**DAY-3**

**DEVOPS**

**NAME : GOWTHAM[24MCRO28]**

STEP -1 :INSTALL DOCKER

1. sudo apt update
2. sudo apt install -y docker.io

STEP 2: ENABLE AND DISABLE

1) sudo systemctl enable docker

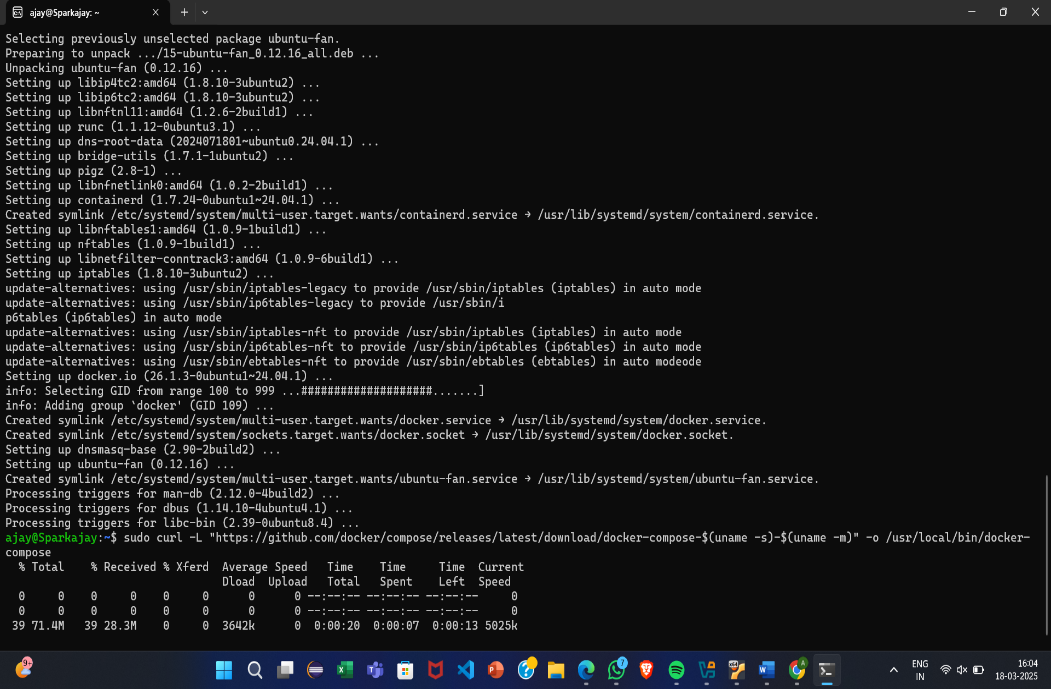
2)sudo systemctl start docker

STEP 3:VERIFY THE INSTALLATION:

docker –version

STEP 4:INSTALL DOCKER COMPOSE

sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose



CREATE AN “HELLO WOLRD: APPLICATION

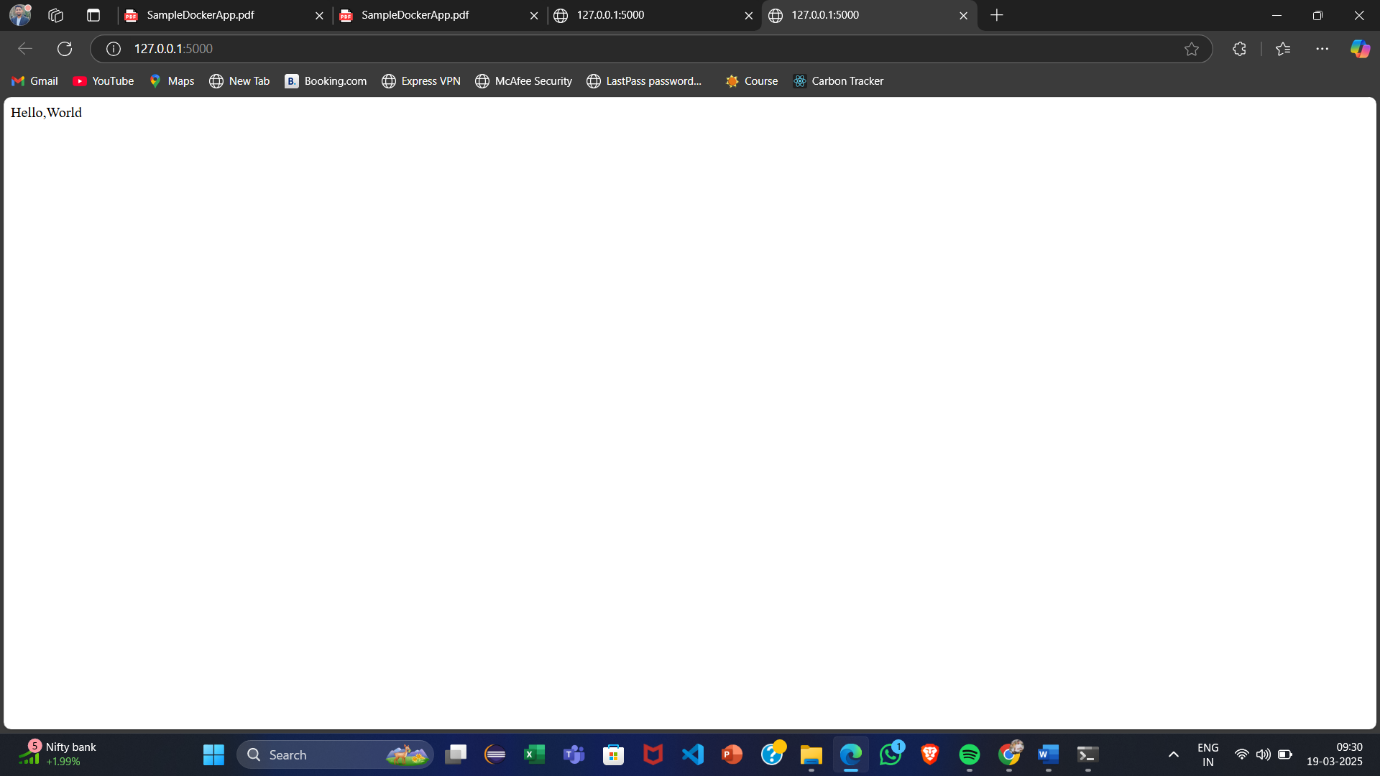
Create a project directory



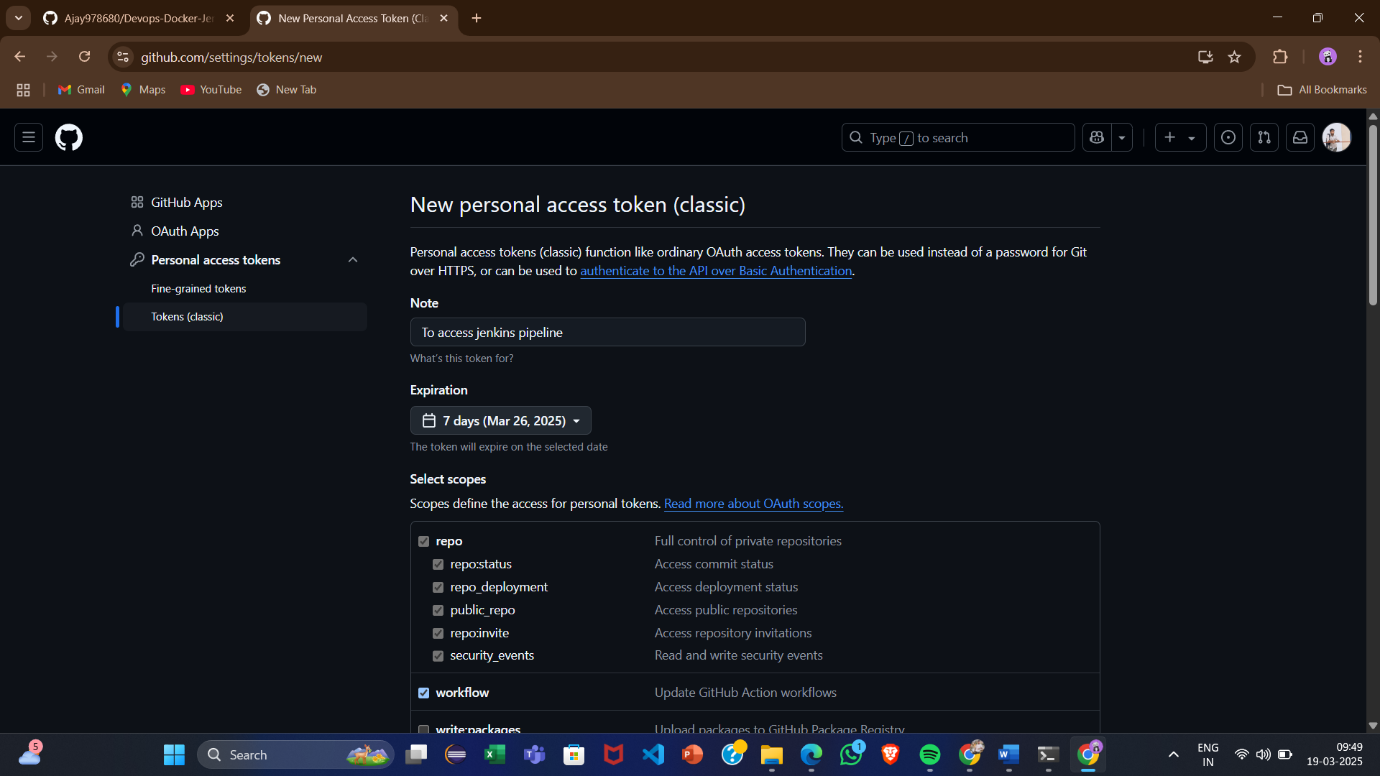
Create the python Application File

Create a file

To Run an Docker



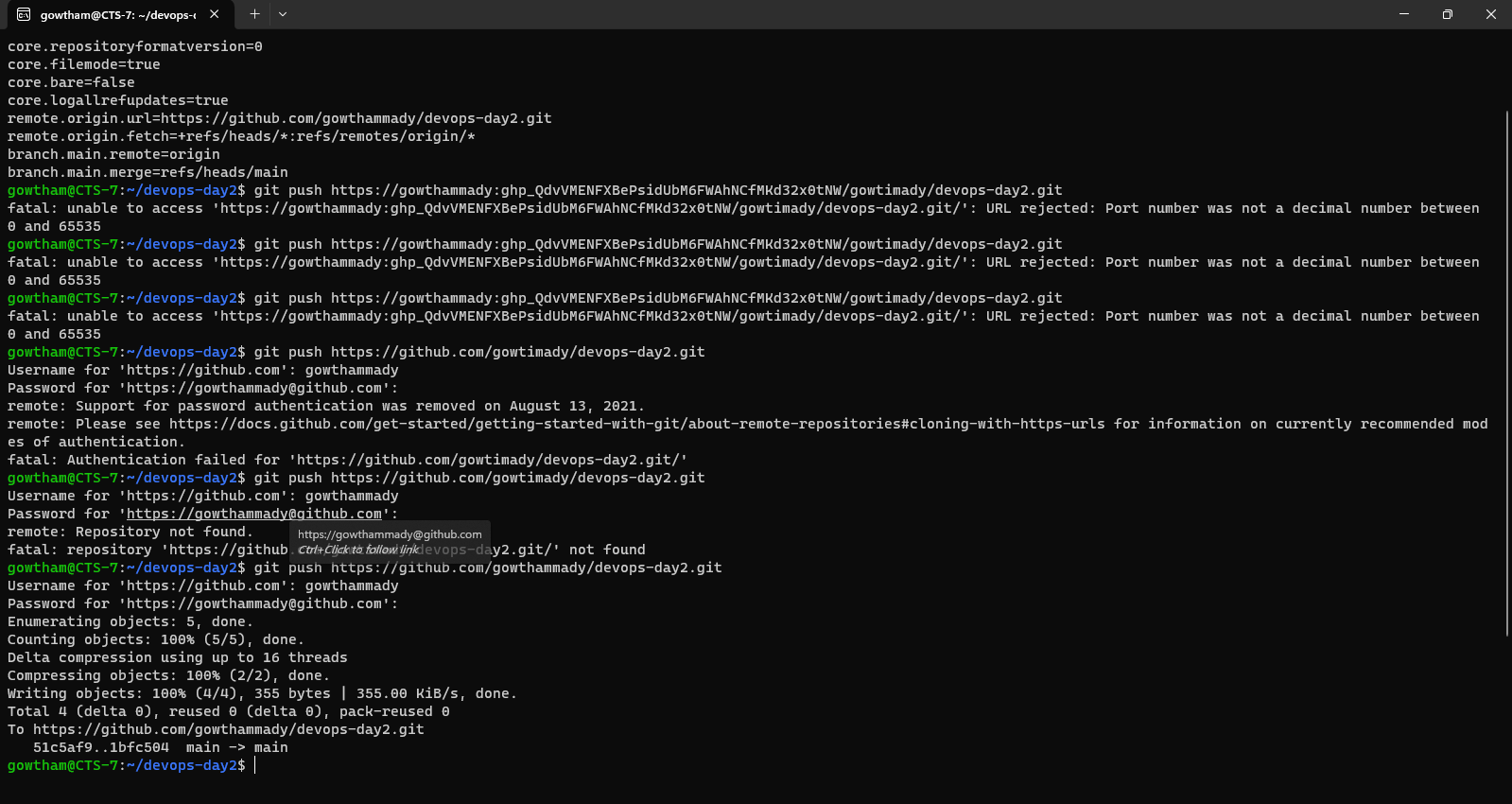
**Devops Jenkins-Docker**

****

**Click on Add Credentials and Fill the details**

In First Time, it Will have Password, in that we will give github token for it.

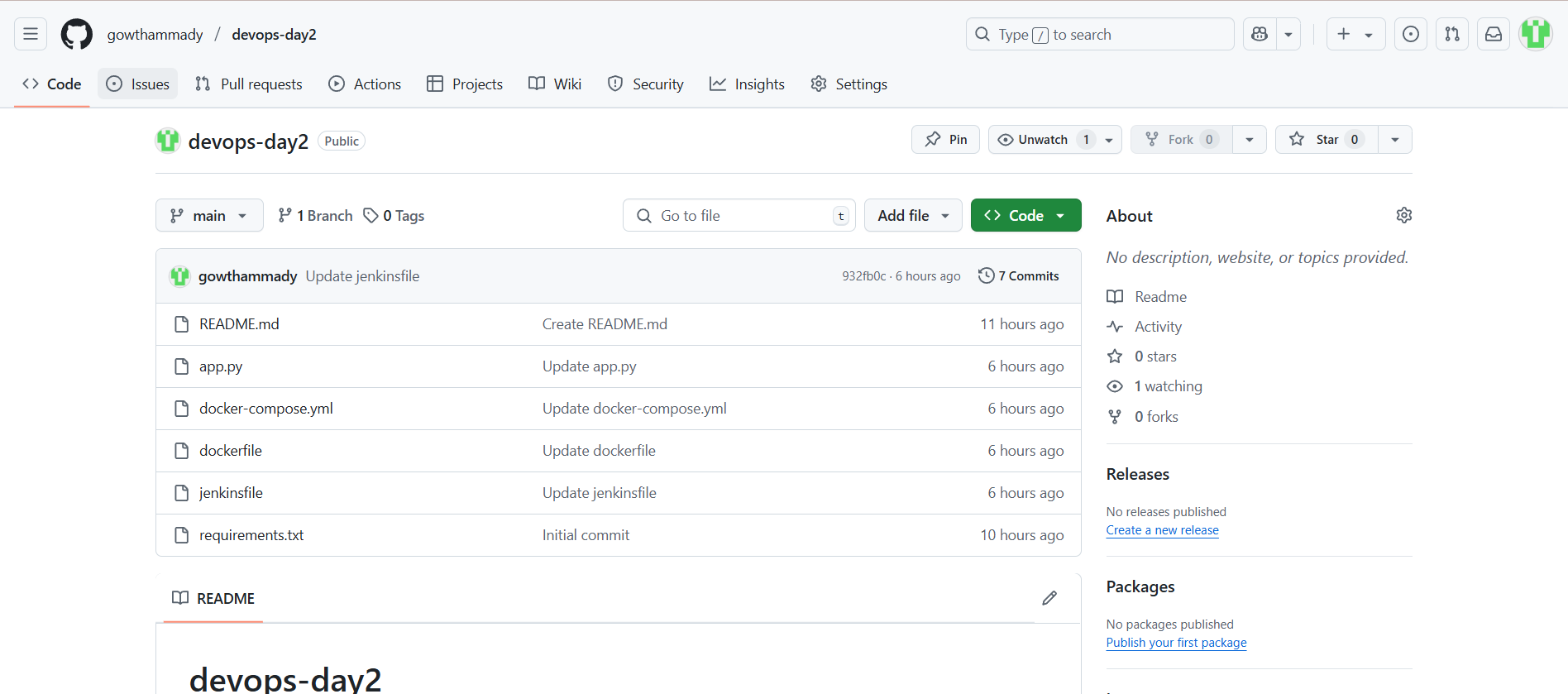
**Clone the Git Repo in Terminal:**

****

And move all other file to github repo folder.

Git fetch – Remote repo Change and haven’t pulled in local.

**The pushed files appear like this :**

****

**Step 1: Generate a Git Personal Access Token**

Before configuring the Jenkins pipeline, you need to generate a **Personal Access Token (PAT)** from your Git service.

**GitHub (Example)**

1. **Log in to GitHub** and navigate to your profile.
2. Go to **Settings** > **Developer Settings** > **Personal Access Tokens**.
3. Click **Generate New Token**.
4. Select the necessary permissions for the token. For example, to clone repositories, select:
   * repo (full control of private repositories)
   * read:org (for organization repository access)
5. Generate the token and **copy it**. This token will act as the password when Jenkins connects to GitHub.

**GitLab (Example)**

1. **Log in to GitLab** and go to **Profile Settings** > **Access Tokens**.
2. Generate a new token with appropriate scopes (e.g., read\_repository).
3. **Save the token** to use in Jenkins.

**Bitbucket (Example)**

1. **Log in to Bitbucket** and go to **Personal Settings** > **App Passwords**.
2. Create an app password with necessary permissions (like repository read).
3. **Save the password** to use in Jenkins.

**Step 2: Store Git Token in Jenkins Credentials**

Once you've generated the Git token, the next step is to store it securely in Jenkins.

1. **Log in to Jenkins** and navigate to the Jenkins dashboard.
2. In the left menu, click on **Manage Jenkins**.
3. Click on **Manage Credentials**.
4. Select the appropriate **scope** (e.g., (Global)).
5. Click on **Add Credentials**.
6. In the **Kind** dropdown, select **Username with password**.
7. In the **Username** field, enter your Git username (e.g., your-username for GitHub).
8. In the **Password** field, paste the **Git token** you generated.
9. Optionally, give it an ID (e.g., git-token-jenkins).
10. Click **OK** to save the credentials.

**Step 3: Configure Jenkins Pipeline**

Now that the Git token is securely stored in Jenkins, you can configure a Jenkins pipeline to use it for Git interactions.

**Example Pipeline Script (Declarative Pipeline)**

You’ll now set up a pipeline that uses Git for the source code. Here’s an example using a declarative pipeline.

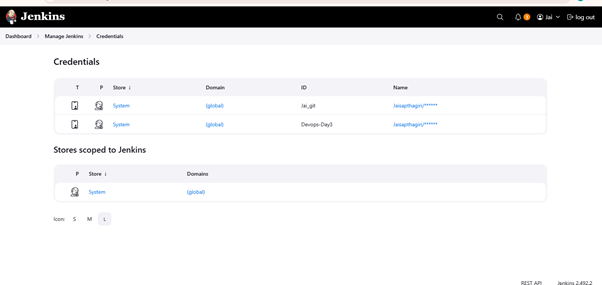
1. **Create a New Pipeline Job**:
   * Go to Jenkins Dashboard.
   * Click **New Item**, select **Pipeline**, and name your pipeline (e.g., Git-Pipeline).
   * Click **OK**.
2. **Configure the Pipeline**:
   * In the pipeline configuration, scroll to the **Pipeline** section.
   * Choose **Pipeline script from SCM**.
   * Set the **SCM** dropdown to **Git**.
   * In the **Repository URL** field, enter your repository URL (e.g., https://github.com/yourusername/your-repository.git).
   * Select **Credentials**. Choose the credentials you created earlier (e.g., git-token-jenkins).

**Step 4: Run the Jenkins Pipeline**

* After configuring the pipeline, click **Save** and then **Build Now** to run the pipeline.
* Jenkins will use the credentials you provided to authenticate with Git, clone the repository, and run the pipeline steps.

**Step 5: Monitor and Troubleshoot**

If the pipeline fails, check the Jenkins job's **Console Output** for debugging information. Common issues can be due to incorrect credentials, Git URL, or permission issues.



**Jenkins Pipeline for Dockerized Application Deployment**

This document provides a step-by-step guide on how the Jenkins pipeline automates the process of fetching the code from GitHub, building a Docker image, pushing it to a container registry, and deploying the application in a running Docker container.

**Pipeline Overview**

The pipeline follows these key steps:

1. **Checkout Code** - Fetch the latest code from the GitHub repository.
2. **Build Docker Image** - Create a Docker image for the application.
3. **Login to Docker Registry** - Authenticate to the container registry.
4. **Push to Container Registry** - Upload the built image to a Docker registry.
5. **Stop & Remove Existing Container** - Stop and remove any existing container with the same name.
6. **Run Docker Container** - Deploy a new container with the updated image.
7. **Post Actions** - Handle success or failure messages.

**Step-by-Step Execution**

**1. Checkout Code**

* Uses Jenkins credentials to authenticate and fetch the latest code from GitHub.
* Ensures secure access using stored credentials instead of exposing raw tokens.

**Implementation:**

stage('Checkout Code') {

steps {

withCredentials([usernamePassword(credentialsId: 'github-nisanthg1010', usernameVariable: 'GIT\_USER', passwordVariable: 'GIT\_TOKEN')]) {

git url: "https://$GIT\_USER:$GIT\_TOKEN@github.com/nisanthg1010/Devops\_Nisanth.git", branch: 'main'

}

}

}

**2. Build Docker Image**

* Builds the Docker image using the Dockerfile present in the repository.
* Tags the image with the latest version.

**Implementation:**

stage('Build Docker Image') {

steps {

sh 'docker build -t $DOCKER\_IMAGE .'

}

}

**3. Login to Docker Registry**

* Uses stored Jenkins credentials to log in securely to the Docker registry.
* Prevents exposing login credentials in the script.

**Implementation:**

stage('Login to Docker Registry') {

steps {

withCredentials([usernamePassword(credentialsId: 'docker\_nisanth', usernameVariable: 'DOCKER\_USER', passwordVariable: 'DOCKER\_PASS')]) {

sh 'echo $DOCKER\_PASS | docker login -u $DOCKER\_USER --password-stdin'

}

}

}

**4. Push to Container Registry**

* Pushes the newly built Docker image to the specified container registry.
* Ensures the latest version of the application is stored and accessible.

**Implementation:**

stage('Push to Container Registry') {

steps {

sh 'docker push $DOCKER\_IMAGE'

}

}

**5. Stop & Remove Existing Container**

* Stops and removes the running container if it exists.
* Prevents conflicts when deploying the new version.

**Implementation:**

stage('Stop & Remove Existing Container') {

steps {

script {

sh '''

if [ "$(docker ps -aq -f name=$CONTAINER\_NAME)" ]; then

docker stop $CONTAINER\_NAME || true

docker rm $CONTAINER\_NAME || true

fi

'''

}

}

}

**6. Run Docker Container**

* Starts a new Docker container with the updated image.
* Maps the internal application port 5000 to 5001 on the host machine.

**Implementation:**

stage('Run Docker Container') {

steps {

sh 'docker run -d -p 5001:5000 --name $CONTAINER\_NAME $DOCKER\_IMAGE'

}

}

**7. Post Actions**

* If successful, displays a success message.
* If failed, displays an error message.

**Implementation:**

post {

success {

echo "Build, push, and container execution successful!"

}

failure {

echo "Build or container execution failed."

}

}

**Conclusion**

This Jenkins pipeline automates the entire process of fetching the code, building a Docker image, pushing it to a registry, and deploying the container. It ensures a seamless CI/CD workflow, making application updates smooth and efficient.

