Channabasaveshwara Institute of Technology
(Affiliated to VTU, Belgaum & Approved by AICTE, New Delhi)
(**NAAC Accredited & ISO 9001:2015 Certified Institution)**
NH 206 (B.H. Road), Gubbi, Tumkur – 572 216. Karnataka

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

Course Name: DEVOPS

Course Code: BCSL657D

B.E- VI Semester

Lab Manual 2025-26

Course Coordinator:
Mr. Dharaneshkumar M L
Assistant Professor, Dept. of AI &DS
CIT, Gubbi

HOD:
Dr. Gavisiddappa
HOD, Dept. of AI &DS
CIT, Gubbi

Channabasaveshwara Institute of Technology

(Affiliated to VTU, Belgaum & Approved by AICTE, New Delhi)

(NAAC Accredited & ISO 9001:2015 Certified Institution)

NH 206, (B.H. Road), Gubbi, Tumkur – 572 216. Karnataka

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

Syllabus

DEVOPS		Semester	**6**
Course Code	**BCSL657D**	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	0:0:2:0	SEE Marks	50
Credits	01	Exam Hours	100
Examination type (SEE)	Practical		

Course objectives:

- To introduce DevOps terminology, definition & concepts
- To understand the different Version control tools like Git, Mercurial
- To understand the concepts of Continuous Integration/ Continuous Testing/ Continuous Deployment)
- To understand Configuration management using Ansible
- Illustrate the benefits and drive the adoption of cloud-based Devops tools to solve real world problems

Sl.NO	Experiments
1	**Introduction to Maven and Gradle:** Overview of Build Automation Tools, Key Differences Between Maven and Gradle, Installation and Setup
2	**Working with Maven:** Creating a Maven Project, Understanding the POM File, Dependency Management and Plugins
3	**Working with Gradle:** Setting Up a Gradle Project, Understanding Build Scripts (Groovy and Kotlin DSL), Dependency Management and Task Automation
4	**Practical Exercise:** Build and Run a Java Application with Maven**,** Migrate the Same Application to Gradle
5	**Introduction to Jenkins:** What is Jenkins?, Installing Jenkins on Local or Cloud Environment, Configuring Jenkins for First Use
6	**Continuous Integration with Jenkins:** Setting Up a CI Pipeline**,** Integrating Jenkins with Maven/Gradle**,** Running Automated Builds and Tests
7	**Configuration Management with Ansible:** Basics of Ansible: Inventory, Playbooks, and Modules, Automating Server Configurations with Playbooks, Hands-On: Writing and Running a Basic Playbook
8	**Practical Exercise:** Set Up a Jenkins CI Pipeline for a Maven Project, Use Ansible to Deploy Artifacts Generated by Jenkins
9	**Introduction to Azure DevOps:** Overview of Azure DevOps Services, Setting Up an Azure DevOps Account and Project
10	**Creating Build Pipelines:** Building a Maven/Gradle Project with Azure Pipelines, Integrating Code Repositories (e.g., GitHub, Azure Repos), Running Unit Tests and Generating Reports
11	**Creating Release Pipelines:** Deploying Applications to Azure App Services, Managing Secrets and Configuration with Azure Key Vault, Hands-On: Continuous Deployment with Azure Pipelines

12	**Practical Exercise and Wrap-Up:** Build and Deploy a Complete DevOps Pipeline, Discussion on Best Practices and Q&A

Course outcomes (Course Skill Set):

At the end of the course the student will be able to:

- Demonstrate different actions performed through Version control tools like Git.
- Perform Continuous Integration and Continuous Testing and Continuous Deployment using Jenkins by building and automating test cases using Maven & Gradle.
- Experiment with configuration management using Ansible.
- Demonstrate Cloud-based DevOps tools using Azure DevOps.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together

Continuous Internal Evaluation (CIE):

CIE marks for the practical course are **50 Marks**.

The split-up of CIE marks for record/ journal and test are in the ratio **60:40**.

- Each experiment is to be evaluated for conduction with an observation sheet and record write-up. Rubrics for the evaluation of the journal/write-up for hardware/software experiments are designed by the faculty who is handling the laboratory session and are made known to students at the beginning of the practical session.
- Record should contain all the specified experiments in the syllabus and each experiment write-up will be evaluated for 10 marks.
- Total marks scored by the students are scaled down to **30 marks** (60% of maximum marks).
- Weightage to be given for neatness and submission of record/write-up on time.
- Department shall conduct a test of 100 marks after the completion of all the experiments listed in the syllabus.
- In a test, test write-up, conduction of experiment, acceptable result, and procedural knowledge will carry a weightage of 60% and the rest 40% for viva-voce.
- The suitable rubrics can be designed to evaluate each student's performance and learning ability.
- The marks scored shall be scaled down to **20 marks** (40% of the maximum marks).

The Sum of scaled-down marks scored in the report write-up/journal and marks of a test is the total CIE marks scored by the student.

Semester End Evaluation (SEE):

- SEE marks for the practical course are 50 Marks.
- SEE shall be conducted jointly by the two examiners of the same institute, examiners are appointed by the Head of the Institute.
- The examination schedule and names of examiners are informed to the university before the conduction of the examination. These practical examinations are to be conducted between the schedules mentioned in the academic calendar of the University.
- All laboratory experiments are to be included for practical examination.
- (Rubrics) Breakup of marks and the instructions printed on the cover page of the answer script to be strictly adhered to by the examiners. **OR** based on the course requirement evaluation rubrics shall be decided jointly by examiners.

- Students can pick one question (experiment) from the questions lot prepared by the examiners jointly.
- Evaluation of test write-up/ conduction procedure and result/viva will be conducted jointly by examiners.

General rubrics suggested for SEE are mentioned here, writeup-20%, Conduction procedure and result in -60%, Viva-voce 20% of maximum marks. SEE for practical shall be evaluated for 100 marks and scored marks shall be scaled down to 50 marks (however, based on course type, rubrics shall be decided by the examiners)

Change of experiment is allowed only once and 15% of Marks allotted to the procedure part are to be made zero.

The minimum duration of SEE is 02 hours

Suggested Learning Resources:

- https://www.geeksforgeeks.org/devops-tutorial/
- https://www.javatpoint.com/devops
- https://www.youtube.com/watch?v=2N-59wUIPVI
- https://www.youtube.com/watch?v=87ZqwoFeO88

1 Introduction to Maven and Gradle: Overview of Build Automation Tools, Key Differences between Maven and Gradle, Installation and Setup

Overview of Build Automation Tools

What is Build Automation?

Build automation is the process of scripting or automating the compilation, testing, packaging, and deployment of software projects. It is a crucial part of Continuous Integration (CI) and Continuous Deployment (CD) in DevOps.

Why Use Build Automation Tools?

- Reduces manual errors in the build process.
- Increases productivity by automating repetitive tasks.
- Ensures consistency in software builds.
- Facilitates continuous integration and deployment.

Popular Build Automation Tools

- Maven (Apache Maven)
- Gradle
- Ant (Apache Ant)
- Bazel (by Google)
- Make (GNU Make)

Among these, Maven and Gradle are the most widely used in Java-based projects.

Introduction to Maven

What is Maven?

Apache Maven is a Java-based build automation and project management tool that follows a convention-over-configuration approach.

Features of Maven

- Uses XML-based configuration (pom.xml).
- Supports dependency management via the Maven Central Repository.
- Automates the build lifecycle (compile, test, package, deploy).
- Integrates with tools like Jenkins, SonarQube, Docker, and Kubernetes.

Maven Build Lifecycle

Maven follows a structured build lifecycle with phases such as:

1. validate – Validates project configuration.
2. compile – Compiles the source code.
3. test – Runs unit tests.
4. package – Packages code (e.g., jar or war).
5. install – Installs package locally.
6. deploy – Deploys artifact to a remote repository.

Introduction to Gradle

What is Gradle?

Gradle is a powerful, flexible, and high-performance build tool that supports multi-language builds (Java, Kotlin, Groovy, C++, etc.).

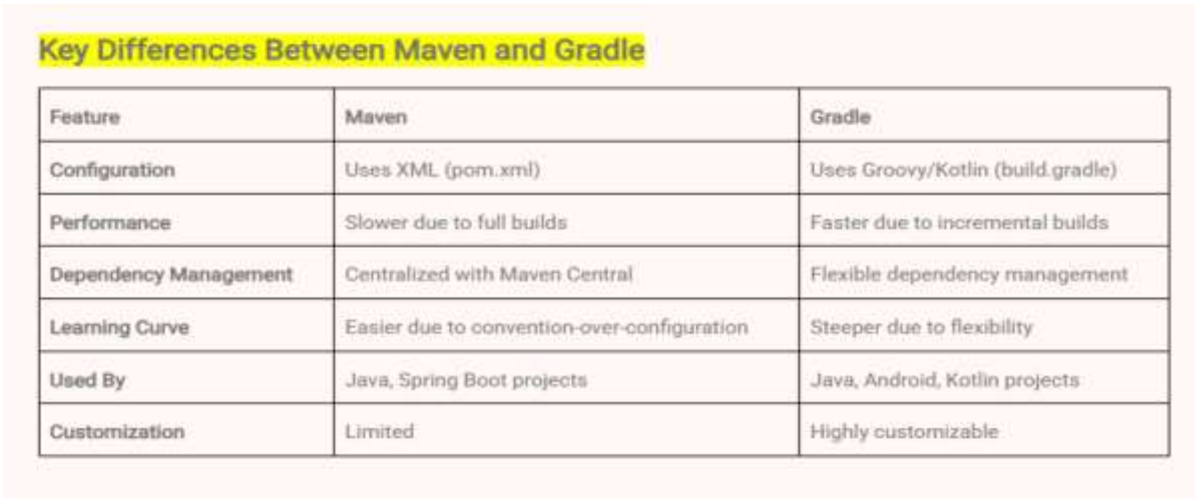
Features of Gradle

- Uses Groovy or Kotlin DSL instead of XML (build.gradle).
- Faster than Maven due to incremental builds.
- Supports dependency management using repositories like Maven Central.
- Integrates with tools like Jenkins, Docker, Kubernetes, and Android Studio.

Gradle Build Lifecycle

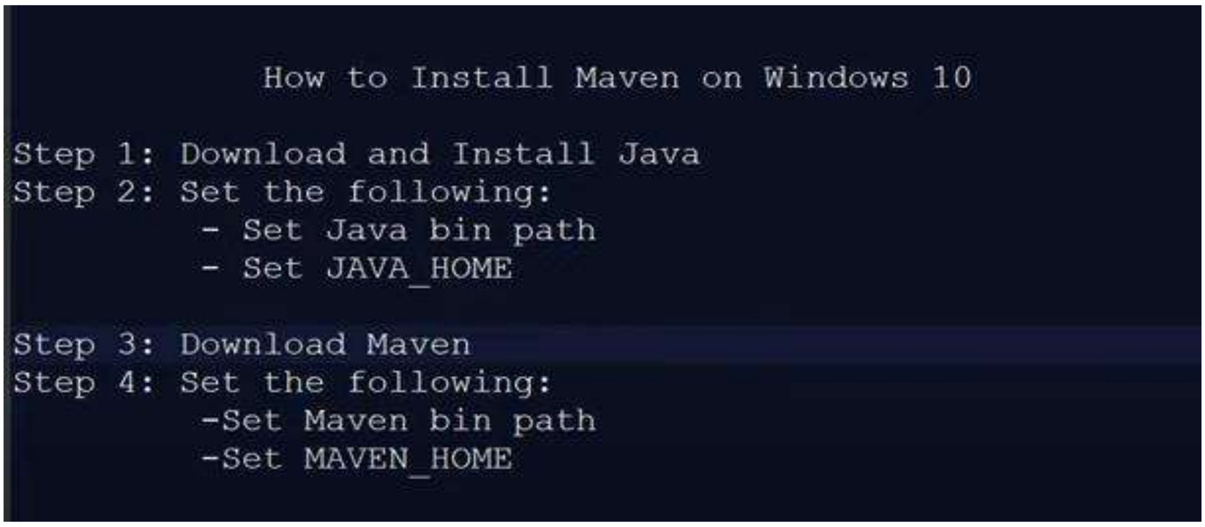
Gradle doesn't follow a fixed lifecycle like Maven but is highly customizable using tasks:

1. clean – Deletes previous build files.
2. build – Compiles and packages the project.
3. test – Runs unit tests.
4. assemble – Creates executable artifacts.



Key Differences Between Maven and Gradle

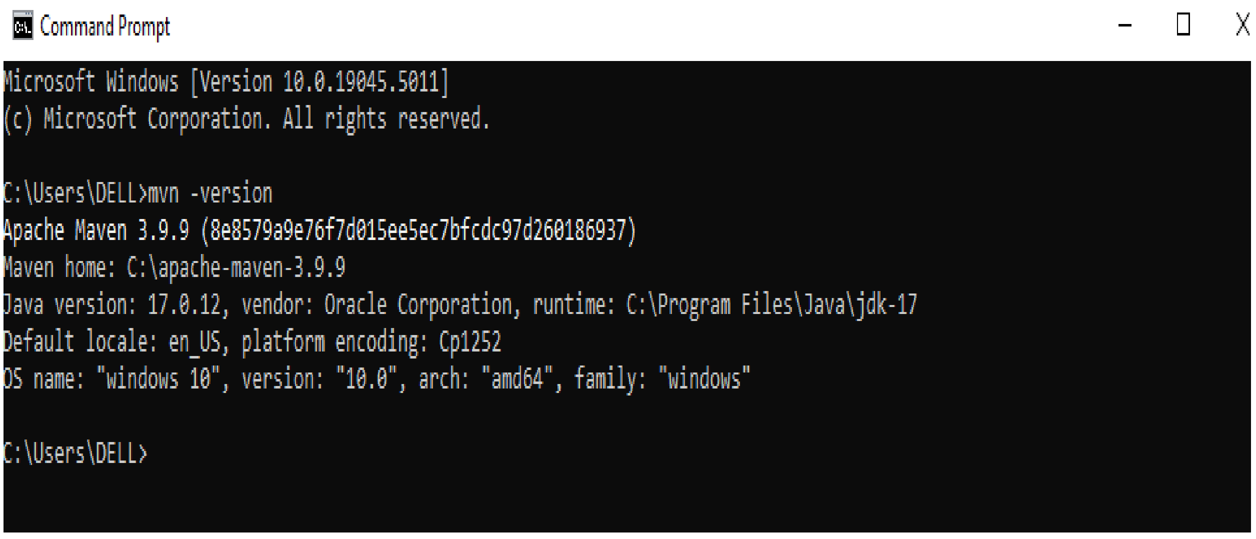
Feature	Maven	Gradle
Configuration	Uses XML (pom.xml)	Uses Groovy/Kotlin (build.gradle)
Performance	Slower due to full builds	Faster due to incremental builds
Dependency Management	Centralized with Maven Central	Flexible dependency management
Learning Curve	Easier due to convention-over-configuration	Steeper due to flexibility
Used By	Java, Spring Boot projects	Java, Android, Kotlin projects
Customization	Limited	Highly customizable



```
        How to Install Maven on Windows 10

Step 1: Download and Install Java
Step 2: Set the following:
         - Set Java bin path
         - Set JAVA_HOME

Step 3: Download Maven
Step 4: Set the following:
         -Set Maven bin path
         -Set MAVEN_HOME
```

How to Install Maven:

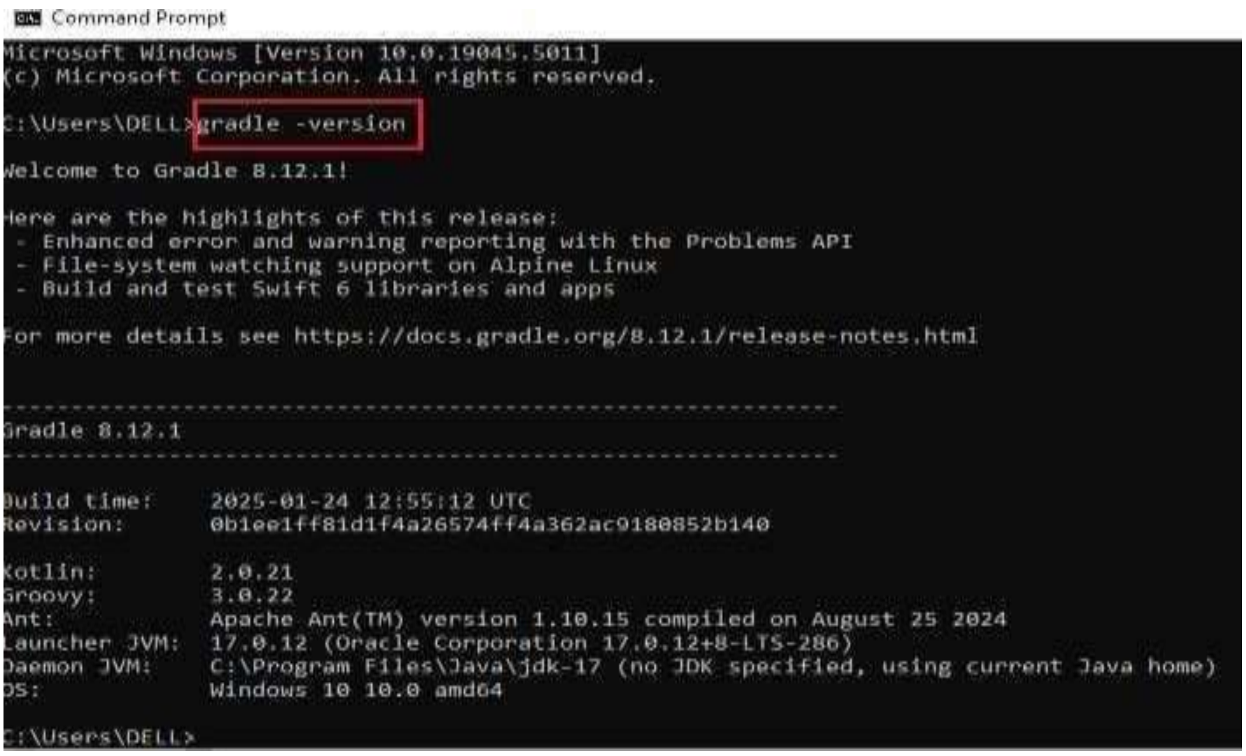
1. **Download Maven:**
   - Go to the Maven Download Page and download the latest binary ZIP file.
2. **Extract the ZIP File:**
   - Right-click the downloaded ZIP file and select **Extract All...** or use any extraction tool like WinRAR or 7-Zip.
3. **Move the Folder:**
   - After extraction, move the extracted **Maven folder** (usually named **`apache-maven-x.x.x`**) to a convenient directory like `C:\Program Files\`.
4. **Navigate to the `bin` Folder:**
   - Open the **Maven folder**, then navigate to the **`bin`** folder inside.
   - Copy the path from the File Explorer address bar(e.g., **`C:\Program Files\apache-maven-x.x.x\bin`**).
5. **Set Environment Variables:**
   - Open the **Start Menu**, search for **Environment Variables**, and select **Edit the system environment variables**.
   - Click **Environment Variables**.
   - Under **System Variables**:
     - Find the **path**, double click on it and click **New**.
     - Paste the full path to the `bin` folder of your Maven directory (e.g., **`C:\Program Files\apache-maven-x.x.x\bin`**).
6. **Save the Changes:**
   - Click **OK** to close the windows and save your changes.
7. **Verify the Installation:**
   - Open Command Prompt and run: **`mvn -v`** If Maven is correctly installed, it will display the version number.



```
Microsoft Windows [Version 10.0.19045.5011]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DELL>mvn -version
Apache Maven 3.9.9 (8e8579a9e76f7d015ee5ec7bfcdc97d260186937)
Maven home: C:\apache-maven-3.9.9
Java version: 17.0.12, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk-17
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"

C:\Users\DELL>
```

How to install Gradle

1. **Download Gradle:**
   Visit the Gradle Downloads Page and download the latest binary ZIP file.
2. **Extract the ZIP File:**
   - Right-click the downloaded ZIP file and select **Extract All...** or use any extraction tool like WinRAR or 7-Zip.
3. **Move the Folder:**
   - After extraction, move the extracted **Gradle folder** (usually named **`gradle-x.x.x`**) to a convenient directory like `C:\Program Files\`.
4. **Navigate to the `bin` Folder:**
   - Open the **Gradle folder**, then navigate to the **`bin`** folder inside.
   - Copy the path from the File Explorer address bar (e.g., **`C:\Program Files\gradle-x.x\bin`**).
5. **Set Environment Variables:**
   - Open the **Start Menu**, search for **Environment Variables**, and select **Edit the system environment variables**.
   - Click **Environment Variables**.
   - Under **System Variables**:
     - Find the **path**, double click on it and click **New**.
     - Paste the full path to the `bin` folder of your Gradle directory (e.g., **`C:\Program Files\gradle-x.x.x\bin`**).
6. **Save the Changes:**
   - Click **OK** to close the windows and save your changes.
7. **Verify the Installation:**
   - Open a terminal or Command Prompt and run: **`gradle -v`** If it shows the Gradle version, the setup is complete.



```
Microsoft Windows [Version 10.0.19045.5011]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DELL>gradle -version

Welcome to Gradle 8.12.1!

Here are the highlights of this release:
 - Enhanced error and warning reporting with the Problems API
 - File-system watching support on Alpine Linux
 - Build and test Swift 6 libraries and apps

For more details see https://docs.gradle.org/8.12.1/release-notes.html


------------------------------------------------------------
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:  17.0.12 (Oracle Corporation 17.0.12+8-LTS-286)
Daemon JVM:    C:\Program Files\Java\jdk-17 (no JDK specified, using current Java home)
OS:            Windows 10 10.0 amd64

C:\Users\DELL>
```

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

What is Maven?

Apache Maven is a build automation and project management tool that simplifies software development by managing dependencies, compiling code, running tests, and packaging applications. It follows a convention-over-configuration approach and is widely used in Java-based projects.

Why Use Maven?

- Standardized project structure.
- Automatic dependency management using pom.xml.
- Integration with CI/CD pipelines (Jenkins, GitHub Actions, Azure DevOps).
- Supports various build lifecycle phases (compile, test, package, deploy).

Creating a Maven Project

Step 1: Install Maven (If Not Installed)

Check if Maven is installed using:

```
mvn -version
```

If not installed, follow Exercise 1 for installation steps.

Step 2: Create a Maven Project Using the Command Line

Open Command Prompt:

```
mvn archetype:generate -DgroupId=com.example -DartifactId=myapp -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Explanation:

- -DgroupId=com.example → Defines the package structure (similar to a namespace).
- -DartifactId=myapp → The project name and the name of the final artifact (JAR file).
- -DarchetypeArtifactId=maven-archetype-quickstart → Uses a pre-defined template for Java projects.
- -DinteractiveMode=false → Skips manual inputs and creates the project automatically.

Step 3: Navigate to the Project Folder

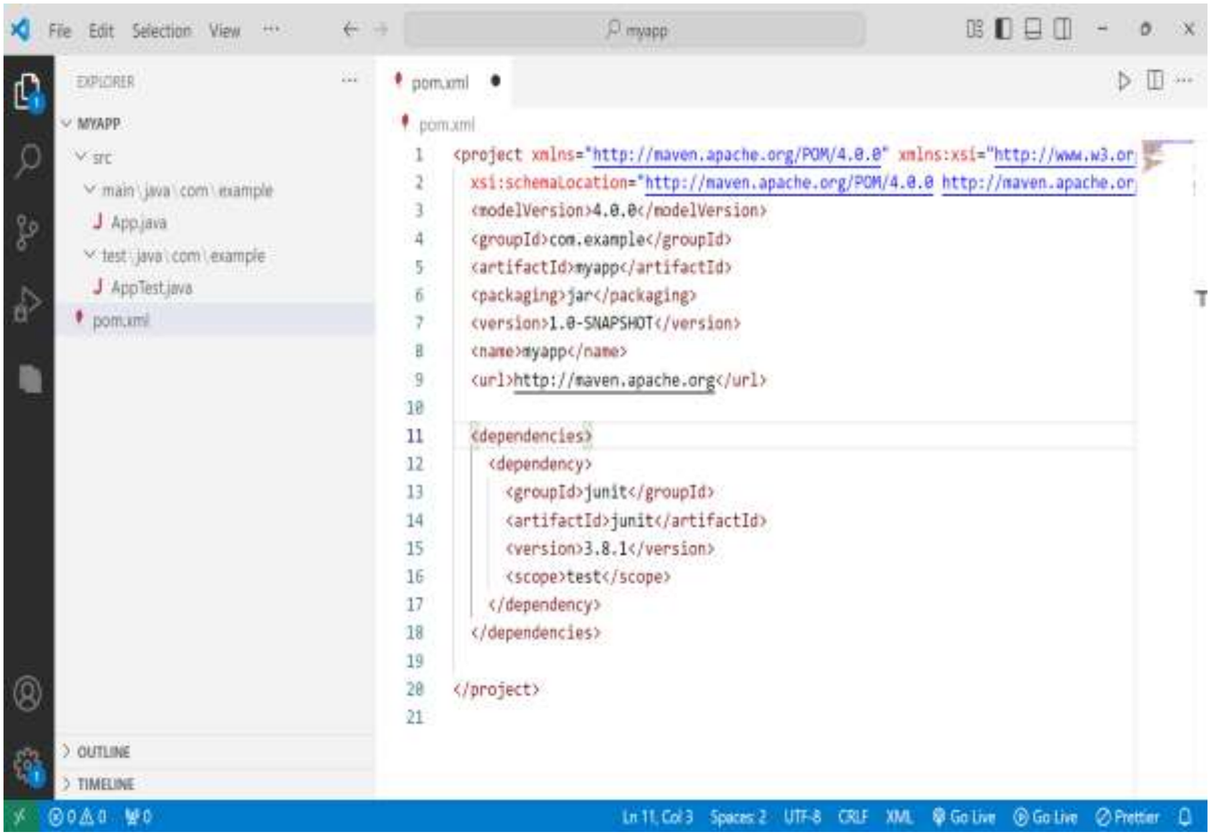
```
cd myapp
```

Step 4: Verify the Project Structure

Maven creates a standard folder structure like this:

```
myapp
|— src
|   ├— main
|   |   └— java
|   |       └— com.example
|   |           └— App.java
|   ├— test
|   |   └— java
|   |       └— com.example
|   |           └— AppTest.java
|— pom.xml
|— README.md
|— .gitignore
```

- src/main/java → Contains the main application code.
- src/test/java → Contains test cases for the application.
- pom.xml → The core configuration file for Maven.



```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.or
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.or
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>myapp</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>myapp</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

</project>
```

Note: Used VS Code Text Editor. Make 'myapp' as your project folder

Understanding the POM File (pom.xml)

The Project Object Model (POM) file is the heart of a Maven project. It defines project metadata, dependencies, plugins, and build configurations.

Example pom.xml File

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

  <groupId>com.example</groupId>
  <artifactId>myapp</artifactId>
  <version>1.0-SNAPSHOT</version>

  <dependencies>
    <!-- Example Dependency: JUnit for Testing -->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <!-- Maven 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>
        </plugins>
    </build>
</project>
```

Key Sections of pom.xml

1. <groupId> - Unique identifier for the project (e.g., com.example).
2. <artifactId> - Project name (e.g., myapp).
3. <version> - Version of the project (e.g., 1.0-SNAPSHOT).
4. <dependencies> - Lists external libraries required for the project.
5. <build> - Specifies plugins for compiling, packaging, and testing.

Managing Dependencies in Maven

Maven automatically downloads required libraries from the Maven Central Repository.

Adding a Dependency

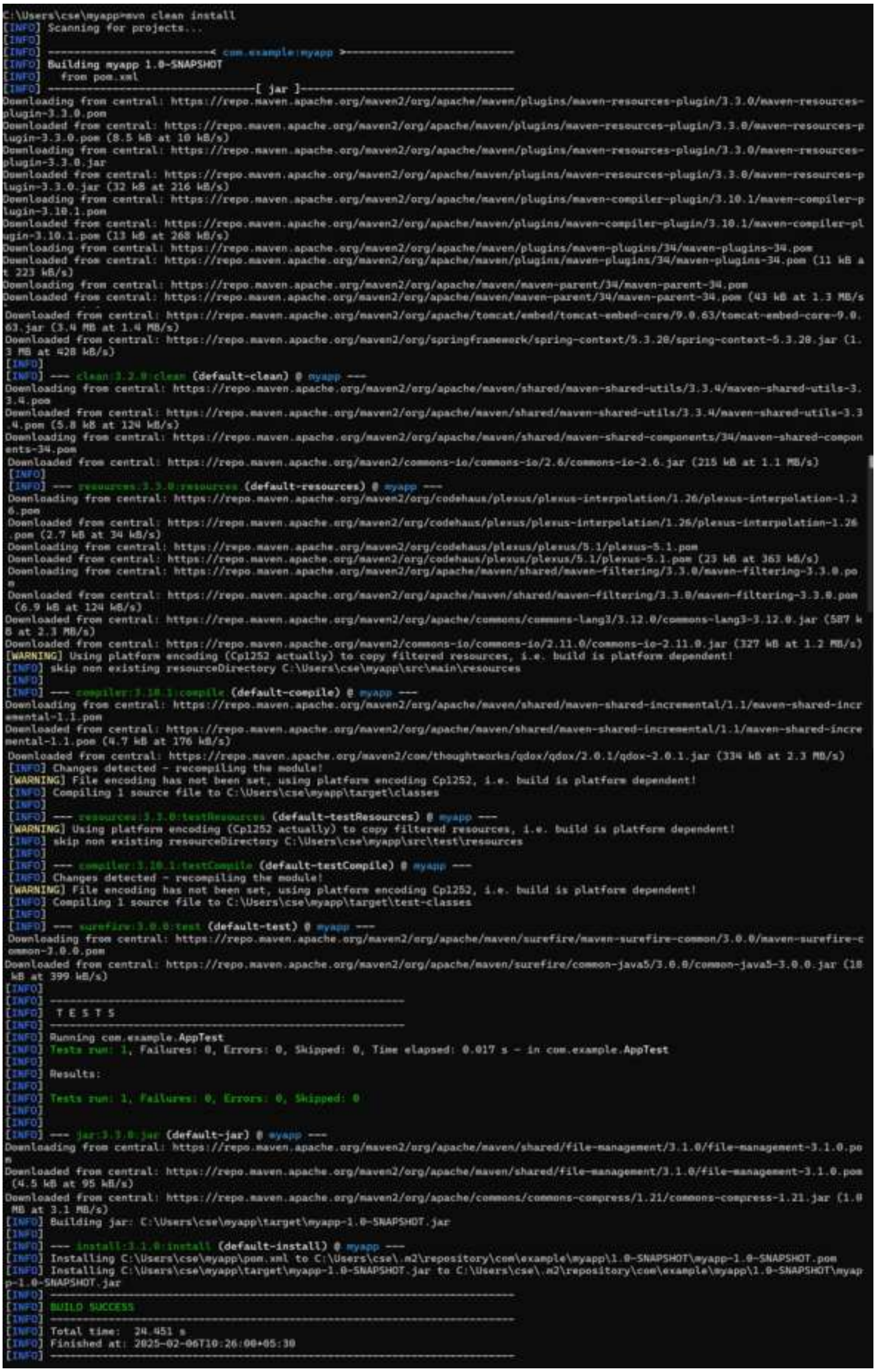
To use Spring Boot, add the following inside the <dependencies> section:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.7.0</version>
</dependency>
```

After adding dependencies, run:

```
mvn clean install
```

This downloads and installs required JAR files.



```
C:\Users\cse\myapp>mvn clean install
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------------< com.example:myapp >-------------------------
[INFO] Building myapp 1.0-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------[ jar ]---------------------------------
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-resources-plugin/3.3.0/maven-resources-
plugin-3.3.0.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-resources-plugin/3.3.0/maven-resources-p
lugin-3.3.0.pom (8.5 kB at 10 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-resources-plugin/3.3.0/maven-resources-
plugin-3.3.0.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-resources-plugin/3.3.0/maven-resources-p
lugin-3.3.0.jar (32 kB at 216 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-compiler-plugin/3.10.1/maven-compiler-p
lugin-3.10.1.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-compiler-plugin/3.10.1/maven-compiler-pl
ugin-3.10.1.pom (13 kB at 268 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/34/maven-plugins-34.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/34/maven-plugins-34.pom (11 kB a
t 223 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/34/maven-parent-34.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/34/maven-parent-34.pom (43 kB at 1.3 MB/s
 Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/tomcat/embed/tomcat-embed-core/9.0.63/tomcat-embed-core-9.0.
 63.jar (3.4 MB at 1.4 MB/s)
 Downloaded from central: https://repo.maven.apache.org/maven2/org/springframework/spring-context/5.3.20/spring-context-5.3.20.jar (1.
 3 MB at 428 kB/s)
 [INFO]
 [INFO] --- clean:3.2.0:clean (default-clean) @ myapp ---
 Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/shared/maven-shared-utils/3.3.4/maven-shared-utils-3.
 3.4.pom
 Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/shared/maven-shared-utils/3.3.4/maven-shared-utils-3.3
 .4.pom (5.8 kB at 124 kB/s)
 Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/shared/maven-shared-components/34/maven-shared-compon
 ents-34.pom
  Downloaded from central: https://repo.maven.apache.org/maven2/commons-io/commons-io/2.6/commons-io-2.6.jar (215 kB at 1.1 MB/s)
  [INFO]
  [INFO] --- resources:3.3.0:resources (default-resources) @ myapp ---
  Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-interpolation/1.26/plexus-interpolation-1.2
  6.pom
  Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-interpolation/1.26/plexus-interpolation-1.26
  .pom (2.7 kB at 34 kB/s)
  Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus/5.1/plexus-5.1.pom
  Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus/5.1/plexus-5.1.pom (23 kB at 363 kB/s)
  Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/shared/maven-filtering/3.3.0/maven-filtering-3.3.0.po
  m
  Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/shared/maven-filtering/3.3.0/maven-filtering-3.3.0.pom
   (6.9 kB at 124 kB/s)
 Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/commons/commons-lang3/3.12.0/commons-lang3-3.12.0.jar (587 k
 B at 2.3 MB/s)
 Downloaded from central: https://repo.maven.apache.org/maven2/commons-io/commons-io/2.11.0/commons-io-2.11.0.jar (327 kB at 1.2 MB/s)
 [WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!
 [INFO] skip non existing resourceDirectory C:\Users\cse\myapp\src\main\resources
 [INFO]
 [INFO] --- compiler:3.10.1:compile (default-compile) @ myapp ---
 Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/shared/maven-shared-incremental/1.1/maven-shared-incr
 emental-1.1.pom
 Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/shared/maven-shared-incremental/1.1/maven-shared-incre
 mental-1.1.pom (4.7 kB at 176 kB/s)
  Downloaded from central: https://repo.maven.apache.org/maven2/com/thoughtworks/qdox/qdox/2.0.1/qdox-2.0.1.jar (334 kB at 2.3 MB/s)
  [INFO] Changes detected - recompiling the module!
  [WARNING] File encoding has not been set, using platform encoding Cp1252, i.e. build is platform dependent!
  [INFO] Compiling 1 source file to C:\Users\cse\myapp\target\classes
  [INFO]
  [INFO] --- resources:3.3.0:testResources (default-testResources) @ myapp ---
  [WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!
  [INFO] skip non existing resourceDirectory C:\Users\cse\myapp\src\test\resources
  [INFO]
  [INFO] --- compiler:3.10.1:testCompile (default-testCompile) @ myapp ---
  [INFO] Changes detected - recompiling the module!
  [WARNING] File encoding has not been set, using platform encoding Cp1252, i.e. build is platform dependent!
  [INFO] Compiling 1 source file to C:\Users\cse\myapp\target\test-classes
  [INFO]
  [INFO] --- surefire:3.0.0:test (default-test) @ myapp ---
  Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/surefire/maven-surefire-common/3.0.0/maven-surefire-c
  ommon-3.0.0.pom
 Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/surefire/common-java5/3.0.0/common-java5-3.0.0.jar (18
  kB at 399 kB/s)
 [INFO]
 [INFO] -------------------------------------------------------
 [INFO]  T E S T S
 [INFO] -------------------------------------------------------
 [INFO] Running com.example.AppTest
 [INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.017 s - in com.example.AppTest
 [INFO]
 [INFO] Results:
 [INFO]
 [INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
 [INFO]
 [INFO]
 [INFO] --- jar:3.3.0:jar (default-jar) @ myapp ---
 Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/shared/file-management/3.1.0/file-management-3.1.0.po
 m
 Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/shared/file-management/3.1.0/file-management-3.1.0.pom
  (4.5 kB at 95 kB/s)
 Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/commons/commons-compress/1.21/commons-compress-1.21.jar (1.0
  MB at 3.1 MB/s)
 [INFO] Building jar: C:\Users\cse\myapp\target\myapp-1.0-SNAPSHOT.jar
 [INFO]
 [INFO] --- install:3.1.0:install (default-install) @ myapp ---
 [INFO] Installing C:\Users\cse\myapp\pom.xml to C:\Users\cse\.m2\repository\com\example\myapp\1.0-SNAPSHOT\myapp-1.0-SNAPSHOT.pom
 [INFO] Installing C:\Users\cse\myapp\target\myapp-1.0-SNAPSHOT.jar to C:\Users\cse\.m2\repository\com\example\myapp\1.0-SNAPSHOT\myap
p-1.0-SNAPSHOT.jar
 [INFO] ------------------------------------------------------------------------
 [INFO] BUILD SUCCESS
 [INFO] ------------------------------------------------------------------------
 [INFO] Total time:  24.451 s
 [INFO] Finished at: 2025-02-06T10:26:00+05:30
 [INFO] ------------------------------------------------------------------------
```

Using Maven Plugins

Maven plugins automate tasks like compiling code, running tests, and packaging the application.

Common Maven Plugins

Plugin	**Purpose**
maven-compiler-plugin	Compiles Java code
maven-surefire-plugin	Runs unit tests
maven-jar-plugin	Packages the project as a JAR file

Practically Using Maven Plugins in a Project

To compile code, run unit tests, and package the application using Maven plugins, follow this step-by-step guide.

Step1: Create Maven Project (use the above procedure)

Step2: Configuring Maven Plugins in pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.9.0</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.9.0</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>2.7.0</version>
      </dependency>
</dependencies>

<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>

      <!-- Surefire Plugin for Unit Testing -->
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-surefire-plugin</artifactId>
         <version>3.0.0-M7</version>
      </plugin>

      <!-- JAR Plugin with Manifest Configuration -->
      <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>
```

This configures Maven to:

✅ Compile Java code using the maven-compiler-plugin.

✅ Run unit tests using the maven-surefire-plugin.

✅ Package the application into a JAR file using the maven-jar-plugin.

Step3: Writing a Sample Java Program (App.java)

```
package com.example;
public class App {
   public static String greet() {
      return "Hello, DevOps!";
   }
   public static void main(String[] args) {
      System.out.println(greet());
   }
}
```

Step4: Writing a Unit Test (AppTest.java)

Modify the AppTest.java file inside src/test/java/com/example/ to test the greet method:

```
package com.example;
```

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;
public class AppTest {
    @Test
    public void testGreet() {
        assertEquals("Hello, DevOps!", App.greet());
    }
}
```

Here:

✅ We use JUnit to test the greet() method.

✅ The assertEquals() method checks if the output matches "Hello, DevOps!".

Step5: Running Maven Commands

a. mvn compile (compile the code)

If successful, you will see

```
[INFO] Compiling 1 source file to target/classes
```

b. mvn test (run unit tests)

Expected output (if the test passes)

```
[INFO] Running com.example.AppTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.01 sec
[INFO] BUILD SUCCESS
```

If the test fails, check the assertion inside AppTest.java.

c. mvn package (package the application)

This will generate the output

```
[INFO] Building jar: /path-to-project/myapp/target/myapp-1.0-SNAPSHOT.jar
```

The JAR file will be inside the target/ folder.

d. java -jar target/myapp-1.0-SNAPSHOT.jar (run the jar file)

Output

```
Hello, DevOps!
```

3. Working with Gradle: Setting Up a Gradle Project, Understanding Build Scripts (Groovy and Kotlin DSL), Dependency Management and Task Automation

What is Gradle?

Gradle is a modern, flexible, and high-performance build automation tool used in Java, Kotlin, and Android projects. It is designed to optimize build performance using incremental builds and task caching.

Why Use Gradle?

- Faster than Maven due to incremental builds.
- Uses Groovy/Kotlin DSL instead of XML.
- Highly customizable build automation.
- Supports dependency management like Maven.
- Works with multiple languages (Java, Kotlin, C++, Python, etc.).

Setting Up a Gradle Project

Step 1: Install Gradle (If Not Installed)

Check if Gradle is installed using:

```
gradle -version
```

If not installed, follow Exercise 1 for installation steps.

Step 2: Create a New Gradle Project

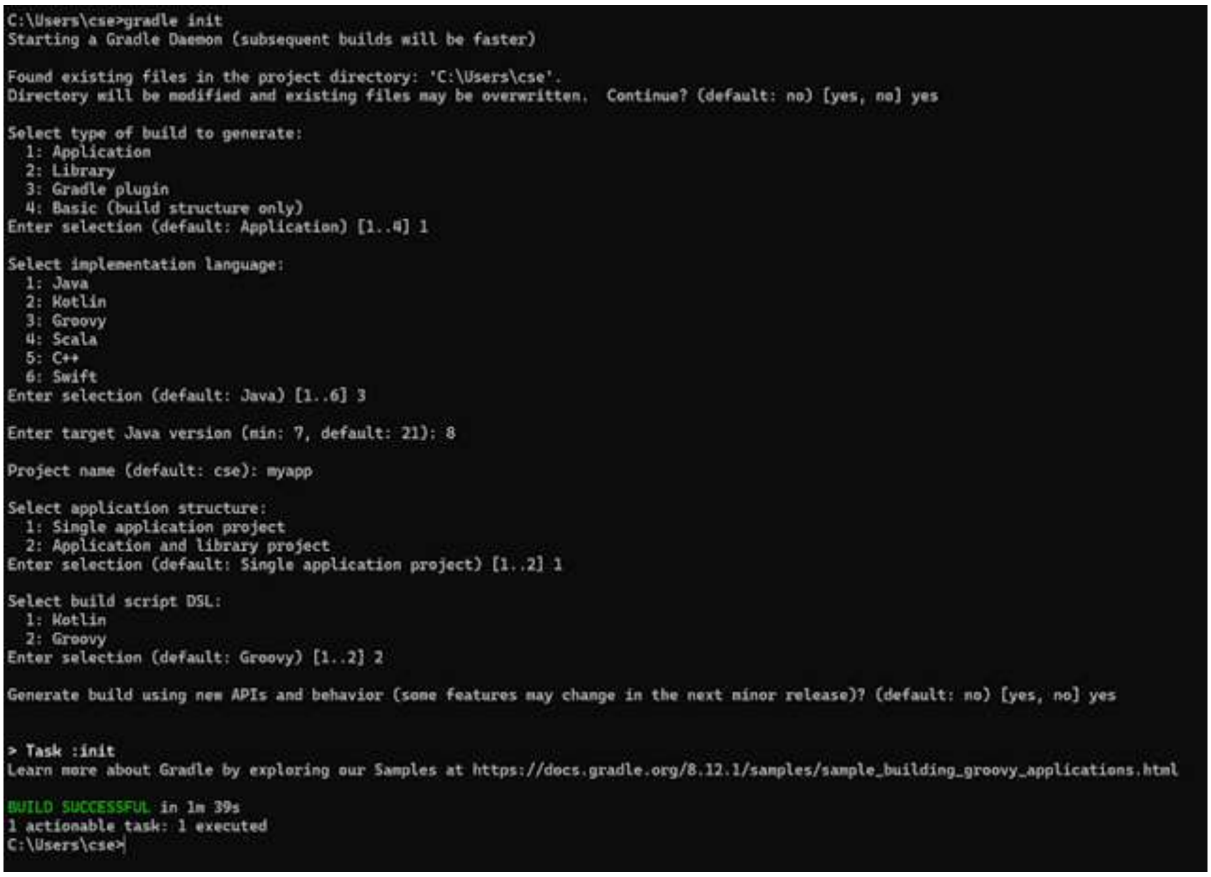
To create a Java-based Gradle project, run:

```
mkdir myapp

cd myapp

gradle init
```

Follow the prompts:

- **Select Project Type:** Choose application.
- **Select Build Script Language:** Choose Groovy or Kotlin.
- **Select Java Version:** Choose 8, 11, 17, or latest.

Output:



```
C:\Users\cse>gradle init
Starting a Gradle Daemon (subsequent builds will be faster)

Found existing files in the project directory: 'C:\Users\cse'.
Directory will be modified and existing files may be overwritten.  Continue? (default: no) [yes, no] yes

Select type of build to generate:
  1: Application
  2: Library
  3: Gradle plugin
  4: Basic (build structure only)
Enter selection (default: Application) [1..4] 1

Select implementation language:
  1: Java
  2: Kotlin
  3: Groovy
  4: Scala
  5: C++
  6: Swift
Enter selection (default: Java) [1..6] 3

Enter target Java version (min: 7, default: 21): 8

Project name (default: cse): myapp

Select application structure:
  1: Single application project
  2: Application and library project
Enter selection (default: Single application project) [1..2] 1

Select build script DSL:
  1: Kotlin
  2: Groovy
Enter selection (default: Groovy) [1..2] 2

Generate build using new APIs and behavior (some features may change in the next minor release)? (default: no) [yes, no] yes


> Task :init
Learn more about Gradle by exploring our Samples at https://docs.gradle.org/8.12.1/samples/sample_building_groovy_applications.html

BUILD SUCCESSFUL in 1m 39s
1 actionable task: 1 executed
C:\Users\cse>
```

Once the setup is complete, navigate to the project folder:

Step 3: Verify the Project Structure

Gradle creates a structured project like this:

```
myapp

|— src

|    ├— main

|    |    └— java

|    |        └— com.example

|    |            └— App.java

|    ├— test

|    |    └— java

|    |        └— com.example

|    |            └— AppTest.java

|— build.gradle  (or build.gradle.kts)
```

|— settings.gradle

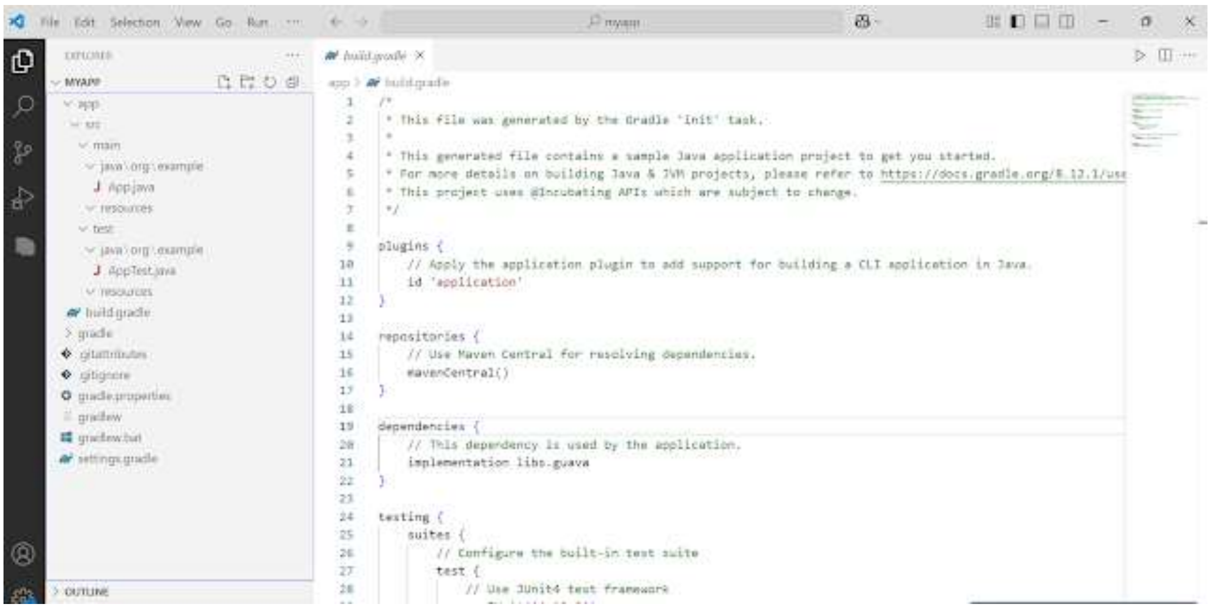
|— gradlew

|— gradlew.bat

|— gradle/

- src/main/java → Contains main application code.
- src/test/java → Contains test cases.
- build.gradle or build.gradle.kts → The Gradle build script (Groovy or Kotlin).

Output:



```
/*
 * This file was generated by the Gradle 'init' task.
 *
 * This generated file contains a sample Java application project to get you started.
 * For more details on building Java & JVM projects, please refer to https://docs.gradle.org/8.12.1/use
 * This project uses @Incubating APIs which are subject to change.
 */

plugins {
    // Apply the application plugin to add support for building a CLI application in Java.
    id 'application'
}

repositories {
    // Use Maven Central for resolving dependencies.
    mavenCentral()
}

dependencies {
    // This dependency is used by the application.
    implementation libs.guava
}

testing {
    suites {
        // Configure the built-in test suite
        test {
            // Use JUnit4 test framework
```

Understanding Gradle Build Scripts

Gradle supports two script languages:

1. Groovy DSL (build.gradle) – Default scripting language.
2. Kotlin DSL (build.gradle.kts) – Modern, type-safe alternative.

Groovy DSL (build.gradle) Example

```
plugins {
   id 'java'
}


repositories {
   mavenCentral()
}
```

```
dependencies {
   implementation 'org.springframework.boot:spring-boot-starter-web:2.7.0'
   testImplementation 'junit:junit:4.13.2'
}

tasks.register('hello') {
   doLast {
      println 'Hello, DevOps with Gradle!'
   }
}
```

Kotlin DSL (build.gradle.kts) Example

```
plugins {
   java
}

repositories {
   mavenCentral()
}

dependencies {
   implementation("org.springframework.boot:spring-boot-starter-web:2.7.0")
   testImplementation("junit:junit:4.13.2")
}

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

Key Sections in the Build Script

- plugins → Specifies project type (e.g., java).
- repositories → Defines where dependencies are downloaded from.
- dependencies → Lists required libraries.
- tasks → Automates build tasks.

Managing Dependencies in Gradle

Gradle uses dependency management similar to Maven.

Adding Dependencies

Inside build.gradle or build.gradle.kts:

```
dependencies {
   // Add Guava dependency correctly
   implementation("com.google.guava:guava:31.1-jre")
   // Spring Boot Starter Web
   implementation("org.springframework.boot:spring-boot-starter-web:2.7.0")
   // JUnit 4 for testing
   testImplementation("junit:junit:4.13.2")
}
```

After adding dependencies, refresh Gradle using:

```
gradle build
```

This downloads and caches required JAR files.

Automating Tasks in Gradle

Gradle tasks automate common actions like compiling code, running tests, and packaging the project.

Common Gradle Tasks

Command	**Purpose**
gradle build	Builds the project
gradle test	Runs unit tests
gradle run	Runs the application (if an application plugin is used)
gradle clean	Deletes previous build files

gradle dependencies	Lists all project dependencies (Note: 📌 Make sure you are inside mygapp/ where build.gradle.kts is located.)
gradle hello	Runs a custom task (from build.gradle)

Running a Custom Task

add the following to your build.gradle.kts file:

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

Run the command:

```
gradle hello
```

Expected Output:

```
Hello, DevOps with Gradle!
```

Running and Packaging the Gradle Project

Step 1: Compile the Code

```
gradle compileJava
```

This compiles the source files in src/main/java.

Step 2: Run Unit Tests

```
gradle test
```

If all tests pass, you'll see:

```
BUILD SUCCESSFUL
```

Step 3: Package the Project into a JAR

```
gradle jar
```

This creates a .jar file inside the build/libs/ directory.

Step 4: Run the Application

```
java -jar build/libs/myapp.jar
```

Expected Output:

```
Hello, DevOps with Gradle!
```

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

Part 1: Build and Run a Java Application Using Maven

Step 1: Create a Maven Project

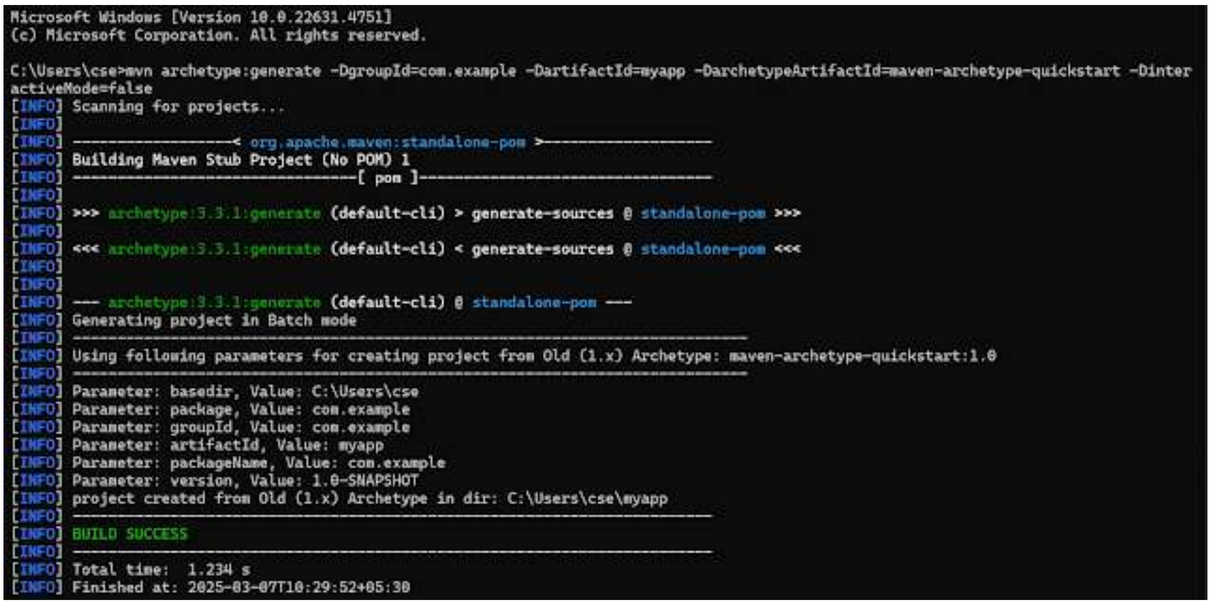
Open a terminal and run:

mvn archetype:generate -DgroupId=com.example -DartifactId=myapp -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

Then, navigate into the project folder:

cd myapp

OUTPUT:



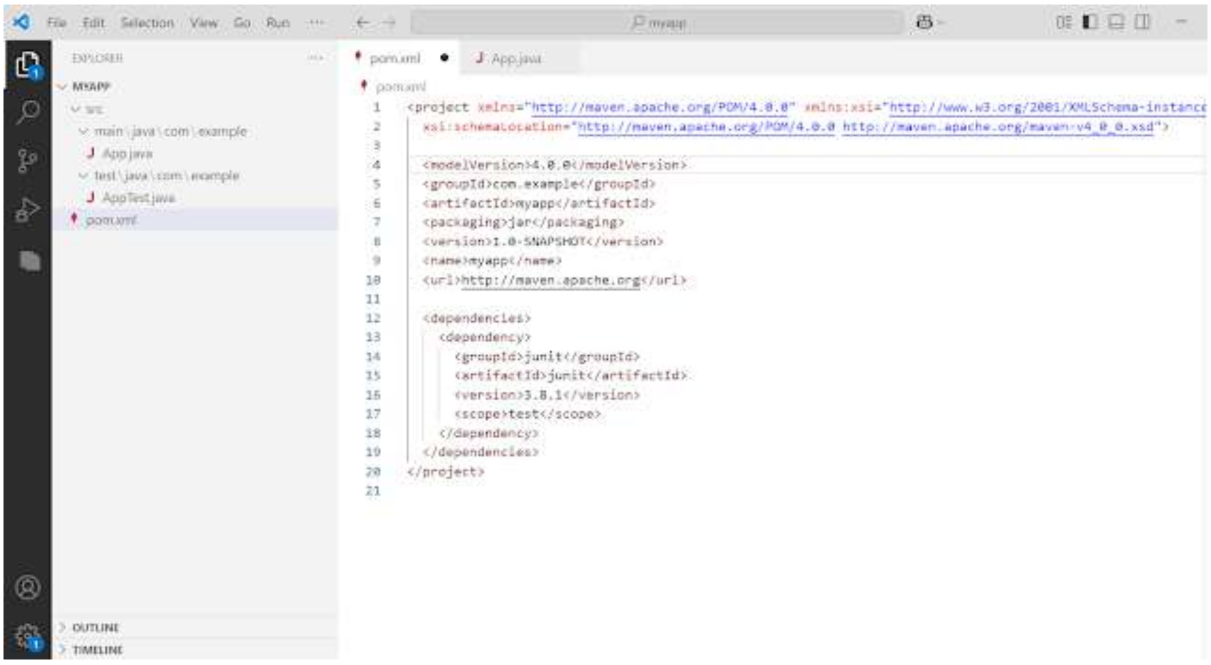
```
Microsoft Windows [Version 10.0.22631.4751]
(c) Microsoft Corporation. All rights reserved.

C:\Users\cse>mvn archetype:generate -DgroupId=com.example -DartifactId=myapp -DarchetypeArtifactId=maven-archetype-quickstart -Dinter
activeMode=false
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------< org.apache.maven:standalone-pom >-------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] --------------------------------[ pom ]---------------------------------
[INFO]
[INFO] >>> archetype:3.3.1:generate (default-cli) > generate-sources @ standalone-pom >>>
[INFO]
[INFO] <<< archetype:3.3.1:generate (default-cli) < generate-sources @ standalone-pom <<<
[INFO]
[INFO]
[INFO] --- archetype:3.3.1:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Batch mode
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Old (1.x) Archetype: maven-archetype-quickstart:1.0
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: basedir, Value: C:\Users\cse
[INFO] Parameter: package, Value: com.example
[INFO] Parameter: groupId, Value: com.example
[INFO] Parameter: artifactId, Value: myapp
[INFO] Parameter: packageName, Value: com.example
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: C:\Users\cse\myapp
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.234 s
[INFO] Finished at: 2025-03-07T10:29:52+05:30
```

Step 2: Verify Project Structure

After running the command, your project will have this structure:

```
myapp
|— src
|   ├── main
|   |   └── java
|   |       └── com.example
|   |           └── App.java
|   ├── test
|   |   └── java
|   |       └── com.example
|   |           └── AppTest.java
|── pom.xml
```

OUTPUT:



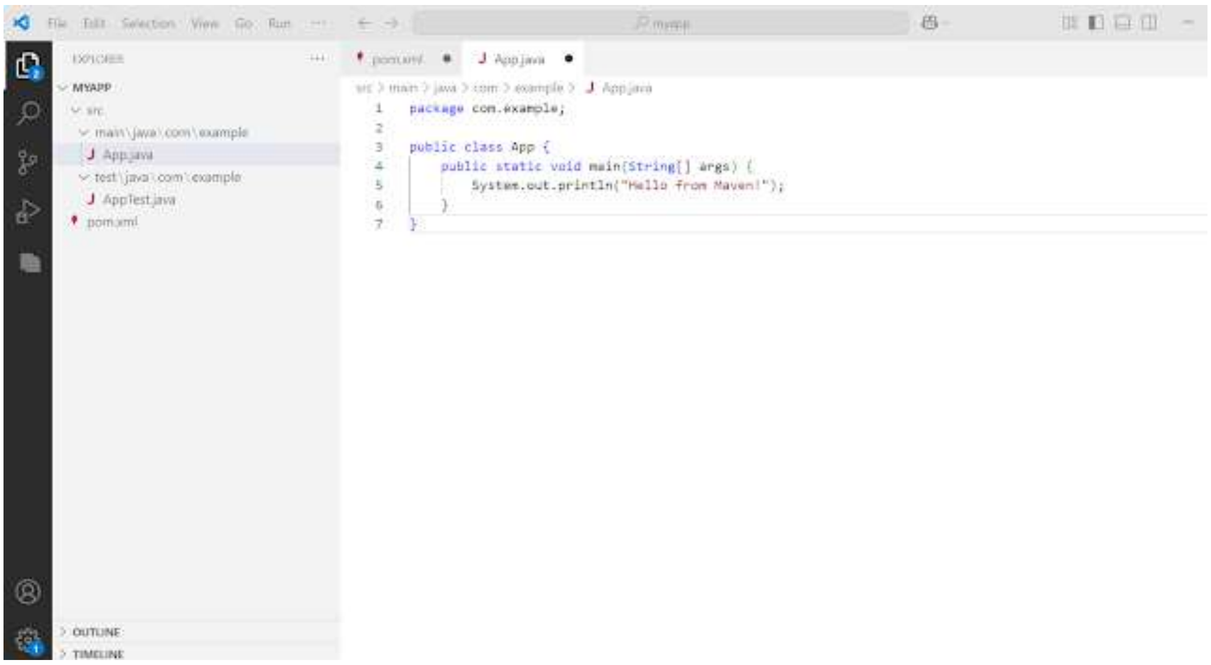
```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>myapp</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>myapp</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Step 3: Update App.java (Main Application)

Modify src/main/java/com/example/App.java:

```
package com.example;


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

OUTPUT:



```java
package com.example;

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

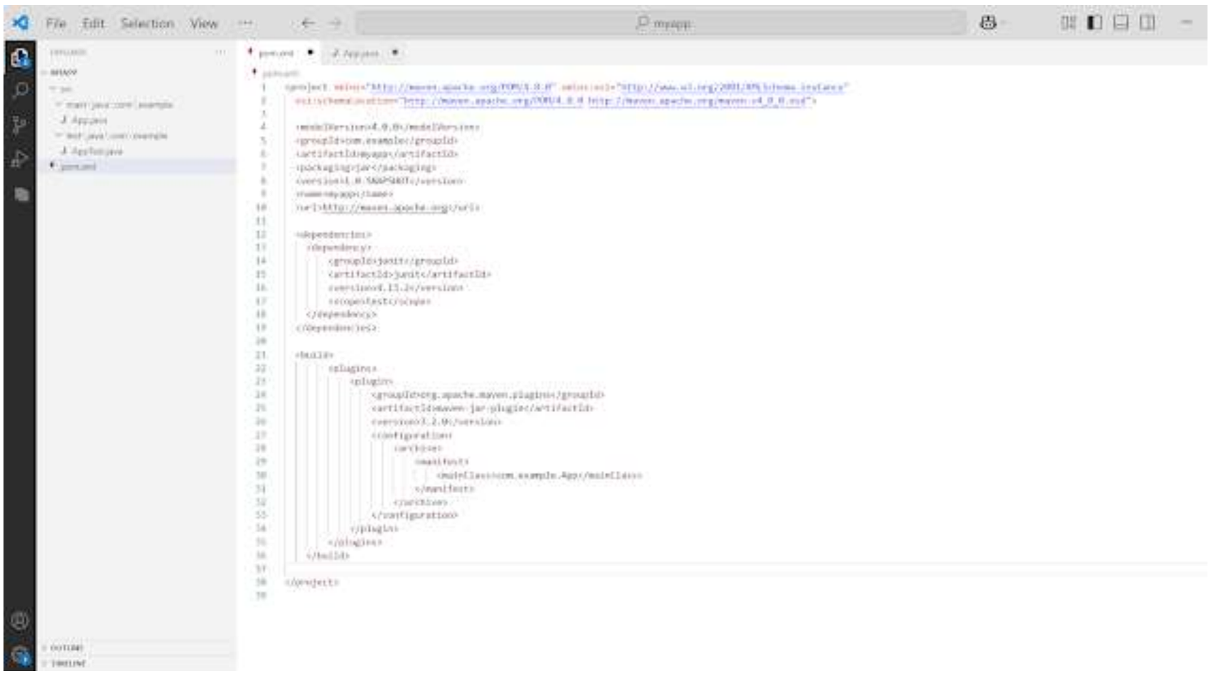
Step 4: Configure pom.xml for JUnit and Plugins

Modify pom.xml:

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

  <groupId>com.example</groupId>
  <artifactId>myapp</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13.2</version>
```

```
        <scope>test</scope>
      </dependency>
    </dependencies>

    <build>
      <plugins>
        <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>
</project>
```

OUTPUT:



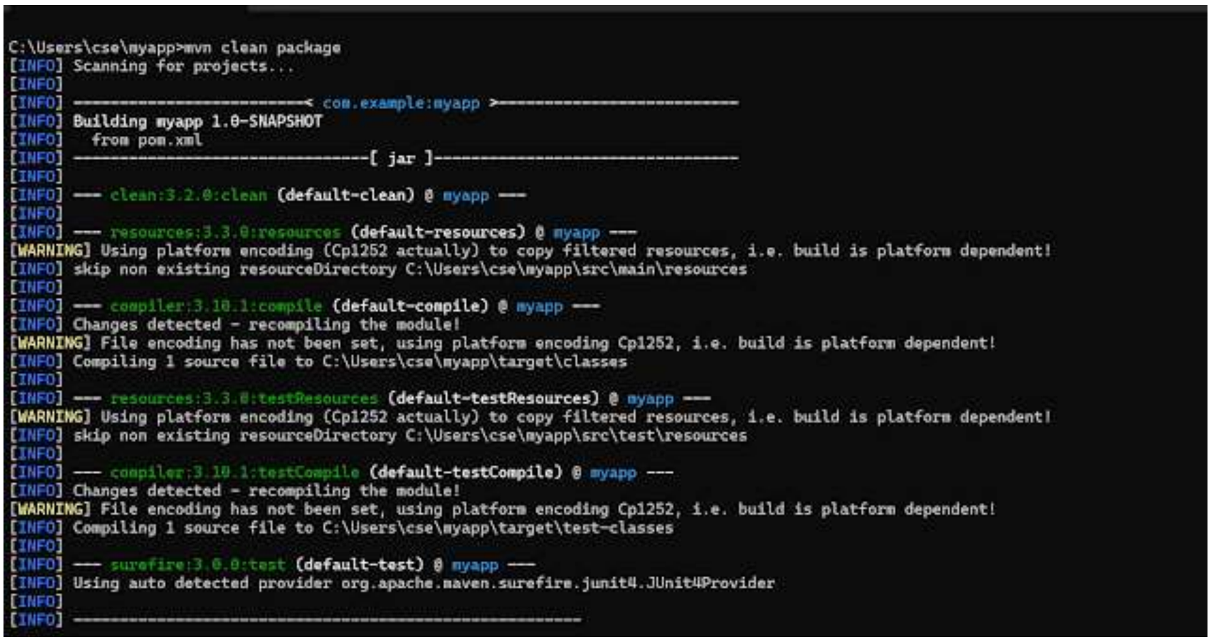
```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>myapp</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>myapp</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
        <plugins>
            <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>

</project>
```

Step 5: Build and Run the Application with Maven

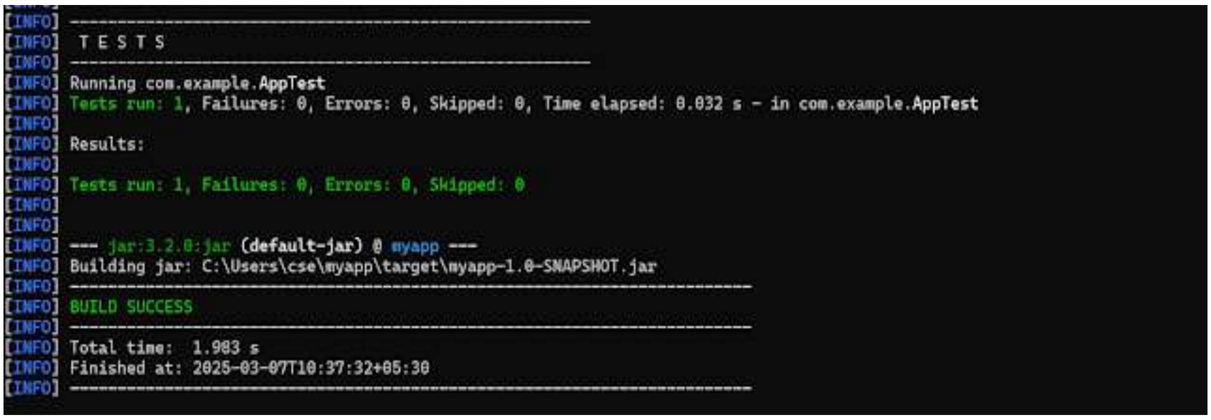
Compile and Build the JAR

mvn clean package

OUTPUT:



```
C:\Users\cse\myapp>mvn clean package
[INFO] Scanning for projects...
[INFO]
[INFO] -----------------------< com.example:myapp >------------------------
[INFO] Building myapp 1.0-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- clean:3.2.0:clean (default-clean) @ myapp ---
[INFO]
[INFO] --- resources:3.3.0:resources (default-resources) @ myapp ---
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\Users\cse\myapp\src\main\resources
[INFO]
[INFO] --- compiler:3.10.1:compile (default-compile) @ myapp ---
[INFO] Changes detected - recompiling the module!
[WARNING] File encoding has not been set, using platform encoding Cp1252, i.e. build is platform dependent!
[INFO] Compiling 1 source file to C:\Users\cse\myapp\target\classes
[INFO]
[INFO] --- resources:3.3.0:testResources (default-testResources) @ myapp ---
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory C:\Users\cse\myapp\src\test\resources
[INFO]
[INFO] --- compiler:3.10.1:testCompile (default-testCompile) @ myapp ---
[INFO] Changes detected - recompiling the module!
[WARNING] File encoding has not been set, using platform encoding Cp1252, i.e. build is platform dependent!
[INFO] Compiling 1 source file to C:\Users\cse\myapp\target\test-classes
[INFO]
[INFO] --- surefire:3.0.0:test (default-test) @ myapp ---
[INFO] Using auto detected provider org.apache.maven.surefire.junit4.JUnit4Provider
[INFO]
[INFO] -------------------------------------------------------
```



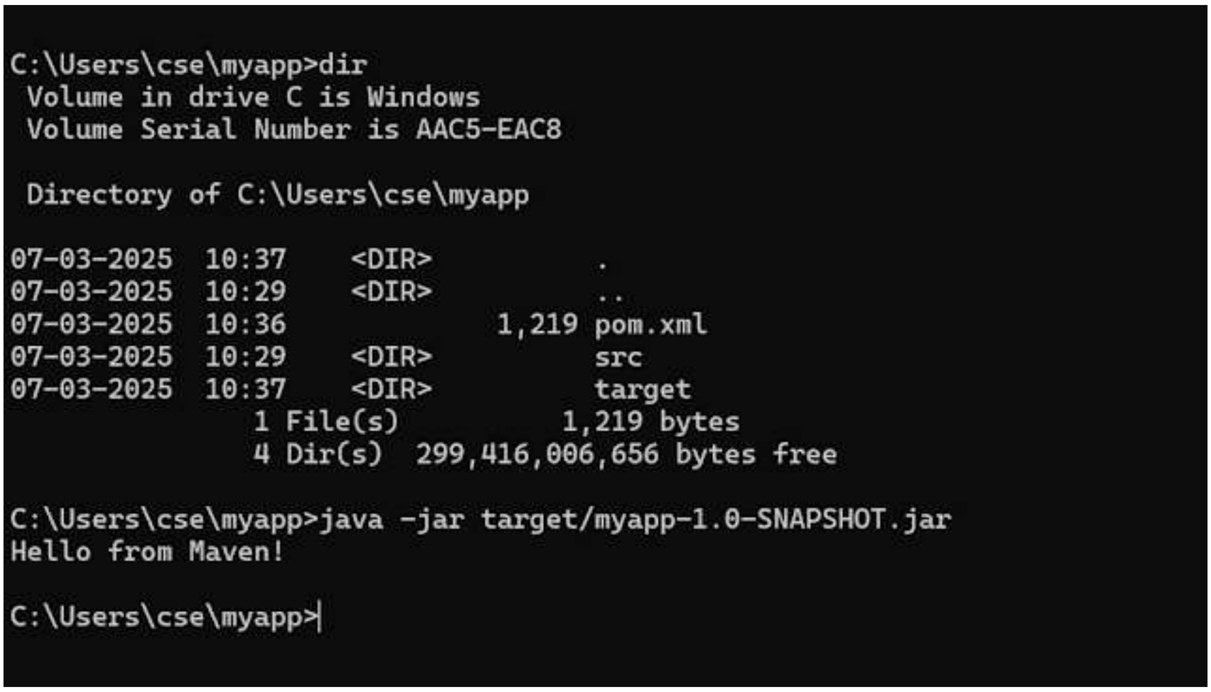
```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running com.example.AppTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.032 s - in com.example.AppTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- jar:3.2.0:jar (default-jar) @ myapp ---
[INFO] Building jar: C:\Users\cse\myapp\target\myapp-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.983 s
[INFO] Finished at: 2025-03-07T10:37:32+05:30
[INFO] ------------------------------------------------------------------------
```

Run the Application

java -jar target/myapp-1.0-SNAPSHOT.jar

Expected Output:

Hello from Maven!



```
C:\Users\cse\myapp>dir
 Volume in drive C is Windows
 Volume Serial Number is AAC5-EAC8

 Directory of C:\Users\cse\myapp

07-03-2025  10:37    <DIR>          .
07-03-2025  10:29    <DIR>          ..
07-03-2025  10:36             1,219 pom.xml
07-03-2025  10:29    <DIR>          src
07-03-2025  10:37    <DIR>          target
               1 File(s)          1,219 bytes
               4 Dir(s)  299,416,006,656 bytes free

C:\Users\cse\myapp>java -jar target/myapp-1.0-SNAPSHOT.jar
Hello from Maven!

C:\Users\cse\myapp>
```

Part 2: Migrate the Same Application to Gradle

Step 1: Create a New Gradle Project

Navigate outside your Maven project folder and run:

mkdir myapp-gradle

cd myapp-gradle

gradle init

Step 2: Copy Java Files from Maven to Gradle

Manually copy the src folder from myapp (Maven project) to myapp-gradle.

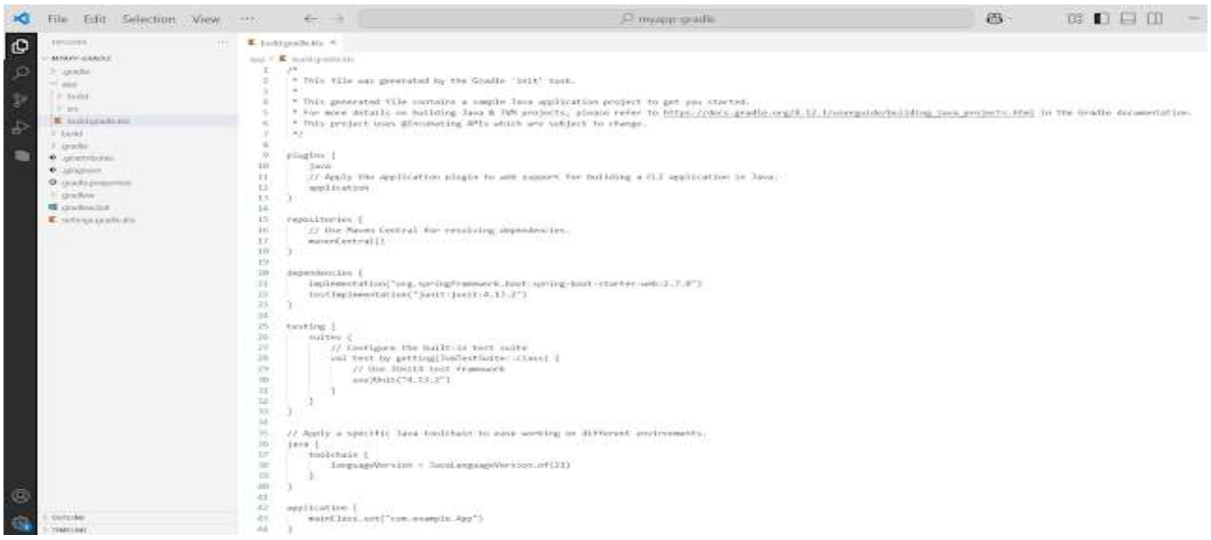
(Note: replace myapp-gradle src folder with myapp src folder)

Step 3: Modify build.gradle.kts (Kotlin DSL)

Edit build.gradle.kts:

```
plugins {
   java
   application
}
repositories {
   mavenCentral()
}
dependencies {
   implementation("org.springframework.boot:spring-boot-starter-web:2.7.0")
   testImplementation("junit:junit:4.13.2")
}
application {
   mainClass.set("com.example.App")
}
```

OUTPUT:



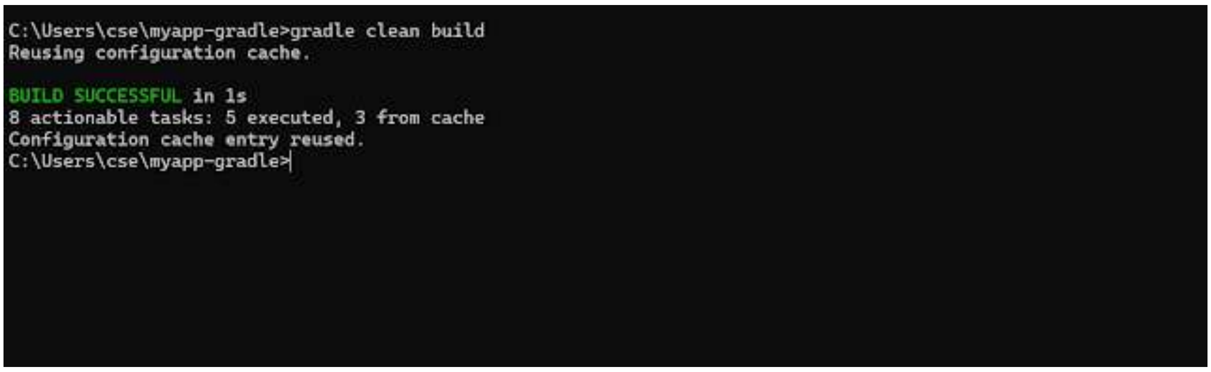
```
/*
 * This file was generated by the Gradle 'init' task.
 *
 * This generated file contains a sample Java application project to get you started.
 * For more details on building Java & JVM projects, please refer to https://docs.gradle.org/8.12.1/userguide/building_java_projects.html in the Gradle documentation.
 * This project uses @Incubating APIs which are subject to change.
 */

plugins {
    java
    // Apply the application plugin to add support for building a CLI application in Java.
    application
}

repositories {
    // Use Maven Central for resolving dependencies.
    mavenCentral()
}

dependencies {
    implementation("org.springframework.boot:spring-boot-starter-web:2.7.0")
    testImplementation("junit:junit:4.13.2")
}

testing {
    suites {
        // Configure the built-in test suite
        val test by getting(JvmTestSuite::class) {
            // Use JUnit4 test framework
            useJUnit("4.13.2")
        }
    }
}

// Apply a specific Java toolchain to ease working on different environments.
java {
    toolchain {
        languageVersion = JavaLanguageVersion.of(21)
    }
}

application {
    mainClass.set("com.example.App")
}
```

Step 4: Build and Run the Application with Gradle

Compile and Build the JAR

gradle clean build

OUTPUT:



```
C:\Users\cse\myapp-gradle>gradle clean build
Reusing configuration cache.

BUILD SUCCESSFUL in 1s
8 actionable tasks: 5 executed, 3 from cache
Configuration cache entry reused.
C:\Users\cse\myapp-gradle>
```

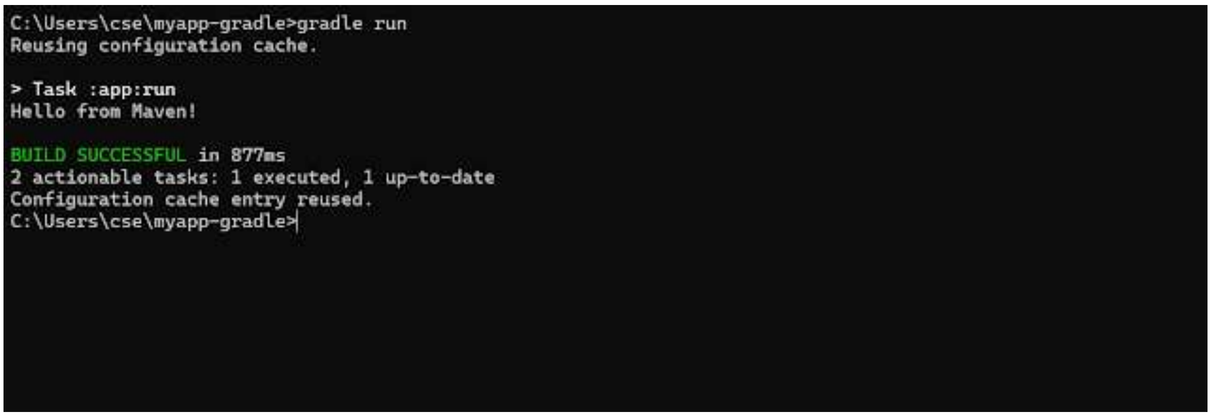
Run the Application

gradle run

Expected Output:

Hello from Maven!

OUTPUT:



```
C:\Users\cse\myapp-gradle>gradle run
Reusing configuration cache.

> Task :app:run
Hello from Maven!

BUILD SUCCESSFUL in 877ms
2 actionable tasks: 1 executed, 1 up-to-date
Configuration cache entry reused.
C:\Users\cse\myapp-gradle>
```

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

1. Introduction to Jenkins

Jenkins is an open-source automation server used for Continuous Integration and Continuous Deployment (CI/CD). It allows developers to automate repetitive tasks such as building, testing, and deploying software efficiently.

1.1 Features of Jenkins

- Open-source and widely used for CI/CD.
- Supports plugins for extensibility.
- Works with various build tools like Maven and Gradle.
- Can be deployed on local machines or cloud environments.
- Provides a web-based GUI for easy management.
- Allows job scheduling and pipeline execution.

1.2 Jenkins Architecture

- Master Node: Manages Jenkins configurations and distributes workloads to agents.
- Agent Node: Executes assigned jobs.
- Plugins: Extend Jenkins functionality.
- Pipelines: Define CI/CD workflows.

2. Installing Jenkins on Local or Cloud Environment

2.1 Prerequisites

- Java (JDK 11 or newer)
- System requirements: Minimum 1GB RAM, 1GHz CPU
- Internet connection for downloading Jenkins and dependencies

2.2 Installing Jenkins on Local Machine (Windows/Linux/Mac)

Step 1: Install Java

- Download and install Java JDK 11 or later.
- Verify installation:

```
java -version
```

Step 2: Download and Install Jenkins

- Download Jenkins from Jenkins official website.
- Install Jenkins based on OS:
    - Windows: Run the .msi installer and follow the setup wizard.

Step 3: Start Jenkins

- Windows: Jenkins runs as a Windows service

Step 4: Access Jenkins Web Interface

- Open a browser and go to:
- http://localhost:8080
- Retrieve the initial admin password:
- sudo cat /var/lib/jenkins/secrets/initialAdminPassword
- Paste the password in the browser prompt.

3. Configuring Jenkins for First Use

Step 1: Initial Setup

1. After logging in, Jenkins will prompt for plugin installation.
2. Select "Install suggested plugins".
3. Create an admin user with username, password, and email.

Step 2: Configure System Settings

1. Navigate to Manage Jenkins > Configure System.
2. Set up system-wide configurations such as Java path and build tool paths.

Step 3: Configure Nodes (Optional)

1. Go to Manage Jenkins > Manage Nodes and Clouds.
2. Add and configure additional agent nodes.

Step 4: Verify Installation

1. Create a sample job:
    - Go to New Item > Freestyle project.
    - Add a build step (e.g., Execute shell command echo "Hello Jenkins").
    - Click Save and Build Now.
2. Check the build logs for output.

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

Prerequisites

Before proceeding, ensure you have the following installed:

- Java Development Kit (JDK) 8 or higher
- Jenkins (Installed and configured on a local or cloud environment)
- Maven or Gradle
- Git (Installed and configured with a repository)

Step 1: Install Required Jenkins Plugins

To integrate Jenkins with Maven/Gradle and automate builds, install the following plugins:

1. Navigate to Jenkins Dashboard > Manage Jenkins > Manage Plugins.
2. Under the Available Plugins tab, search for and install:
   - Pipeline Maven Plugin Database (ID: pipeline-maven)
   - Gradle Plugin (ID: gradle)
   - Git Plugin (ID: git)
3. Restart Jenkins to apply the changes

Note: if the Plugins are not shown in Available Plugins and shown in installed plugin tab, then it indicates that plugins are already installed.

Step 2: Configure Maven/Gradle in Jenkins

For Maven:

1. Go to Manage Jenkins > Global Tool Configuration.
2. Scroll to the Maven section and click Add Maven.
3. Provide a Name (e.g., "Maven 3.8") and set the Maven Home directory path.
4. Click Save.

For Gradle:

1. Navigate to Manage Jenkins > Global Tool Configuration.
2. Scroll to the Gradle section and click Add Gradle.
3. Provide a Name (e.g., "Gradle 7.3") and set the Gradle Home path.
4. Click Save.

Step 3: Create a Jenkins Job

1. From the Jenkins dashboard, click New Item.
2. Enter a name for the project (e.g., "My-CI-Pipeline").
3. Select Freestyle Project or Pipeline (depending on your requirement).
4. Click OK to create the job.

Step 4: Configure Source Code Management (SCM) with Git

To pull the source code from a Git repository:

Example Project:

We will use a sample Maven Java project hosted on GitHub:

- Repository: https://github.com/spring-projects/spring-petclinic.git

1. In the job configuration page, navigate to Source Code Management.
2. Select Git.
3. Enter the repository URL:

   https://github.com/spring-projects/spring-petclinic.git

4. Under Branch Specifier, enter */main (or */master if applicable).
5. If the repository is private, click Add Credentials and enter your GitHub username and access token.
6. Click Save.

Step 5: Configure Build Steps

For Maven Projects:

1. Scroll to the Build section and click Add Build Step > Invoke Top-Level Maven Targets.
2. In the Goals field, enter:

1. clean install
2. Click Save.

For Gradle Projects:

1. Click Add Build Step > Invoke Gradle Script.
2. Select the Gradle Version configured earlier.
3. In the Tasks field, enter:

clean build

4. Click Save.

Step 6: Configure Post-Build Actions

1. Scroll to Post-Build Actions.
2. Click Add Post-Build Action > Publish JUnit Test Result Report.
3. In the Test Report XMLs field, enter:
4. target/surefire-reports/*.xml (for Maven)

   build/test-results/test/*.xml (for Gradle)

5. Click Save.

Step 7: Trigger Automatic Builds

To trigger builds automatically upon code changes:

1. In the Build Triggers section, check Poll SCM.

2. Enter a schedule (e.g., H/5 * * * * to check for changes every 5 minutes).
3. Click Save.

Step 8: Run and Verify the Build

1. Navigate to the Jenkins dashboard and select the newly created job.
2. Click Build Now.
3. Monitor the build status in the Build History section.
4. Click on the latest build and navigate to Console Output to review logs.
5. Verify that the build completes successfully and test results are generated.

7. Configuration Management with Ansible: Basics of Ansible: Inventory, Playbooks, and Modules, Automating Server Configurations with Playbooks, Hands-On: Writing and Running a Basic Playbook.

Objective:

To understand the basics of Ansible — Inventory, Playbooks, and Modules — and use Playbooks to automate server configurations. This includes setting up a Linux environment using Vagrant and VirtualBox, installed via Chocolatey.

What is Ansible?

Ansible is an open-source configuration management and automation tool. It helps system administrators and DevOps engineers to automate:

- Software installation and configuration
- Server provisioning
- Application deployment
- System updates and patching

Key Features of Ansible:

- Agentless: No agent is needed on the target machine. It uses SSH to communicate.
- Simple syntax: Uses YAML to define tasks in Playbooks.
- Idempotent: Running the same playbook multiple times won't cause unintended changes.
- Modular: Uses built-in modules like apt, yum, copy, service, etc.

Ansible Components:

Component	Description
Inventory	List of target machines to automate
Modules	Reusable scripts used to perform actions
Playbook	YAML file that contains tasks to be executed
Task	A single unit of work to be executed on a host

💻 Part 1: Setting Up the Environment on Windows Using Chocolatey

🍫 Step 1: Install Chocolatey

Open PowerShell as Administrator and run:

```
Set-ExecutionPolicy Bypass -Scope Process -Force; `
[System.Net.ServicePointManager]::SecurityProtocol =
[System.Net.ServicePointManager]::SecurityProtocol -bor 3072; `
```

```
iex ((New-Object
System.Net.WebClient).DownloadString('https://community.chocolate
y.org/install.ps1'))
```

Close and reopen PowerShell after installation.

Step 2: Install VirtualBox and Vagrant via Chocolatey

```
choco install virtualbox -y

choco install vagrant -y
```

Part 2: Create and Configure a Linux VM using Vagrant

Step 3: Create a Project Directory

```
mkdir ansible-vm

cd ansible-vm
```

Step 4: Initialize a Vagrant Project

```
vagrant init ubuntu/bionic64
```

This creates a Vagrantfile.

Step 5: Edit the Vagrantfile

Update the file to look like:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/bionic64"
  config.vm.network "forwarded_port", guest: 80, host: 8080
  config.vm.provider "virtualbox" do |vb|
    vb.memory = "2048"
    vb.cpus = 2
  end
end
```

Step 6: Start the Virtual Machine

```
vagrant up
```

Step 7: SSH into the VM

```
vagrant ssh
```

Part 3: Install Ansible inside the Linux VM

Once inside the VM:

```
sudo apt update
sudo apt install ansible -y
```

Verify:

```
ansible --version
```

Part 4: Write and Run Your First Ansible Playbook

Step 8: Create Inventory File

Inside the VM, create inventory.ini:

```
[web]
127.0.0.1 ansible_connection=local
```

Step 9: Create a Playbook

Create setup_webserver.yml:

```
---
- name: Setup Apache Web Server
  hosts: web
  become: yes
  tasks:
    - name: Install Apache
      apt:
        name: apache2
        state: present
        update_cache: yes

    - name: Enable and start Apache
```

```
      service:
        name: apache2
        state: started
        enabled: yes

    - name: Deploy custom HTML page
      copy:
        content: "<h1>Welcome to Ansible Web Server!</h1>"
        dest: /var/www/html/index.html
```

____

Step 10: Run the Playbook

```
ansible-playbook -i inventory.ini setup_webserver.yml
```

____

Part 5: Verify the Web Server

✅ Check Web Server from inside VM:

```
curl http://localhost
```

Expected output:

```
<h1>Welcome to Ansible Web Server!</h1>
```

____

Conclusion:

In this experiment, you have:

- Installed VirtualBox and Vagrant using Chocolatey
- Created a Linux VM using Vagrant
- Installed Ansible and learned its core components
- Wrote and executed a basic Ansible Playbook
- Deployed a simple web server automatically

8. Practical Exercise: Set Up a Jenkins CI Pipeline for a Maven Project, Use Ansible to Deploy Artifacts Generated by Jenkins.

Part 1: Install Jenkins and Maven on Ubuntu (VirtualBox)

Step 1: Update System

sudo apt update

sudo apt upgrade -y

Step 2: Install Java (required for Jenkins & Maven)

sudo apt install openjdk-11-jdk -y

java -version

Step 3: Install Maven

sudo apt install maven -y

mvn -version

Step 4: Install Jenkins

Add Jenkins Repository

wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add -

sudo sh -c 'echo deb https://pkg.jenkins.io/debian binary/ > /etc/apt/sources.list.d/jenkins.list'

Install Jenkins

sudo apt update

sudo apt install jenkins -y

Start and Enable Jenkins

bsudo systemctl start jenkins

sudo systemctl enable jenkins

Access Jenkins

Open browser inside Ubuntu and go to:

http://localhost:8080

Unlock Jenkins

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

Use this password to unlock Jenkins and install **Suggested Plugins.**

Part 2: Create a Simple "Hello World" Java Maven Project

Step 1: Create Directory and Project

```
mkdir hello-maven

cd hello-maven

mvn archetype:generate -DgroupId=com.example -DartifactId=hello-maven -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Step 2: Navigate to the Project

```
cd hello-maven
```

Step 3:

Edit the App.java

```
// File: src/main/java/com/example/App.java

package com.example;


public class App {

   public static void main(String[] args) {

      System.out.println("Hello, Jenkins CI!");

   }

}
```

Edit pom.xml

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

         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

                           http://maven.apache.org/xsd/maven-4.0.0.xsd">


   <modelVersion>4.0.0</modelVersion>


   <groupId>com.example</groupId>

   <artifactId>hello-maven</artifactId>

   <version>1.0-SNAPSHOT</version>
```

```
<properties>
  <maven.compiler.source>11</maven.compiler.source>
  <maven.compiler.target>11</maven.compiler.target>
</properties>

<dependencies>
  <!-- JUnit 4 for testing -->
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <!-- Compiler plugin -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.8.1</version>
      <configuration>
        <encoding>UTF-8</encoding>
        <source>11</source>
        <target>11</target>
      </configuration>
    </plugin>
    </plugins>

  </build>


</project>
```

Edit AppTest.java

```
// File: src/test/java/com/example/AppTest.java
            package com.example;

            import org.junit.Test;
            import static org.junit.Assert.*;

            public class AppTest {
            @Test
            public void testApp() {
            assertTrue(true);
            }
            }
```

Step 4: Build and Test

```
mvn clean install
```

Output will be in:

target/hello-maven-1.0-SNAPSHOT.jar

Part 3: Set Up Jenkins CI Pipeline for the Maven Project

Step 1: Open Jenkins → Create New Job

- Choose Freestyle project
- Name: hello-maven-ci

Step 2: Configure Project

- Source Code Management: None (since local project)
- Build:
  - Add a build step: Invoke top-level Maven targets
  - Goals: clean install
  - POM: path-to-your-project/pom.xml (e.g., workspace/hello-maven/hello-maven/pom.xml)

Note: Keep All Jenkins Projects Inside /var/lib/jenkins/workspace/

Ex:

```
sudo cp -r /home/anil/Desktop/hello-maven /var/lib/jenkins/workspace/
sudo chown -R jenkins:jenkins /var/lib/jenkins/workspace/hello-maven
```

Step 3: Save and Build

Click **Build Now** – Jenkins will compile and build the JAR locally.

____

Part 4: Deploy Artifact Using Ansible

Step 1: Install Ansible

```
sudo apt install ansible -y
```

Step 2: Prepare Ansible Playbook

Create a directory:

```
mkdir ansible-deploy
cd ansible-deploy
```

Create deploy.yml:

```
- name: Deploy Maven Artifact Locally
  hosts: localhost
  tasks:
    - name: Copy JAR to /opt/app
      copy:
        src: /home/user/hello-maven/target/hello-maven-1.0-SNAPSHOT.jar
        dest: /opt/app/hello-maven.jar
```

Note: Change the src path accordings

Step 3: Run the Playbook

```
sudo mkdir -p /opt/app
ansible-playbook deploy.yml
```

9. Introduction to Azure DevOps: Overview of Azure DevOps Services, Setting Up an Azure DevOps Account and Project

Introduction to Azure DevOps

What is Azure DevOps?

Azure DevOps is a cloud-based service from Microsoft that provides developer services to support teams in planning work, collaborating on code development, and building & deploying applications.

Azure DevOps Services – Overview

Azure DevOps offers the following main services:

1. Azure Repos
   - Provides Git repositories or Team Foundation Version Control (TFVC).
   - Used for source code version control.
   - Supports pull requests, branching, and code reviews.
2. Azure Pipelines
   - Used for Continuous Integration (CI) and Continuous Delivery (CD).
   - Supports building, testing, and deploying code automatically to any platform.
3. Azure Boards
   - Agile planning tools: Kanban boards, backlogs, and dashboards.
   - Helps manage tasks, bugs, and user stories.
4. Azure Test Plans
   - Provides manual and exploratory testing tools.
   - Helps ensure software quality and coverage.
5. Azure Artifacts
   - Manages package dependencies (e.g., NuGet, npm, Maven).
   - Allows teams to create, host, and share packages.

Setting Up an Azure DevOps Account and Project

Step 1: Create a Microsoft Account (if not already available)

- Visit: https://signup.live.com
- Use this account to sign in to Azure DevOps.

Step 2: Sign in to Azure DevOps

- Go to: https://dev.azure.com
- Use your Microsoft credentials to log in.

Step 3: Create a New Organization

- An organization in Azure DevOps groups related projects.

- Click on "New Organization", provide a name, select a region.

Step 4: Create a Project

- Inside your organization, click "New Project".
- Fill in:
  - Project name
  - Visibility: Public or Private
  - Choose Version Control: Git or TFVC
  - Select Work Item Process: Agile, Scrum, CMMI, Basic

Step 5: Explore Services

- After project creation, you can start using:
  - Repos for code
  - Boards for task tracking
  - Pipelines for CI/CD
  - Artifacts for packages
  - Test Plans for testing

10. Creating Build Pipelines: Building a Maven/Gradle Project with Azure Pipelines, Integrating Code Repositories (e.g., GitHub, Azure Repos), Running Unit Tests and Generating Reports

Objective:

To configure a self-hosted agent on a local machine, build a Maven or Gradle project using Azure DevOps Pipelines, run unit tests, and generate reports — all without incurring any Azure charges.

Theory Overview

What is a Self-Hosted Agent?

A Self-hosted agent is a personal or institutional computer (e.g., your own laptop or lab PC) that runs Azure Pipelines jobs instead of using Microsoft-hosted (cloud) agents.

- No billing or time limits
- Full control over environment
- Great for offline or classroom scenarios

Steps to Build a Maven/Gradle Project with a Self-Hosted Agent

---

Step 1: Prepare Your System (Self-Hosted Agent)

You'll need:

- A PC with Java, Maven or Gradle installed
- Git installed
- Internet access to download Azure agent

Step 2: Create an Azure DevOps Project (One-Time Setup by Instructor)

1. Go to https://dev.azure.com
2. Sign in and create a new organization (e.g., college-devops-org)
3. Create a project (e.g., BuildPipelineLab)

Step 3: Register Your Self-Hosted Agent

1. In Azure DevOps portal:
   - Go to Project Settings > Agent Pools
   - Click Default pool
   - Select New Agent > Choose OS (Windows/Linux/macOS)

2. Follow the steps shown to:

   Download agent package

   Extract it to a folder, e.g., C:\azagent

   Run these commands:

```
cd C:\azagent

config.cmd
```

3. Provide:
   - DevOps URL
   - Personal Access Token (PAT)
   - Agent name (e.g., labagent1)
   - Choose "run as a service" if prompted
4. Start the agent:

```
run.cmd
```

Step 4: Push Code to Repository

1. Create a repository in Azure Repos or GitHub
2. Push your Maven/Gradle project there

Step 5: Create azure-pipelines.yml File in Your Repo

For Maven:

```
trigger:

- main


pool:

  name: Default  # Refers to your self-hosted agent pool


steps:

- task: Maven@3

  inputs:

    mavenPomFile: 'pom.xml'

    goals: 'clean install'
    publishJUnitResults: true
```

For Gradle:

```
trigger:
- main

pool:
  name: Default

steps:
- script: ./gradlew clean build
  displayName: 'Build with Gradle'
```

Step 6: Run the Pipeline

1. Commit the YAML file to your repository.
2. Azure Pipelines will trigger automatically when changes are pushed.
3. The job will run on your own PC or lab machine (self-hosted agent).

Step 7: View Unit Test Reports

- Go to Pipelines > Runs
- Click the latest run
- Go to Tests tab to view results

If you don't see test results:

- Make sure you're using a testing framework like JUnit/TestNG.
- Ensure reports are being generated in standard format (e.g., target/surefire-reports for Maven).

Advantages of Using Self-Hosted Agent

- Zero cost – no billing risk
- Use lab computers or personal laptops
- More control over software environment
- Ideal for educational institutions

11. Creating Release Pipelines: Deploying Applications to Azure App Services, Managing Secrets and Configuration with Azure Key Vault, Hands-On: Continuous Deployment with Azure Pipelines

Objective

To understand how to deploy applications to Azure App Services using Azure Pipelines, manage secrets using Azure Key Vault, and implement continuous deployment.

---

1. Deploying Applications to Azure App Services

Azure App Services Overview

Azure App Services is a fully managed platform for building, deploying, and scaling web apps.

Steps to Deploy using Azure Pipelines

1. Create an App Service in Azure:
   - Go to Azure Portal → Create a Resource → Web App.
   - Provide details like App name, Runtime stack, Region, and Resource group.
2. Create a Release Pipeline in Azure DevOps:
   - Go to Azure DevOps → Pipelines → Releases.
   - Click on New Pipeline → Select Azure App Service Deployment template.
3. Configure the Artifact:
   - Link the build artifact from your build pipeline (Maven/Gradle).
   - Choose the appropriate version and source.
4. Add a Stage and Deploy:
   - Select the App Service target.
   - Configure App Service connection and app name.
5. Trigger Deployment:
   - Enable continuous deployment trigger.
   - Save and create a release.

---

2. Managing Secrets and Configuration with Azure Key Vault

What is Azure Key Vault?

Azure Key Vault is a cloud service to securely store and access secrets, keys, and certificates.

Using Key Vault in Azure Pipelines

1. **Create a Key Vault:**
   - Go to Azure Portal → Create Key Vault.
   - Store secrets like DB passwords, API keys.
2. **Configure Access:**
   - Assign appropriate access policies (for Azure Pipelines or App Service).

3. **Link Key Vault in Azure Pipeline:**
   - In the pipeline YAML or classic editor:
   - variables:
   - - group: MyKeyVaultSecrets
   - Use secrets like $(MySecret) in scripts or tasks.

4. **Integrate with App Service Configuration**:

   - App Services can directly pull secrets using Managed Identity.

---

3. Hands-On: Continuous Deployment with Azure Pipelines

What is Continuous Deployment (CD)?

Continuous Deployment is the practice of automatically deploying every code change that passes the CI pipeline.

Setup CD with Azure Pipelines:

1. Ensure CI Pipeline is Configured:
   - Generates build artifact after every commit.
2. Create a Release Pipeline:
   - Link the artifact from CI.
   - Add deployment stages (Dev, QA, Prod).
3. Enable Continuous Deployment Trigger:
   - In Artifact section → enable CD trigger.
4. Deploy to Azure App Service:
   - Use the Azure App Service Deploy task.
   - Set environment-specific settings.
5. Monitor and Validate:
   - Check deployment status.
   - Validate on the App Service URL.

Expected Outcome

- You will be able to:

  - Automatically deploy your Java web app to Azure App Services.
  - Use Azure Key Vault to manage sensitive data securely.
  - Configure full CI/CD with Azure Pipelines.

12. Practical Exercise and Wrap-Up: Build and Deploy a Complete DevOps Pipeline, Discussion on Best Practices and Q&A

Objective:

To integrate the knowledge gained from previous experiments by designing and executing a complete DevOps pipeline for a Java-based application. This includes stages such as version control, build automation, testing, packaging, deployment, and optional containerization and monitoring.

Part A: Practical Hands-On Exercise

1. Project Setup

- Choose a Java Application: You can use a sample application (e.g., a Spring Boot app or a basic Java servlet).
- Version Control:
  - Create a new GitHub repository.
  - Push the project code to the repository.
  - Ensure .gitignore excludes target/, .idea/, and other unnecessary files.

---

2. Build Automation with Maven or Gradle

- Create or validate your pom.xml (Maven) or build.gradle (Gradle) file.
- Include:
  - Project dependencies
  - Build plugins
  - Test configuration

Example (for Maven):

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.8.1</version>
      <configuration>
        <source>11</source>
        <target>11</target>
```

```
            </configuration>
          </plugin>
        </plugins>
      </build>
```

---

3. Configure Jenkins

- Install Jenkins (local/cloud – AWS EC2 preferred)
- Install Plugins:
  - Git Plugin
  - Maven Integration Plugin
  - Docker (if using containerization)
- Create a Jenkins Job (Freestyle or Pipeline):
  - Source Code Management: GitHub Repo
  - Build Trigger: "Poll SCM" or Webhook from GitHub
  - Build Steps:

    mvn clean install

    Optional: Run test cases, create artifacts

---

4. Unit Testing

Write unit test cases using JUnit or TestNG.

Ensure Jenkins is configured to fail builds on test failures.

Sample Test (JUnit):

```
@Test
public void testAdd() {
        assertEquals(5, Calculator.add(2, 3));
}
```

---

5. Artifact Management

- Local Option: Save the .jar or .war file in Jenkins workspace or shared location.
- Advanced Option: Use Nexus/Artifactory to store the generated build artifacts.

---

6. Containerization (Optional but Encouraged)

- Create a Dockerfile:
- FROM openjdk:11
- COPY target/myapp.jar myapp.jar
- ENTRYPOINT ["java", "-jar", "myapp.jar"]
- Build Docker Image:
- docker build -t yourusername/myapp:v1 .
- Push to Docker Hub:
- docker push yourusername/myapp:v1

---

7. Deployment

- Local Deployment:
  - Run the .jar or use Docker: docker run -p 8080:8080 yourusername/myapp:v1
- Cloud Deployment:
  - SCP the artifact to AWS EC2 or use Jenkins to automate deployment.
  - Start the application on the cloud server.

---

8. Monitoring and Logging (Optional/Advanced)

- Introduce basic logging using Log4j or SLF4J.
- If time permits:
  - Set up Prometheus for metrics collection.
  - Use Grafana for visualization.
  - Configure alerts for failed builds or crashed containers.

Part B: Best Practices Discussion

Topic	**Description**
CI/CD	Importance of continuous integration & delivery for faster feedback loops.
Fast & Reliable Builds	Keep builds short and optimized. Use build caching and parallel test execution.
Security	Never hardcode credentials. Use environment variables or tools like HashiCorp Vault.
Infrastructure as Code	Use tools like Terraform or Ansible for consistent, automated infra setup.
Branching Strategies	GitFlow or trunk-based development to manage feature and release branches.

Deployment Strategies	Use Blue-Green or Canary deployment to reduce risk.
Container Orchestration	Learn basics of Kubernetes for managing container-based apps.

Part C: Q&A Session

Encourage open interaction on:

- Common Errors:
  - Jenkins build failures
  - Docker image issues
  - Maven dependency problems
- Clarification on Concepts:
  - Difference between CI and CD
  - Use of webhooks
  - Real-world Jenkins pipelines
- Career & Industry Insights:
  - DevOps engineer roles
  - Certification paths (AWS, Docker, Jenkins)
  - Popular tools in the market