

- **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`:

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5   <modelVersion>4.0.0</modelVersion>
6   <groupId>com.example</groupId>
7   <artifactId>simple-project</artifactId>
8   <version>1.0-SNAPSHOT</version>
9   <dependencies>
10     <!-- Add your dependencies here -->
11   </dependencies>
12 </project>
```

```
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
5   <modelVersion>4.0.0</modelVersion>
6   <groupId>com.example</groupId>
7   <artifactId>simple-project</artifactId>
8   <version>1.0-SNAPSHOT</version>
9   <dependencies>
10     <!-- Add your dependencies here -->
11   </dependencies>
12 </project>
```

### 4. Add Dependencies for Selenium and TestNG:

- In the `pom.xml`, add Selenium and TestNG dependencies under the `<dependencies>` section.

```
1 <dependencies>
2   <dependency>
3     <groupId>org.seleniumhq.selenium</groupId>
4     <artifactId>selenium-java</artifactId>
5     <version>3.141.59</version>
6   </dependency>
7   <dependency>
8     <groupId>org.testng</groupId>
9     <artifactId>testng</artifactId>
10    <version>7.4.0</version>
11    <scope>test</scope>
12  </dependency>
13 </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`:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>My Simple Website</title>
7   <link rel="stylesheet" href="style.css">
8 </head>
9 <body>
10   <header>
11     
12   </header>
13   <h1>Welcome to My Simple Website</h1>
14 </body>
15 </html>
```

Example of a simple `style.css`:

```
1 body {
```

```
2 font-family: Arial, sans-serif;
3 background-color: #f4f4f4;
4 text-align: center;
5 }
6 header img {
7   width: 100px;
8 }
9
```

### 6. Upload the Website to GitHub:

- Initialize a Git repository in your project folder:

```
1 git init
```



```
8 }
9
```

#### 6. Upload the Website to GitHub:

- Initialize a Git repository in your project folder:

```
1 git init
2
```

- Add your files and commit them:

```
1 git add .
2 git commit -m "Initial commit"
3
```

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

```
1 git remote add origin <your-repository-url>
2 git push -u origin master
3
```

### 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`:

```
1 <build>
2   <plugins>
3     <plugin>
4       <groupId>org.apache.maven.plugins</groupId>
5       <artifactId>maven-resources-plugin</artifactId>
6       <version>3.2.0</version>
7       <executions>
8         <execution>
9           <phase>prepare-package</phase> <!-- Before packaging -->
10          <goals>
11            <goal>copy-resources</goal>
12          </goals>
13          <configuration>
14            <outputDirectory>${project.basedir}/docs</outputDirectory> <!-- Deploy to /docs
15            folder -->
16          </configuration>
17          <resources>
18            <resource>
19              <directory>src/main/resources</directory>
```

```
18       <includes>
19         <include>**/*</include> <!-- Copy all files in src/main/resources -->
20       </includes>
21     </resource>
22   </resources>
23 </configuration>
24 </execution>
25 </executions>
26 </plugin>
27 </plugins>
28 </build>
29
```

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:

```
1 mvn clean install
2
```

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):

```
1 git add docs/*
2 git commit -m "Deploy site to GitHub Pages"
3
```

- **Push to GitHub:**

```
1 git push origin master # Or the branch you are using, maybe 'main' in some cases
2
```

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:

```
1 https://<your-github-username>.github.io/<your-repository-name>/
2
```

#### Summary:

- **Maven Resources Plugin** is configured to copy static files ( `index.html` , `style.css` , `logo.png` ) into the `/docs` folder.



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:

```
1 package org.test;
2
3 import org.openqa.selenium.WebDriver;
4 import org.openqa.selenium.chrome.ChromeDriver;
5 import org.testng.Assert;
6 import org.testng.annotations.AfterTest;
7 import org.testng.annotations.BeforeTest;
8 import org.testng.annotations.Test;
9
10 import static org.testng.Assert.assertTrue;
11
12 public class WebpageTest {
13     private static WebDriver driver;
14
15     @BeforeTest
16     public void openBrowser() throws InterruptedException {
17         driver = new ChromeDriver();
18         driver.manage().window().maximize();
19         Thread.sleep(2000);
20         driver.get("https://sauravsarkar-codersarcade.github.io/CA-MVN/"); // "Note: You should use your
21         // GitHub-URL here...!!!"
22     }
23
24     @Test
25     public void titleValidationTest(){
26         String actualTitle = driver.getTitle();
27         String expectedTitle = "Tripillar Solutions";
28         Assert.assertEquals(actualTitle, expectedTitle);
29         assertTrue(true, "Title should contain 'Tripillar'");
30     }
31
32     @AfterTest
33     public void closeBrowser() throws InterruptedException {
34         Thread.sleep(1000);
35         driver.quit();
36     }
37 }
38
```

## 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..!!)**

```
1 <dependencies>
2     <!-- Selenium WebDriver dependency -->
3     <dependency>
4         <groupId>org.seleniumhq.selenium</groupId>
5         <artifactId>selenium-java</artifactId>
6         <version>4.8.0</version> <!-- Ensure this is the latest version -->
7     </dependency>
8
9     <!-- TestNG dependency -->
10    <dependency>
11        <groupId>org.testng</groupId>
12        <artifactId>testng</artifactId>
13        <version>7.7.0</version> <!-- Ensure this is the latest version -->
14        <scope>test</scope>
15    </dependency>
16 </dependencies>
17
```

- **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:

```
1 package org.test;
```



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`:

```
1 <build>
2   <plugins>
3     <!-- Maven JAR Plugin -->
4     <plugin>
```

```
5     <groupId>org.apache.maven.plugins</groupId>
6     <artifactId>maven-jar-plugin</artifactId>
7     <version>3.2.0</version>
8     <configuration>
9       <!-- Specify the main class to be executed -->
10      <archive>
11        <manifestEntries>
12          <Main-Class>com.example.MainClass</Main-Class> <!-- Replace with your main class path -->
13        </manifestEntries>
14      </archive>
15    </configuration>
16  </plugin>
17 </plugins>
18 </build>
19
```

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`:

```
1 package com.example;
2
3 public class MainClass {
4     public static void main(String[] args) {
5         System.out.println("Hello, this is a simple output from the main class!");
6     }
7 }
8
```

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:

```
1 mvn clean package
2
```

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:

```
1 java -jar target/your-project-name.jar
2
```

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

```
1 Hello, this is a simple output from the main class!
2
```

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.



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:

```
1 mvn clean package # Clean & build the project
2 mvn clean install # Clean, build & install in local repo
3 mvn deploy        # Deploy to a remote repository
4
```

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`:

```
1 <build>
2   <plugins>
3     <plugin>
4       <groupId>org.apache.maven.plugins</groupId>
5       <artifactId>maven-site-plugin</artifactId>
6       <version>3.12.1</version> <!-- Use latest version -->
7     </plugin>
8   </plugins>
9 </build>
10
```

##### 🔧 Step 2: Run the Site Command

Once the plugin is added, execute:

```
1 mvn site
2
```

#### ✅ 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:

```
1 D:\Idea Projects\CA-MVN\target\site\index.html
2
```

#### 🔍 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.



## 🔥 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

```
1 mkdir D:\my-local-maven-repo
2
```

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

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

```
1 <distributionManagement>
2   <repository>
3     <id>local-repo</id>
4     <url>file:///D:/my-local-maven-repo</url>
5   </repository>
6 </distributionManagement>
7
```

### 📦 Step 3: Run the Deploy Command

```
1 mvn deploy
2
```

#### ✅ 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:

```
1 D:/my-local-maven-repo/
2 |— org/example/CA-MVN/1.0-SNAPSHOT/
3 |   |— CA-MVN-1.0-SNAPSHOT.jar
4 |   |— CA-MVN-1.0-SNAPSHOT.pom
5
```

## 🎯 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`:

```
1 <distributionManagement>
2   <repository>
3     <id>github</id>
4     <url>https://maven.pkg.github.com/YOUR_GITHUB_USERNAME/maven-repo</url>
5   </repository>
6
```