**Docker Challenge**

**Create a customized Docker image using a Dockerfile**.

**Push the image to Docker Hub.**

**Push the same image to Amazon ECR.**

**Step 1: Create a project folder**

mkdir my-custom-nginx

cd my-custom-nginx

**Step 2: Create your custom HTML file**

Create index.html

vi index.html

Paste this:

<h1>Hello from My Custom Docker Image</h1>

<h2>Deployed using Dockerfile</h2>

Save and exit (:wq)

**Step 3: Create Dockerfile**

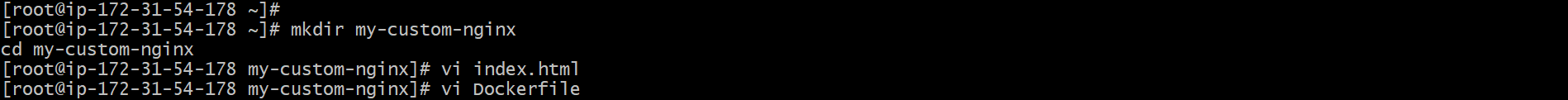
vi Dockerfile

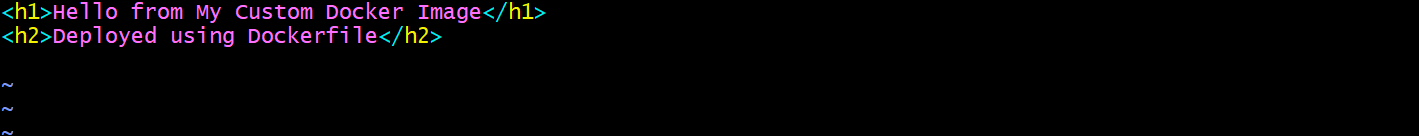
Paste this:

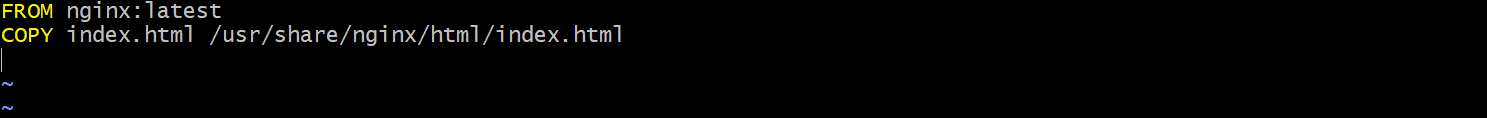
FROM nginx:latest

COPY index.html /usr/share/nginx/html/index.html

Save and exit.







**Step 4: Build Docker Image**

docker build -t mynginxapp:v1 .

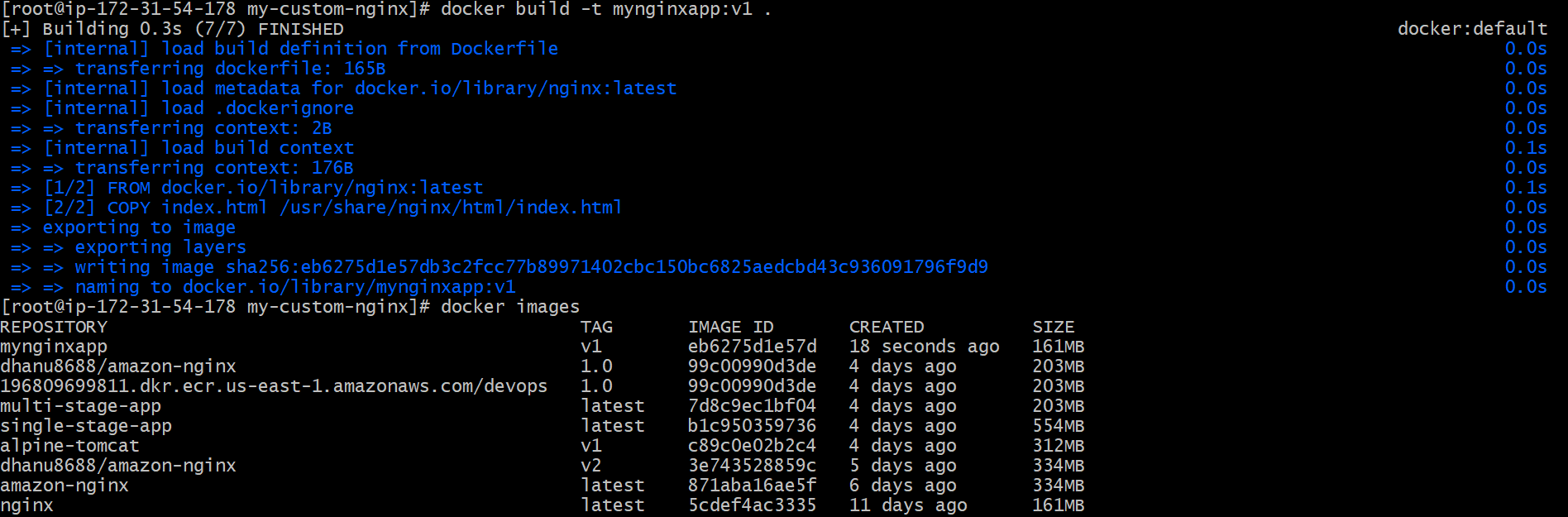
**Step 5: Verify image is created**

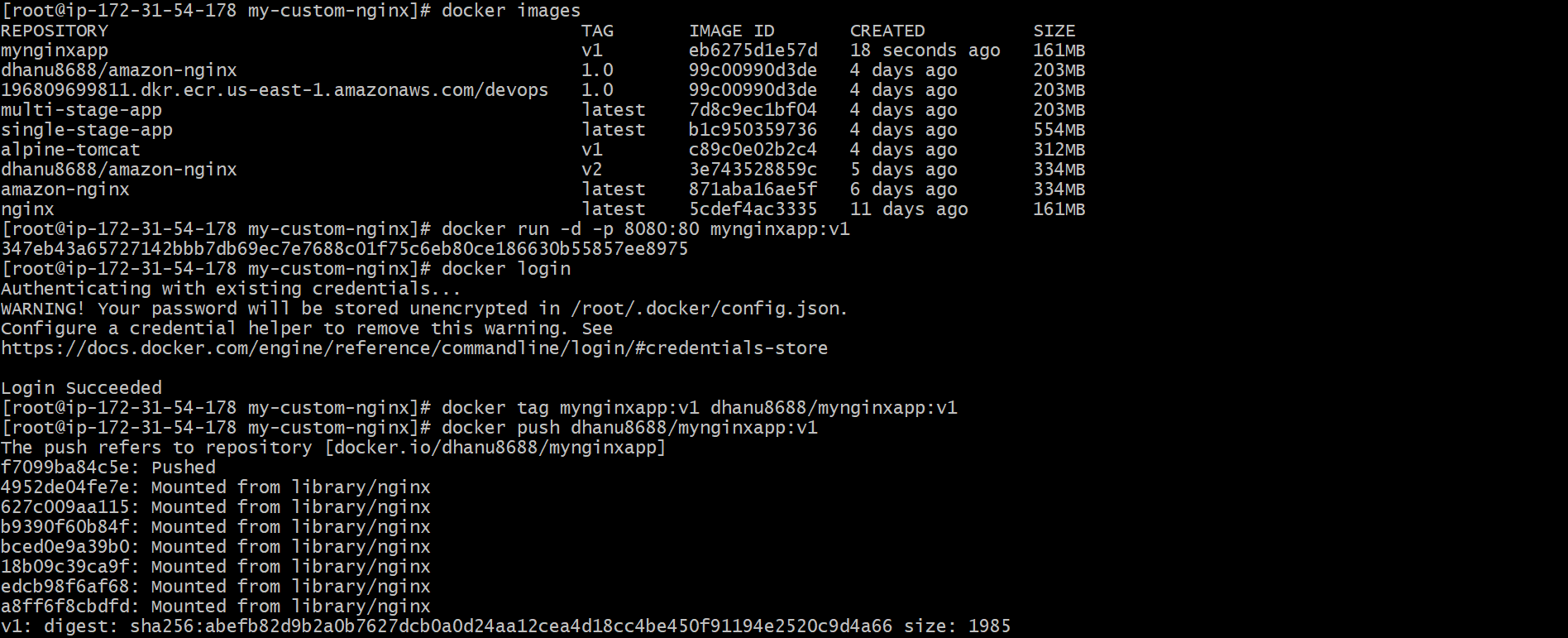
docker images

You should see:  
mynginxapp v1

**Step 6: Run the container and test**

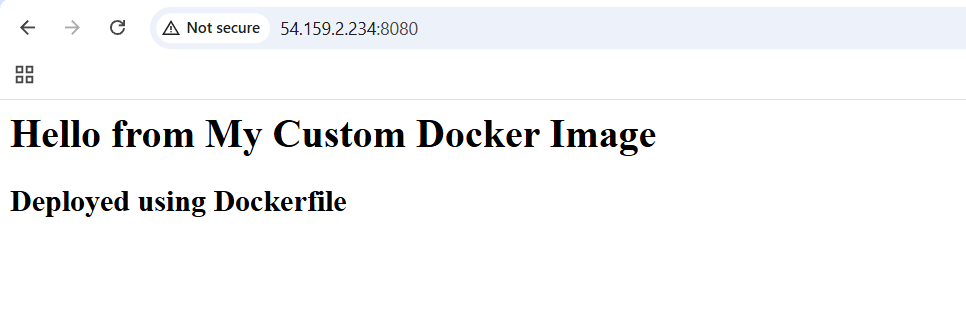
docker run -d -p 8080:80 mynginxapp:v1

****

****

Now open browser:

http://localhost:8080



**PART 2: Push Image to Docker Hub**

**Step 1: Login to Docker Hub**

docker login

Enter Docker Hub username & password.

**Step 2: Tag the image**

Docker Hub requires format:

dockerhubusername/repository:tag

Example:

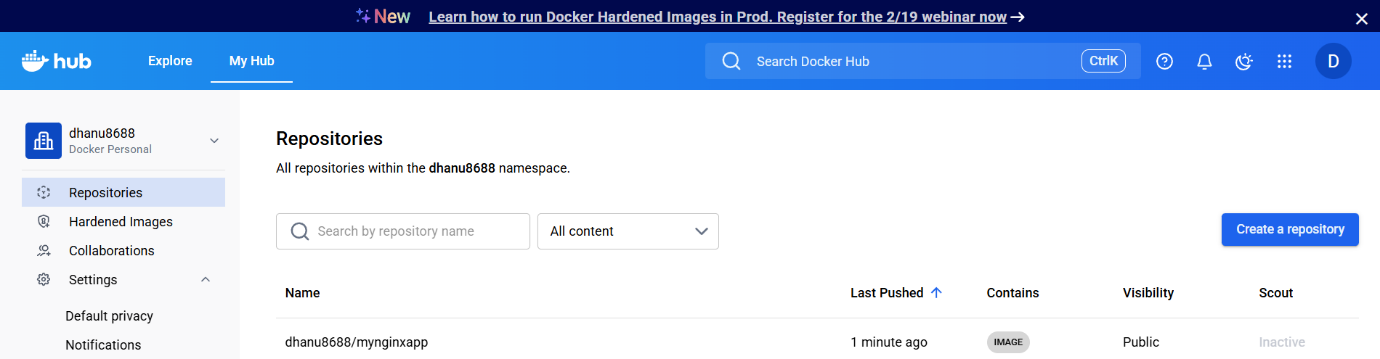
docker tag mynginxapp:v1 <dockerhub-username>/mynginxapp:v1

**Step 3: Push to Docker Hub**

docker push <dockerhub-username>/mynginxapp:v1

**Step 4: Verify in Docker Hub**

Go to Docker Hub → Repositories  
You will see your image uploaded.



**PART 3: Push Same Image to Amazon ECR**

**Step 1: Install AWS CLI**

Check:

aws –version

**Step 2: Configure AWS CLI**

aws configure

Provide:

* AWS Access Key
* AWS Secret Key
* Region (example: ap-south-1)
* Output format: json

**Step 3: Create an ECR Repository**

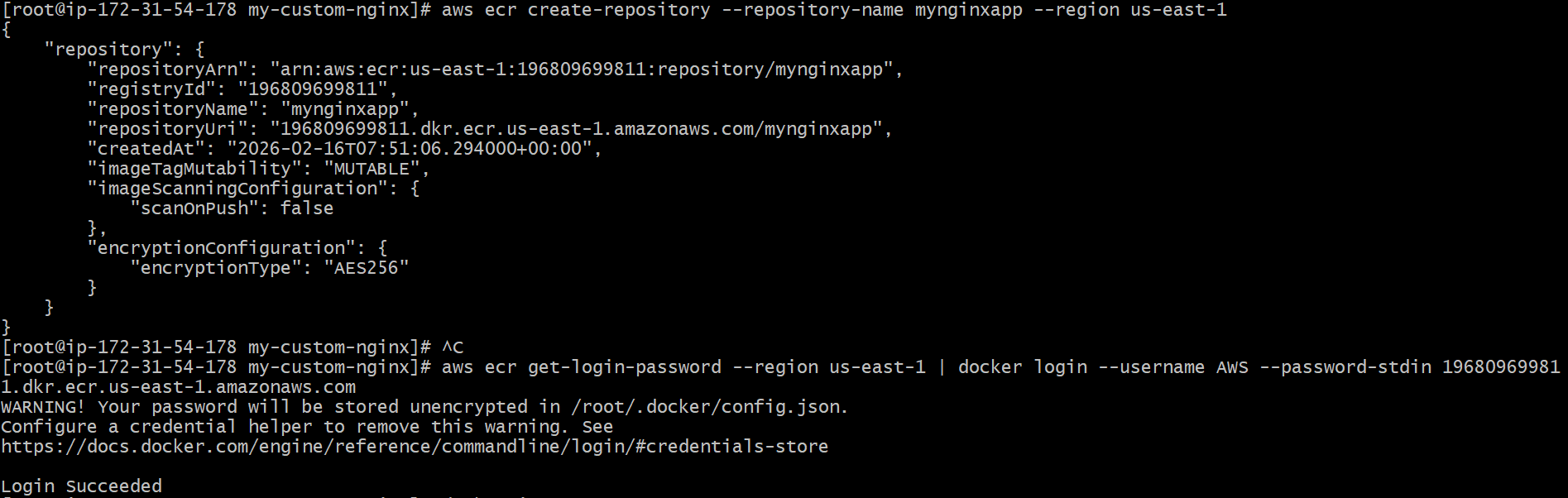
* Go to AWS Console → ECR → Create Repository
* Repository name:
* mynginxapp
* OR create using CLI:
* aws ecr create-repository --repository-name mynginxapp --region us-east-1

**Step 4: Get ECR login command**

Run:

aws ecr get-login-password --region ap-south-1 | docker login --username AWS --password-stdin <account-id>.dkr.ecr.ap-south-1.amazonaws.com

Replace <account-id> with your AWS account number.

****

**Step 5: Tag your Docker image for ECR**

ECR format:

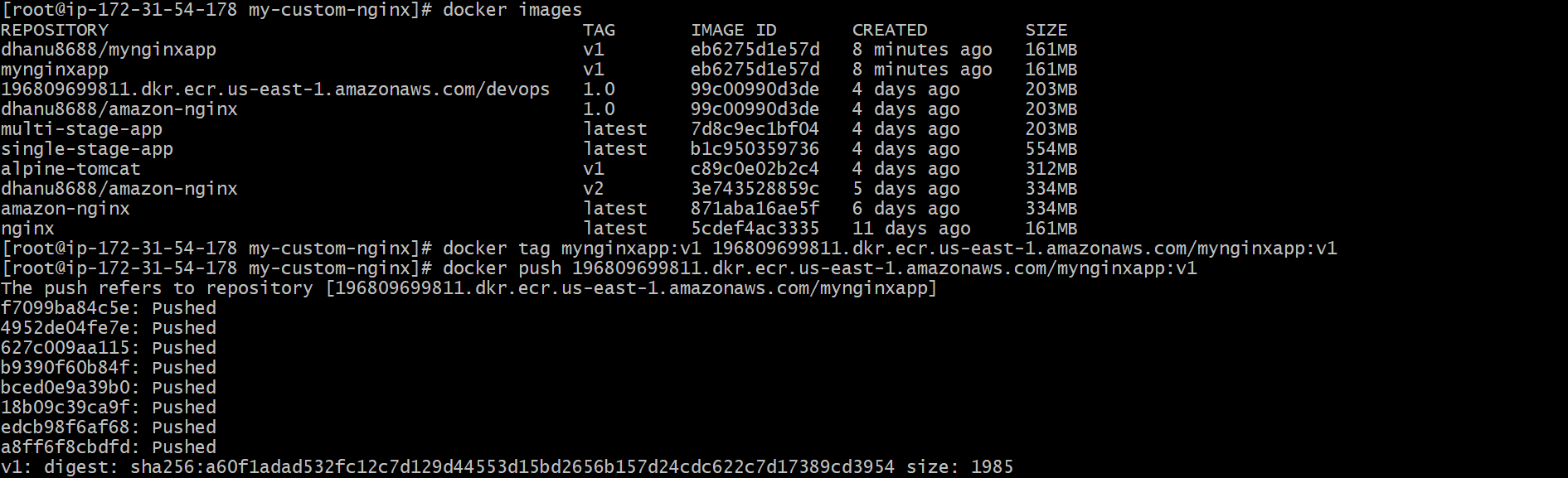
account-id.dkr.ecr.region.amazonaws.com/repo:tag

Command:

docker tag mynginxapp:v1 <account-id>.dkr.ecr.ap-east-1.amazonaws.com/mynginxapp:v1

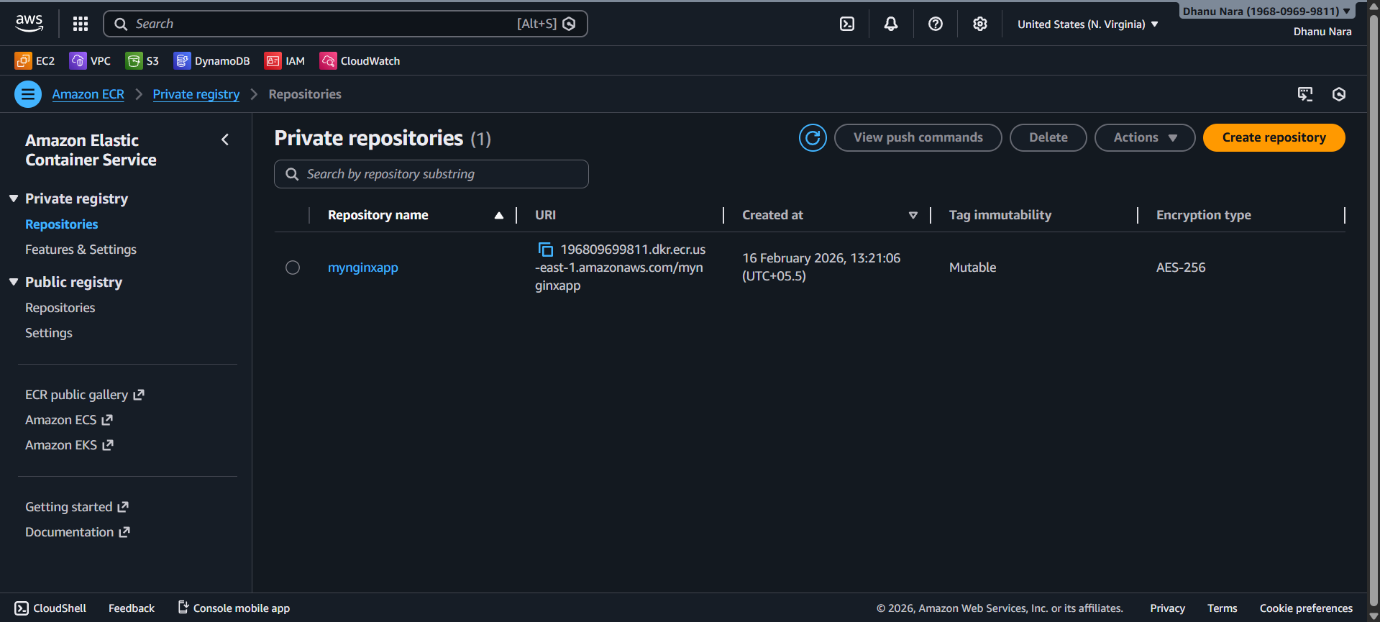
**Step 6: Push image to ECR**

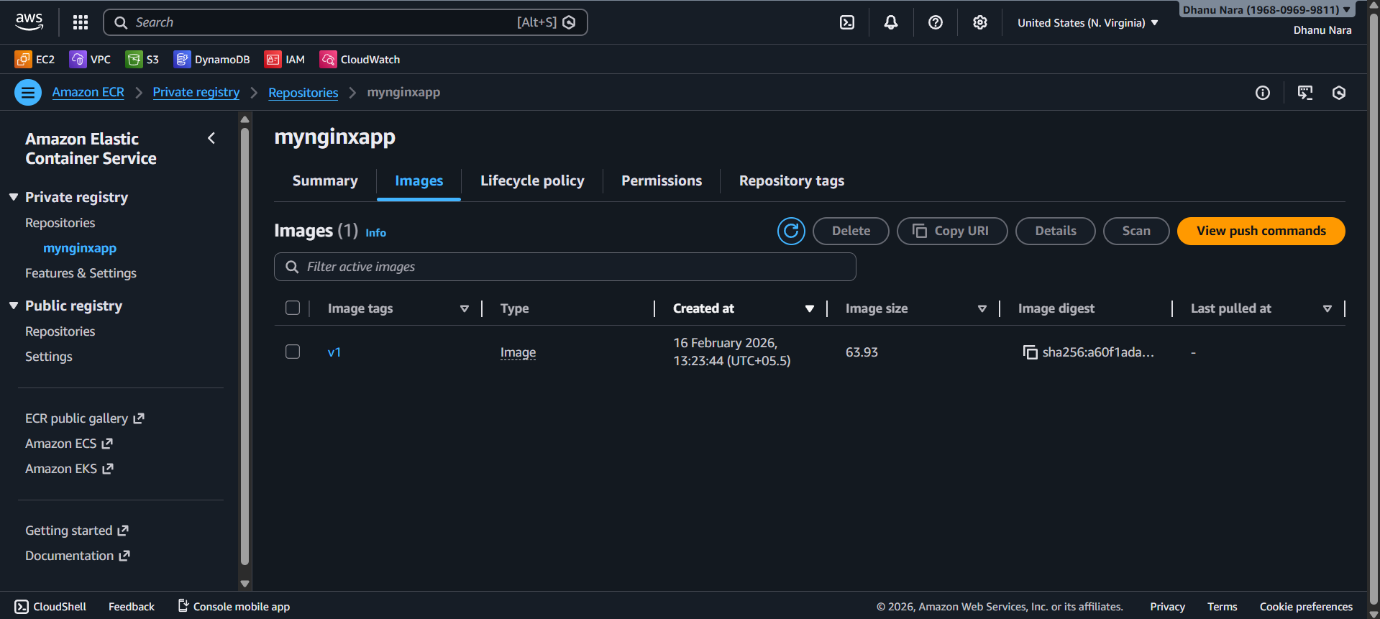
docker push <account-id>.dkr.ecr.us-east-1.amazonaws.com/mynginxapp:v1

****

**Step 7: Verify in AWS Console**

AWS Console → ECR → mynginxapp → Images





**Provision one EC2 instance using Terraform and install Jenkins.**

**Configure AWS Credentials**

Run:

aws configure

Enter:

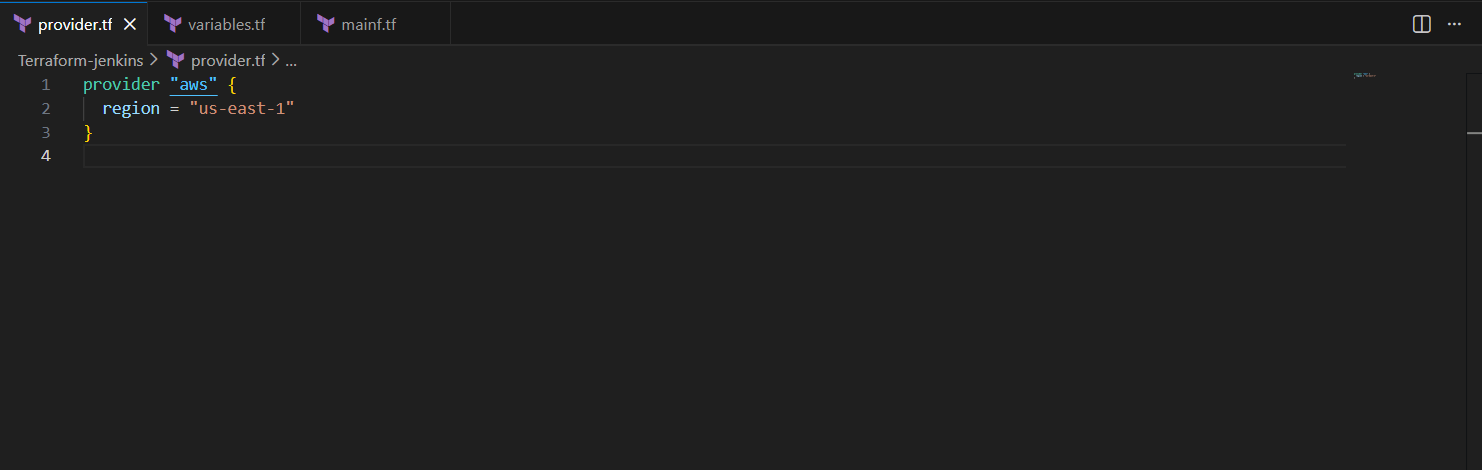
* Access Key
* Secret Key
* Region: us-east-1
* Output: json

**Create Terraform Project Folder**

* mkdir terraform-jenkins
* cd terraform-jenkins

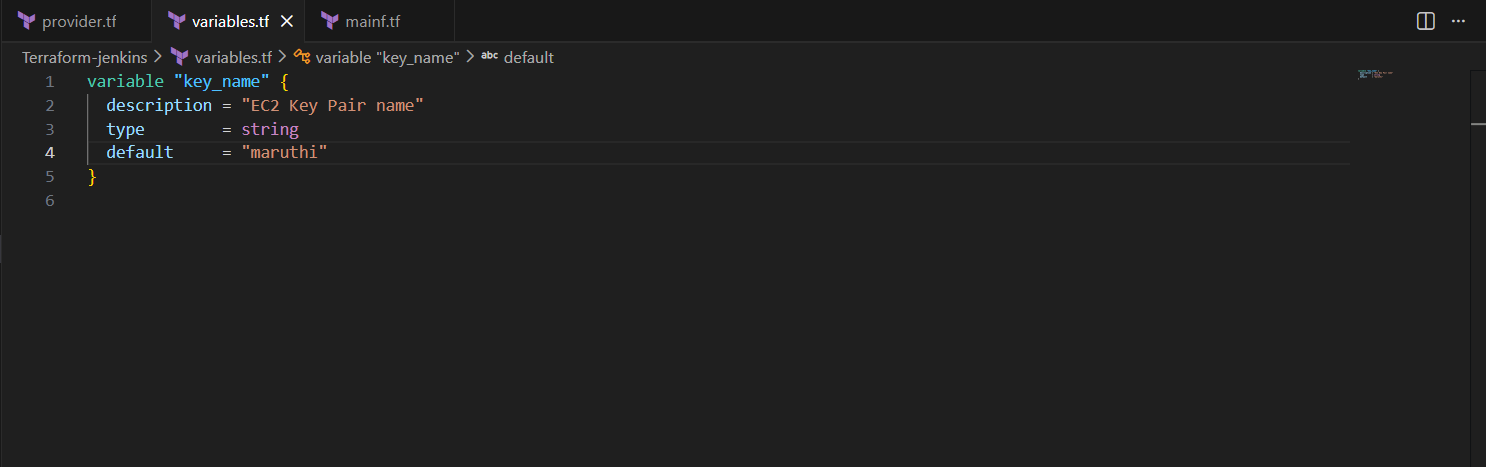
**Create provider.tf**

vi provider.tf

****

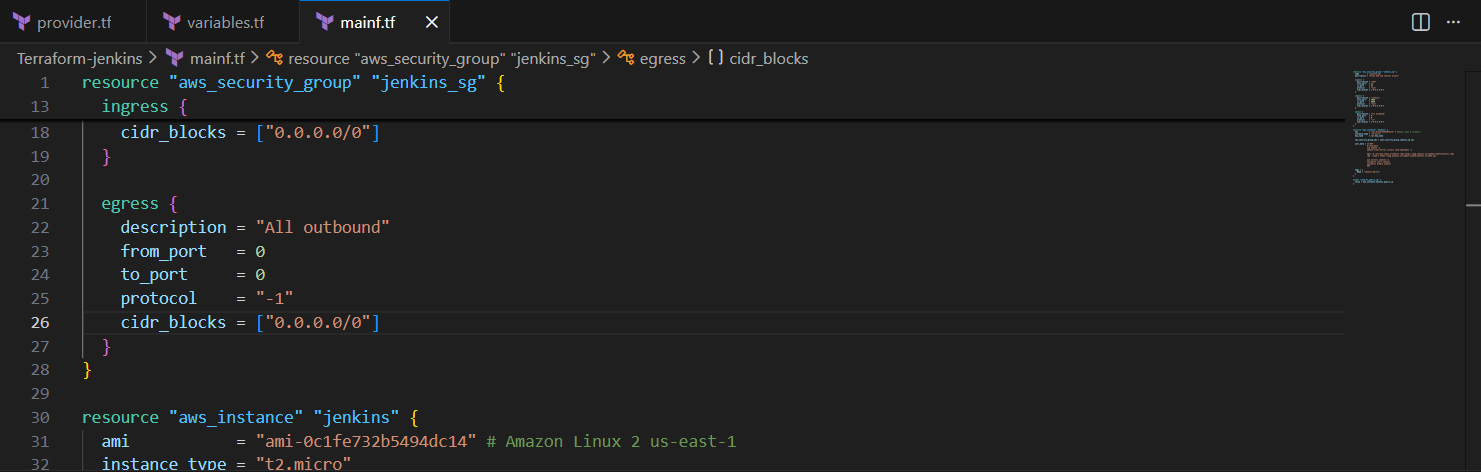
**Create variables.tf**

vi variables.tf

****

**Create main.tf**

vi main.tf

****

resource "aws\_security\_group" "jenkins\_sg" {

name = "jenkins-sg"

description = "Allow SSH and Jenkins access"

ingress {

description = "SSH"

from\_port = 22

to\_port = 22

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

ingress {

description = "Jenkins"

from\_port = 8080

to\_port = 8080

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

egress {

description = "All outbound"

from\_port = 0

to\_port = 0

protocol = "-1"

cidr\_blocks = ["0.0.0.0/0"]

}

}

resource "aws\_instance" "jenkins" {

ami = "ami-0c1fe732b5494dc14" # Amazon Linux 2 us-east-1

instance\_type = "t2.micro"

key\_name = var.key\_name

vpc\_security\_group\_ids = [aws\_security\_group.jenkins\_sg.id]

user\_data = <<-EOF

#!/bin/bash

yum update -y

amazon-linux-extras install java-openjdk11 -y

wget -O /etc/yum.repos.d/jenkins.repo https://pkg.jenkins.io/redhat-stable/jenkins.repo

rpm --import https://pkg.jenkins.io/redhat-stable/jenkins.io-2023.key

yum install jenkins -y

systemctl start jenkins

systemctl enable jenkins

EOF

tags = {

Name = "Jenkins-Server"

}

}

output "jenkins\_public\_ip" {

value = aws\_instance.jenkins.public\_ip

}

**Initialize Terraform**

terraform init

This downloads AWS provider plugin.

**Validate Terraform Code**

terraform validate

**Terraform Plan**

terraform plan

This shows what Terraform will create.

**Apply Terraform (Create EC2 + Install Jenkins)**

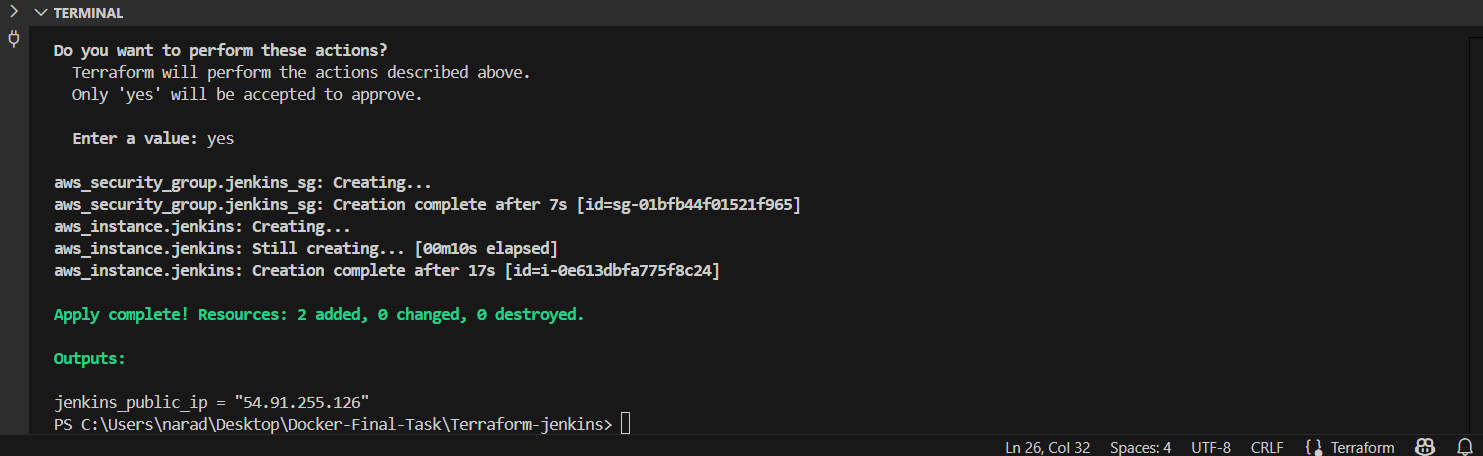
terraform apply

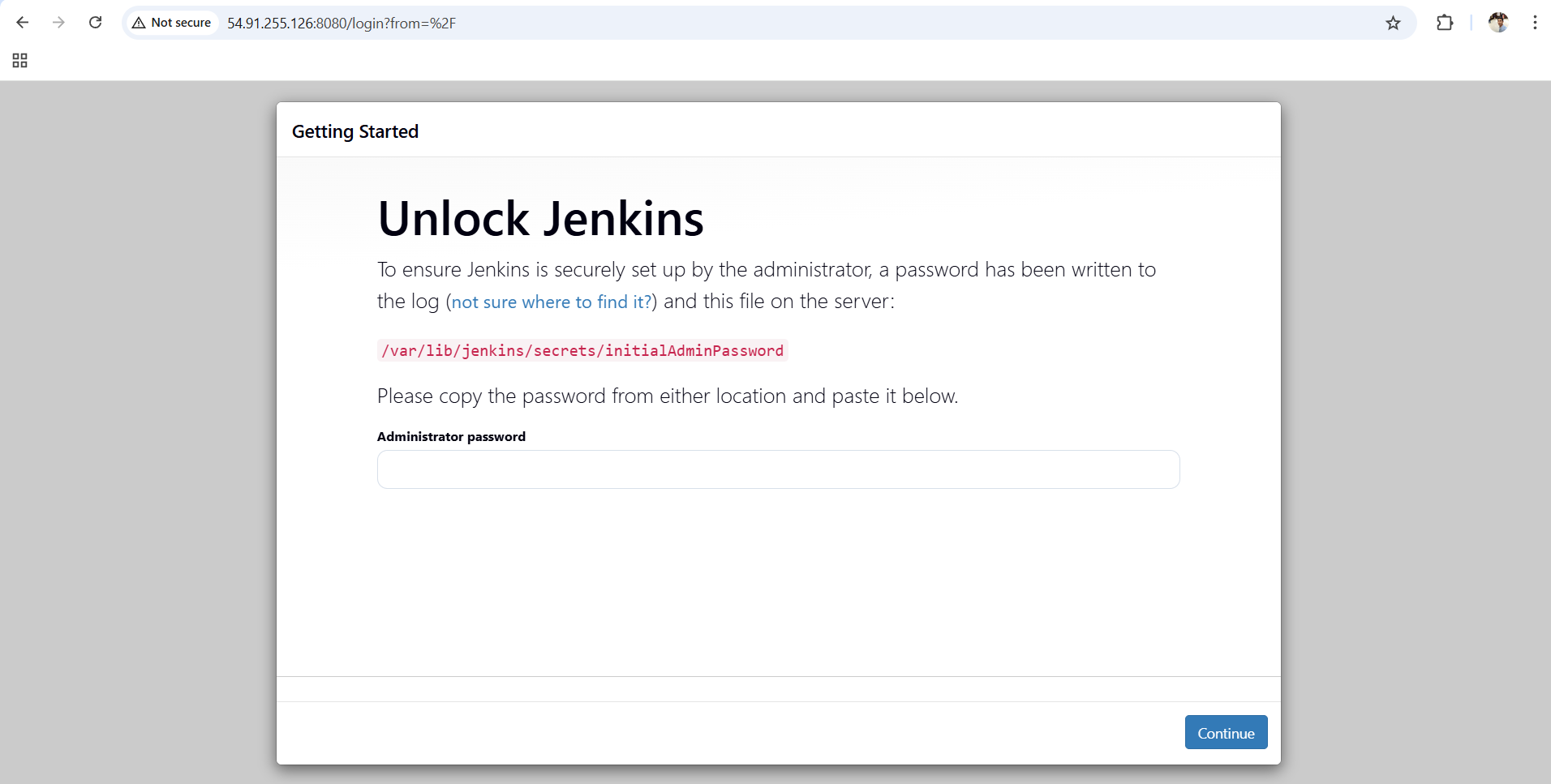
jenkins\_public\_ip = "54.xxx.xxx.xxx"

**Access Jenkins in Browser**

Open:

http://<PUBLIC-IP>:8080

****

****

**Create one Jenkins job to build and push the Docker image to Docker Hub.**

1. Source: <https://github.com/betawins/Python-app.git>

**Prerequisites**

**1. Jenkins should be running**

Open:

http://<EC2-Public-IP>:8080

**2. Install Docker in Jenkins server (EC2)**

Login to EC2 and run:

sudo yum update -y

sudo yum install docker -y

sudo systemctl start docker

sudo systemctl enable docker

Now add Jenkins user to docker group:

sudo usermod -aG docker jenkins

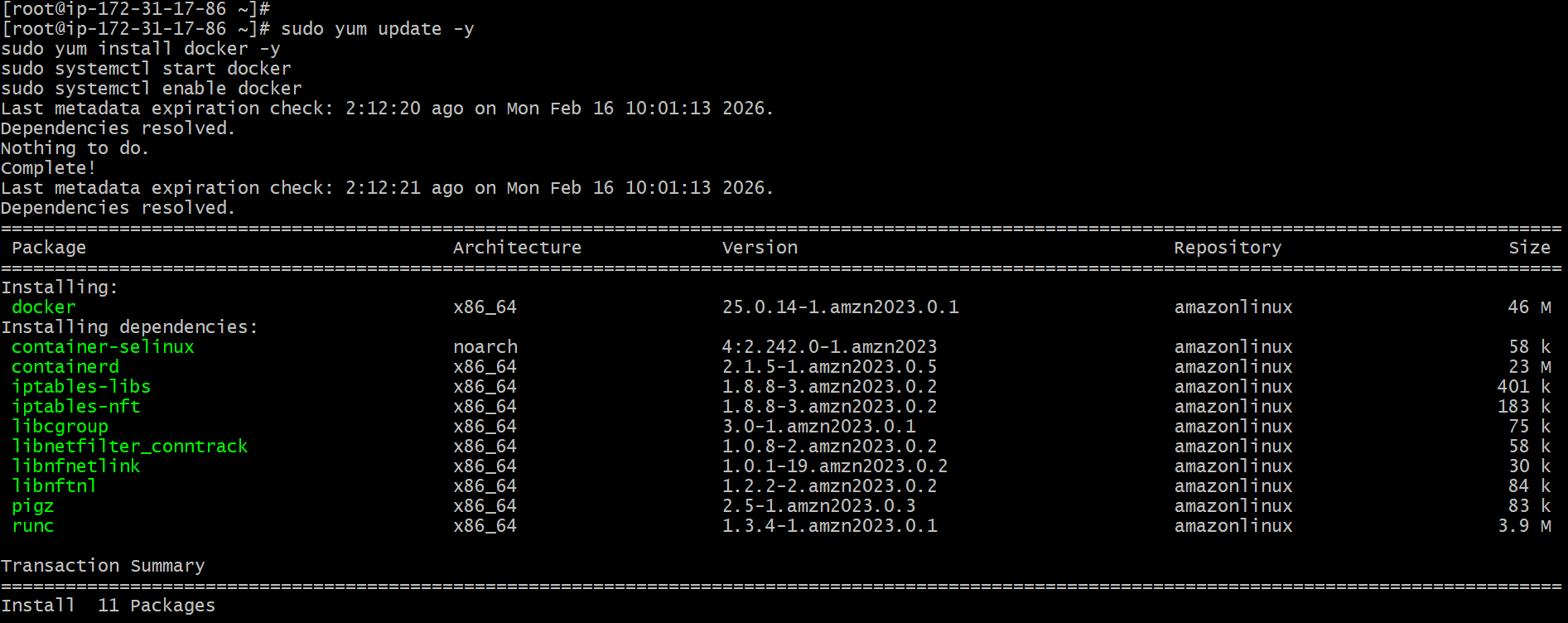
sudo systemctl restart jenkins

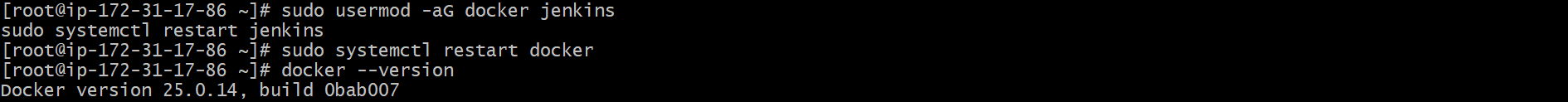
Restart docker too:

sudo systemctl restart docker

Now check docker works:

docker --version





**Install Jenkins Plugins**

Go to Jenkins Dashboard:

**Manage Jenkins → Plugins → Available Plugins**

Install these:

✅ Git plugin  
✅ Docker Pipeline  
✅ Docker plugin  
✅ Credentials Binding Plugin

After installing, restart Jenkins.

**Create DockerHub Credentials in Jenkins**

Go to:

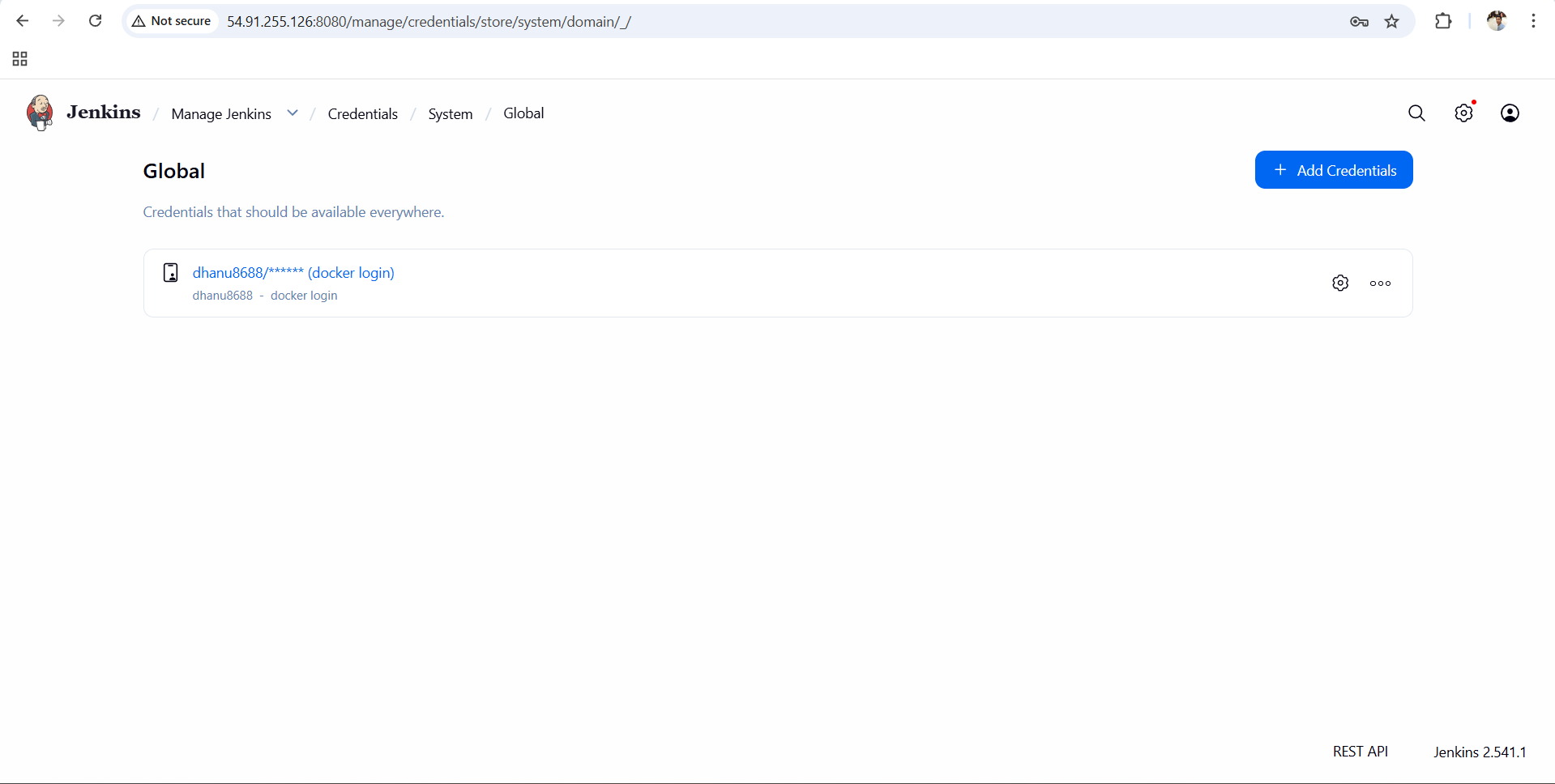
**Manage Jenkins → Credentials → System → Global credentials → Add Credentials**

Fill:

* Kind: **Username with password**
* Username: **your DockerHub username**
* Password: **your DockerHub password**
* ID: dockerhub-creds *(important)*
* Description: DockerHub login

Click **Save**.





**Create Jenkins Job (Pipeline)**

Go to Jenkins Dashboard → **New Item**

1. Enter Job name:
2. docker-python-app
3. Select:  
   ✅ Pipeline
4. Click **OK**

**Configure Jenkins Job**

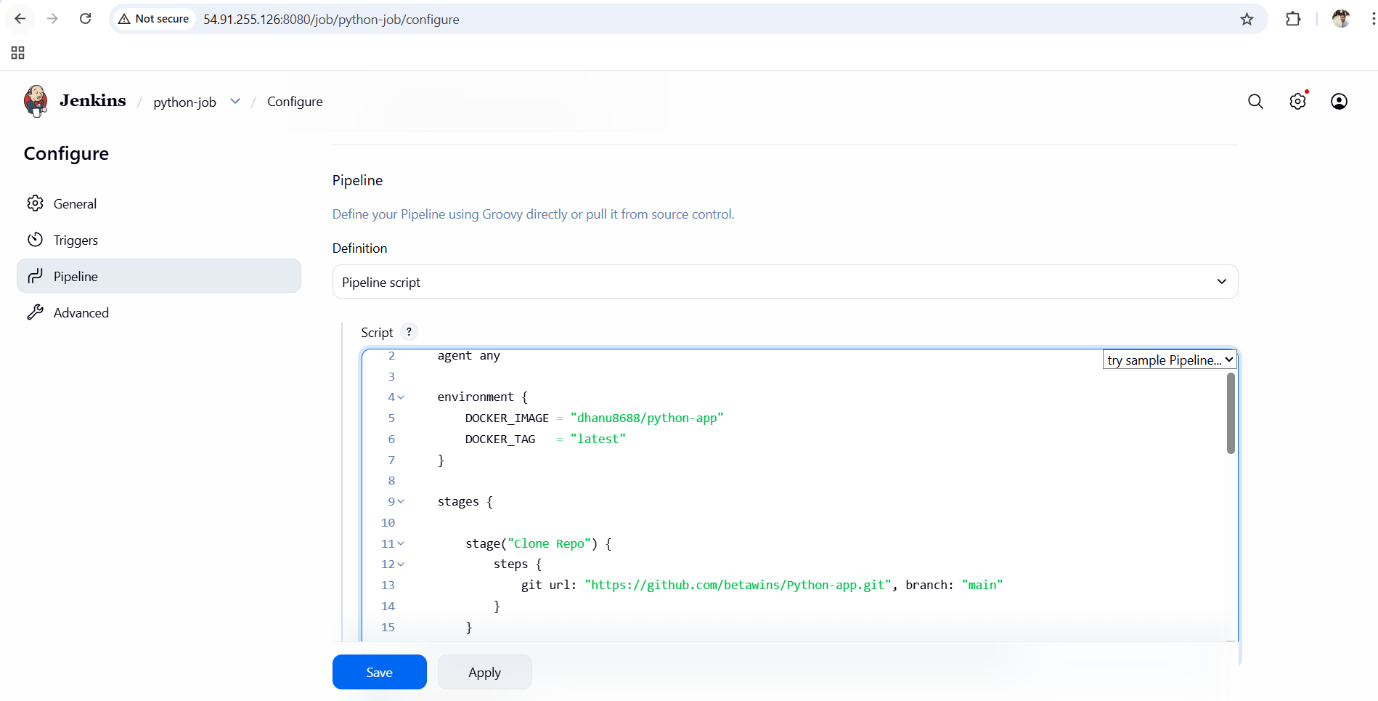
Scroll down to **Pipeline Section**

Select:

* Definition: **Pipeline script**

Now paste this Pipeline script:





pipeline {

agent any

environment {

IMAGE\_NAME = "dhanu8688/python-app"

TAG = "v1"

}

stages {

stage('Checkout Code') {

steps {

git branch: 'main', url: 'https://github.com/betawins/Python-app.git'

}

}

stage('Build Image') {

steps {

sh "docker build -t ${IMAGE\_NAME}:${TAG} ."

}

}

stage('Login to DockerHub') {

steps {

withCredentials([usernamePassword(

credentialsId: 'dhanu8688',

usernameVariable: 'USER',

passwordVariable: 'PASS'

)]) {

sh "echo $PASS | docker login -u $USER --password-stdin"

}

}

}

stage('Push Image') {

steps {

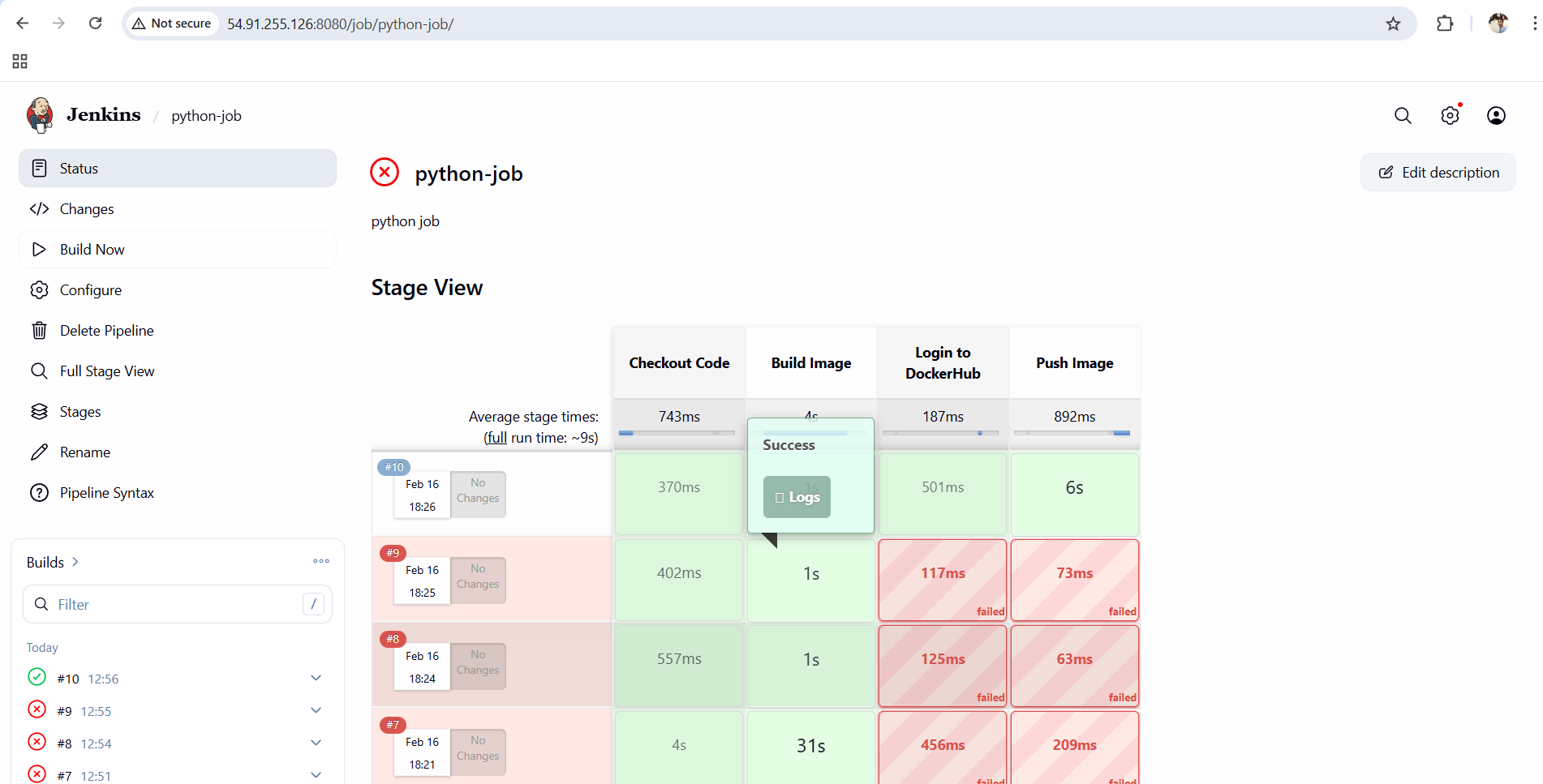
sh "docker push ${IMAGE\_NAME}:${TAG}"

}

}

}

}

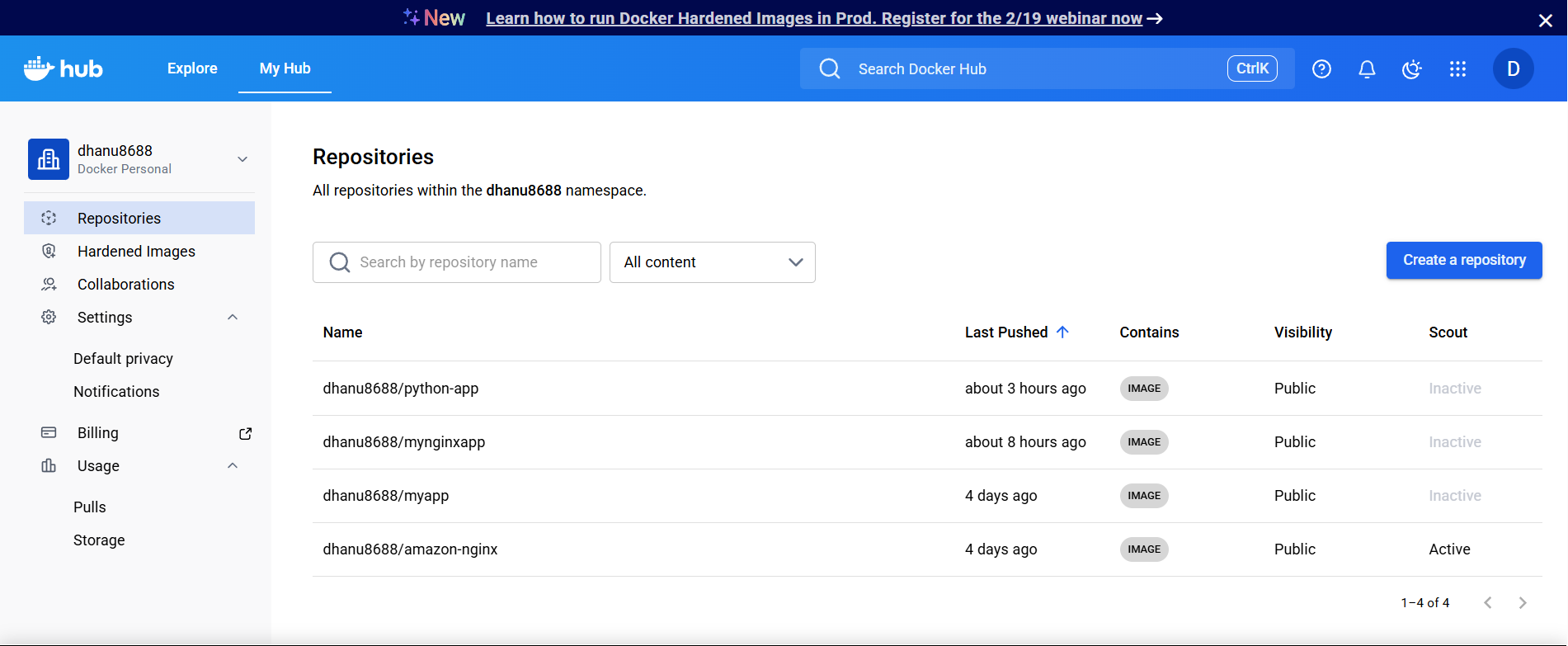


**Verify in DockerHub**

Go to DockerHub → Repositories

You should see:

python-app



:open_file_folder:**Source Codes:** <https://github.com/betawins/docker-tasks.git>  
**From the frontend source code, write a Dockerfile, build a Docker image, run it, and push that image to your Docker registry.**

**Install Docker + Git (if not installed)**

**For Amazon Linux / RHEL:**

yum update -y

yum install docker git -y

systemctl start docker

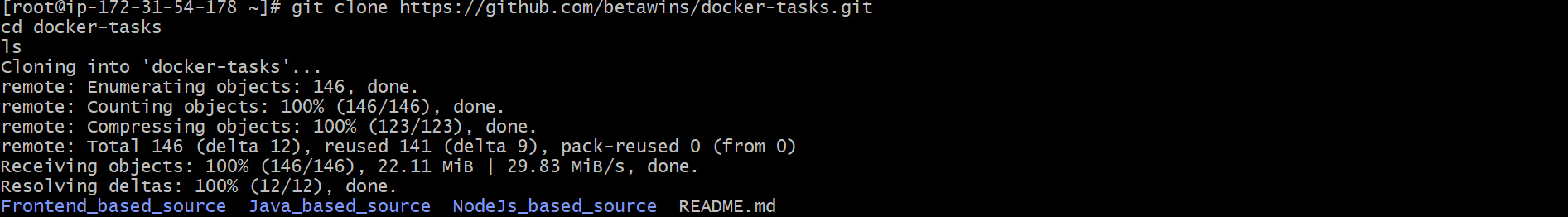
systemctl enable docker

Check Docker:

docker --version

**Clone the repository**

git clone https://github.com/betawins/docker-tasks.git

****

Now go inside:

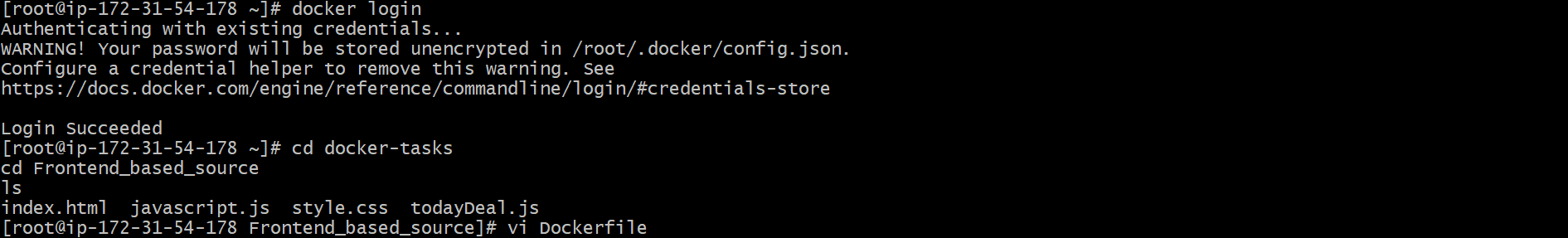
cd docker-tasks

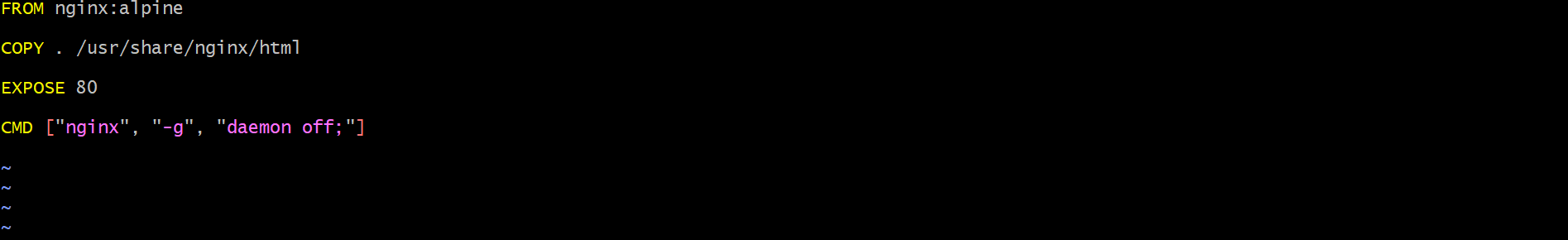
ls

**Go inside frontend source folder**

cd Frontend\_based\_source

ls





**Build Docker Image**

Now build the image:

docker build -t dhanu8688/frontend-app:v1 .

**Run Docker Container**

Run container:

docker run -d --name frontend-container -p 8085:80 dhanu8688/frontend-app:v1

Verify running:

docker ps



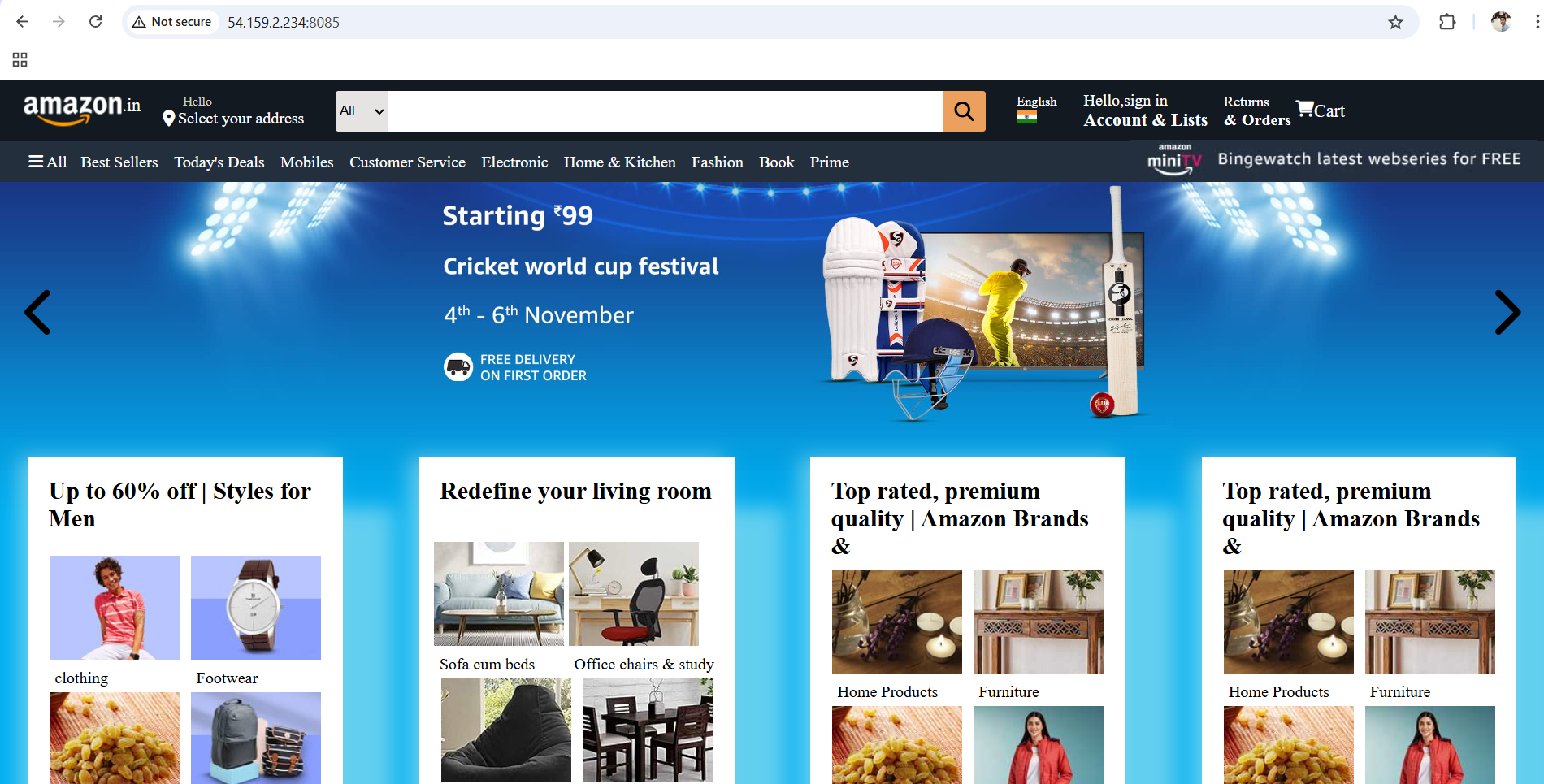


**Access application in Browser**

Open in browser:

**If local machine:**

http://localhost:8085

****

**From the Java-based source code, write a Dockerfile, build, run, and push the image to the Docker registry.**

**Go inside Java source folder**

cd Java\_based\_source

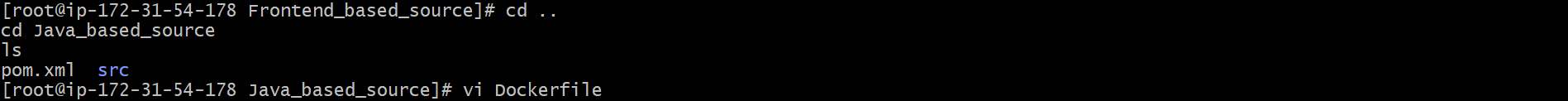
ls

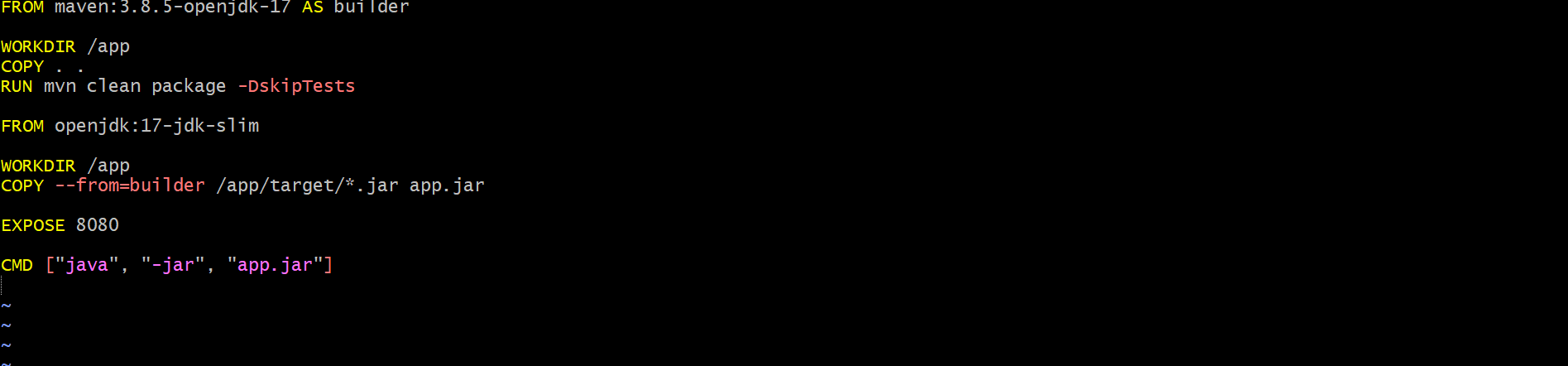
You should see:

* pom.xml
* src/

**Create Dockerfile**

vi Dockerfile

****

****

**Build Docker image**

docker build -t dhanu8688/java-app:v1 .

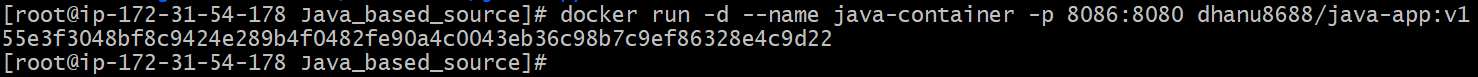
****

**Run the container**

docker run -d --name java-container -p 8086:8080 dhanu8688/java-app:v1

Check container running:

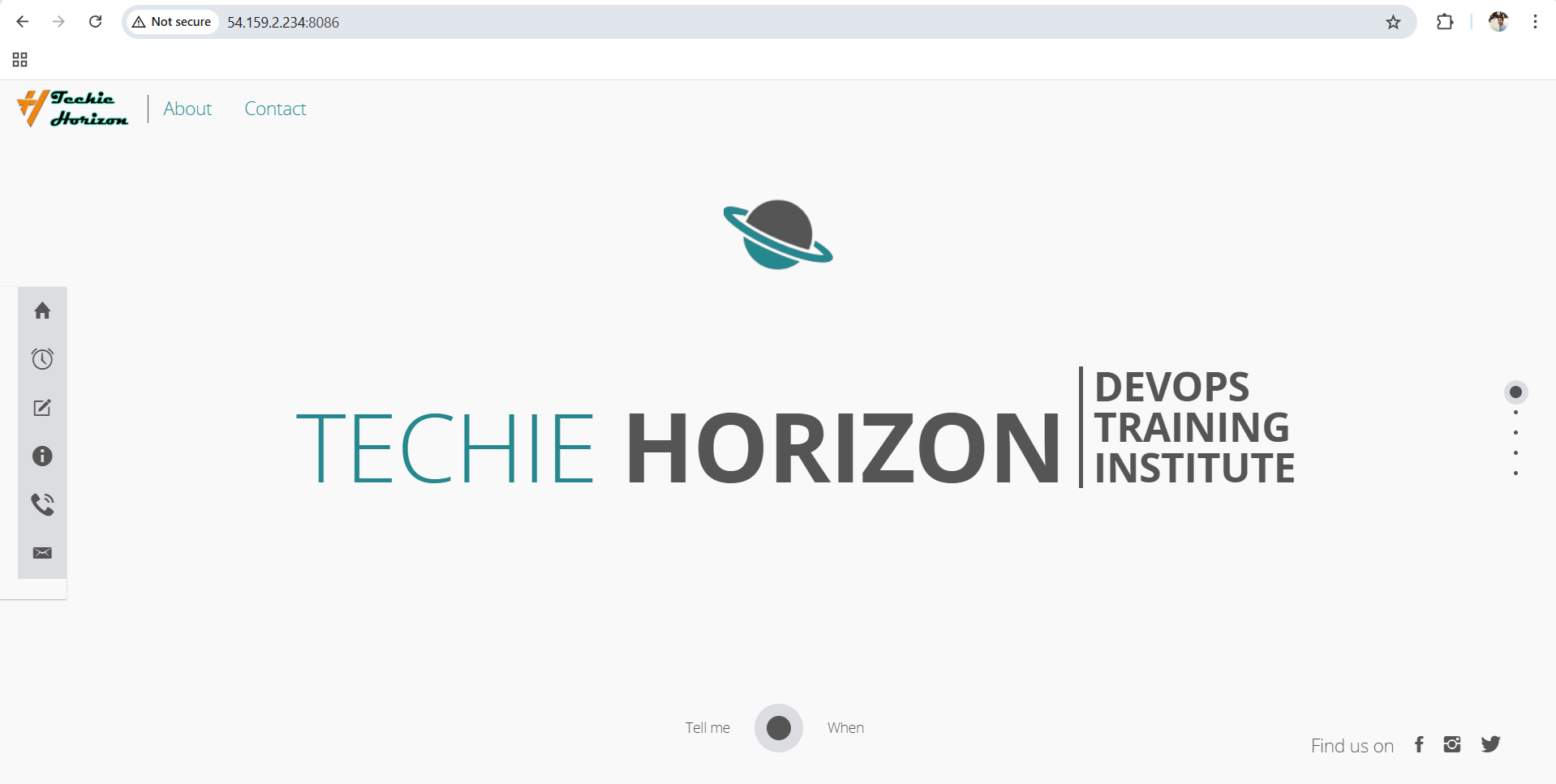
docker ps

****

**Open application in browser**

If your EC2 Public IP is 54.159.2.234, then open:

<http://54.159.2.234:8086>

****

**From the Node.js-based source code, write a Dockerfile, build with tag v1, run, and push it to the Docker registry.**

**Go to Node.js source folder**

cd NodeJs\_based\_source

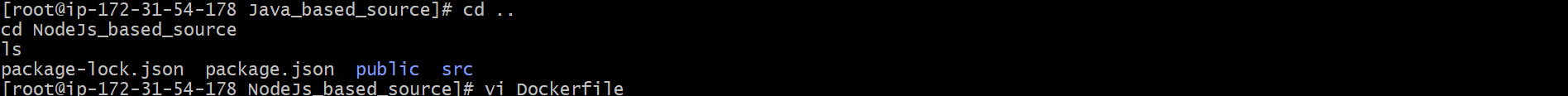
ls

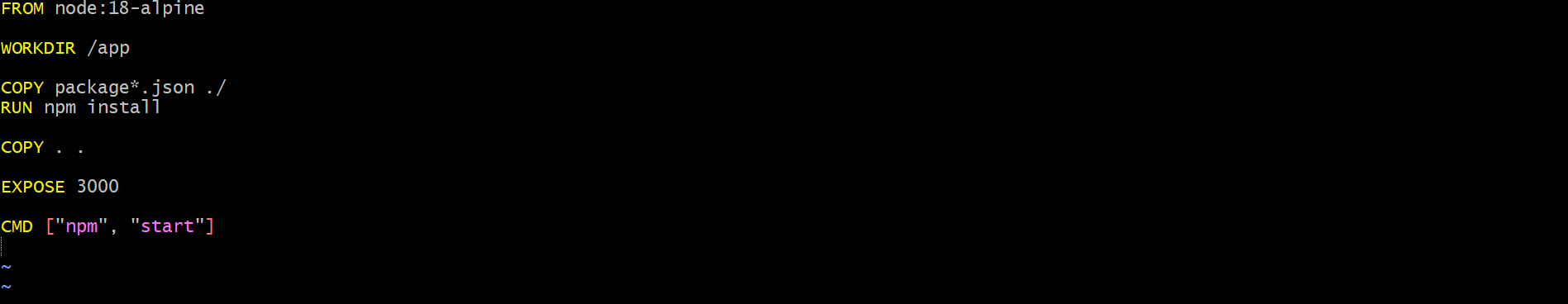
You should see:

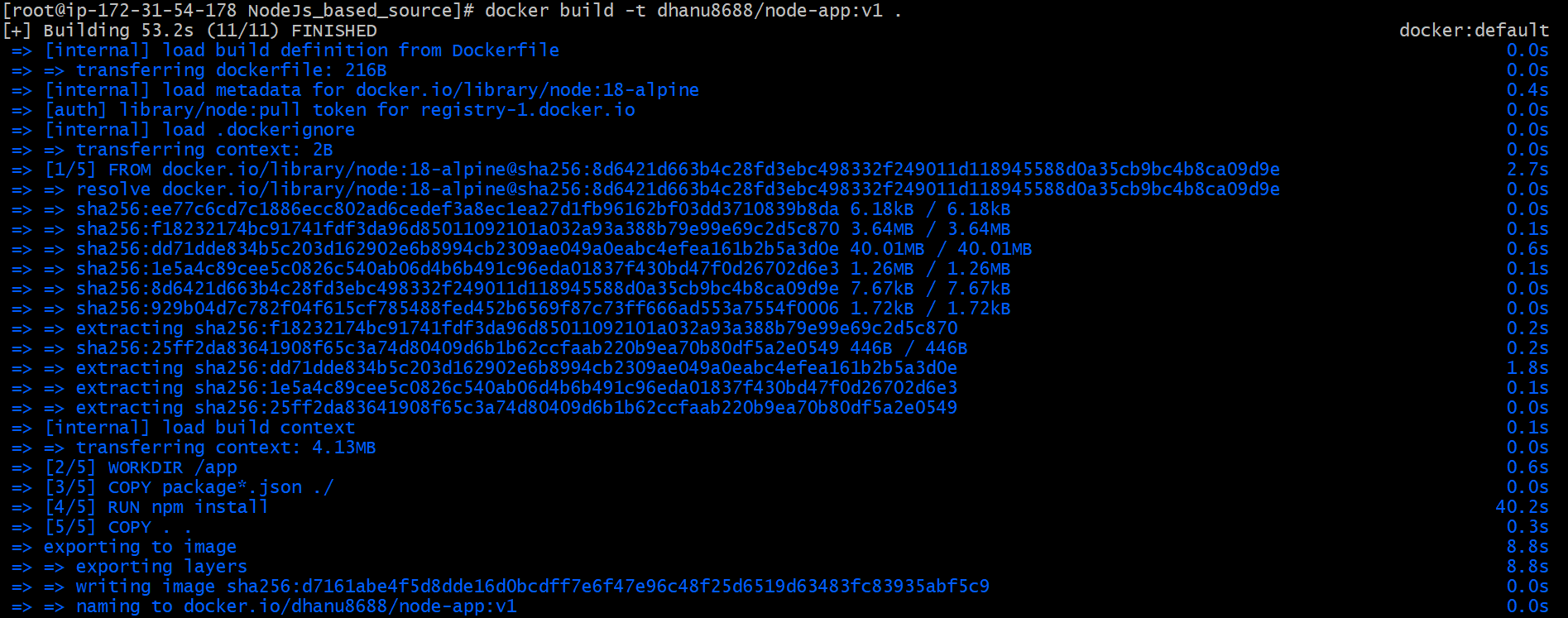
* package.json
* package-lock.json
* public/
* src/

**Create Dockerfile**

* vi Dockerfile

****

****

****

**Build Docker image with tag v1**

docker build -t dhanu8688/node-app:v1 .

Verify image created:

docker images

****

**Run container**

docker run -d --name node-container -p 3000:3000 dhanu8688/node-app:v1

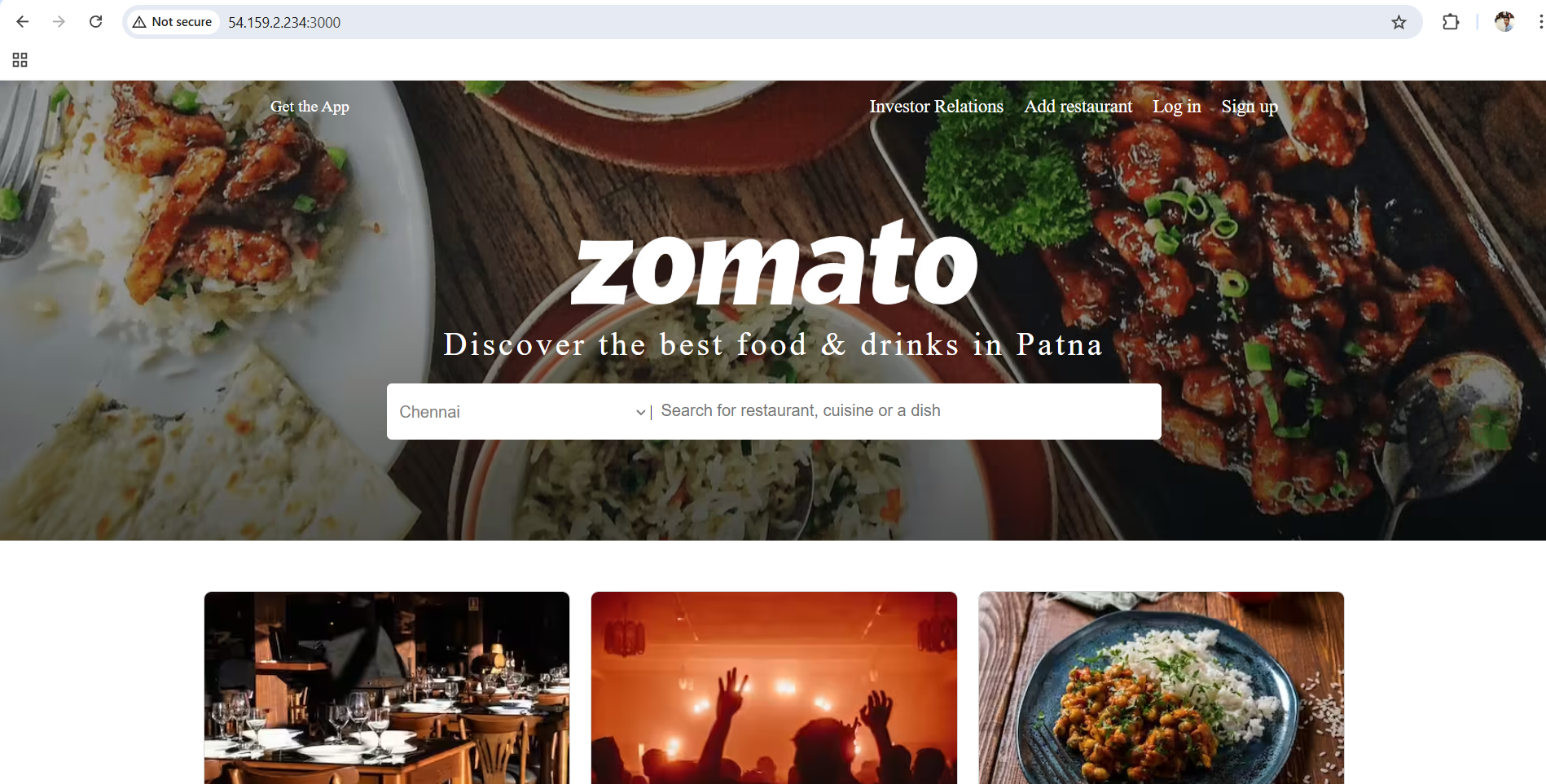
Verify running:

docker ps

**Access application in browser**

If your EC2 Public IP is 54.159.2.234:

http://54.159.2.234:3000

****

**Write a docker-compose file to set up WordPress with a MySQL database.**

**WordPress is an open-source Content Management System (CMS)** used to create and manage websites easily without needing heavy coding.

**Key points:**

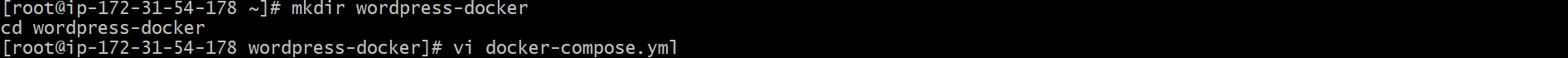
* It is mainly used to build **blogs, business websites, portfolios, and e-commerce sites**.
* It is written in **PHP**.
* It stores all website data (posts, pages, users, settings) inside a **database (MySQL/MariaDB)**.
* It supports **themes** (for design) and **plugins** (for extra features).
* It provides an **admin dashboard** to manage the website.

1. WordPress needs a **MySQL database**, so docker-compose helps run both together.
2. It allows managing **multiple containers as one application**.
3. You can start everything using **one command** (docker-compose up -d).
4. It automatically creates a **network** so WordPress can connect to MySQL easily.
5. It avoids manual container linking and IP address confusion by using **service names**.
6. It supports **persistent storage (volumes)** so database and WordPress data won’t be lost.
7. It makes deployment **repeatable and consistent** across any server.
8. It helps in **easy scaling and modification** (add phpMyAdmin, nginx, etc.).
9. It simplifies stopping/removing everything using one command (docker-compose down).
10. It keeps the full configuration (ports, passwords, images) in **one single file**, making management easy.

**Create a project directory**

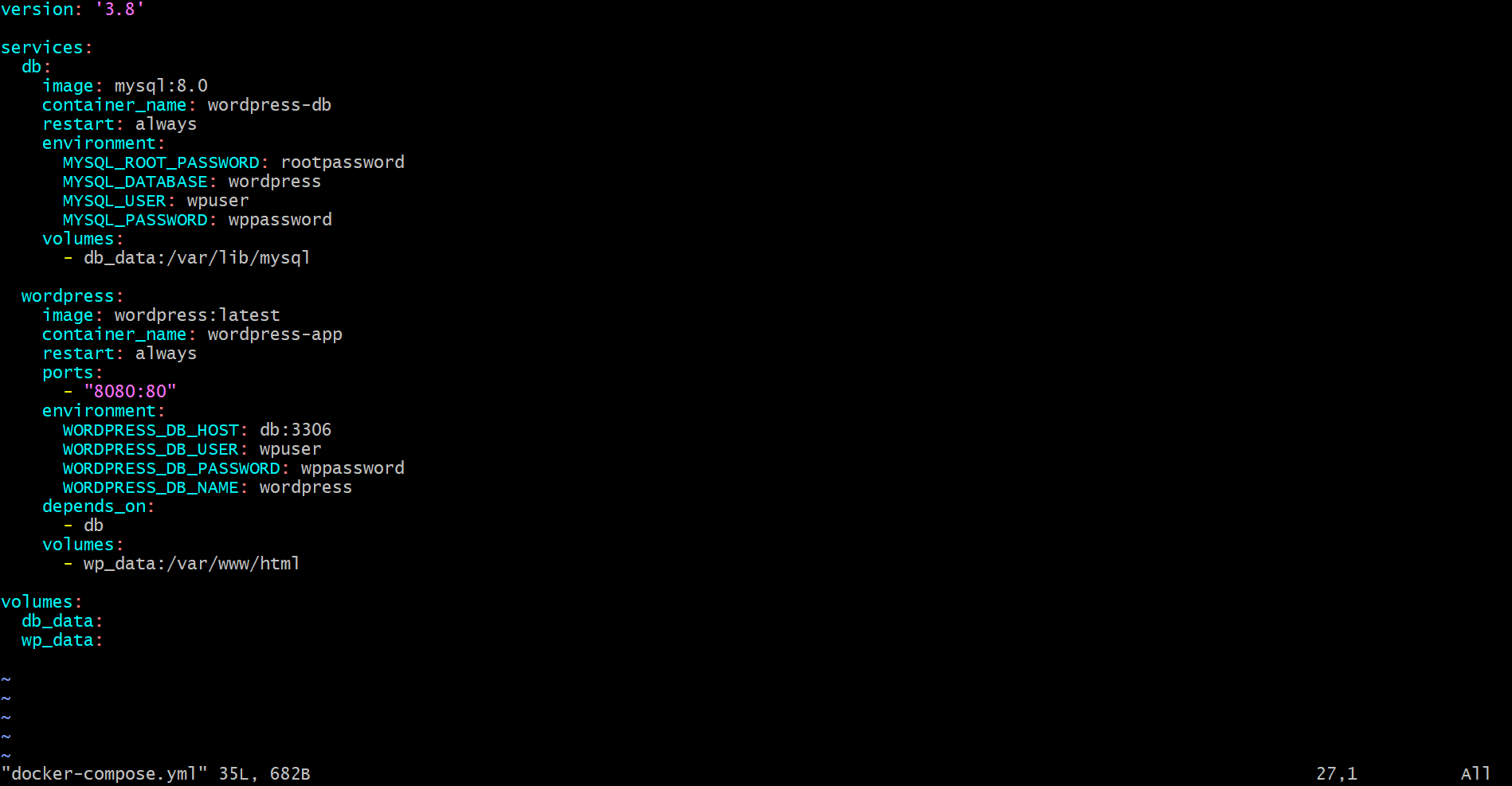
mkdir wordpress-docker

cd wordpress-docker

****

**Create docker-compose.yml file**

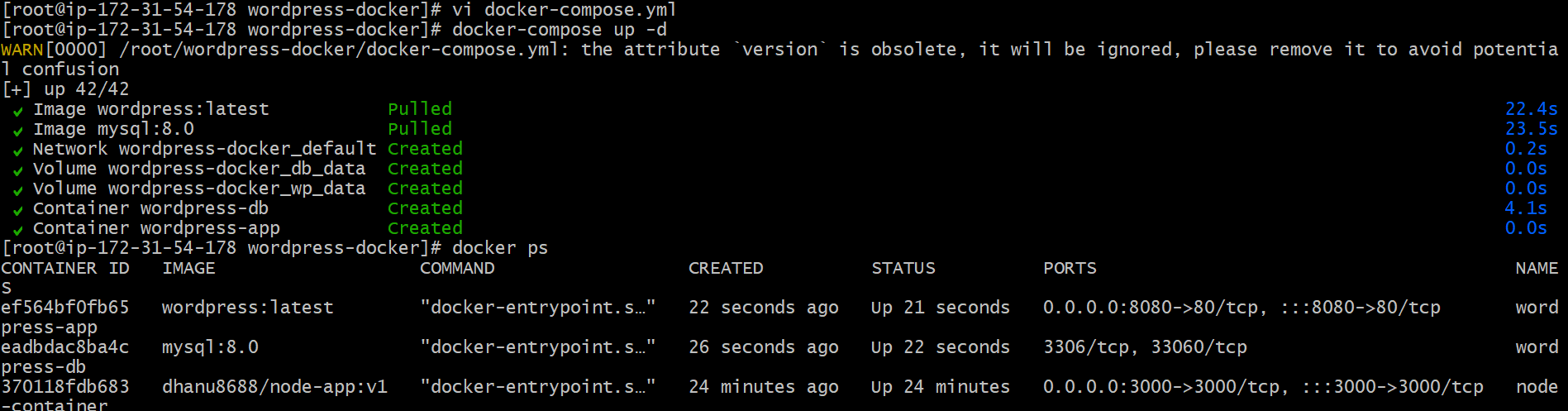
vi docker-compose.yml

****

**Start WordPress + MySQL containers**

Run:

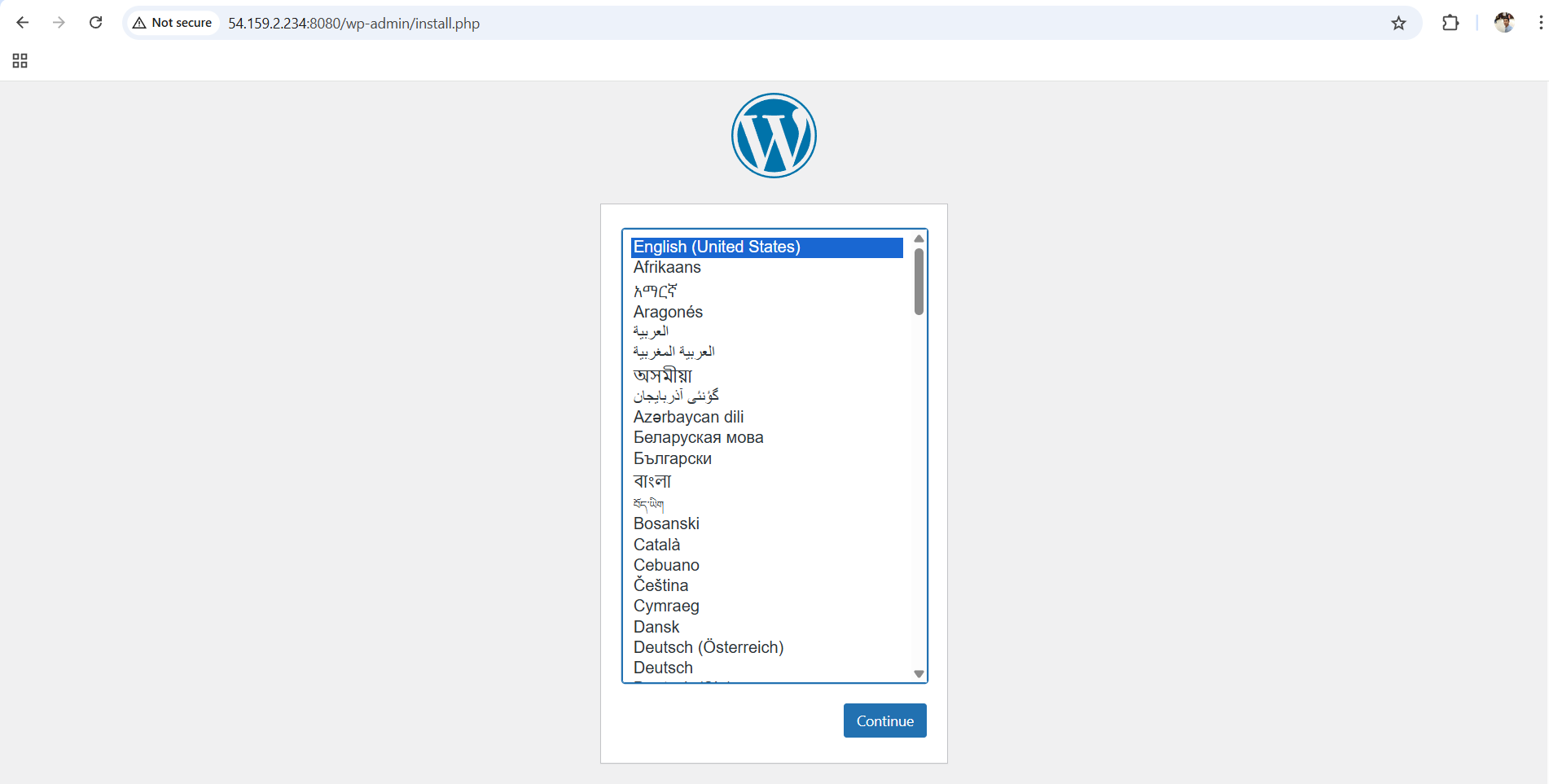
docker-compose up -d

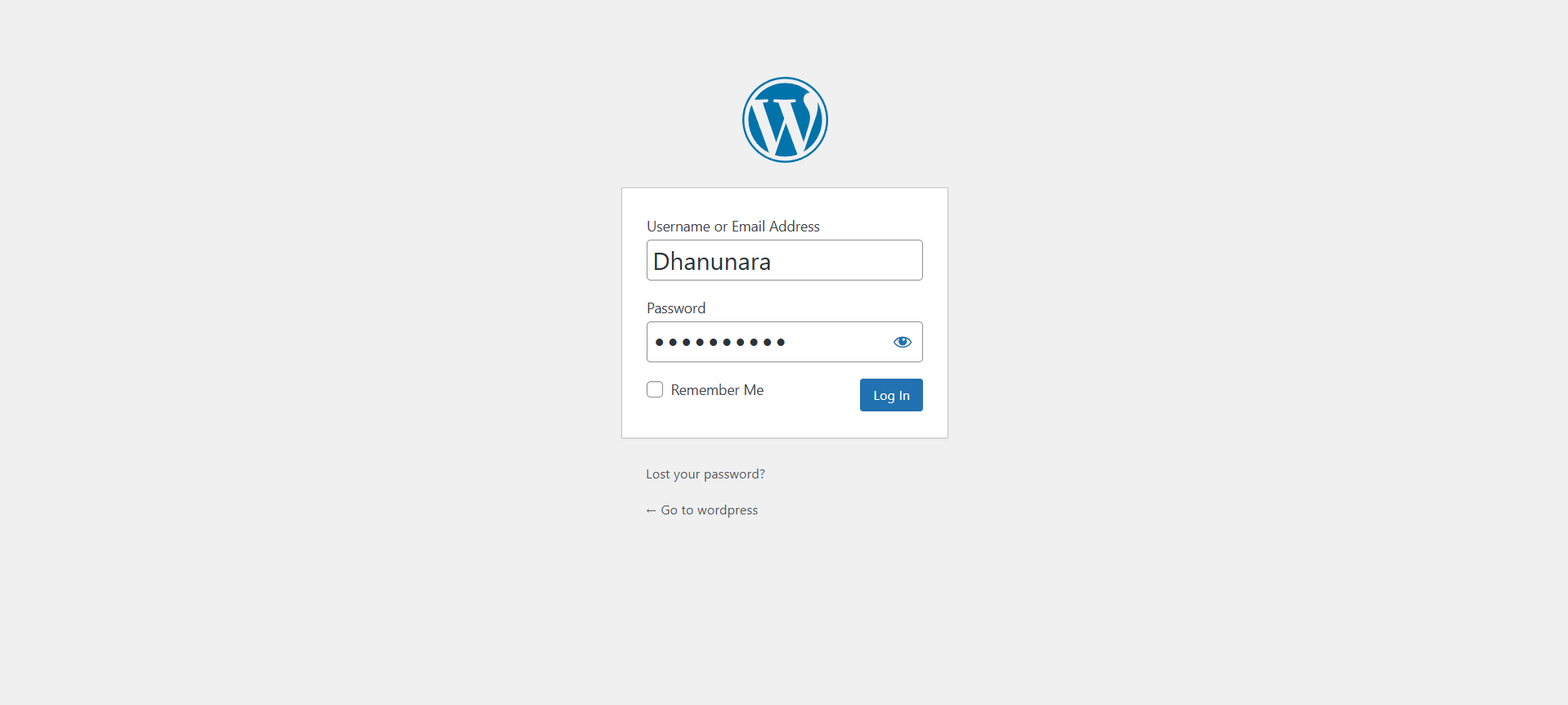
****

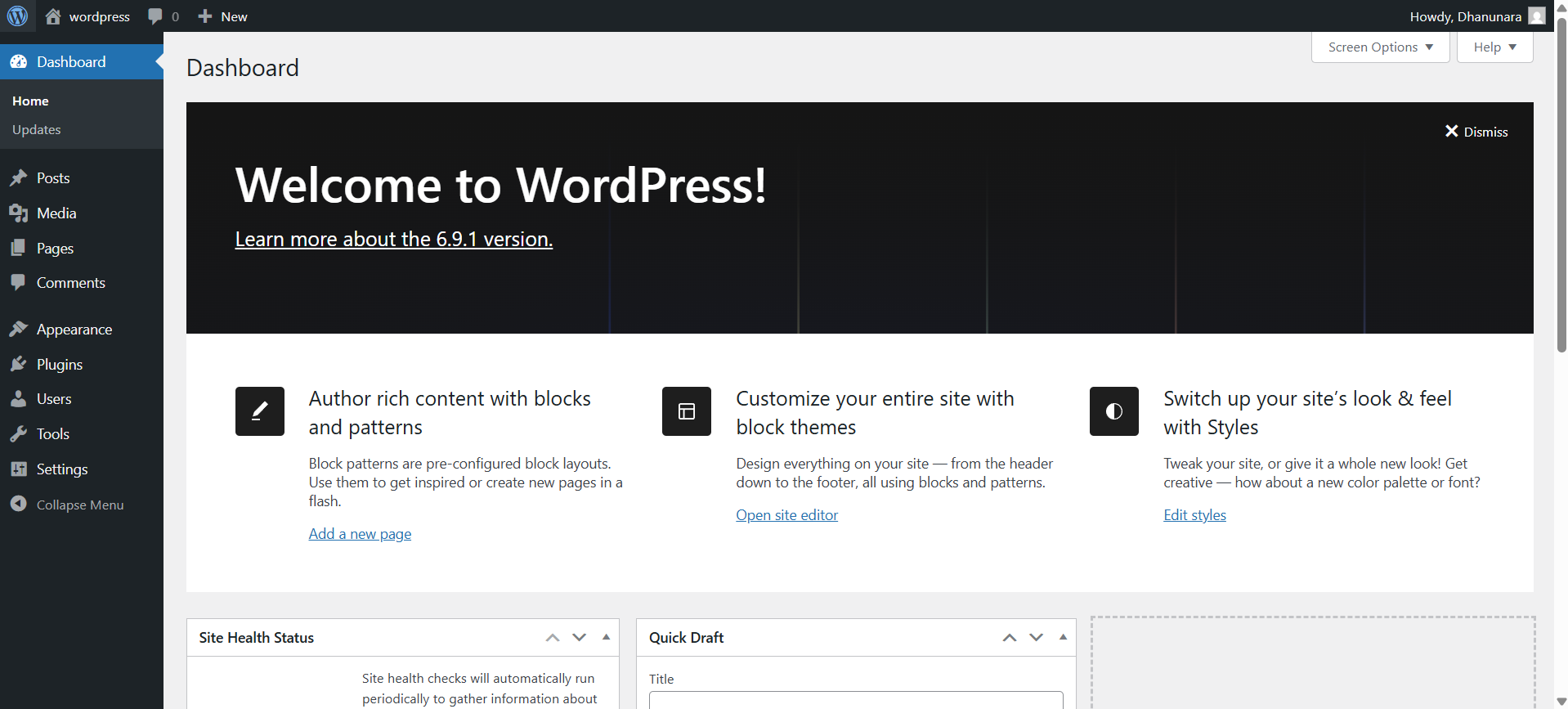
**Complete WordPress Installation**

In browser:

1. Choose Language
2. Click Continue
3. Enter Site Title
4. Enter Admin Username
5. Enter Admin Password
6. Enter Email
7. Click Install WordPress

****

****

****

**Conclusion:**  
Docker Compose is needed because WordPress is a multi-container application, and Compose makes it easy to deploy, manage, and maintain.