Cucumber Tool| Behavior Driven Development (BDD)

- A cucumber is a **tool** based on Behavior Driven Development (BDD) **framework.** It explains the behavior of the application in a simple English text using Gherkin language
- BDD is a methodology to understand the functionality of an application in simple plain text representation.
- The main aim of Behavior Driven Development framework is to make various project roles such as Business Analysts, Quality Assurance, Developers, Support Team understand the application without diving deep into the technical aspects.
- Behavior Driven Development is an extension of Test Driven Development and it is used to test the system rather than testing the particular piece of code
- Cucumber tool is generally used in real time to write acceptance tests for an application. It is generally used by non-technical people such as Business Analysts, Functional Testers etc.

Advantages of Cucumber framework

- Cucumber is an open source tool.
- Plain Text representation makes it easier for non-technical users to understand the scenarios.
- It bridges the communication gap between various project stakeholders such as Business Analysts, Developers, and Quality Assurance personnel.
- Automation test cases developed using the Cucumber tool are easier to maintain and understand as well.
- Easy to integrate with other tools such as Selenium and Capybara.

Use of Behavior Driven Development in Agile methodology

- The advantages of Behavior Driven Development are best realized when non-technical users such as Business Analysts use BDD to draft requirements and provide the same to the developers for implementation.
- In Agile methodology, user stories can be written in the format of feature file and the same can be taken up for implementation by the developers.

- **Essential Components of Cucumber Tool**
  - "fileName.feature" file contains the tests in Gherkin syntax.
  - "fileName.java" contains the step definitions.
  - "TestRunner.java" contains the runner class that drives the execution.

"fileName.feature" file contains the tests in Gherkin syntax.

- Initially, you need to install "Natural" plugin in eclipse
  - First Method: Eclipse >> Help >> Eclipse Marketplace… >> search for "Natural" and install >>restart
  - Second Method: click HERE. Drag the icon and drop it in Eclipse >> install and restart
- Features file **contain high level description of the Test Scenario in simple language**. Each test case in Cucumber is called a scenario, and scenarios are grouped into features. Each scenario contains several steps. It is advisable that there should be a separate feature file, for each feature under test. The extension of the feature file needs to be ".feature".
- Feature files will be always under feature folder in **src/test/Resource folder**
- A simple feature file consist of following components:

Feature: Each Gherkin file begins with the "Feature:" keyword. This keyword doesn't affect the behavior of your Cucumber tests at all. It just gives you a convenient place to put some summary documentation about the group of tests that follow. The description can span multiple lines. The text immediately following on the same line as the Feature keyword is the "name" of the feature (Feature: Login Feature). In valid Gherkin, a "Feature" must be followed by one of the following: Scenario Background Scenario Outline

Background: you can have a single "Background" element per feature file, and it must appear before any of the "Scenario" or "Scenario Outline" elements. Background keyword is used to **group steps that are common to all the tests in the feature file**. This helps keep your scenarios clear and concise.

Scenario: To actually express the behavior we want, each feature contains one or several scenarios. Each scenario is a single concrete example of how the system should behave in a particular situation. Like "Feature", a "Scenario" description can also use multiple lines (Scenario: TekSchool Login Test Scenario)

Scenario Outline: allows a scenario to use its own data table for parameters.
In a scenario outline, each row of an "Examples" table represents a whole scenario to be executed by Cucumber. We can use a placeholder to replace any of the text we like in a step.it does not matter what order the placeholders appear in the table; what counts is that the column header matches the text in the placeholder in the scenario outline. A single "Scenario Outline" can have one or multiple "Examples" tables attached to it.

Given [Preconditions or Initial Context]
When [Event or Trigger]
Then [Expected output]
And [combines multiple "Given", "When" or "Then" statements together]
[Comments start with a "#" character and has to be the first and only thing on a line.]

@tagname Tag names can be any text. One or multiple tags can be used before "Scenario", or "Feature", or "Scenario Outline" elements and the individual "Examples" table under them. If a tag is used before "Feature", it will apply to all scenarios within the feature. The main reasons for tagging is Filtering. Cucumber allows you to use tags as a filter to pick out specific scenarios to run or report on.

Examples: All scenario outlines have to be followed with the Examples section. **This contains the data that has to be passed on to the scenario.** Pipe symbol (|) is used to specify one or more parameter values.

Data Driven Testing in Cucumber:

- Meaning: automatically run a test case multiple times with different input and values. There are different ways to use the data insertion within the *Cucumber*

⇒ **Parameterization without Example Keywork**

In .feature file:
Then user inserts "Test@gmail.com" and "Test1@3"
In Step Definition file:
@Then("^user inserts '(.+)' and '(.+)' $")
public void user_inserts_username_and_Password(String Username, String PassWord) { //}

⇒ **Parameterization with Example and Scenario Outline Keywords (*Scenario Outline* and *Examples* go hand in hand)**

In .feature file:
Scenario Outline:
Then user inserts "<Username >" and "<Password >"
Examples:
Username	Password
AAA@gmail.com	AAA1@3
BBB@gmail.com	BBB1@3
In Step Definition file:
@Then("^user inserts '(.+)' and '(.+)' $")
public void user_inserts_username_and_Password(String Username, String PassWord) { //}

- **Note**: The table must have a header row corresponding to the variables in the Scenario Outline steps.

⇒ **Parameterization using Test Data Tables (Immediately after each step)**

In .feature file:
Then user inserts username and password
| AAA@gmail.com | AAA1@3 |
| BBB@gmail.com | BBB1@3 |

In Step Definition file:
@Then("^user inserts username and password$")
public void user_inserts_username_and_Password(DataTable credentials) {

List<List<String>> data = credentials.raw(); //Import from Java.util

Element.sendKeys(data.get(0).get(0)); // pass the data based on row and column (index)

Element.sendKeys(data.get(0).get(1));}

- One draw back is that reader does not know what sort of data we are entering since it does not have header. Unlike Examples, it does not execute the entire scenario for each row.

⇒ **Test Data Tables Using Maps (Immediately after each step) with header**

- In this test we will pass *Username and Password* two times to the test step. So our test should enter *Username & Password* once, click on *LogIn* button and repeat the same steps again
- Map interface stores date using key and value.

Feature File Scenario

```
Scenario: Successful Login with Valid Credentials
 Given User is on Home Page
 When User Navigate to LogIn Page
And User enters Credentials to LogIn
 | Username    | Password |
    | testuser_1 | Test@153 |
    | testuser_2 | Test@154 |
Then Message displayed Login Successfully
```

The implementation of the above step will be like this:

```
@When("^User enters Credentials to LogIn$")
public void user_enters_testuser_and_Test(DataTable usercredentials) throws Throwable {

//Write the code to handle Data Table
for (Map<String, String> data : usercredentials.asMaps(String.class, String.class)) {
driver.findElement(By.id("log")).sendKeys(data.get("Username"));
    driver.findElement(By.id("pwd")).sendKeys(data.get("Password"));
    driver.findElement(By.id("login")).click();
}

}
```

Additional Tips:

- **Differences between hooks and Background Keyword ?**
  - Hooks where we define generally **Before() and After() those apply to every scenario in the project**. The Background applies only to the current .feature file.
- **Differences between Scenario Outline and DataTable?**
  - **Scenario Outline**
    - This uses Example keyword to define the test data for the Scenario
    - This works for the whole test
    - Cucumber automatically run the complete test the number of times equal to the number of data in the Test Set
  - **Data Table**
    - No keyword is used to define the test data
    - This works only for the single step, below which it is defined
    - A separate code needs to understand the test data and then it can be run single or multiple times but again just for the single step, not for the complete test

"fileName.java" contains the step definitions

- Step definition maps the Test Case Steps in the feature files (introduced by Given/When/Then) to actual code.
- In step definitions, we write Java codes, selenium codes, and Annotations.
- Step Definition is a java method in a class with an annotation above it. An annotation followed by the pattern is used to link the *Step Definition* to all the matching *Steps*, and the *code* is what *Cucumber* will execute when it sees a *Gherkin Step*
- For a step definition to be executed, it must match the given component in a feature. There are two methods:
  - Manually:
    - Suppose we have a step in .feature as below
      Given user is in the login page
    - In step definition, we connect using different annotations and write code in methods
      @Given("^user is in the login page$")
      Public void user_is_in_the_login_page (){ //write your code here}
  - Using TestRunner:
    - run the feature in TestRunner class and copy them from console.

"TestRunner.java" contains the runner class that drives the execution.

- This class drives the execution and it will be based on Junit.
- TestRunner class is used to provide the link between feature file and step definition file
- The purpose is to generate output/report
- the starting point of execution must be from TestRunner class
- No code should be written under the TestRunner class. It should include the tags @RunWith and @CucumberOptions.
- Below code should be outside of public class TestRunner{}
- Following Main **Options are available in Cucumber**:
  - Features: paths of feature files. Default Value: ({})
    - All we need is to specify the folder path and *Cucumber* will automatically find all the '***.features***' extension files in the folder
  - glue: paths of the step definition files. Default Value: ({})
    - It is almost the same think as *Features Option* but the only difference is that it helps *Cucumber* to locate the ***Step Definition file.***
  - tags: what tags in the features files should be executed. Default Value: ({})
    - Execute all tests tagged as @SmokeTest **OR** @RegressionTest (Tags that are **comma-separated** are ORed) tags: = {"@SmokeTest, @RegressionTest"}
    - Execute all tests tagged as @SmokeTest **AND** @RegressionTest (Tags which are passed in separate **quotes** are ANDed) tags: = {"@SmokeTest", "@RegressionTest"}
    - Execute all tests of the feature tagged as @FunctionalTests **but skip** scenarios tagged as @SmokeTest tags: = {"@FunctionalTest", "~@SmokeTest"}
  - dryRun: checks if all the steps have the step definition without execution of any steps. Default Value: (false)
  - monochrome: display the console output in much readable way. Default Value: (false)
  - strict: will fail execution if there are undefined or pending steps. Default Value: (false)
  - format: is used to specify different formatting options for the output reports . Default Value: (false)
    - ***Pretty:*** Prints the *Gherkin* source with additional colors and stack traces for errors.
      format = {"pretty"}
    - ***HTML:*** This will generate a HTML report at the location mentioned in the for-matter itself. Use below code: ***format = {"html:Folder_Name"}***
    - ***JSON:*** This report contains all the information from the gherkin source in JSON Format. This report is meant to be post-processed into another visual format by 3rd party tools such as Cucumber Jenkins. Use the below code:
      format = {"json:Folder_Name/cucumber.json"}
    - ***JUnit:*** This report generates XML files just like Apache Ant's JUnit report task. This XML format is understood by most Continuous Integration servers, who will use it to generate visual reports. use the below code:
      format = { "junit:Folder_Name/cucumber.xml"}

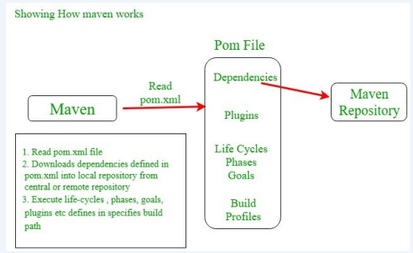
```
package runner;

import org.junit.runner.RunWith;
import cucumber.api.CucumberOptions;
import cucumber.api.junit.Cucumber;
@RunWith(Cucumber.class)
@CucumberOptions(
        plugin = {"json:target/cucumber-reports/CucumberTestReport.json"},
        features = "Feature File Path", // path
        glue = { "Step Definition Path" },
        tags = { "@" },
        dryRun = false,
        monochrome = true,
        strict = false,
        format = { "pretty", "html:target/cucumber-reports/cucumber-pretty",
                "json:target/cucumber-reports/CucumberTestReport.json",
                "rerun:target/cucumber-reports/rerun.txt" })
public class TestRunner {

}
```

Apache Maven | A build automation tool for Java projects

- **Maven is a build automation tool** used primarily for java projects that is based on POM (project object model). It is used for projects build, dependency and documentation. Maven repository is a directory of packaged JAR file with pom.xml file. Maven searches for dependencies in the repositories.
- It is a tool that can be used for building and managing any Java-based project
- Jar files are downloaded from **Maven Central repository.**
- How Maven works?



- **Core Concepts:**
  - **POM Files:** A Project Object Model or POM is the fundamental unit of work in Maven. Project Object Model (POM) Files are XML file that contains information related to the project and configuration information such as dependencies, source directory, plugin, goals etc. used by Maven to build the project. When you execute a maven command you give maven a POM file to execute the commands. Maven reads pom.xml file to accomplish its configuration and operations.
  - **Dependencies and Repositories:** Dependencies are external Java libraries required for Project and repositories are directories of packaged JAR files. The local repository is just a directory on your machine hard drive. If the dependencies are not found in the local Maven repository, Maven downloads them from a central Maven repository and puts them in your local repository.
    - By default the Maven local repository is the .m2 folder. You can copy the jar directly into where it is meant to go. Maven will find this file next time it is runs.
  - **Build Life Cycles, Phases and Goals:** A build life cycle consists of a sequence of build phases, and each build phase consists of a sequence of goals. Maven command is the name of a build lifecycle, phase or goal. If a lifecycle is requested executed by giving maven command, all build phases in that life cycle are executed also. If a build phase is requested executed, all build phases before it in the defined sequence are executed too.
  -
  - **Build Profiles:** Build profiles a set of configuration values which allows you to build your project using different configurations. For example, you may need to build your project for your local computer, for development and test. To enable different builds you can add different build profiles to your POM files using its profiles elements and are triggered in the variety of ways.
  - **Build Plugins:** Build plugins are used to perform specific goal. you can add a plugin to the POM file. Maven has some standard plugins you can use, and you can also implement your own in Java.

Installation process of Maven

The installation of Maven includes following Steps:

1. Verify that your system has java installed or not. if not then install java (Link for Java Installation)
2. Check java Environmental variable is set or not. if not then set java environmental variable. (link to install java and setting environmental variable)
3. Download maven (Link)
4. Unpack your maven zip at any place in your system.
5. Add the bin directory of the created directory apache-maven-3.5.3(it depends upon your installation version) to the PATH environment variable and system variable.
6. open cmd and run **mvn -v** command. If it print following lines of code then installation completed.

```
C:\Users\Ramzy>mvn -v
Apache Maven 3.6.3 (cecedd343002696d0abb50b32b541b8a6ba2883f)
Maven home: C:\apache-maven-3.6.3\bin\..
Java version: 1.8.0_231, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk1.8.0_231\jre
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

"pom.xml" contains the Maven setup

A Project Object Model or POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

        <modelVersion>4.0.0</modelVersion>
        <groupId> com.project.loggerapi </groupId>
        <artifactId>LoggerApi</artifactId>
        <version>0.0.1-SNAPSHOT</version>

        <!-- Add typical dependencies for a web application -->
        <dependencies>
                <dependency>
                        <groupId>org.apache.logging.log4j</groupId>
                        <artifactId>log4j-api</artifactId>
                        <version>2.11.0</version>
                </dependency>
        </dependencies>

</project>
```

Elements used for Creating pom.xml file:

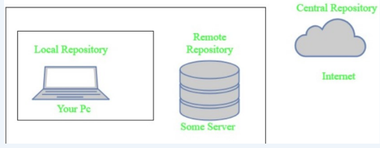
- **project-** It is the root element of the pom.xml file.
- **modelVersion-** modelversion means what version of the POM model you are using. Use version 4.0.0 for maven 2 and maven 3.
- **name-** this element is used to give name to our maven project.
- **dependencies-** dependencies element is used to defines a list of dependency of project. Some commonly used are (Junit, selenium-java, cucumber-java, cucumber-junit, cucumber-extentsreport, Webdrivermanager, cucumber-reporting)
- **dependency-** dependency defines a dependency and used inside dependencies tag. Each dependency is described by its groupId, artifactId and version.
- **groupId-** groupId means the id for the project group. It is unique and Most often you will use a group ID which is similar to the root Java package name of the project like we used the groupId com.project.loggerapi.
- **artifactId-** artifactId used to give name of the project you are building.in our example name of our project is LoggerApi.
- **version-** version element contains the version number of the project. If your project has been released in different versions then it is useful to give version of your project.
- **packaging-** packaging element is used to packaging our project to output types like JAR, WAR etc
- **Plugins:** is where much of reach actions take place. We have comiler, surefire and source plugin

Maven Repository:

a repository is a directory where all the project jars, library jar, plugins or any other project specific artifacts are stored and can be used by Maven easily. Maven has three types of repository :

1. Local repository
2. Central repository
3. Remote repository

Maven searches for dependencies in this repositories. First maven searches in Local repository then Central repository then Remote repository if Remote repository specified in the POM.



Maven life cycle:

- **Maven Compile**: compiles codes and **compiler plugin** is used for compilation
- **Maven Test**: execute test cases using **Surefire Plugin**
  - To run TestNG, we need to configure TestNG.xml in sureFirePlugin.
- **Maven Resources: Source plugin is used.** generates a Jar file when build is successful. Then, you can send this jar file to anyone. They can use this jar file by importing it through project build path.

Maven Commands:

- **mvn clean** cleans the maven project by deleting the **target** directory.
- **mvn install** builds the maven project and installs the project files (JAR, WAR, pom.xml, etc) to the local repository.
- **mvn clean install** executes the entire maven life cycle with a clean installation. Jar also
- **mvn clean install -DskipTests=true** compilation without execution. Jar also
- **mvn test** only test cases shall be executed.
- **mvn –v** used to display maven version information

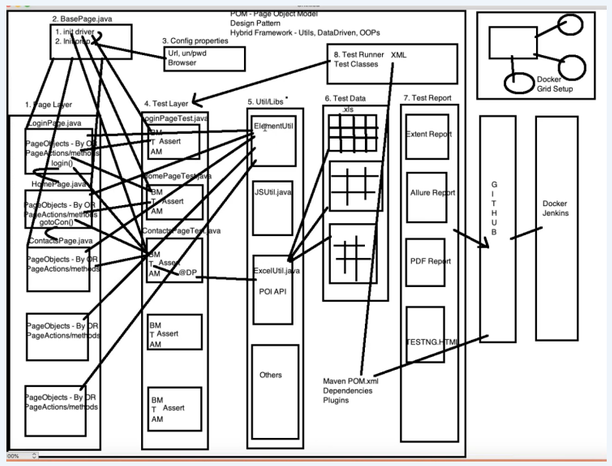
What is wrong with maven?

- Maven is not easy. The build cycle (what gets done and when) is not so clear within the POM.
- some issue arise with missing dependencies in public repositories.

Jar File

- Jenkins >> git >>pom.xml >> surefire plugin >> testing.xml
- Maven creates final Jar file only from src/main/java **NOT** src/test/java. Our test cases in src/main/java will not be stored in Jar file.
- Each dependency has three main parts (**GroupID, ArtifactID, Version**).
- To change the name of Jar file and versioning, modify here.
  *<**groupId**>OctPomSeriesFrameworkTest</**groupId**>*
  *<**artifactId**>OctPomSeriesFrameworkTest</**artifactId**>*
  *<**version**>0.0.1-SNAPSHOT</**version**>*
  *<**packaging**>jar</**packaging**>*

- Common Automation Framework (**CAF**) one framework for all of team.



- POM—Page Object Moded—it is not a framework. It is design pattern– hybrid framework (combination of different utils, OOPs and DataDriven)
- It is only for webpages. Under POM, you create a separate java class for each page.
- Create a maven project. Create packages for all above layers under src/main/java except for Test layer which goes under src/test/java.
- Add testing to library

"config.properties"

- It is dangerous to store hard coded values in the project, also it is against the coding principles. With the help of properties file, we will be eliminating these hard coded value one by one.
- Create a property file: Create a ***New File*** and give the extension as ***.properties***
- **.properties** is a file extension for files used in Java to store the configurable parameters. Each parameter is stored as a pair of strings, one storing the name of the parameter (called the key), and the other storing the value.
- Each line in a .properties file normally stores a single property
- Several formats are possible for each line, including
  - *key = value*
  - *key: value*
  - *key value*
- Note that single-quotes or double-quotes are considered part of the string
- Comment lines in .properties files are denoted by
  - number sign (#)
  - exclamation mark (!)
- **Add spaces to the key**: key\ with\ spaces = This is the value that could be looked up with the key "key with spaces".
- If you want your property to include a backslash, it should be escaped by another backslash

```
path=c:\\wiki\\templates
browser = chrome
url = https://app.hubspot.com/login
username = naveenanimation20@gmail.com
password = Test@12345
accountname = Greetel
```

"Base.java"

- Main Tasks: **create two methods: Initialize driver & Initialize properties**.
  - Initialize Properties from config.properties

```
public Properties init_Properties() {
    prop = new Properties();
    String firstPart = System.getProperty("user.dir");
    String finalAddress = firstPart +
"\\src\\main\\java\\com\\qa\\hubspot\\config\\config.properties";
    try {
        FileInputStream ip = new FileInputStream(finalAddress);
        prop.load(ip);
    } catch (FileNotFoundException e1) { e1.printStackTrace();}
    catch (IOException e) {e.printStackTrace();}

    return prop;}
```

  - Initialize Driver on the basis of browser name

```
public WebDriver init_driver(Properties prop) {
    String browser = prop.getProperty("browser");
    if (browser.equals("chrome")) {
        WebDriverManager.chromedriver().setup();
        driver = new ChromeDriver();
    } else if (browser.equals("firefox")) {
        WebDriverManager.firefoxdriver().setup();
        driver = new FirefoxDriver();
    } else {System.out.println(browser + " was NOT recognized...");}

    driver.manage().window().maximize();
    driver.manage().deleteAllCookies();
    driver.get(prop.getProperty("url"));
    return driver; }
```

PageLayer

- It extends from Base.java. In every page you need to have:
  - Page objects or Object Repository using By Locator

```
By emailID = By.id("username");
By password = By.id("password");
By loginButton = By.id("loginBtn");
```

  - Page class constructor

```
public LoginPage (WebDriver driver) {
this.driver=driver; }
```

  - Page actions/methods

```
public boolean IsSingUpLinkPresent() {
        boolean signup = driver.findElement
(signUpLink).isDisplayed();
        return signup;}

public HomePage doLogin(String Username, String Password) {
        driver.findElement
(emailAddressUser).sendKeys(Username);
        driver.findElement(PassBox).sendKeys
(Password);
        driver.findElement(loginBtn).click();
        return new HomePage();}
```

PageTestLayer

- For each page class, we need to have a test class.
- TestNG annotation will be use in Testclass (BM_T_AM)
- By locators are done either in Page layer or Utility. Better to move to Utility.
- Page layer >> create by locators, constructor and functions. In constructor, try to initialize both webdriver and Utility.
- Assertions are done in Test layer

```
public class LoginPageTest {

    WebDriver driver;
    BasePage basePage;
    Properties prop;
    LoginPage loginPage;

    @BeforeTest
    public void setUp() {
        basePage = new BasePage();          //create object of BasePage
        prop = basePage.init_Properties();  //init properties—always first—
save the return and pass to driver
        driver = basePage.init_driver(prop);  //init driver
        loginPage = new LoginPage(driver);}  //create object of loginPage

    @Test(priority = 1)
    public void verifyPageTitleTest() {
        Assert.assertEquals(loginPage.getPageTitle(), Con-
stant.LOGIN_PAGETITLE);}

    @Test(priority = 2)
    public void verifyPresenceOfSignUpLink() {
        Assert.assertTrue(loginPage.IsSingUpLinkPresent());}

    @Test(priority = 3)
    public void verifyLogin() {
        loginPage.doLogin(prop.getProperty("username"), prop.getProperty
("password"));}

    @AfterTest
    public void browserQuit() {
        driver.close();}}
```

Utility

- Utility >> Constants.java >> public final static String PAGE_TITLE = "HI"

Design of Data Provider

- Precondition: add three apache. Poi dependencies to pom
- Flow of integration: **Excel file >>ExcelUtil.java >> DataProvider >>TestNG**
  - Create excel file >> insert data in rows and columns >> drag and drop into the package (testdata) under src/main/java.
  - Create class excelUtil in Utilities >> this read the data from excel sheet

```
public class ExcelUtil {

    public static Workbook book;
    public static Sheet sheet;

    public static String TESTDATA_SHEET_PATH = "filepath";

    public static Object[][] getTestData(String sheetName) {
    try {
    FileInputStream ip = new FileInputStream(TESTDATA_SHEET_PATH);
        book = WorkbookFactory.create(ip);
        sheet = book.getSheet(sheetName);

        Object data[][] = new Object[sheet.getLastRowNum()][sheet.getRow
(0).getLastCellNum()];

        for (int i = 0; i < sheet.getLastRowNum(); i++) {
            for (int k = 0; k < sheet.getRow(0).getLastCellNum(); k++) {
            data[i][k] = sheet.getRow(i + 1).getCell(k).toString();}}
            return data;
        } catch (FileNotFoundException e) {e.printStackTrace();}
        catch (InvalidFormatException e) {e.printStackTrace();}
        catch (IOException e) {e.printStackTrace();}
        return null;}}
```

  - Integrate with DataProvider

```
@DataProvider
public Object[][] getContactsTestData() {
        Object data[][] = ExcelUtil.getTestData
(Constants.CONTACTS_SHEET_NAME.trim());
        return data; }
```

  - Integrate with TestNG

```
@Test(dataProvider = "getContactsTestData")
public void createNewContactTest(String email, String firstName, String lastName,
String jobTitle) {
        contactsPage.createNewContact(email, firstName, lastName, jobTitle);
```

GIT & GITHUB

- Version Control System (VCS) are systems that track changes made to a digital asset over time
- **Git**
  - Git is a Version Control System (VCS) with an emphasis to handle small and large projects with speed and efficiency
  - It is a collaboration platform for developers
  - "C" language is used in GIT. That is why it is fast
- **What Git can do:**
  - Repository: Git helps to create repositories
  - Versions: different version of the same artifacts can be stored
  - Artifact: helps to manage changes in artifact
  - Comparison: enables comparison of different versions of the same artifact
  - Collaboration: promotes cohabitation among developers
  - Accountability: maintains accountability
- **Advantages of using GIT**:
  - High availability
  - Only one .git directory per repository
  - Superior disk utilization and network performance
  - Collaboration friendly
  - Any sort of projects can use GIT
- **Repository in GIT**: A repository contains a directory named .git, where git keeps all of its metadata for the repository. The content of the .git directory are private to git.
- Differences between GIT and GITHUB:
  - Git is a tool
  - GITHUB is a website. Provides different services. (Task management, bug tracking, hosting services for GIT repo, wiki)
  - Git repository is hosted in GitHub
  - Every company has its own server repo in Git hub as http://google.github.com
- Functions:
  - **GIT PUSH** updates remote refs
  - **GIT CLONE** creates a copy of an existing GIT repo. To get the copy of a central repo, "Cloning" is the most common way used by programmers.
  - **GIT STATUS** shows the differences between the working directory and the index
  - **GIT ADD** adds file changes in your existing directory to your index
  - **GIT RESET** resets your index as well as the working directory to the state of your last commit
- A company can have many branches for each team
  - Master branch (by default) and feature branch
  - You can branch any branch you wish.
- Some important features of GIT
  - Push the code (Check-in)
  - Pull the code (Check-out)
- It is the responsibility of us to get the latest version.
- To use git commands, we need to install GIT first.
- If a SSH key is set for a repo, you need to have the key to push the code. You can pull but not push.
  - It is a specific key for a device.
- Push the code or check in
  - cd <*filepath*> Navigate to the directory in address <filePath>
  - git init initialize Existing directory as a GIT directory (hidden file will be created.)
  - git remote add origin <*url*> Register folder in your local with repository created
  - git status show modified files in working directory as red (Optional)
  - git add . Add complete project directory
    - git add Homepage.java you can specify the file or folder here
  - git status show modified files in working directory as Green (Optional)
  - git commit –m"<*descriptive message*>" adding changes to the local repository
  - git push origin master to transfer the last commit(s) **to** a remote server
- Pull the code or check out for **first time**
  - cd <*filepath*> Go to the directory in address <filePath>
  - git *clone* <*url*> retrieve an entire repository from a hosted location via URL
- Pull code for the second time and go on— it is our responsibility to get the code.
  - cd <*filepath*> Go to the directory in address <filePath>
  - git pull origin master to transfer the last commit(s) **from** a remote server

Plugin used in Eclipse

- Eclipse MarketPlace >> Egit >> install
- Right click on project >> Team >> Share Project
  - First commit
  - Then, fetch from the upstream
  - Then, push

Command Prompt:

- cls clears the screen
- Exit exits current command control
- dir <*folderName*> displays a list of a folder's files and subfolders available in <folderName>
- cd *filepath* displays the name of the current directory
- cd .. Returns one directory back
- mkdir <*filename*> Creates a folder by the name of <filename>
- rmdir <*filename*> Deletes a folder by the name of <filename>
- Ls –alt show all the files in the directory

Jenkins: CICD: Continuous Integration Continuous Delivery (

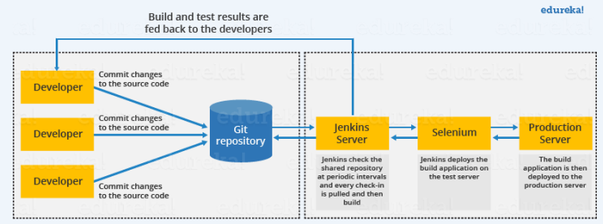
- **Dev Environment:** developers develop the application
- **QA Environment**— used by QA Engineers
  - **QA / Testing Environment (Phase 01)**:
    - Component / feature Testing happens here.
    - Front, back and manual testing happen here.
    - the first thing we do here always is **Sanity Testing** from TestNG.xml
    - If any bug found, it will be sent to dev team to correct and send a new version. (v2)
  - **Stage Environment (Phase 02)**: final tests are done here. Replica of live environment
    - Regression Testing, User Acceptance testing, back end testing also might happen here.
    - If any bug found here, it will go to the first build environment, then QA…, then here again.
    - When product is ready, it is called **RC(Release Candidate)**
- **Production Environment**: (Live Environment)
  - Beta Testing happens here by customers
  - Automation is always done in QA. No automation in production environment.
  - Final code is called : **BUILD** and each build has version.
  - In terms of java this build is called .Jar file with help of Maven (Build Automation Tool)
  - For performance testing, we create a **separate environment**. This is almost same replica of production environment.
  - The entire movement between different environment is done using Jenkins.

What is Jenkins?

Jenkins is an open-source automation tool written in Java with plugins built for Continuous Integration purposes. Jenkins is used to build and test your software projects continuously making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build. It also allows you to continuously deliver your software by integrating with a large number of testing and deployment technologies.

Define the process of Jenkins.

- First, a developer commits the code to the source code repository. Meanwhile, the Jenkins server checks the repository at regular intervals for changes.
- Soon after a commit occurs, the Jenkins server detects the changes that have occurred in the source code repository. Jenkins will pull those changes and will start preparing a new build.
- If the build fails, then the concerned team will be notified.
- If the build is successful, then Jenkins deploys the build in the test server.
- After testing, Jenkins generates feedback and then notifies the developers about the build and test results.
- It will continue to check the source code repository for changes made in the source code and the whole process keeps on repeating.



What are the pre-requisites for using Jenkins?

- A source code repository which is accessible, for instance, a Git repository.
- A working build script, e.g., a Maven script, checked into the repository.

What are the two components that you can integrate Jenkins with?

- Version Control system like GIT, SVN.
- Build tools like Apache Maven.

What are the commands you can use to start Jenkins manually.

To start Jenkins manually open Console/Command line, then go to your Jenkins installation directory. Over there you can use the below commands:

- Start Jenkins: **jenkins.exe start**
- Stop Jenkins: **jenkins.exe stop**
- Restart Jenkins: **jenkins.exe restart**

What is Continuous Integration In Jenkins?

In software development, multiple developers or teams work on different segments of the same web application. So in this case, you have to perform integration testing by integrating all modules. In order to do that an automated process for each piece of code is performed on a daily bases so that all your codes get tested. This process is known as continuous integration.

How do you achieve continuous integration using Jenkins?

- All the developers commit their source code changes to the shared Git repository.
- Jenkins server checks the shared Git repository at specified intervals and detected changes are then taken into the build.
- The build results and test results are shared to the respective developers
- The built application is displayed on a test server like Selenium and automated tests are run.
- The clean and tested build is deployed to the production server.

What are the types of pipelines in Jenkins?

There are 3 types –

- CI CD pipeline (Continuous Integration Continuous Delivery)
- Scripted pipeline
- Declarative pipeline

What is DevOps and in which stage does Jenkins fit in?

DevOps is a software development practice that blends software development (Dev) with the IT operations (Ops) making the whole development lifecycle simpler and shorter by constantly delivering builds, fixes, updates, and features. Jenkins plays a crucial role because it helps in this integration by automating the build, test and deployment process.

Have you run automated tests on Jenkins? How is it done?

Yes, this can be done easily. Automated tests can be run through tools like Selenium or maven. Developers can schedule the test runs. Jenkins displays the test results and sends a report to the developers.

Let us say, you have a pipeline. The first job was successful, but the second failed. What should you do next?

You just need to restart the pipeline from the point where it failed by doing 'restart from stage'.

What is the use of JENKINS HOME directory?

All the settings, logs and configurations are stored in the JENKINS_HOME directory.

What is the relation between Hudson and Jenkins?

You can just say Hudson was the earlier name and version of current Jenkins. After some issues, they renamed the project from Hudson to Jenkins.

Mention some of the useful plugins in Jenkins

- Maven 2 project
- Git
- Amazon EC2
- HTML publisher
- Copy artifact
- Join
- Green Balls

How to create a backup and copy files in Jenkins?

To create a backup all you need to do is to periodically back up your JENKINS_HOME directory.