**Ex-8:**

**step 1: Set Up Your Version Control System (VCS)**

1. **Select a VCS**: Ensure you’re using a VCS like Git.
2. **Create a Repository**: If you haven't already, create a repository for your web application code.

**Step 2: Create Your Build Script**

1. **Create a Build Script**: Write a build script (e.g., build.sh or build.ps1 depending on your OS).
   * This script should contain the necessary commands to build your application.

Example build.sh:

bash

Copy code

#!/bin/bash

echo "Building the application..."

# Add build commands here

1. **Make the Script Executable**:

bash

Copy code

chmod +x build.sh

**Step 3: Set Up Jenkins**

1. **Install Jenkins**: If Jenkins is not installed, install it following the official documentation.
2. **Access Jenkins**: Open Jenkins in your web browser (http://your\_jenkins\_url:8080).
3. **Install Required Plugins**:
   * Go to **Manage Jenkins** > **Manage Plugins**.
   * Ensure that you have the **Git Plugin** installed (for Git repositories).

**Step 4: Create a Jenkins Pipeline Job**

1. **Create a New Item**:
   * Click on **New Item**.
   * Enter a name for your job (e.g., MyWebAppBuild) and select **Pipeline**.
   * Click **OK**.
2. **Configure Pipeline**:
   * In the job configuration, scroll down to the **Pipeline** section.
   * Set the **Definition** to **Pipeline script**.
   * Write a pipeline script to clone the repository and execute the build script:

Example Jenkins Pipeline:

groovy

Copy code

pipeline {

agent any

stages {

stage('Checkout') {

steps {

git 'https://your-repo-url.git'

}

}

stage('Build') {

steps {

sh './build.sh'

}

}

}

triggers {

pollSCM('\* \* \* \* \*') // Polls every minute

}

}

1. **Save the Job**.

**Step 5: Configure Webhook for Automatic Triggers (Optional but Recommended)**

1. **Create a Webhook**: In your VCS (e.g., GitHub, GitLab), navigate to your repository settings.
2. **Add a Webhook**:
   * URL: http://your\_jenkins\_url:8080/github-webhook/
   * Content type: application/json
   * Select the event that triggers the webhook (e.g., push events).

**Step 6: Make a Change to Your Web Application**

1. **Modify Your Web App**: Change the background color in your landing page (e.g., in index.html or CSS file).

html

Copy code

<style>

body {

background-color: lightblue; /\* Change to your desired color \*/

}

</style>

1. **Commit and Push Changes**:

bash

Copy code

git add .

git commit -m "Change background color of the landing page"

git push origin main # or your appropriate branch

**Step 7: Verify the Pipeline Execution**

1. **Check Jenkins**: Go back to your Jenkins dashboard.
2. **Monitor Builds**: Click on your job (MyWebAppBuild) and check the build history. The build should trigger automatically based on your change.
3. **Check Console Output**: Click on the latest build to view console output and ensure your build script executed successfully.

**Conclusion**

You’ve successfully configured a Jenkins pipeline to call a build script whenever there’s a change in your application’s version control system. By modifying the landing page’s background color and pushing the changes, you can verify that the pipeline runs as expected. If you face any issues, check the Jenkins logs for troubleshooting.

Ex-9

<https://www.blazemeter.com/blog/cicd-pipeline-jenkins-github>

**Step 1: Install the Required Plugins**

1. **Access Jenkins**: Open Jenkins in your web browser.
2. **Manage Plugins**:
   * Go to **Manage Jenkins** > **Manage Plugins**.
   * Ensure the following plugins are installed:
     + **Pipeline** (should be installed by default).
     + **Pipeline: Stage View**.
     + **Build Pipeline Plugin** (optional but useful for visual representation).

**Step 2: Create the Pipeline View**

1. **Create a New View**:
   * From the Jenkins dashboard, click on the **+** icon next to the existing views or select **New View**.
   * Enter a name for your view (e.g., MyWebAppPipelineView).
   * Choose **Pipeline** and click **OK**.
2. **Configure the Pipeline View**:
   * In the configuration section, you can add jobs to your view:
     + Select your pipeline job (e.g., MyWebAppBuild) and click **Add**.
   * Set the **Display Name** and customize any other settings as needed.
3. **Save the View**.

**Step 3: Customize User-Defined Messages**

You can define custom messages in your Jenkins pipeline script. Here’s how to do that:

1. **Edit Your Pipeline Job**:
   * Navigate to your pipeline job (MyWebAppBuild).
   * Click **Configure**.
2. **Add User-Defined Messages**: You can include echo statements in your pipeline stages to display messages. Here’s an example of how to customize messages:

Updated Jenkins Pipeline Script:

groovy

Copy code

pipeline {

agent any

stages {

stage('Checkout') {

steps {

echo 'Checking out the code...'

git 'https://your-repo-url.git'

}

}

stage('Build') {

steps {

echo 'Starting the build process...'

sh './build.sh'

echo 'Build completed successfully!'

}

}

}

post {

success {

echo 'Pipeline completed successfully!'

}

failure {

echo 'Pipeline failed. Please check the logs.'

}

}

triggers {

pollSCM('\* \* \* \* \*') // Polls every minute

}

}

**Step 4: Save Changes**

* Save your job configuration after adding the messages.

**Step 5: Test the Pipeline View**

1. **Run the Pipeline**: Trigger a build by making another change to your application or manually starting it.
2. **View the Pipeline**:
   * Go back to the **MyWebAppPipelineView**.
   * You should see your pipeline stages represented visually.
   * Click on the stages to see the console output, which will include your user-defined messages.

**Conclusion**

Now you have a customized pipeline view in Jenkins with user-defined messages that provide clarity on the pipeline's progress. You can easily monitor the status of your builds and get feedback at each stage. If you have any further questions or need more customization, feel free to ask!

Implementing quality gates for static code analysis in your Jenkins pipeline helps ensure that code quality standards are met before allowing further builds or deployments. Here’s how you can integrate quality gates into your Jenkins pipeline using tools like SonarQube for static analysis.

**Step 1: Install SonarQube and Configure It**

1. **Install SonarQube**:
   * Download and install SonarQube on your server or use a cloud version.
   * Follow the official documentation to set it up.
2. **Create a SonarQube Project**:
   * Access the SonarQube dashboard.
   * Create a new project and note down the project key and token.
3. **Configure Quality Gates**:
   * In SonarQube, navigate to **Quality Gates** and create or modify a gate according to your requirements (e.g., coverage, bugs, vulnerabilities).

**Step 2: Install Required Plugins in Jenkins**

1. **Access Jenkins**: Open Jenkins in your web browser.
2. **Manage Plugins**:
   * Go to **Manage Jenkins** > **Manage Plugins**.
   * Ensure the following plugins are installed:
     + **SonarQube Scanner**.
     + **SonarQube Plugin**.

**Step 3: Configure Jenkins to Use SonarQube**

1. **Confi gure SonarQube in Jenkins**:
   * Go to **Manage Jenkins** > **Configure System**.
   * Scroll down to **SonarQube Servers** and add a new SonarQube server.
   * Enter the server URL and authentication token.
2. **Add SonarQube Scanner**:
   * In the same configuration section, under **SonarQube Scanner**, you may specify the installation path or let Jenkins install it automatically.

**Step 4: Update Your Jenkins Pipeline Script**

1. **Edit Your Pipeline Job**:
   * Navigate to your pipeline job (e.g., MyWebAppBuild).
   * Click **Configure**.
2. **Add SonarQube Analysis Stage**: Update your Jenkins pipeline script to include a stage for SonarQube analysis. Here’s an example:

Updated Jenkins Pipeline Script:

groovy

Copy code

pipeline {

agent any

stages {

stage('Checkout') {

steps {

echo 'Checking out the code...'

git 'https://your-repo-url.git'

}

}

stage('Build') {

steps {

echo 'Starting the build process...'

sh './build.sh'

echo 'Build completed successfully!'

}

}

stage('Static Code Analysis') {

steps {

script {

echo 'Starting SonarQube analysis...'

def scannerHome = tool 'SonarQubeScanner' // Ensure you have a SonarQube scanner tool configured

withSonarQubeEnv('SonarQubeServer') { // Use the name configured in Jenkins

sh "${scannerHome}/bin/sonar-scanner -Dsonar.projectKey=your\_project\_key -Dsonar.sources=. -Dsonar.host.url=http://your\_sonarqube\_url"

}

}

}

}

}

post {

success {

echo 'Pipeline completed successfully!'

script {

waitForQualityGate abortPipeline: true // This will check the quality gate status

}

}

failure {

echo 'Pipeline failed. Please check the logs.'

}

}

triggers {

pollSCM('\* \* \* \* \*') // Polls every minute

}

}

**Step 5: Save Changes**

* Save your job configuration after adding the static analysis stage.

**Step 6: Test the Pipeline with Quality Gates**

1. **Run the Pipeline**: Trigger a build by making a change to your application or manually starting it.
2. **Monitor SonarQube**: Check the SonarQube dashboard to see the analysis results.
3. **Verify Quality Gates**:
   * After the analysis, the pipeline will wait for the quality gate status.
   * If the quality gate fails, the pipeline will be aborted, and you will see a message in Jenkins.

**Conclusion**

You have successfully integrated quality gates for static analysis in your Jenkins pipeline using SonarQube. This setup will ensure that only code meeting your quality standards proceeds through the CI/CD process. If you have any further questions or need adjustments, feel free to ask!

Ex-5

Configuring a static code analyzer for your web application involves several steps. Here’s a step-by-step guide using **SonarQube** as an example static code analysis tool.

**Step 1: Set Up SonarQube**

1. **Download and Install SonarQube**:
   * Download SonarQube from the official website.
   * Follow the installation instructions for your operating system.
2. **Start SonarQube**:
   * Navigate to the SonarQube installation directory.
   * Start the SonarQube server:
     + For Linux/Mac:

bash

Copy code

./bin/linux-x86-64/sonar.sh start

* + - For Windows:

cmd

Copy code

.\bin\windows-x86-64\StartSonar.bat

1. **Access SonarQube Dashboard**:
   * Open a web browser and go to http://localhost:9000.
   * The default credentials are usually admin/admin.

**Step 2: Create a New Project in SonarQube**

1. **Log in to SonarQube**:
   * Use the default credentials to log in.
2. **Create a New Project**:
   * Click on **Create Project**.
   * Enter a project key and name.
   * Click **Create**.
3. **Generate a Token**:
   * After creating the project, you will be prompted to generate a token.
   * Copy this token; you’ll need it for the Jenkins integration later.

**Step 3: Configure Quality Profiles**

1. **Access Quality Profiles**:
   * In the SonarQube dashboard, navigate to **Quality Profiles**.
2. **Select or Create a Profile**:
   * You can choose from existing profiles or create a new one.
   * Click on **Create** to make a new profile. Select a language (e.g., JavaScript, Java) and give it a name.
3. **Activate Rules**:
   * After creating the profile, you can activate or deactivate rules according to your coding standards.
   * Browse through the rules and toggle them based on your requirements.
4. **Save Changes**: Make sure to save your profile settings.

**Step 4: Configure Quality Gates**

1. **Access Quality Gates**:
   * Navigate to **Quality Gates** in the SonarQube dashboard.
2. **Create a Quality Gate**:
   * Click on **Create**.
   * Define conditions such as code coverage percentage, number of bugs, code smells, etc.
   * Save the quality gate.

**Step 5: Set Up Your Jenkins Pipeline for Static Analysis**

1. **Install SonarQube Plugin in Jenkins**:
   * Go to **Manage Jenkins** > **Manage Plugins**.
   * Install the **SonarQube Scanner** and **SonarQube Plugin**.
2. **Configure SonarQube in Jenkins**:
   * Go to **Manage Jenkins** > **Configure System**.
   * Scroll to **SonarQube Servers** and add a new SonarQube server.
   * Provide the server URL (http://localhost:9000) and the authentication token you generated earlier.
3. **Add SonarQube Scanner**:
   * Ensure that the SonarQube scanner is configured in the same section.

**Step 6: Update Your Jenkins Pipeline Script**

1. **Edit Your Jenkins Pipeline**:
   * Navigate to your pipeline job (e.g., MyWebAppBuild).
   * Click **Configure**.
2. **Add Static Analysis Stage**: Update your Jenkins pipeline script to include a stage for SonarQube analysis. Here’s an example:

groovy

Copy code

pipeline {

agent any

stages {

stage('Checkout') {

steps {

echo 'Checking out the code...'

git 'https://your-repo-url.git'

}

}

stage('Build') {

steps {

echo 'Starting the build process...'

sh './build.sh'

echo 'Build completed successfully!'

}

}

stage('Static Code Analysis') {

steps {

script {

echo 'Starting SonarQube analysis...'

def scannerHome = tool 'SonarQubeScanner'

withSonarQubeEnv('SonarQubeServer') {

sh "${scannerHome}/bin/sonar-scanner -Dsonar.projectKey=your\_project\_key -Dsonar.sources=. -Dsonar.host.url=http://localhost:9000"

}

}

}

}

}

post {

success {

echo 'Pipeline completed successfully!'

script {

waitForQualityGate abortPipeline: true

}

}

failure {

echo 'Pipeline failed. Please check the logs.'

}

}

triggers {

pollSCM('\* \* \* \* \*') // Polls every minute

}

}

**Step 7: Run Your Pipeline**

1. **Trigger a Build**: Make a change in your code and push it to the repository, or manually trigger a build in Jenkins.
2. **Monitor the Analysis**:
   * After the build completes, go to the SonarQube dashboard.
   * Check the project to see the results of the static code analysis, including any identified issues.

**Step 8: Configure the Dashboard**

1. **Customize Dashboard**:
   * In SonarQube, navigate to your project dashboard.
   * You can customize what metrics and charts are displayed based on your team’s preferences.
   * Use the **Administration** section to configure project settings and visibility.

**Conclusion**

You have successfully configured a static code analyzer using SonarQube, set up quality profiles and gates, and integrated it into your Jenkins pipeline. This will help you maintain code quality by identifying inappropriate coding practices before they reach production. If you have any further questions or need more detailed steps, feel free to ask!

4o mini

Ex-6:

To create a build script using Maven that incorporates static analysis, unit testing, and deployment to a Tomcat server, you can follow these steps. Here’s a step-by-step guide, including folder structure and example script.

### Step 1: Create Folder Structure

Set up your project with the following folder structure:

bash

Copy code

/my-app

│

├── /src

│ ├── /main

│ │ ├── /java

│ │ └── /resources

│ └── /test

│ ├── /java

│ └── /resources

│

├── /lib # For any external libraries

│

├── /build # For build artifacts

│

├── pom.xml # Maven configuration file

│

└── build.sh # Build script

### Step 2: Create pom.xml

Your pom.xml file should define your project and its dependencies, as well as plugins for static analysis and deployment. Here’s a basic example:

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>

<properties>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

<dependencies>

<!-- Add your dependencies here -->

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.1</version>

</plugin>

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>findbugs-maven-plugin</artifactId>

<version>3.0.5</version>

<executions>

<execution>

<goals>

<goal>check</goal>

</goals>

</execution>

</executions>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.22.2</version>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-war-plugin</artifactId>

<version>3.3.1</version>

</plugin>

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat7-maven-plugin</artifactId>

<version>2.2</version>

<configuration>

<url>http://localhost:8080/manager/text</url>

<server>TomcatServer</server>

<path>/my-app</path>

</configuration>

</plugin>

</plugins>

</build>

</project>

### Step 3: Create the Build Script

Create a file named build.sh in the root of your project directory with the following content:

bash

Copy code

#!/bin/bash

# Exit immediately if a command exits with a non-zero status

set -e

# Print each command before executing it

set -x

# Step 1: Clean previous builds

mvn clean

# Step 2: Compile the application

mvn compile

# Step 3: Run static analysis

mvn findbugs:check

# Step 4: Run unit tests

mvn test

# Step 5: Package the application

mvn package

# Step 6: Deploy to Tomcat

mvn tomcat7:deploy

echo "Build and deployment completed successfully!"

### Step 4: Set Permissions

Make sure to set the correct permissions for your build script to make it executable:

bash

Copy code

chmod +x build.sh

### Step 5: Running the Build Script

To run the build process, navigate to the root directory of your project (/my-app) and execute:

bash

Copy code

./build.sh

### Step 6: Configure Tomcat Server

Make sure your Tomcat server is running and that you have configured the settings.xml in your Maven configuration directory (~/.m2) with your Tomcat credentials:

xml

Copy code

<servers>

<server>

<id>TomcatServer</id>

<username>your-username</username>

<password>your-password</password>

</server>

</servers>

### Conclusion

You now have a basic setup that uses Maven to handle building, testing, and deploying a Java application to Tomcat, along with static analysis checks. You can extend the pom.xml and build.sh as needed based on your project's requirements.

Windows:

Here's how to set up and execute a Maven build script on a Windows machine, including creating a folder structure and a build script.

**Step 1: Install Prerequisites**

1. **Install Java Development Kit (JDK)**:
   * Download and install the JDK from the [Oracle website](https://www.oracle.com/java/technologies/javase-jdk11-downloads.html) or [AdoptOpenJDK](https://adoptopenjdk.net/).
   * Set the JAVA\_HOME environment variable to the JDK installation path.
2. **Install Apache Maven**:
   * Download Maven from the [Apache Maven website](https://maven.apache.org/download.cgi).
   * Extract the downloaded zip file to a directory, e.g., C:\apache-maven-3.x.x.
   * Add C:\apache-maven-3.x.x\bin to your system's PATH environment variable.
3. **Install Tomcat** (if not already installed):
   * Download Tomcat from the [Apache Tomcat website](https://tomcat.apache.org/).
   * Extract it to a directory, e.g., C:\apache-tomcat-9.x.x.

**Step 2: Create Project Folder Structure**

Open a command prompt and create the folder structure as follows:

cmd

Copy code

mkdir my-app

cd my-app

mkdir src\main\java src\main\resources src\test\java src\test\resources lib build

**Step 3: Create pom.xml**

In the my-app directory, create a file named pom.xml with the following content:

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>

<properties>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

<dependencies>

<!-- Add your dependencies here -->

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.1</version>

</plugin>

<plugin>

<groupId>org.codehaus.mojo</groupId>

<artifactId>findbugs-maven-plugin</artifactId>

<version>3.0.5</version>

<executions>

<execution>

<goals>

<goal>check</goal>

</goals>

</execution>

</executions>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>2.22.2</version>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-war-plugin</artifactId>

<version>3.3.1</version>

</plugin>

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat7-maven-plugin</artifactId>

<version>2.2</version>

<configuration>

<url>http://localhost:8080/manager/text</url>

<server>TomcatServer</server>

<path>/my-app</path>

</configuration>

</plugin>

</plugins>

</build>

</project>

**Step 4: Create the Build Script**

Create a file named build.bat in the my-app directory with the following content:

bat

Copy code

@echo off

setlocal

rem Exit immediately if a command exits with a non-zero status

set ERRORLEVEL=0

rem Step 1: Clean previous builds

mvn clean

if %ERRORLEVEL% neq 0 exit /b %ERRORLEVEL%

rem Step 2: Compile the application

mvn compile

if %ERRORLEVEL% neq 0 exit /b %ERRORLEVEL%

rem Step 3: Run static analysis

mvn findbugs:check

if %ERRORLEVEL% neq 0 exit /b %ERRORLEVEL%

rem Step 4: Run unit tests

mvn test

if %ERRORLEVEL% neq 0 exit /b %ERRORLEVEL%

rem Step 5: Package the application

mvn package

if %ERRORLEVEL% neq 0 exit /b %ERRORLEVEL%

rem Step 6: Deploy to Tomcat

mvn tomcat7:deploy

if %ERRORLEVEL% neq 0 exit /b %ERRORLEVEL%

echo Build and deployment completed successfully!

endlocal

**Step 5: Configure Tomcat Server**

1. Ensure your Tomcat server is running.
2. Configure the settings.xml in your Maven configuration directory (C:\Users\<YourUsername>\.m2) with your Tomcat credentials:

xml

Copy code

<servers>

<server>

<id>TomcatServer</id>

<username>your-username</username>

<password>your-password</password>

</server>

</servers>

**Step 6: Running the Build Script**

To run the build process:

1. Open a command prompt.
2. Navigate to your project directory:

cmd

Copy code

cd C:\path\to\my-app

1. Execute the build script:

cmd

Copy code

build.bat

**Conclusion**

You've now set up a Windows environment for building a Maven application, including static analysis, unit testing, and deployment to Tomcat. You can expand upon this basic setup as needed for your project's specific requirements.

4o mini

Ex-5 updated:

**Step 1: Install Required Plugins**

1. **Open Jenkins**: Go to your Jenkins instance (usually http://localhost:8080).
2. **Manage Jenkins**: Click on “Manage Jenkins” from the left sidebar.
3. **Manage Plugins**: Select “Manage Plugins”.
4. **Available Tab**: Check for plugins like:
   * Git plugin
   * Pipeline
   * Any additional plugins you might need (e.g., Blue Ocean for a better UI).
5. **Install**: Select the plugins and click “Install without restart”.

**Step 2: Create a New Pipeline Job**

1. **New Item**: From the Jenkins dashboard, click on “New Item”.
2. **Name the Job**: Enter a name for your pipeline (e.g., MyApp-Pipeline).
3. **Select Pipeline**: Choose “Pipeline” and click “OK”.

**Step 3: Configure the Pipeline Job**

1. **Pipeline Configuration**:
   * In the configuration page, scroll down to the “Pipeline” section.
   * Choose "Pipeline script" and enter your pipeline script. Here's a basic example:

groovy

Copy code

pipeline {

agent any

stages {

stage('Clone') {

steps {

git 'https://your-repo-url.git'

}

}

stage('Build') {

steps {

bat 'path\\to\\your\\build-script.bat'

}

}

stage('Test') {

steps {

bat 'path\\to\\your\\test-script.bat'

}

}

}

}

1. **Save Configuration**: Click “Save” to apply your changes.

**Step 4: Configure Webhooks (if using GitHub/GitLab)**

1. **Webhook Setup**: Go to your repository on GitHub/GitLab.
2. **Settings**: Navigate to "Settings" > "Webhooks".
3. **Add Webhook**: Click “Add webhook”.
   * **Payload URL**: http://<your-jenkins-url>/git/notifyCommit?url=https://your-repo-url.git
   * **Content type**: application/json
   * **Triggers**: Select “Just the push event”.
4. **Save**: Click “Add webhook”.

**Step 5: Make Changes to the Application**

1. **Modify the Landing Page**:
   * Open your project in your preferred IDE.
   * Change the background color of the landing page (e.g., in a CSS file or directly in HTML).
2. **Commit and Push Changes**:
   * Commit your changes:

bash

Copy code

git add .

git commit -m "Changed background color of the landing page"

* + Push the changes to the remote repository:

bash

Copy code

git push origin main

**Step 6: Monitor Jenkins Pipeline**

1. **Jenkins Dashboard**: Go back to your Jenkins dashboard.
2. **Build History**: Check the build history for your pipeline job. You should see a new build triggered by the push event.
3. **Console Output**: Click on the build number to view the console output and verify that the build and other steps executed successfully.

**Step 7: Verify the Change**

1. **Access Your Web Application**: Open your web application in a browser.
2. **Check Background Color**: Ensure the landing page displays the new background color.

**Troubleshooting**

* If the pipeline does not trigger, verify:
  + The webhook is correctly configured.
  + Jenkins is accessible from the internet (if using a remote repo).
  + Jenkins logs for any errors.

By following these steps, you should have a Jenkins pipeline set up that builds and tests your application automatically upon changes.

Ex-8

**Step 1: Set Up Jenkins**

1. **Install Jenkins**: Ensure Jenkins is installed and running on your server.
2. **Install Necessary Plugins**: Go to Manage Jenkins > Manage Plugins, and ensure you have the following plugins installed:
   * Git Plugin (for VCS integration)
   * Pipeline Plugin (if not already included)

**Step 2: Create a New Jenkins Pipeline Job**

1. **Create a New Item**: Click on New Item in Jenkins.
2. **Enter Job Name**: Name your job (e.g., WebApp-Build).
3. **Select Pipeline**: Choose Pipeline and click OK.

**Step 3: Configure the Pipeline Job**

1. **Pipeline Configuration**:
   * In the job configuration page, scroll down to the Pipeline section.
   * Choose Pipeline script and enter your pipeline script. Here’s an example script:

groovy

Copy code

pipeline {

agent any

stages {

stage('Checkout') {

steps {

git url: 'https://your-vcs-repo.git', branch: 'main'

}

}

stage('Build') {

steps {

sh 'chmod +x build\_script.sh'

sh './build\_script.sh'

}

}

stage('Deploy') {

steps {

sh './deploy\_script.sh'

}

}

}

stage('Test') {

steps {

echo 'Running tests...'

script {

def testResult = sh(script: './run\_tests.sh', returnStatus: true)

if (testResult != 0) {

error 'Tests failed. Please fix the issues.'

}

}

}

}

triggers {

pollSCM('H/5 \* \* \* \*') // Polls the VCS every 5 minutes

}

}

* + Update the repository URL and scripts accordingly.

1. **Save Configuration**: Click Save.

**Step 4: Configure Webhook (Optional)**

If you want immediate triggers instead of polling:

1. **Configure Webhook in VCS**:
   * Go to your repository settings in your version control system (e.g., GitHub, GitLab).
   * Add a webhook pointing to your Jenkins server: http://<your-jenkins-url>/github-webhook/
   * Choose application/json as the content type.

**Step 5: Make a Change to the Application**

1. **Modify the Background Color**:
   * Open your web application’s landing page CSS file (e.g., styles.css).
   * Change the background color (e.g., from background-color: white; to background-color: blue;).
   * Save the changes and commit them to your repository:

bash

Copy code

git add styles.css

git commit -m "Changed background color of landing page"

git push origin main

**Step 6: Verify Pipeline Execution**

1. **Check Jenkins Dashboard**: Go back to the Jenkins dashboard.
2. **View Job Execution**: Look for the WebApp-Build job. If the webhook is set up correctly, it should have triggered a build automatically. If you are using polling, wait a few minutes and check the job again.
3. **Check Console Output**: Click on the job to view the console output. Ensure that all stages completed successfully.

**Step 7: Validate the Deployment**

1. **Access the Application**: Open your web application in a browser.
2. **Verify the Change**: Check if the background color of the landing page has changed to the new color.

**Troubleshooting**

* If the pipeline does not run:
  + Check Jenkins logs for errors.
  + Ensure the VCS webhook is set up correctly and reachable.
  + Verify that your Jenkins has permissions to access the repository.

This setup allows your Jenkins pipeline to run automatically on code changes, ensuring that your application is always up to date with the latest changes.