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:

1. Demonstrate different actions performed through Version control tools like Git.
2. Perform Continuous Integration and Continuous Testing and Continuous Deployment using Jenkins by building and automating test cases using Maven & Gradle.
3. Experiment with configuration management using Ansible.
4. 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 schedule 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

List of Experiments	DATE	REMARKS
Cycle I		
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,		
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		

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

Introduction to Maven and Gradle

Overview of Build Automation Tools

Build automation tools help developers streamline the process of building, testing, and deploying software projects. They take care of repetitive tasks like compiling code, managing dependencies, and packaging applications, which makes development more efficient and error-free.

Two popular tools in the Java ecosystem are **Maven** and **Gradle**. Both are great for managing project builds and dependencies, but they have some key differences.

Maven

- **What is Maven?** Maven is a build automation tool primarily used for Java projects. It uses an XML configuration file called `pom.xml` (Project Object Model) to define project settings, dependencies, and build steps.
- **Main Features:**
  - Predefined project structure and lifecycle phases.
  - Automatic dependency management through Maven Central.
  - Wide range of plugins for things like testing and deployment.
  - Supports complex projects with multiple modules.

Gradle

- **What is Gradle?** Gradle is a more modern and versatile build tool that supports multiple programming languages, including Java, Groovy, and Kotlin. It uses a domain-specific language (DSL) for build scripts, written in Groovy or Kotlin.
- **Main Features:**
  - Faster builds thanks to task caching and incremental builds.
  - Flexible and customizable build scripts.
  - Works with Maven repositories for dependency management.
  - Excellent support for multi-module and cross-language projects.
  - Integrates easily with CI/CD pipelines.

Key Differences Between Maven and Gradle

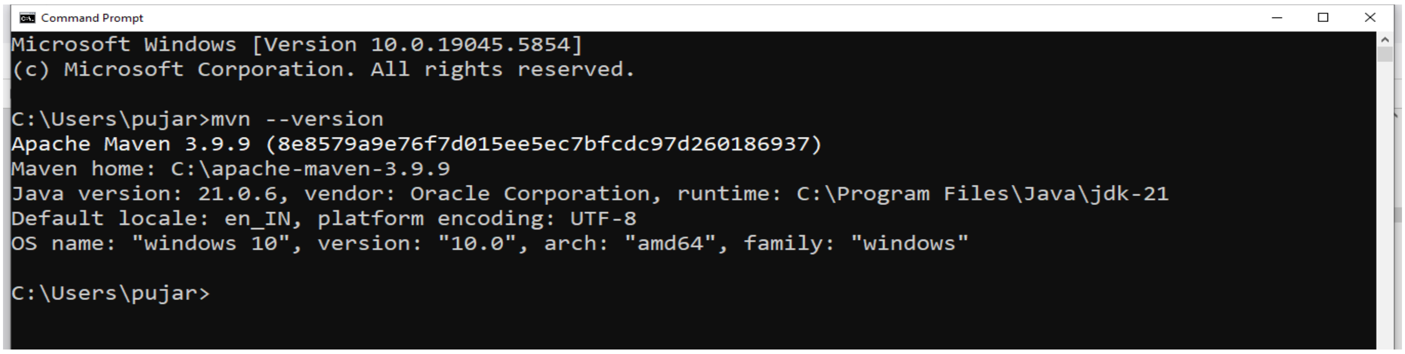
Aspect	Maven	Gradle
Configuration	XML (`pom.xml`)	Groovy or Kotlin DSL
Performance	Slower	Faster due to caching
Flexibility	Less flexible	Highly customizable
Learning Curve	Easier to pick up	Slightly steeper
Script Size	Verbose	More concise
Dependency Management	Uses Maven Central	Compatible with Maven too
Plugin Support	Large ecosystem	Extensible and versatile

Installation and Setup

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.



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.

```
Command Prompt
Microsoft Windows [Version 10.0.19045.5854]
(c) Microsoft Corporation. All rights reserved.

C:\Users\pujar>gradle --version

------------------------------------------------------------
Gradle 8.13
------------------------------------------------------------

Build time:    2025-02-25 09:22:14 UTC
Revision:      073314332697ba45c16c0a0ce1891fa6794179ff

Kotlin:        2.0.21
Groovy:        3.0.22
Ant:           Apache Ant(TM) version 1.10.15 compiled on August 25 2024
Launcher JVM:  21.0.6 (Oracle Corporation 21.0.6+8-LTS-188)
Daemon JVM:    C:\Program Files\Java\jdk-21 (no JDK specified, using current Java home)
OS:            Windows 10 10.0 amd64
```

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

1: Install the Java JDK

- If you haven't installed the **Java JDK** yet, you can follow the link below to download and install it. [Download Java JDK from Oracle]

Working with Maven is a key skill for managing Java-based projects, particularly in the areas of build automation, dependency management, and project configuration. Below is a guide on creating a Maven project, understanding the POM file, and using dependency management and plugins:

Overview of the Project

2: Creating a Maven Project

There are a few ways to create a Maven project, such as using the command line, IDEs like IntelliJ IDEA or Eclipse, or generating it via an archetype.

1. **Using Command Line**:
    - To create a basic Maven project using the command line, you can use the following command:

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

- **`groupId`:** A unique identifier for the group (usually the domain name).
- **`artifactId`:** A unique name for the project artifact (your project).
- **`archetypeArtifactId`:** The template you want to use for the project.
- **`DinteractiveMode=false`:** Disables prompts during project generation.

This will create a basic Maven project with the required directory structure and **`pom.xml`** file.

2. **Using IDEs**

Most modern IDEs (like IntelliJ IDEA or Eclipse) provide wizards to generate Maven projects. For example, in IntelliJ IDEA:

1. Go to **File** > **New Project**.
2. Choose **Maven** from the list of project types.
3. Provide the **`groupId`** and **`artifactId`** for your project.

3: Understanding the POM File

The **POM (Project Object Model)** file is the heart of a Maven project. It is an XML file that contains all the configuration details about the project. Below is an example of a simple POM file:

A basic **`pom.xml`** structure looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<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>my-project</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <dependencies>
        <!-- Dependencies go here -->
    </dependencies>

    <build>
```

```
        <plugins>
            <!-- Plugins go here -->
        </plugins>
    </build>
</project>
```

Key element in `pom.xml`:

- **`<groupId>`:** The group or organization that the project belongs to.
- **`<artifactId>`:** The name of the project or artifact.
- **`<version>`:** The version of the project (often follows a format like `1.0-SNAPSHOT`).
- **`<packaging>`:** Type of artifact, e.g., `jar`, `war`, `pom`, etc.
- **`<dependencies>`:** A list of dependencies the project requires.
- **`<build>`:** Specifies the build settings, such as plugins to use.

4: Dependency Management

Maven uses the `<dependencies>` tag in the `pom.xml` to manage external libraries or dependencies that your project needs. When Maven builds the project, it will automatically download these dependencies from a repository (like Maven Central).

Example of adding a dependency:

```
<dependencies>
    <dependency>
        <groupId>org.apache.commons</groupId>
        <artifactId>commons-lang3</artifactId>
        <version>3.12.0</version>
    </dependency>
</dependencies>
```

- **Transitive Dependencies**
  - Maven automatically resolves transitive dependencies. For example, if you add a library that depends on other libraries, Maven will also download those.
- **Scopes**
  - Dependencies can have different scopes that determine when they are available:
    - **compile** (default): Available in all build phases.
    - **provided**: Available during compilation but not at runtime (e.g., a web server container).
    - **runtime**: Needed only at runtime, not during compilation.
    - **test**: Required only for testing.

5: Using Plugins

Maven plugins are used to perform tasks during the build lifecycle, such as compiling code, running tests, packaging, and deploying. You can specify plugins within the `<build>` section of your **`pom.xml`**.

- **Adding Plugins**
  - You can add a plugin to your `pom.xml` like so:

```
<build>
    <plugins>
        <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>
```

In this example, the **`maven-compiler-plugin`** is used to compile Java code and specify the source and target JDK versions.

1. **Common Plugins**
    - **maven-compiler-plugin**: Compiles Java code.
    - **maven-surefire-plugin**: Runs unit tests.
    - **maven-jar-plugin**: Packages the project as a JAR file.
    - **maven-clean-plugin**: Cleans up the `target/` directory.
2. **Plugin Goals** Each plugin consists of goals, which are specific tasks to be executed. For example:
    - **`mvn clean install`:** This will clean the target directory and then install the package in the local repository.
    - **`mvn compile`:** This will compile the source code.
    - **`mvn test`:** This will run unit tests.

6: Dependency Versions and Repositories

1. **Version Ranges**
    - You can specify a version range for dependencies, allowing Maven to choose a compatible version automatically. for example:

```
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>[30.0,)</version> <!-- Guava version 30.0 or higher -->
</dependency>
```

2. **Repositories**
    - Maven primarily fetches dependencies from Maven Central, but you can also specify custom repositories. For example:

```
<repositories>
    <repository>
        <id>custom-repo</id>
        <url>https://repo.example.com/maven2</url>
    </repository>
</repositories>
```

Working with Maven Project

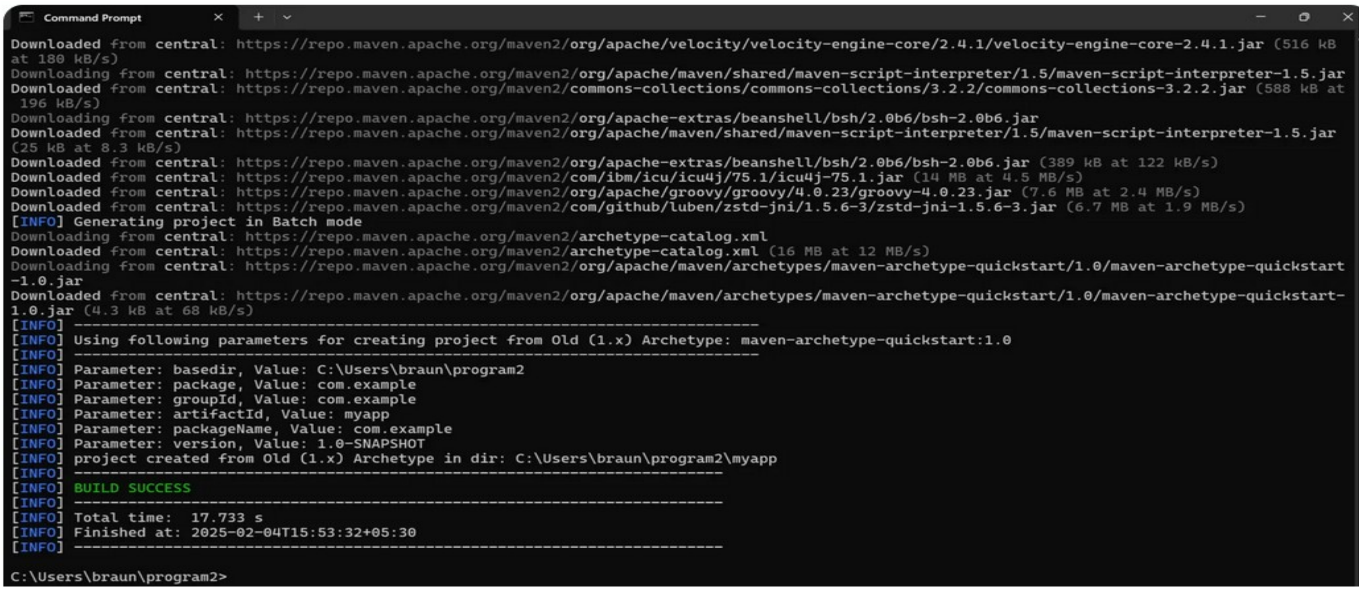
Note: Always create separate folder to do any program.

- Open command prompt.
- **`mkdir program2`** – this will create **program2** folder.
- **`cd program2`** – navigate program2 folder.
- After then follow the below step to working with Maven project.

Step 1: Creating a Maven Project

- You can create a **Maven project** using the **`mvn`** command (or through your **IDE**, as mentioned earlier). But here, I'll give you the essential **`pom.xml`** and **Java code**.
- Let's use the **Apache Commons Lang library** as a **dependency** (which provides utilities for **working with strings**, **numbers**, etc.). We will use this in a **simple Java program** to **work with strings**.

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



Note: See in your terminal below the project folder path showing after executing the cmd manually navigate the path and see the project folder name called **myapp**.

Step 2: Open The `pom.xml` File

- You can manually navigate the **project folder** named call **myapp** and open the file pom.xml and copy the below code and paste it then save it.
- In case if you not getting project folder then type command in your cmd.
  - **cd myapp –** is use to navigate the project folder.
  - **notepad pom.xml** – is use to open pom file in notepad.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<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>
        <!-- JUnit Dependency for Testing -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <!-- Maven Surefire Plugin for running tests -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.22.2</version>
                <configuration>
                    <redirectTestOutputToFile>false</redirectTestOutputToFile>
                    <useSystemOut>true</useSystemOut>
```

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

Step 3: Open Java Code (`App.java`) File

- Open a file **`App.java`** inside the **`src/main/java/com/example/`** directory.
- After opening the **App.java** copy the below code and paste it in that file then save it.

```java
package com.example;

public class App {

    public int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        App app = new App();

        int result = app.add(2, 3);
        System.out.println("2 + 3 = " + result);

        System.out.println("Application executed successfully!");
    }
}
```

Step 4: Open Java Code (`AppTest.java`) File

- Open a file **`AppTest.java`** inside the **`src/test/java/com/example/`** directory.
- After opening the **AppTest.java** copy the below code and paste it in that file then save it.

```java
package com.example;

import org.junit.Assert;
import org.junit.Test;

public class AppTest {

    @Test
    public void testAdd() {
        App app = new App();
        int result = app.add(2, 3);

        System.out.println("Running test: 2 + 3 = " + result);

        Assert.assertEquals(5, result);
    }
}
```

Note: before building the project make sure you are in the project folder if not navigate the project folder type command in your command prompt **`cd myapp`**

Step 4: Building the Project

To build and run this project, follow these steps:

1. **Compile the Project**

```
mvn compile
```

```
Command Prompt
ompiler-javac-2.15.0.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/ow2/asm/asm/9.6/asm-9.6.jar (124 kB at 1.2 MB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/4.0.0/plexus-utils-4.0.0
.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-java/1.2.0/plexus-java-1.2.0.ja
r (58 kB at 523 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/com/thoughtworks/qdox/qdox/2.0.3/qdox-2.0.3.jar (334 kB at
 2.7 MB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-compiler-javac/2.15.0/plexus-co
mpiler-javac-2.15.0.jar (26 kB at 196 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-xml/3.0.0/plexus-xml-3.0.0.jar
(93 kB at 660 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/4.0.0/plexus-utils-4.0.0.
jar (192 kB at 899 kB/s)
[INFO] Recompiling the module because of changed source code.
[WARNING] File encoding has not been set, using platform encoding UTF-8, i.e. build is platform dependent!
[INFO] Compiling 1 source file with javac [debug target 1.8] to target\classes
[WARNING] bootstrap class path is not set in conjunction with -source 8
  not setting the bootstrap class path may lead to class files that cannot run on JDK 8
    --release 8 is recommended instead of -source 8 -target 1.8 because it sets the bootstrap class path automatically
[WARNING] source value 8 is obsolete and will be removed in a future release
[WARNING] target value 8 is obsolete and will be removed in a future release
[WARNING] To suppress warnings about obsolete options, use -Xlint:-options.
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  6.119 s
[INFO] Finished at: 2025-02-04T17:01:55+05:30
[INFO] ------------------------------------------------------------------------
```

2. Run the Unit Tests

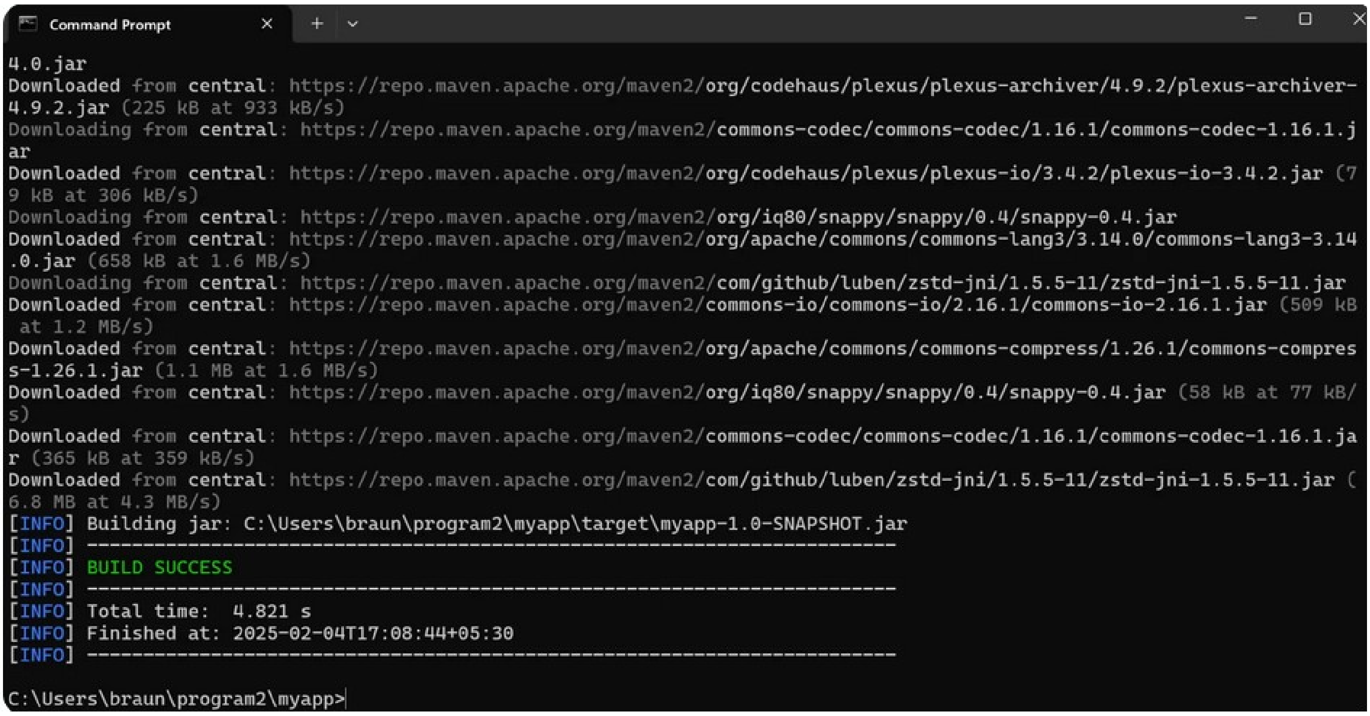
```
mvn test
```

```
[WARNING] File encoding has not been set, using platform encoding UTF-8, i.e. build is platform dependent!
[INFO] Compiling 1 source file with javac [debug target 1.8] to target\test-classes
[WARNING] bootstrap class path is not set in conjunction with -source 8
  not setting the bootstrap class path may lead to class files that cannot run on JDK 8
    --release 8 is recommended instead of -source 8 -target 1.8 because it sets the bootstrap class path automatically
[WARNING] source value 8 is obsolete and will be removed in a future release
[WARNING] target value 8 is obsolete and will be removed in a future release
[WARNING] To suppress warnings about obsolete options, use -Xlint:-options.
[INFO]
[INFO] --- surefire:2.22.2:test (default-test) @ myapp ---
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running com.example.AppTest
Running test: 2 + 3 = 5
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.053 s - in com.example.AppTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  2.680 s
[INFO] Finished at: 2025-02-04T21:09:48+05:30
[INFO] ------------------------------------------------------------------------

C:\Users\braun\program2\myapp>
```

3. Package the project into a JAR

```
mvn package
```

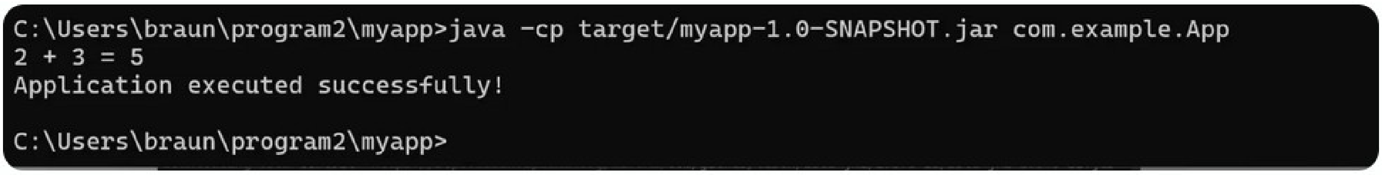


Run the application (using JAR)

```
java -cp target/myapp-1.0-SNAPSHOT.jar com.example.App
```

The above command is used to **run a Java application** from the command line. Here's a breakdown of each part:

- **`java`**: This is the Java runtime command used to run Java applications.
- **`-cp`**: This stands for **classpath**, and it specifies the location of the classes and resources that the JVM needs to run the application. In this case, it's pointing to the JAR file where your compiled classes are stored.
- **`target/myapp-1.0-SNAPSHOT.jar`**: This is the **JAR file** (Java ARchive) that contains the compiled Java classes and resources. It's located in the `target` directory, which Maven creates after you run `mvn package`.
- **`com.example.App`**: This is the **main class** that contains the `main()` method. When you run this command, Java looks for



the `main()` method inside the `App` class located in the `com.example` package and executes it.

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

Gradle Project Overview

1: Setting Up a Gradle Project

- **Install Gradle** (If you haven't already):
  - Follow Gradle installation Program 1 **click here**
- **Create a new Gradle project**: You can set up a new Gradle project using the Gradle Wrapper or manually. Using the Gradle Wrapper is the preferred approach as it ensures your project will use the correct version of Gradle.
- **To create a new Gradle project using the command line:**

```
gradle init --type java-application
```

This command creates a new Java application project with a sample **`build.gradle`** file.

2: Understanding Build Scripts

Gradle uses a DSL (Domain-Specific Language) to define the build scripts. Gradle supports two DSLs:

- **Groovy DSL** (default)
- **Kotlin DSL** (alternative)

Groovy DSL: This is the default language used for Gradle build scripts (**`build.gradle`**). Example of a simple **`build.gradle`** file (Groovy DSL):

```
plugins {
    id 'java'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web:2.5.4'
}

task customTask {
    doLast {
        println 'This is a custom task'
    }
}
```

Kotlin DSL: Gradle also supports Kotlin for its build scripts (**`build.gradle.kts`**). Example of a simple **`build.gradle.kts`** file (Kotlin DSL):

```
plugins {
    kotlin("jvm") version "1.5.21"
}

repositories {
    mavenCentral()
}

dependencies {
    implementation("org.springframework.boot:spring-boot-starter-web:2.5.4")
}

tasks.register("customTask") {
    doLast {
```

```
        println("This is a custom task")
    }
}
```

Difference between Groovy and Kotlin DSL:

- **Syntax**: Groovy uses a more concise, dynamic syntax, while Kotlin offers a more structured, statically-typed approach.
- **Error handling**: Kotlin provides better error detection at compile time due to its static nature.

Task Block: Tasks define operations in Gradle, and they can be executed from the command line using `gradle <task-name>`.

In Groovy DSL:

```
task hello {
    doLast {
        println 'Hello, Gradle!'
    }
}
```

In Kotlin DSL:

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

3: Dependency Management

Gradle provides a powerful dependency management system. You define your project's dependencies in the `dependencies` block.

1. **Adding dependencies**:
    - Gradle supports various dependency scopes such as `implementation`, `compileOnly`, `testImplementation`, and others.

Example of adding a dependency in **`build.gradle`** (**Groovy DSL**):

```
dependencies {
    implementation 'com.google.guava:guava:30.1-jre'
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.7.1'
}
```

Example in **`build.gradle.kts`** (**Kotlin DSL**):

```
dependencies {
    implementation("com.google.guava:guava:30.1-jre")
    testImplementation("org.junit.jupiter:junit-jupiter-api:5.7.1")
}
```

2. **Declaring repositories**: To resolve dependencies, you need to specify repositories where Gradle should look for them. Typically, you'll use Maven Central or JCenter, but you can also configure private repositories.

Example (Groovy):

```
repositories {
    mavenCentral()
}
```

Example (Kotlin):

```
repositories {
    mavenCentral()
}
```

4: Task Automation

Gradle tasks automate various tasks in your project lifecycle, like compiling code, running tests, and creating builds.

1. **Using predefined tasks**: Gradle provides many predefined tasks for common activities, such as:
    - **`build`** – compiles the project, runs tests, and creates the build output.
    - **`test`** – runs tests.
    - **`clean`** – deletes the build output.
2. Example of running the `build` task:

```
gradle build
```

3. **Creating custom tasks**: You can define your own tasks to automate specific actions. For example, creating a custom task to print a message.

- **Example Groovy DSL:**

```
task printMessage {
    doLast {
        println 'This is a custom task automation'
    }
}
```

- **Example Kotlin DSL:**

```
tasks.register("printMessage") {
    doLast {
        println("This is a custom task automation")
    }
}
```

5: Running Gradle Tasks

To run a task, use the following command in the terminal:

```
gradle <task-name>
```

For example:

- To run the `build` task: **`gradle build`**
- To run a custom task: **`gradle printMessage`**

6: Advanced Automation

You can define task dependencies and configure tasks to run in a specific order. Example of task dependency:

```
task firstTask {
    doLast {
        println 'Running the first task'
    }
}

task secondTask {
    dependsOn firstTask
    doLast {
        println 'Running the second task'
    }
}
```

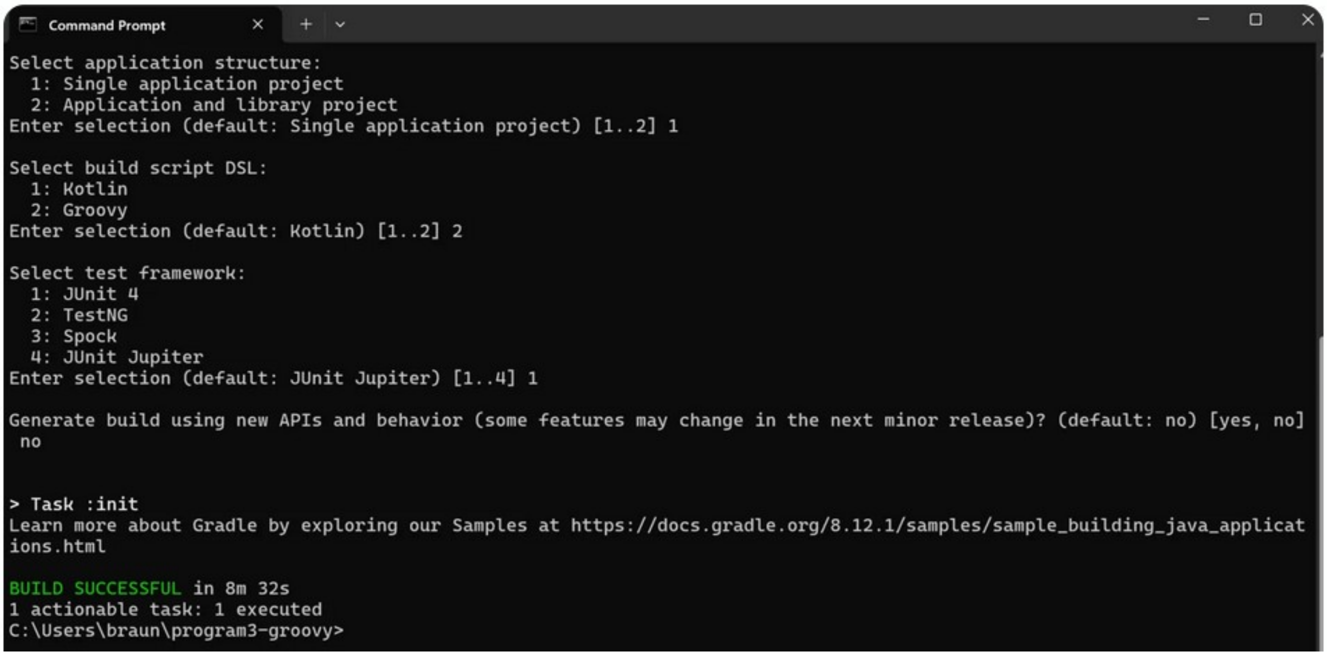
In this case, **`secondTask`** will depend on the completion of **`firstTask`** before it runs.

Working with Gradle Project (Groovy DSL):

Step 1: Create a new Project

```
gradle init --type java-application
```

- while creating project it will ask necessary requirement:
  - **Enter target Java version (min: 7, default: 21):** 17
  - **Project name (default: program3-groovy):** groovyProject
  - **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: Kotlin) [1..2]** 2
  - **Select test framework:**
    - 1: JUnit 4
    - 2: TestNG
    - 3: Spock
    - 4: JUnit Jupiter
      - **Enter selection (default: JUnit Jupiter) [1..4]** 1
  - **Generate build using new APIs and behavior (some features may change in the next minor release)? (default: no) [yes, no]**
    - no



```
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: Kotlin) [1..2] 2

Select test framework:
  1: JUnit 4
  2: TestNG
  3: Spock
  4: JUnit Jupiter
Enter selection (default: JUnit Jupiter) [1..4] 1

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


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

BUILD SUCCESSFUL in 8m 32s
1 actionable task: 1 executed
C:\Users\braun\program3-groovy>
```

Step 2: `build.gradle` (Groovy DSL)

```
plugins {
    id 'application'
}

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

repositories {
    mavenCentral()
```

```
}

dependencies {
    testImplementation 'junit:junit:4.13.2'
}

test {
    outputs.upToDateWhen { false }

    testLogging {
        events "passed", "failed", "skipped"
        exceptionFormat "full"
        showStandardStreams = true
    }
}
```

Step 3: `AdditionOperation.java`(Change file name and update below code)

- After creating project **change the file name**.
- Manually navigate the folder path like **src/main/java/org/example/**
- Change the file name **App.java** to **AdditionOperation.java**
- After then open that file and copy the below code and past it, save it.

```
package com.example;

public class AdditionOperation {
    public static void main(String[] args) {
        double num1 = 5;
        double num2 = 10;

        double sum = num1 + num2;

        System.out.printf("The sum of %.2f and %.2f is %.2f%n", num1, num2, sum);
    }
}
```

Step 4: `AdditionOperationTest.java` (JUnit Test) (Change file name and update below code)

- After creating project **change the file name**.
- Manually navigate the folder path like **src/test/java/org/example/**
- Change the file name **AppTest.java** to **`AdditionOperationTest`.java**
- After then open that file and copy the below code and past it, save it.

```
package com.example;

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

public class AdditionOperationTest {

    @Test
    public void testAddition() {
        double num1 = 5;
        double num2 = 10;
        double expectedSum = num1 + num2;

        double actualSum = num1 + num2;

        assertEquals(expectedSum, actualSum, 0.01);
    }
}
```

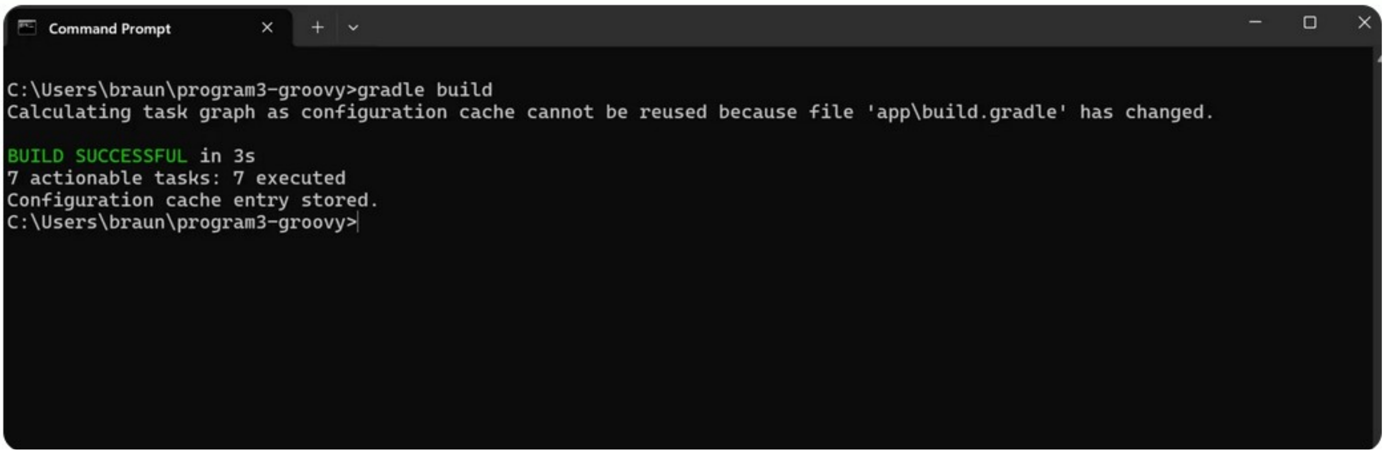
Step 5: Run Gradle Commands

- To **build** the project:

```
gradle build
```

To **run** the project:

```
gradle run
```

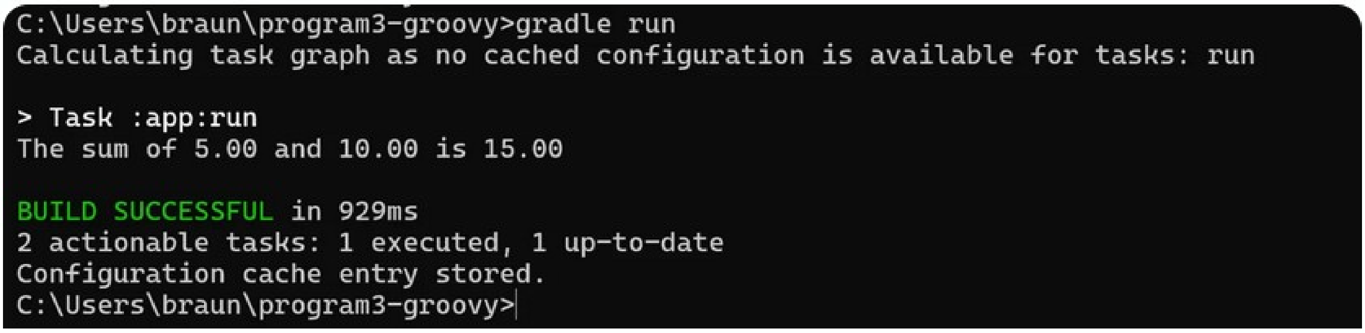


```
C:\Users\braun\program3-groovy>gradle build
Calculating task graph as configuration cache cannot be reused because file 'app\build.gradle' has changed.

BUILD SUCCESSFUL in 3s
7 actionable tasks: 7 executed
Configuration cache entry stored.
C:\Users\braun\program3-groovy>
```

To **run** the project:

```
gradle run
```



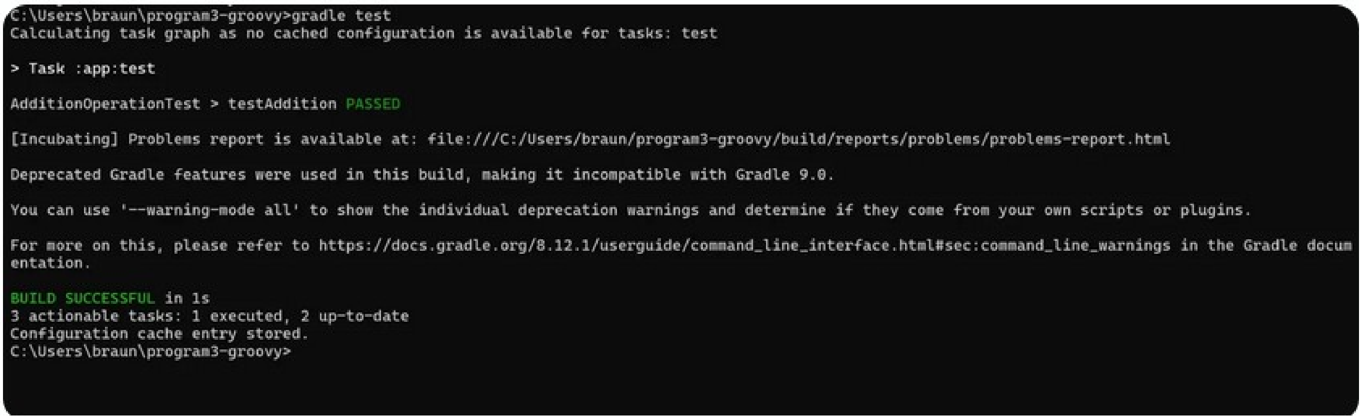
```
C:\Users\braun\program3-groovy>gradle run
Calculating task graph as no cached configuration is available for tasks: run

> Task :app:run
The sum of 5.00 and 10.00 is 15.00

BUILD SUCCESSFUL in 929ms
2 actionable tasks: 1 executed, 1 up-to-date
Configuration cache entry stored.
C:\Users\braun\program3-groovy>
```

To **test** the project:

```
gradle test
```



```
C:\Users\braun\program3-groovy>gradle test
Calculating task graph as no cached configuration is available for tasks: test

> Task :app:test

AdditionOperationTest > testAddition PASSED

[Incubating] Problems report is available at: file:///C:/Users/braun/program3-groovy/build/reports/problems/problems-report.html

Deprecated Gradle features were used in this build, making it incompatible with Gradle 9.0.

You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come from your own scripts or plugins.

For more on this, please refer to https://docs.gradle.org/8.12.1/userguide/command_line_interface.html#sec:command_line_warnings in the Gradle docum
entation.

BUILD SUCCESSFUL in 1s
3 actionable tasks: 1 executed, 2 up-to-date
Configuration cache entry stored.
C:\Users\braun\program3-groovy>
```

Working with Gradle Project (Kotlin DSL):

Step 1: Create a new Project

```
gradle init --type java-application
```

- while creating project it will ask necessary requirement:
    - **Enter target Java version (min: 7, default: 21):** 17
    - **Project name (default: program3-kotlin):** kotlinProject
    - **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: Kotlin) [1..2]** 1
    - **Select test framework:**
        - 1: JUnit 4
        - 2: TestNG
        - 3: Spock
        - 4: JUnit Jupiter
            - **Enter selection (default: JUnit Jupiter) [1..4]** 1
    - **Generate build using new APIs and behavior (some features may change in the next minor release)? (default: no) [yes, no]**
        - no

```
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: Kotlin) [1..2] 1

Select test framework:
  1: JUnit 4
  2: TestNG
  3: Spock
  4: JUnit Jupiter
Enter selection (default: JUnit Jupiter) [1..4] 1

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


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

BUILD SUCCESSFUL in 19s
1 actionable task: 1 executed
C:\Users\braun\program3-kotlin>
```

Step 2: `build.gradle.kts` (Kotlin DSL)

```
plugins {
    kotlin("jvm") version "1.8.21"
    application
}

repositories {
    mavenCentral()
```

```
}

dependencies {
    implementation(kotlin("stdlib"))
    testImplementation("junit:junit:4.13.2")
}

application {
    mainClass.set("com.example.MainKt")
}

tasks.test {
    useJUnit()

    testLogging {
        events("passed", "failed", "skipped")
        exceptionFormat = org.gradle.api.tasks.testing.logging.TestExceptionFormat.FULL
        showStandardStreams = true
    }

    outputs.upToDateWhen { false }
}

java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(17))
    }
}
```

Step 3: `Main.kt` (Change file name and update below code)

- After creating project **change the file name**.
- Manually navigate the folder path like **src/main/java/org/example/**
- Change the file name **App.java** to **Main.kt**
- After then open that file and copy the below code and past it, save it.

```
package com.example

fun addNumbers(num1: Double, num2: Double): Double {
    return num1 + num2
}

fun main() {
    val num1 = 10.0
    val num2 = 5.0
    val result = addNumbers(num1, num2)
    println("The sum of $num1 and $num2 is: $result")
}
```

Step 4: `MainTest`.kt (JUnit Test) (Change file name and update below code)

- After creating project **change the file name**.
- Manually navigate the folder path like **src/test/java/org/com/example/**
- Change the file name **MainTest.java** to **`MainTest`.kt**
- After then open that file and copy the below code and past it, save it.

```
package com.example

import org.junit.Assert.*
import org.junit.Test
```

```
class MainTest {

    @Test
    fun testAddNumbers() {
        val num1 = 10.0
        val num2 = 5.0

        val result = addNumbers(num1, num2)

        assertEquals("The sum of $num1 and $num2 should be 15.0", 15.0, result, 0.001)
    }
}
```

Step 5: Run Gradle Commands

- To **build** the project:

```
gradle build
```

```
Command Prompt
Learn more about Gradle by exploring our Samples at https://docs.gradle.org/8.12.1/samples/sample_building_java_applicat
ions.html

BUILD SUCCESSFUL in 19s
1 actionable task: 1 executed
C:\Users\braun\program3-kotlin>gradle build
Calculating task graph as no cached configuration is available for tasks: build

> Task :app:test

MainTest > testAddNumbers PASSED

[Incubating] Problems report is available at: file:///C:/Users/braun/program3-kotlin/build/reports/problems/problems-rep
ort.html

Deprecated Gradle features were used in this build, making it incompatible with Gradle 9.0.

You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come from your own sc
ripts or plugins.

For more on this, please refer to https://docs.gradle.org/8.12.1/userguide/command_line_interface.html#sec:command_line_
warnings in the Gradle documentation.

BUILD SUCCESSFUL in 5s
7 actionable tasks: 7 executed
Configuration cache entry stored.
C:\Users\braun\program3-kotlin>
```

To **run** the project:

```
gradle run
```

```
C:\Users\braun\program3-kotlin>gradle run
Calculating task graph as no cached configuration is available for tasks: run

> Task :app:run
The sum of 10.0 and 5.0 is: 15.0

[Incubating] Problems report is available at: file:///C:/Users/braun/program3-kotlin/build/reports/problems/problems-rep
ort.html

Deprecated Gradle features were used in this build, making it incompatible with Gradle 9.0.

You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come from your own sc
ripts or plugins.

For more on this, please refer to https://docs.gradle.org/8.12.1/userguide/command_line_interface.html#sec:command_line_
warnings in the Gradle documentation.

BUILD SUCCESSFUL in 1s
2 actionable tasks: 1 executed, 1 up-to-date
Configuration cache entry stored.
C:\Users\braun\program3-kotlin>
```

To **test** the project:

```
gradle test
```

```
C:\Users\braun\program3-kotlin>gradle test
Calculating task graph as no cached configuration is available for tasks: test

> Task :app:test

MainTest > testAddNumbers PASSED

[Incubating] Problems report is available at: file:///C:/Users/braun/program3-kotlin/build/reports/problems/problems-rep
ort.html

Deprecated Gradle features were used in this build, making it incompatible with Gradle 9.0.

You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come from your own sc
ripts or plugins.

For more on this, please refer to https://docs.gradle.org/8.12.1/userguide/command_line_interface.html#sec:command_line_
warnings in the Gradle documentation.

BUILD SUCCESSFUL in 1s
3 actionable tasks: 1 executed, 2 up-to-date
Configuration cache entry stored.
C:\Users\braun\program3-kotlin>
```

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

Step 1: Creating a Maven Project

You can create a **Maven project** using the **`mvn`** command (or through your **IDE**, as mentioned earlier). But here, I'll give you the essential **`pom.xml`** and **Java code**.

- **I'm Using Command Line:**
  - To create a basic Maven project using the command line, you can use the following command:

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

Step 2: Open The `pom.xml` File

- You can manually navigate the **project folder** named call **maven-example** and open the file pom.xml and copy the below code and paste it then save it.
- In case if you not getting project folder then type command in your cmd.
  - **cd maven-example** – is use to navigate the project folder.
  - **notepad pom.xml** – is use to open pom file in notepad.

```
<?xml version="1.0" encoding="UTF-8"?>
<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>maven-example</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <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>
```

Step 3: Open Java Code (`App.java`) File

- Open a file **`App.java`** inside the **`src/main/java/com/example/`** directory.
- After opening the **App.java** copy the below code and paste it in that file then save it.

```
package com.example;
```

```
public class App {
    public static void main(String[] args) {
        System.out.println("Hello, Maven");
        System.out.println("This is the simple realworld example....");

        int a = 5;
        int b = 10;
        System.out.println("Sum of " + a + " and " + b + " is " + sum(a, b));
    }

    public static int sum(int x, int y) {
        return x + y;
    }
}
```

Note: before building the project make sure you are in the project folder if not navigate the project folder type command in your command prompt **`cd` maven-example**.

Step 4: Run the Project

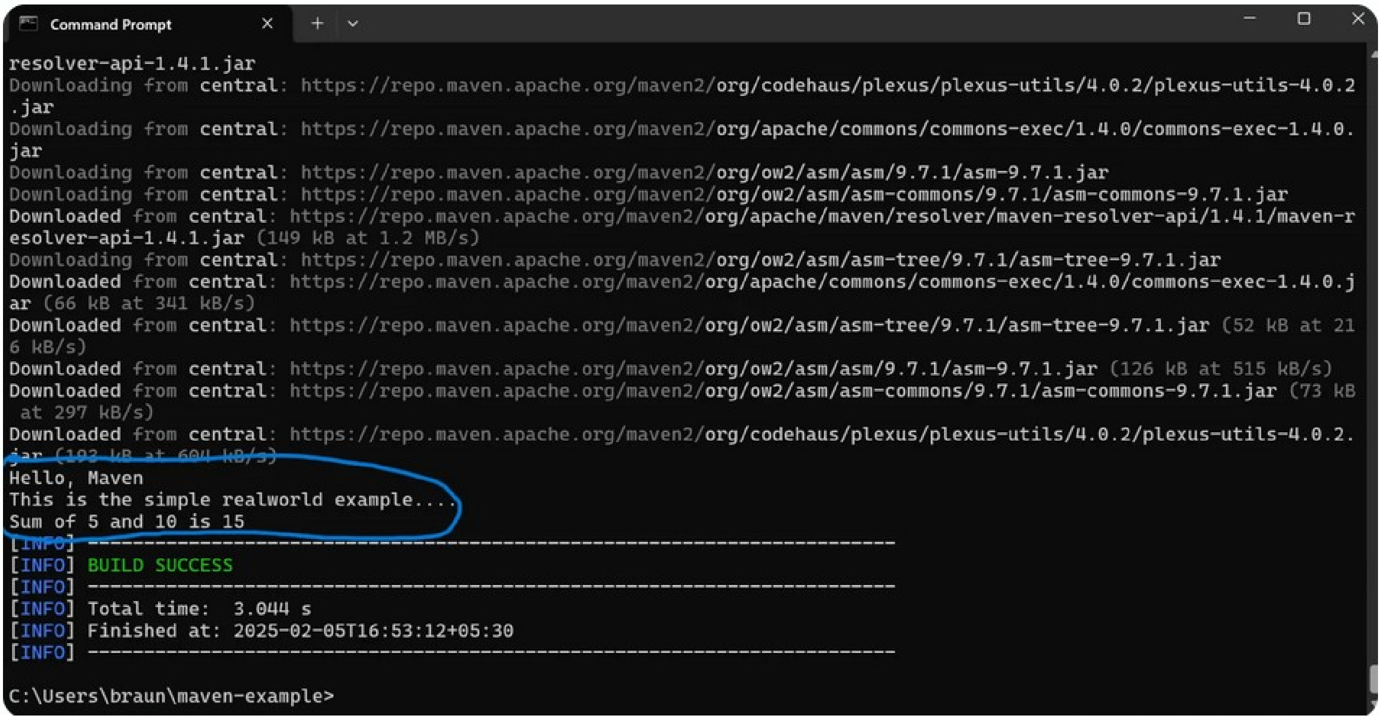
To build and run this project, follow these steps:

- Open the terminal in the project directory and run the following command to build the project.

```
mvn clean install
```

- Run the program with below command:

```
mvn exec:java -Dexec.mainClass="com.example.App"
```



Step 5: Migrate the Maven Project to Gradle

1. **Initialize Gradle**: Navigate to the project directory (**`gradle-example`**) and run:

```
gradle init
```

- It will ask **Found a Maven build. Generate a Gradle build from this? (default: yes) [yes, no]**

  - Type **Yes**
- **Select build script DSL:**
  - 1: Kotlin
  - 2: Groovy
  - Enter selection (default: Kotlin) [1..2]
    - Type **2**
- **Generate build using new APIs and behavior (some features may change in the next minor release)? (default: no) [yes, no]**
  - Type **No**

```
C:\Users\braun\maven-example>gradle init

Found a Maven build. Generate a Gradle build from this? (default: yes) [yes, no] yes

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

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


> Task :init
Maven to Gradle conversion is an incubating feature.
For more information, please refer to https://docs.gradle.org/8.12.1/userguide/migrating_from_maven.html in the Gradle d
ocumentation.

BUILD SUCCESSFUL in 3m 56s
1 actionable task: 1 executed
C:\Users\braun\maven-example>
```

2. Navigate the project folder and open `build.gradle` file then add the below code and save it.

```
plugins {
    id 'java'
}

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

repositories {
    mavenCentral()
}

dependencies {
    testImplementation 'junit:junit:4.12'
}

task run(type: JavaExec) {
    main = 'com.example.App'
    classpath = sourceSets.main.runtimeClasspath
}
```

Step 6: Run the Gradle Project

- **Build the Project**: In the project directory (`gradle-example`), run the below command to build the project:

```
gradlew build
```

```
C:\Users\braun\maven-example>gradlew build
Calculating task graph as no cached configuration is available for tasks: build

[Incubating] Problems report is available at: file:///C:/Users/braun/maven-example/build/reports/problems/problems-repor
t.html

Deprecated Gradle features were used in this build, making it incompatible with Gradle 9.0.

You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come from your own sc
ripts or plugins.

For more on this, please refer to https://docs.gradle.org/8.12.1/userguide/command_line_interface.html#sec:command_line_
warnings in the Gradle documentation.

BUILD SUCCESSFUL in 2s
4 actionable tasks: 4 executed
Configuration cache entry stored.
C:\Users\braun\maven-example>
```

Run the Application: Once the build is successful, run the application using below command:

```
gradlew run
```

```
C:\Users\braun\maven-example>gradlew run
Calculating task graph as no cached configuration is available for tasks: run

> Task :run
Hello, Maven
This is the simple realworld example....
Sum of 5 and 10 is 15

[Incubating] Problems report is available at: file:///C:/Users/braun/maven-example/build/reports/problems/problems-repor
t.html

Deprecated Gradle features were used in this build, making it incompatible with Gradle 9.0.

You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come from your own sc
ripts or plugins.

For more on this, please refer to https://docs.gradle.org/8.12.1/userguide/command_line_interface.html#sec:command_line_
warnings in the Gradle documentation.

BUILD SUCCESSFUL in 1s
2 actionable tasks: 1 executed, 1 up-to-date
Configuration cache entry stored.
C:\Users\braun\maven-example>
```

Step 7: Verify the Migration

- **Compare the Output:** Make sure that both the **Maven** and **Gradle** builds produce the same output:
  - **Maven Output**:

```
Hello, Maven
This is the simple realworld example....
Sum of 5 and 10 is 15
```

- **Gradle Output**:

```
Hello, Maven
This is the simple realworld example....
Sum of 5 and 10 is 15
```

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

Introduction to Jenkins

What is Jenkins?

Jenkins is an open-source automation server widely used in the field of Continuous Integration (CI) and Continuous Delivery (CD). It allows developers to automate the building, testing, and deployment of software projects, making the development process more efficient and reliable.

Key features of Jenkins:

- **CI/CD**: Jenkins supports Continuous Integration and Continuous Deployment, allowing developers to integrate code changes frequently and automate the deployment of applications.
- **Plugins**: Jenkins has a vast library of plugins that can extend its capabilities. These plugins integrate Jenkins with version control systems (like Git), build tools (like Maven or Gradle), testing frameworks, deployment tools, and much more.
- **Pipeline as Code**: Jenkins allows the creation of pipelines using Groovy-based DSL scripts or YAML files, enabling version-controlled and repeatable pipelines.
- **Cross-platform**: Jenkins can run on various platforms such as Windows, Linux, macOS, and others.

Installing Jenkins

Jenkins can be installed on local machines, on a cloud environment, or even in containers. Here's how you can install Jenkins in Window local System environments:

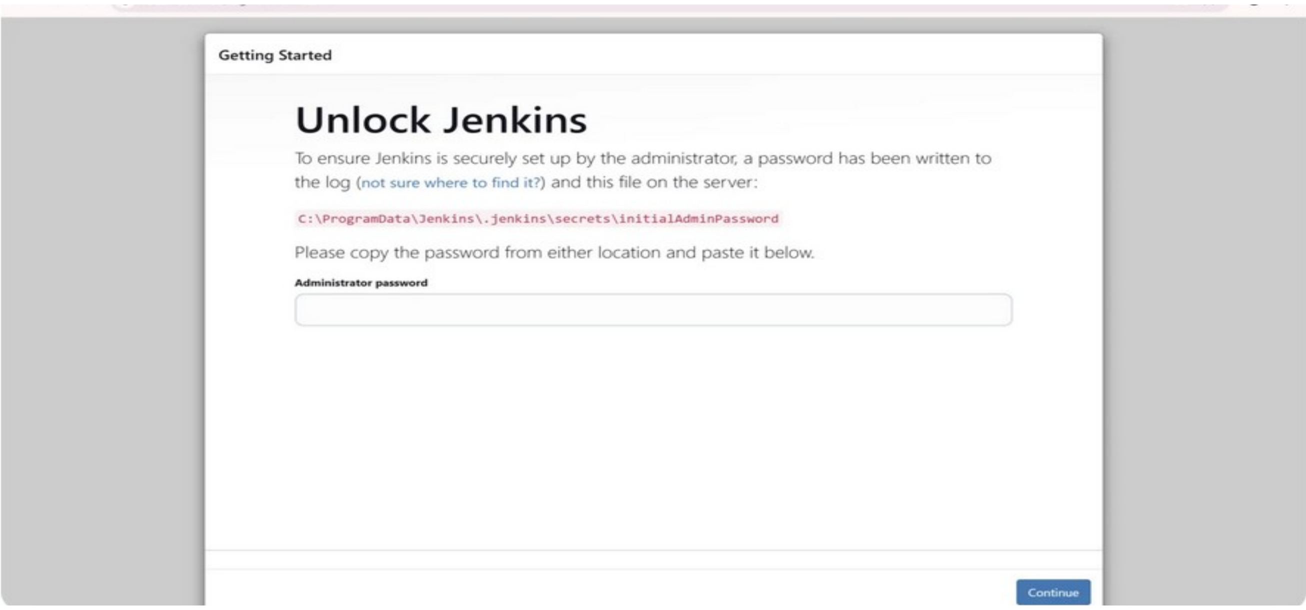
1. **Installing Jenkins Locally**

Step-by-Step Guide (Window):

1. **Prerequisites:**
   - Ensure that J**ava (JDK) is installed** on your system. **Jenkins requires Java 21**. If not then **click here**.
   - You can check if Java is installed by running `java -version` in the terminal.
2. **Install Jenkins on Window System):**
   - Download the Jenkins Windows installer from the official Jenkins website.
   - Run the installer and follow the on-screen instructions. While installing choose **login system**: **run service as LocalSystem (not recommended)**.
   - After then use **default port** or you can **configure you own port like I'm using port 3030** then **click on test** and **next**.
   - After then **change the directory** and choose **java jdk-21** path look like **C:\Program Files\Java\jdk-21**.
   - After then **click next, next** and then it will **ask permission click on yes** and it will start installing.
   - After successfully installed, Jenkins will be **running on port either default port** or **chosen port** like i choose **port 3030** by default (you can access it in your browser at `http://localhost:8080`) or `http://localhost:3030`.

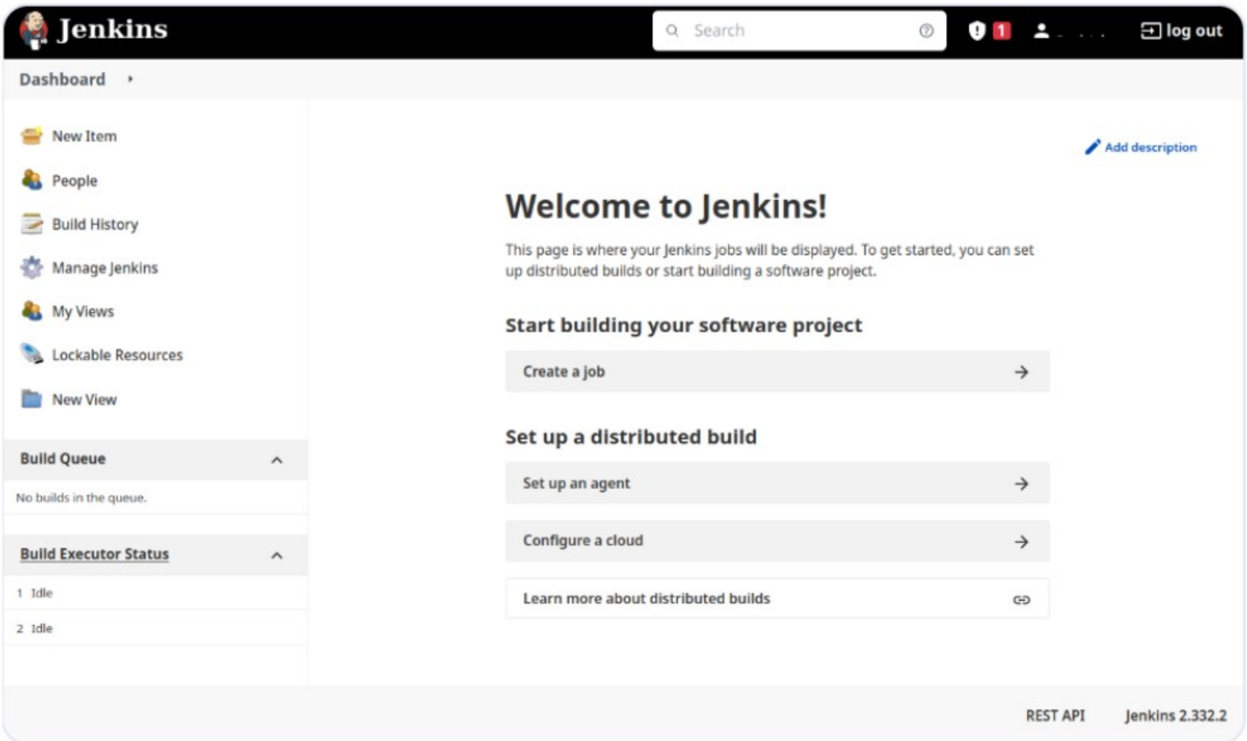
2. Jenkins Setup in browser:

- After opening browser by visiting your local address the browser should look like below screenshot.



- It will ask **administrator password** so you have to **navigate the above highlighted path** and open that **`initialAdminPassword` in notepad** or any software to **see the password**.
- Just c**opy that password and paste** it and **click on continue**.
- It will ask to **customize Jenkins** so **click on install suggested plugin** it will **automatically install all required plugin**.
- After then **create admin profile** by **filling all details** then **click on save and continue** after then **save and finish** after then **click on start using Jenkin**.

Click Start using Jenkins to visit the main Jenkins dashboard:



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

How Is Jenkins Used for Continuous Integration?

Continuous Integration (CI) is a software development practice where developers integrate code into a shared repository frequently, usually several times a day. Jenkins is an open-source automation server that facilitates CI by automating the build, testing, and deployment processes.

With Jenkins, developers can easily detect and fix integration issues early, improving collaboration and accelerating software delivery. By continuously integrating code, teams can maintain a high level of code quality, reduce development time, and minimize the risk of release failures. Continuous Integration Features in Jenkins Continuous integration involves the automatic building and testing of code whenever changes are committed to the version control system. Jenkins provides several features that facilitate CI, including:

Continuous Integration Features in Jenkins

Continuous integration involves the automatic building and testing of code whenever changes are committed to the version control system. Jenkins provides several features that facilitate CI, including:

● Version control system integration: Jenkins integrates with various version control systems (VCS) such as Git, Subversion, and Mercurial. This allows Jenkins to monitor repositories for changes, trigger builds, and incorporate updates automatically.

● Build automation: Jenkins supports build automation using build tools like Maven, Gradle, and Ant. It can compile, package, and deploy code, ensuring that the latest changes are continuously integrated into the

software project.

● Automated testing: Jenkins can execute automated tests for each build, using testing frameworks like JUnit, TestNG, and Selenium. This ensures that any issues introduced during development are quickly detected and reported, allowing developers to address them promptly.

● Pipeline as code: Jenkins Pipeline allows users to define their entire CI/CD pipeline as code using a domain-specific language called “Groovy.” This makes the pipeline easily versionable, shareable, and more maintainable.

● Distributed builds: Jenkins supports distributed builds across multiple build agents, which allows for faster and more efficient build processes by distributing the workload across multiple machines.

● Plugins and extensibility: Jenkins offers a vast ecosystem of plugins that extend its functionality, allowing users to customize and adapt Jenkins to their specific needs. Plugins are available for various tasks, such as

integrating with different VCS, build tools, notification systems, and more.

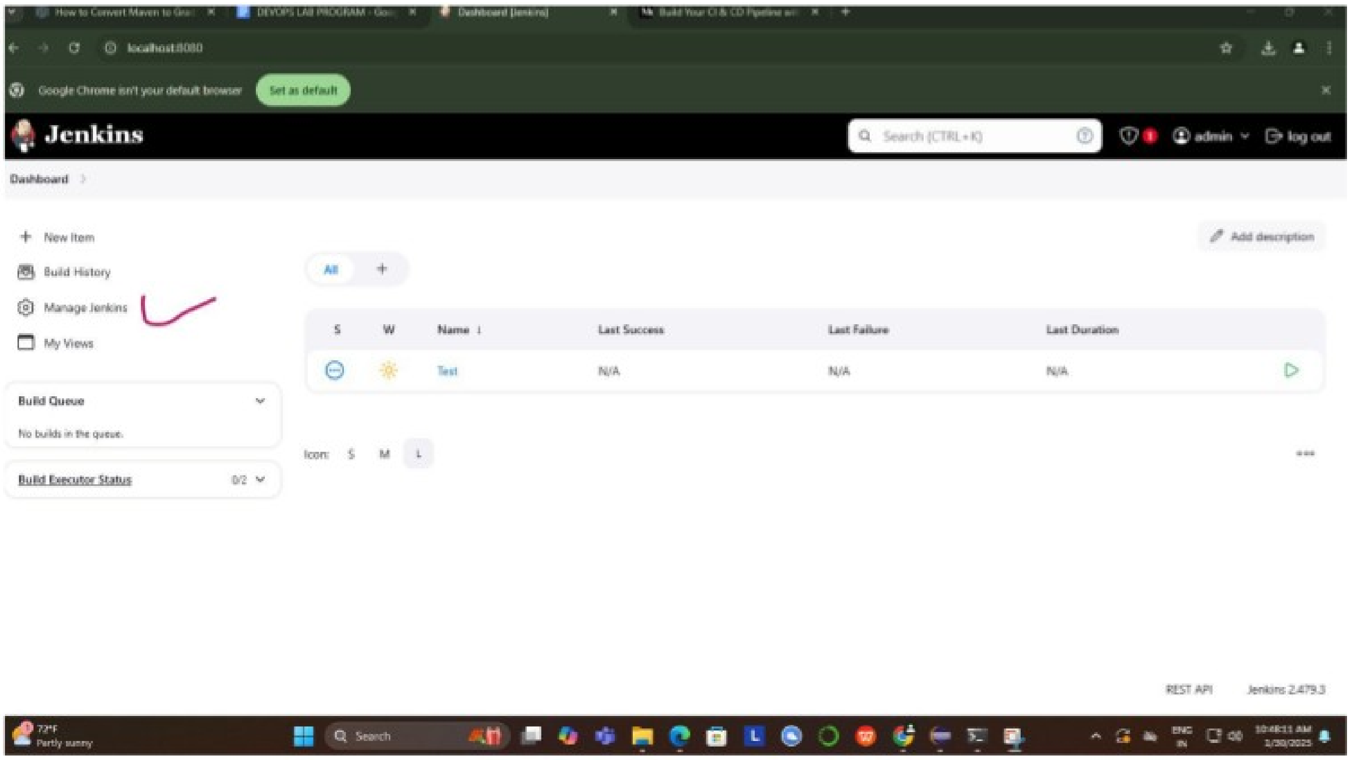
● Notifications and reporting: Jenkins can send notifications through various channels like email, Slack, or other messaging systems to keep the team informed about build status, test results, and potential issues. It also generates reports and visualizations for various metrics, such as test results, code coverage, and build trends.

● Access control and security: Jenkins provides fine-grained access control and user management, allowing administrators to control who can access specific projects, pipelines, or configuration settings. It also supports integration with LDAP and Active Directory for centralized user management.

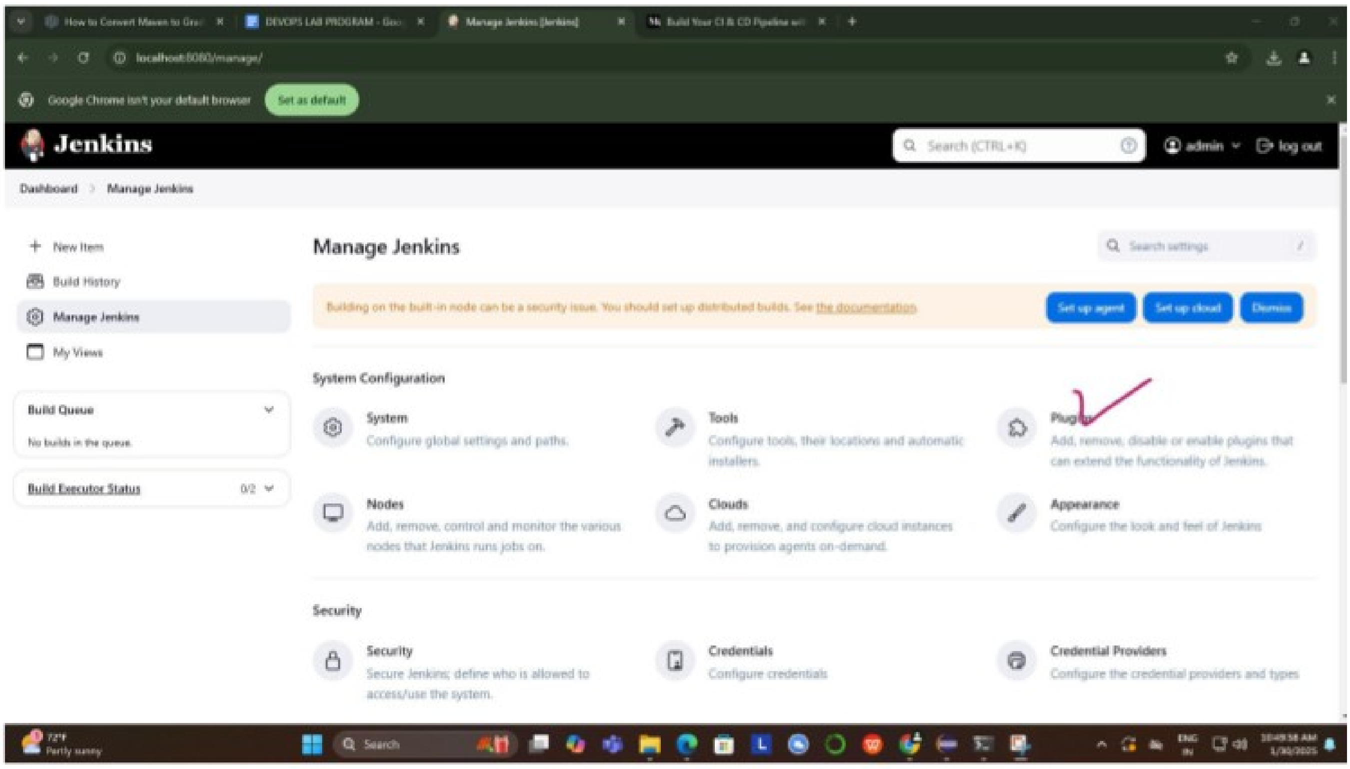
● REST API: Jenkins exposes a REST API that enables users to interact with Jenkins programmatically, allowing for integration with external tools, automation, and custom applications.

STEP1: Now coming to our Program to set CI Pipeline for Maven

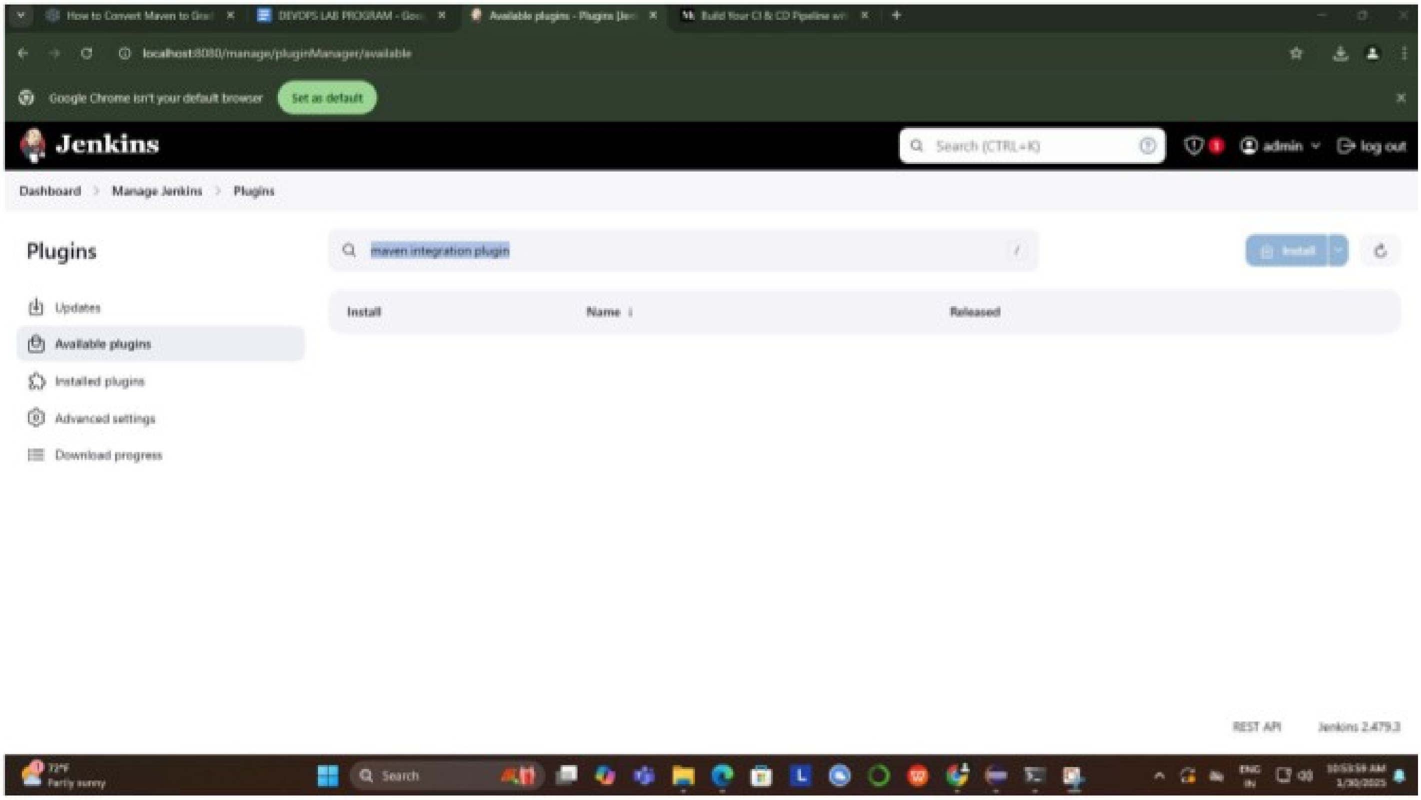
Go to Jenkins Dashboard and Click on Manage Jenkins



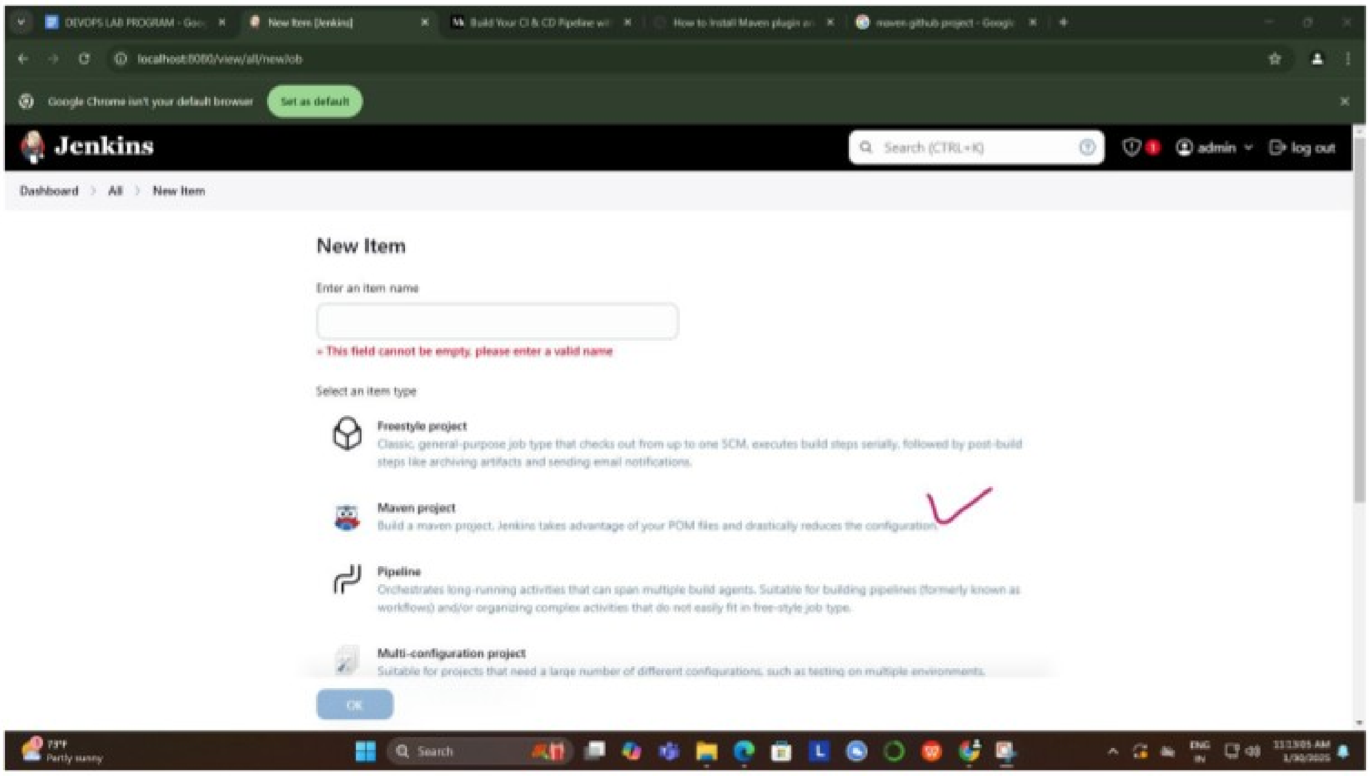
STEP2: Select Plugins



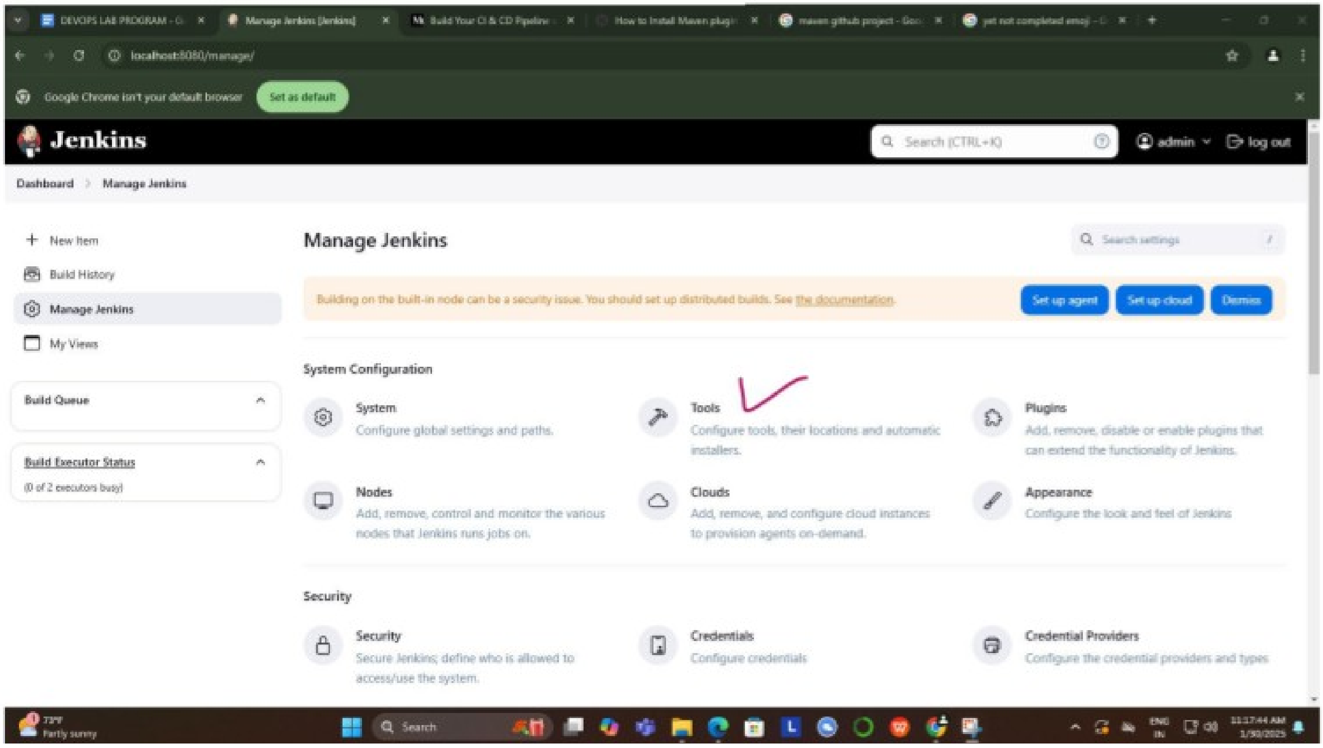
STEP3: Search for Maven IntegrationPlugin in Available Plugins and Install



STEP4: After Maven Integration Plugin is Installed We able to see Maven Project as New Item

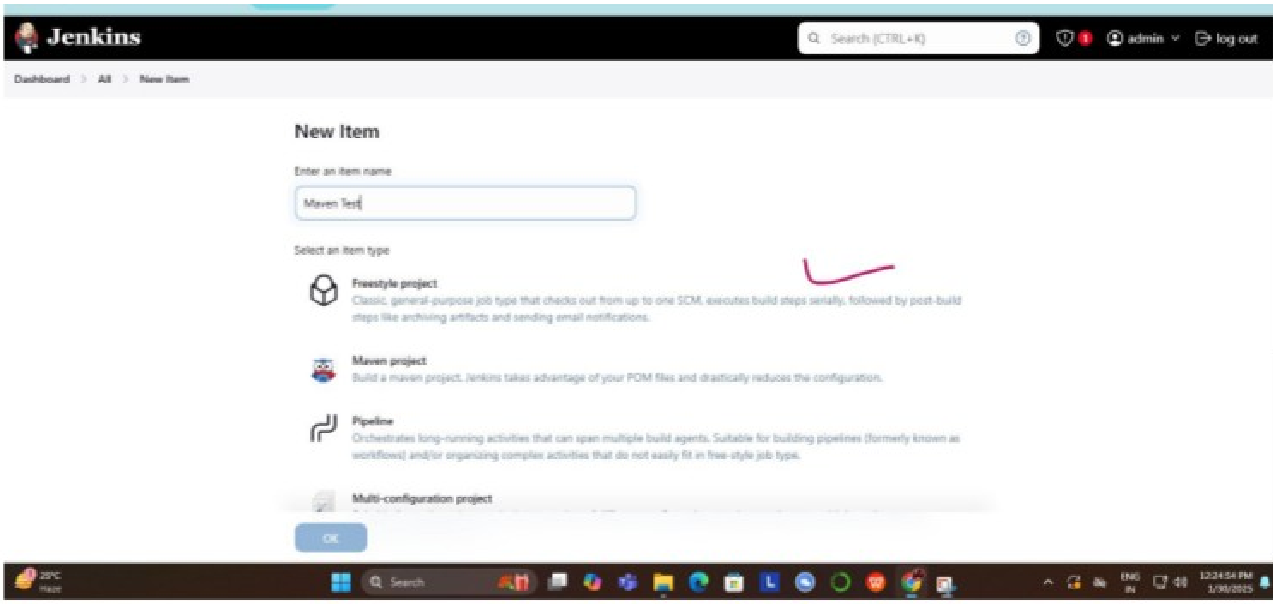


STEP5: YET not completed we have to configure the Location to properly Build and Run Maven Project So again click on Manage Jenkins and select Tools

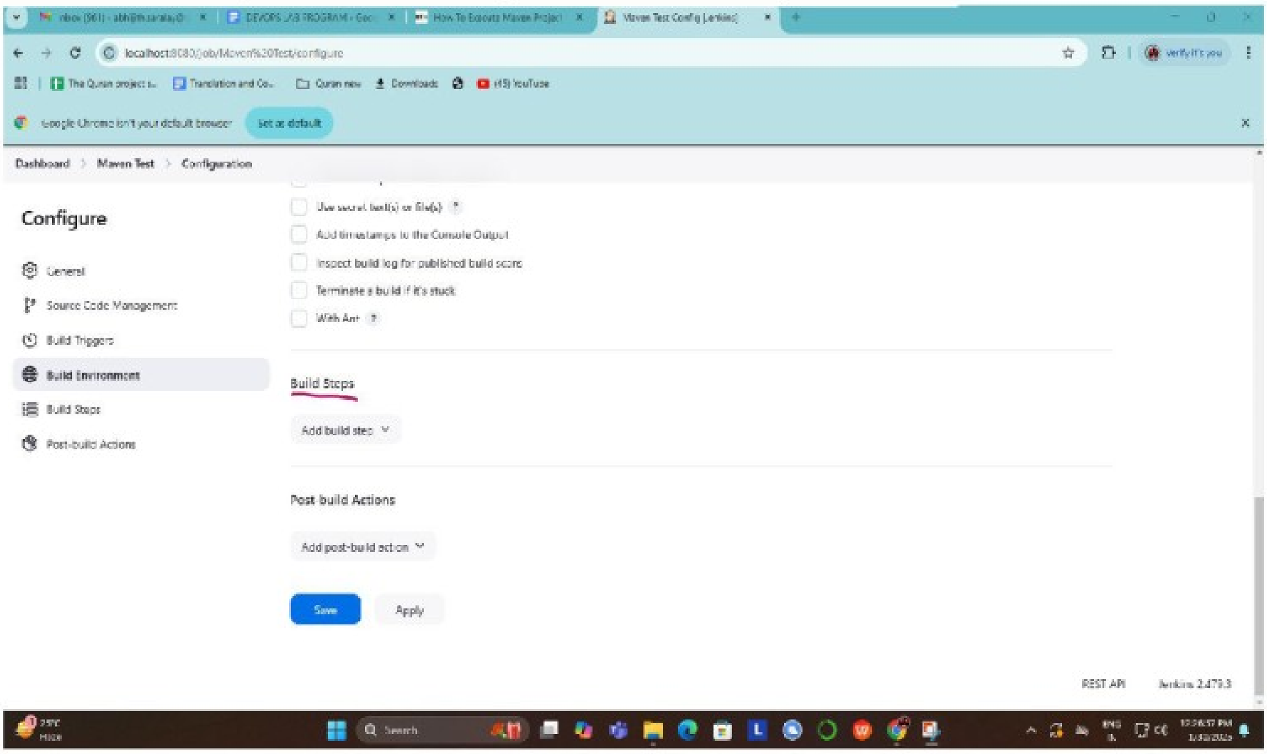


STEP6: Now lets not select Maven Project as new Item as we already have Maven project in local systems lets see how we can run the Maven Project with POM.XML

- Click on New Item
- Provide Item Name and select Freestyle Project



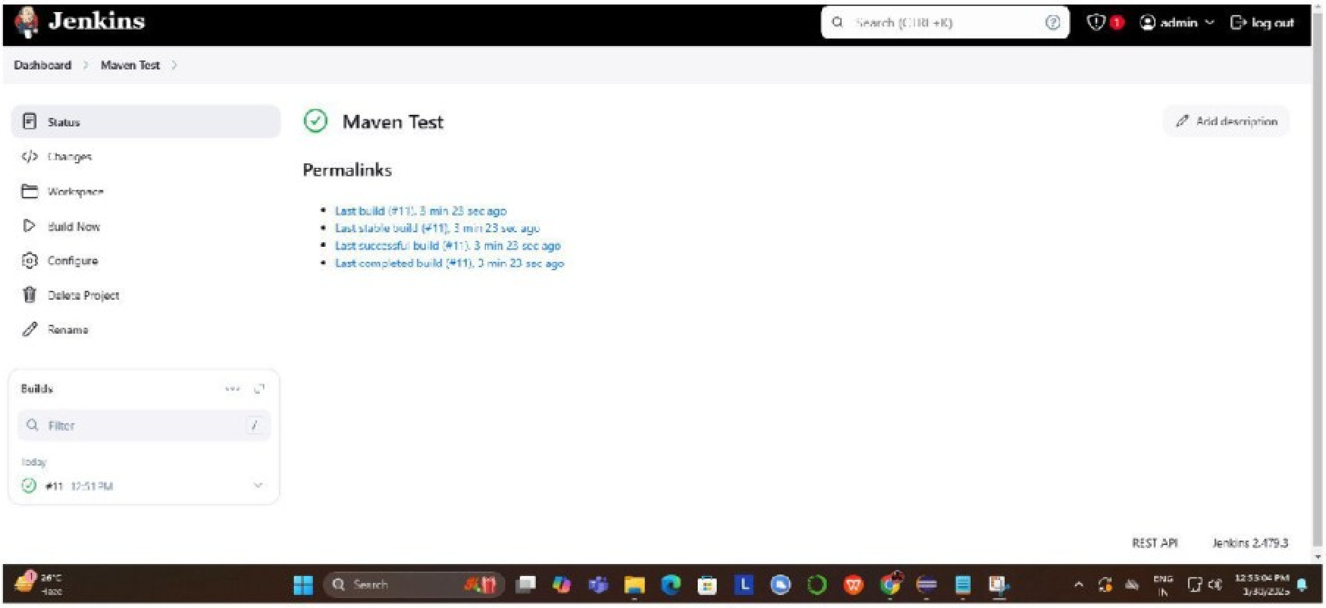
- Scroll down to ‘Build‘ option. Click on ‘Add Build Step‘ and choose the value ‘Invoke top-level Maven targets‘ from the drop down list.



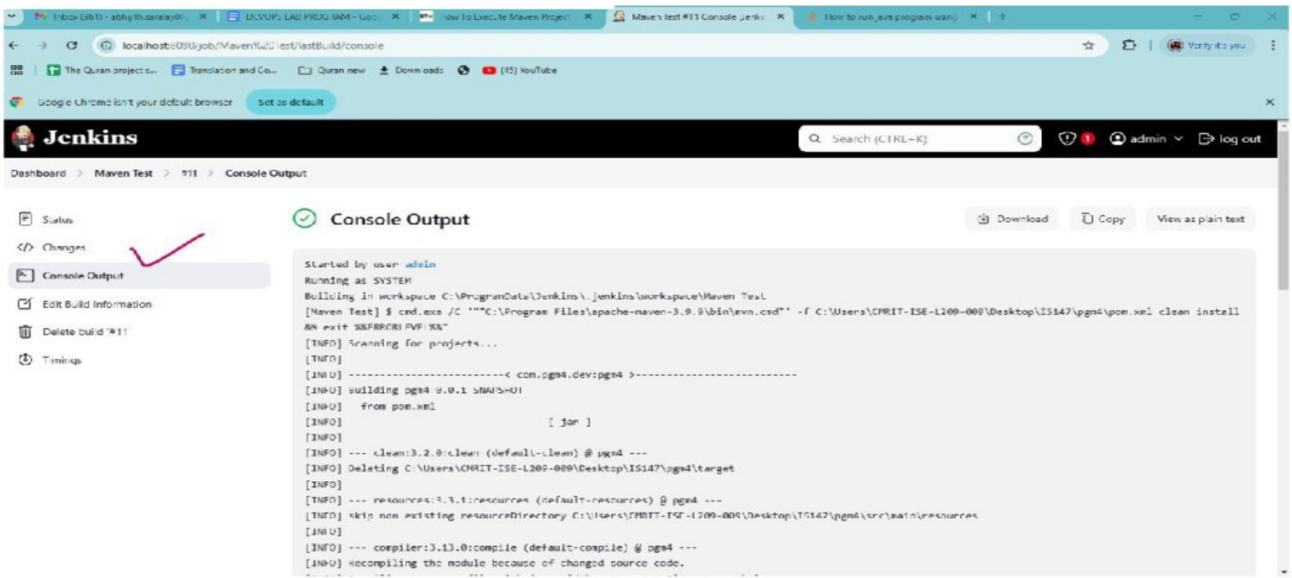
- After selecting Invoke top-level Maven targets opt for proper environment version as in set in previous steps in my case its MAVEN_HOME

- Enter Goal as clean install
- Before you save and apply just below Goal there is Advance option add pom.xml path

Goto pom.xml of your particular pgm and take path in my case its C:\Users\CMRIT-ISE-L209-009\Desktop\IS147\pgm4\pom.xml
After all Steps is over click on Build button the output be as in below



To see either click on build texts and goto console output or u can goto dashboard and opt to see build scripts.



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.

Components required: Java (version jdk 17 or 23 and 21), Ansible
Thoery :
Ansible is a powerful automation tool used for IT tasks such as configuration management, application deployment, and orchestration. It uses simple, human-readable YAML files to define automation tasks. Below are the key concepts of Ansible: inventory, playbooks, modules, and automating server configurations with playbooks.
1. Inventory
An inventory in Ansible is a file that contains information about the servers or machines you want to manage. It defines the hosts that Ansible will interact with. This file is often in INI or YAML format.

INI-style example:

```
Ini
[web_servers]
web1.example.com
web2.example.com
[db_servers]
db1.example.com
db2.example.com
```

YAML-style example::

```
yaml
all:
children:
web_servers:
hosts:
web1.example.com:
web2.example.com:
db_servers:
hosts:
db1.example.com:
db2.example.com:
```

2. Playbooks
A playbook is a YAML file that defines a series of tasks to be executed on one or more hosts. Playbooks can define multiple plays, where each play is a set of tasks run on a group of hosts.

Structure of a Playbook:

- Hosts: Define the target hosts (defined in the inventory).
- Tasks: Define the actions to be executed (like installing software, creating files, etc.).
- Variables: You can define variables to be used within the playbook.
- Handlers: Special tasks that only run when notified by other tasks.
- Roles: Reusable sets of tasks, handlers, and variables.

Basic Playbook Example:

```
yaml
---
```

```
- name: Install Apache and start service
hosts: web_servers
become: yes
tasks:
- name: Install Apache
apt:
name: apache2
state: present
update_cache: yes
- name: Start Apache service
service:
name: apache2
state: started
enabled: yes
```

3. Modules

Ansible modules are the building blocks of Ansible tasks. Each module performs a specific function (e.g., install packages, copy files, start services).

Some common Ansible modules:

- **apt:** Manages packages for Debian/Ubuntu systems.
- **yum:** Manages packages for RHEL/CentOS systems.
- **service:** Manages services (start, stop, restart).
- **copy:** Copies files from the local machine to the remote machine.
- **file:** Manages file properties like permissions.

Example of a module in a playbook:

```
yaml
- name: Install Nginx
apt:
name: nginx
state: present
update_cache: yes
```

4. Automating Server Configurations with Playbooks

You can use playbooks to automate server configurations. For example, automating the installation and configuration of software packages, setting up users, managing services, and more.

Here's an example of automating server configuration to install Apache and configure it with a custom config-uration file:

Example Playbook for Server Configuration:

```
yaml
---
- name: Configure web server
  hosts: web_servers
  become: yes

  tasks:
    - name: Install Apache2
      apt:
        name: apache2
        state: present
```

```
- name: Copy custom Apache config file
  copy:
      src: /local/path/to/apache2.conf
      dest: /etc/apache2/apache2.conf
      owner: root
      group: root
      mode: '0644'
- name: Ensure Apache is running
  service:
       name: apache2
       state: started
       enabled: yes
```

In this example:

- Apache is installed.
- A custom configuration file is copied from the local machine to the server.
- Apache service is started and enabled to start on boot.

NOTE:

- Inventory is a list of managed hosts.
- Playbooks define what tasks should be run on which hosts.
- Modules provide the functionality to perform tasks (like installing software, managing services, etc.).
- You can automate server configurations by writing playbooks that include necessary tasks for the desired configuration.

With Ansible, you can automate server setups, configurations, and management in a repeatable, scalable way. It also allows version-controlled, auditable configurations across your infrastructure.

Writing and running a basic playbook:

Let's walk through writing and running a basic Ansible playbook step-by-step. This example will demonstrate how to set up a basic Apache web server on remote hosts using a playbook.

Prerequisites

- You have Ansible installed on your local machine.
- You have one or more remote hosts (e.g., virtual machines or cloud instances) where you have SSH access and Ansible can run commands.
- You know the IP addresses or hostnames of your remote hosts.
- Ensure SSH keys are set up or passwords are configured for authentication.

Step 1: Setup Your Inventory File

First, you need to create an inventory file that lists your remote servers.

1. Create a file called hosts.ini (or use hosts.yml if you prefer YAML format).

Example of hosts.ini:

```
ini
[web_servers]
192.168.1.10
192.168.1.11
```

Here, we have two remote servers (192.168.1.10 and 192.168.1.11) under the web_servers group.

Step 2: Write the Playbook

Next, create a playbook that will be responsible for setting up Apache on the remote servers.
1. Create a file called install_apache.yml (or any name you prefer).
Example of install_apache.yml Playbook:

```
yaml
---
- name: Install and configure Apache web server
hosts: web_servers # Targets the hosts defined under the "web_servers" group
become: yes
# Use sudo privileges for tasks that require admin access
tasks:
- name: Install Apache
apt:
name: apache2
# Package to be installed
state: present
# Ensures that Apache is installed
update_cache: yes # Update apt cache before installing
- name: Start Apache service
service:
name: apache2
# Apache service name
state: started # Ensures the service is running
enabled: yes
# Ensures Apache starts on boot
- name: Ensure the Apache homepage is available
copy:
content: "<html><body><h1>Welcome to the Apache Server!</h1></body></html>"
dest: /var/www/html/index.html # The default location for the Apache homepage
mode: '0644'
```

This playbook does the following:

- Installs Apache on the remote servers.
- Starts and enables the Apache service.
- Deploys a simple HTML page to /var/www/html/index.html.

Step 3: Run the Playbook
Now that you’ve created your inventory and playbook, it’s time to run it.
Use the following command to run the playbook:

```
bash
ansible-playbook -i hosts.ini install_apache.yml
```

This command will:

Use the hosts.ini inventory to identify which servers to target.
Execute the tasks in install_apache.yml on those remote servers.

Step 4: Verify the Setup
Once the playbook completes successfully, you can verify the setup by:

1. Opening a web browser and navigating to the IP address of one of the remote servers (e.g., http://192.168.1.10).
2. You should see the message Welcome to the Apache Server! Displayed.

Alternatively, you can SSH into the remote servers and verify Apache is running:
bash
systemctl status apache2
This will show if Apache is active and running.
Troubleshooting

- If Ansible reports an error or task failure, check the logs for details on what went wrong. Common issues include missing sudo privileges or package manager issues (e.g., wrong package name or unavailability of repositories).
- Ensure the apt module is appropriate for the target operating system (Ubuntu/Debian). For RHEL-based systems, you may need to use the yum module instead.

Note:

- We created an inventory file (hosts.ini) with remote server IP addresses.
- We wrote a basic playbook (install_apache.yml) to install and configure Apache.
- We ran the playbook with the ansible-playbook command to execute tasks on remote servers.

Result: Successfully Written and run a basic playbook.

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

COMPONENTS REQUIRED: Java (version jdk 17 or 23 and 21), Maven (Version 10), Ansible, Jenkins.

THEORY:
Let's break it down into smaller tasks:
1. Set Up Jenkins for Continuous Integration (CI)
2. Create a Maven Project in Jenkins
3. Configure Jenkins Build Pipeline (CI Pipeline)
4. Use Ansible to Deploy Artifacts from Jenkins

Step 1: Set Up Jenkins for Continuous Integration

1. Install Jenkins:
- If Jenkins is not yet installed, you can follow the official installation guide for your OS: Jenkins Installation Guide.

2. Install Required Plugins:
- Open the Jenkins dashboard.
- Go to Manage Jenkins > Manage Plugins.
- Install the following plugins:
    - Maven Integration Plugin: For Maven support.
    - Ansible Plugin: For Ansible integration.
    - Git Plugin: To clone repositories from GitHub or other Git services.

3. Configure Global Tools in Jenkins:
- Go to Manage Jenkins > Global Tool Configuration.
- Set up JDK (if needed) and Maven (ensure Maven is installed on Jenkins).
- JDK: Configure the JDK that will be used to build your Maven project.
- Maven: Specify the installation of Maven.

4.Set up Ansible on Jenkins (if not already installed):
- Ansible must be installed on the Jenkins server or the nodes where the deployment will happen.
- You can set up Ansible globally under Manage Jenkins > Global Tool Configuration.
- Set the Ansible path on the Jenkins server.

Step 2: Create a Maven Project in Jenkins

1. Create a New Job in Jenkins:
- From the Jenkins dashboard, click New Item.
- Select Freestyle project (or Pipeline if you want to use pipeline scripts).
- Give the job a name (e.g., maven-build-project).

2. Configure Git Repository:
- Under Source Code Management, select Git.
- Provide the repository URL (e.g., GitHub, GitLab).
- Optionally, provide credentials if your repository is private.

3. Add Build Step for Maven:
- Under Build, select Invoke top-level Maven targets.
- Set the Goals to clean install to clean the project and build the .jar (or .war, .ear) file.
- Ensure the Maven Version is set to the one configured earlier.

Example:
Bash

clean install

4. Add Post-Build Actions:
  - You can configure post-build actions to archive the build artifacts (JAR/WAR file) so they can be used later in the pipeline.

Example:
  - Archive the artifact (e.g., target/*.jar or target/*.war).
  - Select Archive the artifacts in the Post-build Actions section.

In Files to archive, specify target/*.jar (or the appropriate artifact type).

Step 3: Configure Jenkins Build Pipeline (CI Pipeline)

1. Add Build Steps for Jenkins Pipeline (Optional):
  - If you want to use a more advanced CI pipeline using a Jenkins Pipeline Script, select Pipeline instead of Freestyle Project when creating a new job.

2. Configure the Pipeline Script:
  - In the Pipeline section, you can define the Jenkinsfile (pipeline script) directly in Jenkins or store it in the Git repository.

Example Jenkinsfile (Declarative Pipeline):

```
groovy
pipeline {
agent any
tools {
maven 'Maven-3.6.3' // Make sure Maven version is configured in Jenkins global tools
}
environment {
DEPLOY_SERVER = "deploy@your-server.com"
}
stages {
stage('Checkout') {
steps {
git 'https://github.com/your/repo.git'
}
}
stage('Build') {
steps {
sh 'mvn clean install' // Build the Maven project
}
}
stage('Archive Artifacts') {
steps {
archiveArtifacts artifacts: 'target/*.jar', allowEmptyArchive: true
}
}
stage('Deploy') {
steps {
ansiblePlaybook playbook: 'deploy.yml', inventory: 'inventory.ini' // Run the Ansible
playbook
}
}
```

```
}
post {
success {
echo 'Build and deploy completed successfully!'
}
failure {
echo 'Build failed!'
}
}
}
```

Step 4: Use Ansible to Deploy Artifacts from Jenkins

1. Create an Ansible Playbook: Ansible will be used to deploy the artifact generated by Jenkins (e.g., a .jar file) to the target server(s).

Create a file called deploy.yml (Ansible playbook) to deploy the artifact:

```
yaml
---
- name: Deploy artifact to server
  hosts: web_servers
  become: yes

  tasks:
   - name: Copy the artifact to the remote server
     copy:
         src: /path/to/your/artifact/target/my-app.jar # Jenkins workspace path
         dest: /opt/myapp/my-app.jar
         # Destination on the server
   - name: Start the application service
     systemd:
        name: myapp
        state: restarted
        enabled: yes
  - name: Ensure the application is running
      uri:
      url: "http://localhost:8080"
      status_code: 200
```

- This playbook copies the artifact (e.g., .jar file) to the remote server, restarts the application service, and checks if the application is running.

2. Create an Ansible Inventory File: Define the hosts where the artifacts will be deployed.

Create a file called inventory.ini (or inventory.yml if using YAML format):

```
ini
[web_servers]
192.168.1.10
192.168.1.11
```

3. Configure Ansible in Jenkins:

- In Jenkins, configure the Ansible Plugin with the path to your Ansible installation.
- You can specify the inventory.ini file in the pipeline and ensure Jenkins can run the playbook.

Step 5: Run the Pipeline

- Now, you can trigger the pipeline manually or set up a webhook for automatic builds.
- Jenkins will:
    1. Checkout your project from Git.
    2. Build the Maven project and generate the artifact.
    3. Archive the artifact.
    4. Use Ansible to deploy the artifact to the remote servers.

RESULT: We have successfully created a Jenkins CI pipeline to build a Maven project then uses Ansible to deploy the generated artifact (e.g., .jar) to the target server. With this setup, we have a fully automated Continuous Integration and Continuous Deployment (CI/CD) pipeline.

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

COMPONENTS REQUIRED: Java (version jdk 17 or 23 and 21), Maven (Version 10), Gradle (Version 10). Jenkins.

THEORY:

Azure DevOps is a cloud-based suite of development tools provided by Microsoft to support the complete software development lifecycle (SDLC). It includes a set of services that helps teamsplan, develop, test, and deliver applications efficiently. Azure DevOps is designed to support bothcontinuous integration (CI) and continuous delivery (CD), and it integrates seamlessly withvarious development platforms.

Overview of Azure DevOps Services

Azure DevOps offers several key services, each catering to different parts of the software development lifecycle:

1. Azure Repos: A set of version control tools (Git or Team Foundation Version Control – TFVC) that enables you to manage your code repositories, track changes, and collaborate with your team.

2. Azure Pipelines: A continuous integration and continuous delivery (CI/CD) service that automates the process of building, testing, and deploying code to different environments (e.g.,development, staging, production).

3. Azure Boards: A tool for agile project management that allows teams to plan, track, and discuss work. It includes features like Kanban boards, Scrum boards, user stories, and backlog management.

4. Azure Test Plans: Provides tools for manual and exploratory testing. It helps in tracking defects and managing test cases to ensure high-quality code.

5. Azure Artifacts: A service that enables teams to host and share packages (like NuGet, npm, and Maven) within their organization, promoting reuse and easier dependency management.

6. Azure DevOps Services for Collaboration: Features like dashboards, Wikis, and collaboration tools help teams work together effectively by providing visibility into the status of projects and workflows.

Setting Up an Azure DevOps Account

1. Create an Azure DevOps Account:

- Visit the [Azure DevOps portal](https://dev.azure.com/).

- If you don't have an existing Microsoft account, create one. If you have a Microsoft account (e.g., Outlook, Hotmail, or Office 365), you can sign in directly.

- After signing in, you’ll be asked to set up your Azure DevOps organization. Choose a unique name for your organization and the region where you want to host your data.

2. Create an Azure DevOps Project:

- After logging in, you’ll be redirected to the Azure DevOps dashboard.

- Click on New Project.

- Enter a name for your project, which will help identify your work in Azure DevOps.

- Choose the visibility of your project:

- Private: Only invited users can access the project.

- Public: The project is accessible by anyone.

- Select the version control system you want to use (either Git or TFVC).

- Choose the process template (Agile, Scrum, or CMMI) to manage your project's workflow.

- Click Create to set up the project.

Project Organization and Access Management

Once your project is created, you can:

1. Manage Users: You can add team members to your project and assign roles and permissions. Permissions can be fine-tuned to control access to various Azure DevOps services (e.g., repositories, pipelines, boards).

2. Set Up Repositories: In the Repos section, you can create Git repositories where you can store your project code, collaborate with your team, and manage branches.

3. Create Pipelines: Under Pipelines, you can configure Continuous Integration (CI) and Continuous Delivery (CD) workflows, defining the steps for building, testing, and deploying your application.

4. Plan Work: Using Azure Boards, you can create work items, user stories, and tasks, and organize them into sprints or iterations.

RESULT: Azure DevOps is a comprehensive suite of tools that helps teams manage and automate the development lifecycle efficiently. From version control and CI/CD to project management and testing, Azure DevOps provides everything a development team needs to deliver high-quality software. The first steps in using Azure DevOps involve setting up an account, creating a project, and configuring repositories, pipelines, and agile workflows.

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

Components required: Java (version jdk 17 or 23 and 21), Maven (Version 10), Gradle (Version 10). Jenkins, Azure account, Git

Theory:

1. Create an Azure DevOps Project

First, make sure you have an Azure DevOps account and a project set up. If you haven’t done that already:

- Go to Azure DevOps.
- Create a new project.

2. Integrate Your Code Repository (GitHub or Azure Repos)

Integrate GitHub Repo

- Go to your Azure DevOps project.
- Click on Pipelines in the left-hand menu.
- Click New Pipeline.
- Choose GitHub as the source and authorize Azure DevOps to access your GitHub repository.
- Select the repository where your project is located.

Integrate Azure Repos

- Go to your Azure DevOps project.
- Click on Pipelines.
- Click New Pipeline.
- Choose Azure Repos Git and select the repository.

3. Define Pipeline Configuration (YAML)

You can configure your pipeline using a YAML file, which is placed in the root of your repository. If you’re working with Maven or Gradle, the general structure of the YAML file will be different, but both follow a similar flow.

For Maven (Java) Project:

```
yaml
trigger:
  branches:
    include:
      - main # Trigger pipeline on main branch (or your default branch)
pool:
  vmImage: 'ubuntu-latest' # You can choose another image like 'windows-latest' or 'macos-latest'

steps:
```

```
- task: UseJavaToolInstaller@1
  inputs:
     versionSpec: '1.8' # You can specify the Java version you are using

- task: Maven@3
  inputs:
    mavenPomFile: 'pom.xml' # Path to your pom.xml file
    goals: 'clean install' # You can use other Maven goals as required
    options: '-X' # Optional: Additional Maven options

 - task: PublishTestResults@2
   inputs:
     testResultsFiles: '**/target/test-*.xml' # Path to the test reports
     testRunTitle: 'Maven Unit Tests'

- task: PublishBuildArtifacts@1
  inputs:
    pathToPublish: '$(Build.ArtifactStagingDirectory)' # Directory to publish
    artifactName: 'drop' # Artifact name
    publishLocation: 'Container'
```

For Gradle (Java) Project:

```
yaml
trigger:
    branches:
    include:
       - main
  pool:
    vmImage: 'ubuntu-latest'
 steps:
- task: UseJavaToolInstaller@1
  inputs:
     versionSpec: '1.8' # Your Java version

- task: Gradle@2
  inputs:
    gradleWrapperFile: './gradlew' # Path to Gradle wrapper (or use gradle if not using wrapper)
```

```
        options: 'clean build'
        workingDirectory: '$(Build.SourcesDirectory)' # Working directory for the build

  - task: PublishTestResults@2
    inputs:
      testResultsFiles: '**/build/test-*.xml' # Path to test results for Gradle
     testRunTitle: 'Gradle Unit Tests'

  - task: PublishBuildArtifacts@1
    inputs:
       pathToPublish: '$(Build.ArtifactStagingDirectory)'
       artifactName: 'drop'
       publishLocation: 'Container'
```

4. Run Unit Tests and Generate Reports

Maven Testing

Maven typically generates test reports in the target directory. The PublishTestResults@2 task picks up these reports and publishes them to Azure DevOps. You can also customize the location if needed.

Gradle Testing

Similarly, Gradle generates test results in the build directory. You can configure the pipeline to look for these test result files and publish them.

5. Publish Build Artifacts

Both Maven and Gradle projects can produce build artifacts (e.g., .jar, .war, .zip, or any other files). You can use the PublishBuildArtifacts@1 task to upload these artifacts as build outputs. These artifacts can later be used in release pipelines.

6. Configure Reporting and Test Results in Azure DevOps

Once the pipeline is triggered and unit tests are executed, Azure DevOps will show detailed test reports and status for each test case. The PublishTestResults@2 task ensures these results are uploaded for analysis.

You can also configure test reporting on the Azure DevOps dashboard for easy tracking of the test execution and coverage.

7. Trigger Pipeline

The pipeline will be triggered when changes are pushed to the repository. You can specify different branch conditions or even manual triggers. Here’s an example of how you can trigger the pipeline only when a change is pushed to the main branch:

```
yaml
trigger:
    branches:
    include:
```

```
            - main # Trigger only on changes to 'main' branch
```

8. Run and Monitor the Pipeline

Once you commit and push the YAML file to your repository, the pipeline will automatically trigger. You can monitor the build process from the Azure DevOps portal:

- Go to Pipelines.
- Click on the running pipeline to see the status, logs, and results.

NOTE:

- Azure Pipelines lets you automate the build, test, and deployment of your Maven or Gradle projects.
- The key tasks are setting up pipelines with correct tools (java , maven / gradle) runnubg unit tests, and generating test reports.

RESULT: Successfully Integrated our Java project with Azure Pipelines, running unit tests, and generating reports.