EXPERIMENT NO:1

Introduction to Maven and Gradle

AIM: Overview of Build Automation Tools, Key Differences Between Maven and Gradle, Installation and Setup

SOFTWARE/TOOLS REQUIRED: Java (version jdk 17 or 23), Maven (Version 10), Gradle (Version 10).

Theory:

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: 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.

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: 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.

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
Dependency Management	Uses Maven Central	Compatible with Maven too
Plugin Support	Large ecosystem	Extensible and versatile

Installation and Setup

Maven Installation:

1. **Download Maven:**
2. **Extract the ZIP File:**
3. **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
4. **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

- Click **OK** to close the windows and save your changes.

5. **Verify the Installation:**
   - Open Command Prompt and run: **mvn -v** If Maven is correctly installed, it will display the version number.

Input : mvn –v

Output : apache maven 3.9.9

Gradle Installation:

1. **Download Gradle:**
2. **Extract the ZIP File:**
3. **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
4. **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
   - Click **OK** to close the windows and save your changes.
5. **Verify the Installation:**
     - Open a terminal or Command Prompt and run: **gradle -v** If it shows the Gradle version, the setup is complete.

Input : gradle –v

Output : hello world

Result: Successfully installation and setup is done to build Maven and Gradle Tools

EXPERIMENT NO:02

Working with Maven

AIM: Creating a Maven Project, Understanding the POM File, Dependency Management and Plugins

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

Theory:

- **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.

Working with Maven

Step 1: Creating a maven project using command line

Create **program2** folder in desktop.

Open command prompt.

cd desktop

cd program2 – navigate to program 2 folder.

Mvn archetype:generate

DgroupId: com.example

ArtifactId: myapp

Package name: com.example.app

Version: enter

Y:enter

Step 2: Open The `pom.xml` File

- You can manually navigate the **project folder** named call
- 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.

❖ 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.

❖ 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.

❖ 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).

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**.

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

- Open a file App.java inside the src/main/java/com/example/ directory

To build and run this project

- **cd myapp**

1. **Compile the Project**
   - mvn compile

2. Run the Unit Test

- mvn test

3.Package the project into a JAR

- **mvn package**

4. Run the application (using JAR)

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

output: Hello world

Theory:

➢ **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.

Result : successfully created maven project and understood the pom file, dependency management and plugins

Experiment No: 3

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

COMPONENTS/ SOTWARE REQUIRED: Java (version jdk 17 or 23), Gradle (Version 10).

Theory:

1: 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**).**

Kotlin DSL: **Gradle also supports Kotlin for its build scripts (**build.gradle.kts**).**

2. 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.

3: Dependency Management

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

the dependencies block.

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.
  Example of running the build task:

1: Create a new gradle Project

- **Install Gradle**
- **Create a new Gradle project**:

To create a new Gradle project using the command line:

Create a folder (program3) in desktop
Open command line
cd desktop
cd program3
gradle init

- **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

Open program 3 folder and Manually navigate the folder path like **src/test/java/org/example/**

To **build** the project:

- **Gradle build**

To run the project

- **gradle run**

Output : Hello world

Working with Gradle Project (Kotlin DSL):

Step 1: Create a new Project

Create a folder (program33) in desktop

Open command line:

cd desktop

cd program33

gradle init

- 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

Open program 33 folder and Manually navigate the folder path like **src/test/java/org/example/**

To **build** the project:

- **Gradle build**

To run the project

gradle run

Output : Hello world

Result: Successfully completed the setting Up of a Gradle Project, Understanding Build Scripts and also Dependency Management and Task Automation.

EXPERIMENT NO: 04

Practical Exercise

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

COMPONENTS/SOFTWARE REQUIRED: Java (version jdk 17 or 23), Maven (Version 10), Gradle (Version 10).

Theory:

Step 1: Creating a Maven Project using command line

Create **program4** folder in desktop.

Open command prompt.

cd desktop

cd program2 – navigate to program 2 folder.

Mvn archetype:generate

DgroupId: com.example

ArtifactId: myapp

Package name: com.example.app

Version: enter

Y:enter

- Open a file App.java inside the src/main/java/com/example/ directory

Open Command prompt:

cd myapp

Mvn clean install

Step 2: Migrate the Maven Project to Gradle

Open command prompt:

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)? (de- fault: no) [yes, no]**
  - Type **No**

Step 3: Run the Gradle Project

Open Command prompt:

Gradlew build

Gradlew run

Step 4: Verify the Migration

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

Result: Successfully Build and Run a Java Application with Maven, Migrate the Same Application to Gradle.

EXPERIMENT NO:05

Introduction to Jenkins

AIM: Installing Jenkins on Local or Cloud Environment, Configuring Jenkins for First Use

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

Theory:

Introduction to 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 (Window):

1. **Prerequisites:**

- Ensure that J**ava (JDK) is installed** on your system.
- check if Java is installed by running **java -version** in the terminal.

2. **Install Jenkins on Window System):**
   - Download the Jenkins Windows
   - Run the installer and follow the on-screen instructions. While installing choose **login system**: **run service as LocalSystem (not recommended)**.

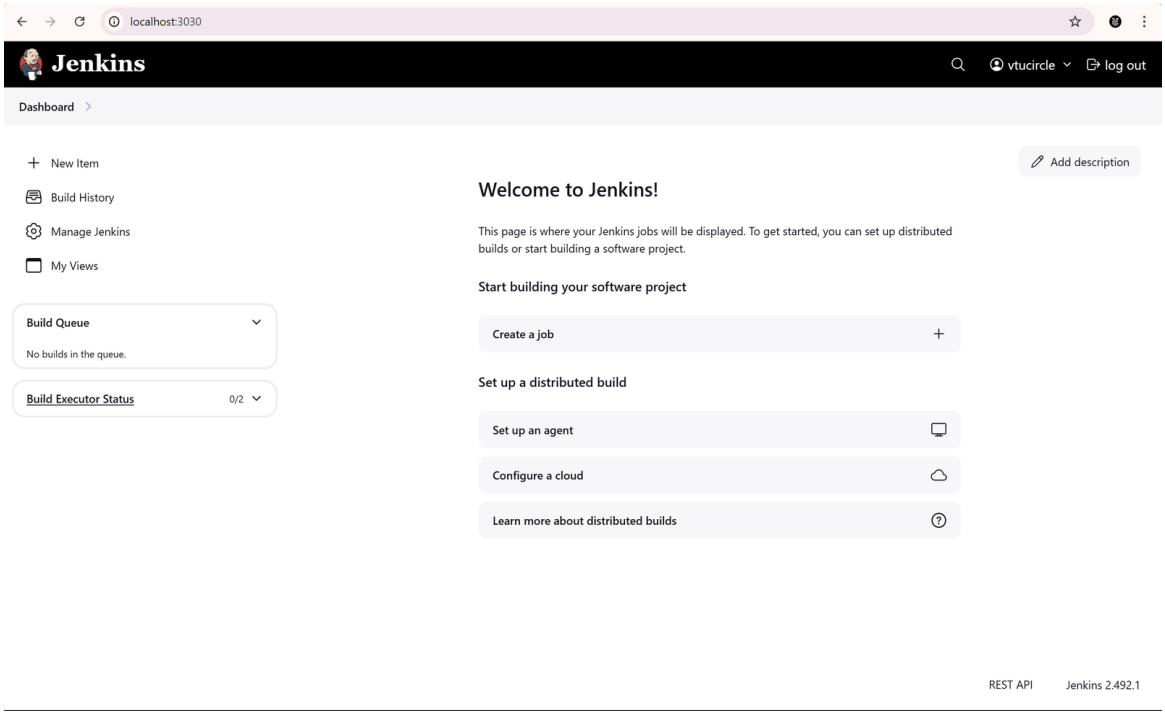
- After then use **default port** or you can **configure your own port 3030**
- After then **change the directory** and choose **java jdk-21** path look like **C:\Program Files\Java\jdk-21**.

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**

`http://localhost:3030`.

- It will ask administrator password, navigate the above highlighted path and open that `initial Admin Password` in notepad or any software to see the password.
- Just copy 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.



RESULT: Successfully installed Jenkins and configured Jenkins for the first use.

EXPERIMENT No:06

Continuous Integration with Jenkins

Aim: Setting Up a CI Pipeline, Integrating Jenkins with Maven/Gradle, Running Automated Builds and Tests

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

Theory: Sampling theorem for strictly band limited signals of finite energy into two equivalent parts:

Step 1: Install Jenkins:

1. Download Jenkins:

2. Prerequisites:

- Ensure that J**ava (JDK) is installed** on your system.
- check if Java is installed by running **java -version** in the terminal.

3. **Install Jenkins on Window System):**

- Download the Jenkins Windows
- 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 3030**
- After then **change the directory** and choose **java jdk-21** path look like **C:\Progra Files\Java\jdk-21**.

- After then **click next, next** and then it will **ask permission click on yes** and it will sta installing.
- After successfully installed, Jenkins will be **running on port either default port** or **chos port**

Search `http://localhost:3030`.

- It will ask administrator password, navigate the above highlighted path and open that `initial Admin Password` in notepad or any software to see the password.
- Just copy 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.

Step 3: Install Required Plugins

To integrate Jenkins with Maven or Gradle, you need to install the respective plugins:

1. Maven Plugin:

- Go to Manage Jenkins > Manage Plugins.
- In the Available tab, search for Maven Integration Plugin and install it.

2. Gradle Plugin:

- In the same Manage Plugins section, search for Gradle Plugin and install it.

3. JUnit Plugin:

- If you're using JUnit for automated testing, make sure the JUnit Plugin is installed for reporting purposes.

Step 4: Configure Jenkins for Maven/Gradle

Maven Integration:

1. Install Maven:

- Go to Manage Jenkins > Global Tool Configuration.
- Under Maven, click Add Maven and specify the Maven version or path to your Maven installation.

2. Configure Maven in Jenkins:

- In Jenkins project configuration, specify Maven as the build tool by selecting Invoke Maven under Build section.
- Provide the goals, such as `clean install` or `clean deploy`.

Gradle Integration:

1. Install Gradle:

- Similarly, install Gradle by going to Manage Jenkins > Global Tool Configuration.
- Under Gradle, click Add Gradle and specify the Gradle version or path to the installation.

2. Configure Gradle in Jenkins:

- In Jenkins project configuration, choose Invoke Gradle script under the Build section.

- Specify Gradle tasks like `clean build`.

Step 5: Create a Jenkins Pipeline/Job

1. Create a New Job:

- From the Jenkins dashboard, click on New Item.
- Choose Freestyle project or Pipeline depending on your needs.

2. Freestyle Project Configuration (Basic):

- Source Code Management: Set up Git or another source control system to fetch the code (e.g., GitHub, Bitbucket).
- Build Environment: Configure the environment, e.g., for Maven, you'll configure Maven goals (e.g., `clean install`).
- Build: In the Build section, you can specify the build commands, such as:
- For Maven: `mvn clean install`
- For Gradle: `gradle build`
- Post-build Actions: You can set up post-build actions to publish test results or deploy to servers.

3. Run Automated Tests:

-You can configure Jenkins to run tests automatically as part of the build.
-For JUnit, specify the test report location to view the results under Post-build Actions.
-You can configure Jenkins to run tests automatically as part of the build.
-For JUnit, specify the test report location to view the results under Post-build Actions.
-You can configure Jenkins to run tests automatically as part of the build.
-For JUnit, specify the test report location to view the results under Post-build Actions.
-configuration for Maven:
groovy
junit '/target/test-.xml'

OR

- Click on new item
- Item name : devops project
- Select freestyle project
- Select git and paste repository address fron github.
- Select poll SCM and enter H/5****
- Go to build steps- execute shell- enter mvn clean install

- Click on build now
- Check the status of the project in status

RESULT: successfully set up CI pipeline in Jenkins using Maven or Gradle. By integrating source control, build tools, and automated testing, you create a fully automated pipeline that ensures consistent and rapid development workflows.

EXPERIMENT No: 09

Introduction to Azure DevOps

AIM: 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 teams plan, develop, test, and deliver applications efficiently. Azure DevOps is designed to support both continuous integration (CI) and continuous delivery (CD), and it integrates seamlessly with various 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:

- 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 project is created, you can:

1. Manage Users: 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, 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.