Project is from Page 16

Maven

Maven is a project management and comprehension tool that provides developers with a comprehensive and standardized way to build projects, manage dependencies, and generate documentation. It uses an XML file, `pom.xml`, to manage the project's configuration.

Key Concepts

1. **Project Object Model (POM)**: The fundamental unit of Maven is the POM, which is defined in the `pom.xml` file. This file contains information about the project and configuration details used by Maven to build the project.
2. **Dependencies**: Maven manages the libraries and plugins your project depends on. It downloads them from a repository and includes them in your project during the build process.
3. **Goals**: Goals are specific tasks that Maven performs, like compiling code, running tests, or packaging the project into a JAR or WAR file.

Structure of `pom.xml`

The `pom.xml` file is the heart of a Maven project. Here's a detailed breakdown of a typical `pom.xml` file:

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

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

    <name>My App</name>
    <description>A simple Maven project</description>
    <url>http://example.com/my-app</url>

    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>
```

```
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.3.8</version>
        </dependency>

<!-- https://mvnrepository.com/artifact/org.apache.kafka/kafka-clients -->
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>3.7.1</version>
</dependency>


        <!-- Additional dependencies can be added here -->
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <!-- Additional plugins can be added here -->
        </plugins>
    </build>

</project>
```

Explanation of `pom.xml` Elements

- `<project>`: The root element of the POM file.
- `<modelVersion>`: The model version of the POM; always set to 4.0.0.

- `<groupId>`: The group or organization the project belongs to, usually a reversed domain name (e.g., com.example).
- `<artifactId>`: The name of the project.
- `<version>`: The version of the project.
- `<packaging>`: The type of artifact (e.g., jar, war).
- `<name>`: The name of the project.
- `<description>`: A brief description of the project.
- `<url>`: The URL where the project can be found.
- `<properties>`: Custom properties for the project, such as compiler settings.
- `<dependencies>`: The external libraries your project depends on.
- `<build>`: Configuration for building the project, including plugins and their configurations.

Dependencies

Dependencies are libraries or modules required by your project to function. Maven manages these by downloading them from a central repository and including them in your project.

Example of Adding a Dependency

To add a dependency, include it in the `<dependencies>` section:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>5.3.8</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Goals

Goals are specific tasks that Maven executes. They are part of plugins. For instance, the `maven-compiler-plugin` has a goal named `compile`.

Common Goals

- `compile`: Compiles the source code.
- `test`: Runs the tests using a testing framework.
- `package`: Packages the compiled code into a JAR or WAR file.

- `install`: Installs the packaged code into the local repository.
- `deploy`: Deploys the packaged code to a remote repository.

Example of Using Goals

To run a goal, you can use the `mvn` command:

```
mvn compile
mvn test
mvn package
```

Use Cases

1. **Simple Java Application**: A basic project that compiles Java code, runs tests, and packages it into a JAR file.
2. **Web Application**: Building and packaging a web application into a WAR file, including dependencies like Spring and Hibernate.
3. **Multi-Module Project**: Managing a project with multiple modules, each having its own POM but inheriting from a parent POM.
4. **Continuous Integration**: Integrating with CI/CD tools like Jenkins, which can trigger Maven goals to automate the build and deployment process.
5. **Dependency Management**: Managing complex dependency trees, ensuring that the right versions of libraries are used.

Example of a Multi-Module Project

Parent POM (`pom.xml` in the root directory):

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

    <groupId>com.example</groupId>
    <artifactId>my-multi-module-project</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>

    <modules>
        <module>new</module>
        <module>module2</module>
    </modules>
</project>
```

Module 1 POM (`new/pom.xml`):

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

    <parent>
        <groupId>com.example</groupId>
        <artifactId>my-multi-module-project</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <artifactId>module1</artifactId>

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

Module 2 POM (`module2/pom.xml`):

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

```
    <parent>
        <groupId>com.example</groupId>
        <artifactId>my-multi-module-project</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <artifactId>module2</artifactId>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.3.8</version>
        </dependency>
    </dependencies>
</project>
```

Defining Goals in the Build Section:
In the `<build>` section, you can specify which plugins to use and configure their goals.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-
4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>my-app</artifactId>
    <version>1.0-SNAPSHOT</version>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
                <executions>
```

```
                    <execution>
                        <id>default-compile</id>
                        <goals>
                            <goal>compile</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.22.2</version>
                <executions>
                    <execution>
                        <id>default-test</id>
                        <goals>
                            <goal>test</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

1. In this example, the `maven-compiler-plugin` is configured to compile Java source code, and the `maven-surefire-plugin` is configured to run tests. Each plugin's goals are specified within `<executions>` and `<goals>` elements.

Using Plugin Goals Directly:
Sometimes you may want to run specific plugin goals without explicitly configuring them in the `pom.xml`. You can do this via the command line:

```
mvn clean
mvn compile
mvn test
mvn package
```

2. The above commands will run the default goals for cleaning the project, compiling the source code, running tests, and packaging the compiled code.

More Detailed Plugin Configuration

You can further customize the behavior of goals by adding more detailed configuration within the `<plugin>` element.

xml
Copy code

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-
4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>my-app</artifactId>
    <version>1.0-SNAPSHOT</version>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
                <executions>
                    <execution>
                        <id>default-compile</id>
                        <phase>compile</phase>
                        <goals>
                            <goal>compile</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.22.2</version>
                <executions>
                    <execution>
                        <id>default-test</id>
```

```
                        <phase>test</phase>
                        <goals>
                            <goal>test</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Jenkins Pipeline Overview

Jenkins Pipeline provides a set of powerful tools to model complex delivery pipelines "as code" via the DSL (Domain Specific Language) in either a declarative or scripted format. This allows you to define your build process in a Jenkinsfile that can be version-controlled alongside your source code.

Types of Jenkins Pipelines

1. **Declarative Pipeline**: A more simplified and opinionated way to write pipeline code. It uses a predefined structure to define the pipeline and aims to make it easier for users to get started with Jenkins Pipelines.
2. **Scripted Pipeline**: Offers more flexibility and is written in Groovy, allowing for complex logic and control over the pipeline.

Jenkinsfile

A Jenkinsfile is a text file that contains the pipeline code. It is typically stored in the root directory of the project's repository.

Structure of a Declarative Pipeline

The basic structure of a declarative pipeline includes:

1. **pipeline**: The root element that specifies the start of the pipeline.
2. **agent**: Defines where the pipeline should run (e.g., any available agent or a specific one).
3. **stages**: Contains one or more stages, each representing a phase in the pipeline.
4. **stage**: Defines a specific stage in the pipeline, which contains steps to be executed.
5. **steps**: The actual tasks to be executed in a stage.

Example of a Declarative Pipeline

```
pipeline {
    agent any

    stages {
```

```
        stage('Checkout') {
            steps {
                // Checkout code from Git repository
                git url: 'https://github.com/your-repo.git', branch:
'main'

            }
        }
        stage('Build') {
            steps {
                // Build the project using Maven
                sh 'mvn clean install'
            }
        }
        stage('Test') {
            steps {
                // Run tests using Maven
                sh 'mvn test'
            }
        }
        stage('Deploy') {
            steps {
                // Deploy the application
                sh 'scp target/your-app.jar
user@server:/path/to/deploy'
            }
        }
    }

    post {
        always {
            // Always executed steps
            echo 'This will always run.'
        }
        success {
            // Executed if the pipeline succeeds
            echo 'Pipeline completed successfully.'
        }
        failure {
            // Executed if the pipeline fails
            echo 'Pipeline failed.'
        }
```

```
    }
}
```

Explanation of the Example Pipeline

1. **agent any**: The pipeline can run on any available agent.
2. **stage('Checkout')**: Checks out the source code from the Git repository.
3. **stage('Build')**: Builds the project using Maven.
4. **stage('Test')**: Runs tests using Maven.
5. **stage('Deploy')**: Deploys the built artifact to a remote server using SCP.
6. **post**: Defines actions to be taken after the pipeline completes.
    - **always**: Echoes a message regardless of the pipeline result.
    - **success**: Echoes a success message if the pipeline succeeds.
    - **failure**: Echoes a failure message if the pipeline fails.

Example of a Scripted Pipeline

```
node {
    try {
        stage('Checkout') {
            // Checkout code from Git repository
            git url: 'https://github.com/your-repo.git', branch:
'main'
        }

        stage('Build') {
            // Build the project using Maven
            sh 'mvn clean install'
        }

        stage('Test') {
            // Run tests using Maven
            sh 'mvn test'
        }

        stage('Deploy') {
            // Deploy the application
            sh 'scp target/your-app.jar user@server:/path/to/deploy'
        }

        // If all stages succeed, mark the build as SUCCESS
        currentBuild.result = 'SUCCESS'
    } catch (Exception e) {
        // If any stage fails, mark the build as FAILURE
```

```
        currentBuild.result = 'FAILURE'
        throw e
    } finally {
        // Post-build actions
        if (currentBuild.result == 'SUCCESS') {
            echo 'Pipeline completed successfully.'
        } else {
            echo 'Pipeline failed.'
        }
    }
}
```

Explanation of the Example Scripted Pipeline

1. **node**: The pipeline will run on any available agent.
2. **stage('Checkout')**: Checks out the source code from the Git repository.
3. **stage('Build')**: Builds the project using Maven.
4. **stage('Test')**: Runs tests using Maven.
5. **stage('Deploy')**: Deploys the built artifact to a remote server using SCP.
6. **try**: Starts a block to handle potential errors in the pipeline.
7. **currentBuild.result = 'SUCCESS'**: Marks the build as successful if all stages succeed.
8. **catch (Exception e)**: Catches any exceptions thrown during the pipeline execution, marks the build as failed, and rethrows the exception.
9. **finally**: Ensures that the post-build actions are executed regardless of whether the build succeeds or fails.
10. **if (currentBuild.result == 'SUCCESS')**: Executes actions if the build succeeds.
11. **else**: Executes actions if the build fails.

Multi Node Pipeline Example

```
node('master') {
    try {
        // Checkout Code Stage
        stage('Checkout Code') {
            node('node1') {
                echo "Checking out code on ${env.NODE_NAME}"
                checkout([$class: 'GitSCM', branches: [[name:
'*/main']], userRemoteConfigs: [[url:
'https://github.com/example/repo.git']]])
```

```
            }
        }

        // Build Stage
        stage('Build') {
            node('node2') {
                echo "Building on ${env.NODE_NAME}"
                sh 'mvn clean compile'
            }
        }

        // Test Stage
        stage('Test') {
            node('node3') {
                echo "Testing on ${env.NODE_NAME}"
                sh 'mvn test'
            }
        }

        // Deploy Stage
        stage('Deploy') {
            node('node4') {
                echo "Deploying on ${env.NODE_NAME}"
                sh 'mvn deploy'
            }
        }
    } catch (Exception e) {
        currentBuild.result = 'FAILURE'
        echo "Pipeline failed: ${e.message}"
        throw e
    } finally {
        echo 'Pipeline finished.'
    }
}
```

Multiple Nodes at the Same Level

```
try {
    // Checkout Code Stage
    node('node1') {
        stage('Checkout Code') {
            echo "Checking out code on ${env.NODE_NAME}"
```

```
            checkout([$class: 'GitSCM', branches: [[name:
'*/main']], userRemoteConfigs: [[url:
'https://github.com/example/repo.git']]])
        }
    }

    // Build Stage
    node('node2') {
        stage('Build') {
            echo "Building on ${env.NODE_NAME}"
            sh 'mvn clean compile'
        }
    }

    // Test Stage
    node('node3') {
        stage('Test') {
            echo "Testing on ${env.NODE_NAME}"
            sh 'mvn test'
        }
    }

    // Deploy Stage
    node('node4') {
        stage('Deploy') {
            echo "Deploying on ${env.NODE_NAME}"
            sh 'mvn deploy'
        }
    }
} catch (Exception e) {
    currentBuild.result = 'FAILURE'
    echo "Pipeline failed: ${e.message}"
    throw e
} finally {
    echo 'Pipeline finished.'
}
```

Project 01

Project Overview

Your organization is implementing continuous integration (CI) practices to streamline the software development lifecycle. As part of this initiative, you will create a Jenkins declarative pipeline for building a simple Maven project hosted on GitHub. This project aims to automate the build process, ensure code quality, and facilitate continuous delivery (CD).

Objectives

- Create a Jenkins pipeline script using declarative syntax.
- Clone a Maven project from a specified GitHub repository.
- Execute the build process and run unit tests.
- Archive build artifacts.
- Provide clear feedback on build status through Jenkins' UI and console output.

Instructions

1. **Setup Jenkins Job**
    - Create a new Jenkins pipeline job.
    - Configure the job to pull the Jenkinsfile from the GitHub repository.
2. **Create Jenkinsfile**
    - Write a declarative pipeline script (`Jenkinsfile`) that includes the following stages:
        - **Clone Repository**: Clone the Maven project from the GitHub repository.

        - **Build**: Execute the Maven build process (`mvn clean install`).
        - **Test**: Run unit tests as part of the Maven build.
        - **Archive Artifacts**: Archive the build artifacts for future use.
3. **Configure Pipeline Parameters**
    - Allow the pipeline to accept parameters such as Maven goals and options for flexibility.
    - Ensure the pipeline can be easily modified for different build configurations.
4. **Run the Pipeline**
    - Trigger the Jenkins pipeline job manually or set up a webhook for automatic triggering on GitHub repository changes.
    - Monitor the build process through Jenkins' UI and console output.
5. **Deliverables**
    - **Jenkinsfile**: A declarative pipeline script with the defined stages and steps.
    - **Jenkins Job Configuration**: Configured Jenkins job that uses the Jenkinsfile from the GitHub repository.
    - **Build Artifacts**: Successfully built and archived artifacts stored in Jenkins.
    - **Build Reports**: Output of the build process, including unit test results, displayed in Jenkins.
    - **Pipeline Visualization**: Visual representation of the pipeline stages and steps in Jenkins, showing the flow and status of each build stage.
    - **Documentation**: Detailed documentation outlining the pipeline setup process, including prerequisites, configuration steps, and instructions for modifying the pipeline.