SAMPLE JENKINS CI-CD PIPELINE API

We are going to study on how to setup a Jenkins and docker base CI CD pipeline for an API project.

- Jenkins is used as automation server (Separated server)
- Separate hosted server for the application
- Git and Github
- Docker and Docker compose
- Doppler as the wallet for environment variables

<div style="border:1px solid black">

curl ifconfig.me

display public ipv4 adddr of server

</div>

Jenkins hosted server – 20.193.255.237 Azure vm
Application hosted server - 20.193.137.245 Azure vm

Initial Setup on client VM ================================

1. install required packages

install git >

```
sudo apt update
sudo apt install -y git
```

install docker > (https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-20-04**)**

```
lsb_release -a ( check os version )

sudo apt install apt-transport-https ca-certificates curl software-properties-common
```

Then add the GPG key for the official Docker repository to your system:
```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

Add the Docker repository to APT sources:
sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu focal stable"

apt-cache policy docker-ce

sudo apt install docker-ce

sudo systemctl status docker

Output● docker.service - Docker Application Container Engine
    Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset:
enabled)
    Active: active (running) since Tue 2020-05-19 17:00:41 UTC; 17s ago
TriggeredBy: ● docker.socket
     Docs: https://docs.docker.com
  Main PID: 24321 (dockerd)
     Tasks: 8
    Memory: 46.4M
    CGroup: /system.slice/docker.service
        └─24321 /usr/bin/dockerd -H fd:// --
containerd=/run/containerd/containerd.sock

Check the command
sudo docker ps

install docker compose > (https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-compose-on-ubuntu-20-04**)**

sudo curl -L
"https://github.com/docker/compose/releases/download/1.29.2/docker-compose-
$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

Next, set the correct permissions so that the docker-compose command is
executable:
sudo chmod +x /usr/local/bin/docker-compose

To verify that the installation was successful, you can run:

docker-compose --version

You'll see output similar to this:
Output
docker-compose version 1.29.2, build 5becea4c

2. install Doppler CLI

------------ # Debian 11+ / Ubuntu 22.04+ -------------------
sudo apt-get update && sudo apt-get install -y apt-transport-https ca-certificates curl gnupg curl -sLf --retry 3 --tlsv1.2 --proto "=https" 'https://packages.doppler.com/public/cli/gpg.DE2A7741A397C129.key' | sudo gpg --dearmor -o /usr/share/keyrings/doppler-archive-keyring.gpg

echo "deb [signed-by=/usr/share/keyrings/doppler-archive-keyring.gpg] https://packages.doppler.com/public/cli/deb/debian any-version main" | sudo tee /etc/apt/sources.list.d/doppler-cli.list

sudo apt-get update && sudo apt-get install doppler

------------- # Older versions of Debian/Ubuntu -----------------
sudo apt-get update && sudo apt-get install -y apt-transport-https ca-certificates curl gnupg curl -sLf --retry 3 --tlsv1.2 --proto "=https" 'https://packages.doppler.com/public/cli/gpg.DE2A7741A397C129.key' | sudo apt-key add –

echo "deb https://packages.doppler.com/public/cli/deb/debian any-version main" | sudo tee /etc/apt/sources.list.d/doppler-cli.list

sudo apt-get update && sudo apt-get install doppler

Now, verify the Doppler CLI was installed by checking its version.
doppler --version
You can also upgrade the CLI to the latest version at any time.
doppler update

chamalka@free-vm-student-profile:~$ doppler --version
v3.75.1
chamalka@free-vm-student-profile:~$ docker --version
Docker version 28.1.1, build 4eba377
chamalka@free-vm-student-profile:~$ docker-compose --version
docker-compose version 1.29.2, build 5becea4c
chamalka@free-vm-student-profile:~$

3. Configure password less Sudo access

Some Jenkins and automations tasks may require password less Sudo access

full password less access (with caution)

edit the sudoers file.
sudo visudo
Add the following line at the end
<username> ALL=(ALL) NOPASSWD: ALL

> You can test it by running a sudo command:
>
> sudo ls /root

This gives user full root privileges without a password. Better to use only in Controlled or trusted environments

password less access for specific commands (recommended)

for more secure setups, restrict sudo to only required commands

<username> ALL=(ALL) NOPASSWD:/usr/bin/tee /etc/nginx/sites-available/*, /usr/bin/ln -s.........

• **Security Risk:** Granting passwordless sudo access can be a security risk, especially if the account is compromised. Use it with caution.
• **Scope:** You can limit the scope of passwordless sudo by specifying particular commands instead of ALL. For example:
john ALL=(ALL) NOPASSWD: /usr/bin/apt-get, /usr/bin/systemctl
This configuration would allow the user john to run apt-get and systemctl commands without a password, but not other sudo commands.
By following these steps, you can configure passwordless sudo access on your Linux system.

4. Configure GitHub as a trusted source

Automatically trust the host

ssh-keyscan github.com >> ~/.ssh/known_hosts

For sometimes, need to run git clone first time

git clone git@github.com:<profile-name>/<repo-name>.git

5. Enable Git Pull via SSH

To allow the VM to pull code from a private GitHub repository, generate an SSH key on the client VM and add it to GitHub:

ssh-keygen -t rsa -b 4096 -C "example@jenkins"
cat ~/.ssh/id_rsa.pub

Then:
1. Go to GitHub - Settings -> SSH and GPG Keys - New SSH Key
2. Title it (e.g., example-test)
3. Paste the public key and click Add

6. Grant Docker Access to SSH User

sudo usermod -aG docker <username> (whoami for find the username)

Apply the group change without rebooting:

newgrp docker

7. Setup nginx

Install nginx

sudo apt install nginx

Enable nginx

ssh -o StrictHostKeyChecking=no <username>@<client-vm-ip> "sudo mkdir -p /etc/nginx/sit..................

Install certbot

sudo apt install certbot python3-certbot-nginx

Jenkins VM Setup ====================================== (Separate server)

1. Generate SSH Key Pair on Jenkins Server

ssh-keygen -t rsa -b 4096 -C "jenkins@azure-deploy"

File location: Press Enter (default ~/.ssh/id_rsa)
Passphrase: Leave empty

This generates:

. ~/.ssh/id_rsa - Private key

. ~/.ssh/id_rsa.pub - Public key

2. Copy Public Key to Client VM

ssh-copy-id -i ~/.ssh/id_rsa.pub <client-vm-username>@<client-vm-ip>
example - ssh-copy-id -i ~/.ssh/id_rsa.pub example-user-name@200.206.521.221

Accept the connection when prompted
Enter the Clients VM password

output:

Number of key(s) added: 1

Now try logging into the machine, with: "ssh 'example-user-name@200.206.521.221'"
and check to make sure that only the key(s) you wanted were added.

3. Install Jenkins on Jenkins server

We are going to download and install Jenkins on Jenkins server. Open source repository contains a Docker Compose configuration for a quick installation of Jenkins. But this setup is not great idea for production systems.

Since this is a fresh ubuntu VM, we need to first install docker and docker-compose, we can follow up the same previous guideline to install those.

https://github.com/vdespa/install-jenkins-docker

Clone this repository
git clone git @github.com..............

Open a terminal window in the same directory where the Dockerfile from this repository is located. Build the Jenkins Docker image:

sudo docker build -t my-jenkins .

Start Jenkins:
Sudo docker compose up -d

check the running container status and detailer
sudo docker ps -a

If you wish to stop Jenkins and get back to it later, run:
docker compose down

Once you are done playing with Jenkins, maybe it is time to clean things up.
Run the following command to terminate Jenkins and to remove all volumes and images used:
docker compose down --volumes --rmi all

If the container was successfully up and running now we can navigate into Jenkins dashboard. By default, Jenkins run on 8080 port.
http://<localhost / server public ipv4 address>:8080

But this may be not work (this url may be mnot reached), Because this is a fresh server and not configured nginx. So we are trying to publicly access port (8080). This port can be closed to outside by the server vendors. Therefore we can set inbound configuration in cloud vendor dashboard and open the port to the public outside. But this will be secure vulnerable since the port expose the public world. The best approach is install nginx and configure nginx configuration to 8080 port.

Install nginx same as previous.

Configure nginx configure file (I am configuring default config file)

Initially I configured to http://ipaddr/jenkins -> but this Jenkins rewriting path rule caused into 500 internal server error

looked nginx logs and found this error.

rewrite or internal redirection cycle while internally redirecting to "/login///////////"

This means:

- Jenkins redirects /jenkins → /login
- Nginx **rewrites /login back to /**
- Jenkins again redirects → /login
- **Infinite rewrite loop**
- Nginx gives up → **500 Internal Server Error**

This is caused by **trying to force Jenkins under /jenkins using rewrites**
Jenkins **does not tolerate this well** unless it is started with a context path.

Set Jenkins URL (VERY IMPORTANT)
In Jenkins UI:
Manage Jenkins → Configure System
Set:
Jenkins URL: http://20.193.255.237

Jenkins works BEST at /
Running it under /jenkins without changing Jenkins startup args **will always cause problems** (exactly what you're seeing).

Serve Jenkins at ROOT /
URL:
http://200.193.255.200
following is the new configuration for serve Jenkins -> sudo docker restart <imgName>

```
location / {
    proxy_pass http://localhost:8080;
    proxy_redirect off;

    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";

    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;

    proxy_buffering off;
}
```

Open your web browser and navigate to http://localhost:8080 (or the port you configured). The **"Unlock Jenkins"** page will appear, prompting for the administrator password.

Retrieve the Initial Admin Password:
The password is automatically generated and stored within the container. You can retrieve it using one of the following methods:

From the Docker logs (easiest): View the logs of your running container. The password will be displayed there during the initial startup.

docker logs <container_name_or_id>

Then unlock Jenkins dashboard and complete the initial setup.

4. Jenkins setup

Install SSH Agent Plugin in Jenkins

. Go to Manage Jenkins - Plugins
. Search for SSH Agent Plugin
. Install and restart Jenkins if required

Add SSH Private Key to Jenkins Credentials

. Go to Manage Jenkins - Credentials - (global) - Add Credentials

. Select:

  Kind: SSH Username with private key
  Username: Jenkins vm username
  Private Key: Select Enter directly, then run: (In Jenkins VM)
  cat ~/.ssh/id_rsa   (copy including header&footer)
  and paste the output.
  ID: azure-ssh-key (must match your pipeline)
  Description: SSH key for Azure VM deployment
  Click Save.

Skip for now

```
pipeline {
  agent any
  options {
    disableConcurrentBuilds()
  }

  parameters {
    string(name: 'REMOTE_HOST', defaultValue: '20.106.51.21', description: 'Remote Azure VM IP')
    string(name: 'REMOTE_USER', defaultValue: 'icieos', description: 'Remote SSH Username')
    string(name: 'APP_DIR', defaultValue: '/home/icieos/supreme-edits-api', description: 'App Directory on Remote VM')
    string(name: 'REPO_URL', defaultValue: 'git@github.com:icieos-enterprise-org/supreme-edits-api.git', description: 'Git Repository URL')
    string(name: 'BRANCH', defaultValue: 'main', description: 'Git Branch to deploy')
    string(name: 'DOPPLER_PROJECT', defaultValue: 'supreme-edits-api', description: 'Doppler Project Name')
    string(name: 'DOPPLER_CONFIG', defaultValue: 'prd', description: 'Doppler Config (env)')
  }

  environment {
    SSH_OPTIONS = "-o StrictHostKeyChecking=no"
  }

  stages {
    stage('Validate SSH Connection') {
      steps {
        sshagent(credentials: ['azure-ssh-key']) {
          sh "ssh ${env.SSH_OPTIONS} ${params.REMOTE_USER}@${params.REMOTE_HOST} 'echo ✅ SSH connection successful'"
        }
      }
    }

    stage('Clone or Pull Code on Remote VM') {
      steps {
        sshagent(credentials: ['azure-ssh-key']) {
          sh """
            ssh ${env.SSH_OPTIONS} ${params.REMOTE_USER}@${params.REMOTE_HOST} '
            set -e
            if [ ! -d "${params.APP_DIR}/.git" ]; then
              echo " 📦 Cloning repository from branch: ${params.BRANCH}"
              rm -rf ${params.APP_DIR}
              git clone --branch ${params.BRANCH} ${params.REPO_URL} ${params.APP_DIR}
            else
              echo " 🔄 Pulling latest code from branch: ${params.BRANCH}"
              cd ${params.APP_DIR}
              git fetch origin
              git checkout ${params.BRANCH}
              git reset --hard origin/${params.BRANCH}
            fi
            '
          """
        }
      }
    }

    stage('Fetch .env from Doppler') {
      steps {
        withCredentials([string(credentialsId: 'doppler-token', variable: 'DOPPLER_TOKEN')]) {
          sshagent(credentials: ['azure-ssh-key']) {
            sh """
              ssh ${env.SSH_OPTIONS} ${params.REMOTE_USER}@${params.REMOTE_HOST} '
              set -e
              export DOPPLER_TOKEN=${DOPPLER_TOKEN}
              echo " 🔐 Fetching environment variables from Doppler..."
              cd ${params.APP_DIR}
              doppler secrets download --project ${params.DOPPLER_PROJECT} --config ${params.DOPPLER_CONFIG} --format env --no-file > .env
              echo " ✅ .env file created"
              '
            """
          }
        }
      }
    }
```

Rest of the script

```
pipeline {

 stages {

  stage('Docker Compose Build & Deploy') {
   steps {
    sshagent(credentials: ['azure-ssh-key']) {
     sh """
      ssh ${env.SSH_OPTIONS} ${params.REMOTE_USER}@${params.REMOTE_HOST} '
       set -e
       echo " 🛠 Running docker-compose with memory limit..."
       cd ${params.APP_DIR}
       export DOCKER_BUILDKIT=1
       docker-compose down --remove-orphans
       docker-compose build --memory=4g || docker compose build --memory=4g
       docker-compose up -d || docker compose up -d
       echo " 🚀 Application deployed successfully"

       # Cleanup unused Docker resources
       echo "Cleaning up unused Docker resources..."
       docker system prune -a --volumes -f
      '
     """
    }
   }
  }
 }

 post {
  always {
   cleanWs()
  }
 }
}
```

5. Sign up for doppler & create workplace & project & config and add secret variables

6. Jenkins dashboard – Create new pipeline

New item
Enter item name & select pipeline
Then need to configure the pipeline

Do not allow concurrent builds
This project is parameterized -> add string parameters

REMOTE_HOST
20.193.137.245
Remote Azure VM IP - client test vm

REMOTE_USER
chamalka
remote ssh username of client test vm

APP_DIR
/home/chamalka/jenkins-hosted-apps
App Directory on Remote client test VM

REPO_URL
git@github.com:chamalkaMarasinghe/sample-jenkins-ci-cd-pipeline-API.git
Git Repository URL

BRANCH
main
Git Branch to deploy

DOPPLER_PROJECT
testing-project
Doppler Project Name

DOPPLER_CONFIG
dev_sample_jenkins_cicd_pipeline_api
Doppler Config (env)

You can trigger a Jenkins pipeline remotely by using an **API token** for authentication when sending an HTTP request. The process involves generating a user-specific API token and configuring the job to accept remote triggers with a designated authentication token.

Step 1: Generate a User API Token

1. Log in to Jenkins and click on your username in the top-right corner, then select **Configure/security**.
2. In the **API Token** section, click **Add new Token** and then **Generate**.
3. **Copy the generated token immediately**, as it will not be shown again. This token will act as your password for remote access.

Step 2: Configure the Jenkins Job/Pipeline

1. Navigate to the configuration page of the specific pipeline job you want to trigger remotely.
2. Scroll to the **Build Triggers** section and check the box for **Trigger builds remotely (e.g., from scripts)**.
3. Enter a unique text string as the **Authentication Token** in the provided field. This token is different from your user API token and is specific to the job.

Step 3: Trigger the Build Remotely

You can now use a script or a command-line tool like curl to send a POST request to the Jenkins API endpoint. Use your Jenkins username and the **user API token** (from Step 1) for authentication, and include the **job token** (from Step 2) in the URL.

For a simple build (no parameters):

curl -X POST
http://<username>:<user_api_token>@<jenkins_url>/job/<job_name>/build?token=<job_token>

Replace <username>, <user_api_token>, <jenkins_url>, <job_name>, and <job_token> with your actual details.

For a parameterized build:

Use the /buildWithParameters endpoint and pass parameters in the request body or as query parameters. The following example uses query parameters

curl -X POST
"http://<username>:<user_api_token>@<jenkins_url>/job/<job_name>/buildWithParameters?token=<job_token>&PARAMETER_NAME=PARAMETER_VALUE"

Note on Security:
For security, it is highly recommended to use HTTPS to encrypt credentials over the network.
When using curl, you might need to use the -I flag or ensure preemptive authentication is used
(which curl handles by prepending the credentials to the URL as shown above) to avoid 403 Forbidden errors.

Alternatively, you can provide the credentials using the -u option in curl
curl -X POST http://<jenkins_url>/job/<job_name>/build?token=<job_token> -u
<username>:<user_api_token>

7. Doppler created service token for accessibility

A Doppler Service Token provides read-only secrets access to a specific config within a project.
It adheres to the principle of least privilege by ensuring an application only has access to a single config within a project for use in live environments.

❗ Don't use a CLI or Personal Token in live environments as it provides write access with the same permissions as the account it was created by.

Requirements
- Doppler CLI
- Access to the config for a project you wish to provide access to

Doppler Dashboard
To generate a Service Token using the dashboard
1. Go to the Project and select a Config
2. Click the **Access** tab.
3. Click on **Generate**.
4. Provide a name for the token and optionally provide the token with write access or assign an expiration.
5. Click on **Generate Service Token**
6. Copy the Service Token as it is only shown once.

Next this token should be added to Jenkins credentials.

Go to Manage Jenkins - Credentials - (global) - Add Credentials

Select:
  Kind: secret text
  Username: DOPPLER_TOKEN
  Password: doppler service token (copied from the doppler dahsboard)
  ID: doppler-token
  Description: doppler token for pull doppler secrets
  Click Save.

8. Some of repository, Jenkins and nginx configuration

GitHub Actions Configure

By icieos admin · 1 min · See views · Add a reaction

Jenkins Setup

1. Enable Remote Trigger & Create Secret Token

- Go to your pipeline job → `Configure`
- Check ✅ **"Trigger builds remotely"**
- Add a **token** (This can be anything)

2. Enable Remote Trigger & Create Secret Token

- Click on the username → Security
- Scroll to **"API Token"** section.
- Click **"Add new Token"** → Give it a name → Click **"Generate"**.
- **Copy and save**

3. Add Environment Secrets

Name	Value
JENKINS_API_TOKEN	
JENKINS_TRIGGER_SECRET	

GitHub Setup

1. Add GitHub Environment Variables

- Go to your GitHub repository.
- Click **Settings** → **Environments** (in the sidebar).
- Click **"New environment"** → name it (e.g., `prod`).
- In the environment, click **"Add environment variable"**

2. Add Environment Variable

Name	Value
REMOTE_HOST	Client VM IP → `20.106.51.21`
REMOTE_USER	Client VM Host → `icieos`
APP_DIR	/home/<client-vm-host>/<project-name> `/home/icieos/kumzits-api`
REPO_URL	`git@github.com:icieos-enterprise-org/kumzits-api.git`
BRANCH	`main`
DOPPLER_PROJECT	`kumzits-api`
DOPPLER_CONFIG	`prd`

🔒 **Secrets**

- `JENKINS_API_TOKEN` : Jenkins API token (user-specific)
- `JENKINS_TRIGGER_SECRET` : Jenkins trigger token

⚙️ **Environment Variables**

- `JENKINS_USER` : Jenkins username (e.g., `icieos`)
- `REMOTE_HOST` : Target VM IP (e.g., `20.106.x.x`)
- `REMOTE_USER` : Remote SSH username (e.g., `icieos`)
- `APP_DIR` : App directory on remote VM (e.g., `/home/icieos/your-app`)
- `REPO_URL` : GitHub repo URL
- `BRANCH` : `develop` (for QA) or `main` (for Prod)
- `DOPPLER_PROJECT` : `client-projects`
- `DOPPLER_CONFIG` : `dev`, `stg`, or `prod` based on the environment

🚨 **Root cause (confirmed) (in nginx config)**

These two lines are **breaking normal HTTP requests** like /crumbIssuer:

proxy_set_header Upgrade $http_upgrade;

proxy_set_header Connection "upgrade";

Why this causes 400 Bad Request

Those headers are **ONLY for WebSockets**.

most correct ngix config for Jenkins without http headers upgrade like for scokets

```
location / {
    proxy_pass http://localhost:8080;
    proxy_redirect off;

    proxy_http_version 1.1;

    # REQUIRED HEADERS
    proxy_set_header Host $host;
    proxy_set_header Authorization $http_authorization;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;

     # Jenkins stability
    proxy_buffering off;
    client_max_body_size 0;
}
```

manage Jenkins -> security
CSRF Protection
Crumb Issuer
Enable proxy compatibility

9. Docker and docker compose files

docker file (node js api)

```
# Use an official Node runtime based on Alpine
FROM node:18-alpine

# Set the working directory in the container
WORKDIR /usr/src/app

# Copy the package.json and package-lock.json (if available)
COPY package*.json ./

# Install dependencies
RUN npm ci --only=production

# Copy the rest of the application code
COPY . .

# Bind the app to port 5005
EXPOSE 5005

# Define the command to run the app
CMD [ "npm", "run", "dev" ]
```

docker-compose.yml

```
version: "3.8"

services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "5005:5005"
    environment:
      - REDIS_HOST=redis
      - REDIS_PORT=6379
    depends_on:
      - redis
    volumes:
      - .:/usr/src/app
      - /usr/src/app/node_modules

  redis:
    image: redis:alpine
```

docker-compose-prod.yml

```
version: "3.8"

services:
  app:
    image: icieos/kids-plan-api:latest
    ports:
      - target: 5005
        published: 5005
        protocol: tcp
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure
      resources:
        limits:
          memory: "1G"
    networks:
      - app-network

  redis:
    image: redis:alpine
    deploy:
      placement:
        constraints: [node.role == manager]
    networks:
      - app-network

networks:
  app-network:
    driver: overlay
```

10. Configure github actions and workflows.

.github/workflows/dev-deployement.yml

```yaml
name: Trigger Jenkinss

on:
 push:
  branches:
   - main

jobs:
 trigger-jenkins:
  runs-on: ubuntu-latest
  environment: dev

  steps:
   - name: Call Jenkins Build and Deploy Pipeline
    run: |
     curl -X POST
"http://20.193.255.237/job/Build%20and%20deploy/buildWithParameters" \
      --user "${{ vars.JENKINS_USER }}:${{ secrets.JENKINS_API_TOKEN }}" \
      --data-urlencode token=${{ secrets.JENKINS_TRIGGER_SECRET }} \
      --data-urlencode REMOTE_HOST=${{ vars.REMOTE_HOST }} \
      --data-urlencode REMOTE_USER=${{ vars.REMOTE_USER }} \
      --data-urlencode APP_DIR=${{ vars.APP_DIR }} \
      --data-urlencode REPO_URL=${{ vars.REPO_URL }} \
      --data-urlencode BRANCH=${{ vars.BRANCH }} \
      --data-urlencode DOPPLER_PROJECT=${{ vars.DOPPLER_PROJECT }} \
      --data-urlencode DOPPLER_CONFIG=${{ vars.DOPPLER_CONFIG }}
```

Add these docker file and docker-compose file to the repository.project root level and push a commit to the github, Now git hub triggeres the Jenkins pipeline and it will be build and deploy the project into the client virtual machine.
Let's check this behavior

See the commit status from the github repository -> actions
Sign to the Jenkins dashboard and see the latest trigger, its status and console output
If those are successfully completed let's inspect furthermore via the client virtual machine directly
sign into the client vm

check the respective container status
sudo docker ps

Sending request to the running container from the client vm itself

chamalka@free-vm-student-profile:~$ curl http://localhost:5008/api
{"status":200,"message":"Hello there !! Welcome to testing jenkins ci/cd pipeline API ! \n API is running successfully \n sec var: kingswood"}c

Now everything working fine within local to the client vm, Then need to configure nginx configuration for access from the outside. In advantage domain name and ssl certification can be configured also.