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:

2. cd path/to/HelloMaven
3. git init
4. git add .
5. 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:

6. HelloMaven/
7. ├── pom.xml
8. └── src
9. ├── main/java/com/example/App.java
10. └── 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., */mainor */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.xmlis 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 targetdirectory.

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:**

```
4. ---
5. - name: Deploy Maven Artifact
6.     hosts: local
7.     become: yes
8.     tasks:
9.       - name: Copy the artifact to the deployment directory
10.          copy:
11. 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 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.