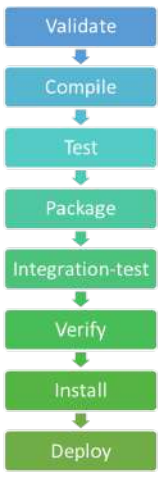
- Pre-clean
- Clean
- Post-clean

The mvn post-clean command is used to invoke the clean lifestyle phases in Maven. When *mvn clean* command executes, Maven deletes the build directory.

Default/Build Life Cycle

Default/Build Life Cycle contains these most important life cycle.

- *validate* – checks the correctness of the project
- *compile* – compiles the provided source code into binary artifacts
- *test* – executes unit tests
- *package* – packages compiled code into an archive file
- *integration-test* – executes additional tests, which require the packaging
- *verify* – checks if the package is valid
- *install* – installs the package file into the local Maven repository
- *deploy* – deploys the package file to a remote server or repository



Site Life Cycle

The Maven site plugin handles the project site's documentation, which is used to create reports, deploy sites, etc.

The phase has four different stages:

- Pre-site
- Site
- Post-site
- Site-deploy

Ultimately, the site that is created contains the project report.

Manage Dependencies

One of the core features of Maven is Dependency Management. Managing dependencies is a difficult task once we've to deal with multi-module projects (consisting of hundreds of modules/sub-projects). Maven provides a high degree of control to manage such scenarios.

Transitive Dependencies Discovery

It is pretty often a case, when a library, say A, depends upon other library, say B. In case another project C wants to use A, then that project requires to use library B too.

Maven helps to avoid such requirements to discover all the libraries required. Maven does so by reading project files (pom.xml) of dependencies, figure out their dependencies and so on.

We only need to define direct dependency in each project pom. Maven handles the rest automatically.

Dependency Scope

Scope & Description
compile This scope indicates that dependency is available in classpath of project. It is default scope.
provided This scope indicates that dependency is to be provided by JDK or web-Server/Container at runtime.
runtime This scope indicates that dependency is not required for compilation, but is required during execution.
test This scope indicates that the dependency is only available for the test compilation and execution phases.
system This scope indicates that you have to provide the system path.
import This scope is only used when dependency is of type pom. This scope indicates that the specified POM should be replaced with the dependencies in that POM's <dependencyManagement> section.

Maven Plugins

Maven is actually a plugin execution framework where every task is actually done by plugins. Maven Plugins are generally used to −

- create jar file
- create war file
- compile code files
- unit testing of code
- create project documentation
- create project reports

Command to execute any plugin is

```
mvn [plugin-name]:[goal-name]
```

There are two types of maven plugins according to Apache Maven,

1. Build Plugins
2. Reporting Plugins

Build Plugins

These plugins are executed at the time of build. These plugins should be declared inside the **<build>** element.

Reporting Plugins

These plugins are executed at the time of site generation. These plugins should be declared inside the **<reporting>** element.

Maven Core Plugins

A list of maven core plugins are given below:

Plugin	Description
clean	clean up after build.
compiler	compiles java source code.
deploy	deploys the artifact to the remote repository.
failsafe	runs the JUnit integration tests in an isolated classloader.
install	installs the built artifact into the local repository.
resources	copies the resources to the output directory for including in the JAR.
site	generates a site for the current project.
surefire	runs the JUnit unit tests in an isolated classloader.
verifier	verifies the existence of certain conditions. It is useful for integration tests.

To see the list of maven plugins, you may visit apache maven official website http://repo.maven.apache.org/maven2/org/apache/maven/plugins/

CA - Experiment 1 - Introduction To Maven & Gradle Build Tools 🖥️⚙️🏗️

Experiment 1: Introduction to Maven and Gradle

🌟 Objective

To understand build automation tools, compare **Maven** and **Gradle**, and set up both tools for software development.

🔹 1. Overview of Build Automation Tools

Build automation tools simplify the process of **compiling, testing, packaging, and deploying** software projects. They manage dependencies, execute tasks, and integrate seamlessly with CI/CD pipelines.

✨ Why Use Build Automation?

✅ Ensures **consistency** across builds
✅ Handles **dependency management** automatically
✅ Reduces **manual errors** and increases efficiency
✅ Integrates with tools like **Jenkins & Azure DevOps**

🚀 Popular Build Automation Tools

- **Apache Ant** → Script-based, manual dependency management
- **Apache Maven** → XML-based, convention-over-configuration model
- **Gradle** → Flexible, fast, and supports Groovy/Kotlin DSL

🔹 2. Key Differences Between Maven and Gradle

Feature	🟠 Maven (XML)	🟢 Gradle (Groovy/Kotlin)
Build Script	`pom.xml`	`build.gradle` / `build.gradle.kts`
Performance	Sequential, slower	Parallel execution, faster
Flexibility	Convention-based	Highly customizable
Dependency Mgmt.	Uses Maven Repository	Supports multiple repositories
Ease of Use	Simple XML structure	Slightly complex but powerful
Caching Support	No build caching	Supports incremental builds
Best For	Standard Java projects	Complex, high-performance builds

📌 **Maven** is great for structured, **enterprise-level** projects.
📌 **Gradle** is ideal for **scalable and performance-driven** applications.

🔹 3. Installation and Setup

🔵 3.1 Installing Maven

✅ Step 1: Install Java (JDK 17 Recommended)

Check if Java is installed:

```
java -version
javac -version

```

✅ Step 2: Download and Install Maven

🔗 **Download from:** Download Apache Maven – Maven
📁 Extract it to a folder (e.g., `C:\Maven`).

✅ Step 3: Configure Environment Variables

📌 **Windows:**

- Add `MAVEN_HOME` → `C:\Maven\apache-maven-<version>`
- Update `Path` → `%MAVEN_HOME%\bin`

📌 **Linux/macOS:**
Add the following to `.bashrc` or `.zshrc`:

```
export MAVEN_HOME=/opt/maven/apache-maven-<version>
export PATH=$MAVEN_HOME/bin:$PATH

```

✅ Step 4: Verify Installation

Run:

```
mvn -version

```

Expected Output:

```
Apache Maven 3.x.x
Maven home: C:\Maven\apache-maven-<version>
Java version: 17.0.4

```

🟢 3.2 Installing Gradle

✅ Step 1: Install Java (JDK 17 Recommended)

Same steps as Maven.

✅ Step 2: Download and Install Gradle

🔗 **Download from:** Gradle | Releases
📁 Extract it to a folder (e.g., `C:\Gradle`).

✅ Step 3: Configure Environment Variables

📌 **Windows:**

- Add `GRADLE_HOME` → `C:\Gradle\gradle-<version>`

- Update `Path` → `%GRADLE_HOME%\bin`

📌 **Linux/macOS:**

Add the following to `.bashrc` or `.zshrc`:

```
export GRADLE_HOME=/opt/gradle/gradle-<version>
export PATH=$GRADLE_HOME/bin:$PATH
```

✅ **Step 4: Verify Installation**

Run:

```
gradle -v
```

Expected Output:

```
Gradle 8.x
Build time: YYYY-MM-DD HH:MM:SS
Kotlin: X.Y
Groovy: X.Y
```

📌 Assessment Questions

1️⃣ What are the **key advantages** of using a build automation tool?
2️⃣ Mention **three** major differences between Maven and Gradle.
3️⃣ How does Gradle achieve **faster build times** compared to Maven?
4️⃣ What are the **necessary environment variables** for setting up Maven and Gradle?
5️⃣ How do you **verify** that Maven and Gradle are installed correctly?

CA - Experiment 2 - Working with Maven: Creating a Maven Project, Understanding the POM File, Dependency Management and Plugins

Maven: Basic Introduction

Maven is a build automation tool primarily used for Java projects. It simplifies the build process, manages project dependencies, and supports plugins for different tasks. It uses XML files (called `pom.xml`) for project configuration and dependency management.

Key Benefits of Maven:

- Dependency management (automates downloading and including libraries).
- Build automation (compiles code, runs tests, creates artifacts).
- Consistent project structure (standardizes how Java projects are set up).
- Integration with CI tools (like Jenkins).

Key Features of Maven:

- **Project Management**: Handles project dependencies, configurations, and builds.
- **Standard Directory Structure**: Encourages a consistent project layout across Java projects.
- **Build Automation**: Automates tasks such as compilation, testing, packaging, and deployment.
- **Dependency Management**: Downloads and manages libraries and dependencies from repositories (e.g., Maven Central).
- **Plugins**: Supports many plugins for various tasks like code analysis, packaging, and deploying.

Steps to Create a Maven Project in IntelliJ IDEA

1. **Install Maven (if not already installed)**:
   - Download Maven from the official website.
   - Set the `MAVEN_HOME` environment variable and update the system `PATH`.
2. **Create a New Maven Project**:
   - Open IntelliJ IDEA.
   - Go to `File` > `New` > `Project`.
   - Select `Maven` from the project types.
   - Choose `Create from Archetype` (optional) or proceed without.
   - Set the project name and location, then click `Finish`.
3. **Set Up the `pom.xml` File**:
   - The `pom.xml` file is where you define dependencies, plugins, and other configurations for your Maven project.
   - Example of a basic `pom.xml`:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```xml
         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.example</groupId>
    <artifactId>simple-project</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- Add your dependencies here -->
    </dependencies>

</project>

```

4. **Add Dependencies for Selenium and TestNG**:
   - In the `pom.xml`, add Selenium and TestNG dependencies under the `<dependencies>` section.

```xml
<dependencies>
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <version>3.141.59</version>
    </dependency>
    <dependency>
        <groupId>org.testng</groupId>
        <artifactId>testng</artifactId>
        <version>7.4.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>

```

5. **Create a Simple Website (HTML, CSS, and Logo)**:
   - In the `src/main/resources` folder, create an `index.html` file, a `style.css` file, and place the `logo.png` image.

Example of a simple `index.html`:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Simple Website</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <header>
        <img src="logo.png" alt="Logo">
    </header>
    <h1>Welcome to My Simple Website</h1>
</body>
</html>

```

Example of a simple `style.css`:

```css
body {
    font-family: Arial, sans-serif;
    background-color: #f4f4f4;
    text-align: center;
}
header img {
    width: 100px;
}

```

6. **Upload the Website to GitHub**:
   - Initialize a Git repository in your project folder:

     ```
     git init

     ```

   - Add your files and commit them:

     ```
     git add .
     git commit -m "Initial commit"

     ```

   - Create a GitHub repository and push the local project to GitHub:

     ```
     git remote add origin <your-repository-url>
     git push -u origin master

     ```

Deployment :

To deploy your Maven project to **GitHub Pages** using the `/docs` folder (** ***having all files inside root folder/dir not recommended***), you can follow these simple steps. This method is easy and doesn't require switching branches—just use the `/docs` folder of your main branch.

Steps to Deploy to GitHub Pages Using `/docs` Folder

1. **Modify Maven Configuration to Copy Static Files to `/docs` Folder**: First, you need to ensure that Maven places your static files (`index.html`, `style.css`, `logo.png`) into the `/docs` folder instead of the `target` directory.

   To do this, configure the **Maven Resources Plugin** in your `pom.xml` to copy the files directly into `/docs`:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-resources-plugin</artifactId>
            <version>3.2.0</version>
            <executions>
                <execution>
                    <phase>prepare-package</phase> <!-- Before packaging -->
                    <goals>
                        <goal>copy-resources</goal>
                    </goals>
                    <configuration>
                        <outputDirectory>${project.basedir}/docs</outputDirectory> <!-- Deploy to /docs
folder -->
                        <resources>
                            <resource>
                                <directory>src/main/resources</directory>
```

```
                                <includes>
                                    <include>**/*</include> <!-- Copy all files in src/main/resources -->
                                </includes>
                            </resource>
                        </resources>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>

```

In this configuration:

- The `maven-resources-plugin` copies all files from `src/main/resources` to the `/docs` folder in the root of your project (not the `target` directory).
- This is done during the `prepare-package` phase, just before Maven prepares the package.

2. **Build the Project**: Run the following Maven command to build your project and copy the resources to the `/docs` folder:

```
mvn clean install

```

After this, your `index.html`, `style.css`, and `logo.png` files should now be inside the `docs` folder in the root of your project.

3. **Push Changes to GitHub**: Now that the files are in the `/docs` folder, they are ready to be served by GitHub Pages. Follow these steps:
   - **Stage the changes** (the updated `/docs` folder):

```
git add docs/*
git commit -m "Deploy site to GitHub Pages"

```

   - **Push to GitHub**:

```
git push origin master  # Or the branch you are using, maybe 'main' in some cases

```

4. **Enable GitHub Pages**: After pushing to the main branch, follow these steps to enable GitHub Pages:
   - Go to your GitHub repository.
   - Navigate to **Settings** > **Pages** (on the left sidebar).
   - Under the **Source** section, select the `main` branch and `/docs` folder as the source.
   - Click **Save**.
5. **Access Your Website**: Your static website is now hosted on GitHub Pages! You can access it at:

```
https://<your-github-username>.github.io/<your-repository-name>/

```

Summary:

- **Maven Resources Plugin** is configured to copy static files (`index.html`, `style.css`, `logo.png`) into the `/docs` folder.
- **Build and push** the changes to your GitHub repository.
- Enable **GitHub Pages** using the `/docs` folder as the source.

By using the `/docs` folder in the main branch, you avoid the complexities of working with a separate `gh-pages` branch, and it's easier to maintain your website alongside your project.

Information about the /docs folder

The `/docs` folder does **not need to be manually created**. When you configure the Maven build to copy resources to the `/docs` folder, Maven will automatically create this folder when you run the `mvn clean install` command (if it doesn't already exist).

Here's a clearer breakdown:

1. **The `/docs` Folder**:
   - This folder will be automatically created during the build process when Maven runs the `maven-resources-plugin`.
   - Maven will copy the contents from `src/main/resources` into the `/docs` folder.
2. **No Need for Manual Creation**: You don't need to manually create the `/docs` folder. Maven handles this automatically based on the configuration in your `pom.xml` file.

After the Build:

Once the build completes, check the root of your project directory, and you should see a `docs` folder with all the copied files inside it.

Summary:

The `/docs` folder is not something you need to create manually. Maven will generate it automatically and place the static files inside it based on your configuration in the `pom.xml` file. After building and pushing the changes to GitHub, you can enable GitHub Pages to serve the content from this folder.

Concise step-by-step guide to deploy your Maven project to GitHub Pages using the `/docs` folder, assuming you already have necessary files inside it.

Here's a concise step-by-step guide to deploy your Maven project to GitHub Pages using the `/docs` folder, assuming you already have your `index.html`, `style.css`, and other necessary files:

1. Prepare Your Maven Project

- Open your `pom.xml` file and configure the `maven-resources-plugin` to copy your static files (like `index.html`, `style.css`, etc.) into the `/docs` folder:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-resources-plugin</artifactId>
            <version>3.2.0</version>
            <executions>
                <execution>
                    <phase>prepare-package</phase>
                    <goals>
                        <goal>copy-resources</goal>
                    </goals>
                    <configuration>
                        <outputDirectory>${project.basedir}/docs</outputDirectory>
                        <resources>
```

```
                            <resource>
                                <directory>src/main/resources</directory>
                                <includes>
                                    <include>**/*</include>
                                </includes>
                            </resource>
                        </resources>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>

```

This ensures that all your static files from `src/main/resources` are copied into the `docs` folder.

2. Build the Project

Run Maven to build the project:

```
mvn clean install

```

This will generate the `/docs` folder and place your static files there.

3. Push Changes to GitHub

- Stage and commit the changes (i.e., the new `/docs` folder with your static files):

```
git add docs/*
git commit -m "Deploy site to GitHub Pages"

```

- Push to your repository (assuming you're working with the `master` branch):

```
git push origin master

```

4. Enable GitHub Pages

- Go to your **GitHub repository**.
- Navigate to **Settings** > **Pages** (in the sidebar).
- Under the **Source** section, select:
  - **Branch**: `main`
  - **Folder**: `/docs`
- Click **Save**.

5. Access Your Website

Your website will now be live at:

```
https://<your-github-username>.github.io/<your-repository-name>/

```

Summary:

1. **Configure Maven**: Set up `maven-resources-plugin` in `pom.xml` to copy files to `/docs`.

2. **Build the Project**: Run `mvn clean install` to generate the `/docs` folder.
3. **Push to GitHub**: Stage and commit the `/docs` folder, then push to the `main` branch.
4. **Enable GitHub Pages**: Configure GitHub Pages to use the `/docs` folder.
5. **Access**: Your site will be hosted on GitHub Pages at `https://<your-username>.github.io/<repo-name>/`.

---

7. **Write a Simple Selenium Test with TestNG**:
   - Create a new Java class `WebPageTest.java` in the `src/test/java` directory.

   Example of a simple TestNG test using Selenium:

```java
package org.test;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;

import static org.testng.Assert.assertTrue;

public class WebpageTest {
    private static WebDriver driver;

    @BeforeTest
    public void openBrowser() throws InterruptedException {
        driver = new ChromeDriver();
        driver.manage().window().maximize();
        Thread.sleep(2000);
        driver.get("https://sauravsarkar-codersarcade.github.io/CA-MVN/"); // "Note: You should use your GITHUB-URL here...!!!"
    }

    @Test
    public void titleValidationTest(){
        String actualTitle = driver.getTitle();
        String expectedTitle = "Tripillar Solutions";
        Assert.assertEquals(actualTitle, expectedTitle);
        assertTrue(true, "Title should contain 'Tripillar'");
    }

    @AfterTest
    public void closeBrowser() throws InterruptedException {
        Thread.sleep(1000);
        driver.quit();
    }
}


```

8. **Run the Test**:
   - In IntelliJ, right-click the `WebPageTest` class and select `Run 'WebPageTest'`.
   - The test will launch Chrome, open the webpage, and validate the title.

Summary of Steps:

1. Set up Maven project and configure `pom.xml`.
2. Create a simple website with HTML, CSS, and a logo image.
3. Upload the project to GitHub.
4. Write and run a Selenium test with TestNG to validate the webpage title.

Testing the title of your website using **Selenium**, **Java**, and **TestNG**:

To test the title of your website using **Selenium**, **Java**, and **TestNG**, follow these steps. This will include the installation of necessary dependencies, creating test scripts, and running tests.

1. Set Up Selenium and TestNG Dependencies

In your Maven project, add the necessary dependencies for **Selenium WebDriver** and **TestNG** to the `pom.xml` file: (**Skip if already added..!!**)

```xml
<dependencies>
    <!-- Selenium WebDriver dependency -->
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <version>4.8.0</version> <!-- Ensure this is the latest version -->
    </dependency>

    <!-- TestNG dependency -->
    <dependency>
        <groupId>org.testng</groupId>
        <artifactId>testng</artifactId>
        <version>7.7.0</version> <!-- Ensure this is the latest version -->
        <scope>test</scope>
    </dependency>
</dependencies>

```

- **Selenium WebDriver**: This is used for browser automation.
- **TestNG**: This is a testing framework used to run Selenium tests.

2. Create Selenium Test Class Using TestNG

Next, create a test class in your `src/test/java` directory. You can name it `WebsiteTitleTest.java`.

Sample Code for Testing Website Title:

```java
package org.test;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;

import static org.testng.Assert.assertTrue;

public class WebpageTest {
    private static WebDriver driver;

    @BeforeTest
    public void openBrowser() throws InterruptedException {
        driver = new ChromeDriver();
        driver.manage().window().maximize();
        Thread.sleep(2000);
        driver.get("https://sauravsarkar-codersarcade.github.io/CA-MVN/"); // "Note: You should use your GITHUB-URL here...!!!"
    }

    @Test
    public void titleValidationTest(){
        String actualTitle = driver.getTitle();
        String expectedTitle = "Tripillar Solutions"; // "Replace with Your HTML WebPage Title"
        Assert.assertEquals(actualTitle, expectedTitle);
    }

    @AfterTest
    public void closeBrowser() throws InterruptedException {
        Thread.sleep(1000);
        driver.quit();
    }

}

```

Code Explanation (Step-by-Step)

📌 **Package Declaration**

- `package org.test;`
  - Defines the package name as `org.test` (helps organize code).

📌 **Imports Required Libraries**

- `import org.openqa.selenium.WebDriver;` → Selenium WebDriver interface.
- `import org.openqa.selenium.chrome.ChromeDriver;` → Controls Google Chrome.
- `import org.testng.Assert;` → Provides assertion methods for validation.
- `import org.testng.annotations.*;` → TestNG annotations for test execution.
- `import static org.testng.Assert.assertTrue;` → Allows direct usage of `assertTrue()`.

📌 **Class Declaration**

- `public class WebpageTest {}` → Defines a **test class** for webpage testing.

📌 **WebDriver Declaration**

- `private static WebDriver driver;`
  - Declares a static WebDriver instance for browser control.

1️⃣ `@BeforeTest` - Setup Method

```java
@BeforeTest
public void openBrowser() throws InterruptedException {

```

- Runs **before any test case** in this class.

- Initializes `ChromeDriver()` (opens Chrome).
- Maximizes the browser window.
- Waits for **2 seconds** (`Thread.sleep(2000)`).
- Navigates to `"https://sauravsarkar-codersarcade.github.io/CA-MVN/"` .

2️⃣ `@Test` - Title Validation Test

```
@Test
public void titleValidationTest() {

```

- Retrieves the **actual title** of the web page using `driver.getTitle()` .
- Defines the **expected title** as `"Tripillar Solutions"` .
- Uses `Assert.assertEquals(actualTitle, expectedTitle);`
  - **Passes** if the title matches `"Tripillar Solutions"` .
  - **Fails** if the title is different.

3️⃣ `@AfterTest` - Cleanup Method

```
@AfterTest
public void closeBrowser() throws InterruptedException {

```

- Runs **after all test cases** in this class.
- Waits for **1 second** (`Thread.sleep(1000)`).
- Closes the browser using `driver.quit();` .

Key Takeaways

✅ **Selenium WebDriver** automates browser interaction.
✅ **TestNG Annotations** manage test setup (`@BeforeTest`), execution (`@Test`), and teardown (`@AfterTest`).
✅ **Assertions (** `Assert.assertEquals()` **)** validate webpage content.

3. Run the Test Using TestNG

Option 1: Run TestNG from IntelliJ IDEA

1. Right-click the `WebpageTest.java` file.
2. Select **Run WebpageTest**.
   - IntelliJ IDEA will execute the TestNG test and show the results in the output console.

Option 2: Run TestNG via Command Line

If you want to run the tests from the command line, use the following Maven command:

```
mvn test

```

This will trigger the TestNG tests and execute all the tests in the `src/test/java` directory.

4. Verify the Test Results

After running the test, TestNG will output the results:

- If the website title matches the expected title, the test will pass.
- If the title is incorrect, the test will fail, and you will get an assertion error with a message indicating that the title doesn't match.

Summary of Steps:

1. **Add Dependencies**:
   - Add `selenium-java`, `testng`, and `webdrivermanager` dependencies in `pom.xml`.
2. **Write Test Class**:
   - Create a class `WebpageTest.java` with methods to launch the browser, navigate to the website, and verify the title.
3. **Run the Tests**:
   - Use IntelliJ IDEA or Maven to run the tests and check the output for validation.
4. **Verify Results**:
   - Ensure that the actual title matches the expected title. If they match, the test passes; if not, it fails.

This approach tests whether the website is correctly deployed and whether the title is as expected on your GitHub Pages site.

Use of **maven-jar-plugin** to **package** the project into a **jar file.** | | Showing how to **run** a **main class** and show simple **output**:

Great! You're on the right track. Here's a quick summary of what we have covered and additional steps you can take to demonstrate the use of **maven-jar-plugin** and how to package the project into a JAR file, including running a `main` class and showing simple output.

Summary of What We've Done So Far:

1. **Website Deployment**:
   - You've deployed your simple HTML, CSS, and assets (like logo.png) to **GitHub Pages** using Maven.
   - You've created a **Selenium Test** to validate the website title using **TestNG** and ran the test to ensure the website is functioning as expected.
2. **Next Steps**:
   - We can show how to use the **maven-jar-plugin** to create a runnable JAR file.
   - Demonstrate running a **main class** inside this JAR to produce a simple output.

Steps to Package the Project as a JAR and Run a Main Class

1. Add `maven-jar-plugin` to `pom.xml`:

To package your Maven project as a JAR file and specify the `main` class, we need to configure the **maven-jar-plugin** in the `pom.xml`.

Add the following configuration to your `pom.xml`:

```
<build>
    <plugins>
        <!-- Maven JAR Plugin -->
        <plugin>
```

```
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>3.2.0</version>
            <configuration>
                <!-- Specify the main class to be executed -->
                <archive>
                    <manifestEntries>
                        <Main-Class>com.example.MainClass</Main-Class> <!-- Replace with your main class path -->
                    </manifestEntries>
                </archive>
            </configuration>
        </plugin>
    </plugins>
</build>

```

This will tell Maven to include the **Main-Class** in the JAR manifest and specify the main class that should be executed when the JAR is run.

2. Create a Main Class:

In your `src/main/java` directory, create a class with a `main` method. For example, create a `MainClass.java` under `com.example`:

```
package com.example;

public class MainClass {
    public static void main(String[] args) {
        System.out.println("Hello, this is a simple output from the main class!");
    }
}

```

This class contains a simple `main` method that prints output when run.

3. Package the Project into a JAR:

After configuring the plugin and creating the `MainClass`, run the following Maven command to build the project and package it into a JAR file:

```
mvn clean package

```

This will clean any previous builds, compile the source code, and package it into a JAR file located in the `target` directory (e.g., `target/your-project-name.jar`).

4. Run the JAR File:

Once the JAR is created, you can run it with the following command:

```
java -jar target/your-project-name.jar

```

This will execute the `main` method from your `MainClass` and print the message:

```
Hello, this is a simple output from the main class!

```

Summary:

1. **Add maven-jar-plugin**: Configure the plugin in `pom.xml` to specify the `main class`.
2. **Create Main Class**: Write a simple `MainClass` with a `main` method that outputs a message.
3. **Package with Maven**: Run `mvn clean package` to package the project into a JAR file.
4. **Run the JAR**: Use `java -jar target/your-project-name.jar` to run the packaged JAR and print the output.

Final Thoughts:

- By showing this process, you demonstrate how Maven is used to automate the build, test, deployment, and packaging of projects.
- The **maven-jar-plugin** allows you to easily create runnable JAR files, and running the `main` class from the JAR file is a common way to run Java applications.

This provides a good conclusion to the topic, demonstrating not only the deployment and testing of the website but also the power of Maven for packaging and managing Java projects.

Important Note :

The **maven-resources-plugin** that we used earlier for copying files (like HTML, CSS, images) to the `/docs` folder doesn't get replaced or removed when you configure the **maven-jar-plugin** for packaging the project. These two plugins serve different purposes and can coexist in your `pom.xml`.

How They Work Together:

1. **maven-resources-plugin**: This plugin is responsible for copying your resources (like HTML, CSS, images, etc.) to the appropriate location in the target directory (e.g., `/docs` or `/target/classes`). We used it to ensure that all static assets were copied to the correct folder for GitHub Pages deployment.
2. **maven-jar-plugin**: This plugin handles the packaging of your Java code into a JAR file. It creates the JAR with the specified resources, including the class files and any configured files (like your `MainClass`), but it doesn’t interfere with the static files used for GitHub Pages.

Key Points:

- **maven-resources-plugin** will still be used to copy your static resources to the `/docs` folder as configured.
- **maven-jar-plugin** will only package your Java classes and any other necessary resources for the JAR file.
- Both plugins can run independently during the build process, so there’s no conflict between them.

Example Setup for Both Plugins:

Here’s how both plugins would coexist in your `pom.xml`:

```xml
<build>
    <plugins>
        <!-- maven-resources-plugin for copying static resources (HTML, CSS, images) -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-resources-plugin</artifactId>
            <version>3.2.0</version>
            <executions>
                <execution>
                    <phase>process-resources</phase>
                    <goals>
                        <goal>copy-resources</goal>
                    </goals>
                    <configuration>
                        <outputDirectory>${project.build.directory}/docs</outputDirectory>
                        <resources>
                            <resource>
                                <directory>src/main/resources</directory>
                                <includes>
                                    <include>**/*</include>
                                </includes>
                            </resource>
                        </resources>
                    </configuration>
                </execution>
            </executions>
        </plugin>

        <!-- maven-jar-plugin for packaging the project into a JAR -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>3.2.0</version>
            <configuration>
                <archive>
                    <manifestEntries>
                        <Main-Class>com.example.MainClass</Main-Class> <!-- Replace with your main class path
-->
                    </manifestEntries>
                </archive>
            </configuration>
        </plugin>
    </plugins>
</build>
```

When You Run `mvn clean package`:

1. **maven-resources-plugin** will copy the static files from `src/main/resources` to the `/docs` folder.
2. **maven-jar-plugin** will package your project's Java classes into a JAR file and add the necessary **Main-Class** entry to the JAR's manifest.

Conclusion:

You don't need to worry about the **maven-resources-plugin** being overridden or replaced. Both plugins work independently, and you can use them together in the same build process to handle different aspects of your project (static resources for GitHub Pages and packaging the Java code into a JAR).

📒 Maven Build Lifecycle Explained : 🏗️

Here's a **clear, concise, and easy-to-understand** documentation on the **Maven Build Lifecycle**:

Introduction

Maven follows a structured build lifecycle, consisting of a sequence of predefined phases. These phases automate tasks like compiling code, running tests, packaging, and deploying the project.

Maven's Three Built-in Lifecycles

1. **Clean Lifecycle** – Cleans the project.
2. **Default (Build) Lifecycle** – Handles project compilation, testing, packaging, and deployment.
3. **Site Lifecycle** – Generates project documentation.

Each lifecycle has a sequence of phases, which execute in order.

---

1. Clean Lifecycle

Used to remove old build files before a new build.

Phases:

- `pre-clean` → Executes tasks before cleaning.
- `clean` → Deletes the `/target` directory (removes compiled files).
- `post-clean` → Executes tasks after cleaning.

Command to Run:

```
mvn clean

```

(This removes all compiled files and resets the build environment.)

---

2. Default (Build) Lifecycle

This is the **main lifecycle** that compiles, tests, and packages the project.

Phases & Explanation:

1. **validate** → Ensures project structure and configuration are correct.
2. **compile** → Compiles the source code.
3. **test** → Runs unit tests using **JUnit/TestNG** (does not require deployment).
4. **package** → Packages the compiled code into a deployable format (e.g., JAR/WAR).
5. **verify** → Runs integration tests to check if the package is valid.
6. **install** → Installs the package into the local repository (`~/.m2/repository`).
7. **deploy** → Uploads the package to a remote repository (e.g., Nexus, Artifactory).

Commands to Run Each Phase:

```
mvn validate     # Check project structure
mvn compile      # Compile the source code
mvn test         # Run unit tests
mvn package      # Create JAR/WAR file
mvn verify       # Run integration tests
mvn install      # Install JAR to local repository
mvn deploy       # Deploy to remote repository

```

> **Note:** Running `mvn package` will also execute **validate, compile, and test** (because earlier phases are executed automatically).

---

3. Site Lifecycle

This lifecycle generates project documentation.

Phases:

- `pre-site` → Prepares documentation.
- `site` → Generates site documentation.
- `post-site` → Finalizes site generation.
- `site-deploy` → Deploys documentation to a web server.

Command to Run:

```
mvn site

```

(This generates a project documentation site inside `target/site`.)

Summary Table:

Lifecycle	Phase	Description
Clean	`clean`	Deletes previous build files.
Build	`validate`	Ensures project correctness.
	`compile`	Compiles Java code.
	`test`	Runs unit tests.
	`package`	Creates JAR/WAR.
	`verify`	Runs integration tests.
	`install`	Installs JAR to local repo.
	`deploy`	Deploys to remote repo.
Site	`site`	Generates documentation.
	`site-deploy`	Deploys documentation.

Conclusion

Maven’s lifecycle ensures an automated, structured build process. Running any phase also executes all previous phases automatically, making builds efficient and repeatable.

For daily use, the most common commands are:

```
mvn clean package   # Clean & build the project
mvn clean install   # Clean, build & install in local repo
mvn deploy          # Deploy to a remote repository

```

Now, you have a **simple and complete** understanding of Maven's lifecycle! 🚀

📌 Maven `site` & `deploy` Commands - Documentation

🚀 1. `mvn site` Command

The `mvn site` command is used to **generate a project website** containing reports like dependencies, build details, test results, and more.

🔹 Steps to Use `mvn site`

📝 Step 1: Add Site Plugin in `pom.xml`

Before running the `site` command, you need to add the **Maven Site Plugin** inside the `<build>` section of your `pom.xml`:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-site-plugin</artifactId>
            <version>3.12.1</version> <!-- Use latest version -->
        </plugin>
    </plugins>
</build>

```

📢 Step 2: Run the Site Command

Once the plugin is added, execute:

```
mvn site

```

✅ What Happens?

- Maven scans your project for available reports.
- Generates an **HTML-based website** inside `target/site/`.
- Includes various reports like dependencies, plugin management, and test results.

🌍 Step 3: Open the Generated Site

After successful execution, open the following file in a browser:

```
D:\Idea Projects\CA-MVN\target\site\index.html

```

🔎 You’ll See Reports Like:

✔ Project Summary
✔ Dependencies Report
✔ Plugin Management
✔ Unit Test Results (if configured)
✔ Code Coverage (if applicable)

📤 2. `mvn deploy` Command

The `mvn deploy` command is used to **upload the built artifact (JAR, POM, etc.)** to a repository for distribution and sharing.

🔹 Steps to Use `mvn deploy`

Since you don't have a remote repository, we will configure a **local repository**.

📝 Step 1: Create a Local Repository

```
mkdir D:\my-local-maven-repo
```

🔧 Step 2: Configure `pom.xml` for Local Deployment

Add the following inside `<project>` in `pom.xml`:

```
<distributionManagement>
    <repository>
        <id>local-repo</id>
        <url>file:///D:/my-local-maven-repo</url>
    </repository>
</distributionManagement>
```

📦 Step 3: Run the Deploy Command

```
mvn deploy
```

✅ What Happens?

- Maven **builds** the project.
- Stores the artifact (JAR, POM, etc.) in `D:/my-local-maven-repo`.

📁 Step 4: Verify Deployment

Navigate to `D:/my-local-maven-repo/` and check if the project is stored correctly:

```
D:/my-local-maven-repo/
├── org/example/CA-MVN/1.0-SNAPSHOT/
│   ├── CA-MVN-1.0-SNAPSHOT.jar
│   ├── CA-MVN-1.0-SNAPSHOT.pom
```

🎯 Conclusion

✔ Use `mvn site` to generate a project website with reports.
✔ Use `mvn deploy` to store artifacts in a local or remote repository.

✔ Ensure you configure the **Maven Site Plugin** and **Distribution Management** in `pom.xml` before running these commands.

Now, you're all set to **document and deploy** your Maven project efficiently! 🚀🎯

✅ **Optional : Adding to Remote Repository || Out Of Syllabus || Just For Information**

You **can deploy your Maven artifacts** (JARs, POMs, etc.) to **GitHub Packages** as a remote repository. 🚀

GitHub Packages acts as a **private Maven repository**, and you can deploy artifacts using **Maven Deploy Plugin** with authentication.

🛠️ Steps to Deploy a Maven Project to GitHub Packages

🔹 1. Create a GitHub Repository

- Go to **GitHub → Create a new repository**
- Name it something like `maven-repo`
- **DO NOT initialize** with a `README`, `.gitignore`, or license.
- Copy your **GitHub Username** and **Personal Access Token (PAT)** (for authentication).

🔧 2. Modify `pom.xml` for GitHub Deployment

Add the **GitHub repository** under `<distributionManagement>` in your `pom.xml`:

```
<distributionManagement>
    <repository>
        <id>github</id>
        <url>https://maven.pkg.github.com/YOUR_GITHUB_USERNAME/maven-repo</url>
    </repository>
</distributionManagement>

```

✅ **Replace** `YOUR_GITHUB_USERNAME` with your actual GitHub username.

🔐 3. Configure Authentication (`settings.xml`)

You need to add **GitHub authentication credentials** inside Maven's `settings.xml` file (found in `C:\Users\YourUser\.m2\` on Windows or `~/.m2/` on Linux/macOS).

Add the following inside `<servers>`:

```
<server>
    <id>github</id>
    <username>YOUR_GITHUB_USERNAME</username>
    <password>YOUR_GITHUB_PERSONAL_ACCESS_TOKEN</password>
</server>

```

✅ **Replace** `YOUR_GITHUB_PERSONAL_ACCESS_TOKEN` with a **GitHub Personal Access Token (PAT)** that has `read:packages` and `write:packages` permissions.

📦 4. Run the Deploy Command

Now, deploy your Maven project using:

```
mvn deploy

```

✅ What Happens?

- Maven builds the project.
- Uploads the artifact (`JAR`, `POM`, etc.) to **GitHub Packages**.
- You can now **use this package in other projects!** 🎯

📥 5. Use Your GitHub Package in Another Maven Project

To **use the deployed package in another project**, add this to the new project's `pom.xml`:

```
<repositories>
    <repository>
        <id>github</id>
        <url>https://maven.pkg.github.com/YOUR_GITHUB_USERNAME/maven-repo</url>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>com.example</groupId>
        <artifactId>YOUR_ARTIFACT_NAME</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
</dependencies>

```

Now, your Maven project will **download the dependency from GitHub Packages** instead of Maven Central. 🎯

🚀 Conclusion

✔ You **can deploy** Maven artifacts to **GitHub Packages**.
✔ Requires **GitHub authentication** (username + PAT).
✔ Can be **used in other projects** like a private Maven repository.

CA - Gradle Notes & Documentation

Introduction to Gradle 🌟

Gradle is a **powerful open-source build automation tool** used for **developing, testing, deploying, and publishing** software applications. It was created by **Hans Dockter, Szczepan Faber, Adam Murdoch, Luke Daley, Peter Niederwieser, Daz DeBoer, and Rene Gröschke** around **13 years ago** as an improvement over **Apache Ant and Apache Maven**. 🚀

Unlike its predecessors, **Gradle supports multiple programming languages** including **Java, Kotlin, Groovy, Scala, and C++**, making it a **versatile** and **industry-standard build tool**. It is used for tasks ranging from **compilation and dependency management** to **packaging and deployment**.

🌍 History of Gradle

Gradle was first introduced in **late 2007** as a **more stable and flexible alternative** to **Apache Ant and Maven**. It addressed many of their limitations while **enhancing performance and scalability**.

- Its **first stable version** was released in **2019**.
- The latest version (as of the last update) is **Gradle 6.6**.

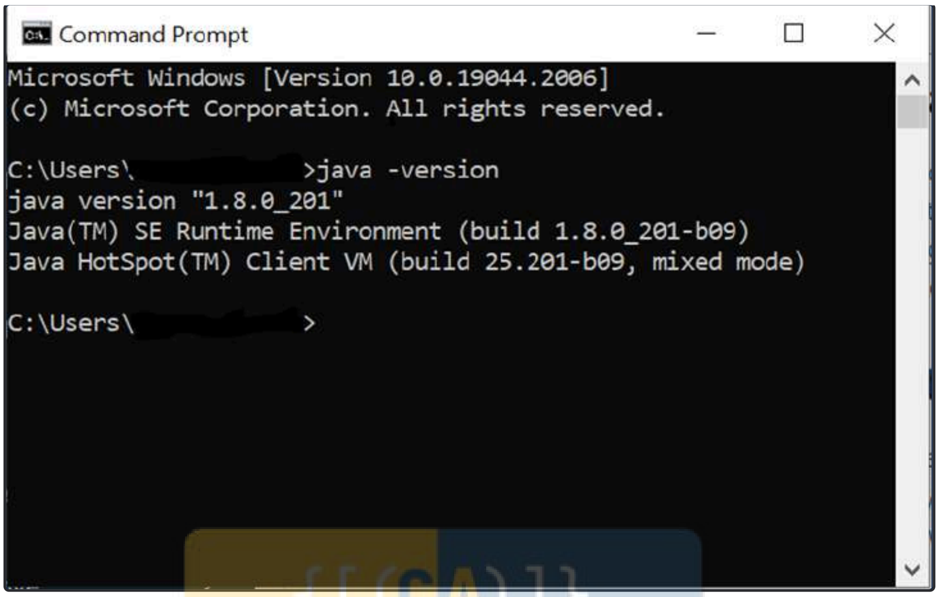
Gradle has now become a **popular choice for developers** due to its **speed, flexibility, and powerful dependency management system**.

🏗 How to Install Gradle on Windows?

Gradle is a flexible build automation tool to build software. It is open-source, and also capable of building almost any type of software. A build automation tool automates the build process of applications. The Software building process includes compiling, linking, and packaging the code, etc. Gradle makes the build process more consistent. Using Gradle we can build **Android, Java, Groovy, Kotlin JVM, Scala, C++, Swift** Libraries, and Applications.

Prerequisites to Install Gradle

To install Gradle, we have to make sure that we have Java JDK version 8 or higher to be installed on our operating system because Gradle runs on major operating systems only Java Development Kit version 8 or higher. To confirm that we have JDK installed on our system. The '**Java -version** 'command is used to verify if you run it on a windows machine.



```
Microsoft Windows [Version 10.0.19044.2006]
(c) Microsoft Corporation. All rights reserved.

C:\Users\[redacted]>java -version
java version "1.8.0_201"
Java(TM) SE Runtime Environment (build 1.8.0_201-b09)
Java HotSpot(TM) Client VM (build 25.201-b09, mixed mode)

C:\Users\[redacted]>
```

You should see output like in the above image it confirms that you have JDK installed and the JAVA_HOME path set properly if you don't have JDK Install. then you should install it first before going to the next step. in this tutorial, we are not covering the installation of JDK. we consider that we have JDK already on the system.

Installing Gradle Manually

Gradle Enterprise provides the installation of Gradle with package manager SDKMAN! but here we are covering the manual installation of Gradle for Microsoft windows machines. before starting we confirmed we have JDK installed properly on the machine.

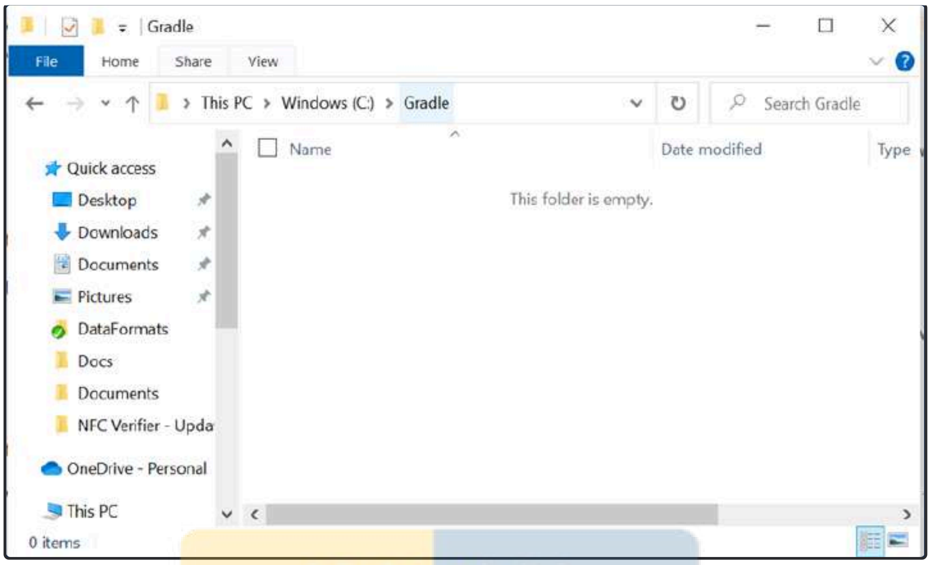
Step 1. Download the latest Gradle distribution

To install Gradle we need the Gradle distribution ZIP file, we can download the latest version of Gradle from the official website. there are other versions of Gradle available but we will strongly suggest using the official. The current release of Gradle is version v8.12.1, released on Jan 24, 2025. it might be another latest release available when you read this article. Always download the latest version. The Gradle distribution zip file is available in two flavors:

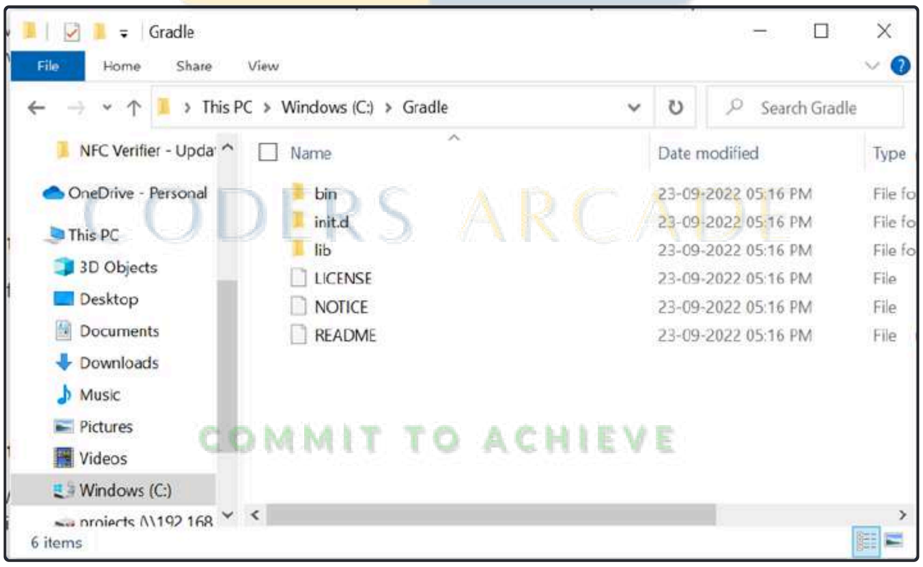
1. Binary-only: Binary version contains only executable files no source code and documentation. You can browse docs, and sources online.
2. Complete, with docs and sources: This package contains docs and sources offline. I recommend going with binary-only.

Step 2. Unpack the Gradle distribution zip file

After successfully downloading the Gradle distribution ZIP file, we need to unzip/unpack the compressed file. In File Explorer, create a new directory **C:\Gradle** (you can choose any path according to your choice) as shown In the image.



Now unpack the Gradle distribution ZIP into **C:\Gradle** using an archiver tool of your choice.



Name	Date modified	Type
bin	23-09-2022 05:16 PM	File fo
init.d	23-09-2022 05:16 PM	File fo
lib	23-09-2022 05:16 PM	File fo
LICENSE	23-09-2022 05:16 PM	File
NOTICE	23-09-2022 05:16 PM	File
README	23-09-2022 05:16 PM	File

Step 3. Configure your system environment

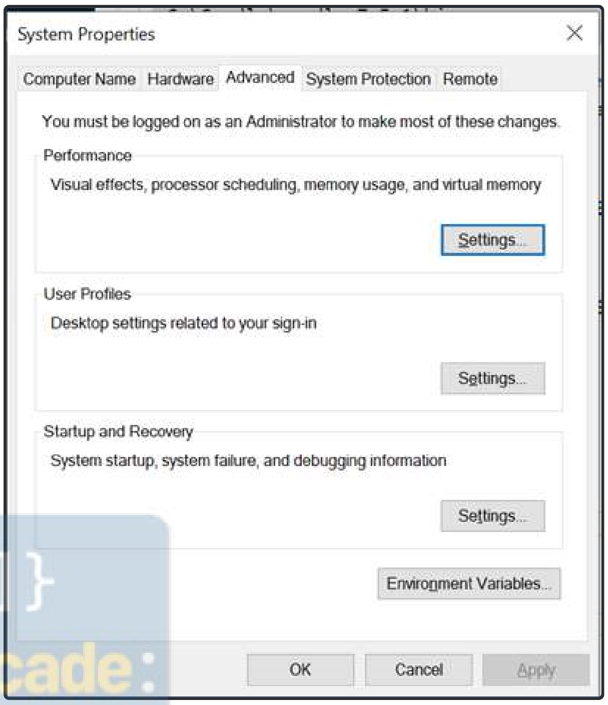
We have successfully unzipped the Gradle distribution Zip file. Now set the System environment **PATH** variable to the bin directory of the unzipped distribution. We have extracted files in the following location. you can choose any location to unzip:

C:\Gradle\bin

To set the Environment variable, Right-click on the “**This PC /Computer”** icon, then select **Properties -> Advanced System Settings -> Environmental Variables.** The following screenshot may help you.

As shown in the following image, In the section **System Variables** select **Path**, then select the **Edit** option.

You will now see the following screen Select **New,** then add an entry for **C:\Gradle\bin** as shown in the image, now click on the **OK** button to save changes.



You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage, and virtual memory

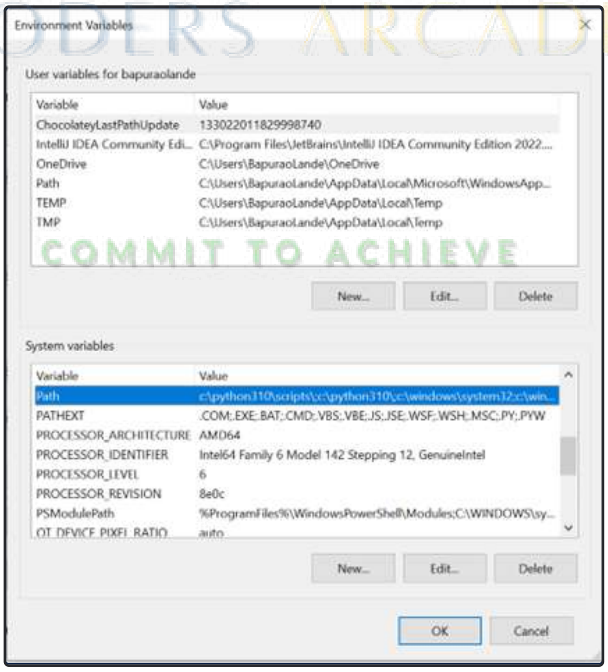
User Profiles
Desktop settings related to your sign-in

Startup and Recovery
System startup, system failure, and debugging information

Step 4. Verifying and confirming the installation

After the successful setting of the environment variable, now it's time to verify the installation. Run the "**gradle -v"** command on the Windows command prompt, which displays the details of the Gradle version installed on the PC. If you get

similar output as in the image, Gradle has been installed successfully, and you can now use Gradle in your projects.

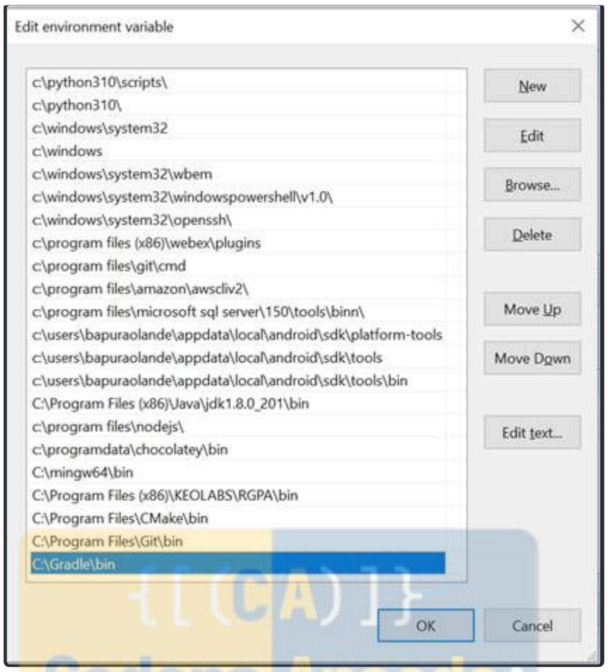


User variables for bapuraolande

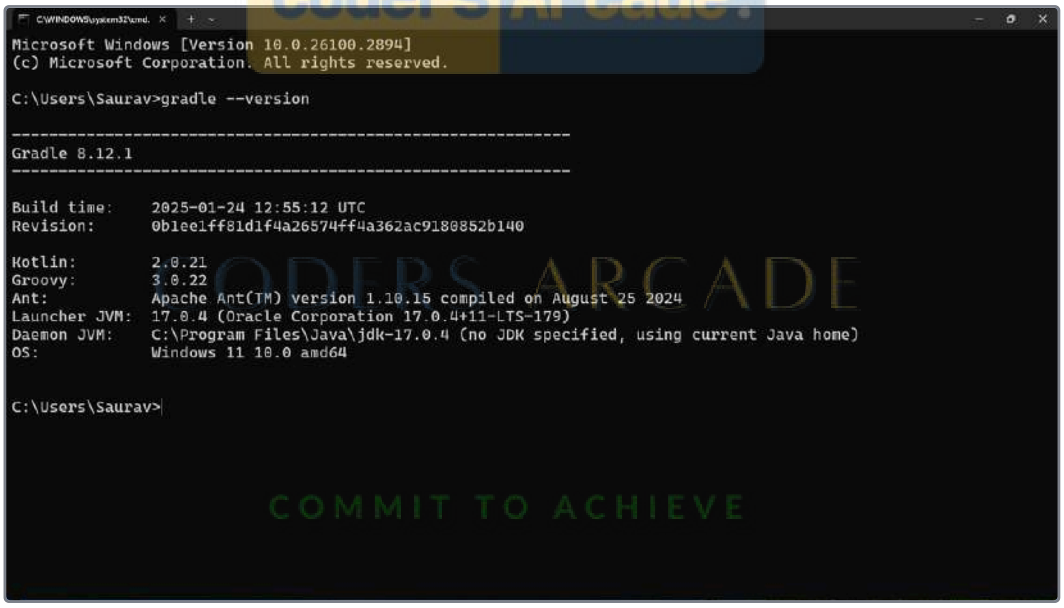
Variable	Value
ChocolateyLastPathUpdate	133022011829998740
IntelliJ IDEA Community Edi...	C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022....
OneDrive	C:\Users\BapuraoLande\OneDrive
Path	C:\Users\BapuraoLande\AppData\Local\Microsoft\WindowsApp...
TEMP	C:\Users\BapuraoLande\AppData\Local\Temp
TMP	C:\Users\BapuraoLande\AppData\Local\Temp

System variables

Variable	Value
Path	c:\python310\scripts\;c:\python310\;c:\windows\system32;c:\win...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC;.PY;.PYW
PROCESSOR_ARCHITECTURE	AMD64
PROCESSOR_IDENTIFIER	Intel64 Family 6 Model 142 Stepping 12, GenuineIntel
PROCESSOR_LEVEL	6
PROCESSOR_REVISION	8e0c
PSModulePath	%ProgramFiles%\WindowsPowerShell\Modules;C:\WINDOWS\sy...
QT_DEVICE_PIXEL_RATIO	auto



```
c:\python310\scripts\
c:\python310\
c:\windows\system32
c:\windows
c:\windows\system32\wbem
c:\windows\system32\windowspowershell\v1.0\
c:\windows\system32\openssh\
c:\program files (x86)\webex\plugins
c:\program files\git\cmd
c:\program files\amazon\awscliv2\
c:\program files\microsoft sql server\150\tools\binn\
c:\users\bapuraolande\appdata\local\android\sdk\platform-tools
c:\users\bapuraolande\appdata\local\android\sdk\tools
c:\users\bapuraolande\appdata\local\android\sdk\tools\bin
C:\Program Files (x86)\Java\jdk1.8.0_201\bin
c:\program files\nodejs\
c:\programdata\chocolatey\bin
C:\mingw64\bin
C:\Program Files (x86)\KEOLABS\RGPA\bin
C:\Program Files\CMake\bin
C:\Program Files\Git\bin
C:\Gradle\bin
```



```
Microsoft Windows [Version 10.0.26100.2894]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Saurav>gradle --version

------------------------------------------------------------
Gradle 8.12.1
------------------------------------------------------------

Build time:    2025-01-24 12:55:12 UTC
Revision:      0b1ee1ff81d1f4a26574ff4a362ac9180852b140

Kotlin:        2.0.21
Groovy:        3.0.22
Ant:           Apache Ant(TM) version 1.10.15 compiled on August 25 2024
Launcher JVM:  17.0.4 (Oracle Corporation 17.0.4+11-LTS-179)
Daemon JVM:    C:\Program Files\Java\jdk-17.0.4 (no JDK specified, using current Java home)
OS:            Windows 11 10.0 amd64


C:\Users\Saurav>
```

🛠️ How Gradle Works?

A **Gradle project** consists of **one or more subprojects**, and each project is made up of **tasks**. Let's break it down:

📌 Gradle Projects

- A **Gradle project** can be a **web application, JAR file, or even a collection of scripts**.
- Each project is made up of **tasks** that perform specific actions, such as **compiling, testing, and packaging**.
- Think of a **Gradle project like a wall**, where **each brick (task) builds up the structure**.

📌 Gradle Tasks

Tasks in Gradle define **what needs to be executed** in a project.
There are **two main types of tasks**:

1️⃣ Default Tasks:

- These are **predefined by Gradle** and **execute automatically** when no specific task is mentioned.
- Example: `init` and `wrapper` tasks.

2️⃣ **Custom Tasks:**

- These are **user-defined tasks** that allow developers to **customize** the build process.
- Example: Let's create a simple **Gradle task** that prints a message:

```
// build.gradle
task hello {
    doLast {
        println 'Welcome to Gradle!'
    }
}

```

💻 **Running the task:**

```
gradle -q hello

```

🖥️ **Output:**

```
Welcome to Gradle!

```

✨ Features of Gradle

Gradle comes with a variety of **powerful features** that make it a preferred build tool for developers. 🚀

🔹 **IDE Support** 🖥️

- Works with **IntelliJ IDEA, Eclipse, NetBeans, and Visual Studio**.

🔹 **Java-Friendly** ☕

- Gradle projects run on **JVM (Java Virtual Machine)** and support **Java-based APIs**.

🔹 **Tasks & Repository Support** 📦

- Supports **Maven** and **Ant repositories**.
- Allows easy **integration** with existing **Maven** and **Ant** projects.

🔹 **Incremental Builds** ⚡

- **Only compiles changes** made after the last build, reducing build time significantly.

🔹 **Dependency Management** 🔄

- Automatically **resolves and downloads dependencies**.
- Works with **Maven and Ivy repositories**.

🔹 **Build Caching** 🔁

- Stores results from previous builds and **reuses them**, reducing build times.

🔹 **Plugin System** 🔌

- Supports **plugins** for various languages (**Java, Kotlin, Scala, C++, Android, and more**).

🔹 **Extensibility** 🚀

- Highly **customizable** with **user-defined** tasks and configurations.

🔹 **Continuous Integration Support** 🏗️

- Works seamlessly with **Jenkins, GitHub Actions, Travis CI, and Azure DevOps**.

🔹 **Multi-Project Support** 🏗️

- Allows **managing multiple projects within the same build**.

💡 Pros & Cons of Using Gradle

✅ Pros

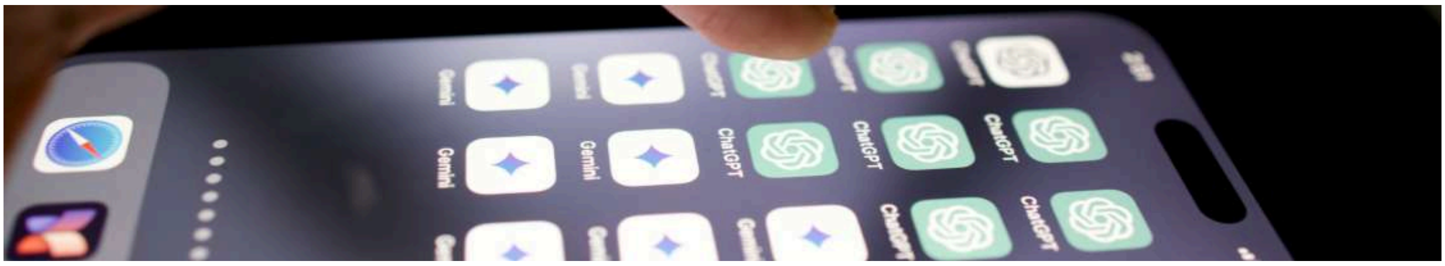
✔ **Declarative Builds** – Uses **Groovy/Kotlin DSL** for configuration.
✔ **Highly Scalable** – Efficient for both **small** and **large projects**.
✔ **Performance Boost** – **Incremental builds** and **parallel execution** speed up the process.
✔ **Cross-Language Support** – Works with **Java, Groovy, Kotlin, Scala, C++**, etc.
✔ **Strong Dependency Management** – Handles **dependencies** efficiently.
✔ **Free & Open Source** – Available under **Apache License 2.0**.

❌ Cons

❗ **Requires Technical Knowledge** – Needs prior understanding of **Groovy/Kotlin DSL**.
❗ **Complex Configurations** – Initial **setup can be tricky** for beginners.
❗ **Resource Consumption** – **Uses more memory** compared to **Maven**.
❗ **Migration Difficulty** – Moving from **Maven/Ant to Gradle** requires effort.

📚 Conclusion

Gradle is a **powerful, flexible, and efficient** build automation tool. It has become a **developer-favorite** for projects of all sizes, from **small applications to large enterprise systems**. With **incremental builds, dependency management, and extensive plugin support**, Gradle makes software development **faster and smoother**. 🚀



CA - Experiment 3 Part 1 - Working with Gradle: Setting Up a Gradle Project, Understanding Build Scripts (Groovy and Kotlin DSL), Dependency Management and Task Automation

Here's the **full step-by-step guide** for working with **Gradle in IntelliJ IDEA**, including **setup, project creation, website deployment, Selenium testing, and JAR packaging**.

🚀 Gradle in IntelliJ IDEA: A Complete Guide

1️⃣ Introduction to Gradle

Gradle is a powerful build automation tool used for **Java, Kotlin, and Android projects**. It provides **fast builds, dependency management, and flexibility** using Groovy/Kotlin-based scripts.

🔹 **Why Use Gradle?**

- **Incremental builds** (faster execution)
- **Dependency management** (supports Maven/Ivy)
- **Customizable tasks**
- **Multi-project support**

2️⃣ Installing and Setting Up Gradle in IntelliJ IDEA (Already Covered In Experiment 1)

Step 1: Install Gradle

- **Windows** (using Chocolatey):

```
choco install gradle

```

- **macOS** (using Homebrew):

```
brew install gradle

```

- **Linux**:

```
sdk install gradle

```

Step 2: Verify Installation

```
gradle -v
gradle --version
```

This should display the Gradle version.

3️⃣ Creating a Gradle Project in IntelliJ IDEA

Step 1: Open IntelliJ IDEA and Create a New Project

1. Click on **"New Project"**.
2. Select **"Gradle"** (under Java/Kotlin).
3. Choose **Groovy or Kotlin DSL (Domain Specific Language)** for the build script.
4. Set the **Group ID** (e.g., `com.example`).
5. Click **Finish**.

Step 2: Understanding Project Structure

```
my-gradle-project
|── build.gradle  (Groovy Build Script)
|── settings.gradle
|── src
|   ├── main
|   |   ├── java
|   |   └── resources
|   ├── test
|   |   ├── java
|   |   └── resources

```

4️⃣ Build and Run a Simple Java Application

Step 1: Modify `build.gradle` (Groovy DSL)

```
plugins {
    id 'application'
}

repositories {
    mavenCentral()
}

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter:5.8.1'
}

application {
    mainClass = 'com.example.Main'
}

```

Step 2: Create `Main.java` in `src/main/java/com/example`

```java
package com.example;

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello from Gradle!");
    }
}

```

Step 3: Build and Run the Project

- **In IntelliJ IDEA**, open the **Gradle tool window** (View → Tool Windows → Gradle).
- Click `Tasks > application > run`.
- Or run from terminal:

```
gradle run

```

5️⃣ Hosting a Static Website on GitHub Pages

Step 1: Create a `/docs` Directory

- Create `docs` inside the **root folder** (not in `src`).
- Add your **HTML, CSS, and images** inside `/docs`.

Step 2: Modify `build.gradle` to Copy Website Files (This is optional)

```
task copyWebsite(type: Copy) {
    from 'src/main/resources/website'
    into 'docs'
}

```

Step 3: Commit and Push to GitHub

```
git add .
git commit -m "Deploy website using Gradle"
git push origin main

```

Step 4: Enable GitHub Pages

- Go to **GitHub Repo → Settings → Pages**.
- Select the `/docs` **folder** as the source.

Your website will be hosted at:

```
https://yourusername.github.io/repository-name/

```

6️⃣ Testing the Website using Selenium & TestNG in IntelliJ IDEA

Step 1: Add Selenium & TestNG Dependencies in `build.gradle`

```
dependencies {
    testImplementation 'org.seleniumhq.selenium:selenium-java:4.28.1' // use the latest stable version
    testImplementation 'org.testng:testng:7.4.0' // use the latest stable version
}

test {
    useTestNG()
}

```

Step 2: Write a Test Script (`src/test/java/org/test/WebpageTest.java` **)**

```
package org.test;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;

import static org.testng.Assert.assertTrue;

public class WebpageTest {
    private static WebDriver driver;

    @BeforeTest
    public void openBrowser() throws InterruptedException {
        driver = new ChromeDriver();
        driver.manage().window().maximize();
        Thread.sleep(2000);
        driver.get("https://sauravsarkar-codersarcade.github.io/CA-GRADLE/");
    }

    @Test
    public void titleValidationTest(){
        String actualTitle = driver.getTitle();
        String expectedTitle = "Tripillar Solutions";
        Assert.assertEquals(actualTitle, expectedTitle);
        assertTrue(true, "Title should contain 'Tripillar'");

    }

    @AfterTest
    public void closeBrowser() throws InterruptedException {
        Thread.sleep(1000);
        driver.quit();
    }


}

```

Step 3: Run the Tests

- Open the **Gradle tool window** in IntelliJ.
- Click `Tasks > verification > test`. **“Recommended“**
- Or run from terminal:

```
gradle test // Fails sometimes due to terminal issues

```

7️⃣ Packaging a Gradle Project as a JAR

Step 1: Modify `build.gradle` for JAR Packaging

```
plugins {
    id 'java'
    id 'application'
}

application {
    mainClass = 'com.example.Main'
}
jar {
    manifest {
        attributes 'Main-Class': 'com.example.Main'  // This tells Java where to start execution
    }
}

```

Step 2: Build and Package the JAR

```
gradle jar

```

The JAR file will be generated in `build/libs/`.

Step 3: Run the JAR

```
java -jar build/libs/<my-gradle-project>.jar

Expected output:
Hello from Gradle!

```

8️⃣ Gradle Lifecycle & Common Commands

Task	Command	Description
Initialize Project	`gradle init`	Creates a new Gradle project.
Compile Code	`gradle build`	Compiles the project.
Run Application	`gradle run`	Runs the application.
Clean Build Files	`gradle clean`	Deletes old build files.

Run Tests	`gradle test`	Executes unit tests.
Generate JAR	`gradle jar`	Packages the project into a JAR.
Deploy to GitHub Pages	`git push origin main`	Pushes website to GitHub.

9️⃣ Conclusion

Gradle provides a **fast, flexible, and scalable build automation system**.

We covered:

1️⃣ **Project Setup** in IntelliJ IDEA
2️⃣ **Building & Running a Java Application**
3️⃣ **Hosting a Static Website on GitHub Pages**
4️⃣ **Testing with Selenium & TestNG**
5️⃣ **Packaging as a JAR**

✅ Now you are **ready to use Gradle for build automation!** 🚀

🌟 Gradle Build Lifecycle: A Simple & Colorful Guide 🚀

Gradle follows a **flexible, task-based lifecycle**, unlike Maven's fixed phases. The build lifecycle in Gradle is divided into three key stages:

🔹 1. Initialization Phase

📌 **What happens here?**

- Determines which projects are part of the build (for multi-project builds).
- Creates an instance of each project.

📌 **Example Command:**

```
gradle help

```

(Shows project details and verifies Gradle setup.)

🔹 2. Configuration Phase

📌 **What happens here?**

- Gradle loads and executes the `build.gradle` file.
- It **configures** tasks but does **not** execute them yet.

📌 **Example Command:**

```
gradle tasks

```

(Lists all available Gradle tasks in the project.)

🔹 **3. Execution Phase**

📌 **What happens here?**

- Gradle executes the **tasks requested by the user**.
- Dependencies are resolved dynamically.
- Supports **incremental builds** (only modified files are compiled).

📌 **Example Command:**

```
gradle build

```

(Builds the project by compiling and packaging files.)

🎯 Key Gradle Tasks & Commands

Task 🛠️	Command	Description
🧹 **Clean**	`gradle clean`	Deletes previous build files.
🛠️ **Compile**	`gradle compileJava`	Compiles the Java source code.
📦 **Package (JAR/WAR)**	`gradle jar`	Packages the project into a JAR file.
✅ **Test**	`gradle test`	Runs unit tests.
🚀 **Run Application**	`gradle run`	Executes the main Java application.
📥 **Dependency Resolution**	`gradle dependencies`	Displays dependency tree.
⏩ **Parallel Execution**	`gradle build --parallel`	Speeds up builds by running tasks in parallel.

🎨 Gradle Lifecycle Visualized

```
Initialization  →  Configuration  →  Execution
(Project setup)    (Tasks loaded)    (Tasks executed)

```

✅ **Gradle is flexible** – tasks can be **customized or skipped** based on project needs. Unlike Maven, Gradle allows **on-demand task execution** rather than following a strict lifecycle.

🌟 Conclusion

🎯 **Gradle is powerful because:**
✅ It **only executes what's necessary** (faster builds).
✅ It **supports parallel execution** for large projects.
✅ It gives developers **more control over task execution**.

With Gradle, you can **automate builds efficiently** and **improve performance** for Java, Kotlin, and Android projects. 🚀

🔄 Maven vs. Gradle: A Detailed Comparison

Feature	Maven 🏗️	Gradle 🚀
Build Script Language	XML (`pom.xml`)	Groovy/Kotlin (`build.gradle` or `build.gradle.kts`)
Performance	Slower due to full rebuilds	Faster with **incremental builds** and parallel execution
Dependency Management	Uses Maven Central & local repository	Supports Maven, Ivy, and custom repositories
Configuration Style	**Declarative** (XML-based, verbose)	**Declarative + Imperative** (more concise, script-based)
Ease of Use	More structured, but verbose	More flexible, but has a learning curve
Incremental Builds	No support (always rebuilds everything)	Supports **incremental builds** (only recompiles changed files)
Build Performance	Slower, builds from scratch	Faster due to task caching and incremental execution
Customization & Extensibility	Limited custom scripts, plugin-based	Highly customizable with dynamic tasks and scripts
Multi-Project Builds	Complex and slower	Native support, optimized for large projects
IDE Support	Supported by IntelliJ IDEA, Eclipse, VS Code	Supported by IntelliJ IDEA, Eclipse, VS Code
Dependency Resolution	Resolves dependencies sequentially	**Parallel dependency resolution** (faster)
Popularity	Older and widely used in enterprise projects	Gaining popularity, especially in **Android & Kotlin** projects
Default Lifecycle Phases	3 lifecycle phases (clean, build, site)	More flexible task execution model
Learning Curve	Easier for beginners due to structured XML	Slightly steeper due to script-based configuration
Best Suited For	**Stable Java projects**, enterprises, and legacy codebases	**Modern Java, Android, and Kotlin projects** needing high performance

🛠️ Which One Should You Choose?

✅ **Use Maven** if you need **stability, structured builds, and an easier learning curve**.
✅ **Use Gradle** if you want **faster builds, better performance, and flexibility** for complex projects.

🎯 **Final Verdict:** If you're working on a simple Java project, **Maven** is a good choice. However, for modern, large-scale, or Android projects, **Gradle** is the better option due to its **speed, flexibility, and scalability**. 🚀

Here is a KOTLIN DSL version for Gradle "This is Optional"

🎯 Gradle Kotlin Build Script (`build.gradle.kts`)

Here's a **simple Gradle build script** written in **Kotlin DSL (** `build.gradle.kts` **)** for a Java application. 🚀

📌 `build.gradle.kts` (Kotlin DSL for Gradle)

```
// Apply necessary plugins
plugins {
    kotlin("jvm") version "1.9.10"  // Kotlin plugin for JVM
    application  // Enables the `run` task
}

// Define project properties
group = "com.example"
version = "1.0.0"
application.mainClass.set("com.example.MainKt")  // Define the main class

// Repositories for dependencies
repositories {
    mavenCentral()  // Fetch dependencies from Maven Central
}

// Dependencies
dependencies {
    implementation(kotlin("stdlib"))  // Standard Kotlin library
    testImplementation("org.junit.jupiter:junit-jupiter:5.9.1")  // JUnit 5 for testing
}

// Enable JUnit 5 for tests
tasks.test {
    useJUnitPlatform()
}

// JAR packaging
tasks.jar {
    manifest {
        attributes["Main-Class"] = "com.example.MainKt"
    }
}

```

📌 `Main.kt` (Inside `src/main/kotlin/com/example/`)

```
package com.example

fun main() {
    println("Hello, Gradle with Kotlin DSL! 🚀")
}

```

📌 Running the Project

💻 Run the application

```
gradle run

```

📦 Build the project

```
gradle build

```

🧹 Clean the build files

```
gradle clean

```

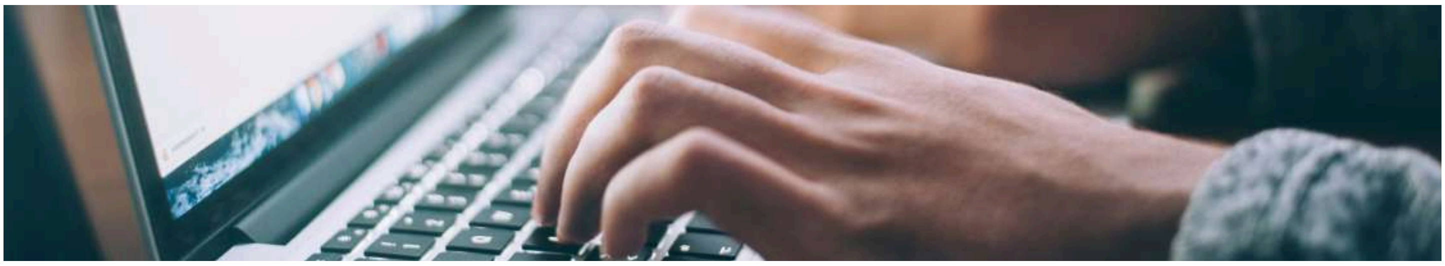
📂 Generate a JAR file

```
gradle jar

```

🌟 Why Use Kotlin DSL for Gradle?

✅ **Type-Safety** - Catch errors at compile-time.
✅ **Better IDE Support** - IntelliJ IDEA provides autocomplete and suggestions.
✅ **Interoperability** - Works seamlessly with Java and Kotlin projects.

This script sets up a **basic Kotlin/Java project with JUnit 5 support, dependency management, and a runnable JAR file**! 🚀



CA - Experiment 2 Part 2 : GRADLE KOTLIN DSL WORKFLOW || IntelliJ Idea

Here's a **detailed and well-formatted** documentation Gradle Kotlin DSL setup, including explanations, code snippets, and color-coded syntax (for reference when viewing in an IDE).

⚠️ **Important Note** : This experiment is only required if **students** or **VTU** asks you to show **GRADLE** with **KOTLIN DSL**, or else you can skip this.

Gradle Kotlin DSL: Setting Up & Building a Kotlin Project in IntelliJ IDEA

1️⃣ Setting Up the Gradle Project

Step 1: Create a New Project

1. Open **IntelliJ IDEA**.
2. Click on **File > New > Project**.
3. Select **Gradle** as the build system.
4. Choose **Kotlin** as the language.
5. Select **Gradle Kotlin DSL** (it will generate `build.gradle.kts`).
6. Name your project (e.g., `MVNGRDKOTLINDEMO`).
7. Set the **JDK** (use **JDK 17.0.4**, since that's your version).
8. Click **Finish**.

2️⃣ Understanding `build.gradle.kts`

After creating the project, the default `build.gradle.kts` file looks like this:

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile

plugins {
    kotlin("jvm") version "1.8.10"  // Use latest stable Kotlin version
    application
}

group = "org.example"
version = "1.0-SNAPSHOT"

```

```
repositories {
    mavenCentral()
}

dependencies {
    implementation(kotlin("stdlib"))  // Kotlin Standard Library
    testImplementation("org.junit.jupiter:junit-jupiter-api:5.8.2")
    testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine:5.8.2")
}

tasks.test {
    useJUnitPlatform()
}

tasks.withType<KotlinCompile> {
    kotlinOptions.jvmTarget = "17"  // Match with your JDK version
}

application {
    mainClass.set("MainKt")  // Update this if using a package
}

```

3️⃣ Creating the Main Kotlin File

Now, create your `Main.kt` file inside `src/main/kotlin/`.
If you're using a package (e.g., `org.example`), it should look like:

```
package org.example

fun main() {
    println("Hello, Gradle with Kotlin DSL!")
}

```

If you're **not** using a package, remove the `package` line and ensure `mainClass.set("MainKt")` in `build.gradle.kts`.

4️⃣ Building and Running the Project

Build the Project

```
./gradlew build

```

Run the Project

```
./gradlew run

```

5️⃣ Packaging as a JAR

To run the project without IntelliJ, we need a **JAR file**.

Step 1: Create a Fat (Uber) JAR

Modify `build.gradle.kts`:

```
tasks.register<Jar>("fatJar") {
    archiveClassifier.set("all")
    duplicatesStrategy = DuplicatesStrategy.EXCLUDE
    manifest {
        attributes["Main-Class"] = "MainKt"
    }
    from(configurations.runtimeClasspath.get().map { if (it.isDirectory) it else zipTree(it) })
    with(tasks.jar.get() as CopySpec)
}
```

Step 2: Build the Fat JAR

```
./gradlew fatJar
```

Step 3: Run the Fat JAR

```
java -jar build/libs/MVNGRDKOTLINDEMO-1.0-SNAPSHOT-all.jar
```

6️⃣ Common Gradle Commands

Command	Description
`./gradlew build`	Builds the project
`./gradlew run`	Runs the application
`./gradlew test`	Runs the tests
`./gradlew clean`	Cleans the build directory
`./gradlew fatJar`	Creates a runnable JAR

7️⃣ Additional Features

Adding Custom Gradle Tasks

Example: A simple task that prints "Hello, Gradle!"

```
tasks.register("hello") {
    doLast {
        println("Hello, Gradle!")
    }
}
```

Run it with:

```
./gradlew hello
```

```

```

8️⃣ Troubleshooting & Fixes

1. `java.lang.NoClassDefFoundError: kotlin/jvm/internal/Intrinsics`

✅ **Fix**: Use a Fat JAR (`fatJar`) to include Kotlin dependencies.

2. `mainClass` **Not Found**

✅ **Fix**: Ensure `mainClass.set("MainKt")` or use the correct package (`org.example.MainKt`).

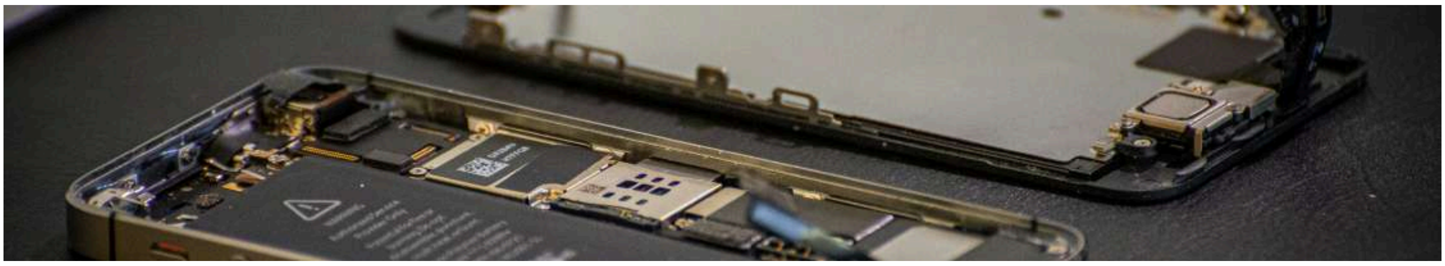
3. Gradle Wrapper Missing

✅ **Fix**: Generate it using:

```
gradle wrapper

```

✅ You're all set! 🚀

This guide ensures you have a **fully working** Gradle project with Kotlin DSL, dependency management, custom tasks, and a runnable JAR. 😊



CA - Experiment 4 - Practical Exercise: Build and Run a Java Application with Maven, Migrate the Same Application to Gradle

🎯 Part 1: Create and Build a Java Application with Maven

🛠️ Step 1: Create a Maven Project in IntelliJ IDEA

1. **Open IntelliJ IDEA**
   - Launch **IntelliJ IDEA** and click on **File** → **New** → **Project**. 🌟
2. **Select Maven**
   - In the **New Project** window, choose **Maven** from the options on the left.
   - Check **Create from archetype** and select **maven-archetype-quickstart**.
   - Click **Next**. 👨‍💻
3. **Enter Project Details**
   - **GroupId**: `com.example`
   - **ArtifactId**: `MVNGRDLDEMO`
   - Click **Next** and then **Finish**. ✨
4. **Wait for IntelliJ to Load Dependencies**
   - IntelliJ will automatically download the Maven dependencies, so just relax for a moment. 😊

📝 Step 2: Update `pom.xml` to Add Build Plugin

To compile and package your project into a `.jar` file, you need to add the **Maven Compiler** and **Jar** plugins. 🔧

1. Open the `pom.xml` file. 📄
2. Add the following inside the `<project>` tag:

```
<build>
    <plugins>
        <!-- Compiler Plugin -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.1</version>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>

        <!-- Jar Plugin -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>3.2.0</version>
            <configuration>
                <archive>
                    <manifest>
                        <mainClass>com.example.App</mainClass>
                    </manifest>
                </archive>
            </configuration>
        </plugin>
    </plugins>
</build>

```

🔨 Step 3: Build and Run the Maven Project

1. **Open IntelliJ IDEA Terminal**
   Press **Alt + F12** to open the terminal. 🖥️
2. **Compile and Package the Project**
   Run the following commands to build the project:

```
mvn clean compile
mvn package

```

3. **Locate the JAR File**
   After running the above, your `.jar` file will be located at:

```
D:\Idea Projects\MVNGRDLDEMO\target\MVNGRDLDEMO-1.0-SNAPSHOT.jar

```

4. **Run the JAR File**
   To run the generated JAR file, use:

```
java -jar target\MVNGRDLDEMO-1.0-SNAPSHOT.jar

```

🚀 Part 2: Migrate Maven Project to Gradle

🔄 Step 1: Initialize Gradle in Your Project

1. **Open Terminal in IntelliJ IDEA**
   Make sure you're in the project directory:

```
cd "D:\Idea Projects\MVNGRDLDEMO"

```

2. **Run Gradle Init Command**
   Execute the following command to migrate your Maven project to Gradle:

```
gradle init --type pom

```

   This command will convert your Maven `pom.xml` into a Gradle `build.gradle` file. 🏗️

🛠️ Step 2: Review and Update `build.gradle`

1. **Open** `build.gradle` in IntelliJ IDEA.
2. Ensure the following configurations are correct:

```
plugins {
    id 'java'
}

group = 'com.example'
version = '1.0-SNAPSHOT'

repositories {
    mavenCentral()
}

dependencies {
    testImplementation 'junit:junit:4.13.2'
}

jar {
    manifest {
        attributes(
            'Main-Class': 'com.example.App'
        )
    }
}

```

🔨 Step 3: Build and Run the Gradle Project

1. **Clean and Build the Project**
   To clean and build your Gradle project, run:

```
gradle clean build

```

2. **Run the Generated JAR File**
   Now, run the generated JAR file using:

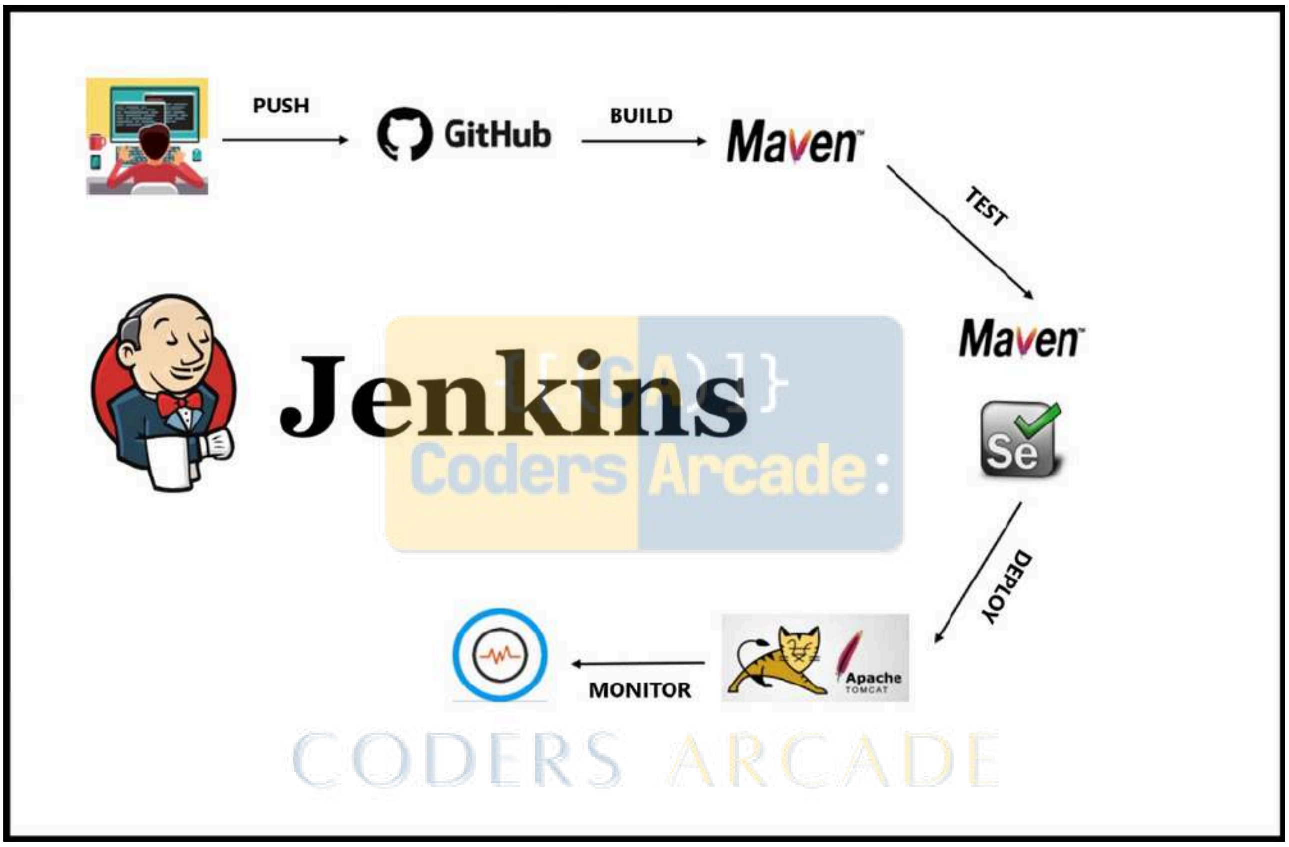
```
java -jar build/libs/MVNGRDLDEMO-1.0-SNAPSHOT.jar

```

🎉 Conclusion

Congratulations! You have successfully:

1. **Created a Maven project** in IntelliJ IDEA. 🎉
2. **Built and packaged** it into a JAR file using Maven. 🛠️
3. **Migrated** the project from Maven to Gradle. 🔄
4. **Built and ran** the project using Gradle. 🚀

CA - Jenkins Notes & Documentation



What is Jenkins

- **Jenkins is a CI CD tool**

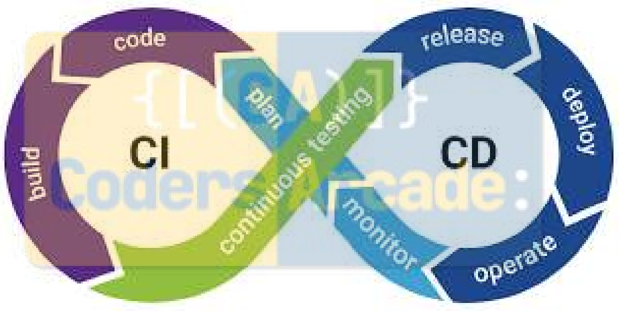
- **Free & Open Source**
- **Written in Java**

What is CI & CD

Watch This Video To Get Detailed Idea About CI/CD Pipeline : DevOps CI CD Pipeline || Simple & Detailed Explanation

Continuous Integration, Delivery & Deployment

CI/CD is **a method to frequently deliver apps to customers by introducing automation into the stages of app development**. The main concepts attributed to CI/CD are continuous integration, continuous delivery, and continuous deployment.



Installation

System Requirements

Memory 256 MB of RAM
Disk Space Depends on your projects
OS Windows, Mac, Ubuntu, Linux
Java 8 or 11 (JDK or JRE)

Installation on Windows

Watch This Video To Seamlessly Install Jenkins : Jenkins Installation - Step by Step Guide
Step 1 : Check Java is installed
Step 2 : Download Jenkins.war file
Step 3 : Goto cmd prompt and run command
java -jar jenkins.war --httpPort=8080
Step 4 : On browser goto http://localhost:8080
Step 5 : Provide admin password and complete the setup

Installation on Mac

Homebrew

Installation on Linux
How to install Jenkins on Amazon AWS EC2 Linux | 8 Steps

2. Jenkins Tutorials: How to install Jenkins on Ubuntu 22.04

Jenkins Configuration

How to change Home Directory

Step 1: Check your Jenkins Home > Manage Jenkins > Configure System
Step 2 : Create a new folder
Step 3 : Copy the data from old folder to new folder
Step 4 : Create/Update env variable JENKINS_HOME
Step 5 : Restart Jenkins
jenkins.xml
JENKINS_HOME

How to setup Git on Jenkins

Step 1 : Goto Manage Jenkins > Manage Plugins
Step 2 : Check if git is already installed in Installed tab
Step 3 : Else goto Available tab and search for git
Step 4 : Install Git
Step 5 : Check git option is present in Job Configuration

Create the first Job on Jenkins

How to connect to Git Remote Repository in Jenkins (GitHub)

Step 1 : Get the url of the remote repository
Step 2 : Add the git credentials on Jenkins
Step 3 : In the jobs configuration goto SCM and provide git repo url in git section
Step 4 : Add the credentials
Step 5 : Run job and check if the repository is cloned

How to use Command Line in Jenkins CLI

Faster, easier, integration
Step 1 : start Jenkins
Step 2 : goto Manage Jenkins - Configure Global Security - enable security
Step 3 : goto - http://localhost:8080/cli/
Step 4 : download jerkins-cli jar. Place at any location.
Step 5 : test the jenkins command line is working

java -jar jenkins-cli.jar -s http://localhost:8080 /help --username <userName> --password <password>

How to create Users + Manage + Assign Roles

How to create New Users
How to configure users

How to create new roles
How to assign users to roles
How to Control user access on projects

Step 1 : Create new users
Step 2 : Configure users
Step 3 : Create and manage user roles Role Based Authorization Strategy Plugin - download - restart jenkins
Step 4 : Manage Jenkins - Configure Global Security - Authorization - Role Based Strategy
Step 5 : Create Roles and Assign roles to users
Step 6 : Validate authorization and authentication are working properly

Jenkins Pipeline Beginner Tutorial

How to create Jenkinsfile

- **What is pipeline**
- **What is jenkins pipeline**
- **What is jenkinsfile**
- **How to create jenkinsfile**

Build > Deploy > Test > Release
Jenkinsfile : Pipeline as a code

Step 1 : Start Jenkins
Step 2 : Install Pipeline Plugin
Step 3 : Create a new job
Step 4 : Create or get Jenkinsfile in Pipeline section
Step 5 : Run and check the output

Jenkins Pipeline
How to get jenkinsfile from Git SCM

Step 1 : Create a new job or use existing job (type : Pipeline)
Step 2 : Create a repository or GitHub
Step 3 : Add Jenkinsfile in the repo
Step 4 : Under Jenkins job > Pipeline section > Select Definition Pipeline script from SCM
Step 5 : Add repo and jenkinsfile location in the job under Pipeline section
Step 6 : Save & Run

Jenkins Pipeline
How to clone a git repo using Jenkinsfile

Show Live Demonstration.

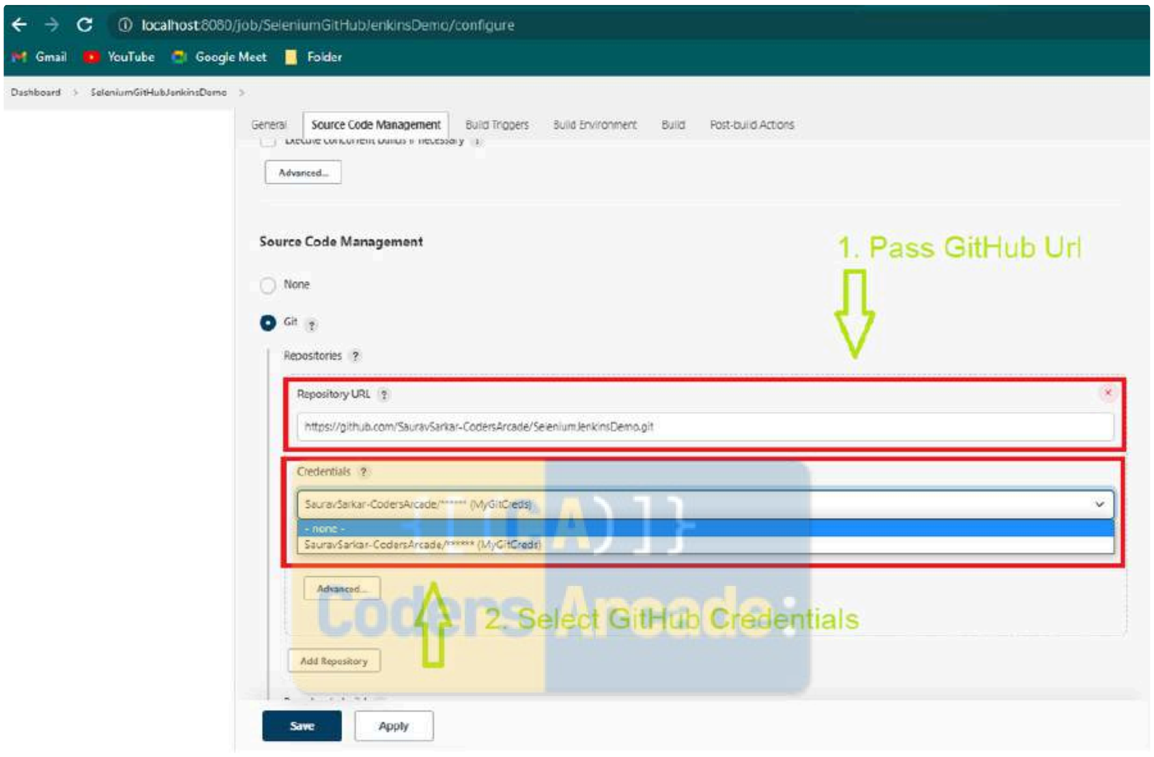
Running Selenium Automation Tests from GitHub via Jenkins CI Agent :

There are two ways in which we can perform Selenium Automation tests from GitHub via Jenkins

- Via **Git Credentials**
- Via **GitHub access token**
- The steps are shown with images.
- In the build, you have to execute the maven commands like: **mvn clean test, etc.**

Method 1:

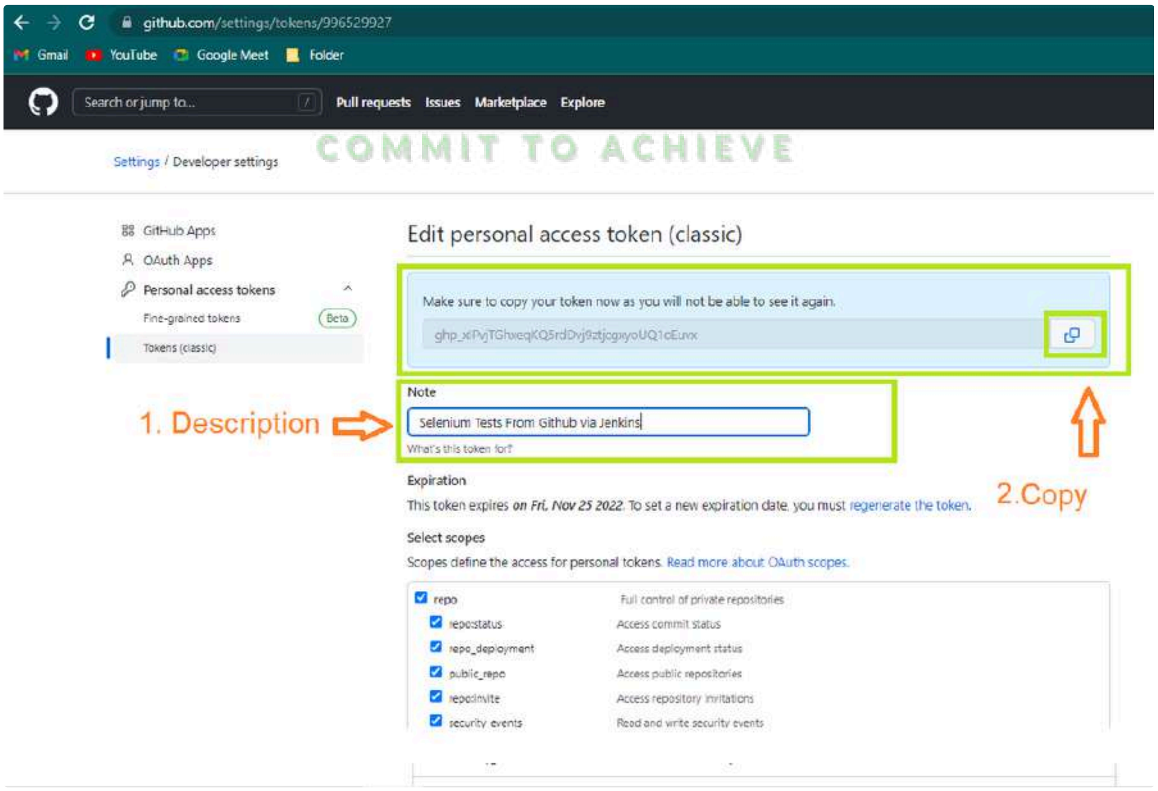
- Via **Git Credentials**


localhost:8080/job/SeleniumGitHubJenkinsDemo/configure
Source Code Management
1. Pass GitHub Url
Repository URL
https://github.com/SauravSarkar-CodersArcade/SeleniumJenkinsDemo.git
Credentials
SauravSarkar-CodersArcade/****** (MyGitCreds)
2. Select GitHub Credentials
Save
Apply

GitHub + Jenkins With Git Credentials

Method 2:

- Via **GitHub Access Token**


github.com/settings/tokens/996529927
Settings / Developer settings
Edit personal access token (classic)
Make sure to copy your token now as you will not be able to see it again.
1. Description
Note
Selenium Tests From Github via Jenkins
2.Copy
Expiration
This token expires on Fri, Nov 25 2022. To set a new expiration date, you must regenerate the token.
Select scopes

GitHub + Jenkins Via Access Token

CA - Experiment 5 - Introduction to Jenkins: What is Jenkins?, Installing Jenkins on Local or Cloud Environment, Configuring Jenkins for First Use

Experiment 5: Introduction to Jenkins

🌟 Objective

To understand the fundamentals of **Jenkins**, install it on a local or cloud environment, and configure it for first-time use.

🔹 1. What is Jenkins?

Jenkins is an **open-source automation server** used for:
✅ **Continuous Integration (CI)** – Automatically testing and integrating code changes
✅ **Continuous Deployment (CD)** – Automating application deployment
✅ **Building Pipelines** – Managing end-to-end software development workflows
✅ **Plugin-Based Extensibility** – Supporting tools like Maven, Gradle, Ansible, Docker, and Azure DevOps

🚀 Why Use Jenkins?

✔ Automates builds and tests
✔ Reduces manual intervention
✔ Improves software quality
✔ Works with multiple tools and platforms

🔹 2. Installing Jenkins

Jenkins can be installed using multiple methods:
1️⃣ **Windows Installer (.msi) – Recommended for Windows**
2️⃣ **Linux Package Manager – Best for Linux Users**
3️⃣ **Jenkins WAR File – Universal method using Java**

We'll cover all three approaches.

🟢 2.1 Prerequisites

🔹 **Java 11 or 17** is required for Jenkins.
Check Java version:

```
java -version
```

If Java is not installed, download it from:
⭕ Download the Latest Java LTS Free

🟠 2.2 Installing Jenkins on Windows (MSI Installer) – Recommended

✅ **Step 1: Download Jenkins**

🔗 **Download from:** Download and deploy
Choose **Windows Installer (.msi)** for an easy setup.

✅ **Step 2: Install Jenkins**

1️⃣ Run the downloaded `.msi` file.
2️⃣ Follow the installation wizard.
3️⃣ Select **Run Jenkins as a Windows Service** (recommended).
4️⃣ Choose the **installation directory** (default: `C:\Program Files\Jenkins`).
5️⃣ Click **Install** and wait for the setup to complete.

✅ **Step 3: Start Jenkins**

1️⃣ Open **Services** (`services.msc`) and ensure **Jenkins** is running.
2️⃣ Open a web browser and go to:

```
http://localhost:8080

```

✅ **Step 4: Unlock Jenkins**

1️⃣ Find the initial **Admin Password** in:

```
C:\Program Files\Jenkins\secrets\initialAdminPassword

```

2️⃣ Copy the password and paste it into the Jenkins setup page.

✅ **Step 5: Install Recommended Plugins**

Jenkins will prompt you to install plugins. Click **"Install Suggested Plugins"**.

✅ **Step 6: Create Admin User**

1️⃣ Set up a **Username**, **Password**, and **Email**.
2️⃣ Click **Save and Finish**.

🔹 **Jenkins is now ready!** 🎉
Access it anytime at:

```
http://localhost:8080

```

🟢 2.3 Installing Jenkins on Linux (Ubuntu/Debian)

✅ **Step 1: Add Jenkins Repository**

```
wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add -
sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'

```

✅ **Step 2: Install Jenkins**

```
sudo apt update
sudo apt install jenkins -y

```

✅ **Step 3: Start Jenkins**

```
sudo systemctl start jenkins
sudo systemctl enable jenkins

```

✅ **Step 4: Access Jenkins**

Find the **initial password** in:

```
sudo cat /var/lib/jenkins/secrets/initialAdminPassword

```

Then open Jenkins in a browser:

```
http://localhost:8080

```

🟠 2.4 Installing Jenkins Using WAR File (Works on Any OS)

This method allows you to run Jenkins without installing it as a service.

✅ **Step 1: Download the Jenkins WAR File**

🔗 Download from: Download and deploy
Choose **Generic Java Package (.war)**.

✅ **Step 2: Run Jenkins Using Java**

Navigate to the folder where the `.war` file is downloaded and run:

```
java -jar jenkins.war --httpPort=8080

```

🔹 This will start Jenkins on port **8080**.

✅ **Step 3: Open Jenkins in Browser**

Go to:

```
http://localhost:8080

```

✅ **Step 4: Unlock Jenkins & Setup**

Follow the **same steps** as the Windows/Linux installation:
✔ Find the initial password
✔ Install plugins
✔ Create an admin user

🔹 **Jenkins is now running without installation!**

To stop Jenkins, press **CTRL + C** in the terminal.

🔹 3. Configuring Jenkins for First Use

✅ 3.1 Understanding the Jenkins Dashboard

After logging in, you will see:
🔹 **New Item** → Create Jobs/Pipelines

- 🔹 **Manage Jenkins** → Configure System, Users, and Plugins
- 🔹 **Build History** → View previous builds
- 🔹 **Credentials** → Store secure authentication details

✅ 3.2 Installing Additional Plugins

Jenkins supports **plugins** for various tools like Maven, Gradle, Docker, and Azure DevOps.

🔹 To install a plugin:
1️⃣ Go to **Manage Jenkins → Manage Plugins**
2️⃣ Search for the required plugin
3️⃣ Click **Install without Restart**

✅ 3.3 Setting Up Global Tool Configuration

Configure Java, Maven, and Gradle in Jenkins:
1️⃣ Go to **Manage Jenkins → Global Tool Configuration**
2️⃣ Add paths for:

- **JDK** (`C:\Program Files\Java\jdk-17`)
- **Maven** (`C:\Maven\apache-maven-<version>`)
- **Gradle** (`C:\Gradle\gradle-<version>`)
  3️⃣ Click **Save**

📌 Assessment Questions

1️⃣ What is **Jenkins** used for in DevOps?
2️⃣ Explain the **difference** between CI and CD in Jenkins.
3️⃣ How do you **install Jenkins** on Windows, Linux, and using the WAR file?
4️⃣ Where can you find the **initial Jenkins password** after installation?
5️⃣ What are some **essential Jenkins plugins** for automation?

📌 Important Note

You can use the below links to easily install **Jenkins** & understand **CI/CD Pipelines** in **DevOps**.

🔹 **Jenkins Installation Video** 🎥 – Follow this step-by-step guide for a **seamless Jenkins setup**: ▶ Jenkins Installation - Step by Step Guide

🔹 **Understanding CI/CD in DevOps** 🚀 – Learn how Jenkins fits into the **CI/CD pipeline**: ▶ DevOps CI CD Pipeline || Simple & Detailed Explanation

Ensure you have **Java installed** before setting up Jenkins. If you face any issues, check the **Jenkins logs** for troubleshooting. ✅

CA - Experiment 6 - Continuous Integration with Jenkins: Setting Up a CI Pipeline, Integrating Jenkins with Maven/Gradle, Running Automated Builds and Tests

Experiment 6: Continuous Integration with Jenkins

Objective

To set up a Continuous Integration (CI) pipeline in Jenkins, integrate it with Git, and run Selenium Java tests using Maven.

Prerequisites

Before proceeding, ensure the following:

✅ **Jenkins is installed and running** on your system. If not, refer to [Experiment 5].
✅ **Git is installed** and configured in Jenkins. (Verify with `git --version`).
✅ **Maven is installed** and configured in Jenkins. (Check with `mvn -version`).
✅ **Selenium Maven Project is ready** with test cases (`src/test/java`).
✅ **Project is stored in two places**:

- Locally on your system (e.g., `D:\Idea Projects\MVNGRDLDEMO`).
- Pushed to **GitHub** with a valid repository link.
  ✅ **Jenkins has access to the GitHub repository** (via credentials).

1. Configuring Jenkins & Git Integration

Step 1: Verify Git Installation in Jenkins

1. Open **Jenkins Dashboard** → **Manage Jenkins** → **Global Tool Configuration**.
2. Under **Git**, verify the installation path (e.g., `C:\Program Files\Git\bin\git.exe`).
3. Click **Save**.

Step 2: Add GitHub Credentials in Jenkins

1. Navigate to **Manage Jenkins** → **Manage Credentials**.
2. Select **Global credentials (unrestricted)** → Click **Add Credentials**.
3. Choose **Username with password** or **SSH Key**, provide details, and click **OK**.

2. Running a Selenium Java Test from a Local Maven Project

Step 1: Create a New Jenkins Job

1. Go to **Jenkins Dashboard** → Click **New Item**.
2. Enter a project name → Select **Freestyle Project**.
3. Click **OK**.

Step 2: Configure the Build Step

1. Scroll to **Build** → Click **Add build step** → **Execute Windows Batch Command**.
2. Enter the following commands (**ensure correct navigation to project directory**):

```
cd D:\Idea Projects\MVNGRDLDEMO
mvn test

```

3. Click **Save** → Click **Build Now** to execute the test.

---

3. Running Selenium Tests from a GitHub Repository via Jenkins

Step 1: Set Up a New Jenkins Job for GitHub Project

1. Go to **Jenkins Dashboard** → Click **New Item**.
2. Enter a project name → Select **Freestyle Project**.
3. Click **OK**.

Step 2: Configure Git Repository in Jenkins

1. Under **Source Code Management**, select **Git**.
2. Enter your GitHub repository URL (e.g., `https://github.com/your-repo-name.git`).
3. Select the **Git credentials** configured earlier.

Step 3: Add Build Step for Maven

1. Scroll to **Build** → Click **Add build step** → **Execute Windows Batch Command**.
2. Enter the Maven test command:

```
mvn test

```

3. Click **Save**.

Step 4: Trigger the Build

1. Click **Build Now** to fetch the code from GitHub and execute the Selenium tests.
2. Check the **Console Output** to verify test execution.

Important Notes

📌 **Prerequisites are crucial!** Make sure Jenkins, Git, Maven, and Selenium projects are set up correctly before proceeding.
📌 **Always navigate to the project directory** before running `mvn test` from a local system.
📌 **Use webhooks** in GitHub to automatically trigger builds when new code is pushed.
📌 **Configure email notifications** in Jenkins for build status updates.

🔗 **Jenkins & Git Integration Video:** [To Be Added - Version 1.2]
🔗 **Running Selenium Tests in Jenkins:** [To Be Added - Version 1.2]
