# Appendix

## Pre-requisites and Common Setup

Before starting any experiment, ensure you have the following installed and configured on your Ubuntu system:

1. **Java Development Kit (JDK):**
   Install OpenJDK (version 11 or above):

   ```
   sudo apt update
   sudo apt install openjdk-11-jdk
   ```

   Verify installation:

   ```
   java -version
   ```

2. **Git (for version control):**

   ```
   sudo apt install git
   ```

   Configure Git (replace with your details):

   ```
   git config --global user.name "Your Name"
   git config --global user.email "your.email@example.com"
   ```

3. **Maven:**

   ```
   sudo apt install maven
   ```

   Verify installation:

   ```
   mvn -version
   ```

4. **Gradle:**

   ```
   sudo apt install gradle
   ```

   Verify installation:

```
gradle -v
```

5. **Jenkins:**

   Follow these steps to install Jenkins:

```
wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -
sudo sh -c 'echo deb https://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'
sudo apt update
sudo apt install jenkins
sudo systemctl start jenkins
sudo systemctl status jenkins
```

   Open Jenkins in your browser at http://localhost:8080.

6. **Ansible:**

```
sudo apt update
sudo apt install ansible
```

   Verify installation:

```
ansible --version
```

7. **Azure DevOps Account:**

   Create an account at Azure DevOps and set up your first project. (Steps for Azure DevOps will be covered in later experiments.)

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

## 1. Understanding DevOps

### What is DevOps?

**DevOps** is a set of cultural philosophies, practices, and tools that combine software development (Dev) and IT operations (Ops). Its goal is to shorten the systems development life cycle while delivering features, fixes, and updates frequently in close alignment with business objectives. DevOps promotes:

- **Collaboration:** Breaking down silos between development and operations teams.
- **Automation:** Automating repetitive tasks (builds, tests, deployments) to improve efficiency.
- **Continuous Integration/Delivery (CI/CD):** Regularly integrating code changes and deploying them to production with minimal manual intervention.
- **Monitoring and Feedback:** Constantly monitoring the performance and behavior of applications in production to rapidly address issues.

### Why is DevOps Used?

- **Speed and Agility:** Faster development cycles and quicker time-to-market.
- **Quality:** Automated testing and integration help catch issues early.
- **Reliability:** Frequent, smaller updates reduce the risk of large-scale failures.
- **Efficiency:** Automation reduces manual errors and repetitive tasks.
- **Scalability:** Processes and infrastructure can grow with the business.

### Examples of DevOps in Action:

- **Continuous Integration (CI):** Tools like Jenkins or Azure Pipelines automatically build and test code on every commit.
- **Continuous Deployment (CD):** Systems automatically deploy tested code to production environments.
- **Monitoring:** Tools such as Prometheus, Grafana, or New Relic continuously monitor application performance.

## 2. Introduction to Maven and Gradle

### What is Maven?

**Maven** is a build automation and project management tool primarily used for Java projects. It uses a central configuration file known as the **POM (Project Object Model)**, written in XML, which defines the project structure, its dependencies, build order, and plugins.

**Key Features and Roles of Maven:**

- **Convention over Configuration:** Maven enforces a standard directory structure (e.g., `src/main/java`, `src/test/java`), which simplifies project setup.
- **Dependency Management:** Automatically downloads and manages external libraries and dependencies from repositories like Maven Central.
- **Build Lifecycle:** Defines a fixed lifecycle (e.g., compile, test, package, install, deploy) which standardizes the build process.
- **Plugin Ecosystem:** Provides a rich set of plugins to extend functionality (e.g., unit testing, code coverage, reporting).

## What is Gradle?

**Gradle** is a modern build automation tool that is known for its flexibility and performance. It uses a **Groovy** or **Kotlin DSL (Domain Specific Language)** to define build logic, which allows for more dynamic and customizable configurations compared to Maven's XML-based approach.

**Key Features and Roles of Gradle:**

- **Flexible Build Scripts:** Instead of a rigid XML file, Gradle build scripts written in Groovy or Kotlin allow you to include logic, conditionals, and loops.
- **Incremental Builds:** Gradle tracks changes in source files and only rebuilds what is necessary, which can significantly speed up the build process.
- **Multi-project Builds:** Easily manages complex projects with multiple modules or subprojects.
- **Extensibility:** A robust plugin system that enables you to integrate various languages, frameworks, and tools.
- **Parallel Execution:** Can run tasks in parallel, optimizing build times for large projects.

## 3. Why Use Maven and Gradle?

Both Maven and Gradle are used to automate the build process and manage project dependencies. Here's why they are so popular in the DevOps ecosystem:

- **Automated Build and Testing:** They allow developers to compile code, run tests, and package applications without manual intervention.
- **Consistent Build Environment:** By enforcing standardized project structures and dependency management, they help avoid the "it works on my machine" problem.
- **Ease of Integration:** Both tools integrate well with Continuous Integration (CI) servers like Jenkins, enabling automated pipelines.
- **Dependency Resolution:** They simplify the process of managing external libraries and ensure that all developers are using the same versions.

## 4. Installing Maven and Gradle on Ubuntu

## A. Installing Maven

**Step 1: Update Your System**

Open a terminal and run:

```
sudo apt update
sudo apt upgrade
```

**Step 2: Install Java (if not already installed)**

Maven requires Java. Install OpenJDK (for example, version 11):

```
sudo apt install openjdk-11-jdk
java -version
```

*Expected Output (example):*

```
openjdk version "11.0.11" 2021-04-20
OpenJDK Runtime Environment (build 11.0.11+9-Ubuntu-0ubuntu2.20.04)
OpenJDK 64-Bit Server VM (build 11.0.11+9-Ubuntu-0ubuntu2.20.04, mixed mode,
sharing)
```

**Step 3: Install Maven**

Run the following command:

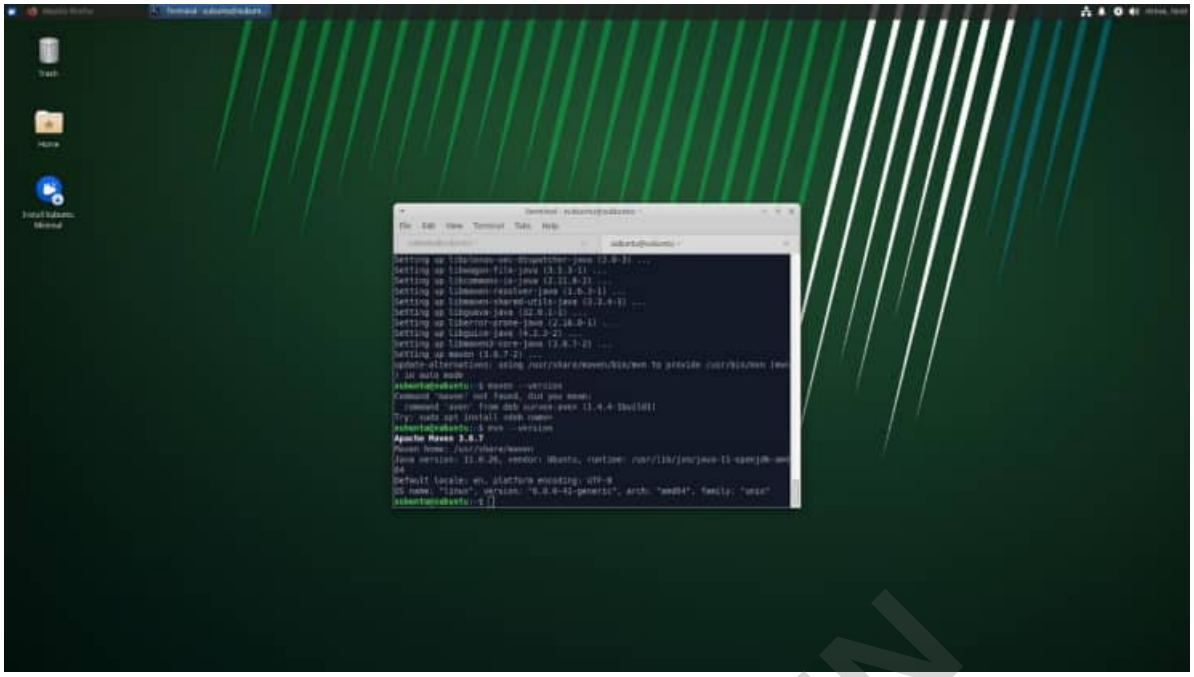
```
sudo apt install maven
```

After installation, check the Maven version:

```
mvn -version
```

*Expected Output (example):*

```
Apache Maven 3.6.3
Maven home: /usr/share/maven
Java version: 11.0.11, vendor: Ubuntu, runtime: /usr/lib/jvm/java-11-
openjdk-amd64
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "5.4.0-xx-generic", arch: "amd64", family: "unix"
```



## B. Installing Gradle

Gradle can be installed in two primary ways: via the Ubuntu repositories (which may not be the latest version) or by manually installing the latest version.

**Option 1: Install from Ubuntu Repositories**

```
sudo apt install gradle
gradle -v
```

*Screenshot Tip:* Capture the terminal output of `gradle -v`.

**Option 2: Install the Latest Version Manually (Recommended)**

1. **Download Gradle:**

   ```
   wget https://services.gradle.org/distributions/gradle-8.0-bin.zip
   ```

2. **Unzip the Downloaded File:**

   ```
   sudo unzip -d /opt/gradle gradle-8.0-bin.zip
   ```

3. **Set Up the Environment Variables:** Add Gradle to your system PATH by appending the following line to your `~/.bashrc` (or `~/.profile`):

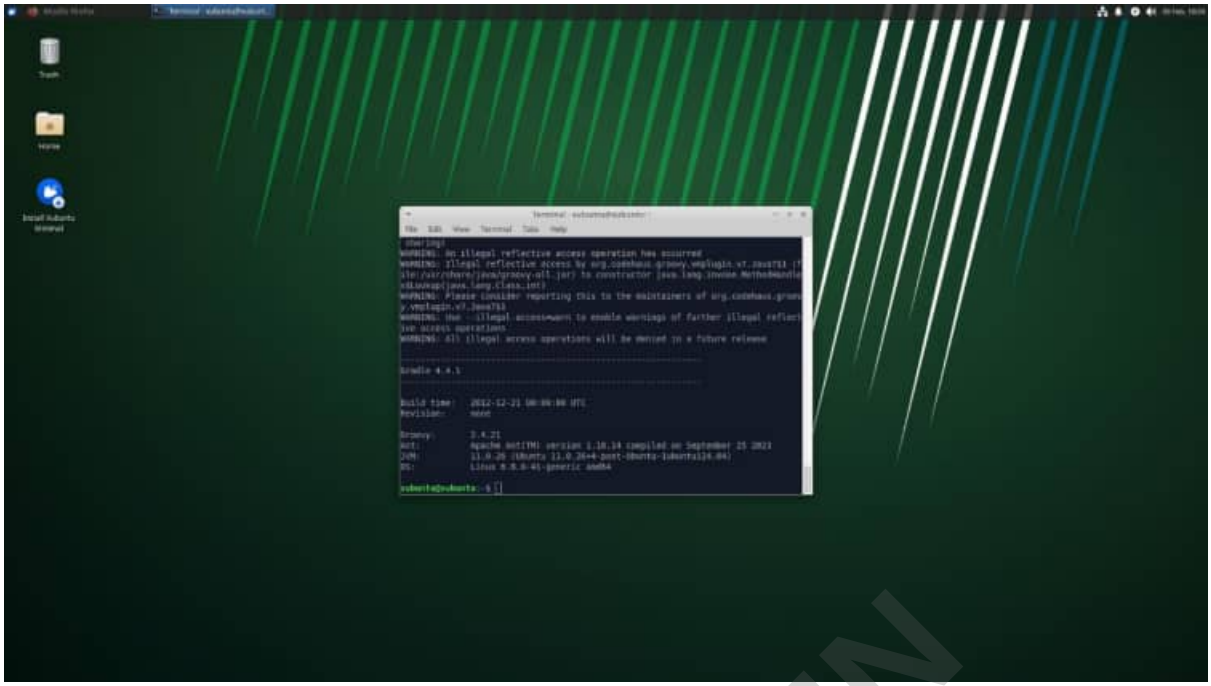
```
echo "export PATH=\$PATH:/opt/gradle/gradle-8.0/bin" >> ~/.bashrc
source ~/.bashrc
```

4. **Verify the Installation:**

```
gradle -v
```

*Expected Output (example):*

```
------------------------------------------------------------
Gradle 8.0
------------------------------------------------------------
Build time:   2022-05-03 12:00:00 UTC
Revision:     <revision-hash>
Kotlin:       1.6.x
Groovy:       3.0.x
Ant:          Apache Ant(TM) version 1.10.x compiled on ...
JVM:          11.0.11 (Ubuntu 11.0.11+9-Ubuntu-0ubuntu2.20.04)
OS:           Linux 5.4.0-xx-generic amd64
```



**5. Detailed Differences Between Maven and Gradle**

Below is a detailed comparison highlighting their main differences:

| Aspect | Maven | Gradle |
|---|---|---|
| **Build Script Language** | Uses an XML-based configuration file (pom.xml). | Uses a DSL based on Groovy or Kotlin (build.gradle or build.gradle.kts). |
| **Configuration Style** | Declarative and rigid – follows strict conventions (convention over configuration). | Flexible and dynamic – allows you to write custom logic and conditions within the build script. |
| **Build Lifecycle** | Provides a fixed lifecycle (e.g., validate, compile, test, package, install, deploy). | Uses a task-based approach where tasks can be defined, customized, and linked in a flexible manner. |
| **Dependency Management** | Manages dependencies through the POM file; downloads them from central repositories (e.g., Maven Central). | Similar dependency management; supports dynamic version resolution and customizable dependency configurations. |

| | | |
|---|---|---|
| **Performance** | Generally slower for large projects because of the fixed build lifecycle and less emphasis on incremental builds. | Often faster, thanks to incremental builds, caching, and parallel execution of tasks. |
| **Extensibility & Plugins** | Rich ecosystem of plugins but customization can be more challenging due to XML's verbosity and limitations in scripting. | Highly extensible through its scripting capabilities; writing custom tasks or plugins in Groovy/Kotlin is more straightforward. |
| **Multi-Project Builds** | Handles multi-module projects well but requires a strict directory layout and a parent POM for aggregating modules. | Excels in multi-project builds with simple configuration, allowing each subproject to be configured in a flexible manner. |
| **Learning Curve** | Easier for beginners due to its structured, convention-based approach but can become complex with large configurations. | More flexible but may have a steeper learning curve initially if you need to leverage its dynamic features and custom logic. |
| **Community & Maturity** | Has been around longer, so many legacy projects and extensive documentation exist. | Relatively newer; it has gained popularity due to its modern features, especially in Android and multi-language projects. |
| **Integration with CI/CD** | Well-supported by most CI/CD tools (like Jenkins, Azure Pipelines) with stable, predictable behavior. | Also integrates seamlessly with CI/CD tools and is often chosen for its faster build times and flexibility in pipeline scripting. |

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

## 1. Introduction to Maven

**Maven** is a powerful build automation and project management tool primarily used for Java projects. It simplifies the build process by:

- Enforcing a **standard project structure** (convention over configuration).
- Managing **dependencies** automatically by downloading them from remote repositories (e.g., Maven Central).
- Defining a clear **build lifecycle** (compile, test, package, install, deploy).
- Allowing the integration of various **plugins** to extend functionality (e.g., testing, reporting).

## 2. Creating a Maven Project

### Step-by-Step Process

#### Step 1: Open Your Terminal

Make sure you have Maven installed (refer to Experiment 1). Open your terminal on your Ubuntu system.

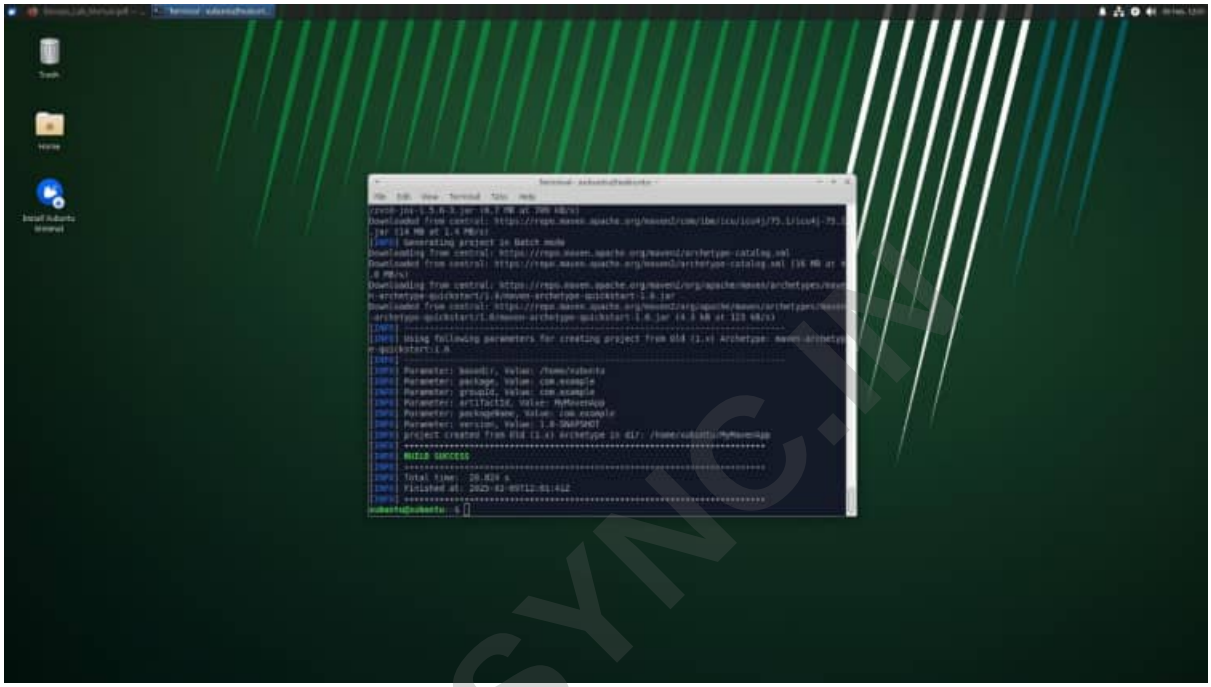
#### Step 2: Use Maven Archetype to Generate a New Project

Maven comes with a set of archetypes that provide you with a standard project template. Use the following command to create a new Maven project:

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

- **groupId**: Uniquely identifies your project's group (like a package name).
- **artifactId**: The name of your project (the resulting artifact).

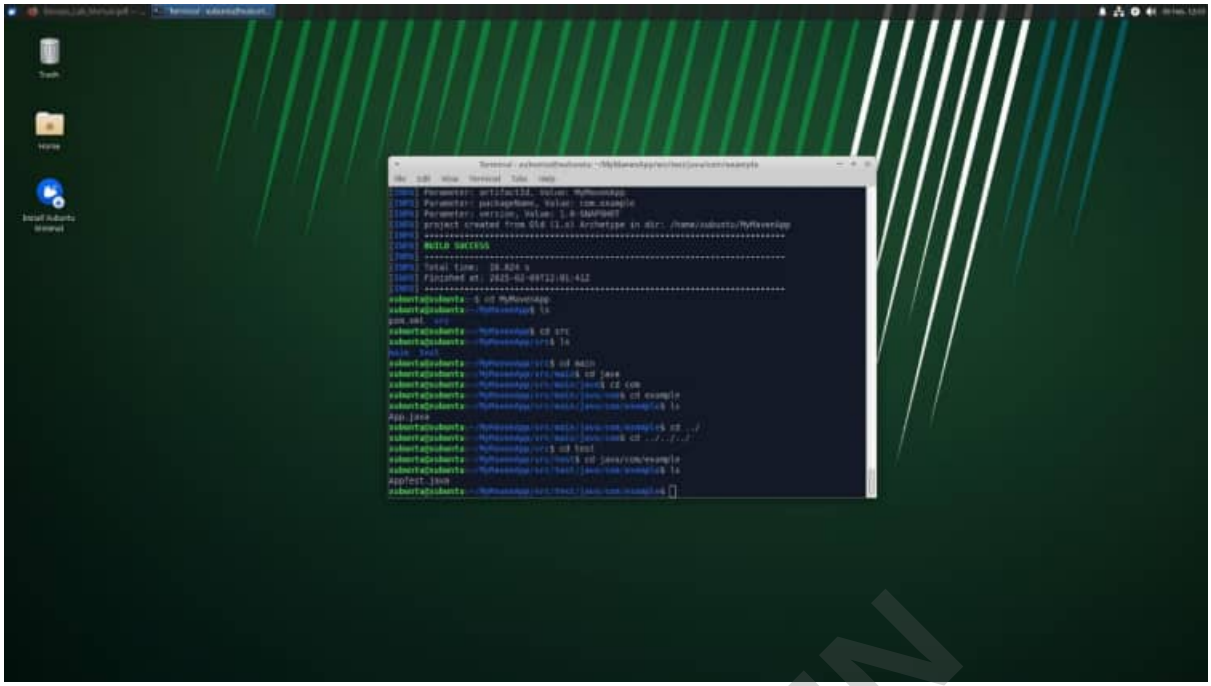
- **maven-archetype-quickstart**: A simple archetype that sets up a basic Java project with a sample unit test.
- **-DinteractiveMode=false**: Runs the command in non-interactive mode, using the provided parameters.



**Step 3: Navigate to Your Project Directory**

Once the command completes successfully, change your directory to the newly created project:

```
cd MyMavenApp
```





## 3. Maven Project Layout and Components

After generating the project, you will notice the following standard Maven directory structure:

```
MyMavenApp/
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── com
    │           └── example
    │               └── App.java
    └── test
        └── java
            └── com
                └── example
                    └── AppTest.java
```

### Explanation of Key Components

- **pom.xml:**

  The **Project Object Model (POM)** file is the core of any Maven project. It contains configuration details such as project coordinates (groupId, artifactId, version), dependencies, plugins, and build settings.
- **src/main/java:**

  This directory holds the source code of your application. In our example, the package structure `com.example` is created, and you have an `App.java` file.
- **src/test/java:**

  This directory is for your test cases. The default example includes a basic test class, `AppTest.java`.

## 4. Understanding the POM File (pom.xml)

The `pom.xml` file is written in XML and is essential to how Maven operates. It includes several key sections:

### Basic Structure of a POM File

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

  <!-- Project Coordinates -->
  <groupId>com.example</groupId>
  <artifactId>MyMavenApp</artifactId>
  <version>1.0-SNAPSHOT</version>

  <!-- Properties: Customize Java version or plugin versions -->
  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
  </properties>

  <!-- Dependencies -->
  <dependencies>
    <!-- Example: JUnit dependency for testing -->
```

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

  <!-- Build: Configuring plugins and build settings -->
  <build>
    <plugins>
      <!-- Example: Maven Compiler Plugin to compile Java code -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>

      <!-- Example: Maven Surefire Plugin to run tests -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.22.2</version>
      </plugin>
    </plugins>
  </build>
</project>
```



## Key Elements Explained

- **Project Coordinates:**
  - `<groupId>`: Acts like a namespace, usually following the reverse domain name convention.
  - `<artifactId>`: The name of the project.
  - `<version>`: The current version of the project.
- **Properties:**

    - Used to define values that can be referenced elsewhere in the POM (e.g., Java source and target versions).
- **Dependencies:**
    - **Dependency Management:** Maven downloads and manages external libraries. In the example, JUnit is added for testing.
    - **Scope:** Determines when a dependency is used (e.g., `compile`, `test`, `runtime`).
- **Build and Plugins:**
    - **Maven Compiler Plugin:** Ensures that your Java code is compiled with the specified Java version.
    - **Maven Surefire Plugin:** Executes unit tests during the build process.



## 5. Dependency Management with Maven

### How Maven Manages Dependencies

- **Automatic Download:** When you specify a dependency in the `<dependencies>` section, Maven automatically downloads the library from a remote repository (usually Maven Central).
- **Transitive Dependencies:** Maven also resolves dependencies required by your dependencies.
- **Version Control:** You can specify precise versions for each dependency, ensuring consistency across builds.

### Example Dependency

To add a dependency for JUnit, include the following snippet in your `<dependencies>` section:

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

- **groupId, artifactId, version:** These three elements uniquely identify the dependency.

- **scope:** The `test` scope ensures that this dependency is only available during the test phase and not included in the final artifact.

## 6. Maven Plugins: Extending Maven Functionality

Plugins are key to Maven's flexibility, adding tasks to your build process.

### Example 1: Maven Compiler Plugin

This plugin is used to compile your Java source code.

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

- **Configuration:** Specifies that the project should be compiled using Java 11.

### Example 2: Maven Surefire Plugin

This plugin runs your unit tests during the `test` phase.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.22.2</version>
</plugin>
```

- **Purpose:** Automatically detects and runs tests in the `src/test/java` directory.

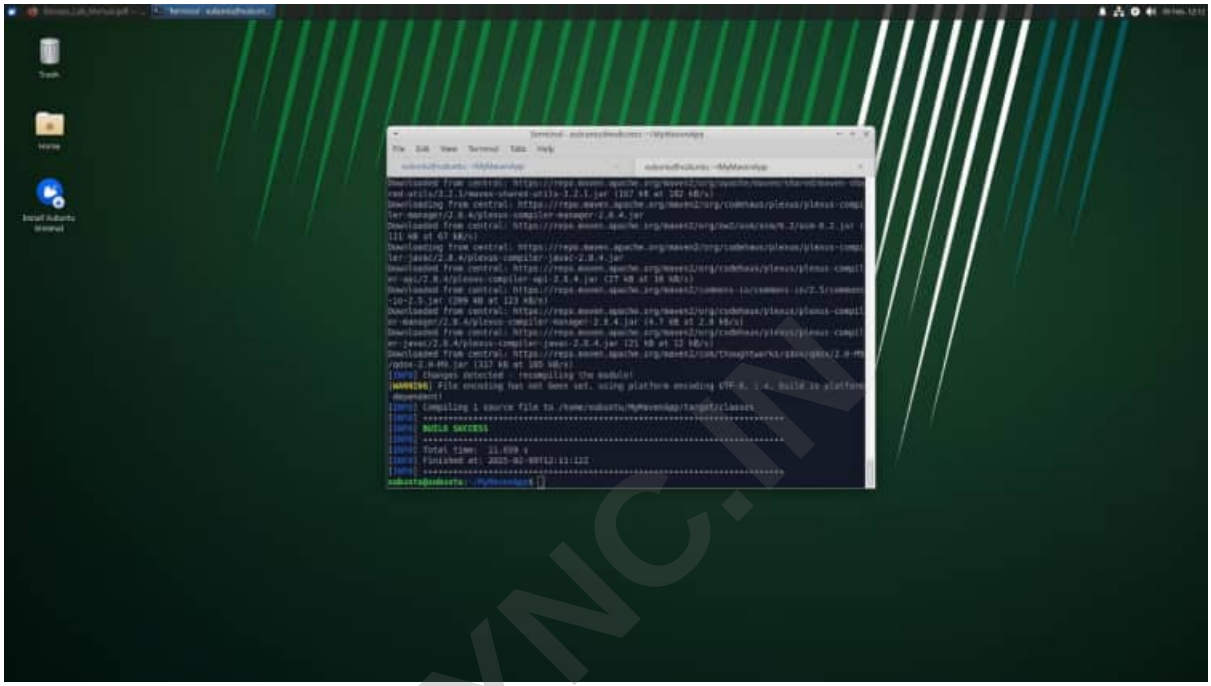
## 7. Building and Testing Your Maven Project

Once your project is set up and your `pom.xml` is defined, you can use Maven commands to build and test your application.

## Common Maven Commands

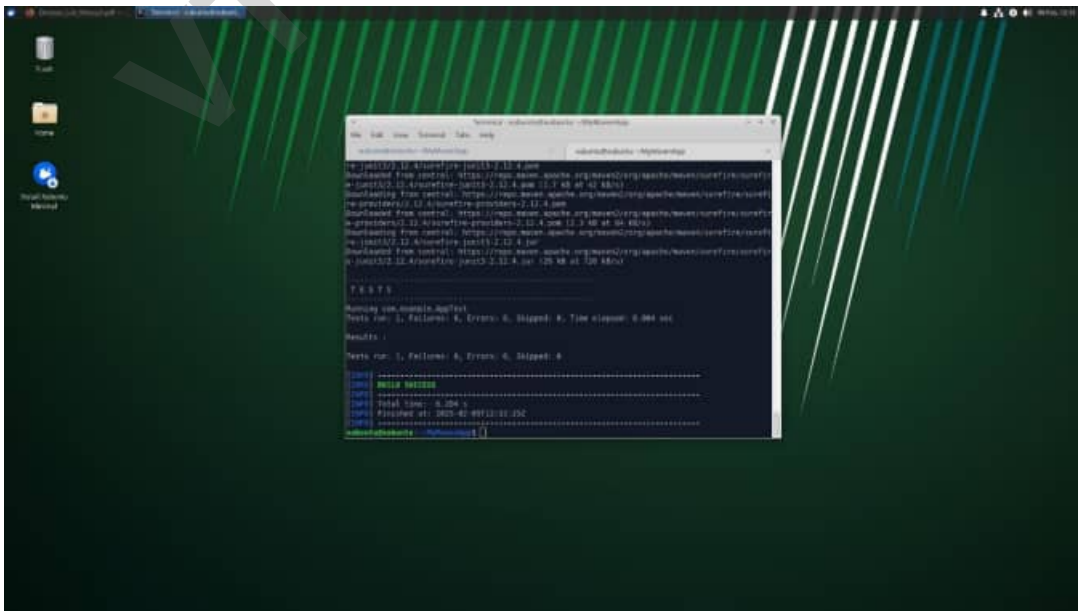
- **Compile the Project:**

```
mvn compile
```



- **Run Unit Tests:**

```
mvn test
```



- **Package the Application:**

  ```
  mvn package
  ```

  This command compiles, tests, and packages your code into a JAR file located in the `target` directory. *Screenshot Tip:* Capture the listing of the `target` directory showing the JAR file.

- **Clean the Project:**

  ```
  mvn clean
  ```

  This removes any files generated by previous builds.

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

## 1. Introduction to Gradle

**Gradle** is a modern build automation tool designed to be highly flexible, fast, and scalable. It is widely used in Java projects, Android development, and many multi-language projects. Here's what makes Gradle stand out:

- **Flexible Build Scripts:** Gradle uses a Domain Specific Language (DSL) based on either Groovy (by default) or Kotlin. This provides a more dynamic and expressive way to define build logic compared to static XML configurations (as used in Maven).
- **Incremental Builds:** Gradle optimizes build times by determining what parts of the project have changed and rebuilding only those parts.
- **Task Automation:** Everything in Gradle is treated as a task, allowing you to create custom tasks or reuse existing ones for compiling code, running tests, packaging, and more.
- **Dependency Management:** Like Maven, Gradle can automatically download and manage dependencies from remote repositories (e.g., Maven Central, JCenter, or custom repositories).



## 2. Setting Up a Gradle Project

### Step-by-Step Instructions

### Step 1: Ensure Gradle is Installed

Before starting, verify that Gradle is installed on your Ubuntu system. Open a terminal and run:

```
gradle -v
```

If you see version information, you're set. If not, follow the Gradle installation instructions from Experiment 1.

### Step 2: Create a New Gradle Project

Gradle offers an interactive command to generate a new project using an initialization script. To create a basic Java application project:

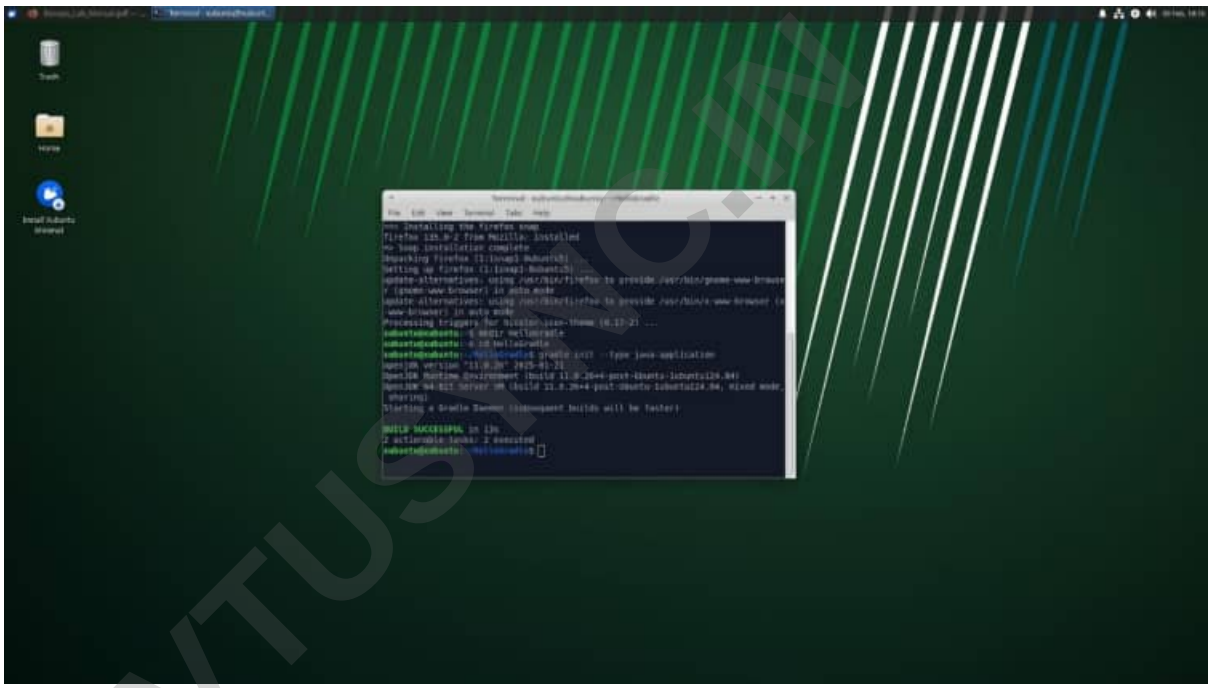
1. **Create a new directory for your project and navigate to it:**

```
mkdir HelloGradle
cd HelloGradle
```

4. **Run the Gradle init command:**

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

You will be prompted to choose between a few options (if you don't use the non-interactive mode). Choose defaults or specify details as needed.



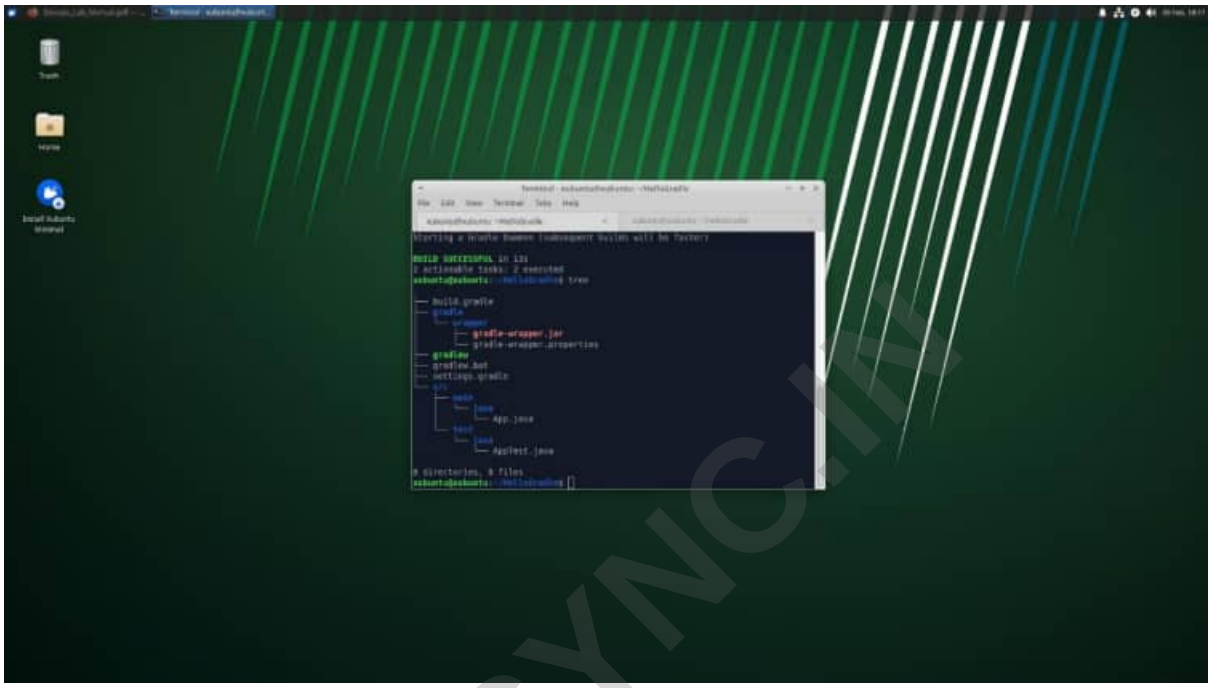


## Step 3: Explore the Project Structure

After project creation, the directory structure typically looks like this:

```
HelloGradle/
├── build.gradle                // The primary build script (Groovy DSL by default)
├── gradle/                     // Contains Gradle wrapper files (if generated)
├── gradlew                     // Unix shell script to run Gradle wrapper
├── gradlew.bat                 // Windows batch script for Gradle wrapper
├── settings.gradle             // Contains project settings and names
└── src
```

```
├── main
│   └── java
│       └── App.java    // Your main application source file
└── test
    └── java
        └── AppTest.java // Your test cases
```





## Explanation of Components:

- **build.gradle:**
  This is the main build script written in Groovy (or Kotlin if you choose). It defines plugins, repositories, dependencies, and tasks.
- **settings.gradle:**
  A small script that defines the project's name and, in multi-project builds, the included subprojects.
- **gradlew / gradlew.bat:**
  The Gradle wrapper scripts. They allow you to run Gradle without requiring a separate installation on every machine by automatically downloading the correct Gradle version.
- **src/main/java:**
  Contains your application's source code.
- **src/test/java:**
  Contains your unit tests.

## 3. Understanding Gradle Build Scripts

A **build script** in Gradle is a programmatic file that instructs Gradle on how to build your project. It can be written in two main DSLs:

- **Groovy DSL (build.gradle):** The traditional and most common syntax.
- **Kotlin DSL (build.gradle.kts):** A statically-typed alternative that leverages Kotlin's language features.

## A. Groovy DSL Example (build.gradle)

Below is an example of a basic `build.gradle` for a Java application using Groovy DSL:



```
plugins {
    // Apply the Java plugin for compiling Java code
    id 'java'
    // Apply the application plugin to add support for building an application
    id 'application'
}

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

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

dependencies {
    // Define your dependencies. For example, JUnit for testing:
    testImplementation 'junit:junit:4.13.2'
}

application {
    // Define the main class for the application.
    mainClass = 'com.example.App'
}
```

```
// A custom task example: printing a greeting
task hello {
    doLast {
        println 'Hello, Gradle!'
    }
}
```

**Explanation:**

- **plugins block:** Declares the plugins used. The `java` plugin adds Java compilation and testing tasks, while the `application` plugin adds tasks for running the application.
- **group/version:** Sets project coordinates.
- **repositories block:** Configures the repository (Maven Central) where dependencies will be resolved.
- **dependencies block:** Lists the libraries your project depends on.
- **application block:** Specifies the main class of your application.
- **Custom task:** Defines a task named `hello` that prints a greeting when run.

## B. Kotlin DSL Example (build.gradle.kts)

Here's how the same build configuration might look in Kotlin DSL:

Create a file named build.gradle.kts (or convert your file) with the following content:

```
plugins {
    // Apply the Java plugin for compiling Java code
    java
    // Apply the application plugin to add support for building an application
    application
}

group = "com.example"
version = "1.0"

repositories {
    mavenCentral()
}
```

```
dependencies {
    // Define dependencies using Kotlin DSL syntax
    testImplementation("junit:junit:4.13.2")
}

application {
    // Set the main class for the application
    mainClass.set("com.example.App")
}

// A custom task example using Kotlin DSL
tasks.register("hello") {
    doLast {
        println("Hello, Gradle with Kotlin DSL!")
    }
}
```

**Explanation:**

- **Kotlin DSL Syntax:** Uses a statically typed syntax, which can be more intuitive for developers familiar with Kotlin.
- **Plugins, Repositories, and Dependencies:** Defined similarly to the Groovy DSL but with Kotlin-style function calls and property access.
- **Custom Task Registration:** Uses `tasks.register` to define a task, similar in functionality to the Groovy DSL.



## 4. Dependency Management in Gradle

### How It Works:

- **Repositories:** Gradle downloads dependencies from defined repositories (e.g., Maven Central).
- **Dependency Notation:** Dependencies are defined in a simple format: `group:artifact:version`.
- **Configuration Types:** Gradle provides various configurations such as `implementation`, `compileOnly`, `runtimeOnly`, and `testImplementation` to manage the scope of each dependency.

### Example:

In our build script examples, we included JUnit for testing:

```
dependencies {
    testImplementation 'junit:junit:4.13.2'
}
```

This line tells Gradle to download JUnit version 4.13.2 from Maven Central and include it in the test classpath.

### Additional Dependency Configurations:

- **implementation:** Used for dependencies required at compile time.
- **runtimeOnly:** Used for dependencies required at runtime but not needed for compilation.

## 5. Task Automation in Gradle

In Gradle, nearly everything is a **task**. Tasks represent individual units of work (compiling code, running tests, packaging applications). Gradle comes with many built-in tasks (provided by plugins) and allows you to define your own custom tasks.

### Built-in Tasks:

- **`compileJava`:** Compiles the source code in `src/main/java`.
- **`test`:** Runs tests in `src/test/java`.
- **`jar`:** Packages compiled code into a JAR file.
- **`run`:** Runs the application (if the application plugin is applied).

### Creating Custom Tasks:

- **Custom Task in Groovy DSL:**

Ensure your `build.gradle` includes the following custom task:

```
groovy
CopyEdit
task hello {
```

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

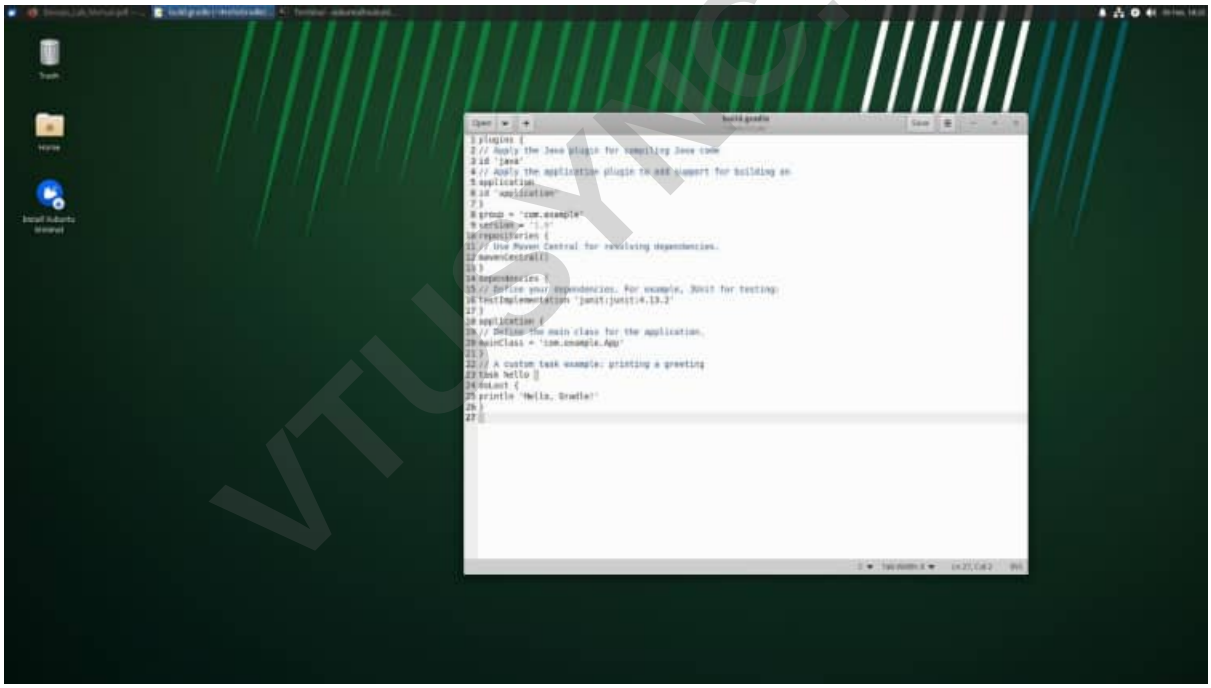
- **Run the Custom Task:**

  - **Command:**

    ```
    gradle hello
    ```

  - **Expected Output:**
    You should see:

    ```
    > Task :hello
    Hello, Gradle!
    ```



- **Custom Task in Kotlin DSL (if using build.gradle.kts):**

Your Kotlin DSL file should include:

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

- **Run the Custom Task:**

  - **Command:**

    ```
    ./gradlew hello
    ```

  - **Expected Output:**
    You should see the greeting printed from the custom task.

## Task Dependencies:

You can make one task depend on another. For instance, you might want your custom task to run after the `build` task:

**Groovy DSL:**

```
task greet(dependsOn: build) {
    doLast {
        println 'Build is complete! Time to celebrate!'
    }
}
```

**Kotlin DSL:**

```
tasks.register("greet") {
    dependsOn("build")
    doLast {
        println("Build is complete! Time to celebrate!")
    }
}
```

*Screenshot Tip:* Capture your terminal output after running these custom tasks.

## 6. Running and Verifying Your Gradle Project

## Common Gradle Commands:

**Build the Project:**

**Compile, Test, and Package:**

- **Command:**

  ```
  gradle build
  ```

- **What it does:**
  - Compiles source code (`compileJava`).
  - Runs tests (`test`).
  - Packages the application into a JAR file (`jar`).
- **Expected Output:**
  Look for a "BUILD SUCCESSFUL" message.
- **Screenshot Tip:** Capture the full output of the `gradle build` command

**Running the Application**

**Run the Application:**

**Command:**

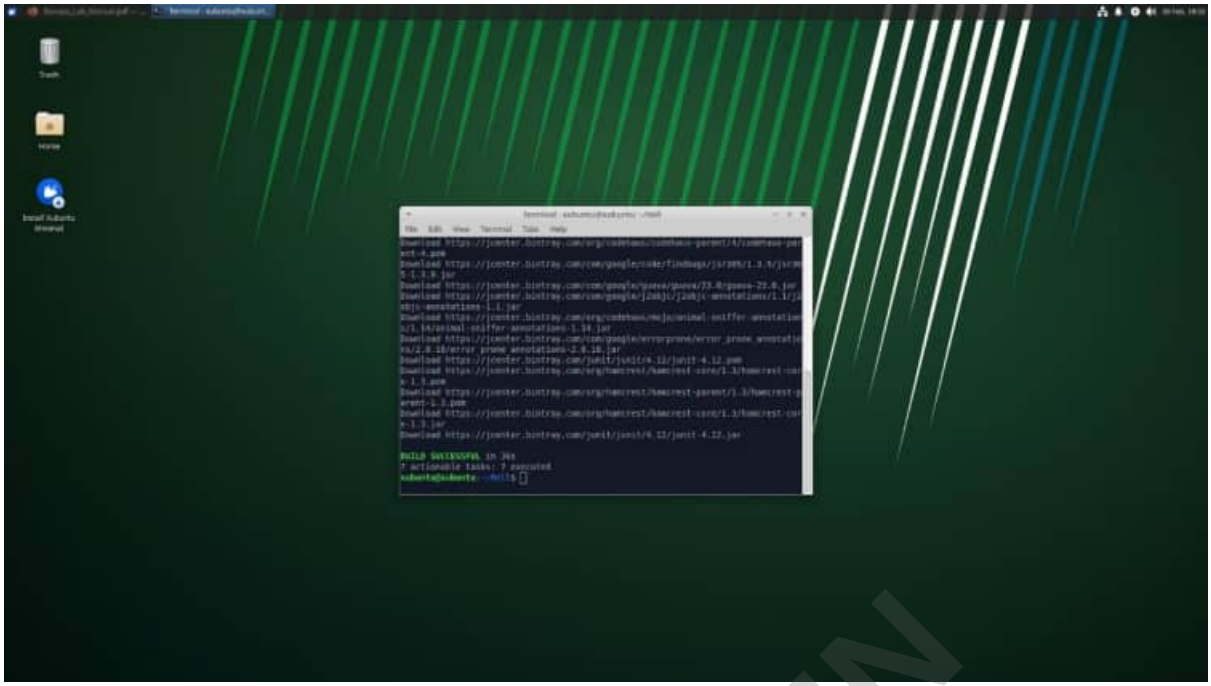
gradle run

**What it does:**

Runs your main class as specified in the application block.

**Expected Output:**

Any output from your application (for example, if your App.java prints a message).



## Executing Custom Tasks

**Run the Custom Task:**

**Command:**

```
gradle hello
```

**Expected Output:**

The custom greeting message you defined earlier.

**Screenshot Tip:** Capture the output showing the greeting.

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

## Part A: Build and Run a Java Application with Maven

### 1. Create the Maven Project

1. **Open your Terminal.**
2. **Generate the Maven Project using the Quickstart Archetype:**

   Type the following command and press **Enter**:

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

   - **What it does:**
     This command creates a new Maven project with the group ID `com.example` and the artifact ID `HelloMaven`. The archetype sets up a basic Java application, including a sample test.
   - **Expected Output:**
     Maven will display messages as it downloads dependencies and generates the project files.
   - **Screenshot Tip:**
     Capture the terminal output showing the successful project generation.
3. **Change Directory into the Newly Created Project:**

```
4. cd HelloMaven
```

   - **Screenshot Tip:**
     Capture the output of the `pwd` or `ls` command to show the project structure.

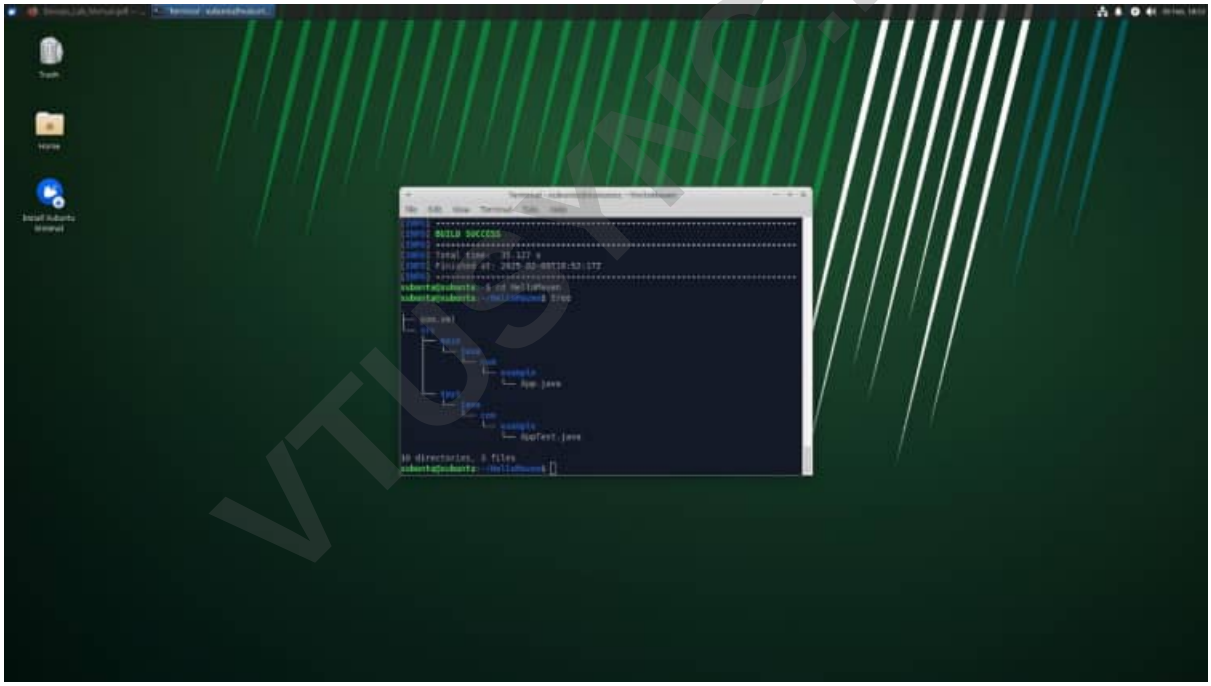
### 2. Explore the Maven Project Structure

Your project directory should have a structure similar to this:

```
HelloMaven/
├── pom.xml
└── src
```

```
├── main
│   └── java
│       └── com
│           └── example
│               └── App.java
└── test
    └── java
        └── com
            └── example
                └── AppTest.java
```

- **`pom.xml`:** The Maven configuration file (POM) that defines your project's coordinates, dependencies, and plugins.
- **`src/main/java`:** Contains your application's source code.
- **`src/test/java`:** Contains unit tests.





## 3. Build the Maven Project

1. **Compile and Package the Application:**

2. `mvn package`

   - **What it does:**

     This command compiles the source code, runs tests, and packages your application into a JAR file (located in the `target` directory).

- **Expected Output:**
  You should see a “BUILD SUCCESS” message along with information on the created JAR file.

## 4. Run the Maven Application

1. **Run the Application Using the JAR File:**

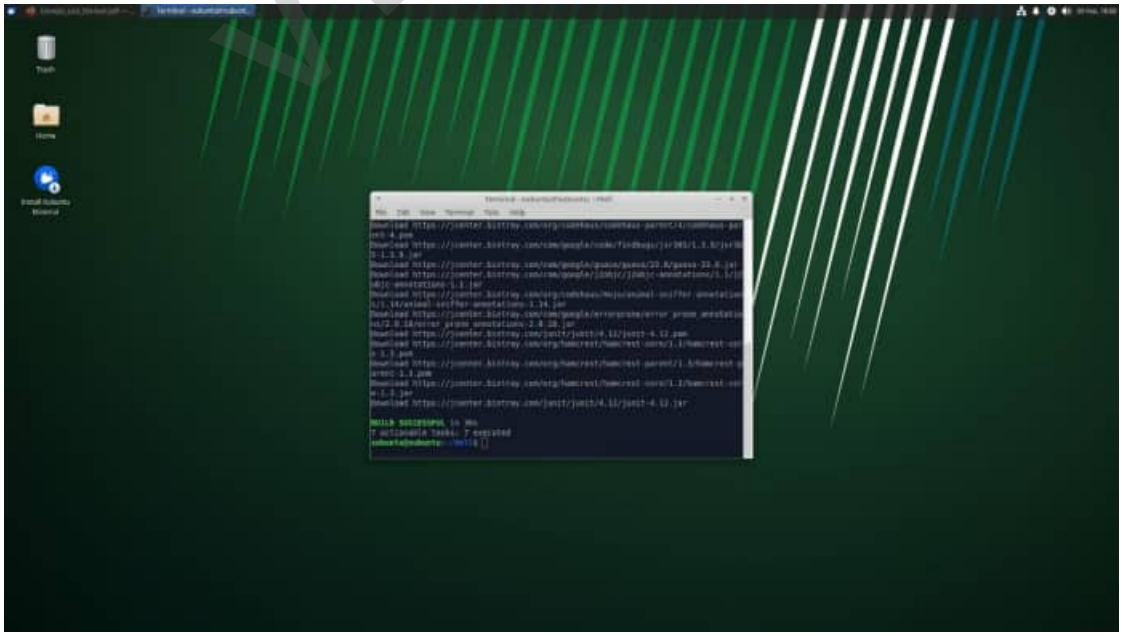
   Execute the following command:

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

   - **What it does:**
     This command runs the `com.example.App` class from the JAR file generated in the previous step.
   - **Expected Output:**
     The output should display:
   - `Hello World!`

     (This is the default message printed by the generated `App.java`.)

   - **Screenshot Tip:**
     Capture the terminal output showing “Hello World!” as proof that the application ran successfully.



## Part B: Migrate the Application to Gradle

In this part, you will create a Gradle project that contains the same Java application code, then build and run it using Gradle.

### 1. Create a New Gradle Project

1. **Open a New Terminal Window or Navigate Back to Your Workspace.**
2. **Create a New Directory for the Gradle Project:**

```
mkdir HelloMavenGradle
cd HelloMavenGradle
```

   - **What it does:**

     Creates and navigates into a folder named `HelloMavenGradle` for your new Gradle project.

5. **Initialize the Gradle Project Using the Java Application Type:**

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

   - **What it does:**

     Gradle will generate a basic Java application project with a default project structure.

   - **Expected Output:**

     You will see interactive prompts or a confirmation message stating that the project has been generated.

### 2. Adjust the Gradle Project to Use the Same Code

The Gradle project structure will look similar to this:

```
HelloMavenGradle/
├── build.gradle
├── settings.gradle
└── src
    ├── main
    │   └── java
    │       └── App.java
    └── test
        └── java
            └── AppTest.java
```

**A. Replace or Update the Source Code:**

1. **Copy the Application Code from the Maven Project:**
    - Open the Maven project directory (`HelloMaven/src/main/java/com/example/App.java`).
    - Copy its contents (which should print "Hello World!") into the Gradle project.
2. **Ensure the Package Declaration Matches:**
    - In the Maven project, the class is in the package `com.example`.
    - If the Gradle project's default `App.java` is not in this package, create the proper directory structure:
    - `mkdir -p src/main/java/com/example`
    - Move (or copy) the `App.java` file into the `com/example` folder:
    - `mv src/main/java/App.java src/main/java/com/example/`
    - Do the same for the test file if you wish to migrate tests.

**B. Update the Gradle Build Script**

1. **Open the `build.gradle` File in Your Editor:**
2. `nano build.gradle`
3. **Modify the `application` Block to Set the Correct Main Class:**

   Change or add the following in the file:

```
application {
    // Update the mainClass to reflect the package structure
    mainClass = 'com.example.App'
}
```

## 3. Build and Run the Gradle Application

1. **Build the Project:**
2. `gradle build`
    - **What it does:**
      Compiles the source code, runs tests, and packages the application.
    - **Expected Output:**
      Look for a "BUILD SUCCESSFUL" message.
3. **Run the Application Using Gradle:**

4. `gradle run`
    - **What it does:**
      Uses the `application` plugin to run the main class defined in the `build.gradle` file.
    - **Expected Output:**
      The output should display:
    - `Hello World!`

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

## 1. What Is Jenkins?

### Definition and Overview

**Jenkins** is an open-source automation server written in Java. It is widely used to facilitate **Continuous Integration (CI)** and **Continuous Deployment/Delivery (CD)** pipelines in software development. Jenkins automates repetitive tasks related to building, testing, and deploying software, helping teams to integrate changes continuously and deliver high-quality applications faster.

### Key Functionalities

- **Continuous Integration/Delivery:** Automatically builds, tests, and deploys code after each commit.
- **Extensible Plugin Ecosystem:** Over 1,500 plugins allow integration with numerous tools such as Git, Maven, Gradle, Docker, and many cloud providers.
- **Automated Builds:** Jenkins can poll version control systems, trigger builds on code changes, and even schedule builds.
- **Pipeline as Code:** With the introduction of Jenkins Pipeline (using either scripted or declarative syntax), you can define your entire build process in a code file (Jenkinsfile) and store it alongside your code.
- **Distributed Builds:** Jenkins can distribute workloads across multiple machines, helping to run tests and builds faster.
- **Monitoring and Reporting:** Provides detailed logs, test reports, and build history.

### Why Use Jenkins?

- **Improved Efficiency:** Automates mundane tasks to free up developer time.
- **Rapid Feedback:** Quickly identifies integration issues with automated tests.
- **Scalability:** Supports distributed builds and parallel testing.
- **Customization:** A highly configurable system that can integrate with nearly every tool in the software development lifecycle.

- **Community Support:** Extensive community and plugin ecosystem that continuously evolve Jenkins' capabilities.

## 2. Installing Jenkins on a Local Machine (Ubuntu)

Below are detailed step-by-step instructions for installing Jenkins on an Ubuntu machine.

### Step 1: Update Your System

Open your terminal and update your system repositories:

```
sudo apt update
sudo apt upgrade -y
```

### Step 2: Install Java

Jenkins requires Java to run. It is recommended to use Java 11 or later. Install OpenJDK 11:

```
sudo apt install openjdk-11-jdk -y
java -version
```

*Expected Output Example:*

```
openjdk version "11.0.11" 2021-04-20
```

### Step 3: Add the Jenkins Repository Key

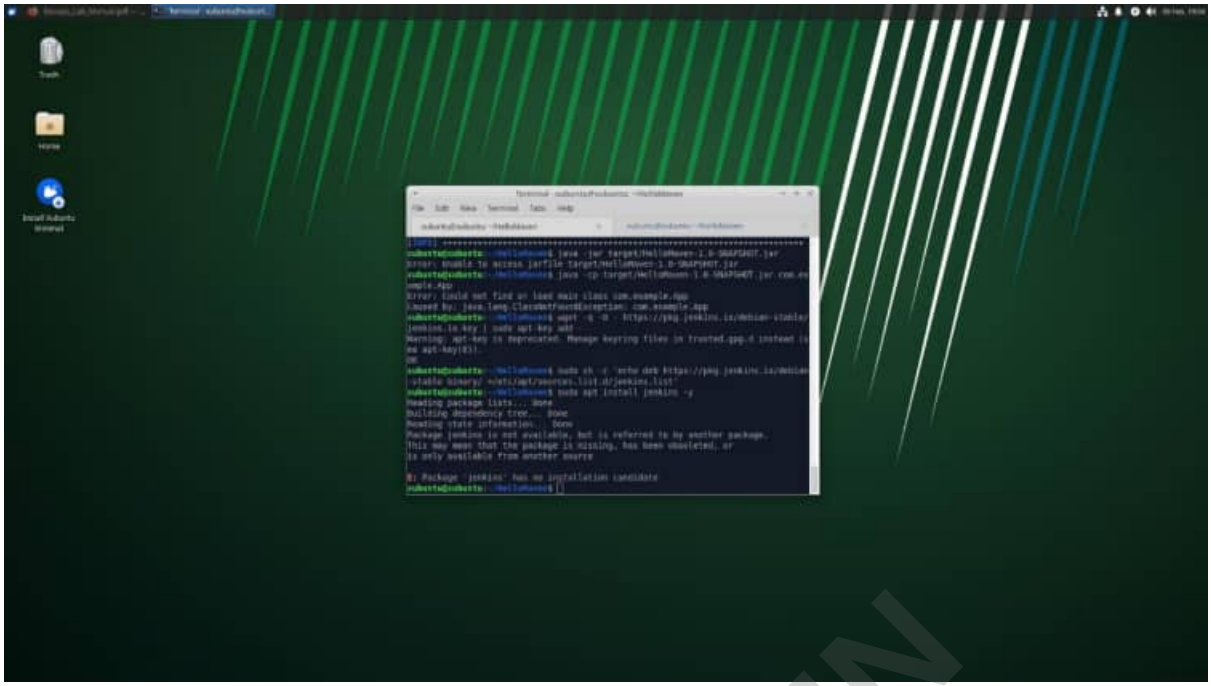
Download and add the repository key so that your system trusts the Jenkins packages:

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

### Step 4: Add the Jenkins Repository

Add the Jenkins repository to your system’s sources list:

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



## Step 5: Update the Repository and Install Jenkins

Update the package list and install Jenkins:

```
sudo apt update
sudo apt install jenkins -y
```

## Step 6: Start and Enable the Jenkins Service

Start Jenkins and configure it to start automatically on boot:

```
sudo systemctl start jenkins
sudo systemctl status jenkins
```

- **What it does:**
  - `start jenkins` launches the Jenkins service.
  - `status jenkins` confirms Jenkins is running.
- *Expected Output:* A status message indicating Jenkins is active (running).

## Step 7: Access Jenkins on the Local Machine

1. **Open a Web Browser:**
   Navigate to:

2. `http://localhost:8080`
3. **Unlock Jenkins:**

   The initial Jenkins screen will ask for an **administrator password**. Retrieve it by running:

4. `sudo cat /var/lib/jenkins/secrets/initialAdminPassword`
    - **What it does:** Displays the auto-generated admin password.
    - *Screenshot Tip:* Capture the terminal output showing the password and the Jenkins unlock screen.
5. **Follow the Setup Wizard:**
    - **Install Suggested Plugins:** Click on "Install suggested plugins" for a typical setup.
    - **Create an Admin User:** Follow prompts to create your first admin user.
    - **Finalize Configuration:** Complete the remaining setup steps (e.g., instance configuration).
    - *Screenshot Tip:* Capture each major step (unlocking Jenkins, plugin installation, and admin user creation).

## 3. Installing Jenkins on the Cloud (Optional)

There are several ways to run Jenkins on the cloud. One common method is to run Jenkins using a Docker container on a cloud virtual machine. Below are instructions using Docker on an Ubuntu cloud server (for example, on AWS EC2, DigitalOcean, or any cloud provider that supports Ubuntu).

### Step 1: Set Up a Cloud VM Running Ubuntu

1. **Provision a New Ubuntu Server:**
    - Create a new VM using your cloud provider's console.
    - Ensure the VM has at least 1–2 GB RAM.
    - Open port **8080** (and port **50000** for agent communication) in the security group settings.
    - Connect to the VM via SSH:
    - `ssh your-username@your-cloud-server-ip`

### Step 2: Install Docker on Your Cloud VM

1. **Update the System:**
2. `sudo apt update`
3. `sudo apt upgrade -y`
4. **Install Docker:**
5. `sudo apt install docker.io -y`
6. `sudo systemctl start docker`
7. `sudo systemctl enable docker`
8. **Verify Docker Installation:**
9. `docker --version`

## Step 3: Run Jenkins in a Docker Container

1. **Pull the Official Jenkins Image:**
2. `sudo docker pull jenkins/jenkins:lts`
3. **Run the Jenkins Container:**
4. `sudo docker run -d --name jenkins -p 8080:8080 -p 50000:50000 -v jenkins_home:/var/jenkins_home jenkins/jenkins:lts`
    - **What it does:**
        - `-d` runs the container in detached mode.
        - `--name jenkins` names the container.
        - `-p 8080:8080` maps port 8080 (Jenkins UI) to the host.
        - `-p 50000:50000` maps the agent communication port.
        - `-v jenkins_home:/var/jenkins_home` mounts a persistent volume for Jenkins data.
    - `sudo docker ps`
5. **Access Jenkins on the Cloud VM:**
    - Open your browser and navigate to:
    - `http://your-cloud-server-ip:8080`
    - Follow the same unlocking and setup wizard process as described in the local installation steps.

## 4. Configuring Jenkins for First Use

After installing Jenkins (locally or on the cloud), complete these steps to configure it for first use:

### Step 1: Unlock Jenkins

- **Retrieve the Admin Password:**
  As described earlier, run:
- `sudo cat /var/lib/jenkins/secrets/initialAdminPassword`
- **Enter the Password:**
  Copy the password into the "Administrator Password" field on the Jenkins unlock page.

## Step 2: Install Suggested Plugins

- **Click "Install suggested plugins":**
  This option installs a set of commonly used plugins (e.g., Git, Maven Integration, Pipeline).
- **Wait for Installation:**
  The plugin installation process might take several minutes.

## Step 3: Create Your First Admin User

- **Enter the Admin User Details:**
  Provide a username, password, full name, and email address.
- **Confirm Creation:**
  Click "Save and Continue" once the details are entered.

## Step 4: Configure the Instance

- **Instance Configuration:**
  Jenkins may ask you to confirm the URL for your Jenkins instance. Verify that it is correct (e.g., `http://localhost:8080` for local installations or your cloud server's IP address for cloud installations).
- **Save the Configuration:**
  Confirm the settings and proceed.

## Step 5: Start Using Jenkins

- **Dashboard:**
  Once the setup is complete, you will be taken to the Jenkins dashboard where you can:
  - Create new jobs (Freestyle projects, Pipelines, etc.)
  - Install additional plugins as needed.

- Monitor builds and view logs.

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

## 1. Overview

### What is a CI Pipeline?

A **Continuous Integration (CI) Pipeline** automates the process of building, testing, and integrating code changes every time code is committed to the repository. This pipeline:

- Automatically checks out the latest code.
- Compiles the application.
- Runs tests to catch errors early.
- Notifies the team of build/test results.

### Why Use Jenkins for CI?

- **Automation:** Jenkins automates the build and test cycle, reducing manual intervention.
- **Immediate Feedback:** Developers get rapid notifications of any integration issues.
- **Extensibility:** With hundreds of plugins available, Jenkins can integrate with version control systems, build tools (Maven, Gradle), testing frameworks, and more.
- **Pipeline as Code:** Using Jenkins Pipelines (defined in a `Jenkinsfile`), you can manage the CI process as part of your source code repository.



## 2. Setting Up a CI Pipeline with Jenkins (Freestyle Project)

This section explains how to create a CI pipeline as a Freestyle project that integrates with a Maven or Gradle project.

### Step 1: Create a New Jenkins Job

1. **Log into Jenkins:**
    - Open your web browser and navigate to your Jenkins URL (e.g., `http://localhost:8080` or your cloud instance URL).
    - Log in with your admin credentials.

2. **Create a New Job:**
    - On the Jenkins dashboard, click on **"New Item"**.
    - **Enter an Item Name:** For example, `Maven-CI` (or `Gradle-CI` if you prefer Gradle).
    - **Select "Freestyle project"**.
    - Click **"OK"**.

## Step 2: Configure Source Code Management (SCM)

1. **Select SCM:**
    - In the job configuration page, scroll down to the **"Source Code Management"** section.
    - Select **"Git"** (if using Git for version control).
2. **Enter Repository Details:**
    - **Repository URL:** Enter the URL of your Git repository (for example, `https://github.com/yourusername/your-maven-project.git`).
    - **Credentials:** If your repository is private, click **"Add"** to provide the necessary credentials.
    - Optionally, specify the **Branch Specifier** (e.g., `*/main`).

## Step 3: Add Build Steps

### A. For a Maven Project

1. **Add Maven Build Step:**
    - Scroll down to **"Build"** and click on **"Add build step"**.
    - Select **"Invoke top-level Maven targets"**.
    - **Goals:** In the Goals field, enter:
    - `clean package`

      This command instructs Maven to clean the previous build artifacts, compile the code, run tests, and package the application into a JAR/WAR file.

    - Optionally, set the **POM File** location if it is not in the default location (`pom.xml`).

**B. For a Gradle Project**

1. **Add Gradle Build Step:**
    - If you are integrating a Gradle project, click on **"Add build step"** and choose **"Invoke Gradle script"** (this option might be available if you have installed a Gradle plugin in Jenkins).
    - **Tasks:** In the tasks field, enter:
    - `clean build`

      This instructs Gradle to clean previous build outputs and then build the project, running tests along the way.

    - **Switches:** If needed, you can add additional flags (for example, `--info` or `--stacktrace` for more detailed output).

## Step 4: Configure Post-build Actions

1. **Publish Test Results:**
    - Scroll down to the **"Post-build Actions"** section.
    - Click **"Add post-build action"** and select **"Publish JUnit test result report"**.
    - **Test Report XMLs:** In the field, enter the pattern that matches your test report files. For example:
    - `**/target/surefire-reports/*.xml`

      for Maven, or a similar path for Gradle projects.

## Step 5: Save and Run the Job

1. **Save the Configuration:**
    - Click **"Save"** at the bottom of the job configuration page.
2. **Trigger a Build:**
    - On the job's main page, click **"Build Now"**.
    - The build will be added to the build history on the left side.
3. **Monitor Build Output:**
    - Click on the build number (e.g., `#1`) and then click **"Console Output"**.

- Verify that Jenkins successfully checks out the code, runs the build commands (Maven or Gradle), and executes tests.
- Look for **"BUILD SUCCESS"** or the equivalent output to confirm that the build and tests passed.

## 3. Setting Up a CI Pipeline with Jenkins (Pipeline as Code)

For greater flexibility and version-controlled CI configuration, you can use a **Jenkins Pipeline** defined in a `Jenkinsfile`.

### Step 1: Create a Pipeline Job

1. **Log into Jenkins** and click on **"New Item"**.
2. **Enter an Item Name:** For example, `Pipeline-CI`.
3. **Select "Pipeline"** and click **"OK"**.

### Step 2: Define the Pipeline Script

1. **Configure the Pipeline:**
   - In the job configuration page, scroll to the **"Pipeline"** section.
   - Choose **"Pipeline script"** (or "Pipeline script from SCM" if you want to load the script from your repository).
2. **Enter the Pipeline Script:**

   Below are sample pipeline scripts for Maven and Gradle projects.

   **Example for a Maven Project:**

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                // Check out code from Git repository
                git url: 'https://github.com/yourusername/your-maven-
project.git', branch: 'main'
            }
```

```
        }
        stage('Build') {
            steps {
                // Run Maven build
                sh 'mvn clean package'
            }
        }
        stage('Test') {
            steps {
                // Optionally, separate test execution if needed
                sh 'mvn test'
            }
        }
    }

    post {
        always {
            // Archive test reports
            junit '**/target/surefire-reports/*.xml'
        }
        success {
            echo 'Build and tests succeeded!'
        }
        failure {
            echo 'Build or tests failed.'
        }
    }
}
```

**Example for a Gradle Project:**

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                // Check out code from Git repository
                git    url:    'https://github.com/yourusername/your-
gradle-project.git', branch: 'main'
            }
```

```
        }
        stage('Build') {
            steps {
                // Run Gradle build
                sh './gradlew clean build'
            }
        }
        stage('Test') {
            steps {
                // Run tests (if not already run in the build stage)
                sh './gradlew test'
            }
        }
    }

    post {
        always {
            // Archive test reports (modify the path according to your
project structure)
            junit '**/build/test-results/test/*.xml'
        }
        success {
            echo 'Build and tests succeeded!'
        }
        failure {
            echo 'Build or tests failed.'
        }
    }
}
```



3. **Save the Pipeline Script:**
    - After entering your pipeline script, click **"Save"**.

## Step 3: Run the Pipeline

1. **Trigger the Build:**
    - On the Pipeline job's main page, click **"Build Now"**.
    - Monitor the build progress through the Pipeline visualization or by clicking on the build number and then **"Console Output"**.
2. **Verify the Results:**

- Confirm that each stage (Checkout, Build, Test) executes successfully.
- Review the archived test reports to verify that tests have run and passed.

# Experiment 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

## 1. Introduction to Ansible

### What Is Ansible?

**Ansible** is an open-source IT automation and configuration management tool. It allows you to manage multiple servers and perform tasks such as:

- **Configuration Management:** Automate the configuration of servers.
- **Application Deployment:** Deploy applications consistently.
- **Orchestration:** Coordinate complex IT workflows and processes.

### Key Concepts in Ansible

- **Inventory:**
  An inventory is a file (usually in INI or YAML format) that lists the hosts (or groups of hosts) you want to manage. It tells Ansible which machines to target.
- **Playbook:**
  A playbook is a YAML file that defines a set of tasks to be executed on your target hosts. It is the heart of Ansible automation. In a playbook, you specify:
  - **Hosts:** The target machines (or groups) on which the tasks should run.
  - **Tasks:** A list of actions (using modules) that should be executed.
  - **Modules:** Reusable, standalone scripts that perform specific actions (e.g., installing packages, copying files, configuring services).
- **Modules:**
  Ansible comes with a large collection of built-in modules (such as `apt`, `yum`, `copy`, `service`, etc.). These modules perform specific tasks on target hosts. You can also write custom modules if needed.

### Why Use Ansible?

- **Agentless:** Ansible uses SSH to communicate with target hosts, so no agent needs to be installed on them.
- **Simplicity:** Playbooks use simple YAML syntax, making them easy to write and understand.
- **Idempotence:** Ansible tasks are idempotent, meaning running the same playbook multiple times yields the same result, ensuring consistency.
- **Scalability:** Ansible can manage a small number of servers to large infrastructures with hundreds or thousands of nodes.

## 2. Installing Ansible on Ubuntu

Before writing a playbook, you need to install Ansible on your control machine (your local Ubuntu system).

### Step 1: Update Your System

Open your terminal and run:

```
sudo apt update
sudo apt upgrade -y
```

### Step 2: Install Ansible

Install Ansible using apt:

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

Verify the installation by checking the version:

```
ansible --version
```

*Expected Output Example:*

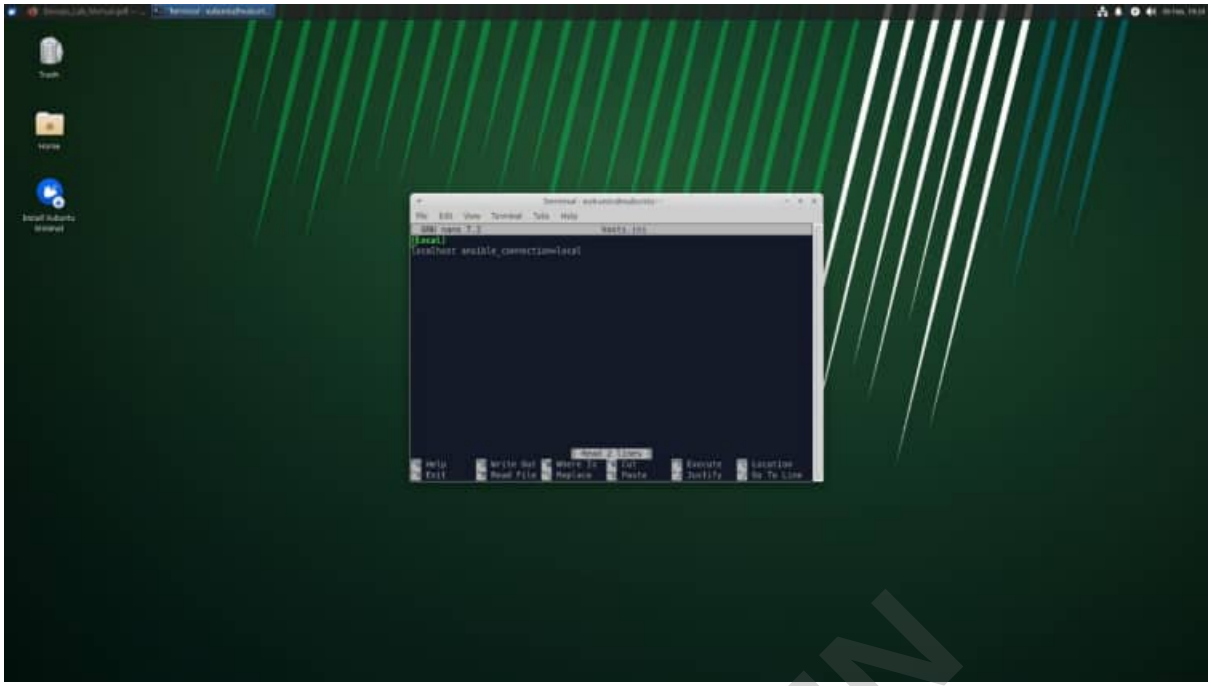
```
ansible 2.9.x
  config file = /etc/ansible/ansible.cfg
  ...
```

## 3. Creating an Ansible Inventory

An inventory file lists the hosts you want to manage. For this experiment, you can use the local host.

### Step 1: Create an Inventory File

1. Open your text editor to create a file called `hosts.ini`:
2. `nano hosts.ini`
3. Add the following content to define the local host:
4. `[local]`
5. `localhost ansible_connection=local`
    - **Explanation:**
        - `[local]` is a group name.
        - `localhost` is the target host.
        - `ansible_connection=local` tells Ansible to execute commands on the local machine without SSH.
6. Save the file by pressing **Ctrl+O** then **Enter**, and exit with **Ctrl+X**.



## 3. Automated Server Configurations

While our experiment covered the basics, here's how you can extend it:

- **Configuring Services:**
  Use modules like `service` to start, stop, or restart services. For example, you can automate the configuration of web servers (e.g., Apache or Nginx).
- **Managing Files and Templates:**
  Use the `copy` or `template` modules to deploy configuration files across your servers. This is useful for maintaining consistent configuration settings.
- **User and Group Management:**
  The `user` and `group` modules allow you to create or modify user accounts, ensuring that the correct permissions and roles are applied automatically.
- **Advanced Orchestration:**
  Ansible playbooks can include conditionals, loops, and error handling to manage more complex setups, ensuring idempotence and consistency across your infrastructure.

## Example: Automating a Web Server Configuration

If you wanted to automate the configuration of an Nginx server, your playbook might include tasks such as:

```
- name: Configure Nginx Web Server
  hosts: webservers
  become: yes
  tasks:
    - name: Update apt cache
      apt:
        update_cache: yes

    - name: Install Nginx
      apt:
        name: nginx
        state: present

    - name: Copy Nginx configuration file
      template:
        src: nginx.conf.j2
        dest: /etc/nginx/nginx.conf
      notify:
        - Restart Nginx

  handlers:
    - name: Restart Nginx
      service:
        name: nginx
        state: restarted
```



## 5. Writing a Basic Ansible Playbook

You will now create a simple playbook that performs two common tasks:

- **Updating the apt cache**
- **Installing a package (e.g., `curl`)**

### Step 1: Create the Playbook File

1. Open your text editor to create a file called `setup.yml`:

```
2. nano setup.yml
```

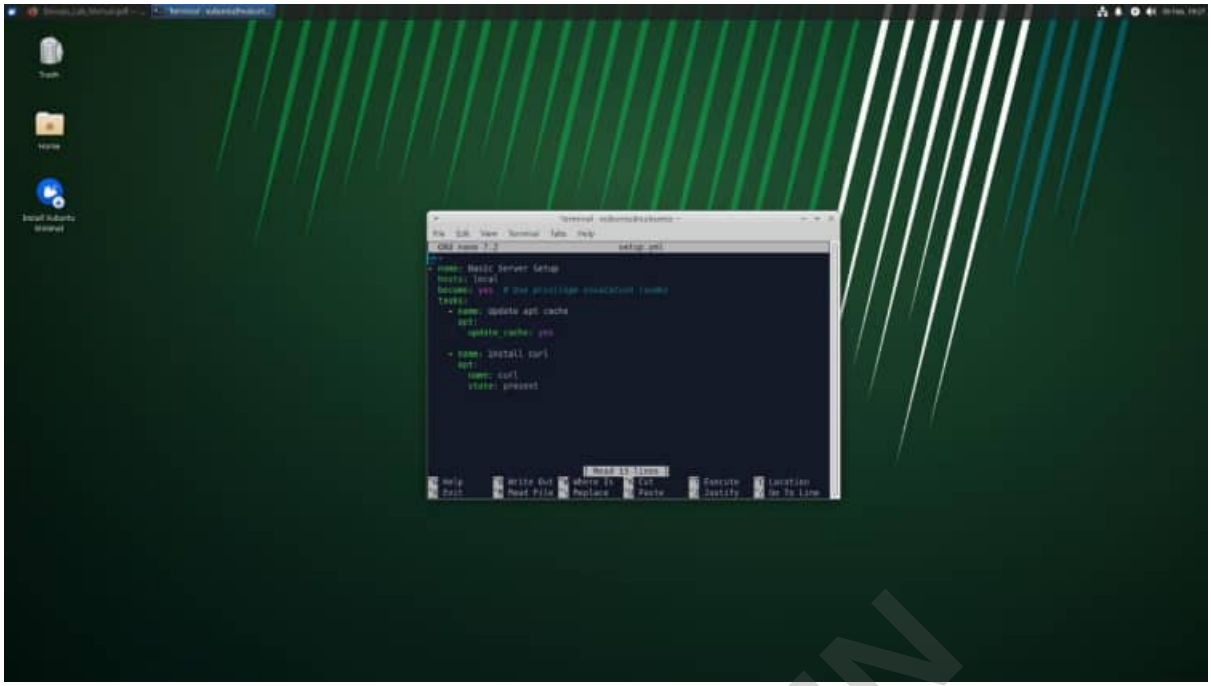
3. Add the following YAML content:

```
---
- name: Basic Server Setup
  hosts: local
  become: yes  # Use privilege escalation (sudo)
  tasks:
    - name: Update apt cache
      apt:
        update_cache: yes

    - name: Install curl
      apt:
        name: curl
        state: present
```

   - **Explanation:**
     - `name`: Provides a descriptive name for the play.
     - `hosts`: Specifies the group or hosts from the inventory file.
     - `become: yes`: Uses sudo to perform tasks that require elevated privileges.
     - **Tasks Section:**
       - **Update apt cache:** Uses the `apt` module to update the package cache.
       - **Install curl:** Uses the `apt` module to install the `curl` package if it isn't already installed.

17. Save and exit the file.



## 6. Running the Ansible Playbook

### Step 1: Execute the Playbook

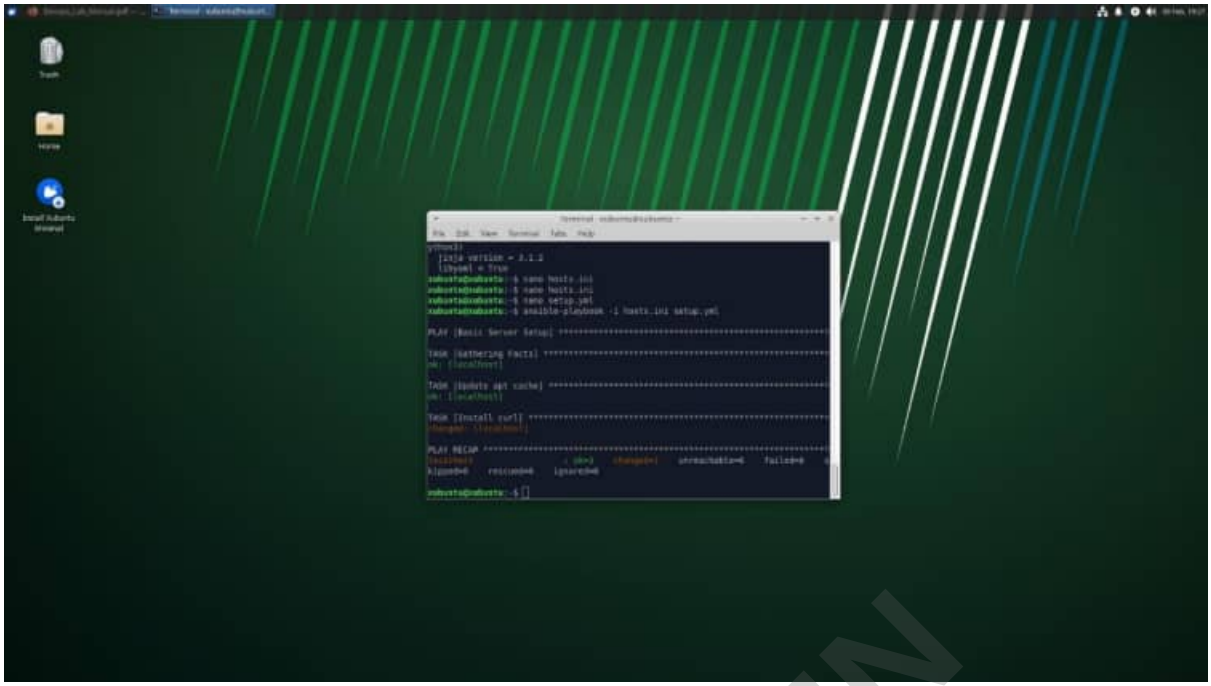
In your terminal, run the following command:

```
ansible-playbook -i hosts.ini setup.yml
```

- **Explanation:**
  - `ansible-playbook`: The command to run an Ansible playbook.
  - `-i hosts.ini`: Specifies the inventory file.
  - `setup.yml`: The playbook file you just created.

*Expected Output:*

- An output that details each task:
  - For example, a task summary might show “ok=2 changed=1” (if the apt cache was updated and curl was installed).


Trash
Home
Install Ubuntu 20.04 LTS
File Edit View Terminal Tabs Help
ansible:
[init] version = 2.1.2
[libyaml] = True
xubuntu@xubuntu:~$ nano hosts.ini
xubuntu@xubuntu:~$ nano hosts.ini
xubuntu@xubuntu:~$ nano setup.yml
xubuntu@xubuntu:~$ ansible-playbook -i hosts.ini setup.yml
PLAY [Basic Server Setup] ***
TASK [Gathering Facts] ***
ok: [localhost]
TASK [Update apt cache] ***
ok: [localhost]
TASK [Install curl] ***
changed: [localhost]
PLAY RECAP ***
localhost : ok=3 changed=1 unreachable=0 failed=0
ignored=0
xubuntu@xubuntu:~$

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

## 1. Overview

In this experiment, you will:

- Set up a Jenkins job to automatically build a Maven project from source control.
- Archive the build artifact (a JAR file) produced by Maven.
- Integrate an Ansible deployment step within Jenkins (using a post-build action) to deploy the artifact to a target location.
- Verify that the artifact is deployed successfully.

This exercise demonstrates how Continuous Integration (CI) and automated configuration management can work together to streamline the build-and-deploy process.



## 2. Prerequisites

Before you begin, ensure that:

- **Jenkins** is installed, running, and accessible (locally or on the cloud).
- **Maven Project:** You have a Maven project available in a Git repository (or stored locally). For this example, we will assume you are using the "HelloMaven" project generated in Experiment 2/4.
- **Git Repository:** The Maven project is committed to a Git repository (e.g., on GitHub) so Jenkins can pull the latest code.
- **Ansible Installed:** Ansible is installed on your control machine (or the Jenkins server) and you have created a basic inventory file (e.g., `hosts.ini`).

*Tip:* Verify installations and repository access before starting this exercise.

## 3. Step 1: Preparing the Maven Project

1. **Ensure Your Maven Project Is in Version Control:**
   If your "HelloMaven" project is not already in a Git repository, navigate to its root and initialize Git:

```
cd path/to/HelloMaven
git init
git add .
git commit -m "Initial commit of HelloMaven project"
```

Then push it to your repository (GitHub, GitLab, etc.).

**Verify the Project Structure:**

Your project should have a standard Maven layout:

```
HelloMaven/
├── pom.xml
└── src
    ├── main/java/com/example/App.java
    └── test/java/com/example/AppTest.java
```

## 4. Step 2: Configuring Jenkins to Build the Maven Project

## A. Create a New Jenkins Job (Freestyle Project)

1. **Log into Jenkins:**
   Open your browser and navigate to your Jenkins URL (e.g., `http://localhost:8080`).
2. **Create a New Job:**
   - Click **"New Item"** on the Jenkins dashboard.
   - **Enter an Item Name:** e.g., `HelloMaven-CI`.
   - **Select "Freestyle project"** and click **"OK"**.

## B. Configure Source Code Management (SCM)

1. **Scroll to the "Source Code Management" Section:**
   - Select **"Git"**.
   - **Repository URL:** Enter your repository URL (e.g., `https://github.com/yourusername/HelloMaven.git`).
   - **Credentials:** If the repository is private, click **"Add"** and provide the necessary credentials.
   - **Branch Specifier:** (e.g., `*/main` or `*/master`).

### C. Add a Maven Build Step

1. **Scroll Down to the "Build" Section:**
    - Click **"Add build step"** and select **"Invoke top-level Maven targets"**.
2. **Configure the Maven Build:**
    - **Goals:** Type:
    - `clean package`

      This command cleans any previous builds, compiles the code, runs tests, and packages the application into a JAR file.

    - **POM File:** (Leave it as default if your `pom.xml` is in the root directory).

## 5. Step 3: Archiving the Artifact

After the Maven build completes, you need to archive the generated artifact so that it can be used later by the deployment process.

1. **Scroll Down to the "Post-build Actions" Section:**
    - Click **"Add post-build action"** and select **"Archive the artifacts"**.
2. **Configure Artifact Archiving:**
    - **Files to Archive:** Type:
    - `target/*.jar`

      This pattern tells Jenkins to archive any JAR file found in the `target` directory.

## 6. Step 4: Integrating Ansible Deployment in Jenkins

Now, integrate an Ansible deployment step into the Jenkins job. You can do this as a post-build action that executes a shell command.

1. **Add Another Post-build Action:**
    - Click **"Add post-build action"** and select **"Execute shell"**.
2. **Configure the Shell Command:**
    - In the command box, add a command to trigger your Ansible playbook. For example:
    - `ansible-playbook -i /path/to/hosts.ini /path/to/deploy.yml`

**Note:**

- Replace `/path/to/hosts.ini` with the full path to your Ansible inventory file.
- Replace `/path/to/deploy.yml` with the full path to your Ansible deployment playbook.

- This command will run after a successful build, deploying the artifact using Ansible.

3. **Save the Jenkins Job:**
   - Click **"Save"** at the bottom of the configuration page.

## 7. Step 5: Writing an Ansible Playbook for Deployment

Create an Ansible playbook that deploys the Maven artifact (the JAR file) generated by Jenkins to a target directory.

### A. Create an Inventory File

If you haven't already, create an inventory file (e.g., `hosts.ini`) that targets the deployment machine. For a local deployment, use:

```
[local]
localhost ansible_connection=local
```

### B. Create the Deployment Playbook

1. **Open Your Text Editor** and create a file called `deploy.yml`:
2. `nano deploy.yml`
3. **Enter the Following YAML Content:**

```
---
- name: Deploy Maven Artifact
  hosts: local
  become: yes
  tasks:
    - name: Copy the artifact to the deployment directory
      copy:
        src:                "/var/lib/jenkins/workspace/HelloMaven-CI/target/HelloMaven-1.0-SNAPSHOT.jar"
```

```
12.          dest: "/opt/deployment/HelloMaven.jar"
```

**Explanation:**

- **hosts:** `local` means the playbook runs on the local machine. Adjust this if deploying to a remote server.
- **become: yes:** Uses sudo privileges to write to system directories.
- **src:** The path should point to the archived artifact in the Jenkins workspace. (Adjust the path if your Jenkins workspace is different.)
- **dest:** The target directory where you want the artifact deployed (ensure this directory exists or modify accordingly).

13. **Save and Exit the File**.

## 8. Step 6: Testing the Complete Pipeline

1. **Trigger a Build in Jenkins:**
   - Navigate to your Jenkins job (`HelloMaven-CI`) and click **"Build Now"**.
   - Monitor the build history. Once the build completes, click the build number (e.g., `#1`) and check the **Console Output**.
   - Look for messages indicating that:
     - The Maven build ran successfully.
     - The artifact was archived.
     - The shell command executed the Ansible playbook.

   *Screenshot Tip:* Capture the console output showing the full pipeline execution, including the deployment step.

2. **Verify Deployment:**
   - Log into your target machine (or check locally) and verify that the artifact has been copied to the destination directory (e.g., `/opt/deployment/HelloMaven.jar`).
   - For example, run:
   - `ls -l /opt/deployment/`
   - The output should list the deployed JAR file.

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

### Overview of Azure DevOps

**Azure DevOps** is a comprehensive suite of cloud-based services designed to support the entire software development lifecycle. It provides tools for planning, developing, testing, delivering, and monitoring applications. Here are the primary services offered:

- **Azure Repos:**
  A set of version control tools that allow you to host Git repositories or use Team Foundation Version Control (TFVC). It offers collaboration features such as pull requests, branch policies, and code reviews.
- **Azure Pipelines:**
  A CI/CD service that helps automate builds, tests, and deployments. It supports multiple languages, platforms, and can run on Linux, Windows, or macOS agents.
- **Azure Boards:**
  A work tracking system that helps teams manage work items, sprints, backlogs, and Kanban boards. It facilitates agile planning and reporting.
- **Azure Test Plans:**
  Provides a solution for managing and executing tests, capturing data about defects, and tracking quality.
- **Azure Artifacts:**
  Allows you to create, host, and share packages (such as Maven, npm, NuGet, and Python packages) with your team, integrating package management into your CI/CD pipelines.

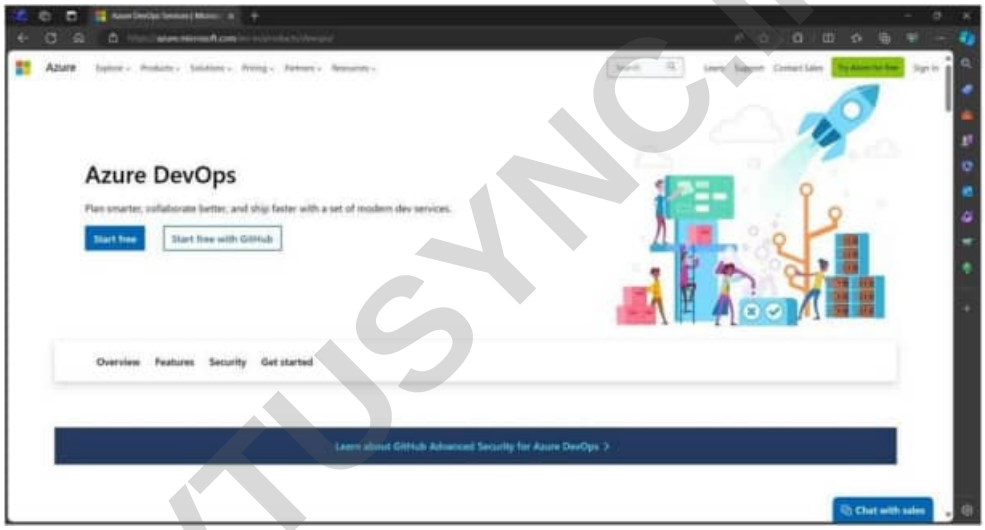
These services integrate with each other and with popular third-party tools to create a cohesive DevOps ecosystem.

# Setting Up an Azure DevOps Account

Before you can start using Azure DevOps services, you need to set up an account and create an organization. Follow these steps:

## Step 1: Sign Up for an Azure DevOps Account

1. **Open Your Web Browser:**
    - Navigate to the Azure DevOps website: https://dev.azure.com.
2. **Sign In or Create a Microsoft Account:**
    - If you already have a Microsoft account (such as Outlook, Hotmail, or Office 365), click **"Sign in"**.
    - If you do not have a Microsoft account, click **"Create one!"** and follow the instructions to create a new Microsoft account.



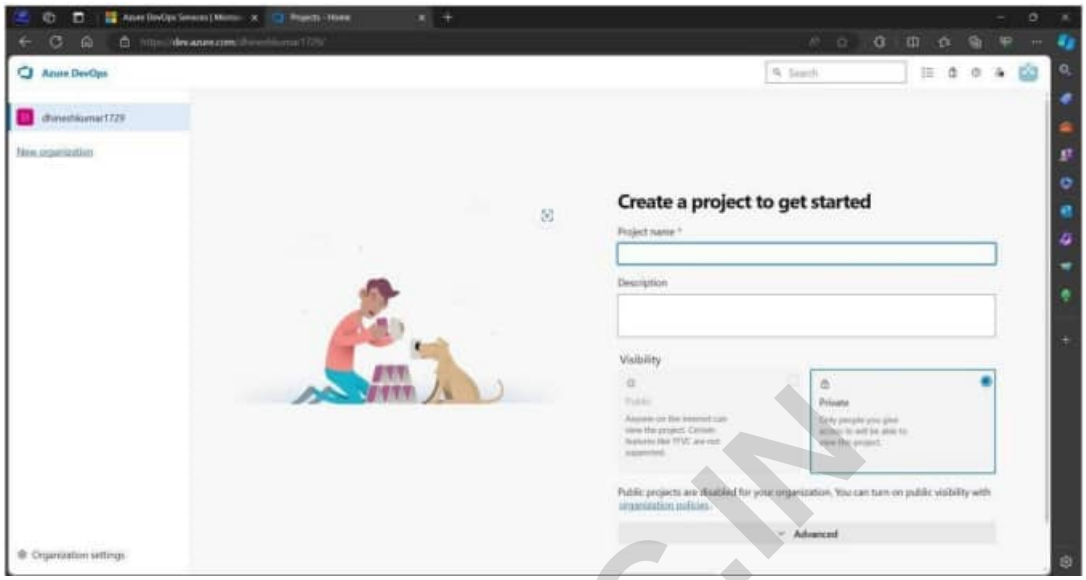



3. **Accept the Terms and Conditions:**
    - Review and accept the terms if prompted.

## Step 2: Create an Azure DevOps Organization

1. **Create a New Organization:**
    - Once signed in, you will be prompted to create an Azure DevOps organization.
    - Enter a unique name for your organization (e.g., `YourCompanyDevOps` or `MyPersonalOrg`).

- **Select a Region:** Choose the geographic region where your data will be stored (select the one closest to you for optimal performance).
- Click **"Continue"** or **"Create"**.




2. **Review Your Organization's Dashboard:**
   - Once created, you will see an overview dashboard for your organization. This dashboard provides navigation links to Repos, Pipelines, Boards, Test Plans, and Artifacts.

## 3. Creating an Azure DevOps Project

After setting up your organization, the next step is to create a project. A project in Azure DevOps is a container for all your source code, pipelines, boards, and other resources.

### Step 1: Create a New Project

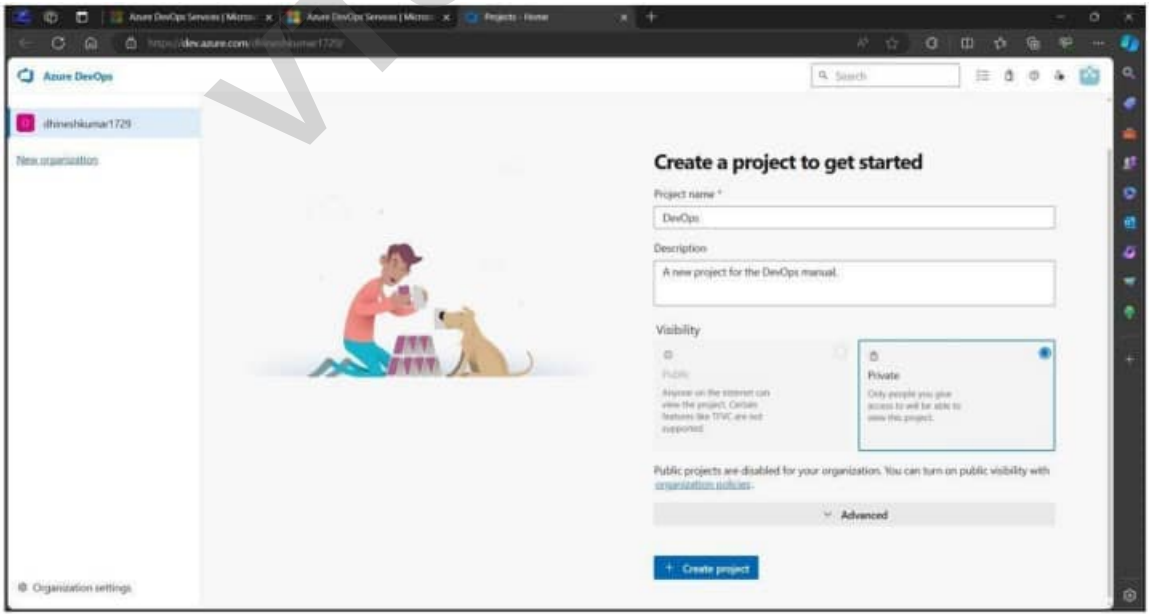
1. **Navigate to "New Project":**
   - On your organization's dashboard, click the **"New Project"** button.
2. **Configure Your Project:**
   - **Project Name:** Enter a descriptive name for your project (e.g., `HelloDevOps`).
   - **Description:** Optionally, provide a brief description (e.g., "A sample project to demonstrate Azure DevOps services").
   - **Visibility:**

- Choose **"Private"** if you want to restrict access to your project.
- Choose **"Public"** if you are okay with the project being accessible to anyone.

- **Advanced Options (Optional):** You can choose a version control system (Git is the default) and a work item process (Agile, Scrum, or Basic). For most beginners, the defaults are recommended.
- Click **"Create"**.

## Step 2: Explore Your Project Dashboard

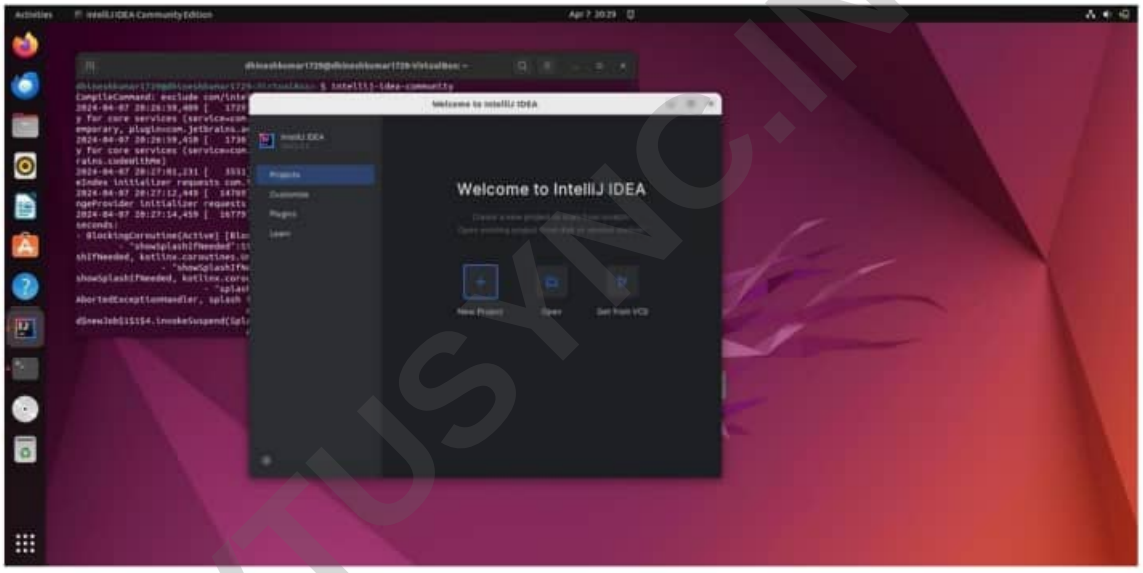
1. **Project Overview:**
   - Once your project is created, you will be directed to the project dashboard. Here you will see navigation options for:
     - **Repos:** Where your code is stored.
     - **Pipelines:** For build and release automation.
     - **Boards:** For work tracking and agile planning.
     - **Test Plans:** For managing and running tests.
     - **Artifacts:** For hosting packages.
2. **Familiarize Yourself with the Interface:**
   - Click through each section (e.g., Repos, Pipelines, Boards) to get a sense of the available features.



## Experiment 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

### Step 1: Installing IntelliJ IDEA

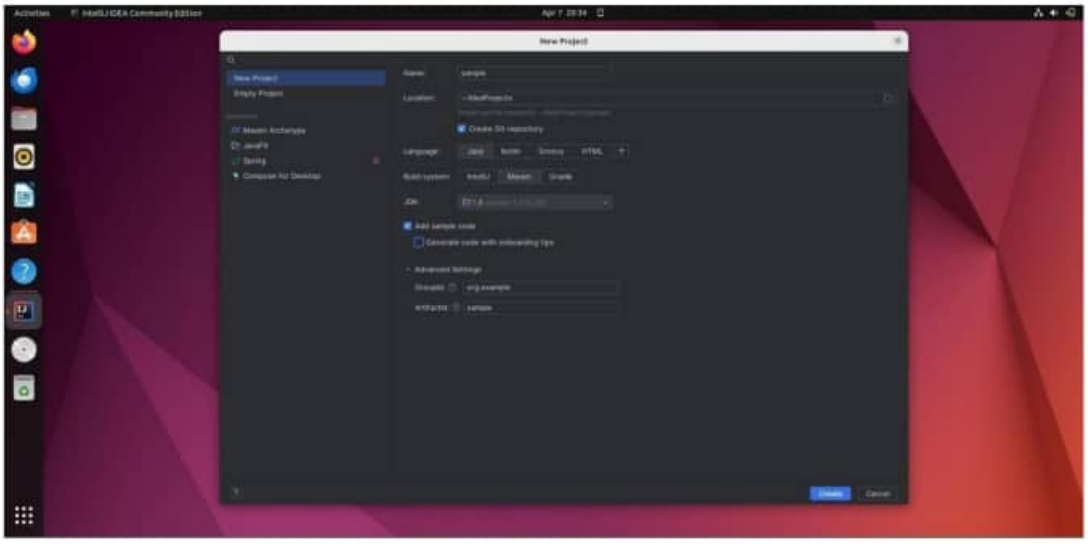
1. Download IntelliJ Idea from https://www.jetbrains.com/idea/download/?section=linux and activate a free trial for 30 days or through the following command. $ sudo snap install intellij-idea-community --community
2. Open the IDE using the below command in the terminal. $ intellij-idea-community
3. Java is a pre-requisite for using this IDE





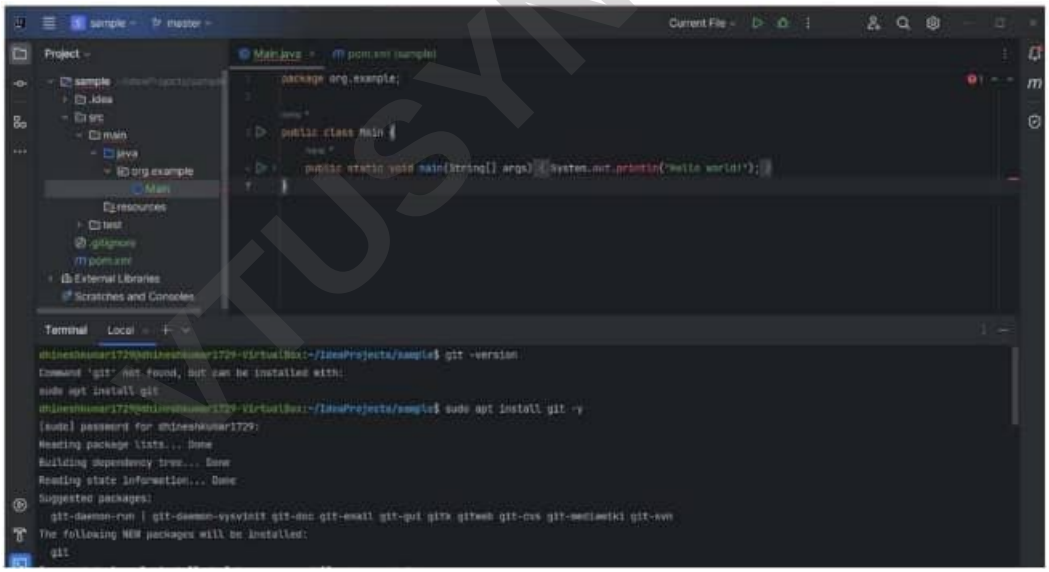

### Step 2: Create a new Maven project using the IDE.

Create a project by clicking on the New Project option, choose an appropriate name and select the following features as mentioned in the image (Build system: Maven andLanguage: Java)



## Step 3: Install GIT and configure it

1. Check whether git is installed using the following command. $ git -version
2. Install git using $ sudo apt install git




3. Once git has been installed execute the following commands one by one.

$ git init $

git add .

$ git commit -m “First Commit”

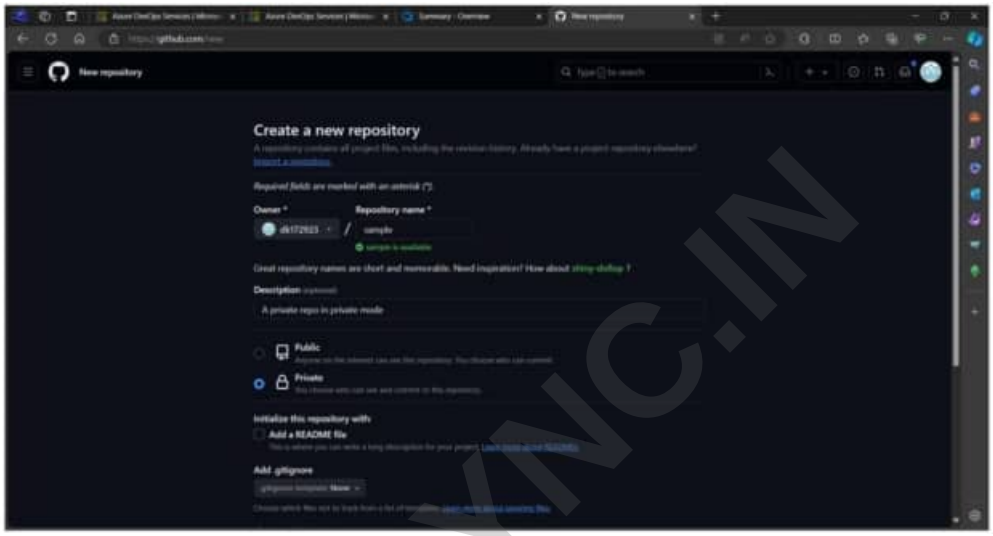
$ git branch -M main

Configure git using the below commands

$ git config --global user.name “Your_Name”

$ git config --global user.email “Your_Email_ID”

Go to https://github.com sign in with your account and create a new private repository with the same name as your maven project.




1. Copy the SSH URL for the created repository.

2. Now go back to IDE’s terminal and execute the following commands

$ ssh-keygen -t rsa -b 4096 -C youremail@gmail.com

(press ENTER for all questions)

$ cat ~/.ssh/id_rsa.pub (copy the printed SSH key)

1. Now go to the SSH and GPG section in the GitHub settings option.

2. Click on the New SSH key button.

3. Choose a suitable name and paste the SSH key into the provided space.

4. Now get back to the IDE’s terminal and type the following command

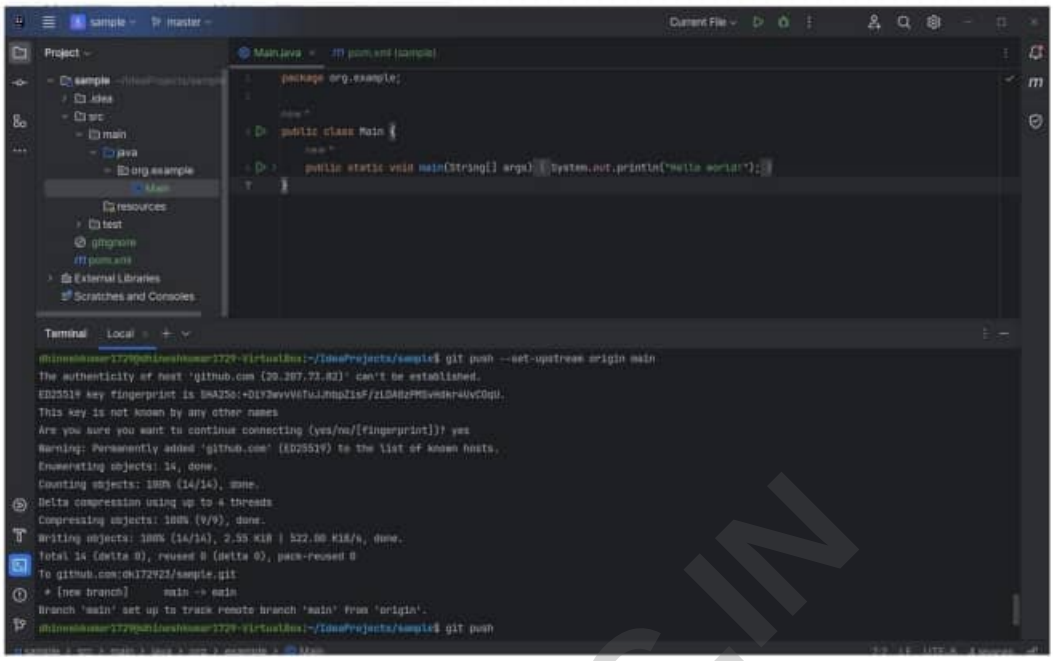
$ git remote set-url origin git@github.com:repo_owner/repo_name.git

(Here, git@github.com:dk172923/sample.git is the remote URL to use SSH dk172923 – GitHub ID sample.git – repository name)

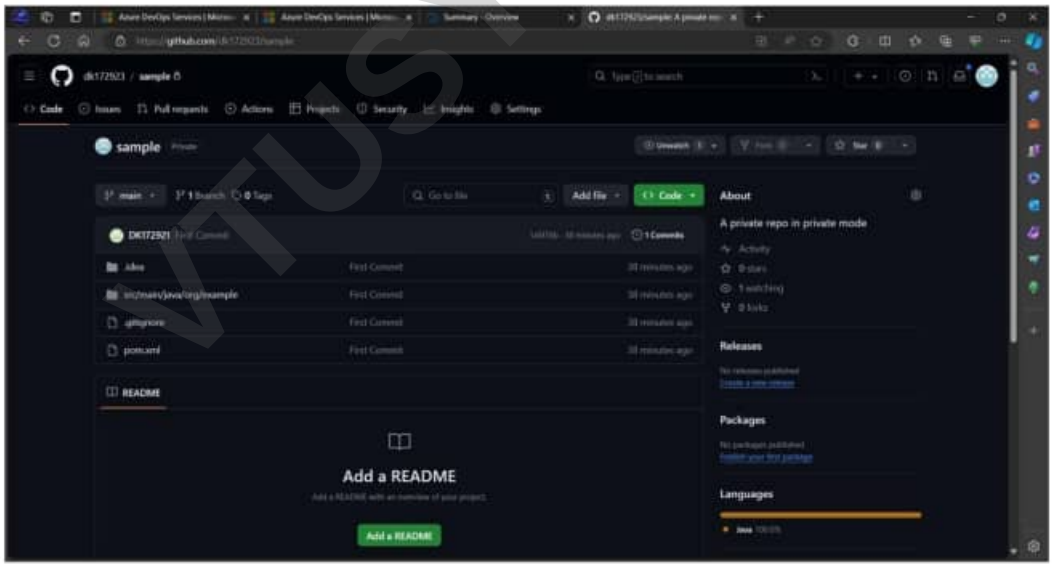
Now finally run the push commands to push the code to the remote repository from the local repository.

$ git push --set-upstream origin main (Type YES for prompted question)

$ git push

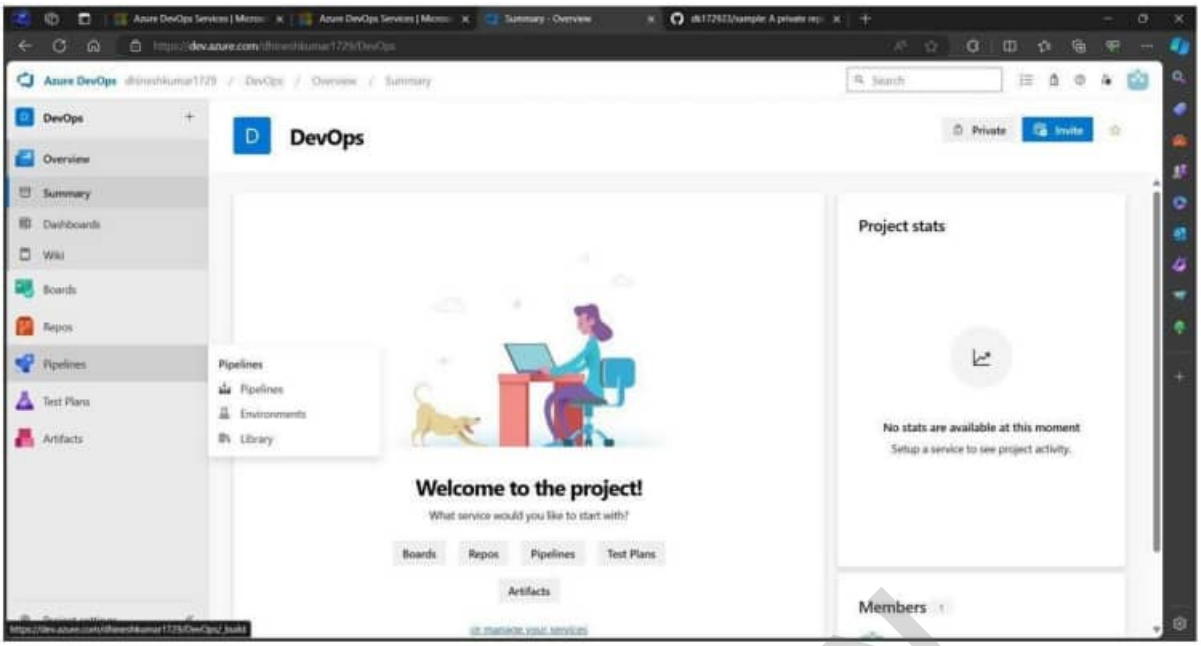



The code has been successfully pushed to the remote repository.



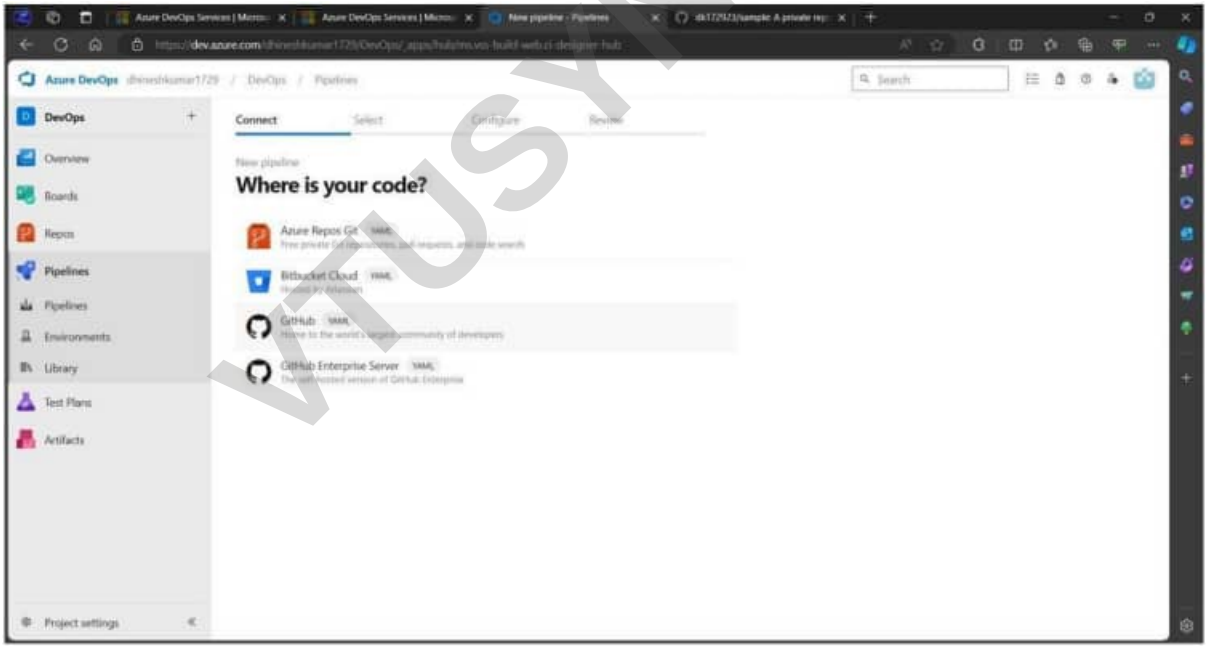

STEP 5: Create a pipeline for the created maven repository.

Click on Pipelines on the left pane of the Azure DevOps website.



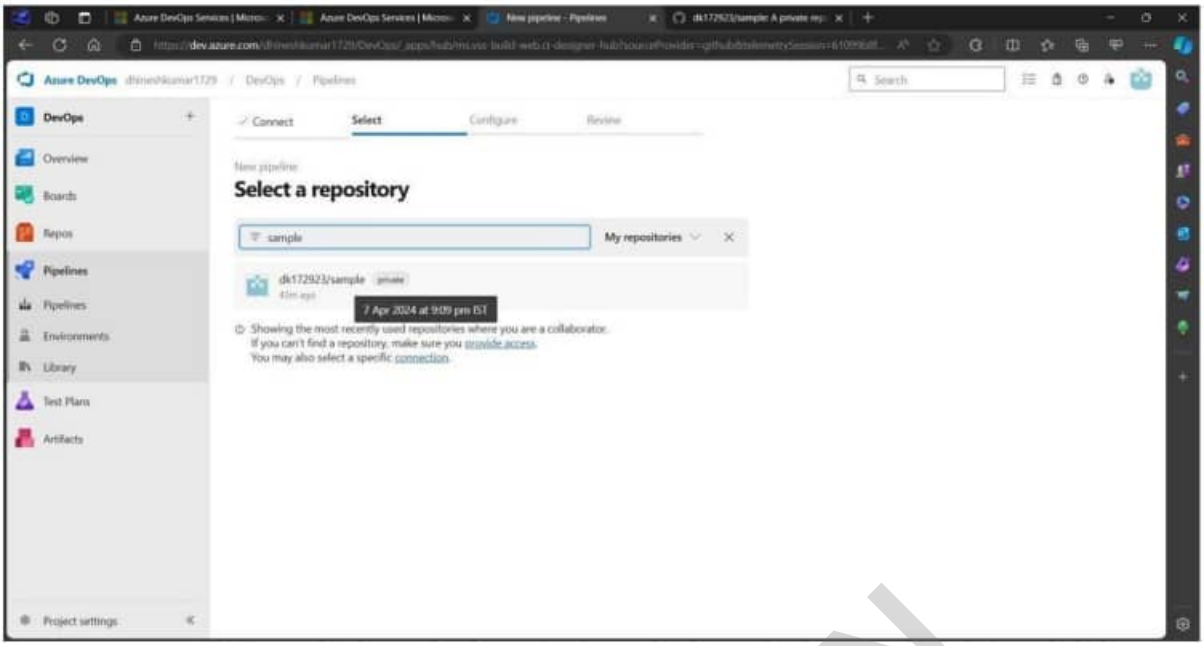

1. Click on the Create Pipeline button.

2. Choose the GitHub YAML option.



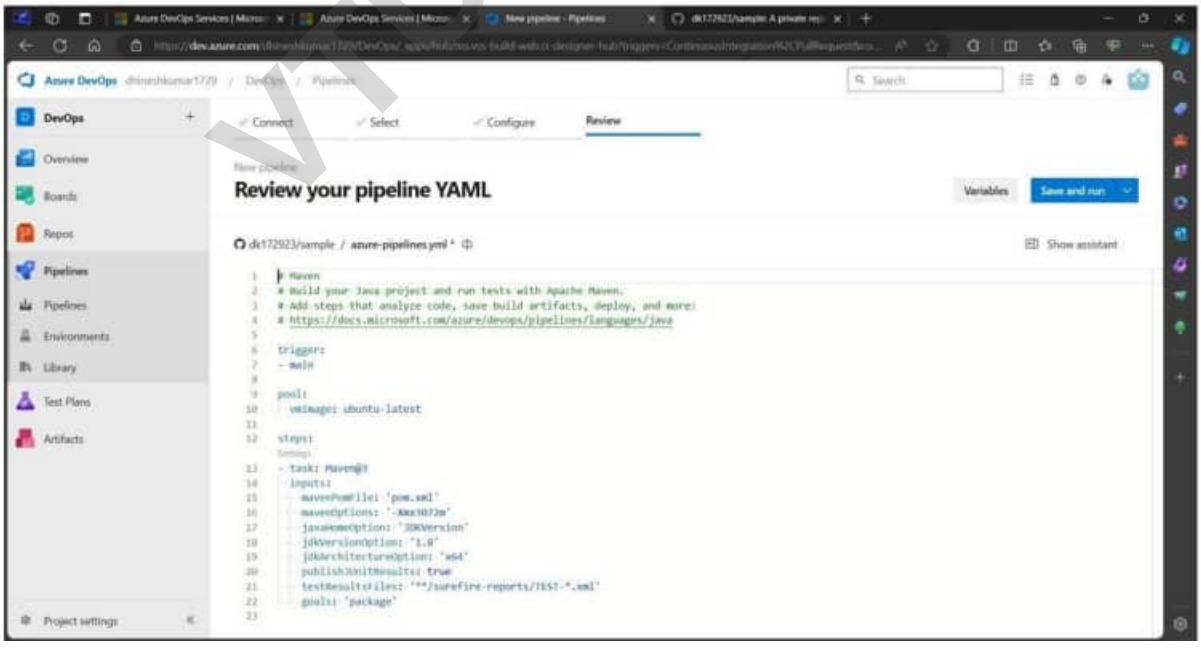

1. Select the created repository “sample” from the available repositories.

2. Give permissions if prompted to do so and sign in to your GitHub account if needed.




3. Choose Maven Pipeline from the given options.

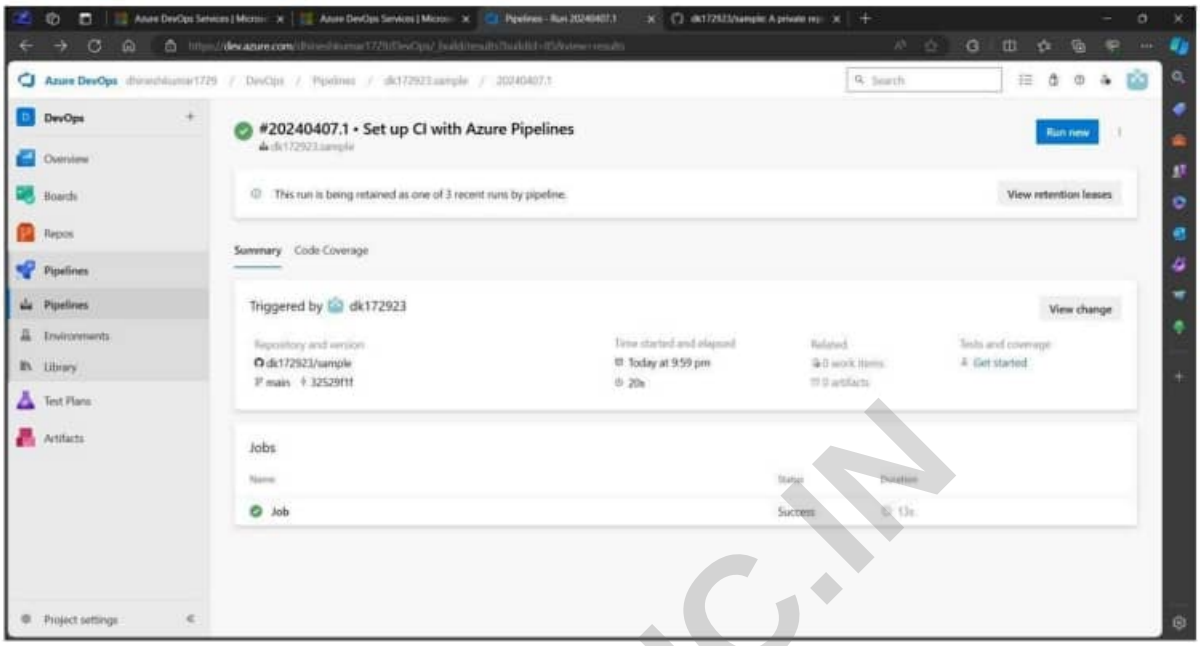
4. A YAML file is automatically created based on the configuration details found in the pom.xml file in the repository.



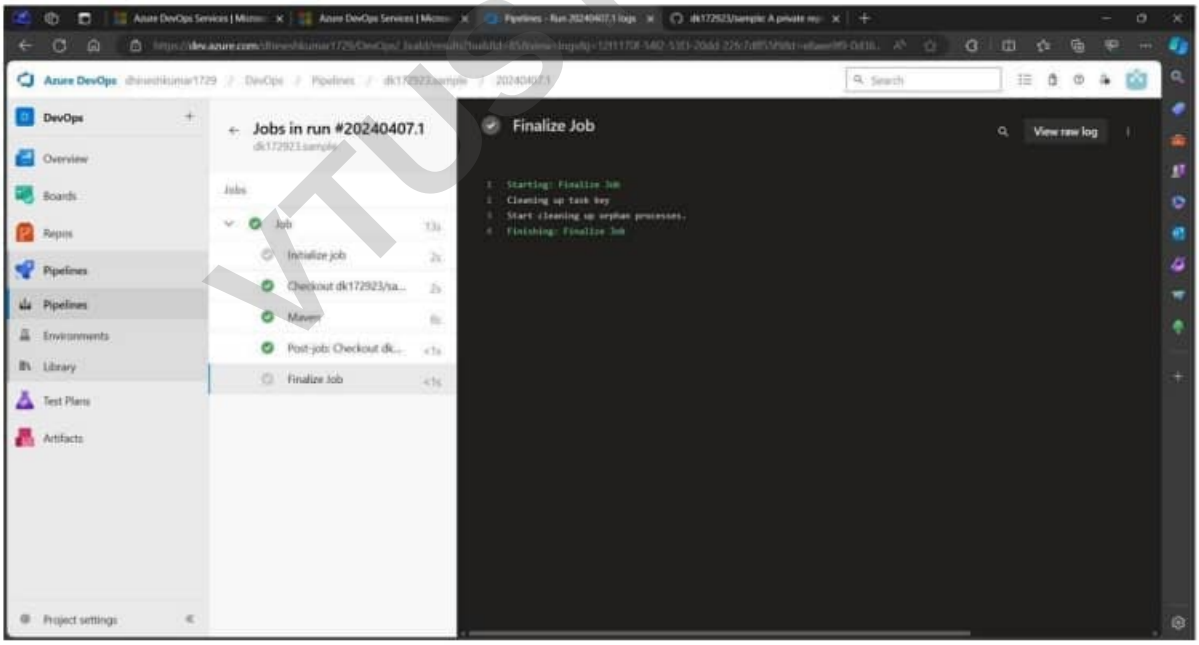

1. Press the Save and Run button.

2. Create a commit message and press the run button again.

3. Click on the created Job from the Jobs table.




A successful job completion would be like this.




Below is a detailed section that explains how to run unit tests and generate reports in an Azure DevOps pipeline after your Maven project has been built. This section explains what happens

during the Maven build process, how test results are generated, and how to publish these results as part of your pipeline execution.

**Running Unit Tests and Generating Reports with Maven in Azure DevOps**

When you build your Maven project using Azure Pipelines, the build process usually includes running unit tests with the Maven Surefire plugin. This plugin executes tests (typically written with JUnit) and produces test result files in XML format. Azure Pipelines can then pick up these XML files and present them as part of the build summary. Below are the steps and details to ensure that your unit tests are executed and the reports are published.

**1. Maven Surefire Plugin and Test Reports**

**What Happens During the Maven Build?**

- **Maven Surefire Plugin:**
  When you run the command mvn clean package (or mvn test), the Surefire plugin automatically executes the unit tests found in the src/test/java directory.
- **Test Report Generation:**
  By default, the Surefire plugin creates XML reports in the target/surefire-reports/ directory. These files usually have names like TEST-<TestClassName>.xml.
- **Importance of Test Reports:**
  These XML files contain detailed information on test execution, including the number of tests run, passed, failed, and any error messages or stack traces.

**2. Configuring Your Azure Pipeline to Publish Test Results**After your Maven build runs and tests are executed, you need to add a step in your Azure Pipeline YAML file that locates these test reports and publishes them in Azure DevOps. This is accomplished by using the **PublishTestResults@2** task.

**YAML Configuration Example for Maven**

Below is a sample snippet of a YAML pipeline configuration that includes both the Maven build step and a step to publish test results:

trigger:

```
  - main

pool:
  vmImage: 'ubuntu-latest'

steps:
  # Step 1: Maven Build and Test Execution
  - task: Maven@3
    inputs:
      mavenPomFile: 'pom.xml'  # Ensure your pom.xml is at the repository root
      goals: 'clean package'
      options: "              # Add any additional Maven options if needed

  # Step 2: Publish Unit Test Results
  - task: PublishTestResults@2
    inputs:
      testResultsFiles: '**/target/surefire-reports/TEST-*.xml'
      mergeTestResults: true
      testRunTitle: 'Maven Unit Test Results'
```

**Explanation of the YAML Steps:**

1. **Maven Build Step:**
    - **Task:** Maven@3
    - **Input:**
        - **mavenPomFile:** Points to your pom.xml.
        - **goals:** The goals clean package ensure that your code is cleaned, compiled, tested, and packaged. The Surefire plugin runs tests during this process.
    - **Outcome:** Maven executes the tests and creates XML reports under the target/surefire-reports/ folder.
2. **Publish Test Results Step:**
    - **Task:** PublishTestResults@2
    - **Input:**
        - **testResultsFiles:** Uses a glob pattern (**/target/surefire-reports/TEST-*.xml) to find all XML test result files.
        - **mergeTestResults:** Set to true so that if there are multiple result files, they are merged into a single report.
        - **testRunTitle:** Provides a title for the test run that will appear in the Azure DevOps test results view.
    - **Outcome:** Azure Pipelines reads the test result XML files and displays a summary in the pipeline's **Tests** tab. You'll be able to see metrics like the number of tests passed, failed, or skipped, and review detailed error messages for any failures.



**3. Running and Verifying the Pipeline**

After committing the YAML file to your repository, the pipeline is triggered (either automatically or manually):

1. **Trigger the Pipeline:**
    - Once your YAML file is saved in your repository, Azure Pipelines will pick up the changes. If not automatically triggered, you can click **"Run pipeline"** manually.
    - **Screenshot Tip:** Capture the pipeline run screen where you see the stages (e.g., Checkout, Maven Build, Publish Test Results).
2. **Monitor the Build Output:**

- Navigate to the **"Build"** or **"Logs"** section of the pipeline run.
- Confirm that the Maven task logs show the execution of tests and that the Surefire reports are created.

3. **Review Test Reports:**
   - Once the build completes, click on the **"Tests"** tab (often found on the pipeline summary page) to review the detailed test results.
   - **Screenshot Tip:** Capture the test report summary, showing the number of tests executed, passed, and failed. You may also capture detailed logs for any failed tests.

## 4. Troubleshooting Tips

- **No Test Results Found:**
  If you see a warning that no test results were found, verify that:
  - Your tests are indeed being executed (check the Maven build log).
  - The path in testResultsFiles correctly matches the location and naming pattern of the generated XML files.
- **Test Failures:**
  If tests fail, review the published test report for detailed error messages and stack traces. Adjust your tests or code as necessary and re-run the pipeline.

# Experiment 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

## 1. Overview

In this experiment, you will:

- **Deploy your build artifact** (e.g., a JAR or WAR file from a Maven/Gradle project) to an Azure App Service.
- **Manage secrets and configuration** securely using Azure Key Vault.
- **Set up a release pipeline** in Azure DevOps that automatically deploys your application when a new artifact is available (continuous deployment).

This experiment bridges the gap between build automation (CI) and release automation (CD) while ensuring secure management of sensitive information.



## 2. Prerequisites

Before you begin, ensure that you have:

- An **Azure DevOps account** with a project set up (see Experiment 9 and Experiment 10).
- A **build artifact** (e.g., your Maven/Gradle artifact) available from a build pipeline.
- An **Azure Subscription** with an **Azure App Service** instance already created to host your application.
- An **Azure Key Vault** instance created in your Azure Subscription for storing secrets (e.g., connection strings, API keys).
- Appropriate permissions to create and manage resources in Azure DevOps and your Azure Subscription.

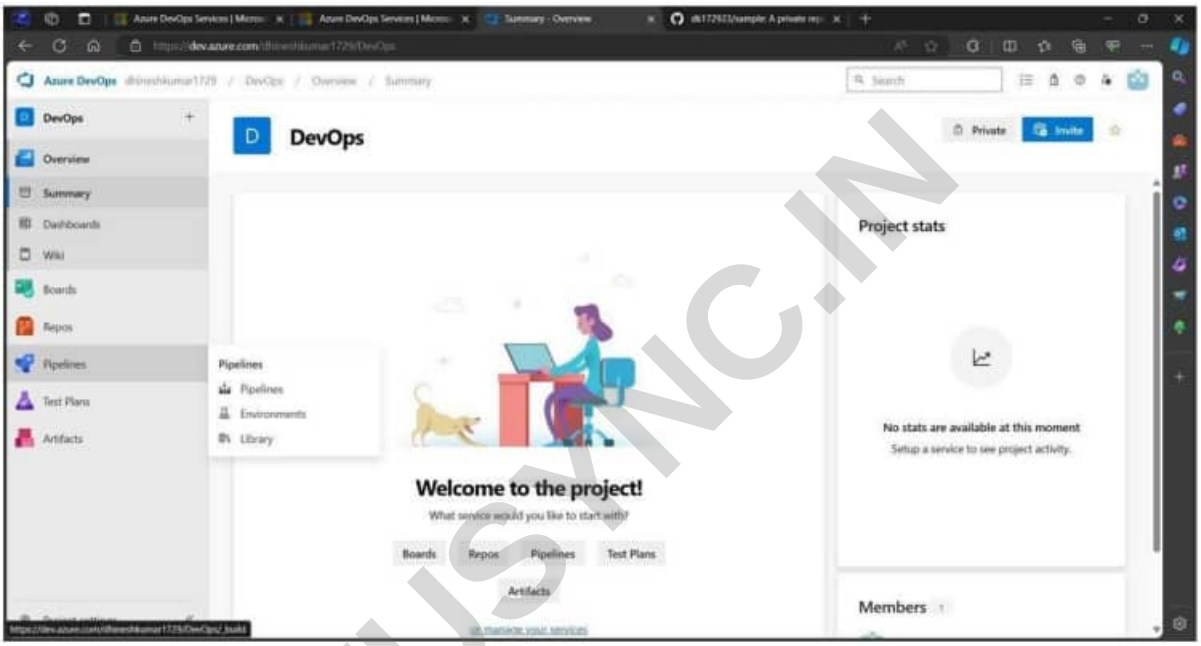
## 3. Creating a Release Pipeline in Azure DevOps

### A. Create a New Release Pipeline

1. **Log in to Azure DevOps:**

- Open your web browser and navigate to your Azure DevOps project (e.g., https://dev.azure.com/YourOrganization).

2. **Navigate to the Releases Section:**
   - In the left-hand menu, click on **"Pipelines"** and then **"Releases"**.
3. **Create a New Pipeline:**
   - Click on **"New pipeline"**.
   - When prompted, select **"Empty job"** (or start with a template if one suits your needs).






**B. Add an Artifact**

1. **Link Your Build Artifact:**
   - In the release pipeline editor, click on **"Add an artifact"**.
   - **Source Type:** Choose the source (e.g., "Build").
   - **Source (Build Pipeline):** Select the build pipeline that produces your Maven/Gradle artifact.
   - **Default Version:** Use the latest version (or specify a branch/tag as needed).
   - Click **"Add"**.

**C. Define a Stage for Deployment**

1. **Add a New Stage:**

- Click on **"Add a stage"** and select **"Empty job"**.
- Rename the stage (e.g., Development or Production).

2. **Configure the Stage:**
   - Click on the stage to open its settings.
   - In the **"Tasks"** view for the stage, click **"+"** to add a new task.

**4. Deploying to Azure App Services**

**A. Add the Azure App Service Deploy Task**

1. **Add Deployment Task:**
   - In the stage tasks, click on **"+ Add"** and search for **"Azure App Service Deploy"**.
   - Click **"Add"** to include it in your stage.
2. **Configure the Azure App Service Deploy Task:**
   - **Azure Subscription:**
     - Click on **"Authorize"** (if required) and select your Azure subscription.
   - **App Service Type:** Choose **"Web App on Windows"** or **"Web App on Linux"** based on your deployment target.
   - **App Service Name:** Enter the name of your Azure App Service instance.
   - **Package or Folder:**
     - Specify the path to your build artifact (for example, $(System.DefaultWorkingDirectory)/_HelloMaven-CI/drop/HelloMaven-1.0-SNAPSHOT.jar or the folder containing your package). Adjust the path based on your artifact's location.
   - **Deployment Options:** Configure any additional options (e.g., deployment slot, resource group) as needed.

**5. Managing Secrets and Configuration with Azure Key Vault**

**A. Create and Configure an Azure Key Vault (in Azure Portal)**

1. **Create a Key Vault (if not already created):**
   - Log in to the Azure Portal.
   - Click on **"Create a resource"** and search for **"Key Vault"**.

- Follow the prompts to create a new Key Vault (enter a name, select subscription, resource group, and region).

2. **Add Secrets to Your Key Vault:**
   - Once the Key Vault is created, navigate to it.
   - Click on **"Secrets"** and then **"Generate/Import"**.
   - Create new secrets (e.g., DBConnectionString, APIKey) and note their names.

**B. Integrate Key Vault with Azure DevOps**

1. **Create a Variable Group Linked to Key Vault:**
   - In your Azure DevOps project, navigate to **"Pipelines"** and then **"Library"**.
   - Click on **"+ Variable group"**.
   - Name the variable group (e.g., KeyVault-Secrets).
   - Enable **"Link secrets from an Azure Key Vault as variables"**.
   - Click **"Authorize"** and select your Azure subscription.
   - Select your Key Vault from the dropdown list.
   - Choose the secrets you want to import into the variable group.
   - Save the variable group.
2. **Use the Variable Group in Your Release Pipeline:**
   - In your release pipeline, click on the **"Variables"** tab.
   - Click **"Variable groups"** and then **"Link variable group"**.
   - Select the variable group you created (KeyVault-Secrets).
   - Now, you can reference these secrets as variables in your deployment tasks (for example, $(DBConnectionString)).



**6. Enabling Continuous Deployment**

**A. Configure Continuous Deployment Trigger**

1. **Enable Continuous Deployment:**
   - In your release pipeline, click on the **"Triggers"** tab.
   - Under **"Artifact filters"** or **"Continuous deployment trigger"**, enable the toggle to allow the release pipeline to trigger automatically when a new build artifact is available.
   - Optionally, you can specify branch filters or artifact version filters.

2. **Save the Release Pipeline:**
    - Click **"Save"** to apply all changes.

**B. Test the Continuous Deployment Pipeline**

1. **Trigger a Build:**
    - Commit a change to your code repository to trigger your build pipeline (or trigger it manually).
    - Verify that the build pipeline creates a new artifact.
2. **Automatic Release:**
    - Once the new artifact is published, the release pipeline should automatically trigger a new release.
    - Monitor the release pipeline execution to ensure that:
        - The artifact is deployed to the Azure App Service.
        - The deployment task uses the configuration and secrets from Key Vault.

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

## 1. Overview

In this experiment, you will create an end-to-end DevOps pipeline that demonstrates the following processes:

- **Version Control:** Code is maintained in a Git repository (e.g., GitHub or Azure Repos).
- **Continuous Integration (CI):**
  - A CI tool (Jenkins and/or Azure Pipelines) automatically checks out the code, builds it using Maven/Gradle, runs unit tests, and archives the build artifact.
- **Artifact Management:** The artifact (e.g., a JAR file) is archived and made available for deployment.
- **Continuous Deployment (CD):**
  - Deployment automation is handled either by an Ansible playbook or an Azure Release pipeline, deploying the artifact to a target environment (such as an Azure App Service or a local server).
- **Secrets and Configuration Management:** Securely manage configuration details and secrets using Azure Key Vault.
- **Pipeline as Code:** Use YAML (for Azure Pipelines) or a Jenkinsfile (for Jenkins) to define your build and release processes.

After setting up and running the pipeline, we will discuss best practices for designing and maintaining such pipelines and open the floor for a Q&A discussion.

## 2. Prerequisites

Before starting, ensure you have completed or have access to the following:

- **Source Code Repository:**
  - A Java project (e.g., “HelloMaven” or “HelloGradle”) hosted on GitHub or Azure Repos.

- The project should follow a standard structure (with pom.xml for Maven or build.gradle for Gradle, and appropriate src/main/java and src/test/java directories).

- **Jenkins and/or Azure DevOps Setup:**
  - Jenkins installed and configured on your local machine or a cloud server (refer to Experiments 5 and 6), or an Azure DevOps project set up with a build pipeline (Experiments 9 and 10).
- **Ansible Installed:**
  - Ansible is installed on your control machine (or Jenkins server) with a basic inventory file (see Experiment 7).
- **Azure Resources:** (Optional but recommended for cloud deployment)
  - An Azure App Service instance created to host your application.
  - An Azure Key Vault instance set up to store sensitive data (e.g., connection strings).
- **Access Credentials:**
  - Permissions to commit code to the repository.
  - Administrative access on Jenkins/Azure DevOps.
  - Proper permissions on Azure to deploy to App Services and to manage Key Vault secrets.

**3. Step-by-Step Pipeline Setup**

**Step 1: Code Repository Preparation**

1. **Ensure Your Project Is in Version Control:**
   - Navigate to your project folder (e.g., “HelloMaven”) and initialize Git if not already done:
   - cd /path/to/HelloMaven
   - git init
   - git add .
   - git commit -m "Initial commit of HelloMaven project"
   - Push the project to your remote repository (e.g., GitHub):
   - git remote add origin https://github.com/yourusername/HelloMaven.git
   - git push -u origin main

**Step 2: Continuous Integration with Jenkins and/or Azure Pipelines**

**A. Jenkins CI Pipeline Setup**

1. **Create a New Jenkins Job:**
   - Log in to Jenkins and click **"New Item"**.
   - Enter a name (e.g., HelloMaven-CI) and select **"Freestyle project"** or **"Pipeline"**.
   - *Screenshot Tip:* Capture the new job creation screen.
2. **Configure Source Code Management:**
   - Under the **"Source Code Management"** section, select **"Git"**.
   - Enter your repository URL (e.g., https://github.com/yourusername/HelloMaven.git) and set branch specifier to */main.
3. **Add Build Steps:**
   - **For Maven Projects:**
     - Add a build step "Invoke top-level Maven targets" and set the goals to:
     - clean package
   - **For Pipeline-as-Code (Jenkinsfile):**
     - Create a Jenkinsfile in your repository with stages for checkout, build, test, and archive. For example:
     - pipeline {
     - agent any
     - stages {
     - stage('Checkout') {
     - steps {
     - git url: 'https://github.com/yourusername/HelloMaven.git', branch: 'main'
     - }
     - }
     - stage('Build') {
     - steps {
     - sh 'mvn clean package'
     - }

```
        }
        stage('Test') {
            steps {
                sh 'mvn test'
            }
        }
        stage('Archive') {
            steps {
                archiveArtifacts artifacts: 'target/*.jar', fingerprint: true
            }
        }
    }
}
```

4. **Run the Jenkins Job:**
    - Click **"Build Now"** and monitor the console output. Ensure that the build is successful and that test reports are generated.

**B. Azure DevOps Build Pipeline Setup**

1. **Create a New Pipeline:**
    - Log in to your Azure DevOps project.
    - Navigate to **"Pipelines"** and click **"New pipeline"**.
    - Select your repository source (GitHub or Azure Repos) and choose your repository.
2. **Define Your YAML Pipeline:**
    - Use the following sample YAML for a Maven project:
    - trigger:
    - - main
    -
    - pool:
    - vmImage: 'ubuntu-latest'
    -
    - steps:
    - - task: Maven@3

- inputs:
- mavenPomFile: 'pom.xml'
- goals: 'clean package'
- - task: PublishTestResults@2
- inputs:
- testResultsFiles: '**/target/surefire-reports/TEST-*.xml'
- mergeTestResults: true
- testRunTitle: 'Maven Unit Test Results'

3. **Run the Pipeline and Verify Test Reports:**
   - Commit and run the pipeline.
   - Navigate to the **"Tests"** tab to view the summary of executed tests

**Step 3: Artifact Management**

1. **Artifact Archiving in Jenkins:**
   - Ensure your Jenkins job archives the artifact (JAR file) using the **"Archive the artifacts"** post-build action.
2. **Artifact in Azure Pipelines:**
   - The build task produces an artifact that can be downloaded or referenced by subsequent release pipelines.

**Step 4: Deployment Automation with Ansible and Azure Release Pipeline**

**A. Using Ansible for Deployment**

1. **Write an Ansible Playbook:**
   - Create a file named deploy.yml:
   - ---
   - - name: Deploy Maven Artifact
   - hosts: deployment
   - become: yes
   - tasks:
   - - name: Copy the artifact to the target directory
   - copy:

- src: "/var/lib/jenkins/workspace/HelloMaven-CI/target/HelloMaven-1.0-SNAPSHOT.jar"
- dest: "/opt/deployment/HelloMaven.jar"
- Adjust paths according to your environment.

2. **Configure Your Ansible Inventory:**
   - Create or update your hosts.ini file:
   - [deployment]
   - target-server ansible_host=your.server.ip ansible_user=yourusername
   - For a local deployment, you can use:
   - [deployment]
   - localhost ansible_connection=local
3. **Integrate Ansible into Your Jenkins/Azure Pipeline:**
   - Add a post-build (or post-release) step to execute the Ansible playbook:
   - ansible-playbook -i /path/to/hosts.ini /path/to/deploy.yml

**B. Using Azure Release Pipeline to Deploy to Azure App Services**

1. **Create a New Release Pipeline in Azure DevOps:**
   - Navigate to **"Pipelines > Releases"**.
   - Click **"New pipeline"** and select an empty job.
2. **Link Your Build Artifact:**
   - Click **"Add an artifact"** and select your build pipeline as the source.
3. **Add a Deployment Stage for Azure App Service:**
   - Create a new stage (e.g., Production).
   - Add the **"Azure App Service Deploy"** task.
   - Configure the task with your Azure subscription, target App Service, and the path to the artifact.
4. **Integrate Key Vault (Optional for Managing Secrets):**
   - Create and link a Variable Group that pulls secrets from Azure Key Vault (refer to previous experiments).
   - Reference these secrets in your deployment tasks (e.g., connection strings).
5. **Enable Continuous Deployment:**
   - In the release pipeline's triggers, enable continuous deployment so that a new release is automatically created when a new artifact is available.

6. **Run the Release Pipeline and Verify Deployment:**
    - Trigger the release pipeline (either manually or automatically).
    - Verify that the application is deployed to your target environment (e.g., by browsing to the Azure App Service URL or checking the deployment directory on the target server).

**Step 5: End-to-End Pipeline Demonstration**

1. **Trigger a Complete Run:**
    - Make a change in your code repository (e.g., modify a welcome message in your Java application) and commit it.
    - This should trigger the CI pipeline (Jenkins or Azure Pipelines), which builds, tests, and archives the artifact.
    - The artifact triggers the CD process (via Ansible or Azure Release), and the application is deployed automatically.
2. **Verify the Entire Workflow:**
    - Check the CI pipeline output (build success and test results).
    - Confirm that the artifact is archived.
    - Review the CD pipeline logs (deployment success, any post-deployment notifications, etc.).
    - Optionally, log into the target environment and verify that the new version of your application is running.

**Step 6: Discussion on Best Practices and Q&A**

**Best Practices:**

- **Pipeline as Code:**
  Use YAML (or a Jenkinsfile) to define your build and release pipelines. This allows you to version control your pipeline configuration alongside your code.
- **Automate Everything:**
  Automate code checkout, builds, tests, artifact archiving, and deployments. Reduce manual interventions to minimize human error.

- **Idempotence:**
  Ensure that your deployment scripts (whether Ansible playbooks or release tasks) are idempotent—running them multiple times produces the same result.
- **Secure Secrets Management:**
  Use Azure Key Vault (or a similar tool) to securely store sensitive data (e.g., API keys, connection strings) and reference these values in your pipelines.
- **Monitoring and Logging:**
  Integrate logging and monitoring into your pipeline. Review test reports, deployment logs, and set up notifications for build failures.
- **Modular and Scalable Design:**
  Break down your pipeline into clear stages (checkout, build, test, deploy) and design it to handle multi-environment deployments (development, staging, production).
- **Continuous Improvement:**
  Regularly review and refine your pipeline. Use metrics and feedback to optimize build times, reduce failures, and ensure high-quality releases.

**Q&A Discussion Points:**

- **Troubleshooting Pipeline Failures:**
  What are common reasons for build/test failures, and how can you diagnose issues from the logs?
- **Handling Rollbacks:**
  What strategies can be implemented for rolling back deployments if a release fails in production?
- **Scaling Pipelines:**
  How do you handle scaling the pipeline for larger projects or multiple microservices?
- **Integrating Additional Tools:**
  What additional tools (e.g., code quality analyzers, security scanners) could be integrated into this pipeline?
- **Real-World Challenges:**
  Discussion of challenges faced in actual DevOps implementations and strategies to overcome them.

*Open the Floor for Questions:*

Encourage participants to ask questions regarding any part of the pipeline—from technical configuration details to high-level best practices.