Jenkins pipeline

In Jenkins, a pipeline is a suite of plugins that supports implementing and integrating continuous delivery pipelines into Jenkins. Jenkins Pipeline allows you to describe and automate complex build, test, and deployment workflows as code. It is defined using a Groovy-based DSL (Domain-Specific Language) and is known as Jenkins Pipeline DSL.

A Jenkins pipeline can be written in two ways:

1. **Declarative Pipeline Syntax:** A simpler and more opinionated syntax that is designed to be easy to read and write.
2. **Scripted Pipeline Syntax:** A more flexible and powerful syntax that allows you to write more complex logic but might be a bit more verbose.

Declarative Pipeline Example:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                // Build your code here
                echo 'Building...'
            }
        }
        stage('Test') {
            steps {
                // Run tests here
                echo 'Testing...'
            }
        }
        stage('Deploy') {
```

```
            steps {
                // Deploy the application
                echo 'Deploying...'
            }
        }
    }
}
```

Scripted Pipeline Example:

```
node {
    // Define stages
    stage('Build') {
        // Build your code here
        echo 'Building...'
    }
    stage('Test') {
        // Run tests here
        echo 'Testing...'
    }
    stage('Deploy') {
        // Deploy the application
        echo 'Deploying...'
    }
}
```

Key Concepts in Jenkins Pipeline:

1. **Agent:**
    - Specifies where the pipeline will run. It could be any available agent (e.g., **any**, **label 'docker'**, etc.).
2. **Stages:**
    - Divides the pipeline into different phases (e.g., build, test, deploy).
3. **Steps:**
    - Individual tasks within a stage. They can be simple commands or complex scripts.
4. **Post:**
    - Defines actions to be taken after the completion of a pipeline or specific stages.
5. **Environment:**
    - Defines environment variables that should be available within the pipeline.
6. **Tools:**
    - Specifies tools required for the pipeline (e.g., JDK, Maven).

Jenkinsfile:

The pipeline configuration is typically stored in a file called **Jenkinsfile** that is placed in the root of your version control repository. Jenkins will automatically detect and use this file to define and execute your pipeline.

Benefits of Jenkins Pipeline:

1. **Code as Infrastructure:**
    - Treat your build and deployment process as code, which can be versioned, reviewed, and tested.
2. **Reusability:**
    - Share and reuse pipeline code across different projects.
3. **Visibility and Monitoring:**
    - Get a visual representation of the pipeline and easily monitor the progress of builds and deployments.

4. **Parallel Execution:**
    - Execute stages or steps in parallel, speeding up the overall pipeline execution time.
5. **Integration with Tools:**
    - Integrate with other tools and services easily using steps or plugins.

Example Workflow:

1. **Checkout Code:**
    - Use the **git** step to check out the source code from the version control repository.
2. **Build:**
    - Compile and build your application using build tools like Maven or Gradle.
3. **Test:**
    - Run unit tests, integration tests, or other testing activities.
4. **Deploy:**
    - Deploy the application to a testing, staging, or production environment.
5. **Post-Deployment:**
    - Optionally, run additional steps after deployment, such as triggering notifications or updating documentation.

Setting Up a Jenkins Pipeline:

1. Create a new pipeline job in Jenkins.
2. Configure the pipeline definition either in the Jenkins job configuration or by using a **Jenkinsfile**.
3. Run the job to execute the pipeline.

Jenkins Pipeline provides a powerful and flexible way to define and automate your continuous integration and delivery workflows, enabling you to manage complex build and deployment processes with ease.

To create a Jenkins pipeline that builds a Java web application from a GitHub repository (**webapp1.git**) and deploys it to Apache Tomcat, you can use a Declarative Pipeline script. Below is an example pipeline script. Please note that this is a basic example, and you may need to adjust it based on your specific project structure, build tools, and deployment requirements.

Jenkins Pipeline Script:

```
pipeline {
agent any


stages {
    stage('CLONE SCM') {
        steps {
            echo 'cloning code from github'
           git branch: 'main', url: 'https://github.com/devopstraininghub/webapp1.git'
        }
    }
    stage('Build Artifact') {
        steps {
            echo 'Build Artifact with maven build tool'
            sh 'mvn clean install'
        }
    }
    stage('Deploy') {
        steps {
            echo 'Deploy to tomcat ap/n server'
            deploy adapters: [tomcat9(credentialsId: '315bc932-835e-49de-bce5-d30a4aeb2688',
path: '', url: 'http://3.87.40.101:8081/')], contextPath: 'facebook', war: '**/*.war'
        }
```

```
        }
    }
}
```

Explanation:

- **Clone SCM Stage:**
  - Checks out the source code from the GitHub repository.
- **Build Stage:**
  - Builds the Java web application using Maven. Adjust the build tool and commands based on your project.
- **Deploy to Tomcat Stage:**
  - Deploys the built WAR file to Tomcat using the Tomcat Manager API.
  - Uses environment variables for Tomcat URL, username, and password.
- **Post Section:**
  - Displays a success or failure message based on the deployment result.

Notes:

1. **Credentials:**
   - Replace **your_tomcat_username** and **your_tomcat_password** with the actual credentials for your Tomcat Manager. Consider using Jenkins credentials to securely manage sensitive information.
2. **Build Tool:**
   - Modify the build step (**sh 'mvn clean install'**) based on the build tool used in your project.
3. **GitHub Repository:**
   - Ensure that the GitHub repository URL is correct (**https://github.com/devopstraininghub/webapp1.git**).

4. **Tomcat URL:**
    - Update the **TOMCAT_URL** variable with the actual URL of your Tomcat instance.
5. **Tomcat Manager Configuration:**
    - Ensure that your Tomcat Manager is configured with the necessary roles and permissions for deploying applications.
6. **Adjust Paths:**
    - Adjust paths, URLs, and commands based on your specific project structure and configurations.

This example provides a starting point, and you may need to customize it further based on your project's characteristics and requirements.

Automating backups in Jenkins

Automating backups in Jenkins can be achieved using the ThinBackup plugin, which provides automated backup capabilities for Jenkins. Below are the steps to install and configure the ThinBackup plugin for automated backups:

Install ThinBackup Plugin:

1. Navigate to your Jenkins instance.
2. Go to "Manage Jenkins" > "Manage Plugins."
3. Click on the "Available" tab.
4. Search for "ThinBackup" in the filter box.
5. Find the "ThinBackup" plugin and check the checkbox next to it.
6. Click on the "Install without restart" button.

Configure ThinBackup:

1. After installing the ThinBackup plugin, go to "Manage Jenkins" > "Configure System."
2. Scroll down to the "ThinBackup" section.
3. Configure the following settings:

- **Backup directory:** Specify the directory where the backups should be stored. For example, **/var/jenkins_home/backup**.
- **Number of backups to keep:** Set the number of backups to retain.
- **Exclude build artifacts from backup:** Optionally, exclude build artifacts from the backup to save space.
- **Backup before every build:** Choose whether to perform a backup before every build.
- **Backup before upgrading plugins:** Choose whether to perform a backup before upgrading plugins.
- **Backup before installing plugins:** Choose whether to perform a backup before installing new plugins.
- **Backup before running a job:** Choose whether to perform a backup before running a job.
- **Backup when running a job fails:** Choose whether to perform a backup when running a job fails.

4. Click the "Save" button to save your ThinBackup configuration.

Trigger a Manual Backup:

If you want to manually trigger a backup at any time, you can use the "Backup Now" option in the "ThinBackup" section of the "Manage Jenkins" page.

Verify Backups:

After configuring ThinBackup, you can verify that backups are being created in the specified backup directory. Each backup is stored in a timestamped directory within the backup location.

Important Notes:

- Make sure the Jenkins user has the necessary permissions to write to the specified backup directory.
- Test the backup restoration process periodically to ensure that it works as expected.
- Adjust backup settings based on your organization's backup retention policies and storage capacity.

ThinBackup simplifies the backup process in Jenkins and provides flexibility in scheduling and configuring backup settings.

Master –Slave node

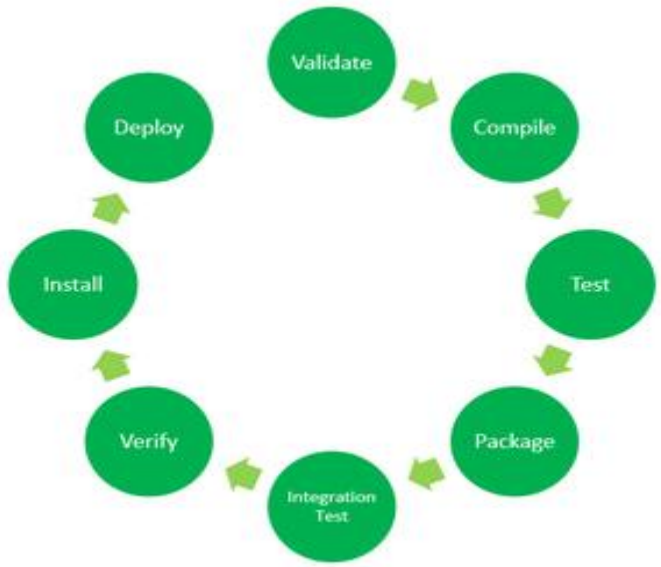
In Jenkins, the master-slave architecture allows you to distribute build and deployment tasks across multiple machines, known as "nodes" or "agents." The master node manages the overall system, while one or more slave nodes carry out the actual build and deployment jobs. This architecture is beneficial for load distribution, scalability, and parallelizing tasks.

- For setting up and run slave node watch master-slave recording classes

Maven

Maven is a widely used build automation and project management tool in the Java ecosystem. It provides a standard way to manage project builds, dependencies, and documentation. Maven uses a declarative approach to project configuration, and it follows the convention-over-configuration principle, which means that it provides sensible defaults for project structures and build processes.

Maven Lifecycle:



Key Concepts:

1. **POM (Project Object Model):**
    - The POM is an XML file that contains information about the project, its dependencies, and the build process. It serves as the configuration file for Maven.
2. **Goals:**
    - Goals represent a task or an action that can be executed during the build process. Common goals include **compile**, **test**, **package**, and **install**.

3. **Plugins:**
    - Maven plugins are extensions that provide additional functionality to Maven. Plugins are used to execute specific goals and tasks during the build lifecycle.
4. **Dependencies:**
    - Dependencies are external libraries or modules required by the project. Maven manages the downloading and inclusion of dependencies in the build process.
5. **Repositories:**
    - Repositories are locations where Maven stores and retrieves artifacts (JARs, WARs, etc.). There are local repositories on the developer's machine and remote repositories on the internet.

Maven install website: https://dlcdn.apache.org/maven/maven-3/3.9.6/binaries/apache-maven-3.9.6-bin.zip

- **Set Environment Variables:**
    - Set the **MVN_HOME** environment variable to point to the installation directory and add the **bin** directory to the **PATH**. This makes it easier to run maven commands from any location.

Maven Commands:

1. **mvn clean**
    - Cleans the build artifacts, removing the **target** directory.
2. **mvn compile**
    - Compiles the source code.
3. **mvn test**
    - Runs the unit tests.
4. **mvn package**
    - Packages the compiled code into a distributable format (e.g., JAR or WAR).
5. **mvn install**

- Installs the packaged artifact into the local Maven repository. This makes it available for other projects on the same machine.

6. **mvn deploy**
    - Deploys the artifact to a remote repository, making it available for other developers or projects.
7. **mvn archetype:generate**
    - Interactively generates a Maven project from an archetype (a project template).
8. **mvn dependency:tree**
    - Displays the project's dependency tree.
9. **mvn site**
    - Generates a project site, including documentation and reports.

Example Maven Project Structure:

A typical Maven project follows a standard directory structure:

```
my-project
|-- src
|   |-- main
|   |   |-- java
|   |   |   `-- (source code)
|   |   `-- resources
|   |       `-- (configuration files)
|   `-- test
|       |-- java
|       |   `-- (test source code)
|       `-- resources
|           `-- (test configuration files)
|-- target
|   `-- (compiled classes and packaged artifacts)
|-- pom.xml
```

Basic Maven Workflow:

1. **Create a Project:**
    - Use the **mvn archetype:generate** command to create a new Maven project from an archetype.
2. **Edit the POM:**
    - Customize the **pom.xml** file to specify project details, dependencies, and build configurations.
3. **Run Maven Commands:**
    - Execute Maven commands such as **mvn clean install** to build and install the project.
4. **Explore the Target Directory:**
    - The compiled classes and packaged artifacts are stored in the **target** directory.
5. **Manage Dependencies:**
    - Declare dependencies in the **pom.xml** file. Maven will automatically download and manage dependencies.
6. **Run Tests:**
    - Use **mvn test** to run unit tests.
7. **Generate Documentation:**
    - Use **mvn site** to generate project documentation.

Maven greatly simplifies the process of building and managing Java projects, and it is widely used in the Java development community. The Maven Central Repository provides a vast collection of libraries and plugins that can be easily integrated into Maven projects.

Java installation on windows

1. Create an Oracle Account:

- Before you start, you need to have an Oracle account. If you don't have one, you can create it on the Oracle website.

2. Download Java SE Development Kit (JDK):

- Visit the Oracle JDK download page: Oracle JDK Downloads

3. Accept License Agreement:

- On the download page, locate the version of JDK you want to install.
- Accept the license agreement by checking the checkbox next to "I reviewed and accept the Oracle Technology Network License Agreement for Oracle Java SE."

4. Sign In with Oracle Account:

- Click on the "Sign In" link next to the license agreement checkbox.
- Sign in with your Oracle account credentials.

5. Download JDK Installer:

- After signing in, click the download link for the JDK installer that corresponds to your Windows architecture (32-bit or 64-bit).

6. Run the Installer:

- Once the installer is downloaded, run the executable file (e.g., jdk-8u391-windows-x64.exe for 64-bit).
- If prompted, grant administrative privileges to the installer.

7. Follow the Installation Wizard:

- The installation wizard will guide you through the installation process.
- Choose the installation directory and installation options.

- Click "Next" and then "Install" to start the installation.

8. Complete the Installation:

- Wait for the installation process to complete.
- Once installed, you may be prompted to register your product. You can choose to do it later.

9. Verify Java Installation:

- Open a Command Prompt and run the following commands to verify the installation:

  java -version

- The output should display the installed Java version.
- **Set Environment Variables:**
  - Set the **JAVA_HOME** environment variable to point to the installation directory and add the **bin** directory to the **PATH**. This makes it easier to run Java commands from any location.

Difference between Maven and Ant:

Aspect	Maven	Ant
Configuration Language	XML	XML
Dependency Management	Centralized (Maven Central)	Manual or external tools
Plugin Ecosystem	Extensive	Modular but not as extensive
Convention vs. Configuration	Convention over Configuration	No strict conventions
Scripting Language	N/A (XML-based)	Java with XML configuration
Incremental Builds	Supported	Requires custom scripting
Project Structure	Enforces standard structure	Flexible project structure
Built-In Lifecycle	Defined and enforced	Customizable
Build Phases	Predefined lifecycle phases	Customizable targets and tasks
Learning Curve	Easy to start, complex for	Steeper learning curve

Aspect	Maven	Ant
	advanced configurations	

This comparison covers key aspects such as configuration language, dependency management, plugin ecosystem, convention vs. configuration, scripting language, support for incremental builds, project structure, built-in lifecycle, and learning curve.

Maven is known for its convention-over-configuration approach, predefined lifecycles, and standardized project structure, making it easier to start with but potentially complex for advanced configurations. On the other hand, Ant provides more flexibility and customization but comes with a steeper learning curve and lacks strict conventions.

The choice between Maven and Ant often depends on project requirements, team preferences, and the desired level of control over the build process.