devops

exp1

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**.

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.

- Uses **XML (pom.xml)** for configuration
- Follows a **standard project structure**
- Manages dependencies via **Maven Central**
- **Easy to learn**, but **less flexible**
- **Slower** builds

Gradle

Gradle is a more modern and versatile build automation 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.

- Uses **Groovy or Kotlin DSL** (simpler scripts)
- **Faster** due to caching and incremental builds
- Highly **customizable**
- Great for multi-language and multi-module projects
- Better for **CI/CD**

🔍 Quick Comparison

Feature	Maven	Gradle

Config Language	XML	Groovy/Kotlin
Speed	Slower	Faster
Flexibility	Less flexible	Highly flexible
Learning Curve	Easier	Steeper
Script Size	Verbose	Concise

🧰 Installing Maven

1. **Download**: Maven Downloads
2. **Extract ZIP** and move it to `C:\Program Files\`
3. **Copy path**: `...\apache-maven-x.x.x\bin`
4. **Set Path**:
   - Search "Environment Variables" → Edit → Add Maven `bin` path
5. **Verify**: Open CMD, run: `mvn -v`

⚙️ Installing Gradle

1. **Download**: Gradle Downloads
2. **Extract ZIP** and move to `C:\Program Files\`
3. **Copy path**: `...\gradle-x.x.x\bin`
4. **Set Path**:
   - Search "Environment Variables" → Edit → Add Gradle `bin` path
5. **Verify**: Run `gradle -v` in CMD

exp2

Working with a Maven Project

🔷 Step 1: Create a New Maven Project

Open Command Prompt:

```
mkdir program2
cd program2
mvn archetype:generate -DgroupId=com.example -DartifactId=myapp -Dar
chetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

> A folder named myapp will be created.

🔷 Step 2: Edit `pom.xml`

Go to the project folder and open `pom.xml`:

```
cd myapp
notepad pom.xml
```

Paste this inside:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.22.2</version>
    </plugin>
  </plugins>
</build>
```

🔷 Step 3: Add Java Code

Open and edit `App.java` in:

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

Paste:

```
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();
        System.out.println("2 + 3 = " + app.add(2, 3));
        System.out.println("Application executed successfully!");
    }
}
```

🔷 Step 4: Add Test Code

Open and edit `AppTest.java` in:

`src/test/java/com/example/AppTest.java`

Paste:

```
package com.example;

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

public class AppTest {
    @Test
    public void testAdd() {
        App app = new App();
        Assert.assertEquals(5, app.add(2, 3));
    }
```

```
}
```

🔷 Step 5: Build and Run

Make sure you're in the `myapp` folder, then run:

```
mvn compile      # Compile
mvn test         # Run unit tests
mvn package      # Package into JAR
java -cp target/myapp-1.0-SNAPSHOT.jar com.example.App  # Run app
```

exp3

☕ Working with Gradle (Groovy DSL)

🔷 Step 1: Create Project

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

When prompted, choose:

- Java version: `17`
- Project name: `groovyProject`
- Structure: `1 (Single app)`
- DSL: `2 (Groovy)`
- Test: `1 (JUnit 4)`
- New APIs: `no`

🔷 Step 2: `build.gradle`

Paste this:

```
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: Code – `AdditionOperation.java`

Replace file at `src/main/java/org/example/AdditionOperation.java`:

```
package com.example;

public class AdditionOperation {
    public static void main(String[] args) {
        double num1 = 5, num2 = 10;
        System.out.printf("The sum of %.2f and %.2f is %.2f%n", num1, num
2, num1 + num2);
    }
}
```

🔷 Step 4: Test – `AdditionOperationTest.java`

Replace file at `src/test/java/org/example/AdditionOperationTest.java`:

```
package com.example;

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

public class AdditionOperationTest {
    @Test
    public void testAddition() {
        assertEquals(15.0, 5 + 10, 0.01);
    }
}
```

🔷 Step 5: Run Commands

```
gradle build     # Build project
gradle run       # Run app
gradle test      # Run tests
```

🧁 Working with Gradle (Kotlin DSL)

🔷 Step 1: Create Project

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

Choose:

- Java version: `17`
- Project name: `kotlinProject`
- Structure: `1 (Single app)`
- DSL: `1 (Kotlin)`

- Test: `1 (JUnit 4)`
- New APIs: `no`

🔷 Step 2: `build.gradle.kts`

Paste this:

```
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.TestExceptionF
ormat.FULL
        showStandardStreams = true
    }
    outputs.upToDateWhen { false }
}

java {
    toolchain {
```

```
        languageVersion.set(JavaLanguageVersion.of(17))
    }
}
```

🔷 Step 3: Code – `Main.kt`

Create file at `src/main/java/org/example/Main.kt`:

```
package com.example

fun addNumbers(num1: Double, num2: Double): Double = num1 + num2

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

🔷 Step 4: Test – `MainTest.kt`

Create file at `src/test/java/org/example/MainTest.kt`:

```
package com.example

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

class MainTest {
    @Test
    fun testAddNumbers() {
        assertEquals(15.0, addNumbers(10.0, 5.0), 0.001)
    }
}
```

🔷 Step 5: Run Commands

```
gradle build     # Build project
gradle run       # Run app
gradle test      # Run tests
```

exp4

🛠️ Java App with Maven → Migrate to Gradle

✅ Step 1: Create Maven Project

Run this in terminal:

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

✅ Step 2: Edit `pom.xml`

Go to the folder:

```
cd maven-example
notepad pom.xml
```

Paste this:

```
<project ...>
  <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: Java Code – `App.java`

Go to: src/main/java/com/example/App.java

Paste:

```
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, 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;
    }
}
```

✅ Step 4: Run Maven Project

```
cd maven-example
mvn clean install
mvn exec:java -Dexec.mainClass="com.example.App"
```

Expected Output:

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

🔁 Migrate to Gradle

✅ Step 5: Initialize Gradle

From `maven-example` folder:

```
gradle init
```

Choose:

- Found Maven project → **yes**
- DSL → **2 (Groovy)**
- New APIs → **no**

✅ Step 6: Edit `build.gradle`

Open `build.gradle` and add:

```
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 7: Run Gradle Project

```
./gradlew build
./gradlew run
```

Expected Output:

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

🎯 Step 8: Verify Output

Compare outputs of **Maven** and **Gradle** – they should be the **same** ✅

exp5

🚀 Introduction to Jenkins

💡 What is Jenkins?

Jenkins is an open-source **automation server** used mainly for:

- **Continuous Integration (CI)** – auto-building and testing when code is committed.
- **Continuous Delivery (CD)** – automating deployment.

🔑 Key Features:

- **CI/CD** pipelines for smoother development.
- **Plugins** for Git, Maven, Gradle, Docker, and more.
- **Pipeline as Code** – define workflows using Groovy DSL or YAML.
- **Cross-platform** – runs on Windows, Linux, macOS, etc.

🖥️ Installing Jenkins on Windows (Local System)

✅ 1. Prerequisites

- Install **Java JDK 21**
- Confirm Java is installed:

```
java -version
```

✅ 2. Download & Install Jenkins

- Go to https://www.jenkins.io/download/
- Download **Windows Installer**
- Run installer:
  - **Run as LocalSystem** (for testing only)
  - Choose port (default `8080` or custom like `3030`)
  - Set JDK path: `C:\Program Files\Java\jdk-21\`

- Finish installation

👉 After install, Jenkins is accessible at:

http://localhost:8080
or
http://localhost:3030 (if custom)

🌐 First-Time Browser Configuration

✅ 1. Unlock Jenkins

- Open your browser and go to `http://localhost:3030`
- It will ask for an **administrator password**
  - Find it in: C:\Program Files\Jenkins\secrets\initialAdminPassword
  - Open with Notepad, copy, and paste into browser

✅ 2. Customize Jenkins

- Choose **"Install Suggested Plugins"**
- Create **admin user** (username, password, email, etc.)
- Click **Save and Continue**
- Finish with **Start using Jenkins**

exp6

CI with Jenkins: Complete Setup Guide

🎯 Objective:

Set up a **CI pipeline** using **Jenkins**, integrated with **Maven** or **Gradle**, and run **automated builds and tests**.

🧩 Required Components

- **Java JDK:** 17 / 21 / 23

- **Maven:** Version 10
- **Gradle:** Version 10
- **Jenkins** (latest LTS)

🛠️ Step 1: Install Jenkins

1. Download Jenkins from:

   👉 https://www.jenkins.io/download

2. Install Jenkins and open your browser:

   🔗 `http://localhost:8080` (or your chosen port)

🛡️ Step 2: Initial Configuration

1. **Unlock Jenkins**

   Copy password from:

   ```
   makefile
   Copy code
   C:\Program Files\Jenkins\secrets\initialAdminPassword
   ```

2. **Install Suggested Plugins**
3. **Create Admin User**
4. **Access Dashboard**

🧩 Step 3: Install Plugins

Go to **Manage Jenkins > Manage Plugins**

- ✔️ **Maven Integration Plugin**
- ✔️ **Gradle Plugin**
- ✔️ **JUnit Plugin**

🔧 Step 4: Configure Tools

🧱 Maven:

- Go to **Manage Jenkins > Global Tool Configuration**
- Under **Maven**:
  - Click **Add Maven**
  - Name: `Maven-10`
  - Option 1: Let Jenkins install it automatically
  - Option 2: Provide installed path manually

🧱 Gradle:

- Same as above under **Gradle**
  - Name: `Gradle-10`

🧪 Step 5: Create a Jenkins Job

Option A: Freestyle Project

- **New Item > Freestyle Project**
- **Source Code Management:** Git
- **Build Tool:**
  - For Maven:

    ```
    bash
    Copy code
    mvn clean install
    ```

  - For Gradle:

    ```
    bash
    Copy code
    gradle build
    ```

- **Post-build:** Add JUnit report location:

```
bash
Copy code
target/surefire-reports/*.xml
```

Option B: Pipeline Project

- **New Item > Pipeline**
- **Use Jenkinsfile**

🔧 Example Jenkinsfile for Maven:

```
groovy
Copy code
pipeline {
    agent any
    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/your-repository.git'
            }
        }
        stage('Build') {
            steps {
                sh 'mvn clean install'
            }
        }
        stage('Test') {
            steps {
                sh 'mvn test'
            }
        }
    }
    post {
        always {
            junit 'target/surefire-reports/*.xml'
        }
```

```
    }
}
```

🔧 Example Jenkinsfile for Gradle:

```
groovy
Copy code
pipeline {
    agent any
    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/your-repository.git'
            }
        }
        stage('Build') {
            steps {
                sh './gradlew clean build'
            }
        }
        stage('Test') {
            steps {
                sh './gradlew test'
            }
        }
    }
    post {
        always {
            junit 'build/test-results/test/*.xml'
        }
    }
}
```

🔁 Step 6: Triggering Builds

🔄 Automatic Trigger

- Enable under **Build Triggers**:
  - GitHub hook trigger
  - Poll SCM (`H/5 * * * *` for every 5 min)

🖱️ Manual Trigger

- Just click **Build Now** from the dashboard.

📊 Step 7: Monitor Results

- **Build History** shows all builds and logs.
- **Test Reports** help you debug failed tests via JUnit plugin.
- **Console Output** shows real-time logs.

🚀 Step 8: (Optional) Deployment

- Add a new **"Deploy" stage** in your Jenkinsfile.
- Use `scp`, `rsync`, `Docker`, or a **cloud deployment command**.
- Can also use **Post-build Actions** in Freestyle jobs.

✅ RESULT

By completing these steps, you'll have:

- Fully automated **CI pipeline** with build + test
- **Maven or Gradle** integration
- Optionally **auto-triggered** and even **auto-deployed**

exp7

Configuration Management with Ansible

Aim:

Understand the basics of Ansible — inventory, playbooks, and modules — and automate server configurations using playbooks.

Components Required:

- Java (JDK 17, 21, or 23)
- Ansible

Theory:

Ansible is a powerful, agentless automation tool used for IT orchestration and configuration management. It uses simple **YAML-based playbooks** to define tasks.

---

1. Inventory:

An inventory defines the target hosts that Ansible manages.

INI Format Example:

```
ini
Copy code
[web_servers]
192.168.1.10
192.168.1.11
```

YAML Format Example:

```
yaml
Copy code
all:
  children:
    web_servers:
      hosts:
        192.168.1.10:
        192.168.1.11:
```

---

2. Playbooks:

Playbooks define **what tasks** to perform on **which hosts** using **which modules**.

Basic Playbook Structure:

```
yaml
Copy code
---
- 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:

Modules perform specific tasks (e.g., install packages, manage files/services).

Common Modules:

- `apt` (Debian-based package manager)
- `yum` (RHEL-based)
- `service`
- `copy`
- `file`

---

4. Automating Server Configurations with Playbooks

Example: Configure Apache Web Server

```
yaml
Copy code
---
- name: Configure web server
  hosts: web_servers
  become: yes
  tasks:
    - name: Install Apache2
      apt:
        name: apache2
        state: present

    - name: Copy custom Apache config
      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
```

Hands-On: Writing and Running a Basic Playbook

Step 1: Create Inventory File

`hosts.ini`

```
ini
Copy code
[web_servers]
```

```
192.168.1.10
192.168.1.11
```

Step 2: Write the Playbook

install_apache.yml

```
yaml
Copy code
---
- name: Install and configure Apache web server
  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

    - name: Ensure homepage is present
      copy:
        content: "<html><body><h1>Welcome to the Apache Server!</h1></b
ody></html>"
        dest: /var/www/html/index.html
        mode: '0644'
```

Step 3: Run the Playbook

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

Step 4: Verify

1. Visit `http://192.168.1.10` or `http://192.168.1.11` in a browser.
2. Use SSH to verify Apache status:

```
bash
Copy code
systemctl status apache2
```

Result:

Successfully written and executed a basic Ansible playbook to install and configure Apache on remote servers.

exp8

Configuration Management with Ansible

Set up a Jenkins CI pipeline for a Maven project and use Ansible to deploy artifacts generated by Jenkins.

🛠️ Components Required

- Java JDK (17, 21, or 23)
- Maven (v10)
- Ansible
- Jenkins

📘 Theory Overview

We automate the build (using Maven via Jenkins) and deployment (using Ansible) processes to create a complete CI/CD pipeline.

🔧 Step-by-Step Guide

Step 1: Set Up Jenkins for CI

1. **Install Jenkins**

   Follow: https://www.jenkins.io/download/

2. **Install Required Jenkins Plugins**
   - Maven Integration Plugin
   - Ansible Plugin
   - Git Plugin
3. **Configure Global Tools**
   - Go to: **Manage Jenkins > Global Tool Configuration**
   - Set up JDK, Maven, and Ansible (provide installation path if needed)
4. **Ensure Ansible is Installed**
   - Must be installed on Jenkins host
   - Set path in Global Tool Config

Step 2: Create Maven Project in Jenkins

1. **Create New Freestyle Project**
   - Name: `maven-build-project`
2. **Configure Git Repository**
   - Source Code Management → Git
   - Provide repo URL and credentials if private
3. **Add Maven Build Step**
   - **Build → Invoke top-level Maven targets**
     - **Goals:** `clean install`
     - **Maven Version:** select configured one

4. **Post-build Action**
   - **Archive the artifacts**
     - Files: `target/*.jar`

Step 3: Jenkins CI Pipeline (Optional: Pipeline Script)

If using a **Pipeline Project**, use the below `Jenkinsfile`:

```
groovy
Copy code
pipeline {
    agent any
    tools {
        maven 'Maven-3.6.3'
    }
    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'
            }
        }
        stage('Archive Artifacts') {
            steps {
                archiveArtifacts artifacts: 'target/*.jar', allowEmptyArchive: true
            }
        }
        stage('Deploy') {
            steps {
                ansiblePlaybook playbook: 'deploy.yml', inventory: 'inventory.ini'
            }
```

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

Step 4: Ansible Deployment

1. Ansible Playbook (`deploy.yml`):

```
yaml
Copy code
---
- name: Deploy artifact to server
  hosts: web_servers
  become: yes
  tasks:
    - name: Copy the artifact to the remote server
      copy:
        src: /path/to/jenkins/workspace/target/my-app.jar
        dest: /opt/myapp/my-app.jar

    - name: Restart the application service
      systemd:
        name: myapp
        state: restarted
        enabled: yes

    - name: Check if application is running
      uri:
        url: "http://localhost:8080"
```

```
    status_code: 200
```

2. Ansible Inventory File (`inventory.ini`):

```
ini
Copy code
[web_servers]
192.168.1.10
192.168.1.11
```

3. Configure Ansible in Jenkins

- Jenkins > Global Tool Configuration
- Set path to Ansible
- Make sure Jenkins has access to inventory and playbook files

---

Step 5: Run the Pipeline

Jenkins will:

1. Clone Git repo
2. Build using Maven
3. Archive `.jar` file
4. Deploy using Ansible

✅ RESULT

A fully functional CI/CD pipeline using Jenkins and Ansible for Maven-based applications is now set up. It automates building and deploying applications, improving efficiency and consistency in delivery.

exp9

Introduction to Azure DevOps

AIM:

To understand Azure DevOps Services and set up an Azure DevOps account and project.

COMPONENTS REQUIRED:

- Java (JDK 17, 21, or 23)
- Maven (v10)
- Gradle (v10)
- Jenkins

THEORY:

Azure DevOps is a cloud-based platform provided by Microsoft that enables teams to manage the **entire software development lifecycle (SDLC)** using a collection of integrated services.

Overview of Azure DevOps Services

1. **Azure Repos**
   - Provides Git or TFVC for source control.
   - Supports collaboration, version tracking, and code history management.
2. **Azure Pipelines**
   - Enables Continuous Integration and Continuous Delivery (CI/CD).
   - Automates building, testing, and deploying code across environments.
3. **Azure Boards**
   - Agile project management tools (Kanban, Scrum).
   - Supports backlogs, sprints, tasks, and issue tracking.
4. **Azure Test Plans**
   - Manual and exploratory testing tools.
   - Helps track bugs, manage test cases, and maintain code quality.

5. **Azure Artifacts**
    - Manage and share build dependencies like **NuGet**, **npm**, and **Maven** packages.
6. **Collaboration Tools**
    - Dashboards, Wikis, and team notifications to improve visibility and collaboration.

Setting Up an Azure DevOps Account

1. **Create an Account**
    - Visit: https://dev.azure.com/
    - Sign in with an existing Microsoft account or create a new one.
    - Choose a unique organization name and region.
2. **Create a Project**
    - Click **New Project** on the dashboard.
    - Provide a project name.
    - Choose:
        - **Visibility**: Private or Public
        - **Version Control**: Git or TFVC
        - **Process Template**: Agile, Scrum, or CMMI
    - Click **Create** to initialize the project.

Project Organization and Access Management

1. **Manage Users**
    - Add team members, assign roles, and set permissions across services.
2. **Repositories**
    - Create and manage Git repositories.
    - Track branches and enable collaborative coding.
3. **Pipelines**
    - Define CI/CD workflows to automate the build-test-deploy cycle.

4. **Boards**
    - Organize tasks, user stories, and work items into sprints or iterations.

RESULT:

Azure DevOps offers a complete suite for modern DevOps workflows—version control, CI/CD, project tracking, and testing. By creating an account and setting up a project, users take the first step toward fully automated and agile software delivery.

exp10

Creating Build Pipelines

AIM:

To build a Maven or Gradle-based Java project using Azure Pipelines, integrate code repositories (GitHub/Azure Repos), run unit tests, and generate reports.

COMPONENTS REQUIRED:

- Java (JDK 17, 21, or 23)
- Maven (v10)
- Gradle (v10)
- Jenkins
- Azure DevOps Account
- Git

THEORY:

1. Create an Azure DevOps Project

- Go to https://dev.azure.com
- Create or log in to your organization and set up a new project.

---

2. Integrate Code Repository

GitHub Integration:

- Navigate to your Azure DevOps project > **Pipelines** > **New Pipeline**
- Select **GitHub**, authorize access, and choose the repository.

Azure Repos Integration:

- Go to **Pipelines** > **New Pipeline**
- Select **Azure Repos Git** and pick the repository.

---

3. Define Pipeline with YAML

For Maven Projects:

```
yaml
Copy code
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

steps:
  - task: UseJavaToolInstaller@1
    inputs:
      versionSpec: '1.8'

  - task: Maven@3
    inputs:
      mavenPomFile: 'pom.xml'
      goals: 'clean install'
      options: '-X'

  - task: PublishTestResults@2
    inputs:
```

```
      testResultsFiles: '**/target/test-*.xml'
      testRunTitle: 'Maven Unit Tests'

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

For Gradle Projects:

```
yaml
Copy code
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

steps:
  - task: UseJavaToolInstaller@1
    inputs:
      versionSpec: '1.8'

  - task: Gradle@2
    inputs:
      gradleWrapperFile: './gradlew'
      options: 'clean build'
      workingDirectory: '$(Build.SourcesDirectory)'

  - task: PublishTestResults@2
    inputs:
      testResultsFiles: '**/build/test-*.xml'
      testRunTitle: 'Gradle Unit Tests'
```

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

---

4. Run Unit Tests and Generate Reports

- **Maven:** Reports are generated in `target/` and picked up via `PublishTestResults@2`.
- **Gradle:** Reports are in `build/` and similarly published using `PublishTestResults@2`.

---

5. Publish Build Artifacts

- The task `PublishBuildArtifacts@1` uploads `.jar`, `.war`, or `.zip` files for use in release pipelines.

---

6. Configure Reporting and Test Results

- Test results are viewable in Azure DevOps under the **Tests** tab of the pipeline run.
- Dashboards can be configured to show test trends, pass/fail stats, etc.

---

7. Triggering the Pipeline

- Pipelines trigger automatically on changes to the specified branch (`main`).
- Manual triggers or custom conditions can also be configured.

---

8. Run and Monitor the Pipeline

- After committing the `.yaml` file, the pipeline runs automatically.
- Go to **Pipelines** > select your pipeline to view logs, test results, and build artifacts.

RESULT:

Successfully built and tested a Java Maven/Gradle project using Azure Pipelines. Integrated source control, executed unit tests, and published test

results and build artifacts.