Project 1

Demo Project:
Complete CI/CD Pipeline with EKS and AWS ECR

Technologies used:
Kubernetes, Jenkins, AWS EKS, AWS ECR, Java, Maven, Linux, Docker, Git

Project description:
- Create private AWS ECR Docker repository.
- Adjust Jenkinsfile to build and push Docker Image to AWS ECR.
- Integrate deploying to K8S cluster in the CI/CD pipeline from AWS ECR private registry.
- So the complete CI/CD project we build has the following configuration:
    a. CI step: increment version
    b. CI step: Build artifact for Java Maven application
    c. CI step: Build and push Docker image to AWS ECR
    d. CD step: Deploy new application version to EKS cluster
    e. CD step: Commit the version update

What I've Learned:
- Setting up and managing private Docker repositories in AWS ECR.
- Adjusting Jenkins pipelines to build and push Docker images to AWS ECR.
- Automate deployment of applications to Kubernetes (EKS) clusters from private Docker registries.
- Create and manage a full CI/CD pipeline with Jenkins, including versioning, building, and deploying applications.
- The importance of integrating container registries with CI/CD pipelines to ensure secure and efficient deployments.
- Best practices for managing Kubernetes deployments in a CI/CD environment, ensuring consistent and reliable application updates.

Table of Contents

1 Prerequisites

Install & configure AWS CLI

Follow instructions at https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html

Create AWS user for this project

~~**Option 1: Create new IAM user**~~

~~Create IAM User: UserForProjects:password~~

~~Create access key → Use case: Command line interface (CLI)~~

~~Access key~~	~~XXXXXXXXXXXXXXXXXX~~
~~Secret access key:~~	~~xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx~~
~~Region~~	~~eu-central-1~~
~~Output format~~	~~json~~

Option 2: Create new IAM Identity Center user

Go to AWS IAM Identity Center (not IAM) in the desired region.

Enable IAM Identity Center.

Enable with AWS Organizations.

Create user:
ProjectsUser
Add email
Create password over mail

Follow video: https://www.youtube.com/watch?v=_KhrGFV_Npw

```
aws configure sso
```

Session name: my-sso
Start url: https://d-9967601073.awsapps.com/start
(found in IAM Identity Center Dashboard, right side, Settings summary
Region: eu-central-1
Scope: (empty)

To use this profile on aws commands, specify the –profile flag, like in ths example:

```
aws s3 ls --profile AdministratorAccess-488378077264
```

If logged out, re-login with:

```
aws sso login --profile AdministratorAccess-488378077264
```

Prepare Java application to containerize

Fork this repo (including all branches): https://gitlab.com/twn-devops-bootcamp/latest/11-eks/java-maven-app

Forked to: https://gitlab.com/redjules/java-maven-app

Clone it locally and put it inside a folder called 'app':

ProjectName
├ app → *Put repository here*
│ ├ *src*
│ └ … (other files)
└ Dockerfile

Prepare Java environment (Java SDK and Maven)

Java SDK

Download Java SDK installer and install.

Maven

Download and decompress Maven binary zip.

Move folder to program files (or anywhere you want).

Add “C:\Program Files\apache-maven-3.9.9\bin” to Environment Variables > System Variables > Path

Test it with: `mvn --version`
Note: VS Code needs to be restarted (close all instances!) so that changes in PATH are recognized.

mvn clean package → Deletes previous builds, builds the project, generates a jar file inside ‘target’ folder

~~**Gradle** (Not needed for this project)~~

~~Download and decompress Gradle zip.~~

~~Move folder to program files (or anywhere you want).~~

~~Add “C:\Program Files\apache-maven-3.9.9\bin” to Environment Variables > System Variables > Path~~

~~gradle init~~

~~gradle build~~

Prepare sample Docker image

Prepare a Dockerfile next to the app folder:

```
Project1
├ app
| └ … (files of the java app)
└ Dockerfile
```

Dockerfile:
This Dockerfile builds the Java app into a tar file, and then runs it.

```
# Start from a Maven image for building Java applications
FROM maven:3.9.4-eclipse-temurin-17-alpine as builder

# Set the working directory inside the container
WORKDIR /usr/app

# Copy the Maven project files (pom.xml and source code)
COPY ./app /usr/app

# Build the Java application using Maven
RUN mvn clean package



# Use a lightweight OpenJDK image to run the application
FROM openjdk:8-jre-alpine

# Set the working directory inside the container
WORKDIR /usr/app

# Copy the built JAR file from the builder stage
COPY --from=builder /usr/app/target/*.jar app.jar

# Set the command to run the Java application
CMD ["java", "-jar", "app.jar"]
```

Run this command (Docker must be running):

```
docker build -t my-app:1.0 .
```

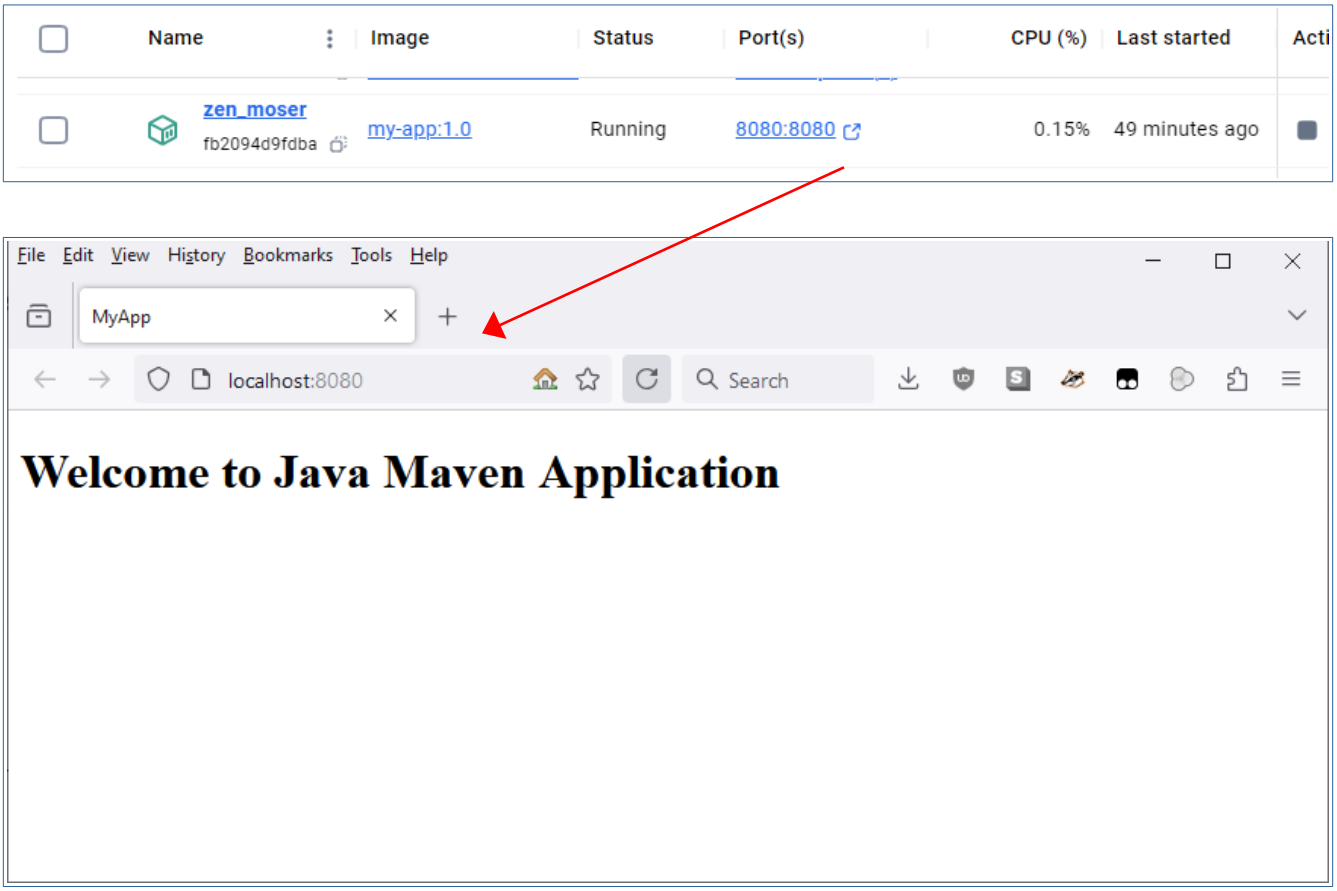
Confirm images has been created: `docker images`

Run image: `docker run -p 8080:8080 my-app:1.0`

Confirm container is running: `docker ps`

Get logs: `docker logs fb20`

Open shell inside container: `docker exec -it fb20 /bin/sh`



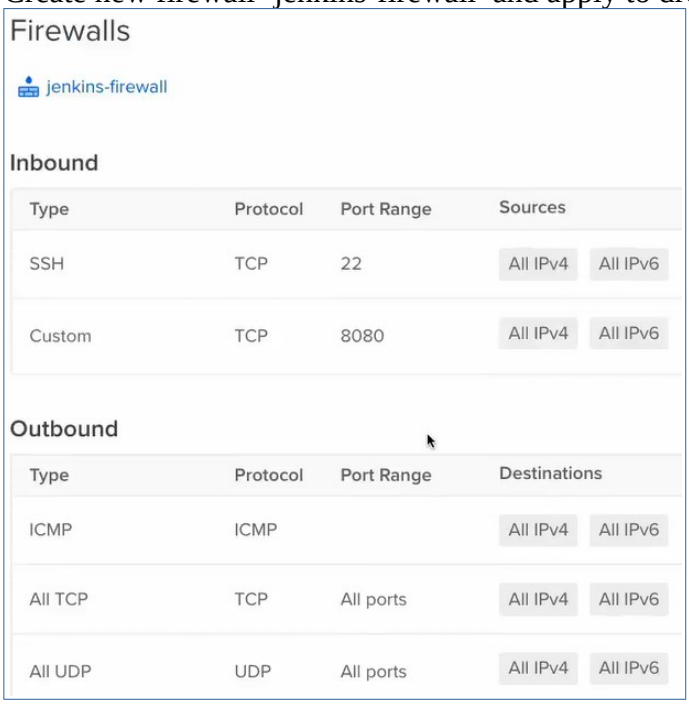
2 Prepare Jenkins Pipeline

Prepare Jenkins server

Option 1 is the method taught in the DevOps bootcamp, but it costs money and for learning purposes it can be replicated locally for free with a Docker container.

Option 1: Jenkins on a DigitalOcean droplet

~~Go to Digital Ocean → Create Droplet~~
~~Region: Amsterdam~~
~~Image is Ubuntu. CPU options: Regular, 8GB 160GB 5TB~~
~~Create Droplet~~
~~Change name of droplet: jenkins-server~~
~~Create new firewall 'jenkins-firewall' and apply to droplet:~~



~~Copy droplet's IP.~~
~~`ssh root@IP`~~

~~Install docker:~~
~~`apt update`~~
~~`apt install docker.io`~~

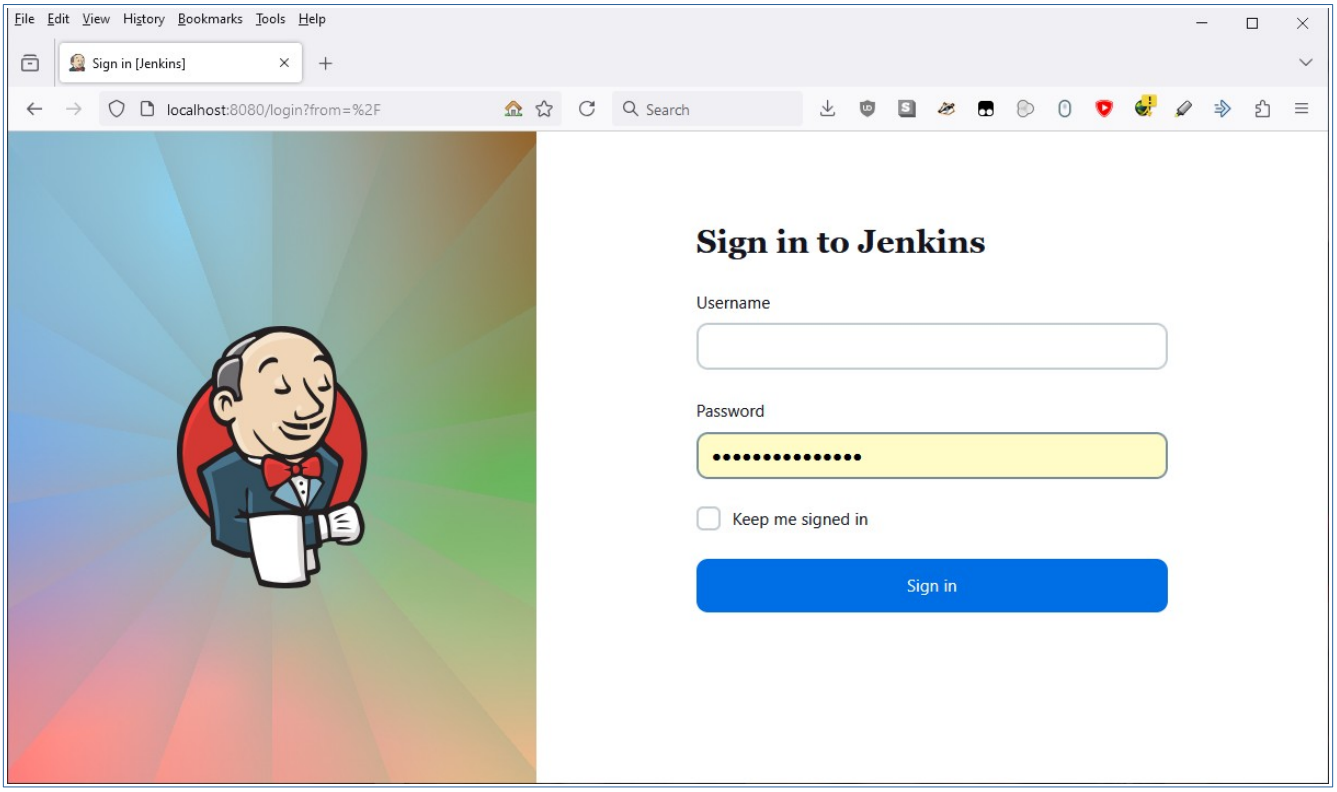
Option 2: Jenkins on a local Docker container

```
docker run -p 8080:8080 -p 50000:50000 -d -v
jenkins_home:/var/jenkins_home jenkins/jenkins:lts
```

Name	Image	Status	Port(s)	CPU (%)	Last started	Actions
hardcore_blackt 4176cecba8a3	jenkins/jenkins:lts	Running	50000:50000 Show all ports (2)	0.21%	6 minutes ago	

Port 50000 is for communication between Jenkins master and workers when Jenkins is on a cluster (not needed for now).

Port 8080 is for browser access. We can access through http://*localhost:8080*.




```
docker ps    # Check the container id, replace it in the next line
docker exec -it 4176cecba8a3932e bash
cat /var/jenkins_home/secrets/initialAdminPassword → Get admin pass
exit
```

Login with admin password → 0e941dc150d8430085a755b9a048f20b

Install suggested plugins
Create first admin user: etiron:P^,uHnv_R=#F8V)
Jenkins URL: http://localhost:8080

Prepare Jenkins to build Java apps

On the Jenkins UI → Tools → Add Maven
Name: maven-3.9
Version: 3.9.9 (or later)
Save

Add Jenkins job

[Do this on Chrome. Jenkins is buggy on Firefox and some menus don't work properly]

Jenkins GUI → Create job: name java-app-job, type Freestyle

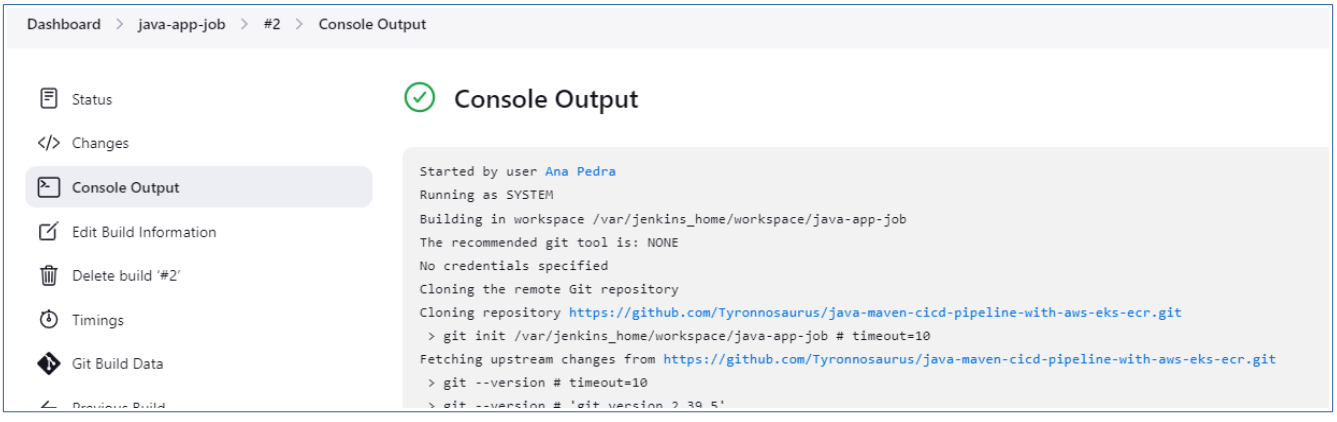
Build steps → Invoke top-level Maven targets
Maven version: maven-3.9
Goals: --version

Dashboard → java-app-job → Configure → Source code management → Select Git

Set https url of the repository (https://gitlab.com/redjules/java-maven-app.git)
Add username & password form Gitlab
ID: gitlab-credentials

Branch: change to `*/jenkins-jobs` (or whichever branch we want to build)

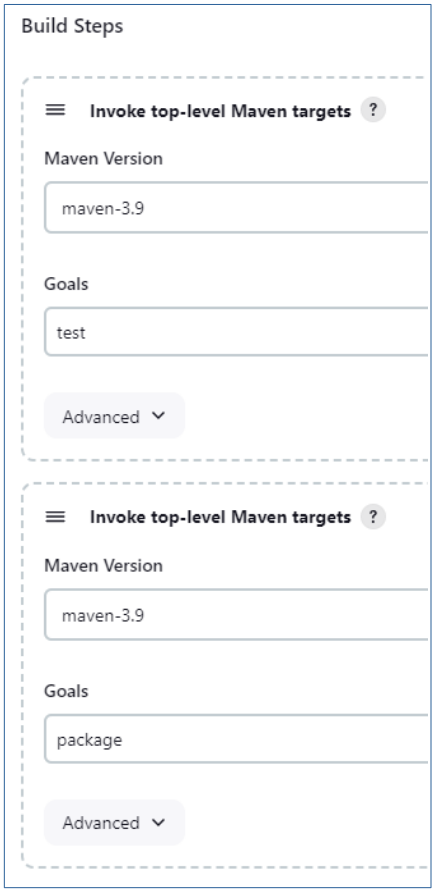
Go back to the job page and click *Build Now*. From now on, the job will first clone the repository at the start of every build, as seen on the build's output:



[The following sections are useful to understand the Build Steps, but we will ultimately discard them and use a Jenkinsfile instead. **Feel free to skip to section "Use a pipeline job with Jenkinsfile"**]

Configure job's Build Steps

Change 'Build steps' to these 2 steps:



Now, when we run the job it will have a Test step and a Build step:

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running AppTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.204 s -- in AppTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  16.261 s
[INFO] Finished at: 2024-10-26T14:43:14Z
[INFO] ------------------------------------------------------------------------
Finished: SUCCESS
```

Build Docker image inside Jenkins

Make docker available in Jenkins container

Stop the Jenkins container and rerun it with an extra volume so that the Jenkins container has access to the host's Docker engine (if it's installed):

On Linux/Mac:

```
docker run -p 8080:8080 -p 50000:50000 -d -v jenkins_home:/var/jenkins_home -v
/var/run/docker.sock:/var/run/docker.sock jenkins/jenkins:lts
```

On Windows:

```
docker run -p 8080:8080 -p 50000:50000 -d -v jenkins_home:/var/jenkins_home -v
"//var/run/docker.sock:/var/run/docker.sock" jenkins/jenkins:lts
```

Install Docker inside Jenkins container:

```
docker ps  # Get container id
docker exec -u 0 -it 8114c72b2a66 bash
curl https://get.docker.com/ > dockerinstall && chmod 777 dockerinstall &&
./dockerinstall
```

Set permissions of the Docker socket file so that we can run

```
ls -l /var/run/docker.sock
chmod 666 /var/run/docker.sock # Read/write for all users
docker ps  # Test that docker cmd works within the container
exit
```

Add a build step to build an image

We have a Dockerfile in the jenkins-jobs branch. Unlike the one seen previously, this one doesn't build the app's jar (since it has already been built by Jenkins), and merely copies and runs it:

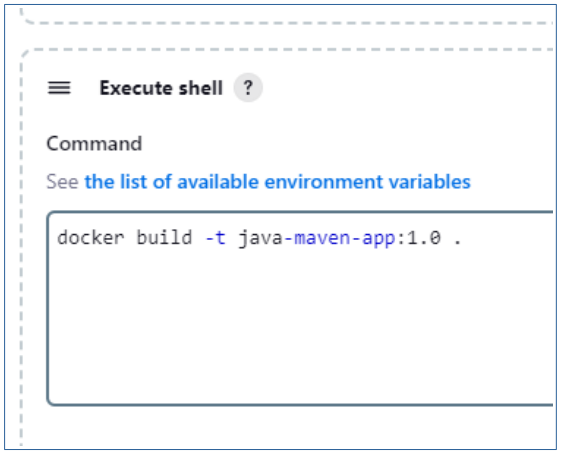
```
FROM openjdk:8-jre-alpine

EXPOSE 8080

COPY ./target/java-maven-app-*.jar /usr/app/
WORKDIR /usr/app
```

```
CMD java -jar java-maven-app-*.jar
```

Add a third build step to the job:



3 Use a pipeline job with Jenkinsfile

Configuring the job stages on the GUI is easy and intuitive but only for simple configurations. It is usually better to use a Jenkinsfile instead.

Prepare Jenkinsfile & use it in a job

We create a new job '*java-maven-pipeline*' of type *Pipeline*.
Configure it with 'Pipeline script from SCM'.
Add the repository url and credentials.
Branch: */jenkins-jobs

Script path: Jenkinsfile

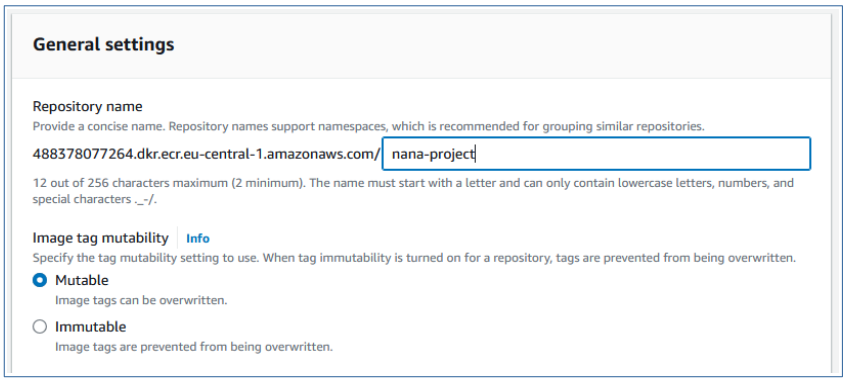
This Jenkinsfile consists of 5 steps:

1. Clone the project from Gitlab
2. Check the current app version (in pom.xml) and calculate the numbering of the next version
3. Build the app
4. Build the app's image & push it to our AWS ECS repo
5. Deploy the updated app in the K8s cluster by reapplying the manifests for the deployment and service.

Prepare AWS ECS Registry

Go to AWS ECR > Create repo

- Name: nana-project
- Mutable (default)
- AES-256 (default)



Login docker into repo

Go to the ECR repo we created → View push commands. We should find a command like this:

```
# Add the AWS registry in Docker
aws ecr get-login-password --region eu-central-1 --profile
AdministratorAccess-488378077264 | docker login --username AWS --
password-stdin 488378077264.dkr.ecr.eu-central-1.amazonaws.com
```

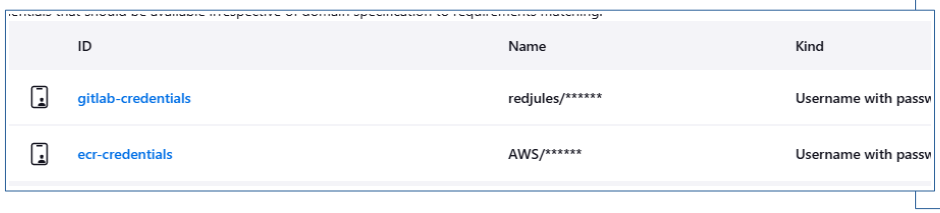
We need to do two things with this command:

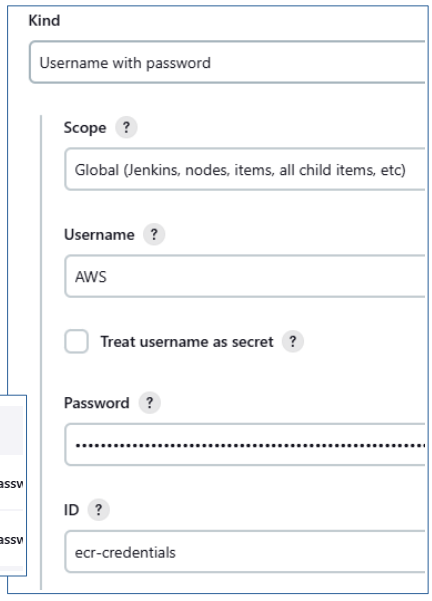
- Copy the repo url (`488378077264.dkr.ecr.eu-central-1.amazonaws.com`) to the Jenkinsfile (as the environment variable DOCKER_REPO_SERVER).

- Run the first part of the command to get the repo password:

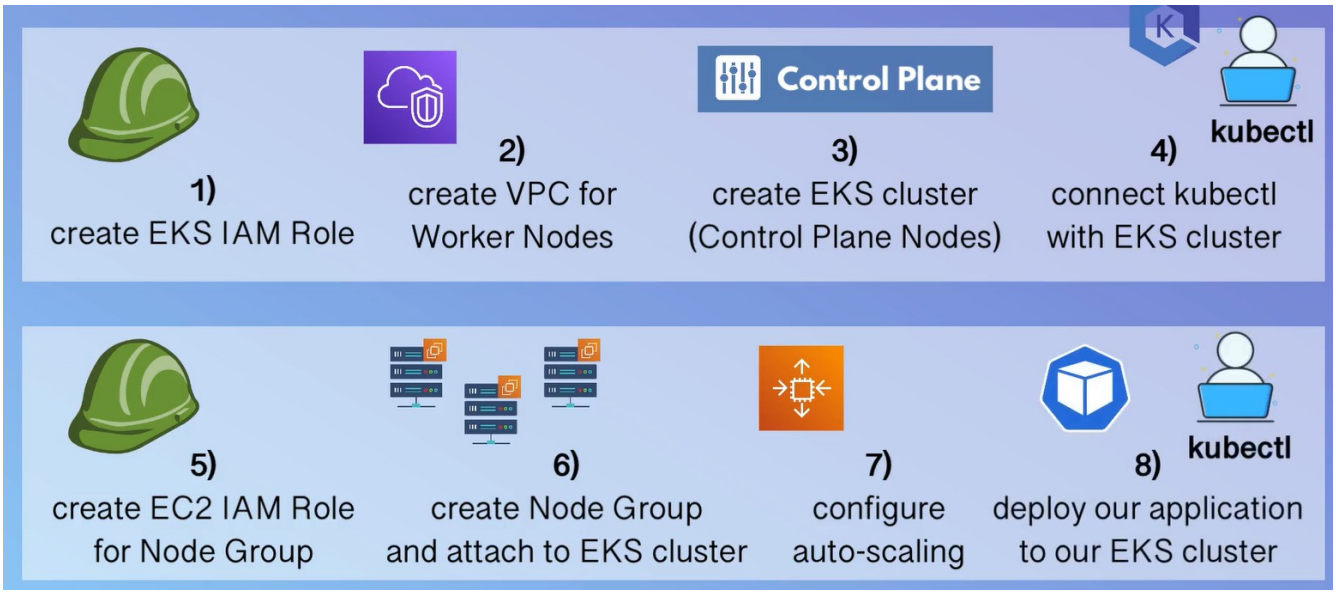
```
aws ecr get-login-password --region eu-
central-1 --profile
AdministratorAccess-488378077264
```

For the script to work, we need to configure some credentials. These can't be hardcoded in the Jenkinsfile, so instead we configure them in *Jenkins Dashboard* > *Manage Jenkins* > *Credentials* > *System* > *Global credentials*:





4 Create Kubernetes Cluster



Create EKS IAM Role
IAM > Roles > Create role
Type: AWS Service
Use case: EKS → EKS Cluster
Name: eks-cluster-role

Create VPC for Worker nodes
Go to AWS CloudFormation for your region. Create a stack.

For the S3 url, use template for private & public VPNs:
https://s3.us-west-2.amazonaws.com/amazon-eks/cloudformation/2020-10-29/amazon-eks-vpc-private-subnets.yaml

Name: eks-worker-node-vpc-stack
Leave everything else as default and submit.

Wait until it is created. Go to Outputs tab, we'll need the info later.

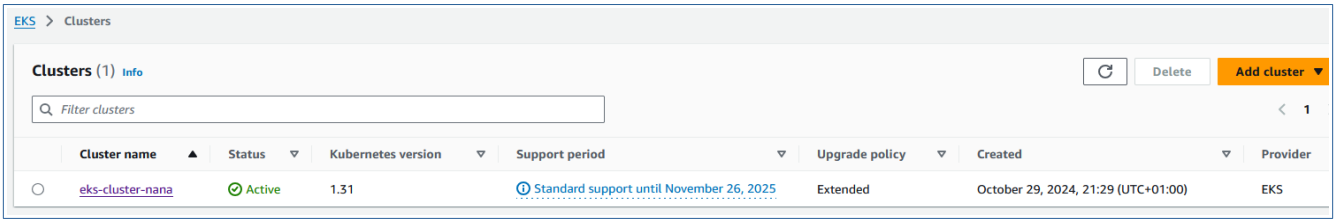
Key	Value	Description
SecurityGroups	sg-0c6e87e83c2e6491e	Security group for the cluster control plane communication with worker nodes
SubnetIds	subnet-0047e578104269496,subnet-09395e65756a6bae5,subnet-05526121a9e473ab2,subnet-08faa6661c1a47b26	Subnets IDs in the VPC
VpcId	vpc-0f84ecf6d073d4025	The VPC Id

Create EKS cluster

In AWS EKS, create a cluster.

- Name: eks-cluster-nana
- Version: 1.31
- Role: eks-cluster-role
- VPC: the one we created for the worker nodes
- Subnets get selected automatically
- Security group: the one that was created before

It takes ~15 min to be ready.



Connect kubectl to the cluster

First we need to attach the relevant permissions for the user logged in the console:
aws sts get-caller-identity # Check current logged in user

Go to IAM > Users > Find the user > Add Permissions > Attach policies directly > Select AmazonEKSClusterPolicy

Run this command to append the new context to the local .kube/config file and also set it as the current context:

```
aws eks update-kubeconfig --name eks-cluster-nana --profile AdministratorAccess-488378077264
```

Copy the local kubeconfig into the Jenkins container:

```
docker cp ".kube/config" 8114c72b2a66:/root/.kube/config
```

Jenkins uses the jenkins user and will need to make modifications inside the .kube folder, so we need to adjust permissions. Run this inside the Jenkins container:

```
docker ps  # Get id of Jenkins container
docker exec -u 0 -it 8114c72b2a66 bash
chown -R jenkins:jenkins $HOME/.kube/
```

Install AWS inside the Jenkins container (search official docs for the correct procedure on Linux).

Login IAM user: `aws configure` (need access key, secret access key, region and format json)

```
aws sts get-caller-identity # Confirm user is logged in
```

During one stage of the pipeline, the 'less' package is require. The Jenkins image is very lightweight and doesn't have it, so we must manually install it:

```
docker ps  # Get id of Jenkins container
docker exec -u 0 -it 8114c72b2a66 bash
apt-get install less
```

Test that Jenkins can correctly connect to the cluster by running these commands inside the container:

```
kubectl get namespaces
kubectl cluster-info
```

Create EC2 IAM Role

IAM > Roles > Create role
Type: AWS Service
Use case: EC2 > EC2
Permission policies:

- AmazonEKSWorkerNodePolicy
- AmazonEC2ContainerRegistryReadOnly
- AmazonEKS_CNI_Policy

Name: eks-node-group-role

Create NodeGroup of worker nodes

Go to the EKS cluster > Compute tab > Add node group

- Name: eks-node-group
- Role: eks-node-group-role
- Compute configuration: Amazon Linux 2 (AL2_x86_64), On demand, t3.small, 20GiB
- Scaling configuration: 2 desired, 2 minimum, 2 maximum.
- Network configuration: 'Configure remote access' enabled
- EC2 Key pair: create a RSA .pem key pair

- Allow remote access from All

Create and confirm with `kubectl get nodes`.

Prepare Jenkins to deploy to K8s

1) Open a console in the Jenkins container and install kubectl:

```
docker ps  # Get container id
docker exec -u 0 -it 8114c72b2a66 bash

curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s
https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/
amd64/kubectl

chmod +x ./kubectl
mv ./kubectl /usr/local/bin/kubectl
```

2) Install AWS IAM Authenticator:

```
curl -Lo aws-iam-authenticator https://github.com/kubernetes-sigs/aws-iam-
authenticator/releases/download/v0.5.9/aws-iam-authenticator_0.5.9_linux_amd64

chmod +x ./aws-iam-authenticator

mv ./aws-iam-authenticator /usr/bin
```

~~3) Create 'config' file in host (because the Jenkins image has no text editor):~~

[Unnecessary, we already prepare a '.kube/config' in the previous section]

```
apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority-data: /etc/kubernetes/pki/ca.crt  Copy it from your own .kube/config
    server: <endpoint-url>  Replace with the 'API server endpoint' from the cluster's page on AWS.
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: aws
  name: aws
current-context: aws
users:
- name: aws
  user:
    exec:
      apiVersion: client.authentication.k8s.io/v1beta1
      command: /usr/bin/aws-iam-authenticator
      args:
        - "token"
        - "-i"
        - <cluster-name> Replace with cluster name
```

~~Inside the container, create folder: `mkdir /var/jenkins_home/.kube`~~

~~Outside the container: `docker cp config 8114c72b2a66:/var/jenkins_home/.kube/`~~

~~4) Create AWS user with minimal permissions and give credentials for Jenkins.~~ → We'll just use an existing user.

4) Give AWS user credentials to Jenkins: On the host computer, if we've logged in to AWS we'll have the file [home]/.aws/credentials. Open it and use the info to create two credentials of type Secret Text:

5) Install envsubst: it is a small tool used within the Jenkinsfile to replace environmental values inside the Kubernetes manifests right before applying them to the cluster. The Jenkins image doesn't include it so we have to add it manually inside the container. envsubst is included in gettext:

```
docker ps  # Get container id
docker exec -u 0 -it 8114c72b2a66 bash
apt-get install -y gettext
```

Configure git

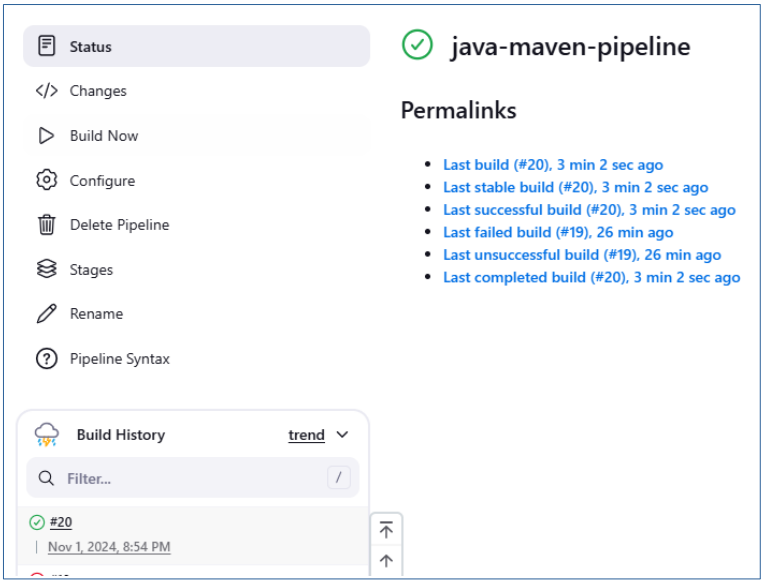
At the end of the pipeline, Jenkins will try to git commit the updated pom.xml with the changed version number. For that, git must have a user end email configured. This is done through Jenkins > Manage Jenkins > System > Git plugin.

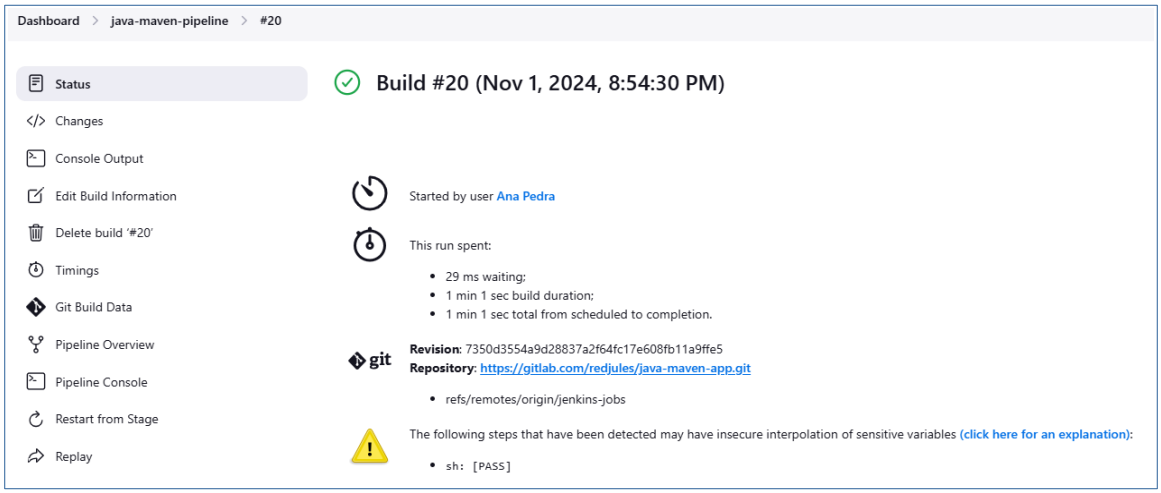
Set the user.name and user.email values. In case of doubt, you can get them from your usual computer with:

```
git config user.name
git config user.email
```

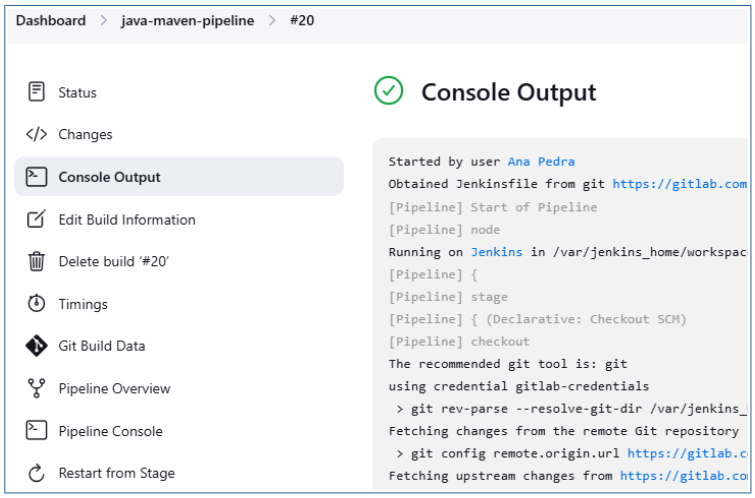
5 Results

We can now click 'Build Now' inside the job. A new build will appear in the Build History. It will take a minute to finish and if everything works well it fill show a green tick:

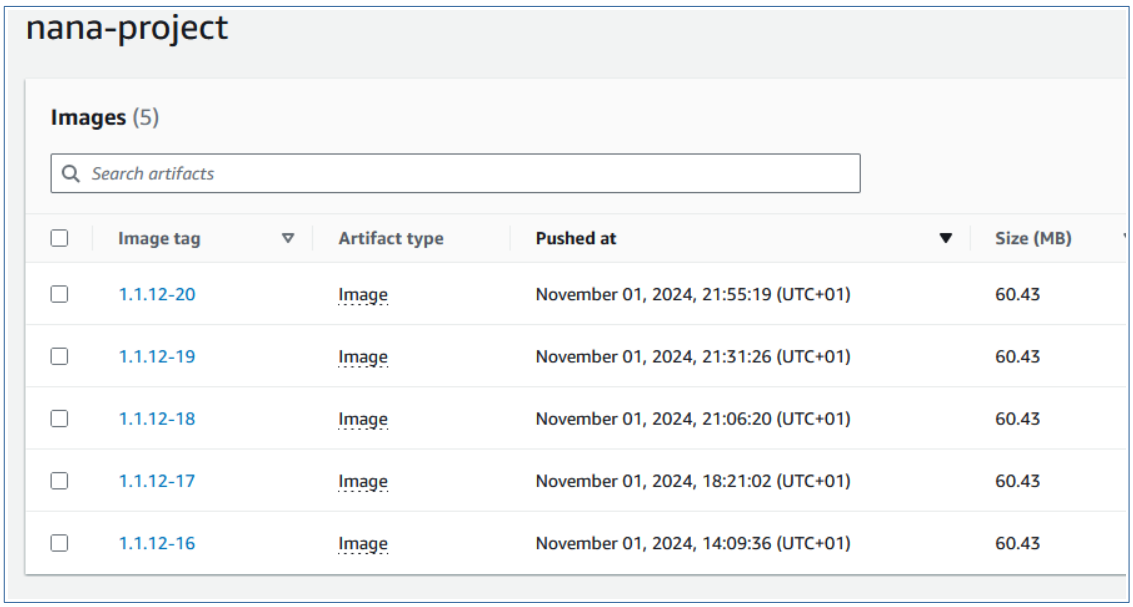




If the job fails, you can click on the specific build in the Build History and check the console output for useful logs.



On every successful execution, the pipeline pushes the created image to the ECR repo:

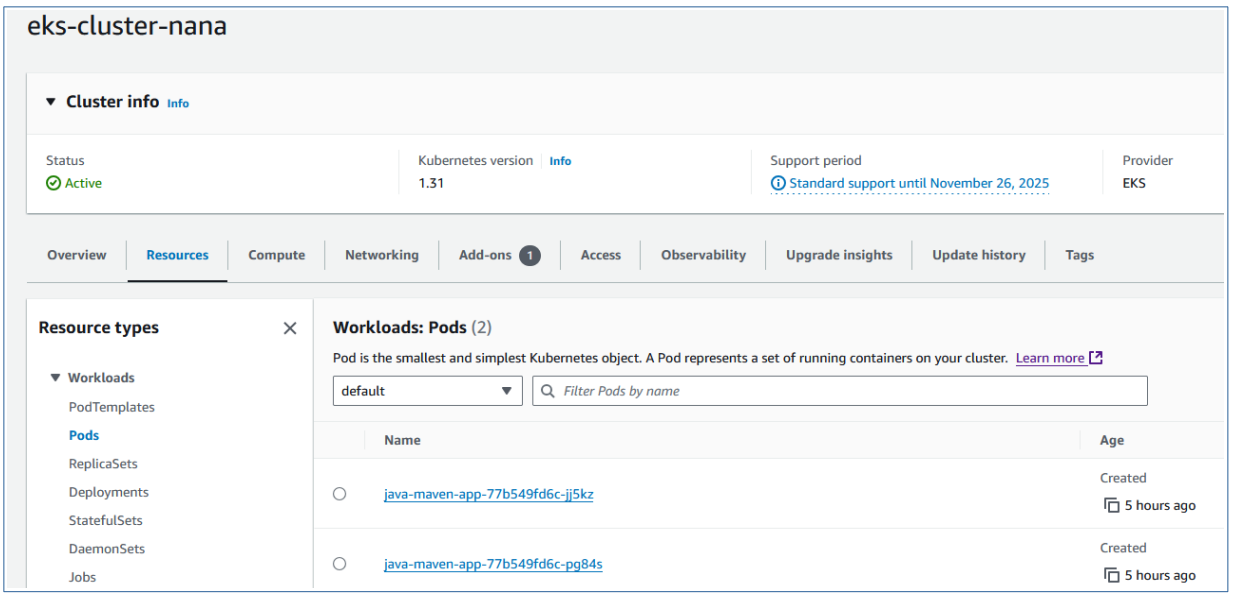


The deployment has been created, which itself creates 2 pods. A service has also been created:

```
root@8114c72b2a66:~# kubectl get deploy
NAME             READY   UP-TO-DATE   AVAILABLE   AGE
java-maven-app   2/2     2            2           57m
```

```
root@8114c72b2a66:~# kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
java-maven-app-ff9759967-sfxxj    1/1     Running   0          9m39s
java-maven-app-ff9759967-zb2rq    1/1     Running   0          9m37s
```

```
root@8114c72b2a66:~# kubectl get svc
NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
java-maven-app   ClusterIP   10.100.63.59    <none>        80/TCP    58m
```



Moreover, the last build stage successfully pushed a new commit to the repository: