Chennai Institute of Technology

Chennai – 69

DevOps Practical's Notebook

Subject Code: CCS342

Prepared by: Mr. Pratham Verma

Chennai Institute of Technology, Chennai

Lab Practical: Create Maven Build Pipeline in Azure

Aim

This lab aims to create a continuous integration and continuous delivery (CI/CD) pipeline in Azure DevOps to automate the build process for a Maven project.

Procedure

1. **Create an Azure DevOps Project (if you don't have one already):**
    - Sign in to your Azure DevOps account and navigate to "Organizations" -> "[Your Organization Name]".
    - Click on "New project" and choose a name for your project.
2. **Define the Maven Project (Locally):**
    - You can use an existing Maven project or create a sample project using Maven Archetypes (https://maven.apache.org/archetypes/index.html).
    - Ensure you have a `pom.xml` file configured with your project dependencies and build tasks.
3. **Connect your Code Repository (Optional):**
    - If your code resides in a version control system like Git, connect the repository to your Azure DevOps project. You can use services like GitHub, Azure Repos, etc.
4. **Create a CI/CD Pipeline:**
    - Go to "Pipelines" -> "Pipelines" and select "New pipeline".
    - Choose "Starter pipeline" and select your code repository (if applicable) or "Empty job".
5. **Configure the Pipeline YAML:**
    - Edit the YAML script for your pipeline. Here's an example:

YAML

```yaml
pool:
  vmImage: 'ubuntu-latest'  # Adjust VM image if needed

stages:
- stage: Build
  jobs:
  - job: Build_Job
    steps:
    - task: Maven@3  # Use the appropriate Maven version task
      inputs:
        mavenVersion: '3.x'  # Specify your Maven version
        mavenPomFile: 'pom.xml'  # Path to your pom.xml file
        goals: 'clean package'  # Maven goals to execute (clean and package)
```

```
        publishMavenArtifacts: true  # Optionally publish artifacts

* Explanation:
    * `pool` defines the virtual machine image used for the pipeline execution.
    * `stages` define the pipeline stages (e.g., Build).
    * `jobs` define the jobs within a stage (e.g., Build_Job).
    * `task` specifies the "Maven@3" task to use.
    * `inputs` configure the task with Maven version, pom.xml location, build goals, and optionally publishing artifacts.
```

6. **Save and Queue the Pipeline:**
    - Save the YAML script and queue the pipeline to run.
7. **Monitor the Pipeline Run:**
    - Go to "Pipelines" -> "Builds" and view the pipeline execution details. You can see the build status, logs, and artifacts (if published).

Inference

- A successful pipeline run indicates that your Maven project has been built and artifacts (compiled code) are generated (if publishing is enabled).
- The pipeline logs provide details about the build process and any potential errors.

Result

- You will have a functional CI/CD pipeline in Azure DevOps that automatically builds your Maven project upon code changes pushed to the repository (if connected) or manual execution.

Lab Practical Solution: Run Regression Tests using Maven Build Pipeline in Azure

Aim:

This lab practical aims to demonstrate how to integrate regression test execution with a Maven build pipeline deployed in Azure DevOps.

Procedure:

1. **Create an Azure DevOps Project**
    - Login to Azure DevOps (https://azure.microsoft.com/en-us/products/devops) and create a new project or use an existing one.
2. **Define the Maven Build Pipeline**
    - Go to **Pipelines** -> **Releases** and create a new pipeline.
    - Choose the template **"Empty pipeline"**.
    - In the YAML editor, paste the following code:

YAML

```yaml
stages:
- stage: Build  # Define a build stage
  jobs:
  - job: BuildTests  # Define a job within the build stage
    pool:  # Define the agent pool to run the job on
      vmImage: 'ubuntu-latest'
    steps:
    - script: mvn clean install  # Run maven clean and install commands
      displayName: 'Build and Install Tests'
    - publish: $(System.DefaultWorkingDirectory)/target/*.jar  # Publish the generated JAR file as artifact

- stage: Test  # Define a test stage
  jobs:
  - job: RunTests  # Define a job within the test stage
    dependsOn: Build  # Define dependency on the Build stage
    pool:  # Define the agent pool to run the job on
      vmImage: 'ubuntu-latest'
    steps:
    - download: $(Build.ArtifactStagingDirectory)  # Download the JAR artifact from the Build stage
    - script: mvn test  # Run maven test command to execute regression tests
      displayName: 'Run Regression Tests'
    - publish: $(System.DefaultWorkingDirectory)/target/surefire-reports/*.xml  # Publish test reports as artifacts
```

Explanation:

- The pipeline defines two stages: `Build` and `Test`.
- The `Build` stage uses the `mvn clean install` command to build and install the project with its dependencies.
- The `Build` stage publishes the generated JAR file as an artifact.

- The `Test` stage depends on the successful completion of the `Build` stage.
- The `Test` stage downloads the JAR artifact from the previous stage.
- The `Test` stage runs the `mvn test` command to execute the regression tests.
- The `Test` stage publishes the generated test reports (usually located in the `target/surefire-reports` directory) as artifacts.

3. **Run the Pipeline**
    - Save the pipeline YAML file and queue a new pipeline run.
    - Monitor the pipeline execution in the **Runs** tab.

Inference:

A successful pipeline run indicates that the Maven build process executed without errors, the tests were downloaded and executed successfully, and the test reports were published as artifacts.

Result:

By examining the published test reports (usually in JUnit XML format), you can analyze the test results and identify any failing tests that require further investigation.

Lab Practical: Install Jenkins in Cloud

Aim

The aim of this lab practical is to set up and configure Jenkins, an open-source automation server, in a cloud environment. This will enable you to automate the building, testing, and deployment of your applications.

Procedure

1. Choosing a Cloud Provider:

- Select a cloud provider of your choice like Amazon Web Services (AWS), Microsoft Azure, or Google Cloud Platform (GCP). Each platform offers different options for deploying Jenkins.

2. Instance Creation:

- Launch a virtual machine (VM) instance in your chosen cloud platform. Ensure the VM meets the minimum requirements for Jenkins.
- Configure the VM with an appropriate operating system (OS) like Ubuntu or CentOS.

3. Install Java:

- As Jenkins requires Java for execution, install the latest Java Development Kit (JDK) on your cloud VM following the provider's instructions.
- Verify the Java installation by running `java -version` in the terminal.

4. Download and Install Jenkins:

- Download the latest LTS (Long Term Support) version of the Jenkins WAR file from the official Jenkins website https://www.jenkins.io/download/.
- Transfer the WAR file to your cloud VM using a method like SCP or cloud storage.
- Use the `wget` command to download the WAR file directly if allowed by your cloud provider's security settings.

5. Configure Jenkins:

- Start the Jenkins service using a command like `java -jar jenkins.war` (the actual command might differ based on your OS).
- Access the Jenkins web interface by opening your web browser and navigating to `http://<VM_IP_ADDRESS>:8080` (replace `<VM_IP_ADDRESS>` with your VM's IP address).
- The initial setup will prompt you to unlock Jenkins using a randomly generated password located in the console output where Jenkins was started.

6. Install Plugins (Optional):

- Jenkins offers a vast library of plugins to extend its functionality.
- Visit the "Manage Jenkins" -> "Plugins" section and browse/search for plugins relevant to your automation needs.
- Install the desired plugins and restart Jenkins for them to take effect.

7. Create a Sample Job (Optional):

- To verify your Jenkins installation, create a simple job.
- Go to "New Item" and choose a job type like "FreeStyle Project".
- Configure the job to perform a basic task like executing a shell script or building a sample project.
- Save the job and run it to test the Jenkins functionality.

8. Secure Jenkins (Recommended):

- Implement security measures to protect your Jenkins server. This may involve configuring user authentication, authorization, and network access controls.

Inference

By successfully completing this lab, you will have gained hands-on experience in deploying Jenkins in a cloud environment. This allows you to automate your software development lifecycle, improving efficiency and consistency.

Result

A functioning Jenkins server running in your cloud environment, ready to automate your build, test, and deployment pipelines. You should be able to access the Jenkins web interface and create jobs to manage your software development process.

Lab Practical: Create a CI Pipeline Using Jenkins

Aim

This lab aims to establish a continuous integration (CI) pipeline using Jenkins. The pipeline will automate the build and test process for a sample software project upon code changes.

Procedure

1. **Setting Up Jenkins:**
    - Ensure you have a running Jenkins server. If not, download and install it from the official website https://www.jenkins.io/download/.
    - After installation, configure Jenkins by following the initial setup wizard.
2. **Creating a New Project:**
    - Login to the Jenkins dashboard.
    - Click on "New Item" and select "Pipeline" from the options.
    - Provide a name for your project (e.g., "CI_Pipeline_Project").
    - Choose "Pipeline script from SCM" under the "Pipeline" section.
3. **Source Code Management (SCM) Configuration:**
    - Select the SCM tool you're using (e.g., Git).
    - Enter the URL of your Git repository containing the sample project code.
    - Specify the branch name that triggers the pipeline execution (e.g., "main").
    - Leave the credentials section blank for now (we can add them later if needed).
4. **Defining the Pipeline Script (Jenkinsfile):**
    - In the script editor, paste the following code (replace `<path_to_your_project>` with the actual path within your repository):

Groovy

```
pipeline {
    agent any

    stages {
        stage('Checkout Code') {
            steps {
                git branch: '<branch_name>', credentialsId: '', url:
'<repository_url>'
            }
        }
        stage('Build Project') {
            // Replace the following steps with your project-specific build
commands
            steps {
                sh 'cd <path_to_your_project>'
```

```
                sh './build.sh' // Assuming you have a build script named
build.sh
            }
        }
        stage('Run Tests') {
            // Replace the following steps with your project-specific test
commands
            steps {
                sh 'cd <path_to_your_project>'
                sh './test.sh' // Assuming you have a test script named
test.sh
            }
        }
    }

    post {
        always {
            archiveArtifacts artifacts: '**/*.log' // Archive logs from all
stages
        }
        success {
            // Optional: Send notifications on successful builds (e.g.,
email)
        }
        failure {
            // Optional: Send notifications on failed builds (e.g., email)
        }
    }
}
```

- Save the Jenkinsfile.

5. **Building the Pipeline:**
   - Click on "Build Now" to trigger the initial pipeline execution. This will fetch the code from your repository and run the build and test stages.
6. **Monitoring and Verification:**
   - Observe the pipeline execution progress on the Jenkins dashboard.
   - Each stage will display its status (Success/Failure).
   - Click on each stage to view the console output for detailed logs.
   - Verify if the build and tests passed successfully.
7. **Testing Code Changes:**
   - Make a modification to your project code in the repository (e.g., fix a bug).
   - Push the changes to your remote repository branch.
   - Observe Jenkins automatically trigger a new pipeline run due to the code change detection.
   - Monitor the new build and ensure successful completion of all stages after your code changes.

Inference

By creating a CI pipeline with Jenkins, we achieve the following:

- **Automation:** The build and test process becomes automated, eliminating manual intervention and reducing errors.
- **Faster Feedback:** Developers receive quicker feedback on code changes, allowing for faster bug fixes and improvements.
- **Improved Quality:** Automated testing helps identify and fix issues early in the development cycle, leading to better code quality.
- **Continuous Integration:** Code changes are integrated frequently, minimizing code divergence and conflicts.

Result

This lab demonstrates the creation of a basic CI pipeline using Jenkins. You can customize the pipeline script further to integrate specific build tools, testing frameworks, and notification workflows based on your project requirements. By implementing CI, you establish a foundation for continuous development practices, leading to higher quality softwAare and faster releases.

Lab Practical: CI/CD Pipeline with Jenkins and Cloud Deployment

Aim

This lab practical aims to establish a continuous delivery (CD) pipeline using Jenkins. The pipeline will automate the process of fetching code from a version control system (VCS), building the application, and deploying it to a cloud environment.

Procedure

1. Setting Up Jenkins

- Install Jenkins on your system. You can find installation instructions for various platforms on the Jenkins website https://www.jenkins.io/download/.
- Start and configure Jenkins according to your needs. This might involve setting up security measures and installing additional plugins.

2. Configuring the Pipeline

- In Jenkins, create a new pipeline job. You can choose between a freestyle project or a pipeline script. We'll use a pipeline script for this exercise.
- Within the pipeline script editor, define the pipeline using Jenkins Pipeline DSL (Domain Specific Language). Here's an example script:

Groovy

```
pipeline {
    agent any

    stages {
        stage('Checkout Code') {
            steps {
                git branch: 'main', credentialsId: 'github-credentials', url:
'https://github.com/your-username/your-repo.git'
            }
        }
        stage('Build Application') {
            steps {
                sh 'mvn clean install' // Replace with your build command (e.g.,
npm install, etc.)
            }
        }
        stage('Deploy to Cloud') {
            steps {
                // Deployment logic specific to your cloud provider (see examples
below)
            }
        }
    }
}
```

Explanation of the Script:

- `agent any`: This specifies that the pipeline can run on any available Jenkins agent.
- `stages`: This defines the different stages of the pipeline.
    - `Checkout Code`: This stage fetches code from the specified Git repository using credentials stored in Jenkins (replace `github-credentials` with your actual credential ID).
    - `Build Application`: This stage executes the build command specific to your project (e.g., `mvn clean install` for Maven projects).
    - `Deploy to Cloud`: This stage contains the deployment logic to your cloud platform (details in the next step).
- You'll need to replace the placeholder commands with your project's specific build and deployment commands.

3. Cloud Deployment Configuration

This stage depends on your chosen cloud provider. Here are some examples:

- **AWS**: Install the AWS plugin for Jenkins and configure credentials for accessing your AWS account. You can then use pipeline steps like `aws s3 cp` to upload build artifacts to S3 buckets and utilize tools like AWS CodeDeploy for automated deployments to EC2 instances or Elastic Beanstalk environments.
- **Azure**: Install the Azure DevOps plugin for Jenkins and configure your Azure service connection. The pipeline can then leverage Azure CLI commands or pre-built functions within the plugin to deploy applications to Azure App Service, Azure Functions, or Virtual Machines.
- **Google Cloud Platform (GCP)**: Install the Google Cloud plugin for Jenkins and configure your GCP service account. The pipeline can utilize `gcloud` commands to deploy applications to Cloud Run, App Engine, or Compute Engine instances.

4. Running the Pipeline

- Save your pipeline script and configure the job in Jenkins.
- Trigger the pipeline manually or set up triggers based on events like code commits to your VCS.
- Jenkins will execute the pipeline stages sequentially. You can monitor the progress of each stage on the Jenkins dashboard.

Inference

By creating a CI/CD pipeline in Jenkins, you achieve several benefits:

- **Automation**: Manual tasks like code retrieval, building, and deployment are automated, reducing human error and increasing efficiency.
- **Faster deployments**: New code versions can be deployed to production quicker, enabling faster delivery cycles.
- **Improved consistency**: The pipeline ensures a consistent build and deployment process across environments.
- **Early detection of issues**: Build failures and errors are identified early in the pipeline, allowing for faster troubleshooting.

Result

A successful run of the pipeline should result in:

- Code being fetched from the VCS repository.
- The application being built successfully.
- The application being deployed to the target cloud environment.
- A confirmation message displayed on the Jenkins dashboard indicating successful deployment.

Lab Practical: Ansible Playbook for Simple Web Application Infrastructure

Aim

This lab aims to create an Ansible playbook that automates the setup of a basic web application infrastructure. This includes installing and configuring a web server (Nginx) and deploying a static HTML webpage.

Procedure

1. Setting Up the Environment:

- **Install Ansible:** Ensure Ansible is installed on your local machine. You can follow the official documentation for your operating system https://docs.ansible.com/ansible/latest/installation_guide/index.html.
- **Prepare Inventory File:** Create a file named `inventory` in your project directory. This file lists the servers Ansible will manage. Here's an example:

```
[webserver]
server1.example.com
```

Replace `server1.example.com` with the actual hostname or IP address of your web server.

- **Prepare Playbook File:** Create another file named `webserver.yml` in your project directory. This file will contain the Ansible playbook.

2. Building the Playbook:

Open the `webserver.yml` file and paste the following content:

YAML

```yaml
---
- hosts: webserver
  become: true

  tasks:
    # Update package lists
    - name: Update package lists
      apt: update_cache=yes

    # Install Nginx web server
    - name: Install Nginx web server
      apt: name=nginx state=present

    # Start Nginx service
    - name: Start Nginx service
      service: name=nginx state=started enabled=yes

    # Create directory for web content
    - name: Create directory for web content
      file: path=/var/www/html state=directory
```

```
    # Copy the index.html file
    - name: Copy index.html file
      copy:
        src: index.html
        dest: /var/www/html/index.html
        owner: www-data
        group: www-data

  handlers:
    # Restart Nginx on config changes
    - name: Restart Nginx on config changes
      service: name=nginx state=restarted
        when: changed
```

Explanation of the Playbook:

- `hosts: webserver`: This line defines the group of servers the playbook will target (in this case, `webserver` from the inventory file).
- `become: true`: This grants the user running the playbook root privileges on the target server.
- **Tasks**: This section defines a series of tasks that Ansible will execute on the target server(s).
    - The first task updates the package lists on the server.
    - The second task installs the Nginx web server package.
    - The third task starts the Nginx service and ensures it automatically starts on boot.
    - The fourth task creates a directory `/var/www/html` to store the web content.
    - The fifth task copies a sample `index.html` file (not included here) to the created directory, setting ownership to `www-data`.
- **Handlers**: This section defines actions triggered under specific conditions.
    - The handler restarts the Nginx service whenever a configuration change occurs (indicated by the `when: changed` condition).

3. Deploying the Playbook:

- **Place the index.html file:** Create a simple HTML file named `index.html` containing your desired content in your project directory.
- **Run the Playbook:** Navigate to the directory containing the `inventory` and `webserver.yml` files. Execute the following command:

```
ansible-playbook webserver.yml
```

4. Verification:

Once the playbook finishes execution, access your web server's IP address in a web browser. You should see the content of your `index.html` file displayed.

Inference

This lab demonstrates how Ansible can automate the setup and configuration of a basic web application infrastructure. By using playbooks, we can manage infrastructure in a repeatable and consistent manner.

Here are some key takeaways:

- Ansible uses YAML syntax for defining playbooks.
- Playbooks define tasks to be executed on target servers.
- Roles can be created to group related tasks for reusability.
- Inventory files define the target servers managed by Ansible.
- Variables can be used to store values and customize playbooks.

Result

This lab provides a foundational understanding of using Ansible for web application infrastructure automation. With further exploration, you can learn to deploy more complex applications, manage configurations, and automate various infrastructure tasks.

Lab Practical: Build a Simple Application using Gradle

Aim

This lab aims to introduce you to Gradle, a powerful build automation tool for Java projects. By building a simple application, you'll gain hands-on experience with Gradle's core functionalities like project creation, dependency management, and task execution.

Procedure

1. Setting Up the Environment

- Ensure you have Java (version 8 or above) installed on your system. You can verify this by running `java -version` in your terminal.
- Download and install an IDE that supports Gradle plugins, such as IntelliJ IDEA or Eclipse.

2. Creating a Gradle Project

- Open your terminal and navigate to your desired project directory.
- Run the following command to initialize a new Gradle project:

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

This command will prompt you for some configuration options. Choose the following:

- **Project type:** Application
- **Implementation language:** Java
- **Source compatibility:** Choose a compatible Java version based on your system (e.g., Java 17)

Gradle will generate the essential project structure with a `build.gradle` file containing the build configuration.

3. Writing the Application Code (Simple Greeter)

- Inside the `src/main/java` directory, create a package named `com.example.myapp` (or your preferred package name).
- Within the package, create a new Java class named `Greeter.java`.
- Add the following code to `Greeter.java`:

Java

```java
package com.example.myapp;

public class Greeter {

    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

This simple code defines a `Greeter` class with a `main` method that prints "Hello, World!" to the console.

4. Configuring Dependencies (Optional)

- Gradle allows you to manage external libraries your application depends on. For instance, if you wanted to use a logging library, you could add it to the `build.gradle` file:

Gradle

```
dependencies {
    implementation 'org.slf4j:slf4j-api:1.8.0-beta4'
    runtimeOnly 'org.slf4j:slf4j-simple:1.8.0-beta4'
}
```

This snippet defines two dependencies:

- `slf4j-api`: The core logging API
- `slf4j-simple`: A simple logging implementation

5. Building and Running the Application

- Open a terminal in your project directory.
- Execute the following Gradle task to compile your Java code:

```
gradle build
```

This task will compile the source code and create an executable JAR file (usually located in `build/libs`).

- Run your application using the generated JAR:

```
java -jar build/libs/your-application-name.jar
```

Replace `your-application-name.jar` with the actual name of your JAR file.

6. Exploring Gradle Tasks

Gradle offers various built-in tasks that automate different stages of the development process. Here are some commonly used tasks:

- `clean`: Deletes generated files like compiled classes and JARs.
- `assemble`: Builds your application, including compilation and packaging.
- `run`: Executes your application's main method.

You can explore these tasks by running `gradle tasks` in your terminal. The output will list all available tasks and their descriptions.

7. Additional Considerations

- Gradle allows writing custom build scripts to automate complex tasks. Explore the Gradle documentation for details on advanced build customization.
- Gradle integrates seamlessly with version control systems like Git, allowing you to manage build configurations alongside your source code.

Inference

By completing this lab, you've gained practical experience with Gradle's core functionalities:

- **Project Creation:** Gradle simplifies project setup with pre-defined configurations for common project types like Java applications.
- **Dependency Management:** Gradle handles external libraries your application relies on, ensuring consistent versions and resolving conflicts.
- **Task Automation:** Gradle provides built-in and custom tasks to automate repetitive tasks like compiling, building, and testing your application.

This lab has laid the groundwork for you to explore Gradle's vast capabilities in managing complex build processes and integrating with continuous integration and delivery (CI/CD) pipelines for efficient software development.

Result

This lab has successfully demonstrated how to build a simple Java application using Gradle. You've created a project structure, written application code, configured dependencies (optional), built and run the application, and explored some essential Gradle tasks.

Lab Practical: Ansible Installation, Roles, and Playbooks

Aim:

This lab practical aims to equip you with the skills to install and configure Ansible, a popular IT automation tool. You will learn how to create roles for modular code reuse and write playbooks, the core automation units in Ansible.

Procedure:

1. Setting Up the Environment

- **Control Node:** Choose a machine to act as your Ansible control node. This is where you will install Ansible and manage your infrastructure. Ensure the control node has a non-root user with sudo privileges.
- **Managed Nodes:** Identify the machines you want to manage with Ansible (servers, network devices, etc.). These are referred to as managed nodes. Configure passwordless SSH access from the control node to each managed node. This allows Ansible to execute tasks without manual intervention.

2. Installing Ansible

On your control node, follow the installation instructions specific to your operating system.

- **For Debian/Ubuntu:**

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

- **For Red Hat/CentOS (Requires EPEL repository):**

Bash
```
sudo yum install epel-release
sudo yum install ansible
```

3. Verifying Installation

Once installed, run the following command to verify the Ansible version:

Bash
```
ansible --version
```

4. Creating an Inventory File

An inventory file defines the managed nodes Ansible will interact with. There are several formats, but a simple static inventory is a good starting point. Create a file named `hosts` (or any preferred name) in the `/etc/ansible` directory with the following structure:

```
[group_name]  # Define a group of managed nodes
managed_node1.example.com
managed_node2.example.com

[another_group]  # Define another group (optional)
other_node1.example.com
```

5. Writing Your First Playbook

Playbooks are YAML files that define the tasks Ansible will execute on managed nodes. Create a new file named `sample_playbook.yml` in your preferred working directory. Here's a basic example:

YAML

```yaml
---
- name: Install Apache Web Server  # Play name
  hosts: all  # Target all hosts in the inventory
  become: true  # Use sudo privileges for tasks
  tasks:
    - name: Install Apache package
      apt:  # Use the apt module for Debian/Ubuntu
        name: apache2
        state: present
```

Explanation of the Playbook:

- `---`: YAML document start marker
- `- name: Install Apache Web Server`: Defines the play name (optional but descriptive)
- `hosts: all`: Specifies the target group from the inventory (here, all)
- `become: true`: Grants sudo privileges for tasks within the play
- `tasks:`: Defines the tasks to be executed
    - `- name: Install Apache package`: Defines a task with a descriptive name
        - `apt:`: Uses the `apt` module for package management (adjust for other systems)
            - `name: apache2`: Specifies the package to install
            - `state: present`: Ensures the package is installed

6. Running the Playbook

Navigate to the directory containing your playbook and run the following command:

Bash

```
ansible-playbook sample_playbook.yml
```

This will attempt to install Apache on all managed nodes listed in the `all` group of your inventory.

7. Creating Ansible Roles (Optional)

Ansible roles provide a way to modularize your configuration code. A role encapsulates tasks, variables, and files related to a specific configuration area (e.g., web server setup, database configuration).

Here's a basic structure for an Ansible role:

```
roles/
  my_web_server/
    defaults/  # Contains default variables for the role
    tasks/  # Contains task files for the role
```

```
vars/  # Contains additional variables specific to the role
handlers/  # Contains handler tasks (optional)
meta/  # Contains role metadata (optional)
```

8. Conclusion

This lab practical provided a foundational understanding of Ansible installation, roles, and playbooks. You can now leverage Ansible to automate various IT tasks, improving efficiency and consistency in your infrastructure management.

Inference:

- Ansible offers a powerful and agentless approach to infrastructure automation.
- Playbooks provide a declarative way to define desired configurations.
- Roles enable code reuse and improve maintainability of complex configurations.

Result:

By successfully completing this lab, you will have achieved the following:

- **Installed Ansible on your control node:** You will have verified the installation using the `ansible --version` command.
- **Created an inventory file:** This file defines the managed nodes Ansible can interact with, allowing you to target specific groups of machines for configuration changes.
- **Written a basic playbook:** You will have created a YAML file outlining the tasks Ansible executes on managed nodes. The example playbook demonstrated installing the Apache web server.
- **Successfully executed a playbook:** Running `ansible-playbook sample_playbook.yml` will attempt to install Apache on all managed nodes listed in the `all` group of your inventory. Verifying the installation on managed nodes confirms successful Ansible execution.
- **(Optional) Gained an understanding of Ansible roles:** The lab introduced the concept of roles for modularizing configuration code. You learned about the basic directory structure of an Ansible role.

This practical exercise equips you with the foundational skills to leverage Ansible for automating IT tasks across your infrastructure. As you gain experience, you can create more complex playbooks, utilize roles effectively, and explore advanced features like variables, conditionals, and loops to automate a wider range of configurations.