**What is Cucumber?**

* Cucumber is the BDD (Behavior Driven Development) framework thru which we can design and run automation test cases
* Cucumber does not automate your test cases
* We can implement different features using Cucumber BDD like Parameterization, Execution control, Hooks, Reports, Automation Utilities, Data Driven tests, etc.
* Cucumber BDD framework supports different Automation test cases such as Web , Mobile, API, Unit Testing which are written in Java/Ruby language.
* Here, Test Cases/Requirements are defined with BDD methodology (using Gherkin syntax)
* No much coding is required to implement Framework functionalities

**What is Gherkin?**

* It is a language to describe software’s behavior
* It’s Business Readable, Domain specific language
* In Testing point of view, it’s language to define the test cases
* Statements written in plain English language can be interpreted differently by different people and it may lead to confusion and mistakes. To overcome this problem, Gherkin is introduced to describe behavior of software so that no one will be confused if everybody uses same language or template

**Keywords used in Cucumber:**

* **Scenario**
  + In cucumber, Test case is called as ‘Scenario’
  + And this test case or scenario is written in Gherkin language
  + Scenario contains Steps (which are nothing but test case steps)
  + Scenario uses following keywords (case sensitive):
    - **Given**
      * Pre-conditions are mentioned under Given
    - **When**
      * Here, describes user actions
    - **Then**
      * To observe expected output
      * The Observation should be related to business value
    - **And**
      * If there are multiple pre-conditions, actions or expected outputs, we use ‘And’ keyword to add them
      * It is used when there are additional pre-conditions, actions or expected outputs representing Positive statement
    - **But**
      * If there are multiple pre-conditions, actions or expected outputs, we use ‘But’ keyword to add them
      * It is used when there are additional pre-conditions, actions or expected outputs representing Negative statement
  + E.g. **Scenario**: Make minimum due payment

**Given** User is on Pay Credit Card Bill page

**When** User enters all required details and selects minimum amount option

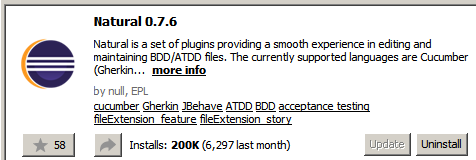
**And** user clicks on Pay button

**Then** Credit Card payment confirmation page displayed

**But** Error page is not displayed

* **Feature**
  + Feature represents Business Requirement
  + Feature File acts as a Test Suite which contains of all Scenarios
  + It’s high level business requirement; under this, we can design multiple Scenarios to test that a business requirement
  + In cucumber, feature file contains scenarios
  + We can create feature file with “.feature” extension (similar to java class in Java)
  + The text that immediately follows ‘Feature’ keyword in the same line, that text is the Title of the Feature File
  + The naming conventions for Feature file should be lower case with .feature extension
  + Feature file should contain either scenario or scenario outline
* **Feature File**
  + All the scenarios (in Given/when/then format) we write in a Feature File
  + “.feature” is the extension of feature file
  + Scenarios (test cases) we automate using Step Definition and Test Runner
* **Scenario Outline**
* **Step Definition**
  + While creating automation code for the scenario in a feature file, we map each line in the Step Definition
  + \*\*StepDefinition file matched with TagName and Descriptions in Feature file\*\*
  + \*\*There should be one mapping implementation for each Gherkin line\*\*

**Installing Cucumber:**

* We will add/install Cucumber plugin into **eClipse** editor
* For that, we have to first install Java, Eclipse editor for Java, set corresponding environment variables
* And then **install Cucumber plugin** into eClipse
* Goto Eclipse > Help > Eclipse Marketplace > search for ‘Natural’ plugin; which is used for maintaining/editing BDD files and supports Cucumber(Gherkin language)
* Install ‘Natural’ plugin
* 

**Creating Cucumber Project:**

* We can create Cucumber Project in eclipse once Cucumber plugin (‘Natural’) is installed
* Create Maven Projects in eClipse with **‘maven-archetype-quickstart’** project template; since this maven project template supports Cucumber BDD with Gherkin.

**Dependencies for Cucumber:**

* Add following dependencies in POM.xml after creating Project (you can find them in Maven Repository site)
  + Cucumber-Java
  + Cucumber-Junit/Cucumber-testng
* Install ‘Natural’ plugin; we already did it in above steps (installing Cucumber)

**Components used in Cucumber Framework:**

* **Feature File**
  + We already saw it
* **Step Definition**
  + We already saw it
* **Test Runner**
  + We use Test Runner to run the automation code
  + We have to provide Feature file and corresponding Step Definition info to Test Runner

**Feature File Creation:**

* Create a feature file under ‘src/test/java’ folder
* Create a new Package (here I created ‘com.FeatureFiles’)
* Under the package, create a simple/plain file with **extension .feature** (here I created ‘LoginToApp.feature’)
* And start writing scenarios in there as per given syntax:

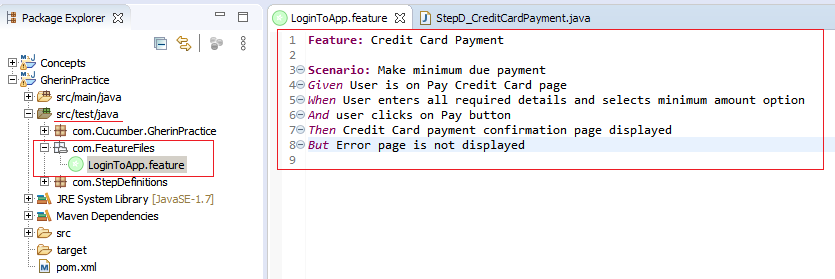
Feature: <feature name or business case>

Scenario: <scenario name>

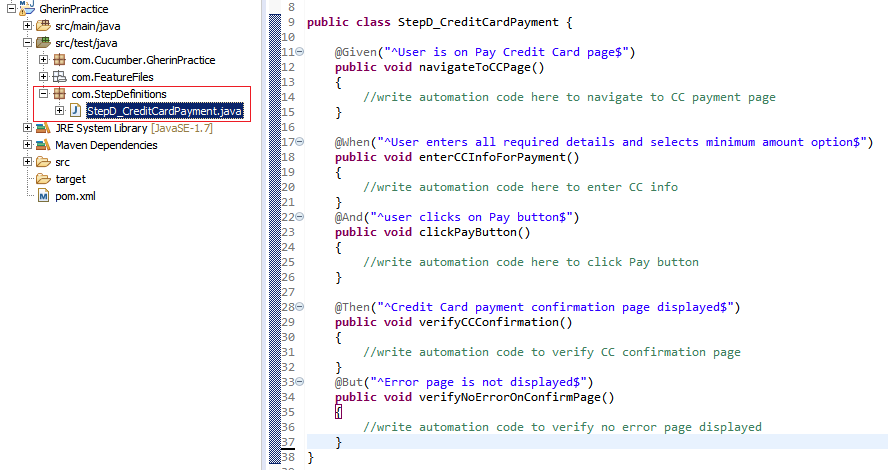
Given <pre-conditions>

When <actions to be performed>

Then <expected output>

* Here, note that “:” is given after ‘Feature’ and ‘Scenario’ only
* 

**Creation of Step Definition File:**

* Create another Package under ‘src/test/java’ folder
* Create a new Package (here I created ‘com.StepDefinitions’)
* Create a .java file and implement Step Definitions there for Feature file
* 
* As an example, Feature line “*Given* User is on Pay Credit Card page” is implemented like

@Given("^User is on Pay Credit Card page$")

**public** **void** navigateToCCPage()

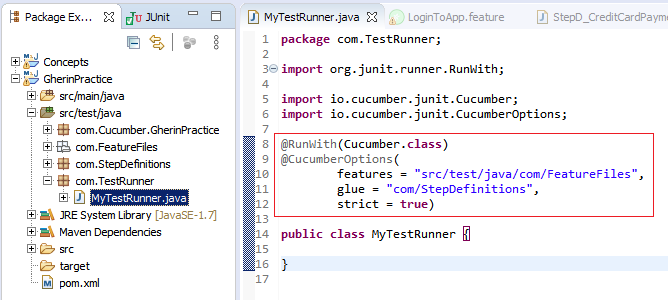
{

//write automation code here to navigate to CC payment page

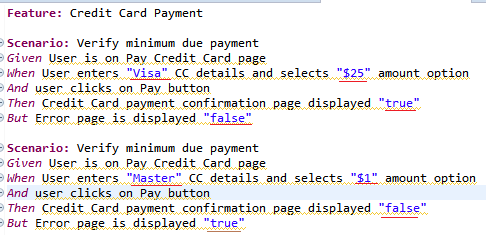
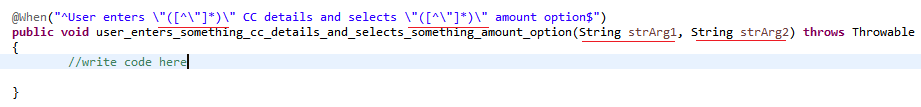
}

* Feature line is written between “^” and “$” symbols
* This line content should be same as it is in feature file
* \*\*StepDefinition file matched with TagName and Descriptions in Feature file\*\*
* \*\*There should be one mapping implementation for each Gherkin line\*\*
* You can have multiple similar tags i.e. multiple @Given, or @Then; however, it get’s executed bases on feature file whatever founds in there, will execute that piece of code in step definition file
* There is another way to create Step Definition File using ‘**Tidy Gherkin’** chrome plugin
* If you have install it as an app in Chrome, the goto <https://apps> in chrome
* Open “Tidy Gherkin” and copy the Feature file content there and select Java tab and you will get Step Definition
* Only the thing that you need to change/modify Package name, class name and method bodies according to your actual package, class names and method bodies
* There is one more way to generate the Step Definition code:
* Let’s consider you just created a Step Definition java file with nothing into the class block.
* And you have created Feature file and Test Runner file
* Then when you run the Test Runner file, your eclipse output console shows you the expected or missing step Definition code below the message “You can implement missing steps with the snippets below:”
* You can copy the code from there and paste it into Step Definition class block.
* Of course, you need to change method bodies according to your need

**Creating Test Runner File:**

* Create another Package
* Create a new Java File
* As a best practice, This test runner’s package and StepDefinition file’s package be placed in same folder; here we have placed both under ‘src/test/java’ folder
* 
* Above highlighted code is must into this Test Runner file to make it a test runner
* Note that, this piece of code is added above Class block.
* Into @CucumberOptions annotation, there are two important elements:
  + “**feature**”
    - Provide the path of Package where .feature file is located
    - Here, you can give that specific .feature file path too
    - If package path provided, meaning all the feature files will be executed under this package
  + “**glue**”:
    - Provide the Package name where StepDefinition file is located
* Now if you simple run this file, ‘StepD\_CreditCardPayment.java’ will be executed.

**Simple Regular Expression:**

* When we have to provide different data/dynamic data for same implementation, we use Regular expression. Let’s see below example
* 
* Here are two scenarios, almost all steps are same except some input values (and expected outputs). Note that, expected steps are slightly modified to avoid redundant step definition code; now we can use same code with these dynamic values
* For that we don’t have to create another Step Definition steps for another scenario
* We can use same step definition for both scenarios
* Here, Cucumber understand that if the text is inside double quotes “”, meaning these are dynamic values
* When we generate step definition in ‘Tidy’ chrome plugin, it automatically generate regular expression at the place of “xxxx” values; see in below screenshot
* That regular expression meaning, code can accept any value
* 

@When("^User enters \"([^\"]\*)\" CC details and selects \"([^\"]\*)\" amount option$")

**public** **void** user\_enters\_something\_cc\_details\_and\_selects\_something\_amount\_option(String strArg1, String strArg2) **throws** Throwable

{

//write code here

}

* Here, first dynamic string i.e. “visa” will go inside String variable ‘strArg1’ and second dynamic value will go inside ‘strArg2’
* You can use these variables into the corresponding method block

**Cucumber-Junit Framework (API Testing):**

To start with this framework development, we should have knowledge of followings:

* Rest Assured API Testing
  + Rest Assured Testing Basics
  + Serialization and De-Serialization using POJO classes
  + Request and Response Spec Builder
* Cucumber Framework
  + Understanding of Cucumber Framework
* Maven
  + Maven – build Management tool basics

Now we will create Cucumber-Junit framework for API Testing.

Create Project and add dependencies:

* Create a Maven Project in Eclipse IDE (with ‘maven-archetype-quickstart’ template)
* Add followings dependencies in Pom.xml
  + **Cucumber-java**
  + **Cucumber-junit**
  + **Junit**
  + **Rest-assured (from io.rest-assured)**
  + **Jackson-databind (along with below two dependencies)**
    - **Jackson-core**
    - **Jackson-annotations**
  + **Gson**
* Note: Jackson-\* jars should have of same version
* Also add below dependencies ( JARs) from Rest-Assured.io
  + **Rest-assured…**
  + **Json-path…**
  + **Xml-path…**
  + **Json-schema-validator…**
  + Spring-moc-mvc…
  + Scala-support…
  + Kotlin-extensions…
  + Spring-web-test-client…
* Make sure following plugins are installed in Eclipse IDE (Help > Eclipse Marketplace)
  + **Natural**
* **Add static Packages** manually in your class(.java) file
  + **import static io.restassured.RestAssured.\*;**
  + **import static org.junit.Assert.\*;**
* NOTE: If added dependencies in pom.xml not getting downloaded or added to your project, then check eclipse menu Project>Build Automatically. This option should be selected.
* If you goto eClipse Project > right click > Properties >Java Build Path > Libraries; and if you see Junit library outside of ‘Maven Dependencies’ folder, then **Delete it**; since we already have added it into Maven Dependencies (pom.xml). Having duplicate Junit entries or multiple junit versions, may create some problem while running the program.
* Also, Goto eclipse menu >Windows >Preferences > Java > Editor > Typing > “Escape Text when pasting into string literal”.. select this option.

This setting will be useful when you paste a JSON file body into string double quote( “”).

**Cucumber BDD API Framework Development:**

To start with this framework development, we should have knowledge of followings:

* Rest Assured API Testing
  + Rest Assured Testing Basics
  + Serialization and De-Serialization using POJO classes
  + Request and Response Spec Builder
* Cucumber Framework
  + Understanding of Cucumber Framework
* Maven
  + Maven – build Management tool basics

Now, we will see step by step Framework implementation:

* Below is the API Contract file for Add, Get, Delete Place API. We are going to create a framework using this contract:

*Google Maps Add API (POST):*

*This API Will add new place into Server*

***Complete URL :***[*https://rahulshettyacademy.com*](https://rahulshettyacademy.com/)***/maps/api/place/add/json?key=******qaclick123***

***Base URL****:*  [*https://rahulshettyacademy.com*](https://rahulshettyacademy.com/)

***Resource****: /maps/api/place/add/json*

***Query Parameters****: key =qaclick123*

***Http Method****: POST*

***Sample Body*** *:*

*{*

*"location": {*

*"lat": -38.383494,*

*"lng": 33.427362*

*},*

*"accuracy": 50,*

*"name": "Frontline house",*

*"phone\_number": "(+91) 983 893 3937",*

*"address": "29, side layout, cohen 09",*

*"types": [*

*"shoe park",*

*"shop"*

*],*

*"website": "http://google.com",*

*"language": "French-IN"*

*}*

***Sample Response***

*{*

*"status": "OK",*

*"place\_id": "928b51f64aed18713b0d164d9be8d67f",*

*"scope": "APP",*

*"reference": "736f3c9bec384af62a184a1936d42bb0736f3c9bec384af62a184a1936d42bb0",*

*"id": "736f3c9bec384af62a184a1936d42bb0"*

*}*

*Google Maps Delete API (POST):*

*This API Will delete existing place from Server*

***Complete URL****: https://rahulshettyacademy.com/maps/api/place/delete/json?key=qaclick123*

***Base URL****: https://rahulshettyacademy.com*

***Resource****: /maps/api/place/delete/json*

***Query Parameters:*** *key*

***Http request*** *: POST*

***Sample Body*** *:*

*{*

*"place\_id":"928b51f64aed18713b0d164d9be8d67f" //(This value comes from Add place response)*

*}*

***Sample Response***

*{*

*"status": "OK"*

*}*

*Google Maps get Place API (GET):*

***This API Will get existing place details from Server***

***Complete URL :*** *http://rahulshettyacademy.com/maps/api/place/get/json?place\_id=xxxx&key=qaclick123*

***Base URL****:*  [*https://rahulshettyacademy.com*](https://rahulshettyacademy.com/)

***Resource****: /maps/api/place/get/json*

***Parameters****: key, place\_id //( place\_id value comes from Add place response)*

***Http request****: GET*

*Note: Key value is hardcoded and it is always qaclick123*

***Sample Response for the Provided Place\_Id***

*{*

*"location":{*

*"lat" : -38.383494,*

*"lng" : 33.427362*

*},*

*"accuracy":50,*

*"name":"Frontline house",*

*"phone\_number":"(+91) 983 893 3937",*

*"address" : "29, side layout, cohen 09",*

*"types": ["shoe park","shop"],*

*"website" : "http://google.com",*

*"language" : "French-IN"*

*}*

*Google Maps Put Place API (PUT):*

*This API Will update existing place in Server with new values*

***Complete URL :*** *http://rahulshettyacademy.com/maps/api/place/get/json?place\_id=xxxx&key=qaclick123*

***Base URL*** *:*  [*https://rahulshettyacademy.com*](https://rahulshettyacademy.com/)

***Resource****: /maps/api/place/update/json*

***Query Parameters****: key, place\_id //(place\_id value comes from Add place response)*

***Http Method****: PUT*

*Note: Key value is hardcoded and it is always qaclick123*

***Sample Request:***

*{*

*"place\_id":"8d2573bdf6ceec0e474c5f388fa917fb",*

*"address":"70 winter walk, USA",*

*"key":"qaclick123"*

*}*

***Sample Response for the Provided Place\_Id***

*{*

*"location":{*

*"lat" : -38.383494,*

*"lng" : 33.427362*

*},*

*"accuracy":50,*

*"name":"Frontline house",*

*"phone\_number":"(+91) 983 893 3937",*

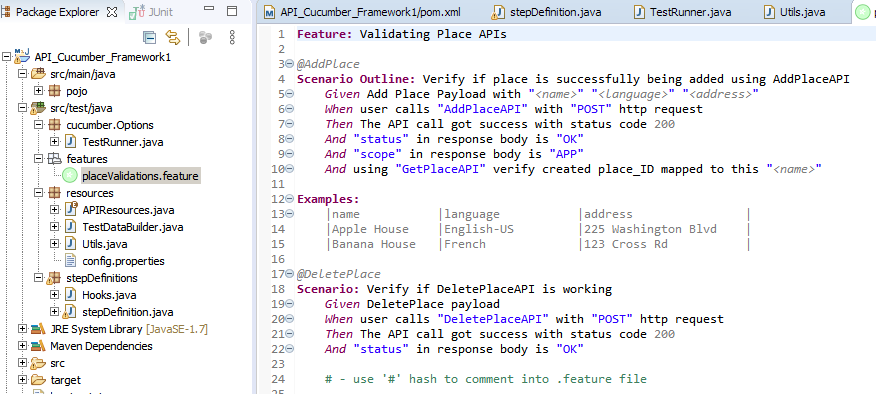
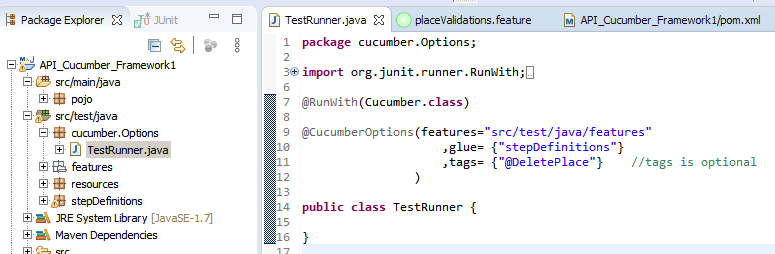
*"address" : "29, side layout, cohen 09",*

*"types": ["shoe park","shop"],*

*"website" : "http://google.com",*

*"language" : "French-IN"*

*}*

* Created Maven-Java project in eclipse as mentioned above
* Added all dependencies in pom.xml files (dependencies jars are mentioned above)
* Created ‘features’ package in src/test/java
* Created .feature file into this package to add the PLACE API scenarios
* 
* 2 scenarios are added here; AddPlace and DeletePlace
* **Dynamic values** are mentioned into double quotes **“”**.
* Tried to keep same statements which are common in both scenarios (except dynamic values)
* here, 'When', 'Then', 'And' statement is exactly same in 2nd scenario as first scenario (except parameters)
* so, when you run the TestRunner, StepDefinition will be constructed with two Given methods and one When, Then, And methods in console; which need to be added into stepDefinition
* In first scenario, we are going to test it for **multiple input data sets**; so data set has been created under **“Examples:”** section. First row is the column name or header. 2nd and 3rd row indicates the input rows or input date sets; each column is separated with pipe **“|”**
* And corresponding column names are mentioned into the scenarios with the **“<xxxxx>”** format.
* Meaning, when AddPlace scenario gets executed, it will be executed twice; each for added test data sets.
* We have given the **tags** to each scenario with **@ sign**; e.g @AddPlace, @DeletePlace
* These tags can be used to execute specific scenarios; we have to mention the tag name in TestRunner class if specific scenarios have to be executed.
* In this way, you can create Regression, smoke test suite by giving @Regression tag (as an e.g.)
* Tagging is Optional.
* We can give **multiple Tags** for same scenarios e.g. like “@AddPlace @Regression @Smoke”
* Note that, **‘#’** can be used to **comment out** a statement into Feature file
* Now, Create ‘**StepDefintions’** Package under same folder src/test/java
* Create ‘StepDefintion’ class under it
* Based on Feature file, create skeleton for StepDefintion class methods (details have been given above on how to create stepdefinition skeleton)
* OR just create StepDefinition class and execute TestRunner class, java console will give you stepdefinition skeleton.
* Now, create ‘**cucumber.Options’** class in src/test/java folder
* Create ‘**TestRunner’** class into it
* 
* It is a bridge between Feature file and the StepDefintion and we run the test using this TestRunner (right click on TestRunner class and run as JUnit test; your test scenarios will get executed)
* Provide feature file path into ‘**features’** parameter. If there are multiple feature files and you want to execute all, just provide the path to those feature files folder (same as shown in above code). If you want to execute only specific feature file, then given the feature file path e.g. “src/test/java/features/placeValidations.feature”
* Provide “stepDefinitions” package name into the “**glue**” parameter. You can give the path as well.
* Here ‘**tags’** is optional. It is used to mention which test scenarios have to be executed; that should be same tags given in .Feature files
* In this class, you just need to add **@RunWith(Cucumber.class)** and **@CucumberOptions()** tags; don’t need to add anything into TestRunner class body.
* Now, we will create POJO classes for the API body;
* Here, we have to provide multiple attributes in AddPlace API (see contract)

**Sample Body** :

{

"location": {

"lat": -38.383494,

"lng": 33.427362

},

"accuracy": 50,

"name": "Frontline house",

"phone\_number": "(+91) 983 893 3937",

"address": "29, side layout, cohen 09",

"types": [

"shoe park",

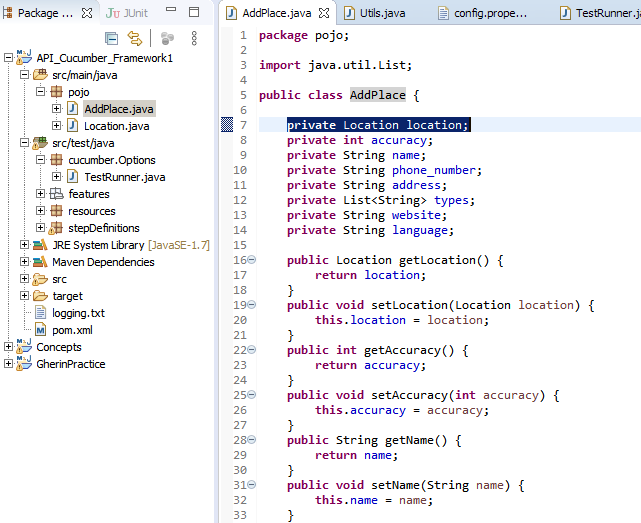
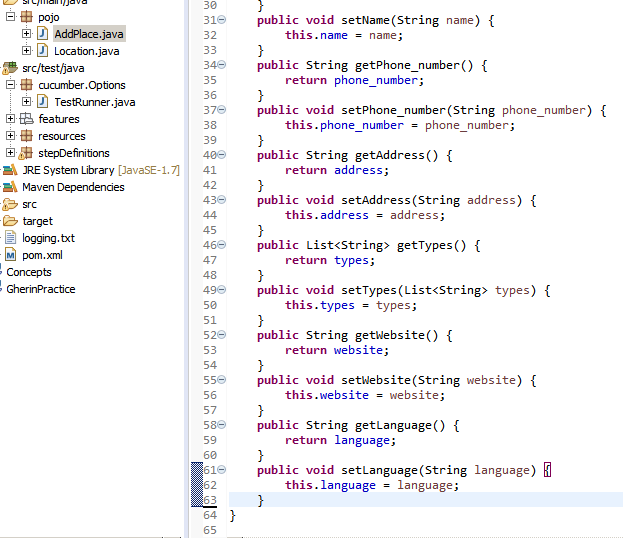
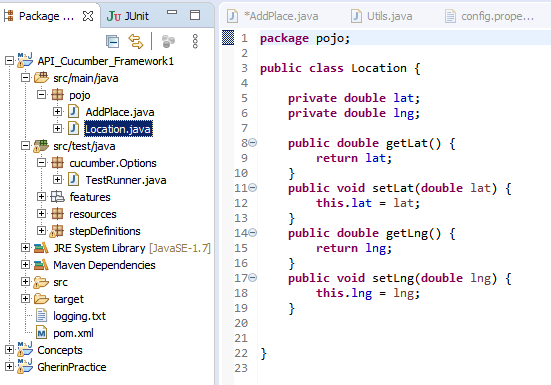
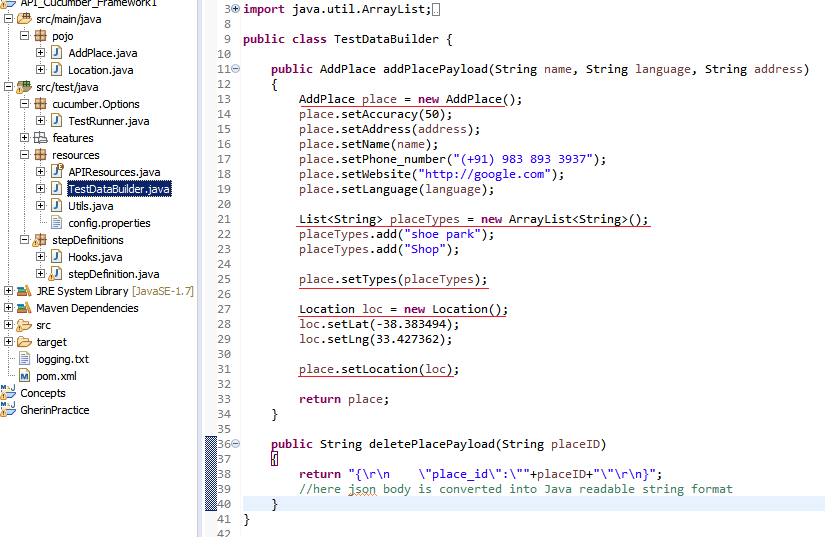
"shop"

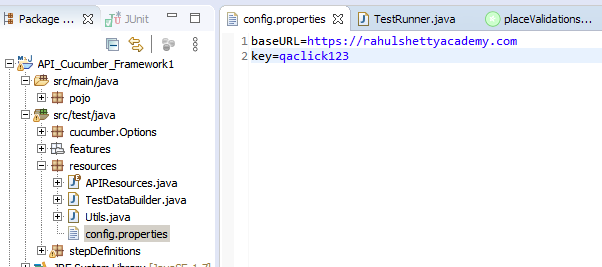
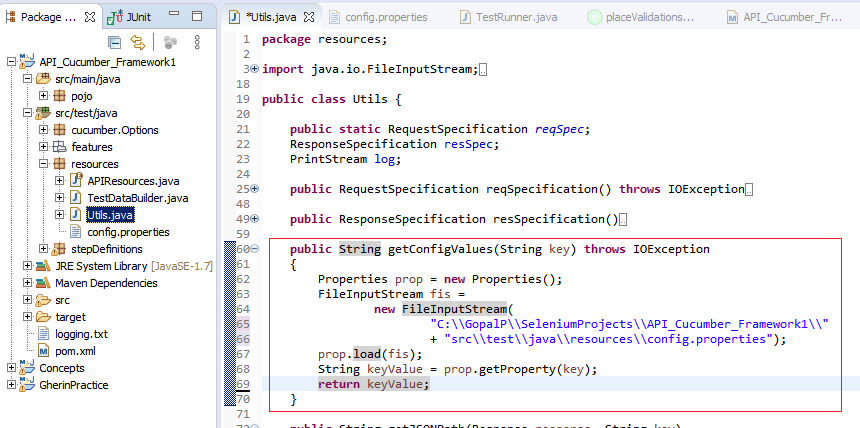
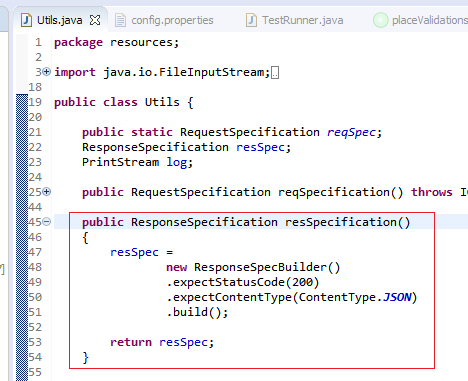
],

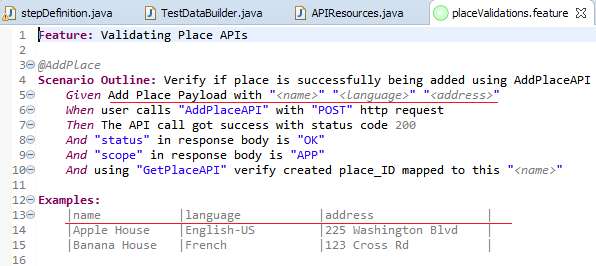
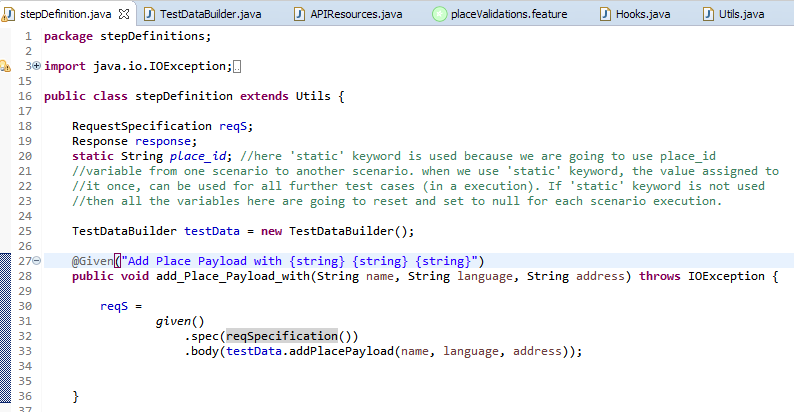
"website": "http://google.com",

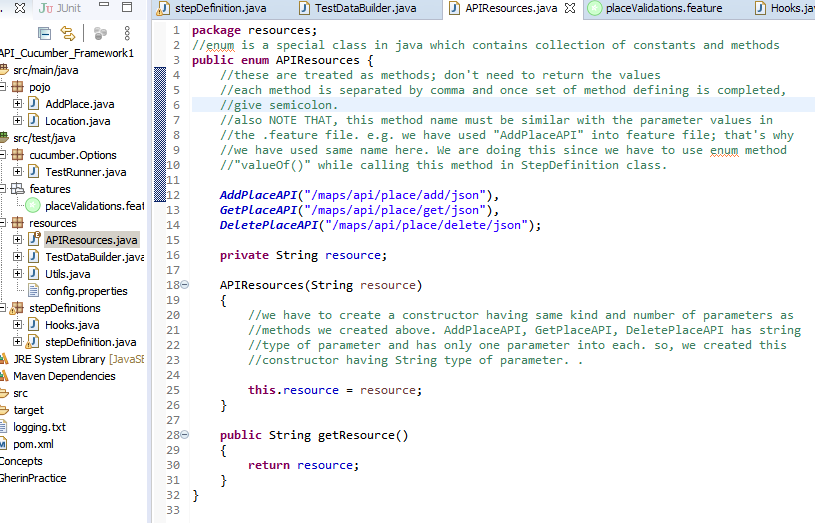
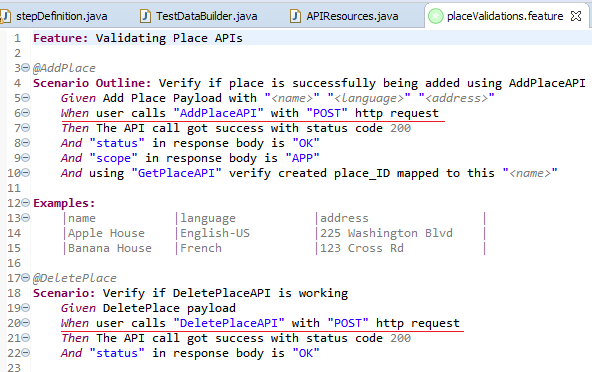
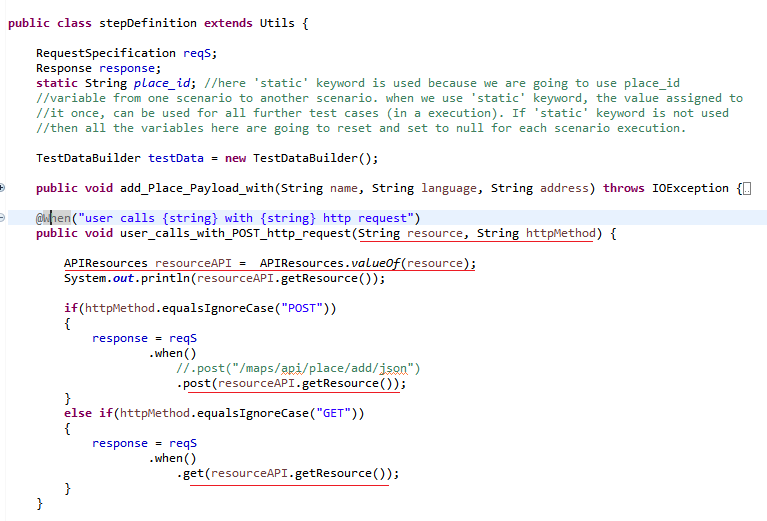
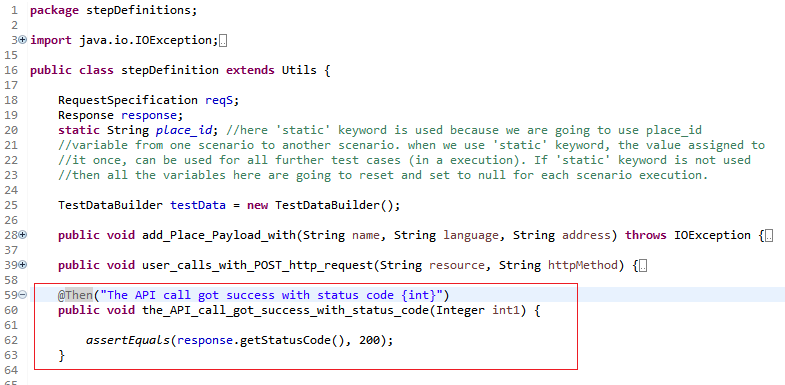
"language": "French-IN"

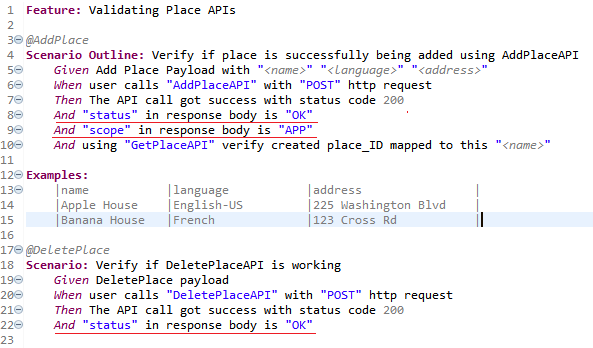
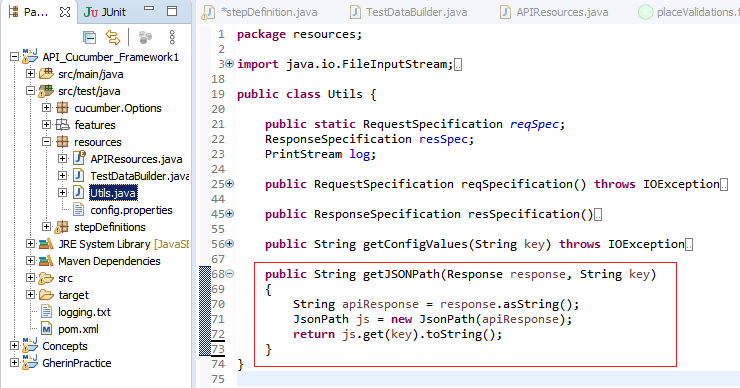
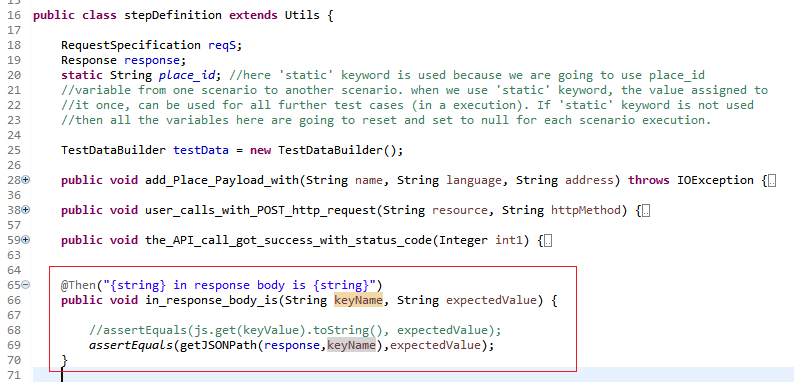
}

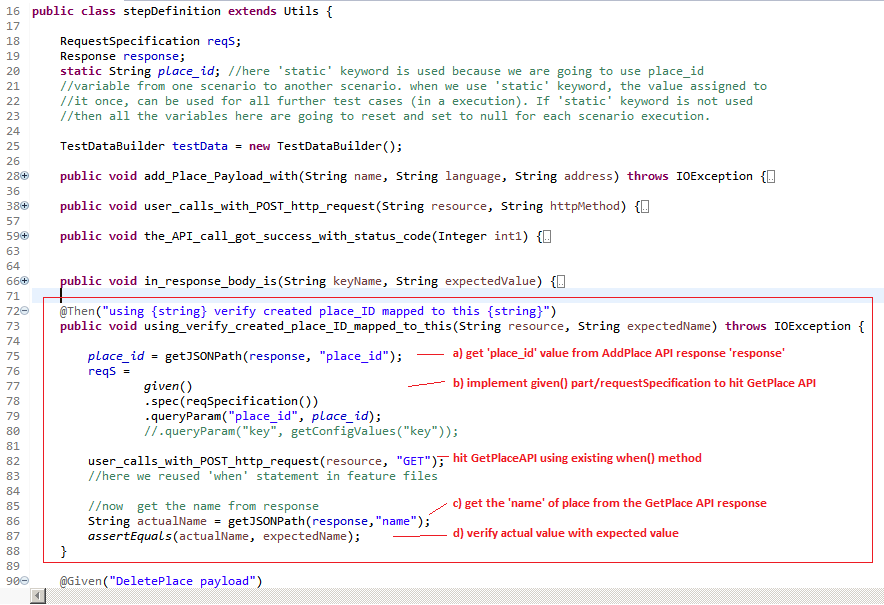
* Create **POJO classes** for it.
* Create POJO package in src/**main**/java folder.
* Create ‘AddPlace’ class and declare the variables for each attributes mentioned into the above sample body
* ‘Location’ is sub json which contains few more attributes; we have to create two classes; one for main AddPlace json body and another for “Location” sub-jason file.
* Also, create corresponding **Getter/Setter methods**
* 
* 
* 
* POJO classes are implemented; these are without data.
* Now, we have to send the test data to POJO classes (from the test scenarios constructed in feature file)
* For that, construct ‘**TestDataBuilder’** class which will construct AddPlace API body and DeletePlace API body.
* Create this class in ‘resource’ package
* Create methods into it which will construct AddPlace API body and DeletePlace API body.
* Create a object of AddPlace pojo class, assign attribute values to the object and return that object wherever required. We will use that object in StepDefinition When() method
* 
* Here, we are providing name, language, address as input variables to ‘addPlacePayload’ method because that’s coming from test scenarios.
* Other values are hard coded here (assuming those are unchangeable); otherwise we can bring them to from Test scenarios or config.properties file.
* Also, we can create POJO class for DeletePlace payload, but it just required a simple json having one attribute, place\_ID; so here we directly return the json body into ‘deletePlacePayload()’ instead of creating Pojo class for it.

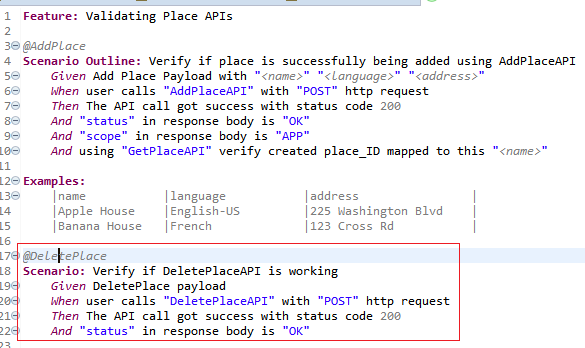
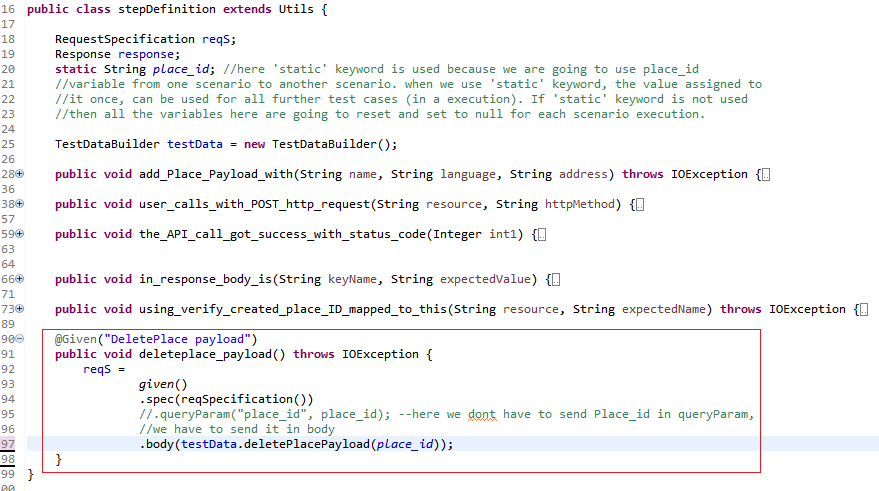
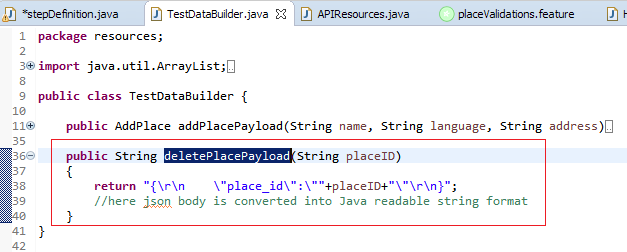
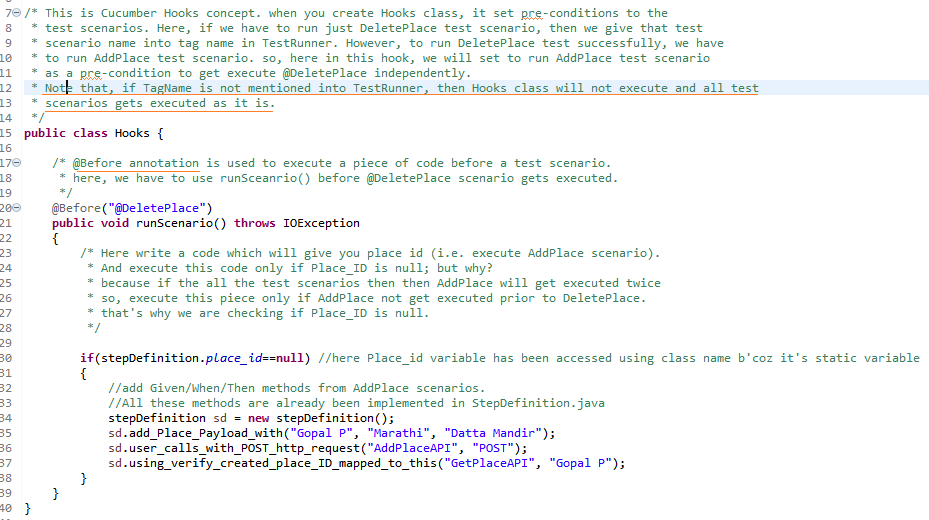
* Now, we have to implement common part to all test scenarios e.g. baseURI, queryParameters etc.
* For that Create ‘**Resources’** package and create ‘**Utils’** class into it
* Also, create ‘**Config.properties’** file with key/value
* 
* Implement a method into ‘Utils’ class which will get the value from Config.properties file
* 
* Now implement methods for common part of request and response using ‘**RequestSpecification’**, ‘**ResponseSpecification’** inbuilt interfaces.
* 
* Here we create ‘reqSpec’ object of ‘RequestSpecification’ interface and set all required common specification to it using **‘RequestSpecBuilder()’**; here baseURL , query parameter and ContentType is set.
* We added baseURI and query parameter using ‘getConfigValues()’ method which we saw in above section.
* We declared RequestSpecification reqSpec as **‘public Static’** because we have to set only once when test gets started to execute; we don’t need to get it assign it again and again.
* If we don’t declare it as ‘static’, ‘reqSpec’ will be reset and for each test case.
* Into the same code, we are creating log file to log the request/response details
* When test case gets executed, ‘logging.txt’ file will be created under project folder.
* **‘.addFilter()’** is used to log request/response details into log file
* Here, we have checked ‘if(reqSpec==null)’ to avoid the log file getting created fresh for each test scenario; it should maintain the logs for all test scenarios in a execution cycle.
* Similarly, add **‘ResponseSpecification()’** method into this ‘Utils’ class
* 

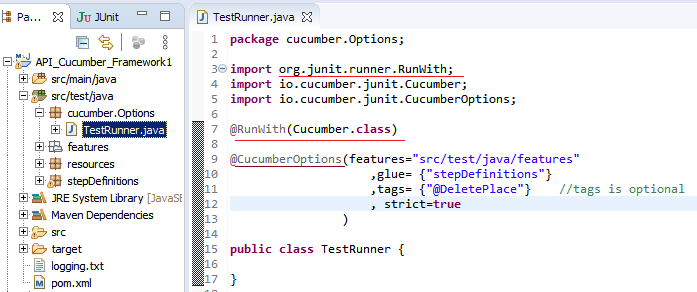

* Now construct AddPlaceAPI - Given() into StepDefinition.
* You know that what goes into Given() i.e. API attributes like BaseURI, Query paramenters, ContentType etc. and API body.
* 
* 
* We already created a method of all request specification in Utils class; so we have call that corresponding method into **given().spec()**
* We already created a method which constructs AddPlaceAPI body into TestDataBuilder class; so just call that method into **given().body()** and provide the dynamic values from the test scenarios (given() statement)


* Now, we will construct when() part of AddPlace API
* You know that, resource url goes into when().
* If there are hundreds of APIs we have to test having hundreds of resource url, then it’s difficult to manage and hardcode all these resource urls into your stepDefinition code.
* That’s why we will define separate block which stores all resource urls.
* That block is a special one called ‘enum’ class which contains collection of constants and methods;
* Here we have added methods for each resource url; each for AddPlace, GetPlace, DeletePlace.
* READ the comments into the below enum class snap; just to understand what that means
* 
* Now construct when() method into StepDefinition class based on the feature file.
* 
* Below constructed when() will work for both AddPlace and DeletePlace ‘when’ statement.
* 
* Here are storing the API response into ‘response’ variable of ‘Response’ type
* We will use this ‘response’ to check expected values into it.
* Now, construct Then() method to verify response code if it’s 200.
* 
* .getStatusCode() is built-in method for Java ‘Response’ class when gets the response code from the API response.
* Here, we may use ‘int1’ variable instead of hardcoded 200 in AssertEquals().

* Now, implement another Then() method in StepDefinition
* This Then() method will execute three statements in feature files
* 
* Here, we have to read the ‘status’, ‘scope’ attributes and their values from API Response and have to verify the values are expected in there, or not.
* So we have to implement a common method which will read the response and get the asked attribute and it’s value.
* This common method ‘getJSONPath()’ is constructed into ‘Utils’ class as below:
* 
* As you know we have stored API response in a ‘response’ variable in StepDefinition class
* We will get the keyName and the expected value from feature file
* So, we just have to pass the response and keyName to getJSONPath() which will process the json response and get the asked key value; as below:
* 

* Now, we will implement another Then() method in stepDefinition which will hit GetPlace API using the place\_ID (which we get in the response of AddPlace API) in query paramter, get the the PlaceName and verify ‘PlaceName’ against the one which we posted in AddPlace API.
* Here, we have to perform four different steps: a) get the Place\_ID from the AddPlace API response, b) hit the GetPlace API, c) fetch the PlaceName from GetPlace API response and d) verify PlaceName from GetPlaceAPI response against the one which you had sent in AddPlace API.
* 
* Below is the existing when method in StepDefinition itself
* 

* Now, implement DeletePlace API i.e. second scenario in feature file.
* If you observe, we already implemented when(), then() methods and these can be re-used for this scenario. So now we just have to implement given() method.
* 
* See below implementation:
* 
* We already implemented ‘deletePlacePayload()’ which will construct payload json. This method can be passed into .body().
* 
* Now, when you run the TestRunner class, AddPlace and DeletePlace scenario can be executed
* Execution can be done step by step based on .feature files
* If ‘Tags’ are mentioned into ‘TestRunner’ class, then only corresponding scenarios will be executed.
* However, if you mentioned into tags that ‘@DeletePlace’ then only DeletePlace scenario will be executed; But it will be failed; because it required to execute Add PlaceAPI, GetPlaceAPI. i.e. AddPlace scenario is the pre-requisite for DeletePlace scenario.
* Then how to handle this pre-requisites part.
* It would be handled with ‘**Hooks’** concept.
* Create ‘Hooks’ class as below in ‘StepDefinitions’ package.
* 

* You can run the TestRunner.java from command prompt too using mvn command.
* C:/…java project folder…. > **mvn test**
* It will compile the project and run it.
* Maven will look for runner class into src/test/java folder to execute it.
* 
* Only those test scenarios will get executed which have tags similar given in TestRunner class.
* Instead of giving the Tag Name in TestRunner class, we can provide it thru command prompt while executing it as:
* C:/…java project folder…. > **mvn test -Dcucumber.options=“- -tags @DeletePlace”**
* Here –D stands for parameters to be provided to maven command and we are providing this parameter: **cucumber.options=“- -tags @DeletePlace”**

* Now, let’s see how to add **HTML Reporting** to this framework.
* Get the below code from github <https://github.com/damianszczepanik/maven-cucumber-reporting> and **paste it** **in POM.xml file under <Build><plugins> …. </plugins></Build> section**.

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<configuration>

<testFailureIgnore>true</testFailureIgnore>

</configuration>

</plugin>

<plugin>

<groupId>net.masterthought</groupId>

<artifactId>maven-cucumber-reporting</artifactId>

<version>(check version above)</version>

<executions>

<execution>

<id>execution</id>

<phase>verify</phase>

<goals>

<goal>generate</goal>

</goals>

<configuration>

<projectName>cucumber-jvm-example</projectName>

<!-- optional, per documentation set this to "true" to bypass generation of Cucumber Reports entirely, defaults to false if not specified -->

<skip>false</skip>

<!-- output directory for the generated report -->

<outputDirectory>${project.build.directory}</outputDirectory>

<!-- optional, defaults to outputDirectory if not specified -->

<inputDirectory>${project.build.directory}/jsonReports</inputDirectory>

<jsonFiles>

<!-- supports wildcard or name pattern -->

<param>\*\*/\*.json</param>

</jsonFiles>

<!-- optional, defaults to outputDirectory if not specified -->

<classificationDirectory>${project.build.directory}/classifications</classificationDirectory>

<classificationFiles>

<!-- supports wildcard or name pattern -->

<param>sample.properties</param>

<param>other.properties</param>

</classificationFiles>

<parallelTesting>false</parallelTesting>

<!-- optional, set true to group features by its Ids -->

<mergeFeaturesById>false</mergeFeaturesById>

<!-- optional, set true to get a final report with latest results of the same test from different test runs -->

<mergeFeaturesWithRetest>false</mergeFeaturesWithRetest>

<!-- optional, set true to fail build on test failures -->

<checkBuildResult>false</checkBuildResult>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

* Here, we have to make a little change; version is not mentioned into ready made above code. It just mentioned as <version>(check version above)</version>
* Enter actual version here; you will get it on the same page from where you are copying this code. e.g. <version>**5.3.0**<version>
* Now try to understand how this code works

<execution>

<id>execution</id>

<phase>verify</phase>

<goals>

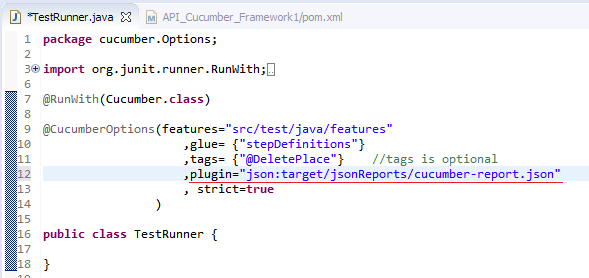
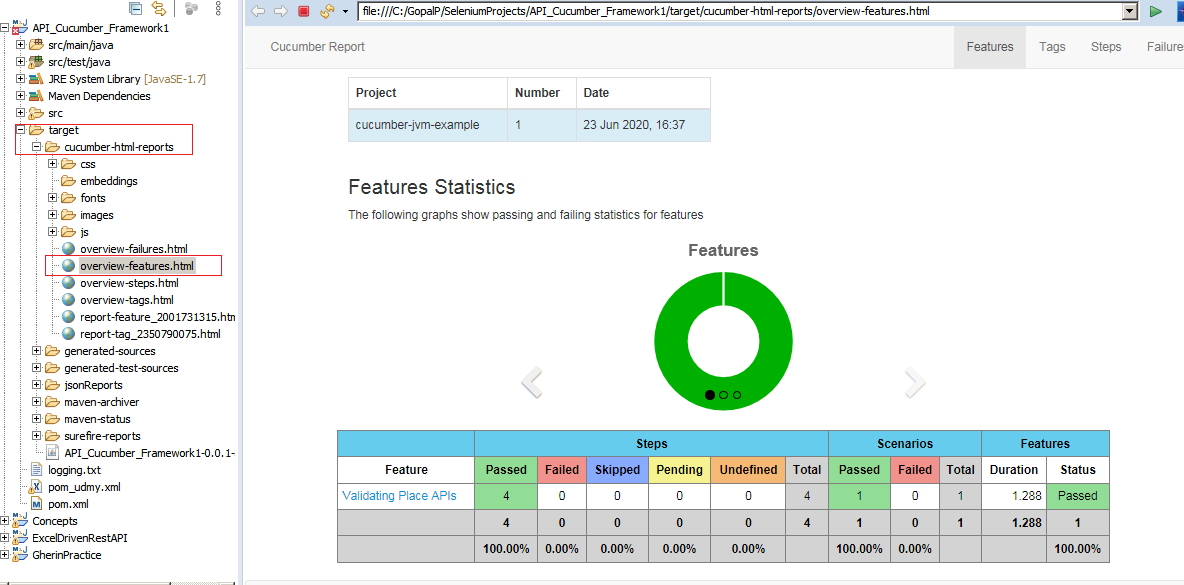
<goal>generate</goal>

</goals>

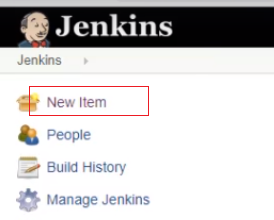
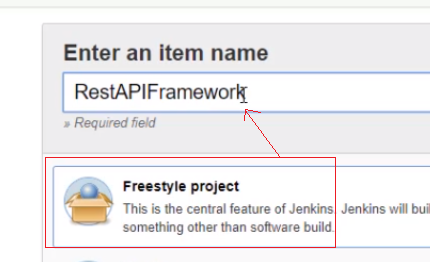
* Meaning of this part is that the report will be generated in **‘Verify’ phase** of the maven. Maven has multiple phases like Compile, Test, Verify
* i.e. in ‘compile’ project will be compiled, in ‘test’ project will be compiled and get execute and in ‘verify’ report will be generated.

<!-- output directory for the generated report --> <outputDirectory>${project.build.directory}</outputDirectory>

<!-- optional, defaults to outputDirectory if not specified --> <inputDirectory>${project.build.directory}/jsonReports</inputDirectory>

* this part tells about expected input directory and output directory.
* This code will require a **report (as input) in JSON format** which is generated after execution is done. It will try to get from project directory/jsonReports folder (${project.build.directory}/jsonReports)
* After getting the report in JSON format, it will convert it into HTML report and store in output folder at project level (${project.build.directory})
* However, in above framework, we did not generate JSON report;
* Now, we will generate it.
* For that, add a small piece in TestRunner file as below
* 
* This will generate a report in JSON format and stored it into ‘project level folder/target/jsonReports/cucumber-report.json’
* Now when you run the project using mvn cmd as c:\...project folder… > **mvn test verify**
* Mvn will execute ‘test’ phase and ‘verify’ phase.
* After completing ‘test’ phase, ‘cucumber-report.json’ will be generated and saved into /target/jsonReports’ folder
* And then in ‘verify’ phase, this report will be taken as input and above **Plug-in** (which we pasted in pom.xml) will convert it into **HTML report format** and saved into Project Folder/Target/cucumber-html-reports
* 
* You can see multiple statistics within same report; when you click feature links, ‘tags’, ‘steps’ etc.
* Also, instead of giving the ‘Tag’ in TestRunner file, you can also provide the tag from MVN command as below:
* **mvn test verify –Dcucumber.options=“- -tags @AddPlace”**

**Run your Framework from Jenkins (CI tool):**

* First we will set up Jenkins on the machine
* Download ‘Generic Java Package (.war)’ from Jenkins download page
* Keep the downloaded war (jenkins.war) file in your folder
* We have to start the Jenkins on server (here, your local machine)
* For that, Goto Command Prompt, goto the folder where ‘jenkins.war’ file is present
* Hit “C:\...folderpath....\> **java -jar Jenkins.war -httpPort 9090**” command.
* You can give any port number like 8080, 9090 etc.
* NOTE: you will get Jenkins installation steps in google.
* Let’s assume your Jenkins service is up and running and you logged in to Jenkins server on <http://localhost:9090>
* Create New Item
* 
* Select FreeStyle Project and give the Project Name. Click Ok.
* 
* Project will be created and you will be in the Project page ‘General’ section.
* Click on ‘Advanced’ in the first section
* Select ‘Use Custom Workspace’ option and give your Java Project folder path from your local machine.
* Note that, since I have created this Java project in local machine, we are selecting ‘Use Custom Workspace’ option. Otherwise, we can select ‘GIT’ option in ‘Source Code Management section’ and provide Git url to the project from there.
* Now, Goto ‘Build’ section. Click on ‘Add Build Step’
* Select ‘Invoke top-level Maven Targets’
* Into the ‘Goals’, provide the maven command which we used above to run the project e.g.

**mvn test verify –Dcucumber.options=“- -tags @AddPlace”**

**mvn test**

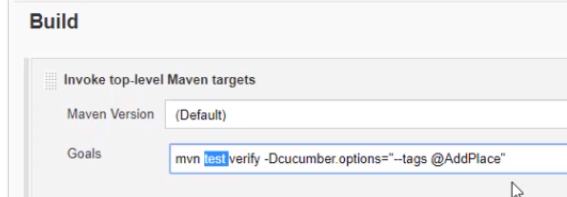
**mvn test verify**

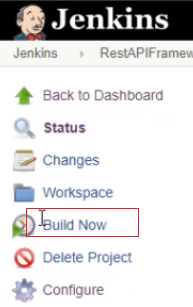
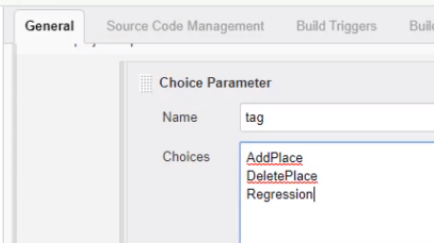
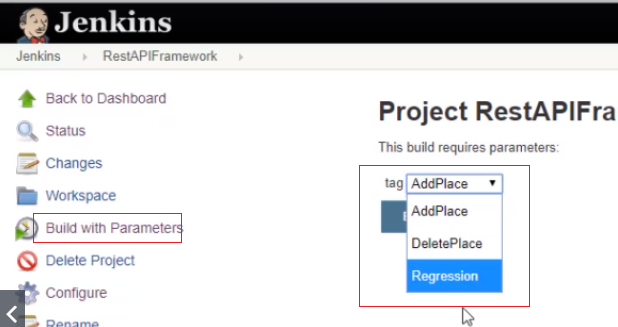
* NOTE that, while giving the command in ‘Goals’, we don’t need to given ‘mvn’ keyword since Jenkins will take care of it. So, we have to give command in ‘Goals’ as:

**test verify –Dcucumber.options=“- -tags @AddPlace”**

**test**

**test verify**

****

* Now, goto main page again and click on ‘Build Now’
* 
* When build is completed, Click on Build# and then click in ‘Console Output’ where you can see report (the same which we see in eclipse console when running the java project)
* Here, we have hardcoded tag ‘@AddPlace’ in above command.
* Instead of that, we can **parameterize** it in Jenkins
* For that, click on ‘Configure’ on Project page to navigate back to config page
* Select ‘**This Project is Parameterized’** in General section
* Click on ‘Add Parameter’ > ‘Choice Parameter’
* Give Name e.g. ‘Tag’
* Enter all ‘tags’ which you can just select before executing the framework from jenkins
* 
* Into ‘Goals’ text box modify the command as below, :
* **test verify –Dcucumber.options=“- -tags @“$tag””**
* here, we provide parameter name ‘tag’ with $ sign.
* When we provide $ sign in Jenkins, then it assumed as a variable.
* Save the changes
* Navigate back to project page
* You can see ‘**Build with Parameter’** link instead of ‘Build Now’
* Click ‘Build with Parameter’
* It will ask to select the provided parameters
* 
* Select the parameter value
* Selected value will be replaced ‘$tag’ in the given command and the command will be executed.
* After running the build, you can click ‘Workspace’ link on Project page where you can see folder structure similar the project folder you see in eclipse editor
* click on ‘Target’ folder, you can see the cucumber HTML report (the same which we see in above framework example)

**GIT:**

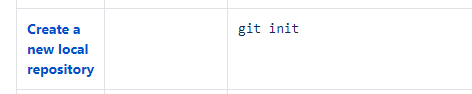
* **GIT** is an open source distributed Version Control System, designed to handle all types of projects with efficient way
* ‘SVN’ is similar tool like GIT.
* **GITHUB** is a central place where you can upload your work/projects/documents and another person can download the latest version from that central place to work upon.
* To upload or download or to communicate with GITHUB; you have to run **GIT commands;** GIT commands can be used once you install GIT application on your local machine i.e. using GIT application, we run GIT commands to communicate with GitHub.
* Now let’s see how GIT and GitHub works.
* Download GIT from Git’s website (<https://git-scm.com/downloads>) and install it on your local machine
* Create an account on GITHUB website
* Login to GitHub and create a “Repository” there
* Now, lets assume that you have to upload the projects/Docs on GitHub then Goto Command prompt
* Note that, you can read basic Git Commands here (<https://confluence.atlassian.com/bitbucketserver/basic-git-commands-776639767.html>)
* From the command prompt, you have to tell Git very first, who you are
* For that, hit following commands
* 
* E.g. C:/>…you project path…/> **git config - -global user.name “suresh p”**
* C:/> …you project path…/> **git config - -global user.email “**[**patilsureshc54@gmail.com**](mailto:patilsureshc54@gmail.com)**”**
* Note, here enter the email id which is used while creating an account for GITHUB.
* Now goto you project folder on command prompt; let say your project folder location is C:\GopalP\SeleniumProjects\API\_Cucumber\_Framework1

Then goto

C:/> GopalP\SeleniumProjects\API\_Cucumber\_Framework1 >

Hit below command, which will create .git file into that folder and initialize communication point with GitHub.

C:/> GopalP\SeleniumProjects\API\_Cucumber\_Framework1 > **git init**

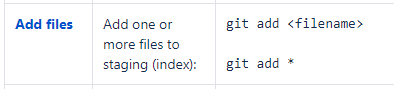
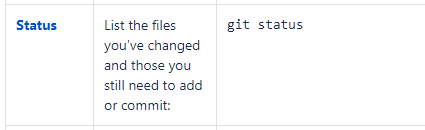
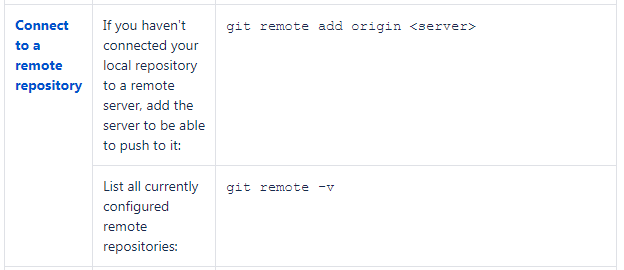
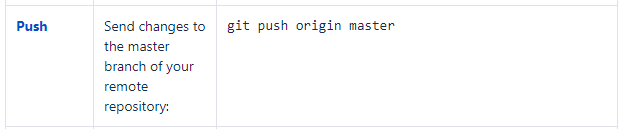
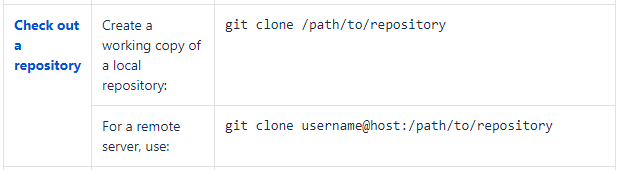
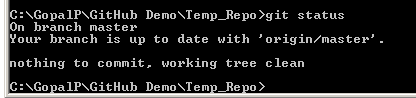
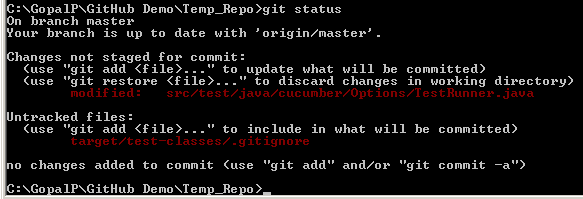
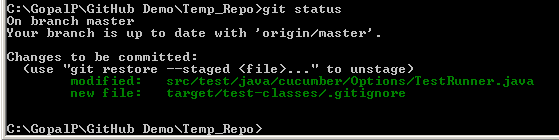
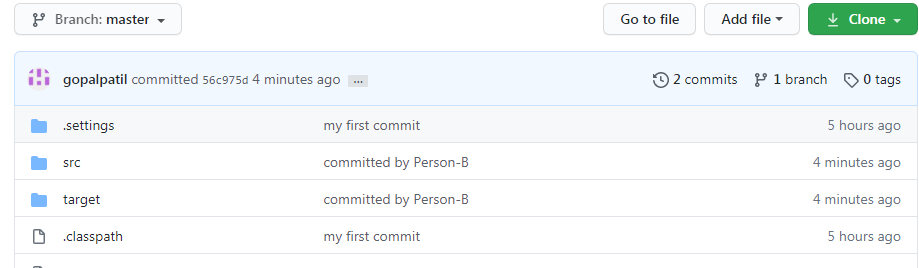
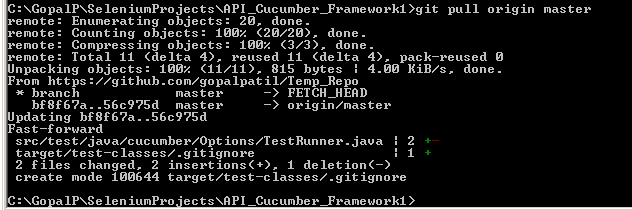
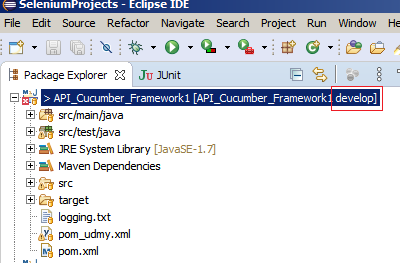
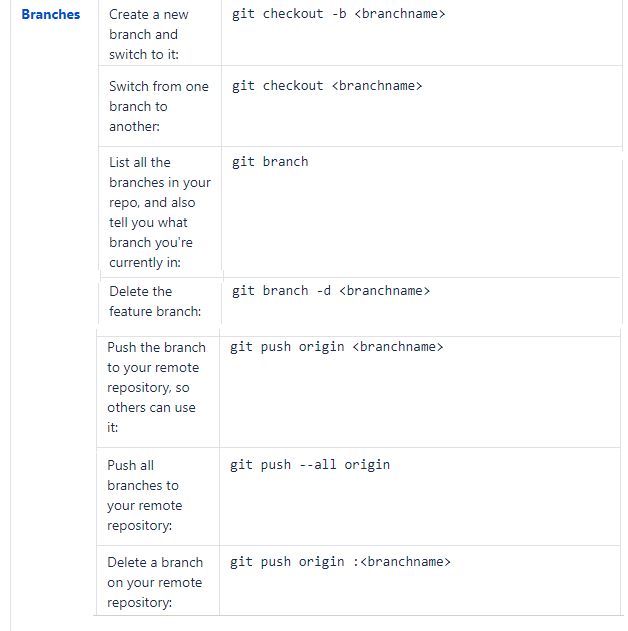
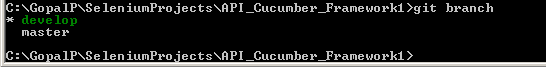


* Now, you have to upload the files into GitHub Repo
* For that there is an order to do that

Add the required files in Stash (Stash is temp location of Git)

Mark the files (which are in Stash) as Commit

Push the committed files to GitHub

* Run below command to add files into Stash
* \* is used to add all files from the given location; you can give specific file **path+name** too
* C:/> …\API\_Cucumber\_Framework1 > **git add \***
* This is also called as ‘**Staging’**
* 
* You can run below command to check the Status of the files (which need to be committed)
* C:/> …\API\_Cucumber\_Framework1 > **git status**
* 
* Now, Commit the files for upload using below command
* Here, **-m** stands for add message for this commit
* C:/> …\API\_Cucumber\_Framework1 > **git commit -m “my first commit”**
* ****
* Now it’s ready the files to get Push into GitHub Repository
* Till the time, all the operations done/affected in local machine
* Before push, you have to tell your GitHub Repository location to get push the files
* C:/> …\API\_Cucumber\_Framework1 > **git remote add origin** [**https://github.com/gopalpatil/Temp\_Repo.git**](https://github.com/gopalpatil/Temp_Repo.git)
* (Note that, instead of ‘github.com’ in url, there would be your company’s domain where company’s github account is set.)
* This repository location can be get from GitHub Repository page.
* This command will connect your local location to GitHub Repository
* 
* Now push the files using below command
* C:/> …\API\_Cucumber\_Framework1 > **git push origin master**
* It will ask you to enter your GitHub Login credentials
* Enter it
* Your files will be pushed to GitHub Repo.
* Here, ‘master’ in above command is nothing but the ‘Master’ branch; meaning push the files into Master branch.
* 
* Similarly, if some other person-B stars working on the same project; he has to download the project from GitHub on his machine.
* He will also setup Git on his machine and run the Git command to communicate with GitHub Repo
* If the other person-B is getting the project **Very FIRST Time** from GitHub, then he will use below command.
* C:/> …Other Persons Folder Path ..> **git clone** [**https://github.com/gopalpatil/Temp\_Repo.git**](https://github.com/gopalpatil/Temp_Repo.git)
* While doing this, Git may ask you GitHub login credentials
* Note that **‘clone’ command** is being used Very First time when you wanted to get complete project from GitHub; otherwise or from next time, you have to use ‘**Pull’ command.**
* url in the above ‘clone’ command, can get from GitHub Repository > Clone/download button.
* 
* Repository will be downloaded in the folder where you have run above command
* Now Person-B is ready to work on the code in his local machine.
* When Person-B is run below command and there is nothing changed in any of the file
* C:/> … his local project folder… > **git status**
* Then he will get this response
* 
* If person-B changes something in the files in his local machine and run the ‘git status’ command, he will get below response
* 
* This shows modified files list (and not matching with GitHub Repo)
* If person-B wanted to upload the changes, then has to run “git add \*” command
* C:/> … his local project folder… > **git add \***
* After this if user hit C:/> … his local project folder… > **git status**
* He can see below result
* 
* Now person-B has to commit the files as
* C:/> … his local project folder… > **git commit –m “committed by Person-B”**
* Now he has to push the code change in Master branch using below command
* C:/> … his local project folder… > **git push origin master**
* You can see the changes in GitHub Repo with the comments
* 
* Now, Person-A wanted to get the latest changes from GitHub, then he will use the ‘Pull’ command
* C:/> … A’s local project folder… > **git pull origin master**
* It will get the latest changes from the GitHub and download into the Local Repo
* Person-A can see the changes into his local code. Only the changed files will get pulled from GitHub.
* 
* Here we saw, push and pull commands.
* Now let’s see ‘**Branch’**
* In above examples, we worked on ‘master’ branch i.e. we push files to ‘master’ branch and pull files from ‘master’ branch.
* ‘Master’ branch is main branch where working code/files generally kept.
* If developer wanted to work on another sprint/release then they create an another branch; generally it called a ‘develop’ branch. Developers works on that branch, perform pull/push operations until final stage.
* When everything is finalized and developed then that working code from ‘develop’ branch will be merged into Master branch.
* To create a new Branch, run below command
* C:/> … A’s local project folder… > **git checkout –b develop**
* Here ‘develop’ is branch name (you can give any name)
* This command will create new Branch viz ‘develop’ and switch you into that branch
* When you go to Eclipse and see the project, you can see the java code is currently pointing to ‘develop’ branch
* 
* 
* When you run ‘git branch’ command, it will list all the branches and your current working branch.
* 
* It shows two branches here and ‘develop’ is current working branch for me.
* So that, now person-A is working in ‘develop’ branch; so whatever changes he will make, will be reflected into that branch. Master branch will be untouched.
* As usual, person-A will add the updated code into Staging, then do commit on the file and then push to the GitHub.
* C:/> … A’s local project folder… > **git add \***
* C:/> … A’s local project folder… > **git commit –m “Release 1.1 change by person-A”**
* C:/> … A’s local project folder… > **git push origin develop**
* Note that, all these commands will be executed in ‘develop’ branch since we already switched into that branch using ‘**git checkout –b develop’** command
* Once person-A pushed the develop branch, it will be reflected into GitHub. Now you could see two branched in there i.e. Master and Develop.
* Now, Person-B wants to work on the develop branch; he will pull the code from GitHub Develop branch, then switch himself into that branch, then will work on local code then push it again into develop branch.
* C:/> … B’s local project folder… > **git pull origin develop**
* C:/> … B’s local project folder… > **git checkout develop   
  Note:** this is important step; otherwise, you will keep working on Master branch.
* Here, we did not use ‘**git checkout –b develop’** because we are not creating new branch here. **-b** being used only while creating new branch.
* Now, person-B will work the into this code and push the changes again to develop branch
* C:/> … B’s local project folder… > **git add \***
* C:/> … B’s local project folder… > **git commit –m “Release 1.1 change by person-B”**
* C:/> … B’s local project folder… > **git push origin develop**
* Now, assume that the code in ‘develop’ branch is also completed and now you have to merge it in Master branch. Then…
* Check your branch; if you are not in Master branch then switch into master branch
* C:/> … B’s local project folder… > **git branch**
* C:/> … B’s local project folder… >**git checkout master**
* Then safer side, get latest code from Master branch
* C:/> … B’s local project folder… > **git pull origin master**
* Now merge the develop into master
* C:/> … B’s local project folder… > **git merge develop**
* Here, ‘develop’ branch will be merged into ‘master’; since you are switched to ‘master’ branch using ‘checkout’ command; i.e. your active branch is ‘master’.
* Now, there could be a scenario of **Merge Conflict;** when that can occur?
* Let’s say there is updated code (e.g. {a,b,c,d}) in Master branch. Person-B created an another branch ‘develop’ and updated a code as {a,b,c,d,1,2,3}. Meanwhile person-A changes the code in Master branch it self as {a,b,c}; now person-B wants to merge the master branch into develop branch the Git will confuse since master branch has been changed or different than the person-B (when he take the master copy to create another branch). At this point of time; Merge Conflict error will occur and we have to solve it manually.
* Error will show the filename. When you goto that File, it indicates that what’s there in Master branch recently and what do you have from old master branch.
* You have to discuss with the team and delete or add required part into the file and then your have to perform ‘add’, ‘commit’, ‘merge’ commands.