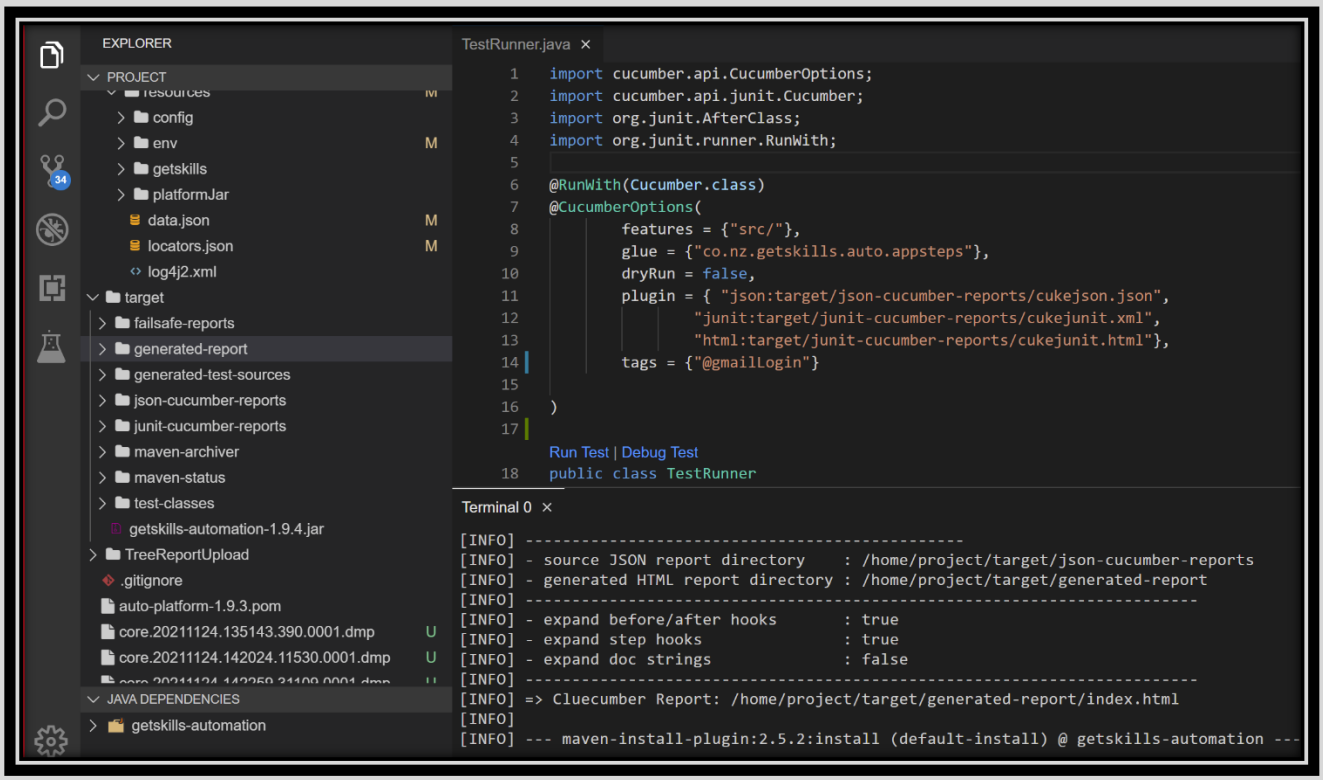

Figure 14: Generated Report -JSON

3. The generated HTML report is found at /home/project/target/generated-report.

4. If any test fails the auto-generated report then render's clarity where it is failing, one can see the captured screenshots at **/home/project/target/generated-report/attachments**

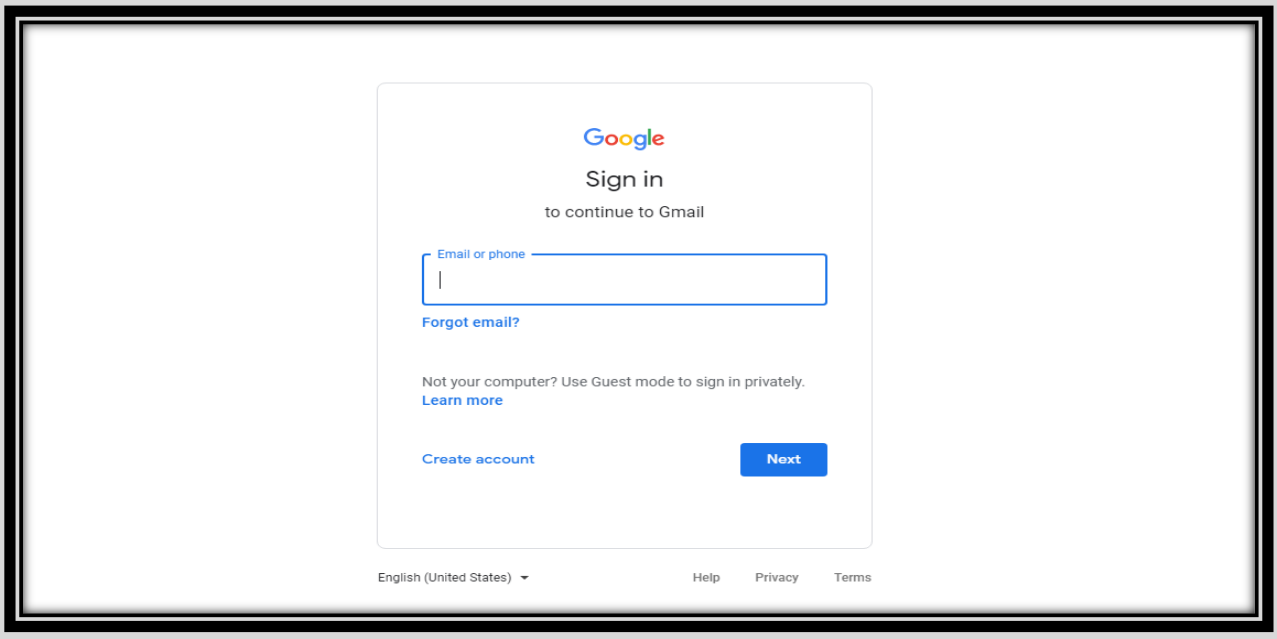

Figure 15: Screenshots in auto-generated report

It shows the page that we are getting on login.

Reviewing Report

1. Right-click on target/generated-report as shown below

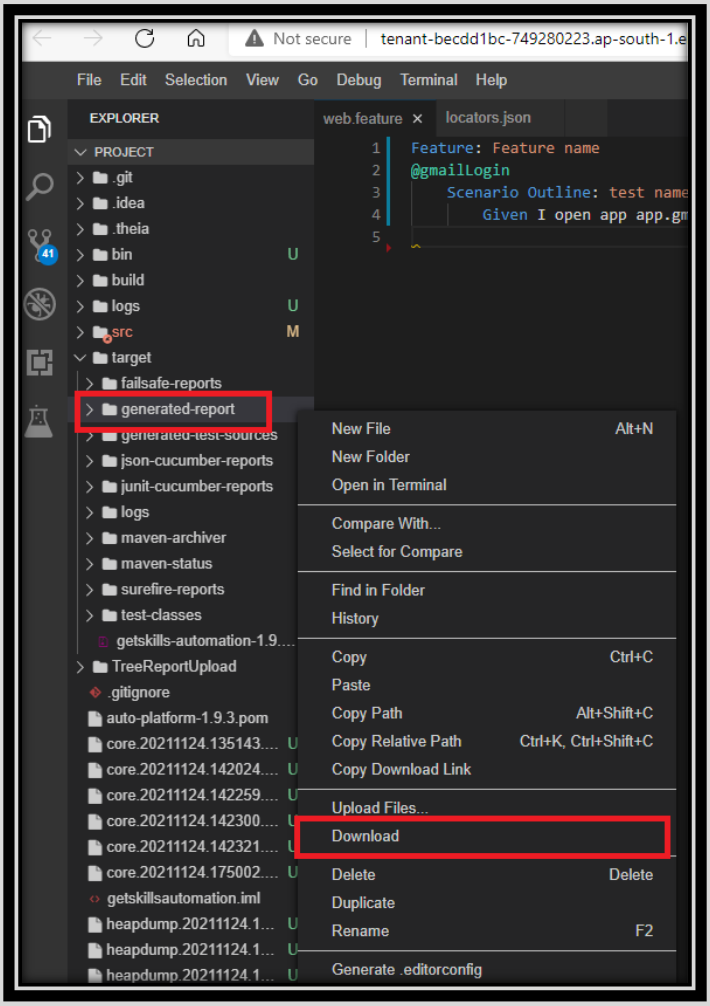

Figure16: Auto-generated report

2. A file will be downloaded as shown below

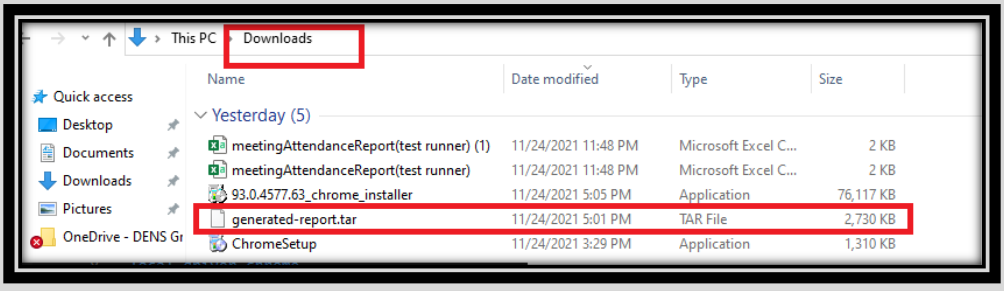

Figure17: Download the auto-generated report

3. Extract this file in your folder as shown below

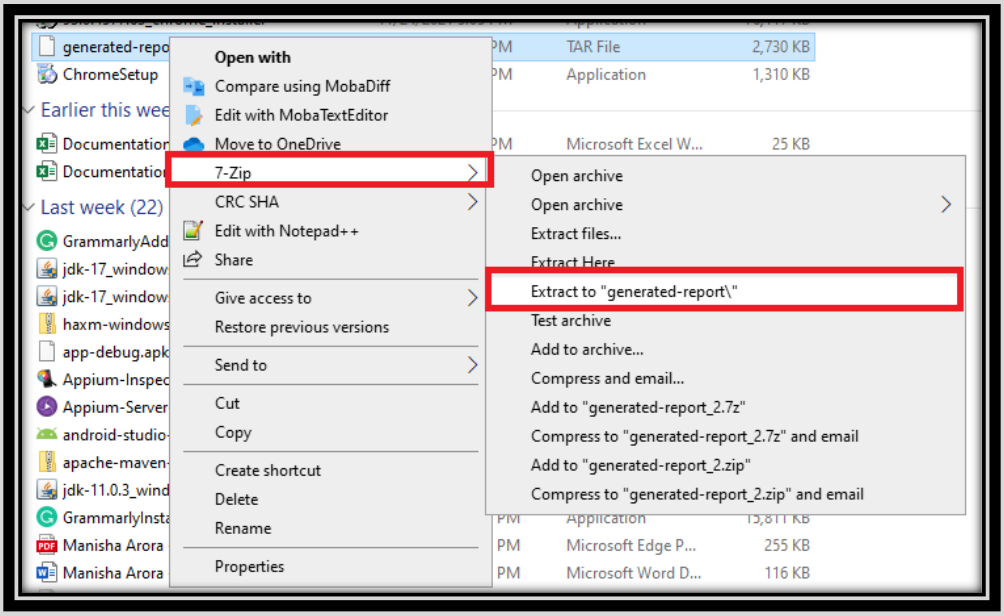

Figure18: Extract to auto-generated report

4. After the extraction, a folder name **generated-repor**t will appear in the same directory as shown below,

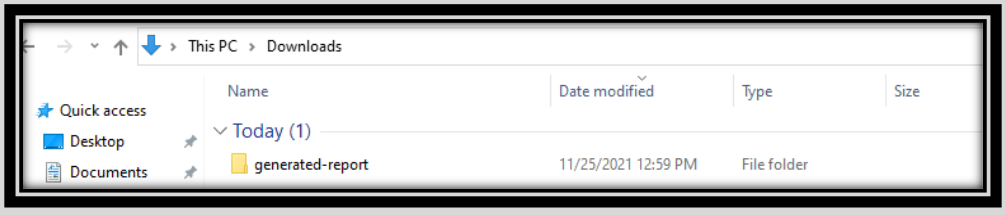

Figure 19: Downloaded file

5. Inside this folder open the index file in the browser to see the report.

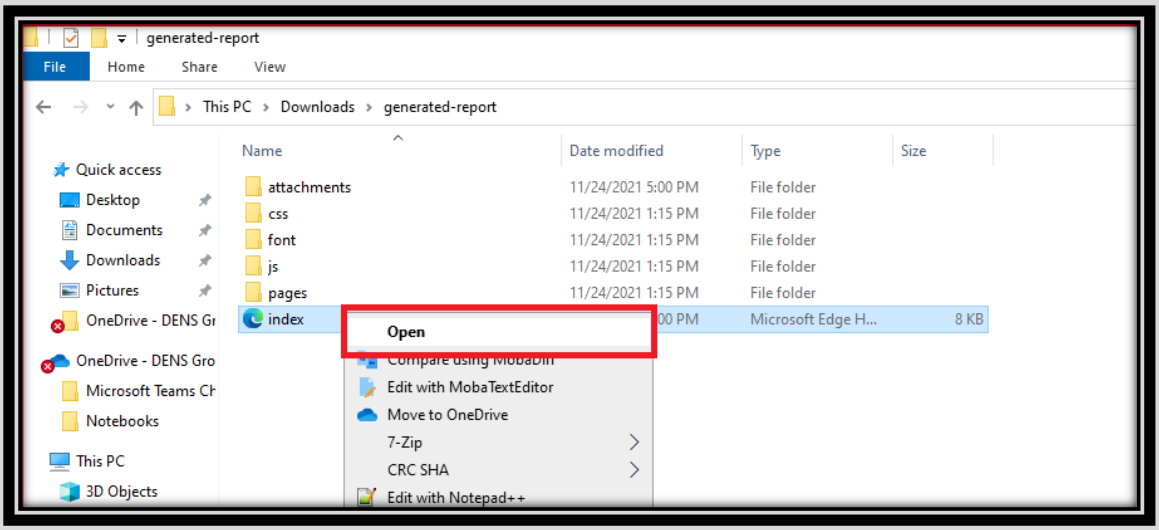

Figure 20: Screenshots in auto-generated report

6. Here is example of the report shown below

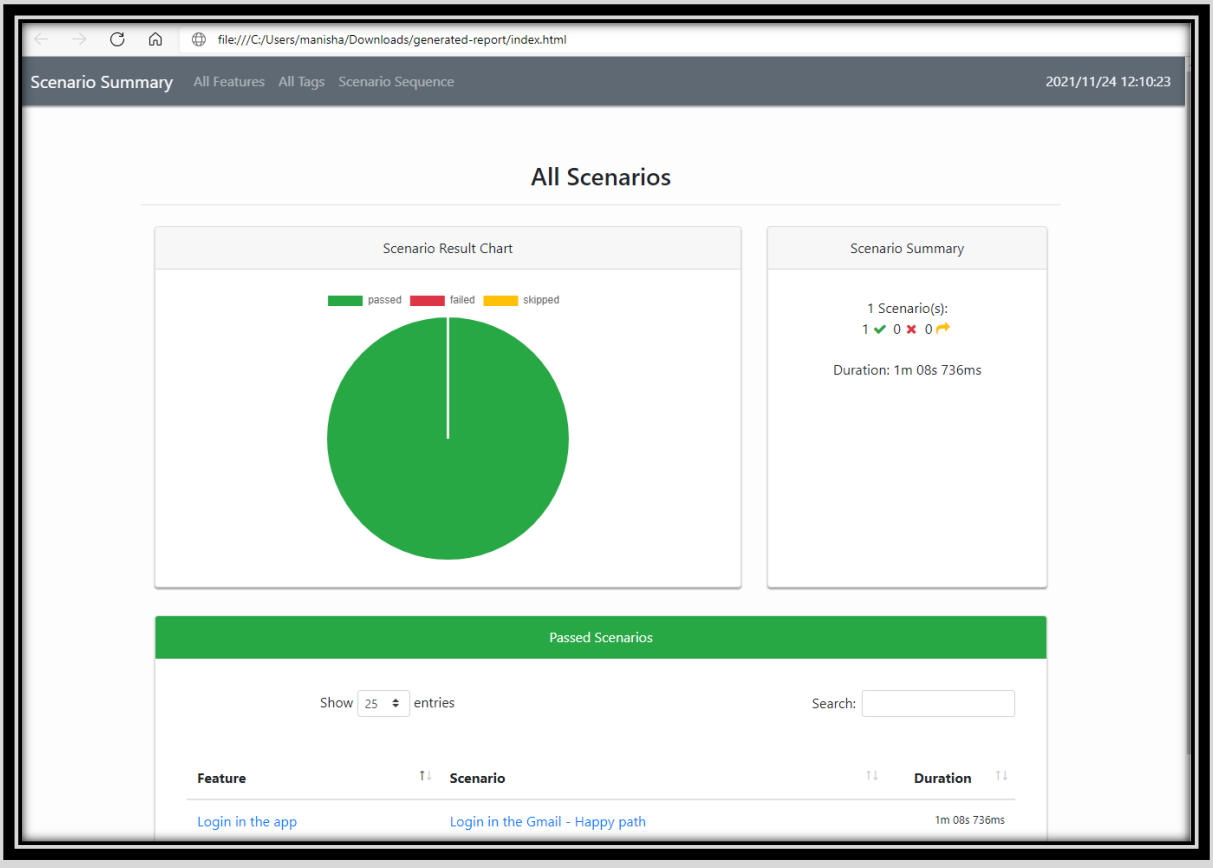

Figure 21: Scenario

REFERENCES

www.javatpoint.com
www.w3schools.com
www.geeksforgeeks.com