Jenkins

Jenkins is an open-source automation server that provides hundreds of plugins to support building, deploying, and automating any project. Here's a step-by-step guide to setting up Jenkins for a Java project:

Step 1: Install Jenkins

- Download Jenkins: Go to the official Jenkins website and download the Jenkins package suitable for your operating system. Jenkins can be installed on Windows, macOS, and Unix-like operating systems.
- Install Jenkins: Follow the installation instructions for your platform. Usually, this involves executing the downloaded package and following the on-screen instructions.

Step 2: Setup Jenkins

- Initial Setup: After installation, navigate to http://localhost:8080 in your web browser to start the setup.
- Unlock Jenkins: Use the initial admin password found in the log or the specified file (e.g., /var/lib/jenkins/secrets/initialAdminPassword) to unlock Jenkins.
- Install Plugins: Choose to install suggested plugins or select specific plugins you need. Ensure you include plugins for Java and Git if they're not part of the suggested set.
- Create Admin User: Follow the prompt to create an admin user for Jenkins.

Step 3: Configure Java Project in Jenkins

- Create a New Job: On the Jenkins dashboard, click on “New Item”, enter a name for your project, and select “Freestyle project”.
- Source Code Management: Under the Source Code Management tab, select Git or another version control system you use, and enter your repository URL and credentials if needed.
- Configure Build Triggers: Choose how you want Jenkins to trigger builds (e.g., on push to the repository, periodically, etc.).
- Configure Build Environment: Ensure Jenkins knows where your Java and Maven (or Gradle) installations are. You might need to configure this in the “Global Tool Configuration” under “Manage Jenkins”.

Step 4: Add Build Steps

- Add Build Steps: In your project configuration, go to the Build section, click on “Add build step”, and select “Invoke top-level Maven targets” for Maven projects. For Gradle, there's a similar step, or you might need to use the “Execute shell” step for command-line builds.
- Configure Build: Enter the goals for Maven (e.g., clean install) or the tasks for Gradle.
- Post-build Actions: Optionally, configure post-build actions, such as archiving artifacts, deploying to a server, or sending notifications.

Step 5: Run Build

- Save Configuration: Click "Save" to save your project configuration.
- Build the Project: Click on “Build Now” to start your first build. Monitor the build progress and output in the Build History.

Step 6: Monitor and Maintain

- Dashboard: Use the Jenkins dashboard to monitor the builds.
- Adjust Configurations: Refine your build, trigger, and post-build configurations as needed.
- Set Up Notifications: Configure email or other types of notifications to get alerted about build successes or failures.

Using Git for Windows (Local)

1. Download and install Git for Windows

```
https://git-scm.com/downloads
```

2. Using Git for Windows (Local Repository Version Control)

   On your project folder:
   (use Git Bash to easily access some linux commands like touch and ls)

   $ git init
   $ git config user.name "john"
   $ git config user.email "john@gmail.com"
   $ touch .gitignore

   Gitignore files are items that will **NOT** be added to the repository.

   Sample content of gitignore file:

   ```
   Myreports/
   *.txt
   Otherfiles/
   ```

   Check files for staging:
   $ git status

   Stage files for tracking:
   $ git add <filename>

   Or
   $ git add *

   Then check:
   $ git status

   Unstage files
   $ git rm --cached <filename>
   $ git rm --cached *

   Perform a local commit
   $ git commit -m "my first commit"

   Show commit history
   $ git log

   Do some project code changes for a second commit.

   Check changes in code
   $ git diff

Try committing a second time

$ git commit -m "my second commit"

Jump to an old commit

$ git checkout <hash>

From here, you can make changes and perform a new commit

$ git commit -m "new commit"

Or discard changes and go back to previous master head

$ git switch -

Keep changes and create a branch from there

$ git switch -c <branch name>

Create a new branch from master and jump to it

$ git checkout -b <branch name>

Check which branch you are on

$ git branch

Jump to a branch

$ git branch <branchname>

Using GitHub

Upload local project to github repo

Create a github repo and clone it locally

$ git clone <github repo url>

Explain windows credential manager where github account login is cached in local computer

Note: you may also programmatically add and set origin github repo url

$ git remote add origin <github repo url>
$ git remote -v

Push your local repo (with correct origin url)

$ git push origin –all
$ git push origin master

Note: ignored files set in .gitignore will not be uploaded to remote github repo

Pull updates from github remote repo to your system

$ git pull <github repo>

Git and Github Activity

1. Open source case (owner and contributor)
   a. Owner creates a repo
   b. Contributor visits the owner's repo and forks it to his own github account
   c. Owner sees who forks his repo
   d. Contributor can submit issues by visiting the original repo
   e. Owner sees issues
   f. Contributor can now clone his forked version and work on it
   g. Contributor makes changes to his local copy and makes a push to his forked repo.
      Note: if branch push only
      \> git push origin <branch name>

   h. Contributor, using his account, can now make a pull request toward the owner of the original repo
   i. Owner sees the pull request, checks the code submitted by the contributor and decides if he approves it or not.
   j. If owner approves, owner can choose to merge the contributors forked version to original by merging. Verify changes to original repo.
   k. If owner disapproves, he can send message to contributor thru the pull request made. The contributor can then make changes, re-push to his fork and issue another pull request.

Behavior-Driven Design (BDD) and the Cucumber Framework

Behavior-Driven Development (BDD) is a software development approach that has evolved from Test-Driven Development (TDD). It encourages collaboration among developers, QA, and non-technical or business participants in a software project. BDD focuses on obtaining a clear understanding of desired software behavior through discussion with stakeholders. It extends TDD by writing test cases in a natural language that non-programmers can read.

Cucumber Framework
Cucumber is a popular tool that supports BDD. It allows you to write test cases that anybody can understand, regardless of their technical knowledge. These test cases are written in Gherkin language, which is a business readable, domain-specific language. Cucumber reads these Gherkin documents and validates that the software behaves as described. Each line of the Gherkin document is tied to a piece of code that will test the application.

How Cucumber is Used in Java
In a Java project, Cucumber tests are typically stored in feature files, which contain the Gherkin syntax. Here's how to use Cucumber with Java:

1. Add Cucumber to Your Java Project

   To use Cucumber with Java, you first need to add the Cucumber dependencies to your project's build file. For Maven, this would be in your pom.xml:

```
<dependencies>
  <dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>YourCucumberVersion</version>
  </dependency>
  <dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-junit</artifactId>
    <version>YourCucumberVersion</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

2. Write Feature Files

   Create **.feature** files in your project where you define your scenarios using Gherkin syntax. For example, a simple login feature could be described in a feature file like this:

```
Feature: Login

 Scenario: Successful login with correct credentials
  Given I am on the login page
  When I enter valid credentials
  Then I should be redirected to the dashboard
```

3. Implement Step Definitions

   For each step in your feature file, you need to implement a Java method that executes that step. These methods are annotated with @Given, @When, @Then, @And, and @But.

```
import io.cucumber.java.en.*;

public class LoginSteps {

  @Given("^I am on the login page$")
  public void i_am_on_the_login_page() {
    // Code to navigate to login page
  }

  @When("^I enter valid credentials$")
  public void i_enter_valid_credentials() {
    // Code to enter login credentials
  }

  @Then("^I should be redirected to the dashboard$")
  public void i_should_be_redirected_to_the_dashboard() {
    // Code to check redirection
  }
}
```

4. Run Cucumber Tests

   You can run Cucumber tests directly from your IDE or via the command line. To integrate with JUnit, you can use the @RunWith(Cucumber.class) annotation in a Java class. This class doesn't contain any code; it serves as an entry point for JUnit to run your Cucumber tests.

```
import io.cucumber.junit.Cucumber;
import io.cucumber.junit.CucumberOptions;
import org.junit.runner.RunWith;

@RunWith(Cucumber.class)
@CucumberOptions(features = "src/test/resources/features", glue = "your.step.definitions.package")
public class RunCucumberTest {
}
```

Benefits of Using BDD and Cucumber with Java

- Improved Communication: Since feature files are written in natural language, it's easier for developers, testers, and business stakeholders to understand the requirements and functionality.
- Documentation: The feature files also serve as documentation for the behavior of the system.
- Automated Testing: Cucumber automates the testing of software behavior as described in feature files, ensuring that the software meets business requirements.

Cucumber with Java is a powerful combination for implementing BDD in your development process. It bridges the communication gap between technical and non-technical team members by using simple, clear language to describe software features and behavior.

Cucumber sample

To create a simple example of using Cucumber for testing links on the website https://psa.gov.ph, let's follow these steps. We'll check if the homepage is accessible and then verify the presence of a specific link on the page. This example assumes basic familiarity with Java project setup, Maven, and the Cucumber framework.

Step 1: Project Structure
Assuming you're using Maven, your project directory might look something like this:

```
src
└── test
    ├── java
    │   └── com
    │       └── example
    │           └── stepdefs
    │               └── LinkCheckSteps.java
    └── resources
        └── features
            └── WebsiteLinkCheck.feature
```

Step 2: Add Dependencies in pom.xml
Include Cucumber, JUnit, and Selenium WebDriver for browser automation:

```
<dependencies>
  <!-- Cucumber -->
  <dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>7.0.0</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-junit</artifactId>
    <version>7.0.0</version>
    <scope>test</scope>
  </dependency>
  <!-- JUnit -->
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
    <scope>test</scope>
  </dependency>
  <!-- Selenium WebDriver -->
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>4.1.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Step 3: Feature File
Create a WebsiteLinkCheck.feature file in src/test/resources/features:

```
Feature: PSA Website Link Check

 Scenario: Home page is accessible
  Given I open the browser
  When I navigate to "https://psa.gov.ph"
  Then I should see the page title contains "Philippine Statistics Authority"

 Scenario: Verify a specific link presence on the homepage
  Given I am on the "https://psa.gov.ph" page
  When I look for the link with the text "About Us"
  Then I should find at least one link
```

Step 4: Step Definitions
Create a LinkCheckSteps.java in src/test/java/com/example/stepdefs:

```
package com.example.stepdefs;

import io.cucumber.java.en.*;
import org.junit.Assert;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class LinkCheckSteps {
  WebDriver driver;
  @Given("I open the browser")
  public void i_open_the_browser() {
    driver = new ChromeDriver();
  }

  @When("I navigate to {string}")
  public void i_navigate_to(String url) {
    driver.get(url);
  }

  @Then("I should see the page title contains {string}")
  public void i_should_see_the_page_title_contains(String title) {
    Assert.assertTrue(driver.getTitle().contains(title));
    driver.quit();
  }

  @Given("I am on the {string} page")
  public void i_am_on_the_page(String url) {
    driver = new ChromeDriver();
    driver.get(url);
  }

  @When("I look for the link with the text {string}")
  public void i_look_for_the_link_with_the_text(String linkText) {
    // This is just to find the element, assertion happens in the next step
    driver.findElement(By.linkText(linkText));
  }

  @Then("I should find at least one link")
  public void i_should_find_at_least_one_link() {
    // Assuming the element is found in the previous step, simply quit the browser
    driver.quit();
  }
}
```