**Enterprise Java Microservices CI/CD with Jenkins, GitOps, ArgoCD and Kubernetes.**

**Steps:**

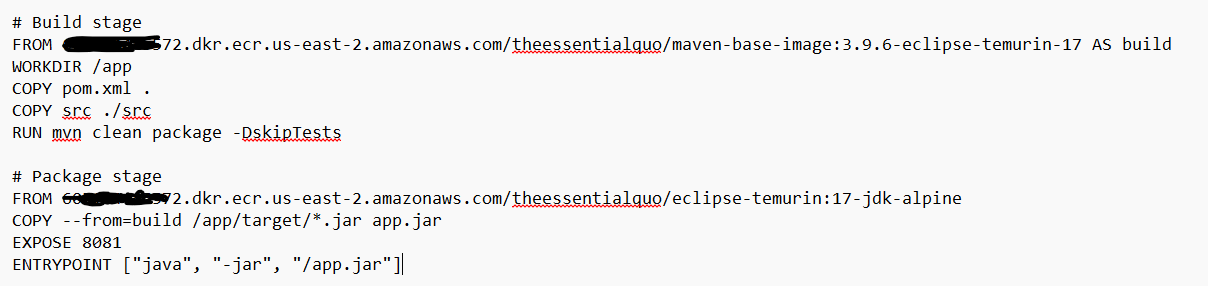
**1.Spring boot microservices image creation**Create the microservices, create the docker images and upload it to ECR.

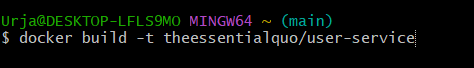
* We have created two java microservices:

User-service: manages users  
Product-service: manages products

* Create the docker images for those services. We will need a Dockerfile in each project folder where we have the pom.xml. The base images that are used in the docker file have been stored on ECR to avoid ‘too many requests’ errors.

Dockerfile :





To test the docker images and the code is running we can run the docker images locally and test on the local host url.

* Once the images have been created, login to ECR and push those images from local to ECR.  
  



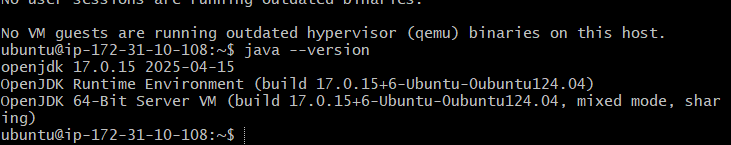
**2.Code commit and webhook trigger with Jenkins. Maven build and unit testing.**

We will be using Jenkins as a CI tool, where the webhook will trigger Jenkins whenever there is a code change.

* Start an EC2 instance (t2.large preferred) and connect it via ssh to install Jenkins on it.
* Connect he ec2 instance via ssh and install java 17 which is required for the Jenkins.

sudo apt update

sudo apt install -y openjdk-17-jdk



* Also install maven since it’s a java project on the ec2 instance  
  sudo apt install maven -y
* Add the Jenkins repository and key

curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key | sudo tee \

/usr/share/keyrings/jenkins-keyring.asc > /dev/null

echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] \

https://pkg.jenkins.io/debian-stable binary/ | sudo tee \

/etc/apt/sources.list.d/jenkins.list > /dev/null

* Install Jenkins

sudo apt update

sudo apt install -y Jenkins

* Start and enable Jenkins

sudo systemctl start jenkins

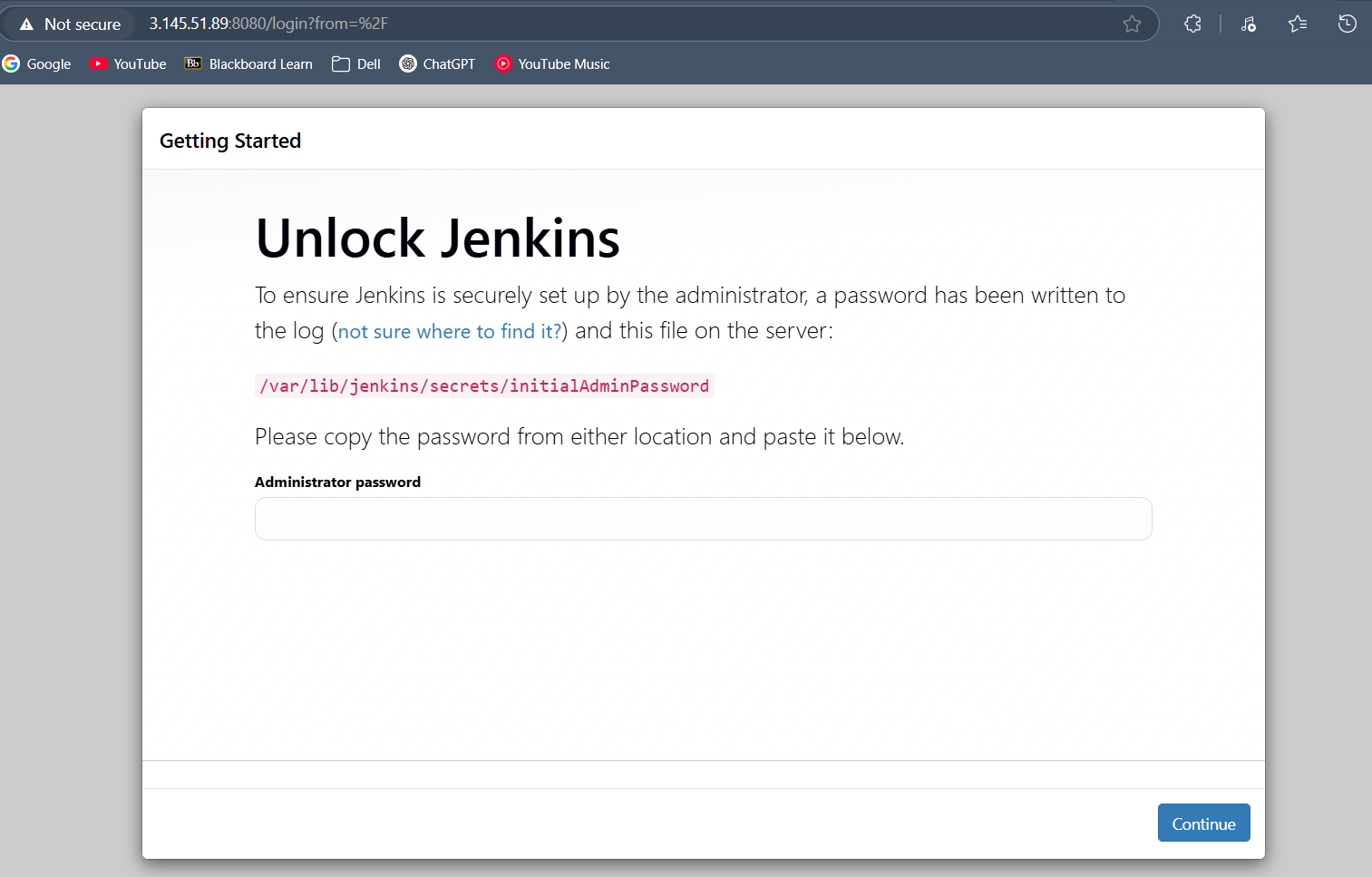
sudo systemctl enable Jenkins

\*\*\*\*(when you restart the ec2 instance, use this command to reload the previous jenkin.service file)\*\*\*\*\*\*\*

Sudo systemctl daemon-reload

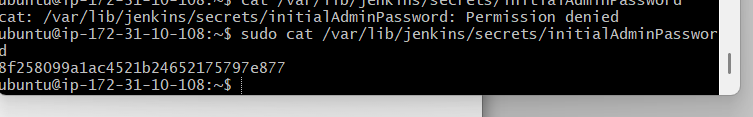
* Now in the browser go to the public ip with port 8080,change the ip when you restart the instance.

<http://3.145.51.89:8080/login?from=%2F>

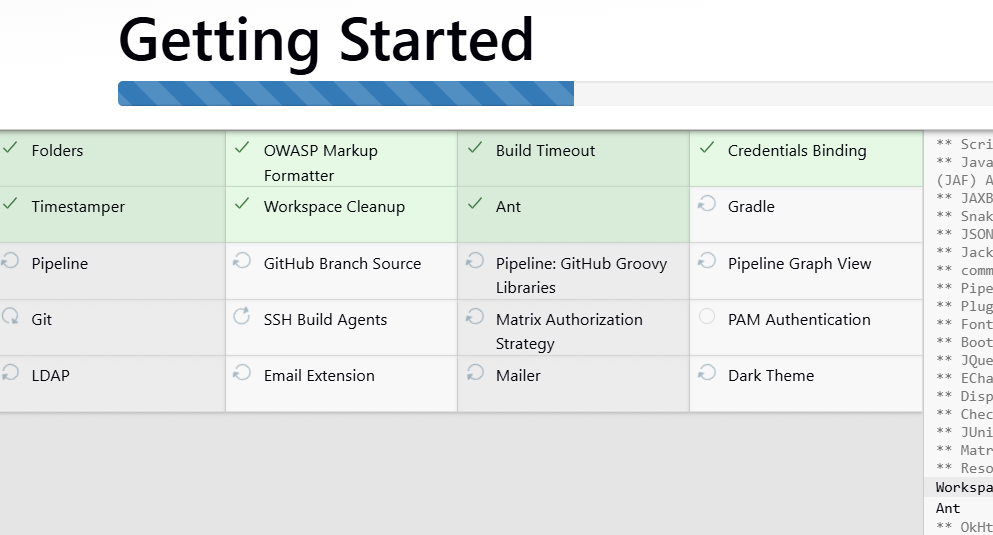


* Then we can logi to Jenkins and for the first time we need to get the password from the location provided.

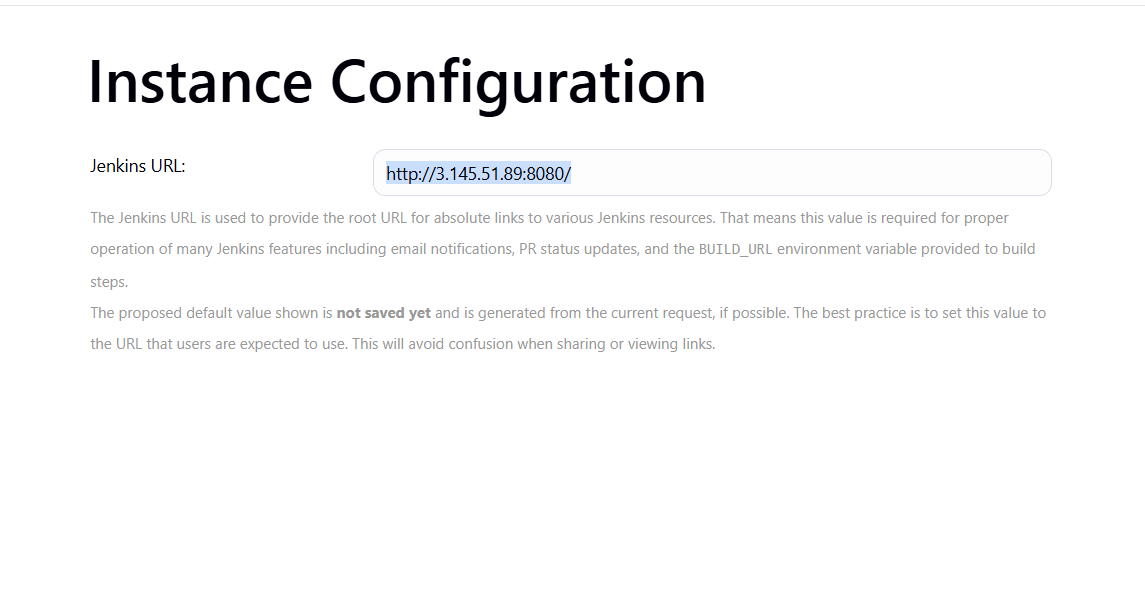
Sudo cat /var/lib/jenkins/secrets/initialAdminPassword

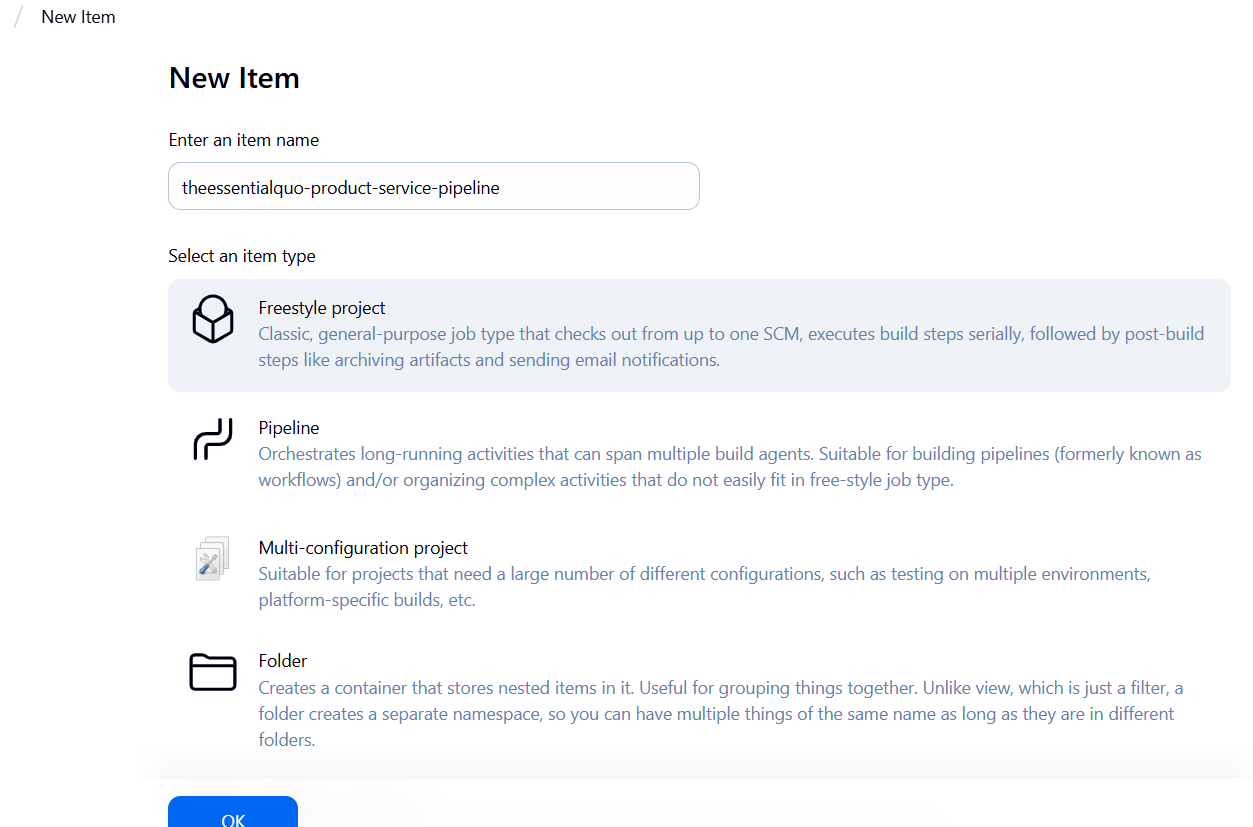


* Install the suggested plugins

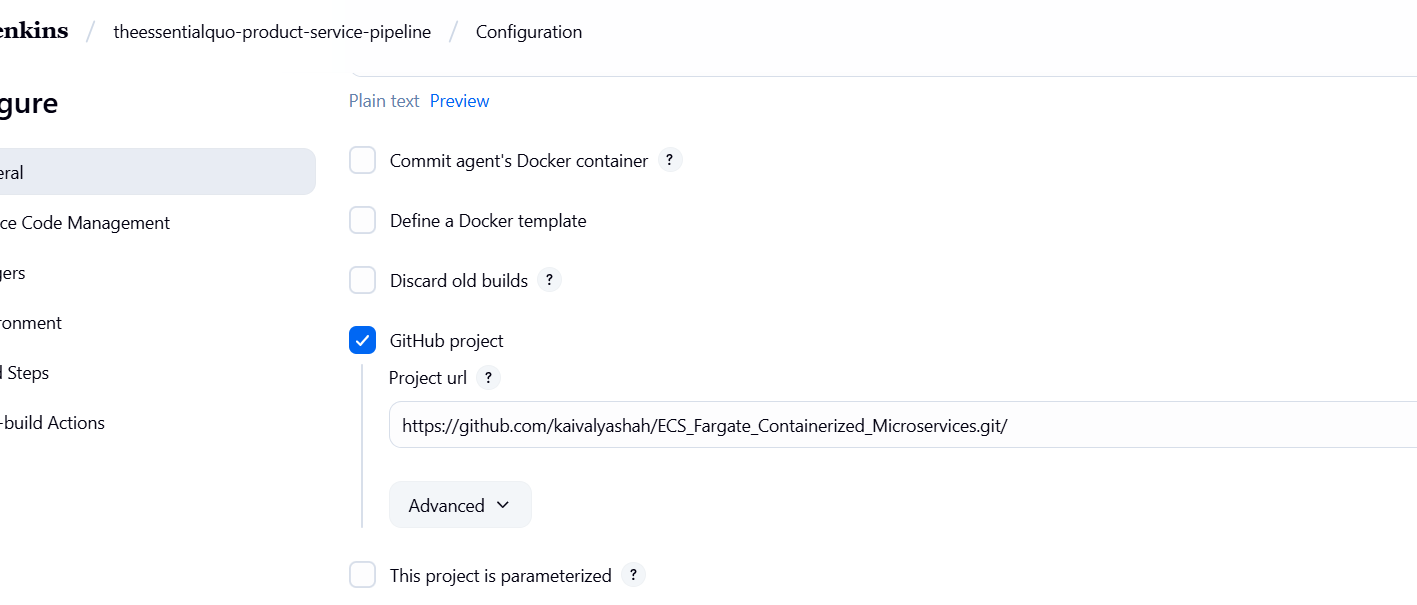


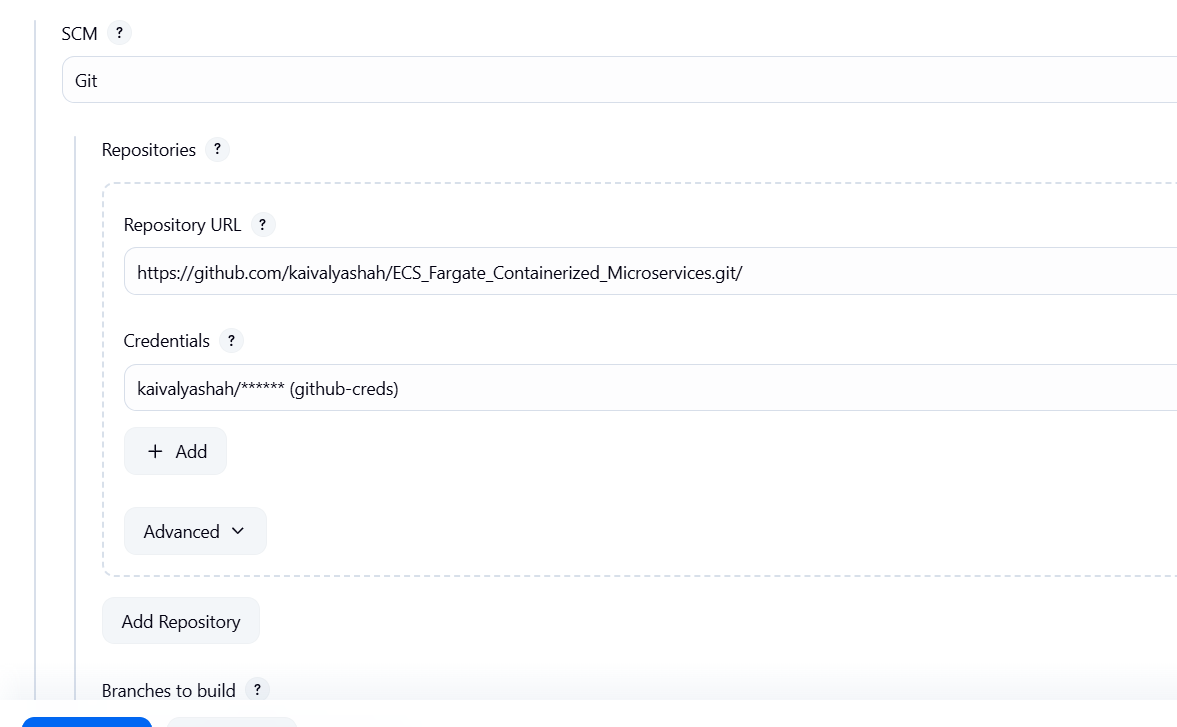
* then we will get a Jenkins url

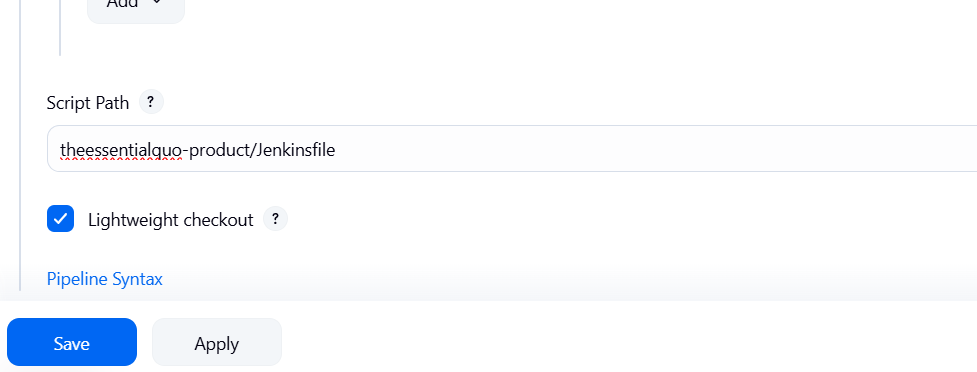


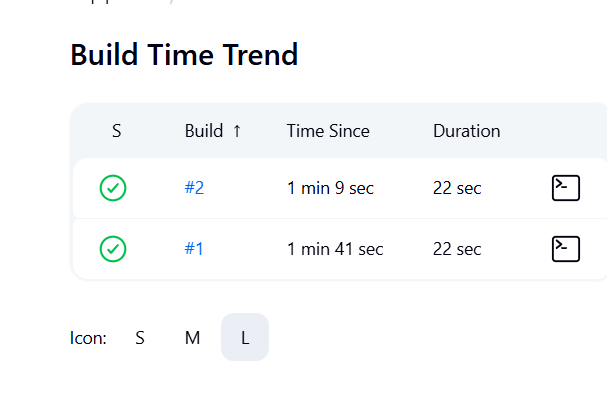
* Now check for the plugins like git, pipeline and aws etc whatever is required.
* Now create a pipeline in Jenkins, a pipeline project.  
  
* Now create a Jenkinsfile which will be used from our git to run the groovy commands to run the build.

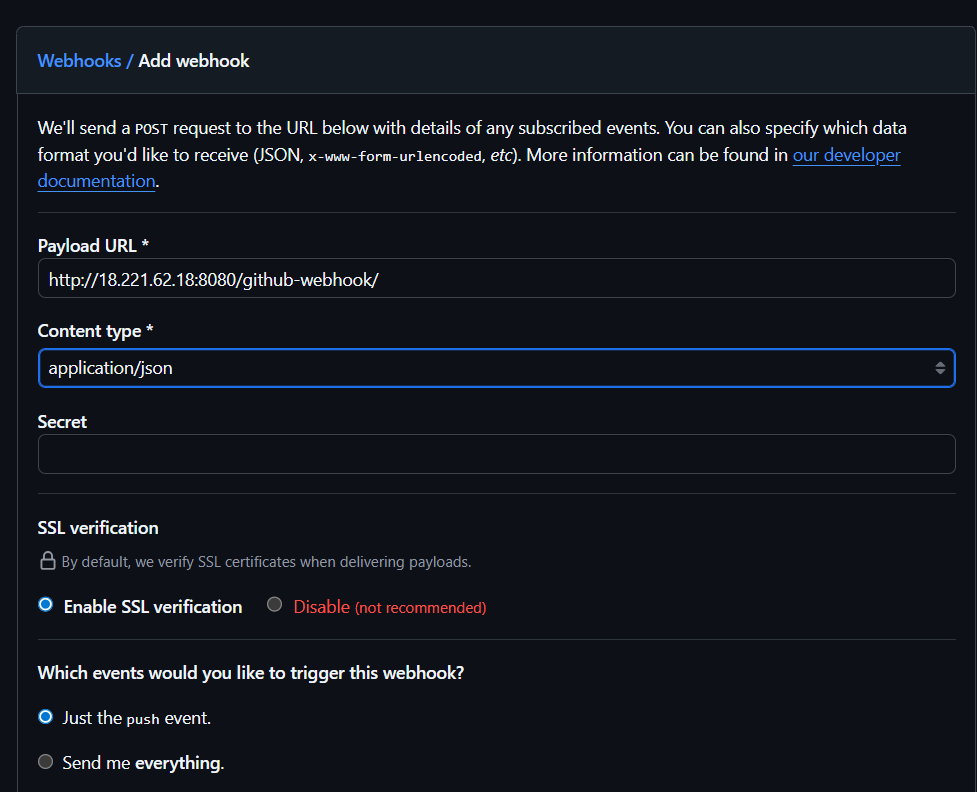


* Now in the pipeline creation select it as a github project and give the repo url.  
   
* Now in the trigger select the github hook trigger.
* In the pipeline select pipeline from scm where the coe will be executed from the jenkinsfile from the scm i.e. here git.



* Give the path to the jenkinsfile correctly where it is located in the git repo.  
  
* Once done we can save the pipeline and run a build.



* Also in the github in your repo settings add a webhook trigger in the webhook section.  
  
* Now every time when there is update in the git by push, the pipeline will be triggered.

**3.Static Code Analysis with SonarQube**

* SonarQube helps to check the code quality of the project. It analyzes your source code for bugs, vulnerabilities, code smells, duplications, and test coverage.
* Jacoco is used for Java code coverage. It’s a Java library/plugin that measures how much of your code is executed by your unit tests.
* SonarQube itself does not run tests.
* Instead, it relies on external tools (like JaCoCo) to generate test coverage reports.
* Then SonarQube imports those reports and shows them in its dashboard.
* Add the sonarqube plugin in Jenkins to let Jenkins.
* Now in the ec2 instance we need to configure it run sonarqube easily. Also it uses postgres to store its data. So to run a sonarqube server and postgres we will use docker, where we will run containers of postgres and sonarqube server which we gt from docker hub.
* First install docker and docker compose in to your ec2 instance. We use docker compose here since we are running multiple conatiners that are related.

sudo apt update && sudo apt upgrade -y

sudo apt install -y docker.io docker-compose

sudo systemctl enable --now docker

sudo usermod -aG docker $USER

* Adjust Kernel Settings (needed for Sonar’s Elasticsearch)

sudo sysctl -w vm.max\_map\_count=262144

sudo sysctl -w fs.file-max=65536

echo -e "vm.max\_map\_count=262144\nfs.file-max=65536" | sudo tee /etc/sysctl.d/99-sonarqube.conf

sudo sysctl –system

* Now write a docker-compose.yml which will have instructions to run the container.

version: "3.8"

services:

db:

image: postgres:15

container\_name: sonarqube-db

environment:

POSTGRES\_USER: sonar

POSTGRES\_PASSWORD: sonar

POSTGRES\_DB: sonarqube

volumes:

- postgres\_data:/var/lib/postgresql/data

sonarqube:

image: sonarqube:community

container\_name: sonarqube

depends\_on:

- db

environment:

SONAR\_JDBC\_URL: jdbc:postgresql://db:5432/sonarqube

SONAR\_JDBC\_USERNAME: sonar

SONAR\_JDBC\_PASSWORD: sonar

ports:

- "9000:9000"

volumes:

- sonar\_data:/opt/sonarqube/data

- sonar\_extensions:/opt/sonarqube/extensions

- sonar\_logs:/opt/sonarqube/logs

volumes:

postgres\_data:

sonar\_data:

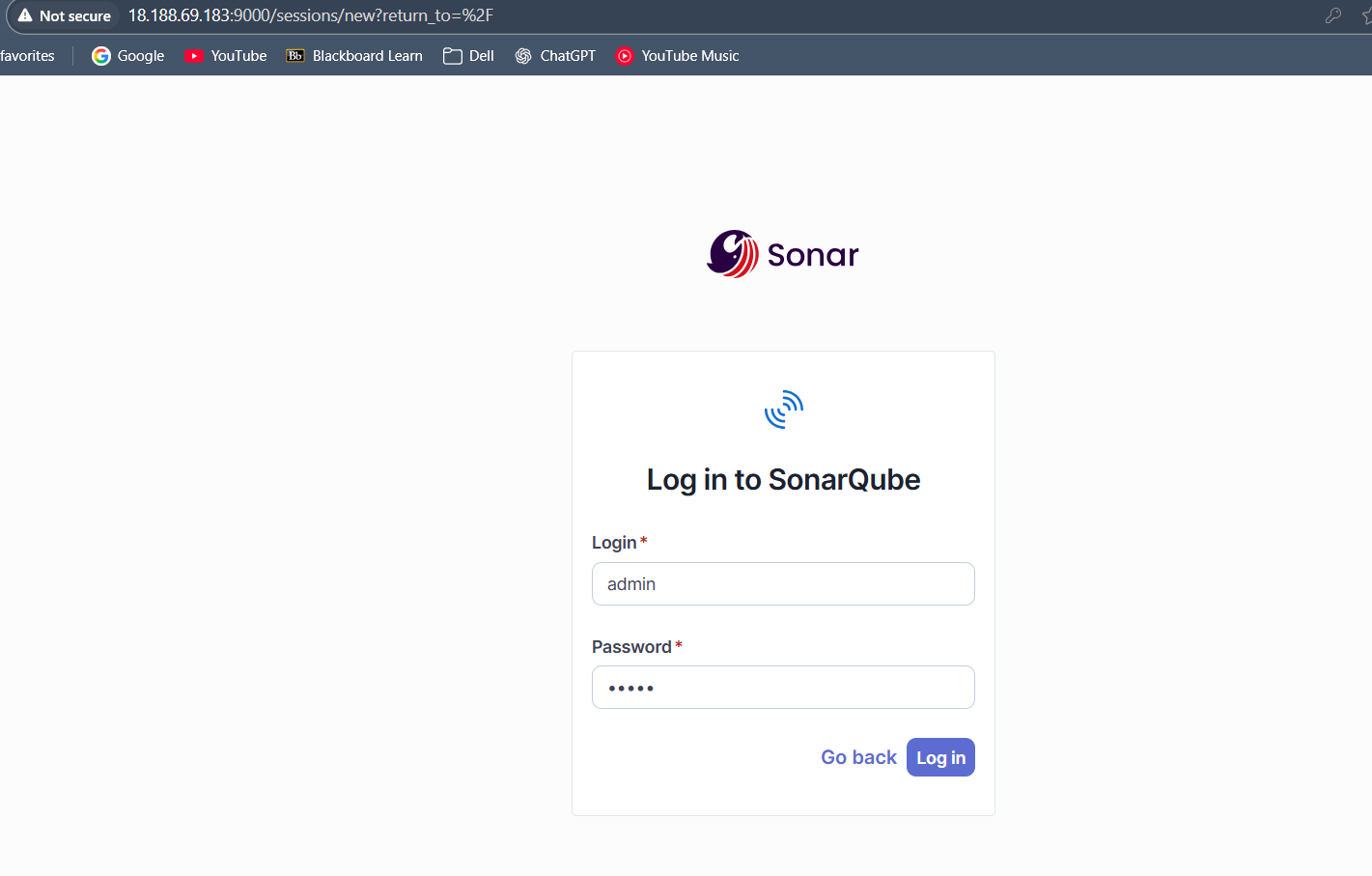
sonar\_extensions:

sonar\_logs:

* Now start sonarqube and postgres db  
  cd /opt/sonar

docker-compose up -d

* Once done open the prowser and go to ip:9000 and login with admin admin.



* Generate Token

Click your profile (top-right) → My Account → Security.

Enter token name (e.g., jenkins-token) → click Generate.

Copy the token (you won’t see it again).

* Integrate Jenkins

In Jenkins → Manage Jenkins → Plugins → install SonarQube Scanner for Jenkins.

Go to Manage Jenkins → System → SonarQube servers:

Name: SonarLocal

Server URL: http://<your-ubuntu-server-ip>:9000

Credentials: add Secret Text → paste your SonarQube token.

Tick “Environment variables”.

* Configure Webhook (for Quality Gate)

In SonarQube: Administration → Configuration → Webhooks → Add

Name: Jenkins

URL: http://<your-ubuntu-server-ip>:8080/sonarqube-webhook/

This lets Jenkins’ waitForQualityGate() block until SonarQube reports Pass/Fail.

* Auto-start on reboot (optional)

# Create a simple systemd unit that brings the stack up at boot

sudo bash -c 'cat >/etc/systemd/system/sonar-compose.service <<EOF

[Unit]

Description=SonarQube via docker compose

Requires=docker.service

After=docker.service

[Service]

Type=oneshot

WorkingDirectory=/opt/sonar

ExecStart=/usr/bin/docker compose up -d

ExecStop=/usr/bin/docker compose down

RemainAfterExit=yes

[Install]

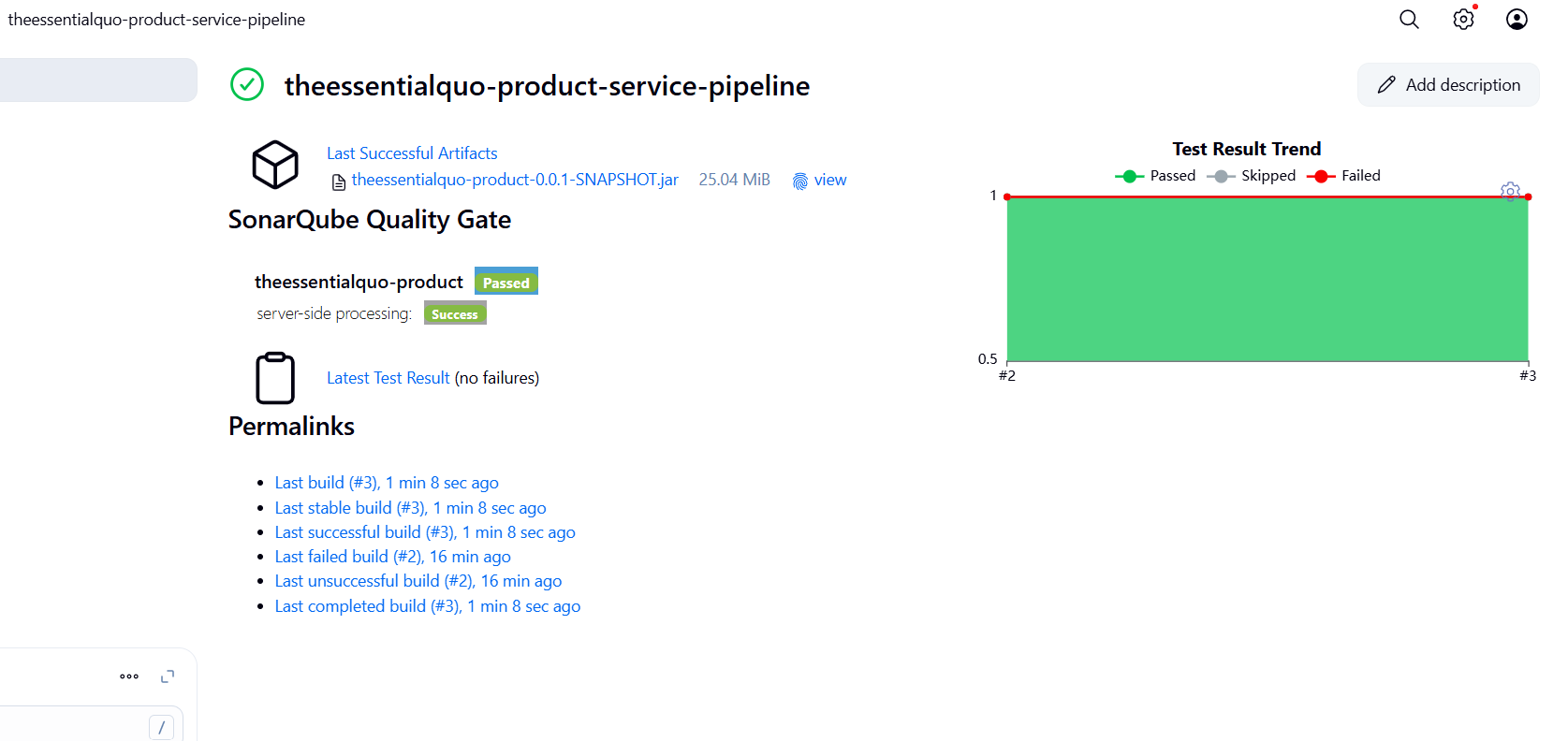
WantedBy=multi-user.target

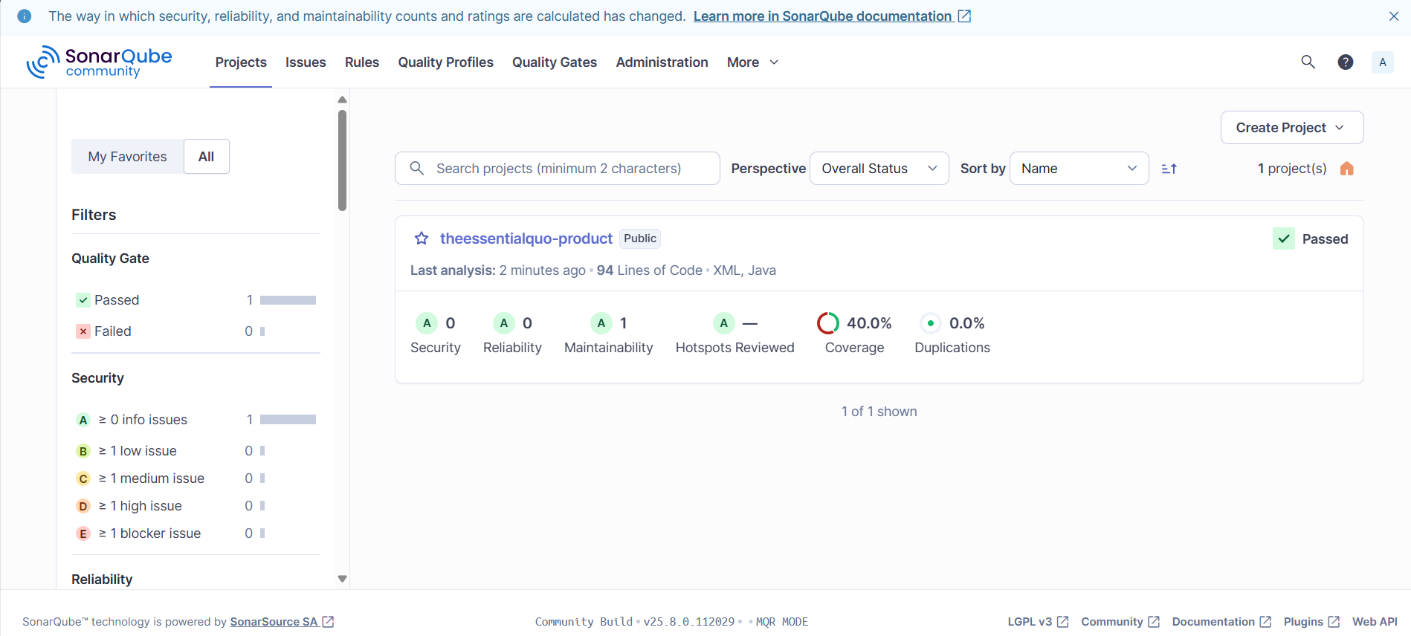
EOF'

sudo systemctl daemon-reload

sudo systemctl enable --now sonar-compose

* Then run the pipeline and check for the sonarqube.





**4.Docker image build and push to ECR**

* Now we need to push docker image of the application to ECR. So we will add these stages to the Jenkinsfile.
* Instal aws cli on the Jenkins instance and configure the aws.
* Make sure Docker is running

sudo systemctl enable --now docker

sudo systemctl status docker --no-pager

* Put the jenkins user in the docker group

sudo usermod -aG docker Jenkins

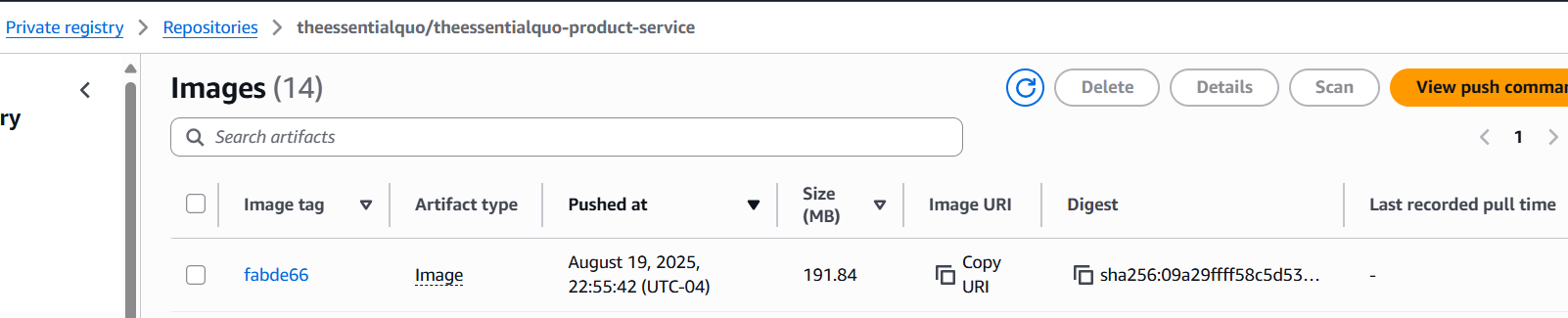
* Restart Jenkins so it picks up the new group

sudo systemctl restart Jenkins

* We have added three stages in the pipeline:  
  stage('ECR Login (for base image pull)')

stage('Docker Build & Tag')

stage('ECR Login & Push')  
So first we will need access to ecr to pull the stored base images to run the java application.Then it builds the image and tags it. Then the final images are pushed to the ecr.



* **Note**: when the instance is stopped and started again, and we are not using the elastic ip, then we should configure the webhooks with the new ip.
  + Update the webhook in the github in the repo setting with the new ip.
  + Run the ‘ sudo systemctl daemon-reload ‘, to reload the daemon.
  + Run the docker compose to run the images of postgres and sonarqube.

cd /opt/sonar

docker-compose up -d

sudo systemctl enable --now sonar-compose

* + Sonarqube -> Administration -> Configuration -> Webhooks -> update ip
  + Jenkins -> manage -> new ip of Jenkins, sonarqube server new ip.

**5.Kubernetes Deployment Via Argo CD (GitOps)**

* Jenkins builds the image → updates Git repo → Argo CD sees it → Kubernetes applies it.
* We will install the CLI tools (kubectl/eksctl/helm), create or connect to the EKS cluster, then install Argo CD and the AWS Load Balancer Controller (for Ingress).
* Configure aws cli on the instance.
* Then install kubectl:  
  KVER=$(curl -L -s https://dl.k8s.io/release/stable.txt)

curl -LO "https://dl.k8s.io/release/${KVER}/bin/linux/amd64/kubectl"

sudo install -m 0755 kubectl /usr/local/bin/kubectl

kubectl version –client

* Then install eksctl:  
  curl -sL "https://github.com/eksctl-io/eksctl/releases/latest/download/eksctl\_$(uname -s)\_amd64.tar.gz" | tar xz

sudo mv eksctl /usr/local/bin/

eksctl version

* Then install helm:  
  curl -s https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash

helm version

* Now create the eks cluster:  
  eksctl create cluster \

--name tq-dev \

--region us-east-2 \

--with-oidc \

--managed \

--nodegroup-name ng \

--nodes 2 \

--node-type t3.large

* Then check for the nodes:  
  aws eks update-kubeconfig --region us-east-2 --name tq-dev

kubectl get nodes

* Now Get nodegroup IAM role name:

NG\_ROLE=$(aws eks describe-nodegroup \

--cluster-name tq-dev --nodegroup-name ng --region us-east-2 \

--query "nodegroup.nodeRole" --output text | awk -F/ '{print $NF}')

And then Attach ECR read-only:

aws iam attach-role-policy \

--role-name "$NG\_ROLE" \

--policy-arn arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly

* Ensure OIDC is associated (safe to run again)

eksctl utils associate-iam-oidc-provider --cluster tq-dev --region us-east-2 --approve

* Policy for controller

curl -o iam\_policy.json https://raw.githubusercontent.com/kubernetes-sigs/aws-load-balancer-controller/v2.7.1/docs/install/iam\_policy.json

aws iam create-policy --policy-name AWSLoadBalancerControllerIAMPolicy --policy-document file://iam\_policy.json \

|| echo "Policy may already exist"

* Service account with that policy

ACCOUNT\_ID=$(aws sts get-caller-identity --query Account --output text)

eksctl create iamserviceaccount \

--cluster tq-dev \

--namespace kube-system \

--name aws-load-balancer-controller \

--attach-policy-arn arn:aws:iam::${ACCOUNT\_ID}:policy/AWSLoadBalancerControllerIAMPolicy \

--region us-east-2 \

--override-existing-serviceaccounts \

--approve

* Install the controller

VPC\_ID=$(aws eks describe-cluster --name tq-dev --region us-east-2 --query "cluster.resourcesVpcConfig.vpcId" --output text)

helm repo add eks https://aws.github.io/eks-charts

helm repo update

helm upgrade --install aws-load-balancer-controller eks/aws-load-balancer-controller \

-n kube-system \

--set clusterName=tq-dev \

--set region=us-east-2 \

--set vpcId=$VPC\_ID \

--set serviceAccount.create=false \

--set serviceAccount.name=aws-load-balancer-controller

* Then check the status with this kubectl command:  
  kubectl -n kube-system rollout status deploy/aws-load-balancer-controller
* Install Argo CD:

kubectl create namespace argocd || true

kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml

kubectl -n argocd patch svc argocd-server -p '{"spec":{"type":"LoadBalancer"}}'

kubectl -n argocd get svc argocd-server

* Grab the external address, login with admin and the password:

kubectl -n argocd get secret argocd-initial-admin-secret \

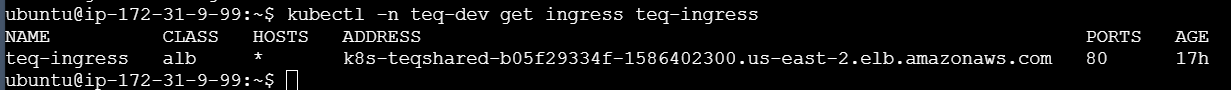
-o jsonpath='{.data.password}' | base64 -d; echo

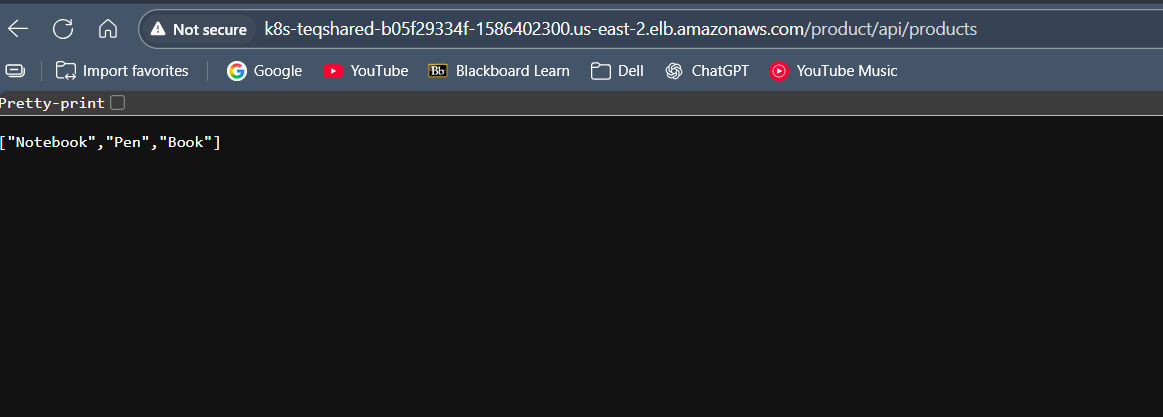
* Quick health checks:

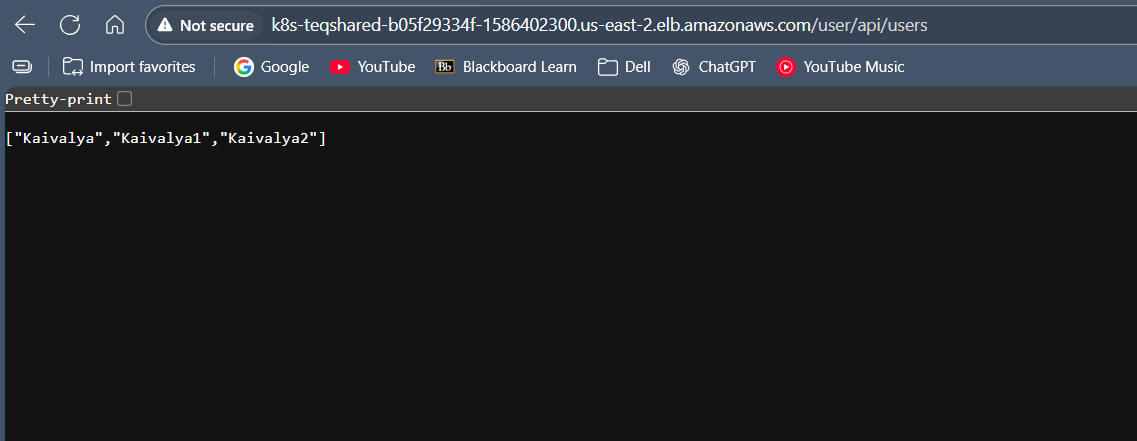
kubectl get nodes -o wide

kubectl get pods -A

* Then create the service.yaml, depoloyment.yaml and teq-ingress.yaml where the manifest are written.
* Create a git repo teq-gitops and push these files where argocd will detect any changes and deploy the cluster as per the manifests.
* Once the pods are running we can verify it via the ingress ALB DNS address.







**6.Monitoring Deployment**

* First we need to have mocrometer, Actuator, Prometheus , in the pom.xml as dependencies.
* Install Prometheus and Grafana stack via Helm chart.

kubectl create namespace monitoring

helm repo add prometheus-community https://prometheus-community.github.io/helm-charts

helm repo update

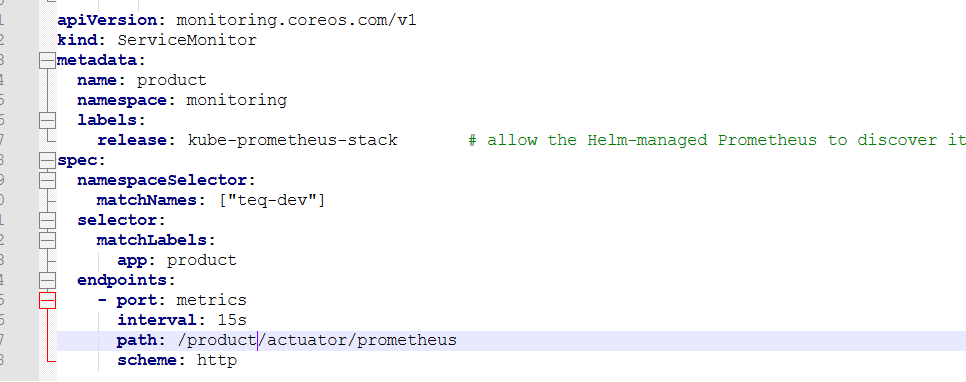
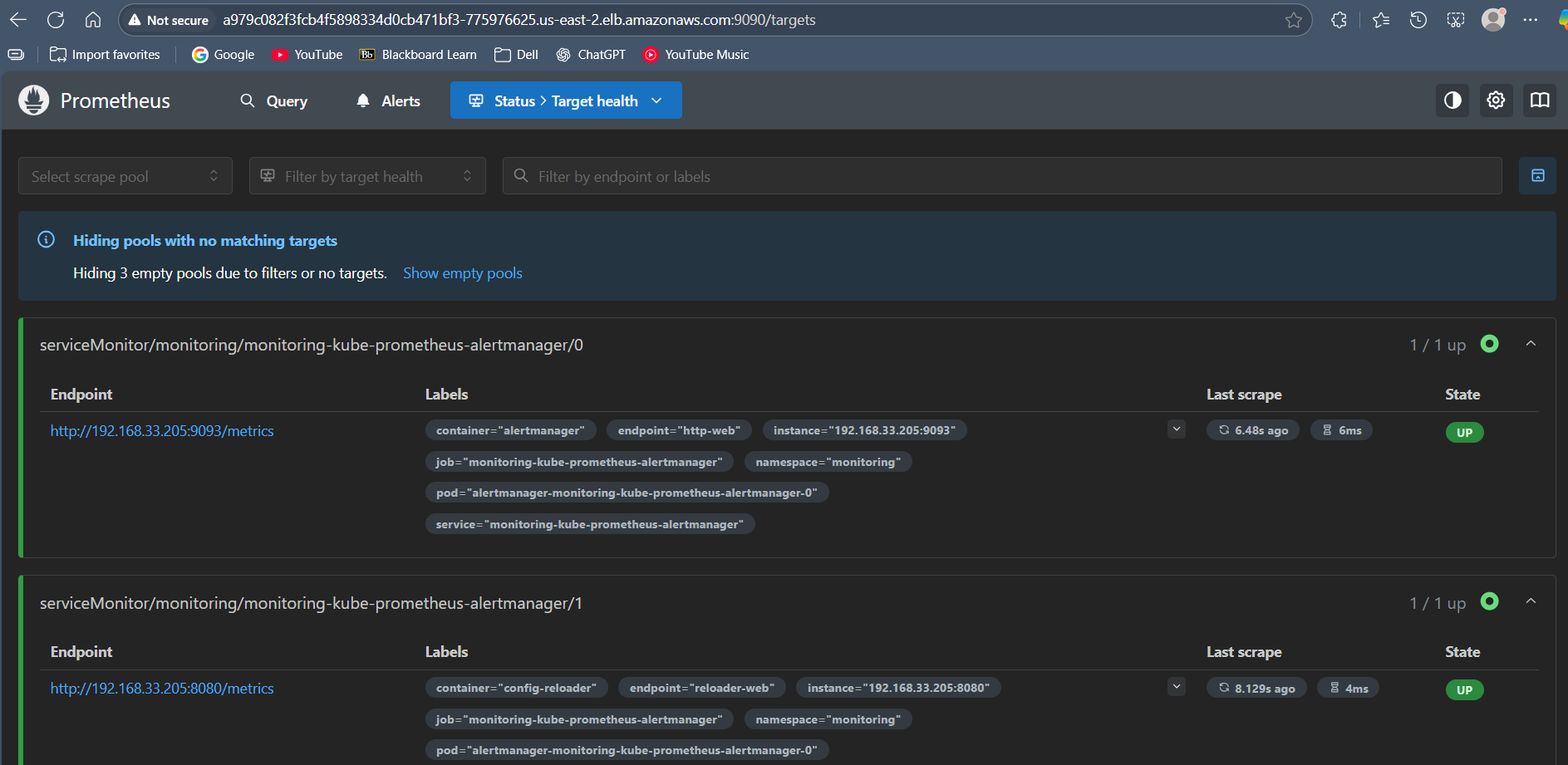
# Basic install (good defaults). You can add a custom values file later.

helm upgrade -i kube-prometheus-stack prometheus-community/kube-prometheus-stack \

-n monitoring \

--set grafana.defaultDashboardsEnabled=true \

--set prometheus.prometheusSpec.scrapeInterval="15s"

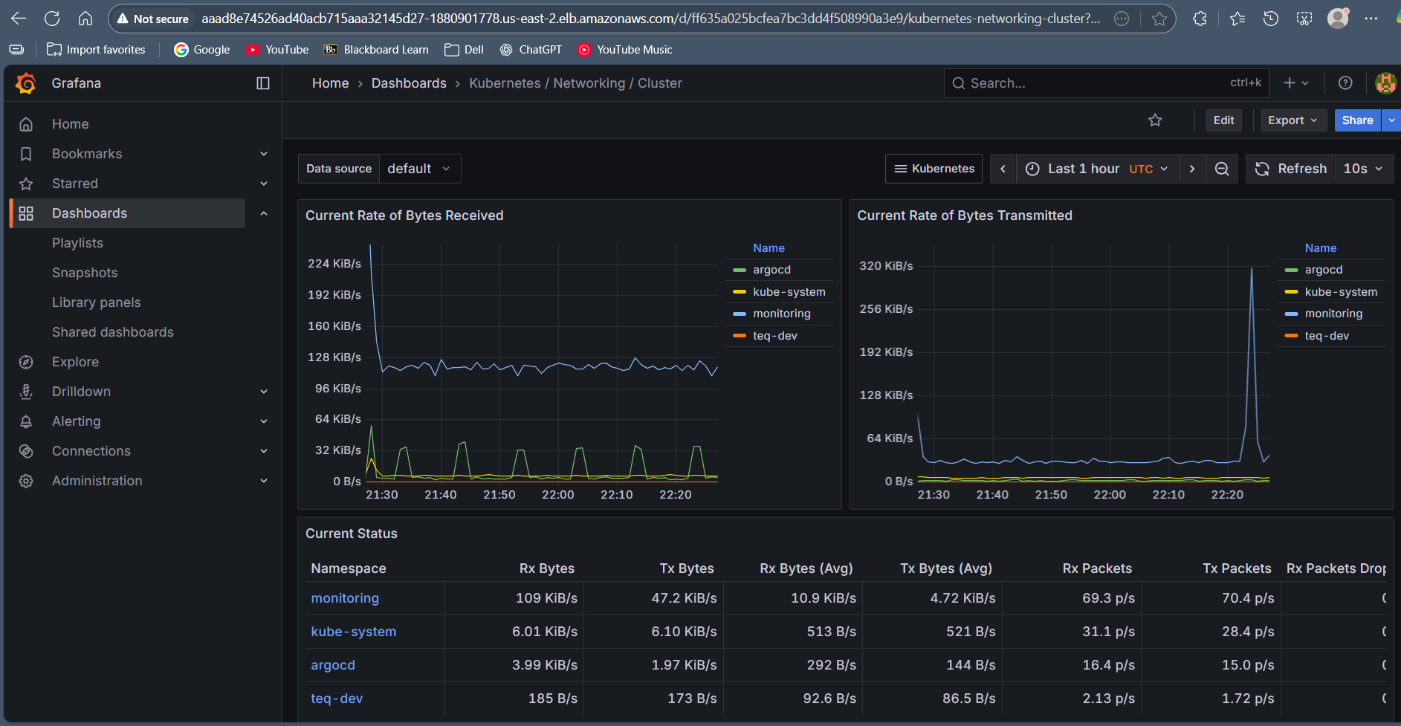
* Now update the service.yaml of both services for exposing Prometheus.  
  add this :  
    
  
* Prometheus-> Status → Targets should show your scrape jobs (kube-state-metrics, node-exporter, etc.) as UP.  
  
* Grafana -> Dashboards → Manage. Look for:

Kubernetes / Compute Resources / Namespace (Pods, Nodes) – shows CPU, memory per namespace.

Kubernetes / Networking – traffic and errors.

Kubernetes / API Server – API health and request latency.

Node Exporter / Nodes – CPU, memory, disk, network usage at node level.

* 

**FLOW:  
Flow (end-to-end)**

1. **Developer pushes code** to the app repo (e.g., product service).
2. **Jenkins pipeline** runs:
   * builds the JAR (Maven) → builds a **Docker image**,
   * pushes the image to **ECR** (tagged with commit SHA and/or latest),
   * opens your **GitOps repo** and edits environments/dev/product/deployment.yaml to set:
   * image: <acct>.dkr.ecr.<region>.amazonaws.com/...-product-service:<new-sha>
   * commits & **pushes** that YAML change.
3. **Argo CD** is watching the GitOps repo path (e.g., environments/dev/product):
   * sees the new commit,
   * shows the app as **OutOfSync**, then **Syncs** (automated),
   * applies the updated Deployment to **EKS**.
4. **EKS** pulls the new image from **ECR** and performs a **rolling update**:
   * spins up new pods with the new image,
   * **readiness/liveness probes** make sure pods are healthy before traffic shifts,
   * old pods are terminated once new ones are ready.
5. **Service** (ClusterIP) keeps a stable virtual name (product, user) and points to the healthy pods via labels.
6. **Ingress** (managed by the AWS Load Balancer Controller) ensures:
   * a single **ALB** exists for your namespace,
   * **path-based routing**:
     + http(s)://ALB-DNS/users → Service: user → Pods (port 8080)
     + http(s)://ALB-DNS/products → Service: product → Pods (port 8082)
   * optional TLS via an ACM certificate on the ALB.

**References:**

<https://www.youtube.com/watch?v=e42hIYkvxoQ>

<https://www.youtube.com/watch?v=Gd9Aofx-iLI>

Two : <https://www.youtube.com/watch?v=jNPGo6A4VHc&t=1735s>

<https://www.youtube.com/watch?v=JGQI5pkK82w&t=27s>