**Cucumber**

**Topics Covered :**

---------------------------------------------------------

1. Gherkin keywords

* Feature
* Scenario
* Given
* When
* And
* But
* Then

1. Types of Frameworks
2. What is Feature
3. What is Feature file
4. StepDefinition file
5. TestRunner Class
6. Passing data with the help of regular expression

Ex: *And* enter user name "Admin" and password "admin123"

1. Passing data with help of Datatable

Ex:

**Feature:** Login to HRM Application

**Scenario:** Login with valid credentials

*Given* User is on HRMLogin page

*When* User enters valid credentials

| Admin | admin123 |

*Then* user should be able to log in successfully and be taken to a new page

@When("User enters valid credentials")

**public** **void** entersValidCredential(DataTable dataTable) **throws** InterruptedException{

System.***out***.println("Credentials Entered");

List<String> signUpForm = dataTable.asList();

String userName = signUpForm.get(0);

String passWord = signUpForm.get(1);

System.***out***.println(userName);

System.***out***.println(passWord);

}

1. Parameterisation : For parameterization we have to write

Scenario Outline and provide Examples keyword

Ex:

**Feature:** sign up form

**Scenario Outline:** user with only mandatory fields

*Given* user is on sign up form page

*When* enters only the mandatory fields user *<username>* and *<password>*

*Then* click on the sign up button

**Examples:**

| username | password |

| sai | Cucumber123 |

| monika | Automation123 |

@When("^enters only the madatory fields user (.+) and (.+)$")

**public** **void** enters\_only\_the\_madatory\_fields\_user\_and(String username, String password) **throws** Throwable {

System.***out***.println(username);

System.***out***.println(password);}

* **Background keyoword** allows only preconditions and the hook concept for preconditions and postconditions.

1. CucumberOptions

* features
* glue
* plugin
* dryRun
* monochrome
* tags

**Cucumber**

----------------

**What is Cucumber?**

-Cucumber is a tool/framework that supports BDD ( Behaviour Driven Development ).

**Framework Types:**

1. **Predefined Framework**

* TestNg ( TDD 🡪 Test Driven Development Frame work )
* JUnit ( TDD 🡪 Test Driven Development Frame work )
* Cucumber ( BDD 🡪 Behaviour Development )

**2. Userdefined Framework**

-------------------------------------

* DataDriven
* keywordDriven
* HybridDriven

**Differences between TDD and BDD**

------------------------------------------------

**1. TDD Framework**

-----------------------------

**Process of TDD**

----------------------

1. From the inputs, develop the test scripts

2. Create an XML file and run the XML file in TestNg or JUnit

3. If the test script fails, develop the code until it meets the expected results.

4. Refactor the code

5. Repeat the above steps for the new test Cases

In the TDD framework based on manual test cases they develop test scripts using the java programming language and Selenium framework.

**Disadvantages**

---------------------

-These test scripts are not understandable to non-technical people or business persons

-Also there is a change of missing development in some of the functionalities

**TDD**

*---------*

-Process starts with writing test cases

-Written in the programming language

-collaboration is between only developers and testers

**2. BDD Framework**

-----------------

Manual

Test Cases 🡪 Feature Files 🡪 Step Definition File 🡪 JUnit Class Runner

**Process of BDD**

-----------------------

1. BDD is an extension to the TDD, in BDD we start with writing a behaviour ( Feature file )

2. Later we develop the code or automate the simple English language into the programming language

3. Run the behaviour if success move to the subsequent behaviour otherwise fix the errors in the functional code to acheive the application behaviour.

4. Repeat the steps

**BDD**

---------

-Process starts with writing the behaviour

-Feature file is in simple language and can be understood easily

-Collaboration is between Testers, Developers, and Business people

==================================================================

**Steps starting with cucumber**

----------------------------------------

Step: 1

----------

1. Create a Folder in the project for storing feature files

2. Create a file with an extension feature for creating a feature file

3. Create a feature file with the following keywords:

Gherkin keywords

---------------------

1. Feature – a Title for your feature

2. Description - detailing the feature

3. Scenario – a Title for your scenario

4. Given – I want to a step with precondition

5. When – I complete action

6. And – some other action

7. And – yet another action

8. Then – validate the outcome

9. And – check more outcomes

Step: 2

--------

1. Create a package for step definitions

2. Create step definition for feature file

Step: 3

---------

1. Create a package for TestRunner Class (cucumberOptions)

2. Create a TestRunner Class

3. Running with JUnit TestRunner Class is below :

package cucumberOptions;

import org.junit.runner.RunWith;

import io.cucumber.junit.Cucumber;

import io.cucumber.junit.CucumberOptions;

@RunWith(Cucumber.class)

@CucumberOptions(

features="src/test/java/features",

glue="stepDefinitions")

public class TestRunner {

}

**4. Running with TestNg TestRunner**

package cucumberOptions;

import io.cucumber.testng.AbstractTestNGCucumberTests;

import io.cucumber.testng.CucumberOptions;

@CucumberOptions(

features = "src/test/java/features",

glue="stepDefinitions")

public class TestNgTestRunner extends AbstractTestNGCucumberTests{

}

3. Run the Test Runner

**Note:** We run the TestRunner class to execute Cucumber features.

**Note** : Actions in the feature file can be created logically so that we can avoid creating multiple stepdefinitions .

Ex: *And* enter user name "Admin" and password "admin123"

* By matching the feature file step in the other scenarios using dynamically passing of data, we can avoid creating multiple steps.

**Tagging Feature to control test Cases :**

**--------------------------------------------------------------**

* We have to remember **tags** keyword, we use **tags** keyword in TestRunner class in @CucumberOptions to control test cases.
* Using @name for Scenarios and tags in TestRunner Class, we can trigger only those test cases or test Scenarios at the time of execution, so in that way we can easily control test cases.

**To Run only particular test case :**

**----------------------------------------------------**

@RunWith(Cucumber.**class**)

@CucumberOptions(

features="src/test/java/features",

glue="stepDefinitions",tags="@RegTest")

**public** **class** TestRunner {

}

**To Run multiple Test Cases** : ( **or** keyword is used )

--------------------------------

@RunWith(Cucumber.**class**)

@CucumberOptions(

features="src/test/java/features",

glue="stepDefinitions",tags="@RegTest **or** @SanityTest")

**public** **class** TestRunner {

}

**To Run test case with both** tags : ( **and** keyword is used )

----------------------------------

@RunWith(Cucumber.**class**)

@CucumberOptions(

features="src/test/java/features",

glue="stepDefinitions",tags="@RegTest **and** @SanityTest")

**public** **class** TestRunner {

}

**To omit only particular test case and run rest of all**:

--------------------------------------------------------

( **not** keyword is used )

@RunWith(Cucumber.**class**)

@CucumberOptions(

features="src/test/java/features",

glue="stepDefinitions",tags="@RegTest **not** @SanityTest")

**public** **class** TestRunner {

}

**For common prerequisites we use Background Keyword :**

----------------------------------------------------

**Note:** A **Background** keyword is only used when there is only one precondition.

* When we have to run any prerequisites before running any scenario, we use the **Background keyword** so that the prerequisite is executed first and then the scenario is executed.

**Background:**

*Given* browser is validated

*When* browser is launched

*Then* url of the application is entered

**Hooks : ( @Before , @After )**

**------------------------------------------**

* The **@Before** and **@After** in StepDefinition files are used if you have both pre and post conditions.
* For **@Before** and **@After**, we pass the name in parenthesis to control and use only for those scenarios.

@Before("@MobileTest")

**public** **void** setup() {

System.***out***.println("launch the app");

}

@After("@MobileTest")

**public** **void** tearDown() {

System.***out***.println("Close the application");

}

* When you don’t pass a name to the hooks, they will apply to all the scenarios.

**CucumberOptions:**

---------------------------------

* features : to specify the location of the feature files
* glue : to specify the location of the stepdefinition files
* plugin : to generate pretty reports of cucumber
* dryRun : running and checking unmapped feature files with stepdefinition
* monochrome : to generate neat format in the console
* tags : to control the test cases

@RunWith(Cucumber.**class**)

@CucumberOptions(

features="src/test/java/features",

glue="stepDefinitions",tags=" @MobileTest",

dryRun= **true**,

monochrome= **true**,

plugin= {"pretty","html:target/cucumber.html"})

================================================================================================================

**Cucumber Framework:**

**----------------------------------------------------------------------------**

**Steps:**

**Step1: Create a project with group id and artifact id and include all the dependencies.**

* create new File 🡪 other 🡪 maven 🡪 maven project
* Naming group id 🡪 group id is same like your name of the package
* Naming artifact id 🡪 artifact id is like your project name
* Download dependencies based on your requirement

**Step2: Create a package for feature files and create feature files with .feature extension**

* New 🡪 File 🡪 nameofthefeature.feature

**Step3: Create a package for stepDefinition files :**

**Step4: Create a TestRunner packages for TestRunner Class**

**Step5: Create a utils package for utils**

* In utils we can have reusable code

Example:  **Dependency injection**

Step1: Whenever the properties are reused in another class of the same state, provide them in a separate class (Class TestContextSetup).

Step2: Create a Constructor and pass the parameter of TestContextSetup tcs first, and declare a variable with the name of the class. This provides knowledge for those properties which are reusable.( following program will make you understand about these steps ).

Step3: Then use them in another Class again by creating constructor With TestContextSetup tcs as a parameter and declare a variable with Class name and use them.

**Reusable properties:**

**public** **class** TestContextSetup {

**public** WebDriver driver;

**public** String homePageName, topDealsName;

}

**Providing knowledge of the properties which are shared between different classes so that they can be reused.**

**public** **class** searchProductSteps {

**public** WebDriver driver;

**public** String homePageName, topDealsName;

TestContextSetup tcs;

**public** searchProductSteps(TestContextSetup tcs) {

**this**.tcs =tcs;

}

**The best practices to follow while developing the code:**

1. **Single responsibility principle**
2. **Loosely coupled**
3. **Avoid tightly coupled**

**The Single Responsibility Principle (SRP):**

The idea behind the SRP is that every class, module, or function in a program should have one responsibility/purpose in a program.

Benefits of SRP:

* When an application has multiple classes, each of them following this principle, then the applicable becomes more maintainable, easier to understand.
* The code quality of the application is better, thereby having fewer defects.
* Onboarding new members are easy, and they can start contributing much faster.
* Testing and writing test cases is much simpler

**Code Optimization :**

1. **Maintain separate package for page objects**

* Create a package
* For each page, maintain a separate class for page objects

Ex: **public** **class** SearchProductPage {

**public** WebDriver driver;

**public** SearchProductPage(WebDriver driver) {

**this**.driver = driver;

}

// declaration of the elements

By search = By.*xpath*("//input[@type='search']");

By getName = By.*xpath*("//h4[contains(text(),Tomato)]");

// actions on the elements

**public** **void** searchProduct(String name) {

driver.findElement(search).sendKeys(name);

}

**public** String getProductName() {

String productName = driver.findElement(By.*xpath*("//h4[contains(text(),Tomato)]")).getText().split("-")[0]

.trim();

**return** productName;

}

}

* Call the methods in step definition.
* By following the above steps the code will be clean and maintainable

1. **Create PageFactory package which is responsible for creating an objects**
2. **Call the methods to get the particular object**
3. **Create TextContextSetup for reusable properties ( imp )**
4. **Create page object manager for page objects**
5. **Create webdriver manager for managing webdriver setup and tear down**
6. **Create utils package for generic methods**

**Ex: 1. Switching to child window**

* **For waits**

1. **Create global properties file for global properties like, url, browser name etc**
2. **Create hooks class for post and preconditions**
3. **Run test scenarios parallel to save time**

**The following is the code to run parallel in Cucumber:**

@Override

@DataProvider(parallel = **true**)

**public** Object[][] scenarios() {

**return** **super**.scenarios();

}

Important points:

* pageObjectClass needs driver 🡪 pageObjectManager 🡪 TestContextSetup ( constructor is createad in tcs where all the objects are created and global variables are declared and shared all the classes )
* TestContextSetup will share all the required properties to all the required classes.

**Taking Screenshot for failed TestCases:**

* Use annotation @AfterStep hook
* After that add @AfterStep annotation to a method that takes a screenshot
* If any failed test cases are identified after every step, @AfterStep with method is executed.

Ex**:**

@AfterStep

**public** **void** takeScreenshot(Scenario scenario) **throws** IOException {

**if** (scenario.isFailed()) {

TakesScreenshot takeScreenshot = ((TakesScreenshot) driver);

File srcFile = takeScreenshot.getScreenshotAs(OutputType.***FILE***);

File destFile = **new** File("./src/test/resources/Screenshot/screenshot.png");

FileUtils.*copyFile*(srcFile, destFile);

}

}

**How to run failed test cases only: ( rerun )**

* You have to remember **rerun** property to run only failed test cases

**How to use rerun :**

@CucumberOptions(features = "src/test/java/features",

glue = "stepDefinitions",

plugin = {"html:target/cucumber.html","json:target/cucumber.json",

"rerun:target/failed\_senarios.txt"

}

* After executing the above testRunner all the failed test cases will be located in one folder. If you would like, you can run testRunner separately for failed test cases.
* The failed testcase folder is mapped to features property as shown below in order to run failed test cases.
* @CucumberOptions(features = "@target/failed\_senarios.txt",
* glue = "stepDefinitions",
* plugin = {"html:target/cucumber.html","json:target/cucumber.json"
* }

**Note: The following link will help you understand cucumber dependency injection.**

[**https://cucumber.io/docs/cucumber/state/**](https://cucumber.io/docs/cucumber/state/)

**What is the difference between dependency injection and inheritance?**

One criticism of inheritance is that it tightly couples parent class with child class. It is harder to reuse the code and write unit tests. That's why most developers prefer dependency injection as a way to reuse code. **Dependency injection is a way to inject dependencies into a class for use in its methods**.

**How to Run test cases using maven in eclipse:**

* Project 🡪 run as 🡪 run configurations 🡪 maven 🡪 add a file 🡪

At goals field give it as **test** 🡪 provide base directory location 🡪 run

**How to Run test cases externally using command prompt:**

* Open command prompt 🡪 give the project location 🡪 mvn test

**How to control test cases from command prompt:**

mvn test -Dcucumber.filter.tags="@smoke"

Supported properties are:

cucumber.ansi-colors.disabled= # true or false. default: false

cucumber.execution.dry-run= # true or false. default: false

cucumber.execution.limit= # number of scenarios to execute (CLI only).

cucumber.execution.order= # lexical, reverse, random or random:[seed] (CLI only). default: lexical

cucumber.execution.wip= # true or false. default: false.

cucumber.features= # comma separated paths to feature files. example: path/to/example.feature, path/to/other.feature

cucumber.filter.name= # regex. example: .\*Hello.\*

cucumber.filter.tags= # tag expression. example: @smoke and not @slow

cucumber.glue= # comma separated package names. example: com.example.glue

cucumber.plugin= # comma separated plugin strings. example: pretty, json:path/to/report.json

cucumber.object-factory= # object factory class name. example: com.example.MyObjectFactory

cucumber.snippet-type= # underscore or camelcase. default: underscore

**Jenkins: ( pending )**

**Git and github:**

**Git:** --> Local Repository

**Github:**  --> Global Repository or Central Repository or Remote Repository

**Why Git ? ( Pending )**

**Why github ? ( pending )**

**How to push the code into github:**

**Steps:**

1. You need to have the code in your local machine
2. Open command prompt
3. Swift to the project directory
4. Then follow the below commands

|  |  |  |
| --- | --- | --- |
| [**Tell Git who you are**](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) | Configure the author name and email address to be used with your commits.  Note that Git [strips some characters](http://stackoverflow.com/questions/26159274/is-it-possible-to-have-a-trailing-period-in-user-name-in-git/26219423#26219423) (for example trailing periods) from user.name. | git config --global user.name "Sam Smith"  git config --global user.email sam@example.com |

* Then create new local repository using below commands

|  |  |  |
| --- | --- | --- |
| [**Create a new local repository**](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init) |  | git init |

* Add one or more files to staging using below commands

|  |  |  |
| --- | --- | --- |
| [**Add files**](https://www.atlassian.com/git/tutorials/saving-changes#git-add) | Add one or more files to staging (index): | git add <filename>  git add \* |

* Check the status of the files

|  |  |  |
| --- | --- | --- |
| [**Status**](https://www.atlassian.com/git/tutorials/inspecting-a-repository#git-status) | List the files you've changed and those you still need to add or commit: | git status |

* Commit changes to head ( but not yet to the global git hub repository )
* To commit the changes follow the below commands

|  |  |  |
| --- | --- | --- |
| [**Commit**](https://www.atlassian.com/git/tutorials/saving-changes#git-commit) | Commit changes to head (but not yet to the remote repository): | git commit -m "Commit message" |
|  |  |

By following the above commands now our code is ready to push into the remote repository – Follow below commands to push the code to remote repository

* First connect the local repository to the remote server

|  |  |  |
| --- | --- | --- |
| [**Connect to a remote repository**](https://www.atlassian.com/git/tutorials/syncing#git-remote) | If you haven't connected your local repository to a remote server, add the server to be able to push to it: | git remote add origin <server> |
|  |  |

* **To list all currently configured remote repositories**

|  |  |
| --- | --- |
| List all currently configured remote repositories: | git remote -v |

* To push the code to the master branch of your remote repository

|  |  |  |
| --- | --- | --- |
| [**Push**](https://www.atlassian.com/git/tutorials/syncing#git-push) | Send changes to the master branch of your remote repository: | git push origin master |

**Branches:**

* To create a new branch follow the below command

|  |  |  |
| --- | --- | --- |
| [**Branches**](https://www.atlassian.com/git/tutorials/using-branches) | Create a new branch and switch to it: | git checkout -b <branchname> |
|  |  |

* To switch from one branch to another branch

|  |  |
| --- | --- |
| Switch from one branch to another: | git checkout <branchname> |

* To list all branches and also to tell at what branch you are in

|  |  |
| --- | --- |
| List all the branches in your repo, and also tell you what branch you're currently in: | git branch |

* **Delete the feature branch**

|  |  |
| --- | --- |
| Delete the feature branch: | git branch -d <branchname> |

* To push the branch to your repo

|  |  |
| --- | --- |
| Push the branch to your remote repository, so others can use it: | git push origin <branchname> |

* **To push all branches to your repo**

|  |  |
| --- | --- |
| Push all branches to your remote repository: | git push --all origin |

* **Delete a branch on your repo**

|  |  |
| --- | --- |
| Delete a branch on your remote repository: | git push origin :<branchname> |