Day-2 GOKILA N

step-by-step guide to setting up a simple Python "Hello, Docker!" Flask application using Docker and Docker Compose.

---

1. Install Docker

First, install Docker to get the Docker engine running on your system:

sudo apt install -y docker.io

- **Explanation:** Installs Docker on your system using the apt package manager. The -y flag auto-confirms any prompts.

---

2. Start and Enable Docker Service

Start the Docker service and enable it to start automatically at boot time:

sudo systemctl start docker

sudo systemctl enable docker

- **Explanation:** The start command starts the Docker daemon, and enable ensures Docker runs on startup.

---

3. Verify Docker Installation

Verify that Docker was installed correctly by checking its version:

docker --version

- **Explanation:** Displays the installed Docker version to confirm the installation.

---

4. Install Docker Compose

Now, install Docker Compose, a tool to define and manage multi-container Docker applications:

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

sudo chmod +x /usr/local/bin/docker-compose

- **Explanation:** The first command downloads the latest Docker Compose binary, and the second command makes it executable.

---

5. Verify Docker Compose Installation

Check the installed version of Docker Compose:

docker-compose --version

- **Explanation:** Displays the installed Docker Compose version to verify the installation.

---

6. Create Project Directory

Create a directory for your project and navigate into it:

mkdir ~/docker-python-app

cd ~/docker-python-app

- **Explanation:** Creates a directory for your project and navigates into it.

---

7. Create the app.py file

Create a Python file app.py for the Flask application:

nano app.py

Paste the following Flask application code:

```
from flask import Flask


app = Flask(__name__)


@app.route('/')
def hello_world():
    return 'Hello, world Running inside the docker!'


if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

- **Explanation:** A simple Flask app with one route (/) that returns a greeting message. The Flask server listens on all interfaces (0.0.0.0) and port 5000.

---

8. Create requirements.txt

Create a requirements.txt file to list Python dependencies:

nano requirements.txt

Add the following content:

flask

- **Explanation:** Lists the Flask library as the required dependency for your project.

---

9. Install pip (if not already installed)

Ensure pip is installed to handle Python package installations:

sudo apt update

sudo apt install python3-pip

- **Explanation:** Updates the package list and installs pip to handle Python packages.

---

10. Create Dockerfile

Create a Dockerfile that defines how the Docker image should be built:

nano Dockerfile

Add the following content:

```
# Use the official Python image from Docker Hub
FROM python:3.9-slim

# Set the working directory inside the container
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Make port 5000 available to the world outside the container
EXPOSE 5000

# Define the environment variable for Flask to run in production mode
ENV FLASK_ENV=production
```

Run app.py when the container launches

CMD ["python", "app.py"]

- **Explanation:** This Dockerfile defines the Python environment, installs dependencies, exposes port 5000, and starts the Flask app inside the container.

---

11. Create docker-compose.yml

Create a docker-compose.yml file to manage the application's services:

nano docker-compose.yml

Add the following content:

version: '3.8'

services:

  web:

    build: .

    ports:

      - "5000:5000"

    environment:

      - FLASK_ENV=development

    volumes:

      - .:/app

    restart: always

- **Explanation:** This Compose file:
    - Defines the web service.
    - Builds the image from the current directory.
    - Maps port 5000 from the host to the container.
    - Mounts the current directory (.) into the container to enable live code reloading.
    - Restarts the container if it crashes.

---

12. Add User to Docker Group (if needed)

To avoid using sudo with Docker commands, add your user to the Docker group:

sudo usermod -aG docker $USER

newgrp docker

- **Explanation:** The first command adds your user to the Docker group, and the second command applies the changes to your current session.

---

13. Build and Run the Application

Now, you can build and start the Flask app container using Docker Compose:

docker-compose up --build

- **Explanation:** This command builds the Docker image and starts the container based on the docker-compose.yml configuration. The --build flag forces a rebuild of the Docker image.

---

14. Access the Application

Once the container is running, open your browser and navigate to:

http://localhost:5000

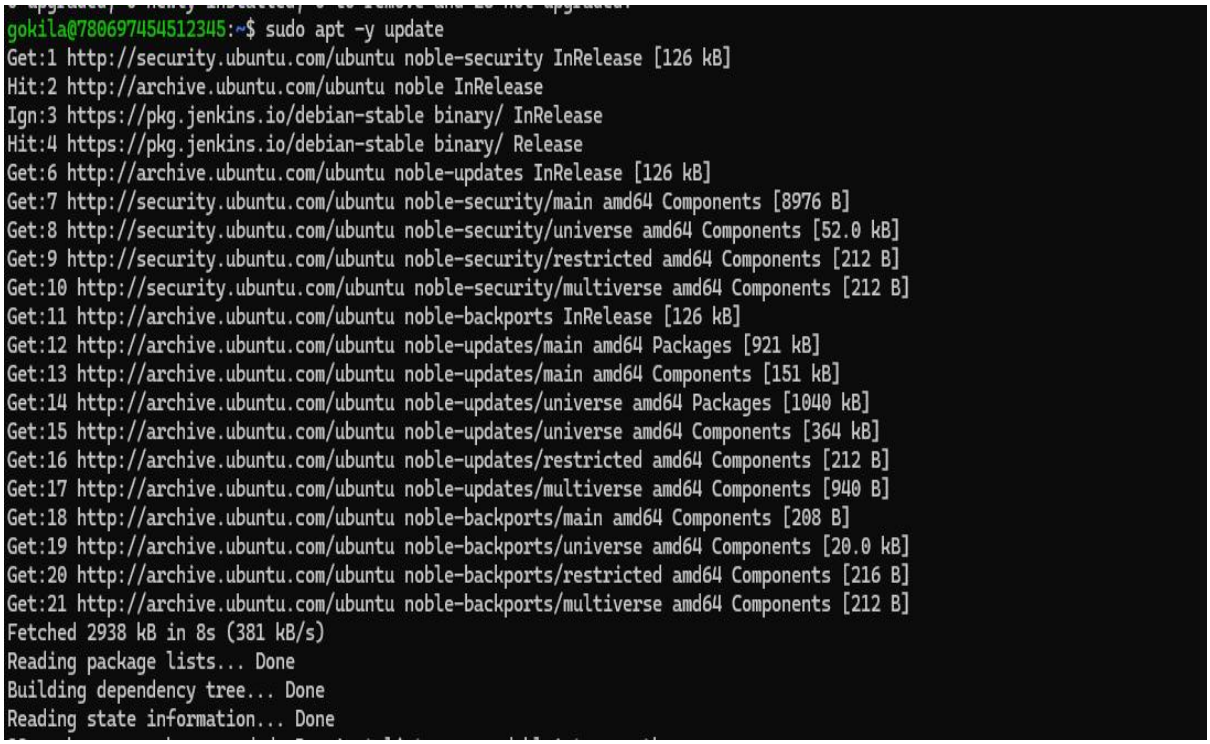
You should see the message: **"Hello, Docker Python App!"**

Summary of Commands

1. Install Docker:
2. sudo apt install -y docker.io
3. Start and enable Docker service:
4. sudo systemctl start docker
5. sudo systemctl enable docker
6. Install Docker Compose:
7. sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
8. sudo chmod +x /usr/local/bin/docker-compose
9. Create project directory:
10. mkdir ~/docker-python-app
11. cd ~/docker-python-app
12. Create app.py with Flask code.
13. Create requirements.txt with flask.

14. Install pip (if needed):
15. sudo apt update
16. sudo apt install python3-pip
17. Create Dockerfile with the configuration.
18. Create docker-compose.yml with service definition.
19. Add your user to the Docker group (if necessary):
20. sudo usermod -aG docker $USER
21. newgrp docker
22. Build and run the app:
23. docker-compose up --build

Now your "Hello, Docker!" Flask app should be running inside a Docker container, accessible at http://localhost:5000.



```
gokila@780697454512345:~$ sudo apt install docker.io
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
docker.io is already the newest version (26.1.3-0ubuntu1~24.04.1).
0 upgraded, 0 newly installed, 0 to remove and 28 not upgraded.
gokila@780697454512345:~$ sudo systemctl start docker
gokila@780697454512345:~$ sudo systemctl enable docker
gokila@780697454512345:~$ docker --version
Docker version 26.1.3, build 26.1.3-0ubuntu1~24.04.1
gokila@780697454512345:~$ sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname -m)" -o /usr/l
ocal/bin/docker-compose
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
  0     0    0     0    0     0      0      0 --:--:--  0:00:02 --:--:--     0Warning: Failed to open the file /usr/local/bin/docker-compose: Text f
ile busy
  0 71.4M    0     0    0     0      0      0 --:--:--  0:00:02 --:--:--     0
```

```
gokila@780697454512345:~/goki$ cat app.py
from flask import Flask

app = Flask(__name__)

@app.route("/")
def home():
    return "Flask is running inside Docker!"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

```
gokila@780697454512345:~/goki$ cat requirements.txt
flask
gokila@780697454512345:~/goki$ cat nano dockerfile
cat: nano: No such file or directory
FROM python:3.12.3

WORKDIR /app

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 5000

CMD [ "python", "app.py" ]
gokila@780697454512345:~/goki$ cat docker-compose.yml
version: '3.8'

services:
  web:
    build: .
    ports:
      - "5000:5000"

    volumes:
      - .:/app

    restart: always
```

```
nisanth@LAPTOP-G3U57BJM:~/docker-app$ sudo docker ps
CONTAINER ID   IMAGE                  COMMAND           CREATED        STATUS          PORTS                                       NAMES
a9a9d9673cb3   docker-python-app-web  "python app.py"   14 hours ago   Up 26 minutes   0.0.0.0:5000->5000/tcp, :::5000->5000/tcp   docker-python-app-web-1
```

```
gokila@780697454512345:~/goki$ sudo docker-compose build
WARN[0000] /home/gokila/goki/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it to avoid p
otential confusion
Compose can now delegate builds to bake for better performance.
 To do so, set COMPOSE_BAKE=true.
[+] Building 0.2s (11/11) FINISHED                                                                                    docker:default
 => [web internal] load build definition from dockerfile                                                                        0.0s
 => => transferring dockerfile: 200B                                                                                            0.0s
 => [web internal] load metadata for docker.io/library/python:3.12.3                                                            0.0s
 => [web internal] load .dockerignore                                                                                           0.0s
 => => transferring context: 2B                                                                                                 0.0s
 => [web 1/5] FROM docker.io/library/python:3.12.3                                                                              0.0s
 => [web internal] load build context                                                                                           0.0s
 => => transferring context: 37.53kB                                                                                            0.0s
 => CACHED [web 2/5] WORKDIR /app                                                                                               0.0s
 => CACHED [web 3/5] COPY requirements.txt .                                                                                    0.0s
 => CACHED [web 4/5] RUN pip install --no-cache-dir -r requirements.txt                                                         0.0s
 => [web 5/5] COPY . .                                                                                                          0.0s
 => [web] exporting to image                                                                                                    0.0s
 => => exporting layers                                                                                                         0.0s
 => => writing image sha256:e8440673633cd4963f1d70236109d9824f1d13b0840329d2fe30e7c50a36a67a                                    0.0s
 => => naming to docker.io/library/goki-web                                                                                     0.0s
 => [web] resolving provenance for metadata file                                                                                0.0s
[+] Building 1/1
 ✔web  Built                                                                                                                    0.0s
```

Hello, Docker Python App!

Jenkins Pipeline Through Git Token - Setup Procedure

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.

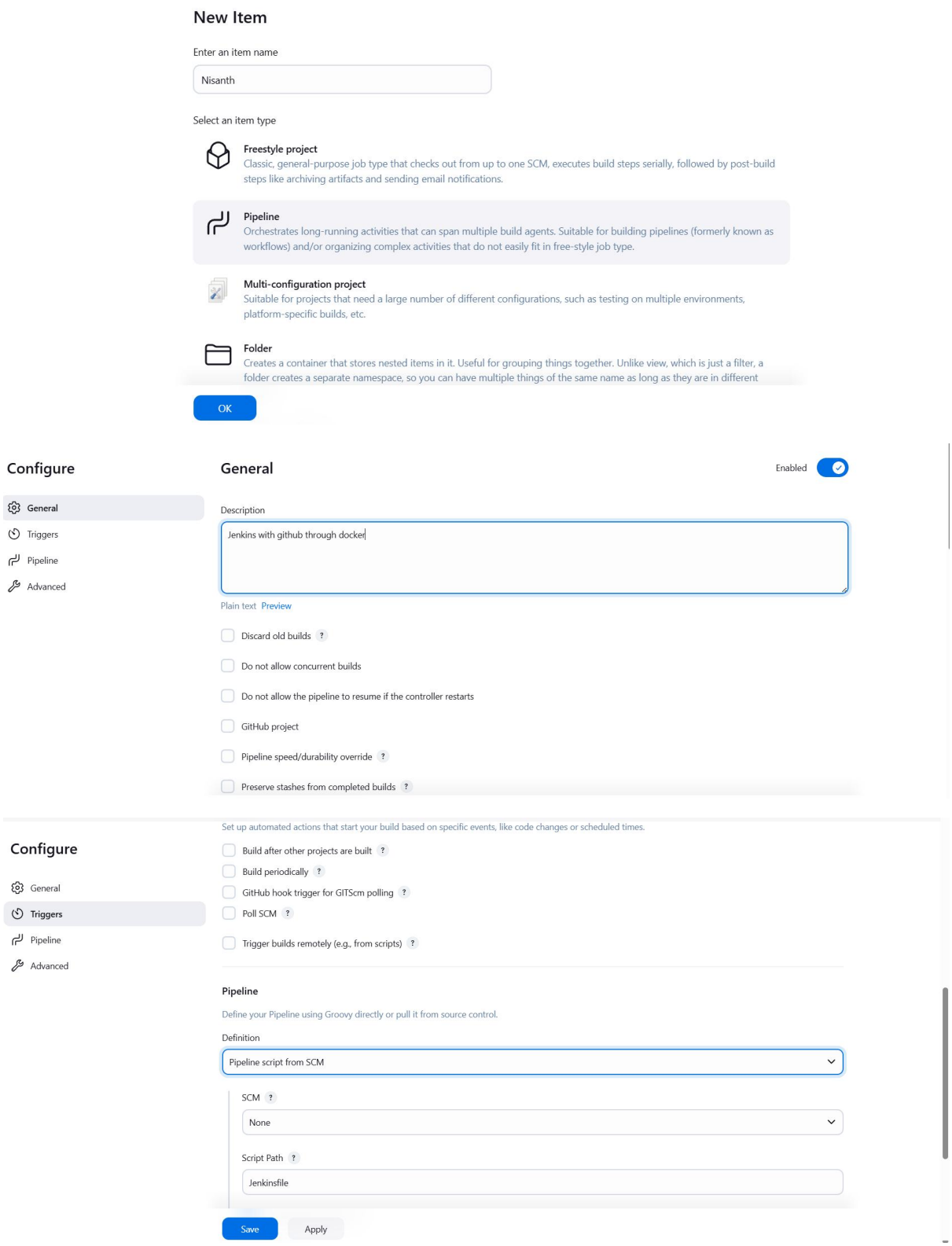
1. **Create a New Pipeline Job**:
    - Go to Jenkins Dashboard.
    - Click **New Item**, select **Pipeline**, and name your pipeline (e.g., Git-Pipeline).
    - Click **OK**.
2. **Configure the Pipeline**:
    - In the pipeline configuration, scroll to the **Pipeline** section.
    - Choose **Pipeline 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).

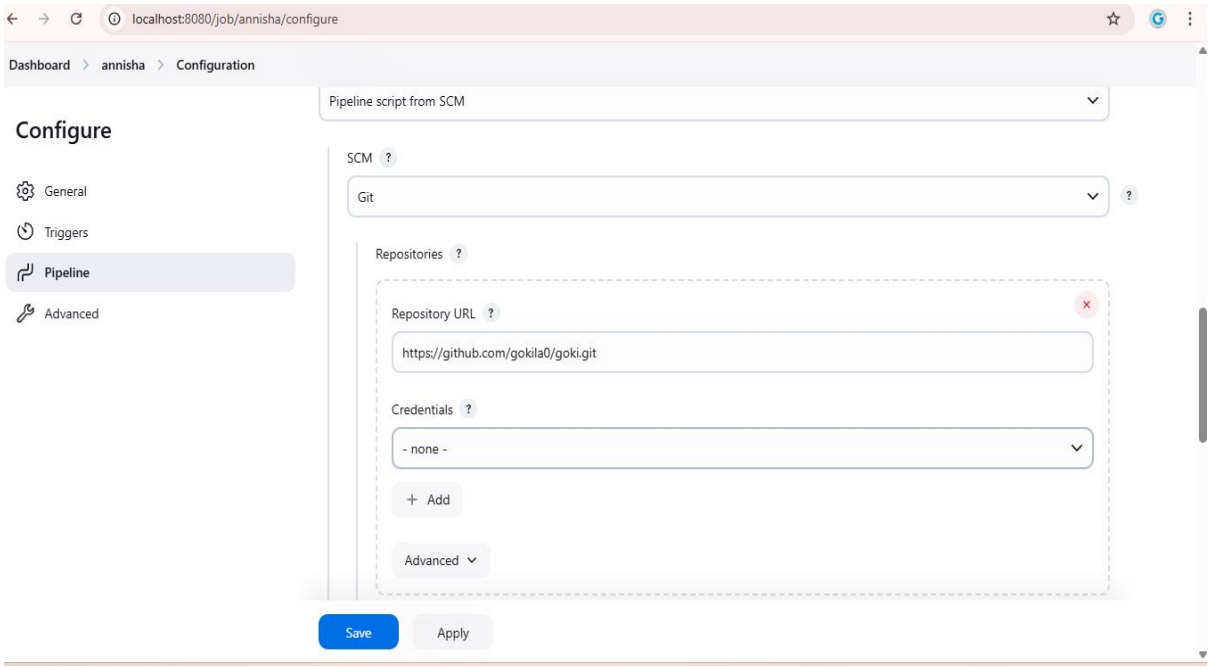
Step 4: Run the Jenkins Pipeline

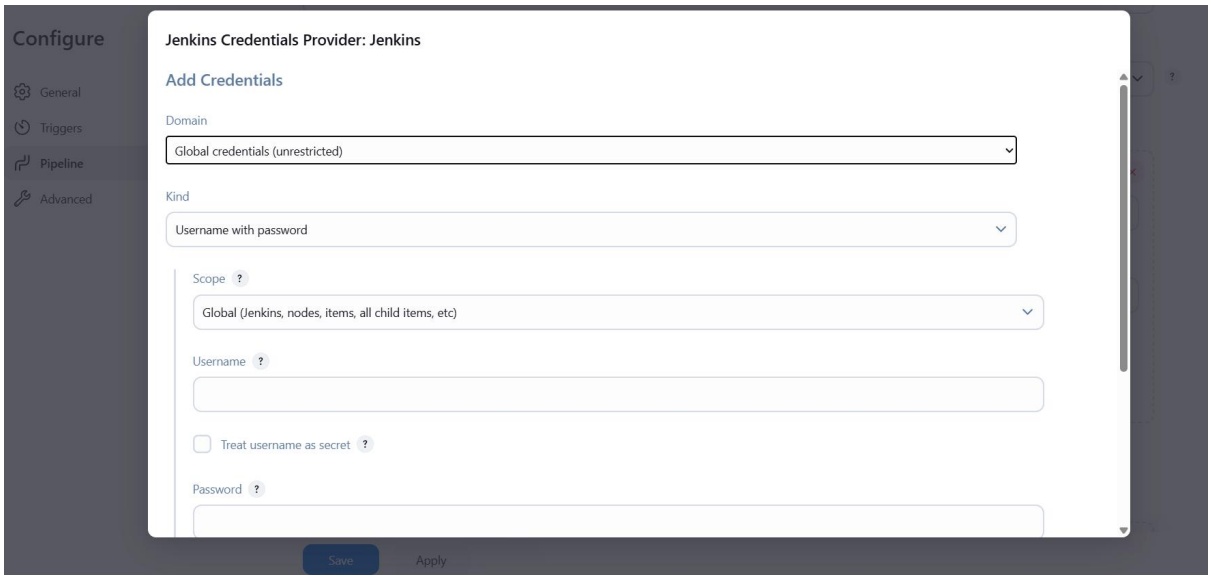
- After configuring the pipeline, click **Save** and then **Build Now** to run the pipeline.
- Jenkins will use the credentials you provided to authenticate with Git, clone the repository, and run the pipeline steps.

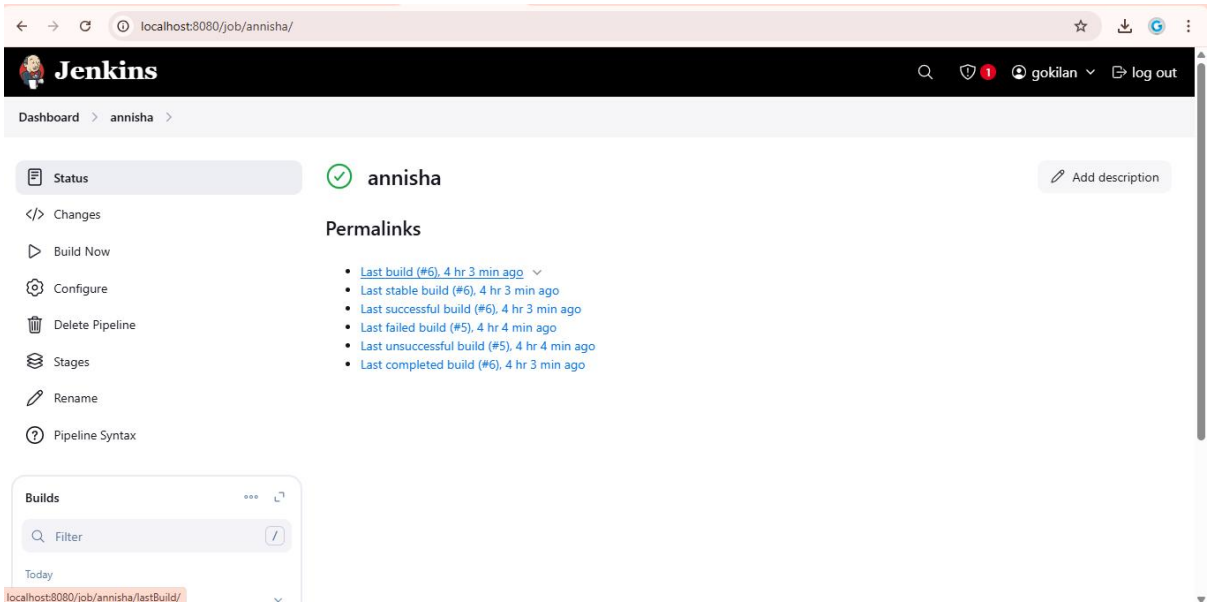
Step 5: Monitor and Troubleshoot

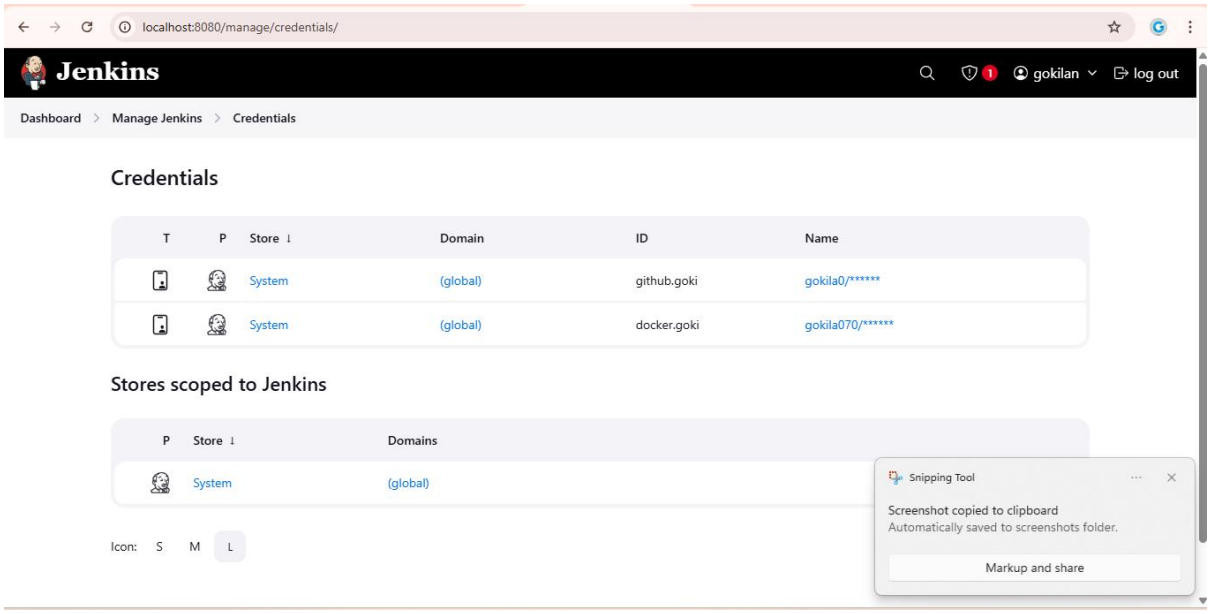
- If the pipeline fails, check the Jenkins job's **Console Output** for debugging information. Common issues can be due to incorrect credentials, Git URL, or permission issues.

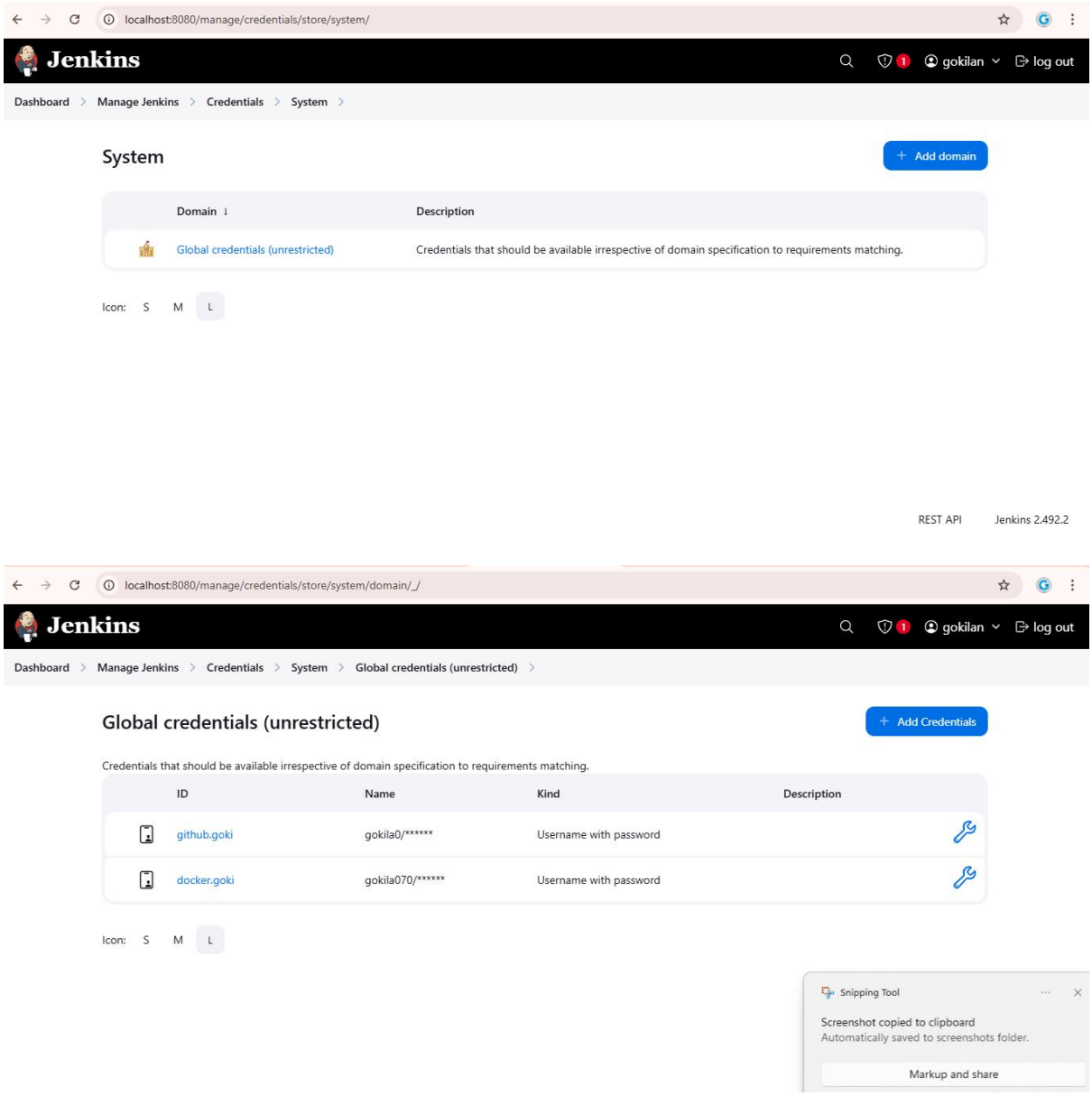



localhost:8080/job/annisha/configure
Dashboard > annisha > Configuration
Configure
General
Triggers
Pipeline
Advanced
Pipeline script from SCM
SCM
Git
Repositories
Repository URL
https://github.com/gokila0/goki.git
Credentials
- none -
+ Add
Advanced
Save
Apply


Configure
General
Triggers
Pipeline
Advanced
Jenkins Credentials Provider: Jenkins
Add Credentials
Domain
Global credentials (unrestricted)
Kind
Username with password
Scope
Global (Jenkins, nodes, items, all child items, etc)
Username
Treat username as secret
Password
Save
Apply


localhost:8080/job/annisha/
Jenkins
gokilan
log out
Dashboard
annisha
Status
Changes
Build Now
Configure
Delete Pipeline
Stages
Rename
Pipeline Syntax
annisha
Add description
Permalinks
Last build (#6), 4 hr 3 min ago
Last stable build (#6), 4 hr 3 min ago
Last successful build (#6), 4 hr 3 min ago
Last failed build (#5), 4 hr 4 min ago
Last unsuccessful build (#5), 4 hr 4 min ago
Last completed build (#6), 4 hr 3 min ago
Builds
Filter
Today
localhost:8080/job/annisha/lastBuild/


localhost:8080/manage/credentials/
Jenkins
gokilan
log out
Dashboard
Manage Jenkins
Credentials
Credentials
T
P
Store
Domain
ID
Name
System
(global)
github.goki
gokila0/******
System
(global)
docker.goki
gokila070/******
Stores scoped to Jenkins
P
Store
Domains
System
(global)
Icon: S M L
Snipping Tool
Screenshot copied to clipboard
Automatically saved to screenshots folder.
Markup and share



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. 🚀

Hello, Docker Python App!