SWE 645

Component-based Software Development

Project 2

By:

Abhishek Samuel Daniel – G10393582

Thanuj Pathapati – G01378475

Venkata Krishna Navaneeth Polineni – G01379924

Keerthan Srinivas – G01386121

URL:

AWS URL: http://3.135.228.151/ss645/Student%20Form.html

Kubernetes Application: URL: http://a2fed3adafba14133a33d5beceab7d84-1725638604.us-east-2.elb.amazonaws.com/ss645/

Github Repo Link: https://github.com/ASDPKP/hw2-645

Part 1: Creating GitHub Repository:

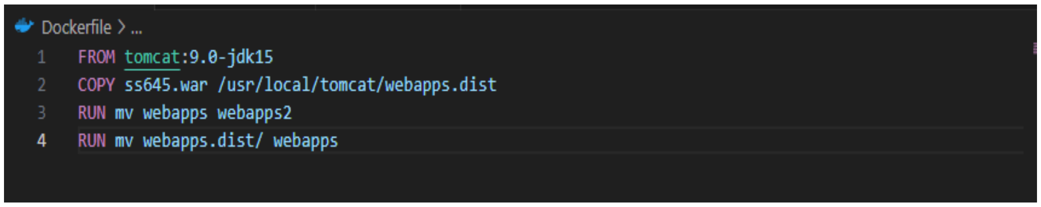
- Pushing the necessary files into Github.
  - Create a github account and make a new repository.
  - Commit and push all the necessary files into the GIT repository using the either the terminal or the UI on GIT
- The next step is to create the .war file.
  - To achieve this, we place all the files created form HW1 – Part2 and open a shell.
  - The current working directory is moved to the folder with all the necessary files using the `$cd <path to the required directory>`
  - The next step is to enter the following command into the shell to compress them into a .war file. `$jar -cvf <war-file-name>.war .`
  - The .war file is created.

Part 2: Installing DOCKER into the system:

- The first step is to open the shell.
- Next check if docker is already installed by running the following command:
  - `$docker –version`
- If Docker is installed, the version number of the current docker installation is displayed.
- If not installed on the machine, an error is displayed telling you that "docker" is not a recognized command. In this case you will have to install docker onto your machine by installing docker desktop using the following link https://docs.docker.com/desktop/install/windows-install/

Part 3: Defining a Dockerfile:

- Next, we create a dockerfile. The content of out docker file must be as follows:



```
Dockerfile > ...
FROM tomcat:9.0-jdk15
COPY ss645.war /usr/local/tomcat/webapps.dist
RUN mv webapps webapps2
RUN mv webapps.dist/ webapps
```

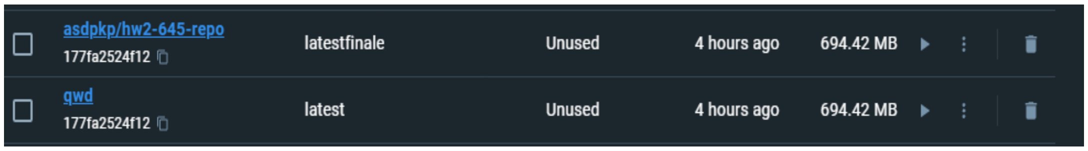
- The from defines the docker image we are going to use in our dockerfile
- The copy command copies the <source file> to <destination path>.
- I ran into an issue when building and running container. The http://localhost:8080 resulted in a "404 Bad Request". After a bit of debugging I understood the supplication which is supposed to be in "webapps" folder have been moved to the "webapps.dist/" folder so in order to overcome this, I first move the contents of my "webapps" folder to a new folder called "webapps2". And then moved all the files in my "webapps.dist" folder to the "webapps" directory. This was done using the last two commands seen the snippet above.

Part 4: Building the Image:

- After defining the Dockerfile (It should be saved exactly as "Dockerfile"). I built the image using the following command

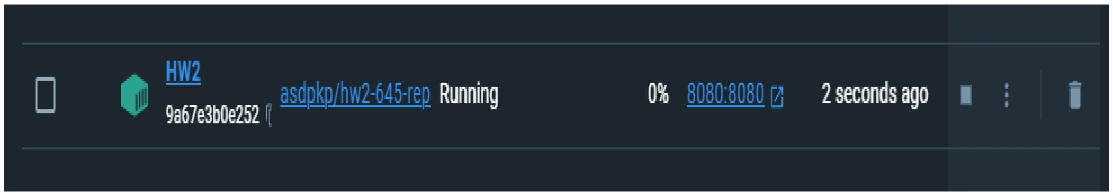
```
PS C:\Users\endso\Desktop\StudentForm> docker build -l <docker-image>
```

- The above command creates the docker image on the docker desktop.
- You can see the image you created on there.



```
asdpkp/hw2-645-repo   latestfinale   Unused   4 hours ago   694.42 MB
177fa2524f12
qwd                   latest         Unused   4 hours ago   694.42 MB
177fa2524f12
```

- To run the image, click on the triangle next to the dots on the right side of the screen.
- Then enter the port number, give a name to the container, and click next.
- The containers windows should then look like this.



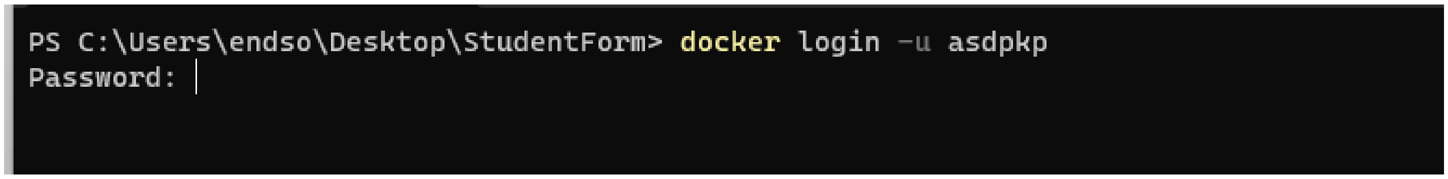
```
HW2            asdpkp/hw2-645-rep   Running   0%   8080:8080   2 seconds ago
9a67e3b0e252
```

-

- Then open a browser window and type the following to launch the image: http://localhost:8080/<war-file-name>

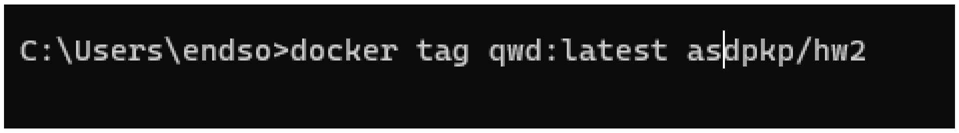
Part 5: Tagging and pushing the docker image to docker hub.

- We need to first create an account on dockerhub and then a docker repository. Using those credentials login into dockerhub using the command.



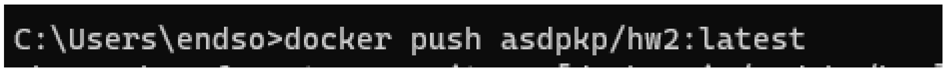
```
PS C:\Users\endso\Desktop\StudentForm> docker login -u asdpkp
Password:
```

- We then tag the image using our images using <username>/<docker-repository>. The command for tagging the image is



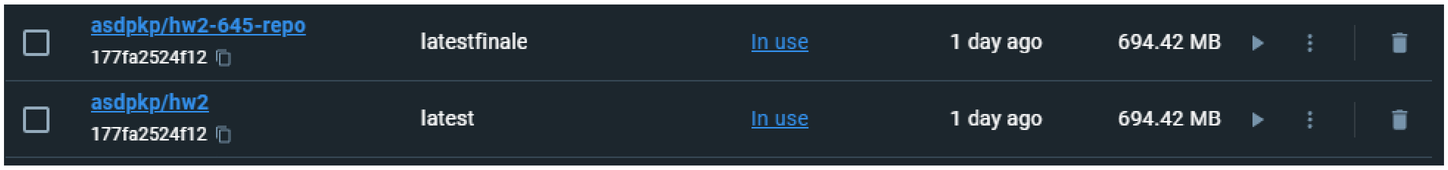
```
C:\Users\endso>docker tag qwd:latest asdpkp/hw2
```

- After tagging the image you can push the image into your repository using the push command as follows:



```
C:\Users\endso>docker push asdpkp/hw2:latest
```

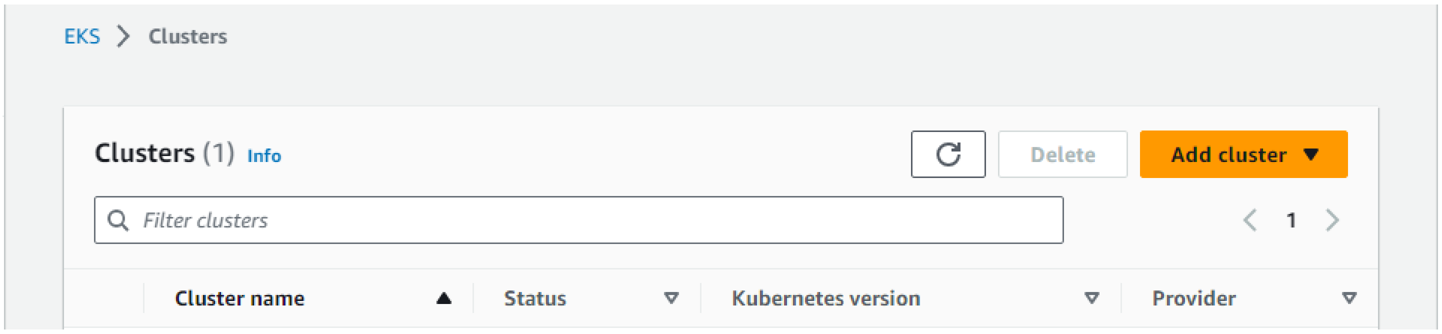
- You can check whether your docker image had been pushed into your repository by logging into your docker hub account.



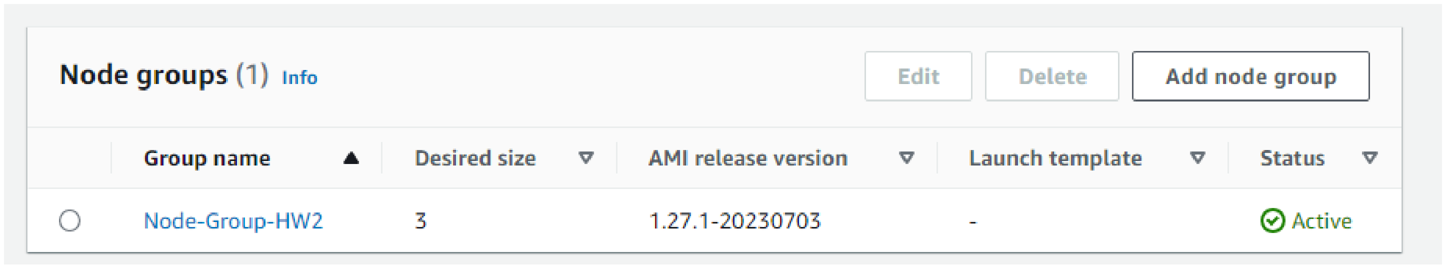
asdpkp/hw2-645-repo 177fa2524f12	latestfinale	In use	1 day ago	694.42 MB	
asdpkp/hw2 177fa2524f12	latest	In use	1 day ago	694.42 MB	

- **Part 6: Creating the Kubernetes cluster using Amazon EKS**:

  - I logged into my AWS account I used for HW1.
  - Before creating a cluster, we must create a new EKS role that manages the cluster. To do this we go to the search box and type IAM. After the IAM page open we click on roles and then click on create a new role.
  - In the IAM role creation page, we choose AWS Simple Service then at the bottom of the page there is a dropdown box which says, “Choose a service to view use case”. Here we choose the EKS option and then select EKS-cluster next.
  - On the next page we assign permissions to the role. On this page selecting the EKS – Cluster on the pervious page automatically chooses the “**AmazonEKSClusterPolicy**”. Click on next and name your role and finally click create.
  - Since we need three nodes in our cluster, we also need to create a new role that handles the nodes in the cluster. We repeat step 2 and on the creation page we leave AWS Simple Service as the default option and choose EC2 under the Use Case section.
  - On the policies page we need to add three policies namely **AmazonEKSWorkerNodePolicy, AmazonEC2ContainerRegistryReadOnly, AmazonEKS_CNI_Policy.** On the next page we name our role and click create.
  - Two new roles are created one for our Cluster and the other for our Nodes.
  - In the search box I entered EKS. This opens the Elastic Kubernetes Service hosted by AWS.
  - This opens the EKS dashboard. Here we click on Add Cluster to create a new cluster.



  - The cluster creation page opens. Here, we first give a name to our cluster. Here you can see that the EKS role we create before is automatically filled into the Cluster service role section. We leave on the other options on default and finally create.
  - If you think that cluster is taking abnormally long to get created. Don’t panic as this normal.
  - After the node has been created click on the node and select the “Compute” sub-section.
  - Here we select the Add node group section option to create the node group.

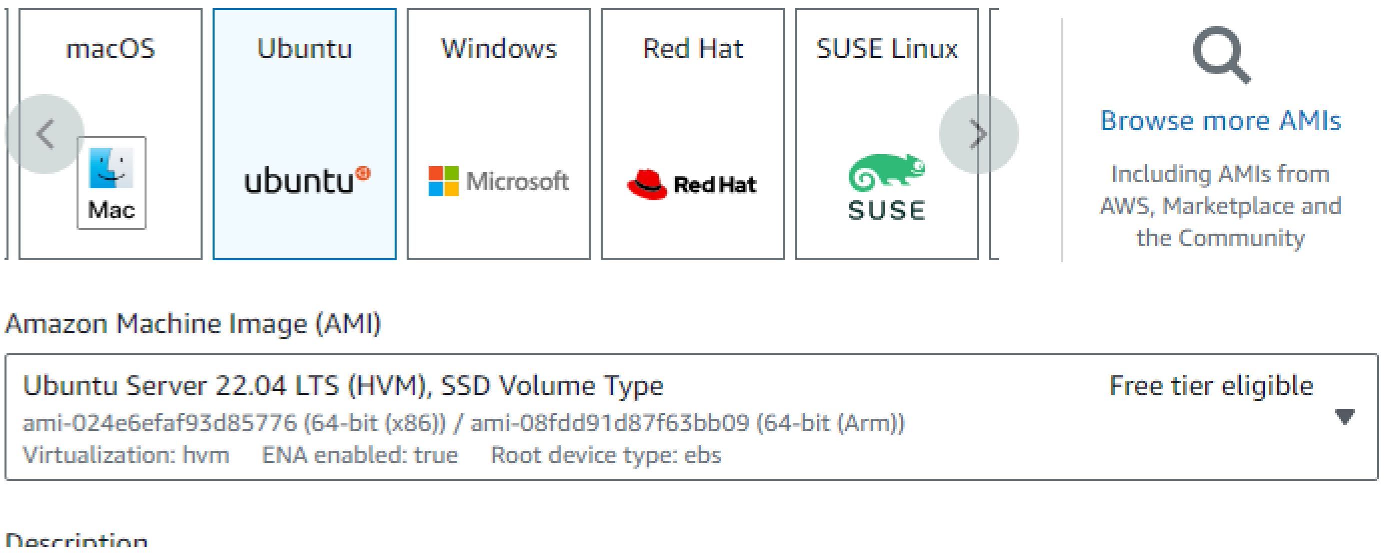


Group name	Desired size	AMI release version	Launch template	Status
Node-Group-HW2	3	1.27.1-20230703	-	Active

  - Name your node group and here too we see that the Node IAM role is automatically filled with a role that we created before.

- On the “Set compute and scaling configuration” page we need to set the minimum and maximum nodes to 3. Leave all the other setting on default options and finally select create. Node group creation also takes a few minutes.
- To check if the nodes have been created go to the EC2 dashboard and you will see three nameless instances. These are the nodes.
- With this we finished the creation of our clusters.

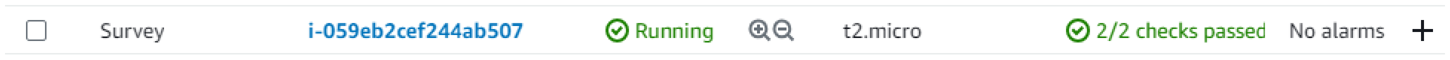
- **<u>Part 7: Creating an instance for Jenkins</u>**:
    - We need to create a new sentence. First search for EC2 in the search box.
    - Next, we need to click on Launch Instance and select on Launch. Give a name to the instance. Next select Ubuntu in the Application and OS Images section.
    - In the instances we select t2.medium (I chose t2.meduim as it can handle 3 nodes much better tha t2.micro).



Amazon Machine Image (AMI)

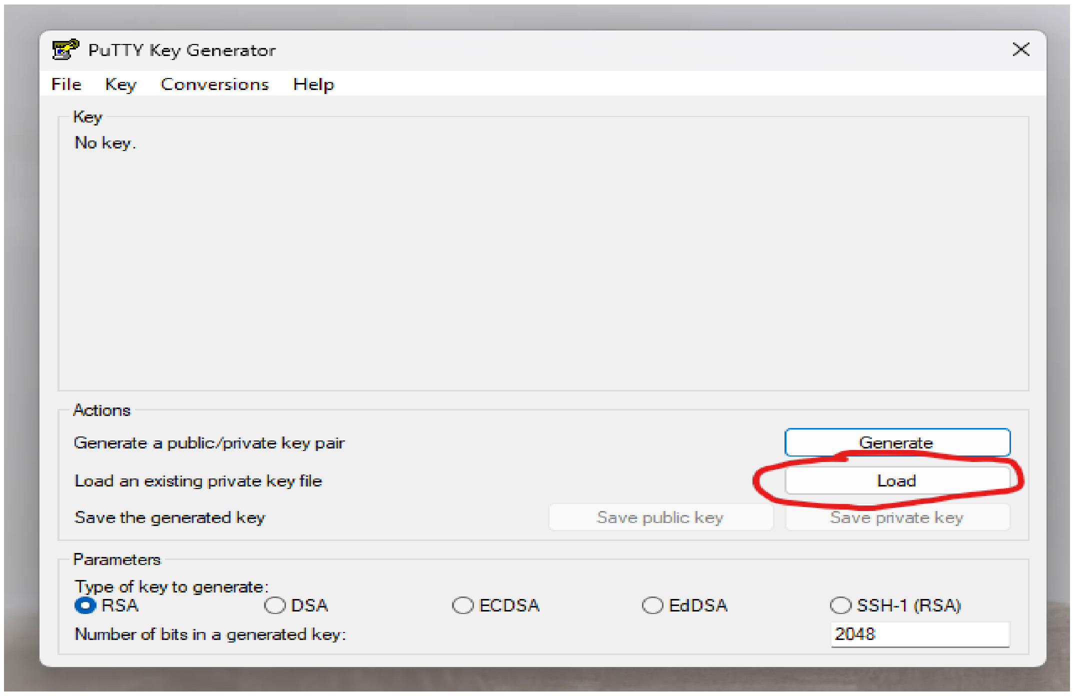
Ubuntu Server 22.04 LTS (HVM), SSD Volume Type — Free tier eligible
ami-024e6efaf93d85776 (64-bit (x86)) / ami-08fdd91d87f63bb09 (64-bit (Arm))
Virtualization: hvm ENA enabled: true Root device type: ebs

- Next, we need to create key pair to access the instance. If using PuTTY select the .ppk version else the .pem version. Save them as a CSV file.
- On the next section we need to edit the network setting to allow the instance to use the ports 22, 443, 80 and 8080. To do this click on edit. Then we click on add security group rule and in the type select HTTP, the port 80 is automatically set. Similarly for 22 we select SSH and for 443 we select source as HTTPS. For enabling port 8080 we set the Type as Custom TCP and enter the port manually. For all the above ports set the Source Type to Anywhere.
- Leave the rest setting on default and press launch. Your instance is created. It should look something like this.

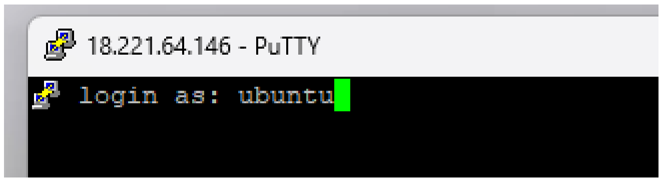


☐	Survey	i-059eb2cef244ab507	Running		t2.micro	2/2 checks passed	No alarms

- **Part 8: Setting up Jenkins:**
  - Remember the .pem/.ppk file we created we earlier? Its time to use them.
  - If like me you are on a Windows, we first install PuTTY. The steps for installing PuTTY can be found here: https://www.ssh.com/academy/ssh/putty/windows/install
  - On your system open PuTTY gen and click on load. If

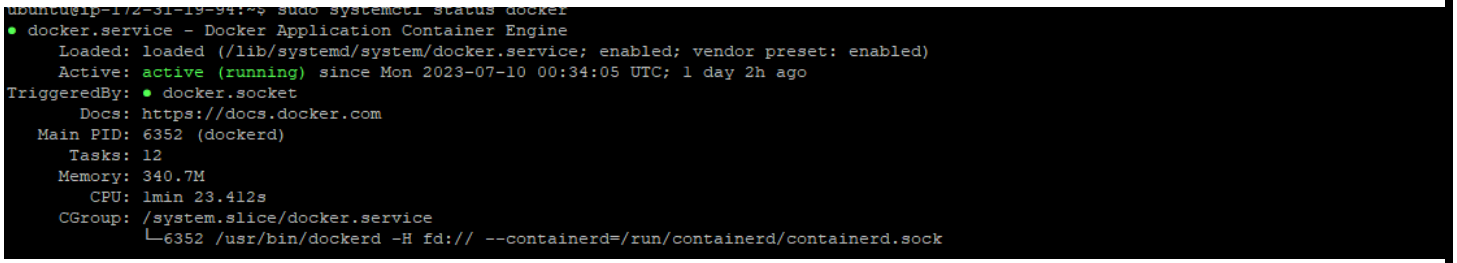


  - Go to the directory where you saved the .pem file. Select the pem file and click on Save Private key. This converts your .pem file into a .ppk file. You can use this to login to the EC2 instance.
  - Next, we open PuTTY. In Host Name field we paste the IPV4 address of our instance. Leave the port as it is and on the left-hand side under the Connection category we expand as follows SSH->Auth->Credentials. In the field "Private key file for authentication" we place our .pem file.
  - Go back to the Session page and select Open. This opens the terminal for our Ubuntu Instance. In the "Login as" window we enter "ubuntu" as it is the server we are logging into.



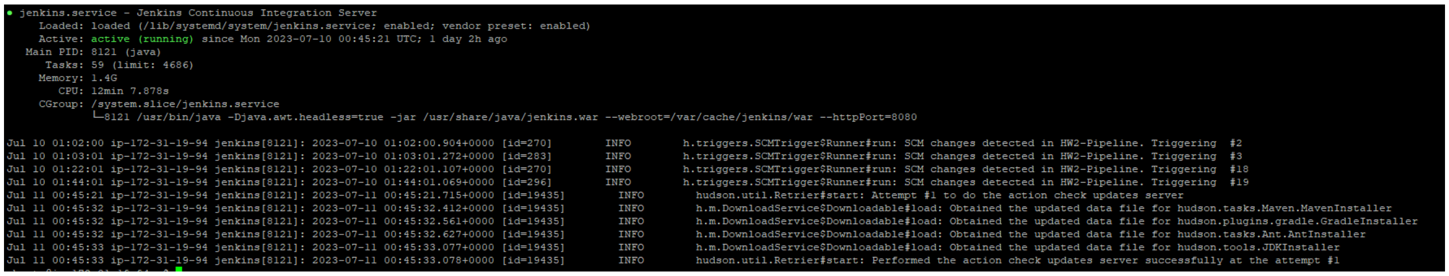
```
login as: ubuntu
```

- First, we need to update the instance by using $ sudo apt-get update.
- Next, we need to install java on the server and to do this we use $ sudo apt install default-java.
- The next part is to setup AWS CLI on the server to do this we first need to install the unzip module. We can do this using $ sudo apt install unzip
- After installing unzip, we proceed with installing aws cli. To do this we enter the following commands into our terminal:
  - curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
  - unzip awscliv2.zip
  - sudo ./aws/install
- After installing CLI we need to install docker on our machine. To do this we simply type in $ sudo apt install docker.io
- Check whether docker is running by typing in $ sudo systemctl status docker. If not running, we can start docker using $sudo systemctl start docker and $sudo systemctl enable docker.



```
ubuntu@ip-172-31-19-94:~$ sudo systemctl status docker
● docker.service - Docker Application Container Engine
     Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
     Active: active (running) since Mon 2023-07-10 00:34:05 UTC; 1 day 2h ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
   Main PID: 6352 (dockerd)
      Tasks: 12
     Memory: 340.7M
        CPU: 1min 23.412s
     CGroup: /system.slice/docker.service
             └─6352 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
```

- Next, we need to install Jenkins. To do this we need to give the following commands to the instance.
  - 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
  - sudo apt-get update
  - sudo apt-get install jenkins

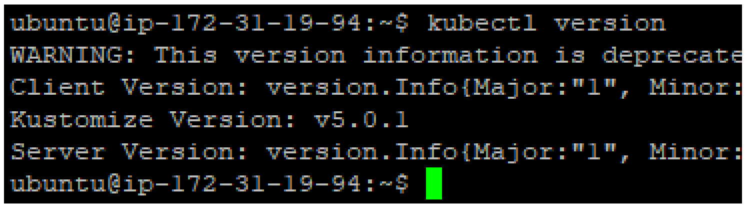


```
● jenkins.service - Jenkins Continuous Integration Server
     Loaded: loaded (/lib/systemd/system/jenkins.service; enabled; vendor preset: enabled)
     Active: active (running) since Mon 2023-07-10 00:45:21 UTC; 1 day 2h ago
   Main PID: 8121 (java)
      Tasks: 59 (limit: 4686)
     Memory: 1.4G
        CPU: 12min 7.878s
     CGroup: /system.slice/jenkins.service
             └─8121 /usr/bin/java -Djava.awt.headless=true -jar /usr/share/java/jenkins.war --webroot=/var/cache/jenkins/war --httpPort=8080

Jul 10 01:02:00 ip-172-31-19-94 jenkins[8121]: 2023-07-10 01:02:00.904+0000 [id=270]       INFO       h.triggers.SCMTrigger$Runner#run: SCM changes detected in HW2-Pipeline. Triggering  #2
Jul 10 01:03:01 ip-172-31-19-94 jenkins[8121]: 2023-07-10 01:03:01.272+0000 [id=283]       INFO       h.triggers.SCMTrigger$Runner#run: SCM changes detected in HW2-Pipeline. Triggering  #3
Jul 10 01:22:01 ip-172-31-19-94 jenkins[8121]: 2023-07-10 01:22:01.107+0000 [id=270]       INFO       h.triggers.SCMTrigger$Runner#run: SCM changes detected in HW2-Pipeline. Triggering  #18
Jul 10 01:44:01 ip-172-31-19-94 jenkins[8121]: 2023-07-10 01:44:01.069+0000 [id=296]       INFO       h.triggers.SCMTrigger$Runner#run: SCM changes detected in HW2-Pipeline. Triggering  #19
Jul 11 00:45:21 ip-172-31-19-94 jenkins[8121]: 2023-07-11 00:45:21.715+0000 [id=19435]       INFO       hudson.util.Retrier#start: Attempt #1 to do the action check updates server
Jul 11 00:45:32 ip-172-31-19-94 jenkins[8121]: 2023-07-11 00:45:32.412+0000 [id=19435]       INFO       h.m.DownloadService$Downloadable#load: Obtained the updated data file for hudson.tasks.Maven.MavenInstaller
Jul 11 00:45:32 ip-172-31-19-94 jenkins[8121]: 2023-07-11 00:45:32.561+0000 [id=19435]       INFO       h.m.DownloadService$Downloadable#load: Obtained the updated data file for hudson.plugins.gradle.GradleInstaller
Jul 11 00:45:32 ip-172-31-19-94 jenkins[8121]: 2023-07-11 00:45:32.627+0000 [id=19435]       INFO       h.m.DownloadService$Downloadable#load: Obtained the updated data file for hudson.tasks.Ant.AntInstaller
Jul 11 00:45:33 ip-172-31-19-94 jenkins[8121]: 2023-07-11 00:45:33.077+0000 [id=19435]       INFO       h.m.DownloadService$Downloadable#load: Obtained the updated data file for hudson.tools.JDKInstaller
Jul 11 00:45:33 ip-172-31-19-94 jenkins[8121]: 2023-07-11 00:45:33.078+0000 [id=19435]       INFO       hudson.util.Retrier#start: Performed the action check updates server successfully at the attempt #1
```

- Next, we need to install kubectl on the instance. To do this we first need to install snapd. This is done using the following command: $ sudo apt install snapd

- Followed by $ sudo snap install kubectl –classic.
- Check whether kubectl is installed using the command: $kubectl version. This should return a version number for kubctl.



```
ubuntu@ip-172-31-19-94:~$ kubectl version
WARNING: This version information is deprecate
Client Version: version.Info{Major:"1", Minor:
Kustomize Version: v5.0.1
Server Version: version.Info{Major:"1", Minor:
ubuntu@ip-172-31-19-94:~$
```

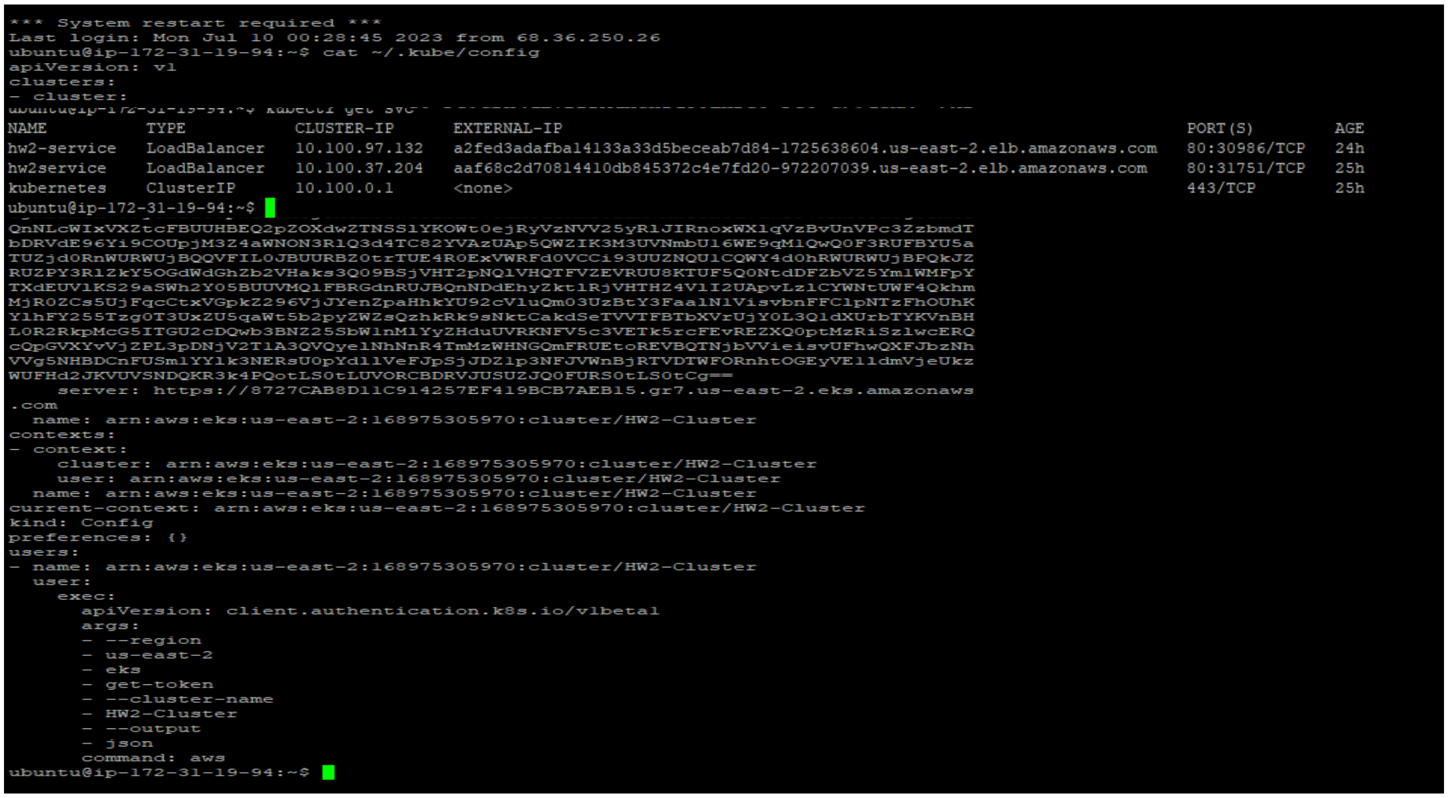
- In order to check if the current instance is created by the same user as that who created the cluster we type in the following command: $ aws sts get-caller-identity.
- The output should look similar to:

```
{
    "UserId": "AIDASAMPLEUSERID",
    "Account": "123456789012",
    "Arn": "arn:aws:iam::123456789012:user/DevAdmin"
}
```

- If the UserID matches that of the one who created the cluster, we can continue our setup. If not, we need to configure our instance to login to the user who created the cluster.
- To configure the current instance, we use: $aws configure.
- This gives the following output in a sequential order:

```
$ aws configure
AWS Access Key ID [None]: accesskey
AWS Secret Access Key [None]: secretkey
Default region name [None]: us-west-2
Default output format [None]:
```

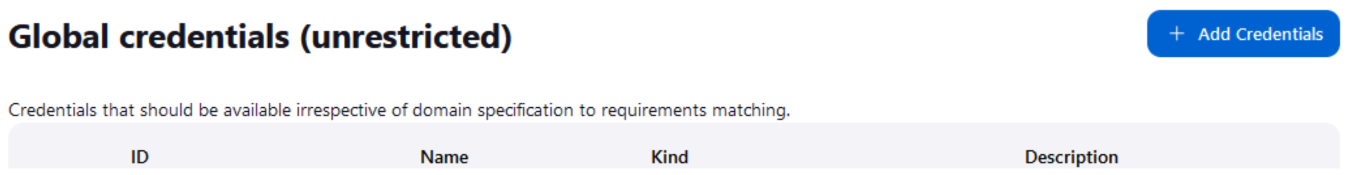
- We need to first the Access Key ID, then the Secret Access Key, the us region in which the aws instance was created and leave the default output format as it is.
- After configuring the instance, we need to check if the instance has the kubeconfig file for the cluster we created. To do this we type: $ sudo cat ~/.kube.config.
- If the output for the above command is an error telling you that a kubeconfig file is not present in the current instance. We need to import the kubeconfig file for the cluster.
- To do this we type: $ aws eks update-kubeconfig –region <region-name> --name <name-of-cluster>.
- To check if the kubeconfig file has been imported you can enter: $ sudo cat ~/.kube/cofig.



```
*** System restart required ***
Last login: Mon Jul 10 00:28:45 2023 from 68.36.250.26
ubuntu@ip-172-31-19-94:~$ cat ~/.kube/config
apiVersion: v1
clusters:
- cluster:
NAME          TYPE           CLUSTER-IP      EXTERNAL-IP                                                               PORT(S)        AGE
hw2-service   LoadBalancer   10.100.97.132   a2fed3adafba14133a33d5beceab7d84-1725638604.us-east-2.elb.amazonaws.com   80:30986/TCP   24h
hw2service    LoadBalancer   10.100.37.204   aaf68c2d70814410db845372c4e7fd20-972207039.us-east-2.elb.amazonaws.com    80:31751/TCP   25h
kubernetes    ClusterIP      10.100.0.1      <none>                                                                    443/TCP        25h
ubuntu@ip-172-31-19-94:~$
QnNLcWIxVXZtcFBUUHBEQ2pZOXdwZTNSSlYKOWt0ejRyVzNVV25yRlJIRnoxWXlqVzBvUnVPc3ZzbmdT
bDRVdE96Yi9COUpjM3Z4aWNON3RlQ3d4TC82YVAzUAp5QWZIK3M3UVNmbU16WE9qMlQwQ0F3RUFBYU5a
TUZjd0RnWURWUjBQQVFIL0JBUURBZ0trTUE4R0ExVWRFd0VCCi93UUZNQU1CQWY4d0hRWURWUjBPQkJZ
RUZPY3RlZkY5OGdWdGhZb2VHaks3Q09BSjVHT2pNQlVHQTFVZEVRUU8KTUF5Q0NtdDFZbVZ5YmlWMFpY
TXdEUVlKS29aSWh2Y05BUUVMQlFBRGdnRUJBQnNDdEhyZktlRjVHTHZ4VlI2UApvLzlCYWNtUWF4Qkhm
MjR0ZCs5UjFqcCtxVGpkZ296VjJYenZpaHhkYU92cVluQm03UzBtY3FaalN1VisvbnFFClpNTzFhOUhK
YlhFY255Tzg0T3UxZU5qaWt5b2pyZWZsQzhkRk9sNktCakdSeTVVTFBTbXVrUjY0L3QldXUrbTYKVnBH
L0R2RkpMcG5ITGU2cDQwb3BNZ25SbWlnMlYyZHduUVRKNFV5c3VETk5rcFEvREZXQ0ptMzRiSzlwcERQ
cQpGVXYvVjZPL3pDNjV2T1A3QVQyelNhNnR4TmMzWHNGQmFRUEtoREVBQTNjbVVieisvUFhwQXFJbzNh
VVg5NHBDCnFUSmlYYlk3NERsU0pYdllVeFJpSjJDZlp3NFJVWnBjRTVDTWFORnhtOGEyVEl1dmVjeUkz
WUFHd2JKVUVSNDQKR3k4PQotLS0tLUVORCBDRVJUSUZJQ0FURS0tLS0tCg==
    server: https://8727CAB8D11C914257EF419BCB7AEB15.gr7.us-east-2.eks.amazonaws
.com
  name: arn:aws:eks:us-east-2:168975305970:cluster/HW2-Cluster
contexts:
- context:
    cluster: arn:aws:eks:us-east-2:168975305970:cluster/HW2-Cluster
    user: arn:aws:eks:us-east-2:168975305970:cluster/HW2-Cluster
  name: arn:aws:eks:us-east-2:168975305970:cluster/HW2-Cluster
current-context: arn:aws:eks:us-east-2:168975305970:cluster/HW2-Cluster
kind: Config
preferences: {}
users:
- name: arn:aws:eks:us-east-2:168975305970:cluster/HW2-Cluster
  user:
    exec:
      apiVersion: client.authentication.k8s.io/v1beta1
      args:
      - --region
      - us-east-2
      - eks
      - get-token
      - --cluster-name
      - HW2-Cluster
      - --output
      - json
      command: aws
ubuntu@ip-172-31-19-94:~$
```

- Or also type in $kubectl get svc. The output should include your cluster.
- Finally, the Jenkins setup is complete and now you can open up Jenkins on your system using the following url: http://<EC2-instance-IP>:8080. This should take you to the Jenkins setup page.
- To access the password for the initial login we need to enter the following command on the terminal.
    - sudo cat /var/lib/Jenkins/secrets/initialAdminPassword/
- The output should show your password. Copy and paste the paste the password on the password field on your initial Jenkins page.
- Select the install the recommened plugins and click next.
- Enter the desired credentials for your Jenkins account. After this step Jenkins should show you the url for logging into your Jenkins account.
- Copy this and enter your Jenkins dashboard.
- First, we need to install the necessary plugins for Jenkins to communicate with aws and docker.
- To do this firct click on “Manage Jenkins” -> “Plugins” -> “Available plugins”.
- In the search type aws-pipeline. Only one plugin should pop-up. Install this.
- Next, we need to install the docker-workflow plugin. We need to type the name of the plugin in the search bar and install the plugin.
- Restart you Jenkins by pasting the url followed by a “/restart”. This restarts your Jenkisn instance and prompts you enter your Jenkins credentials.

- Next, we create the credentials that Jenkins will use to access your docker. To do this we click on the arrow next to your account name and select credentials. Next, click on “(global)” and click on “Add Credentials”.

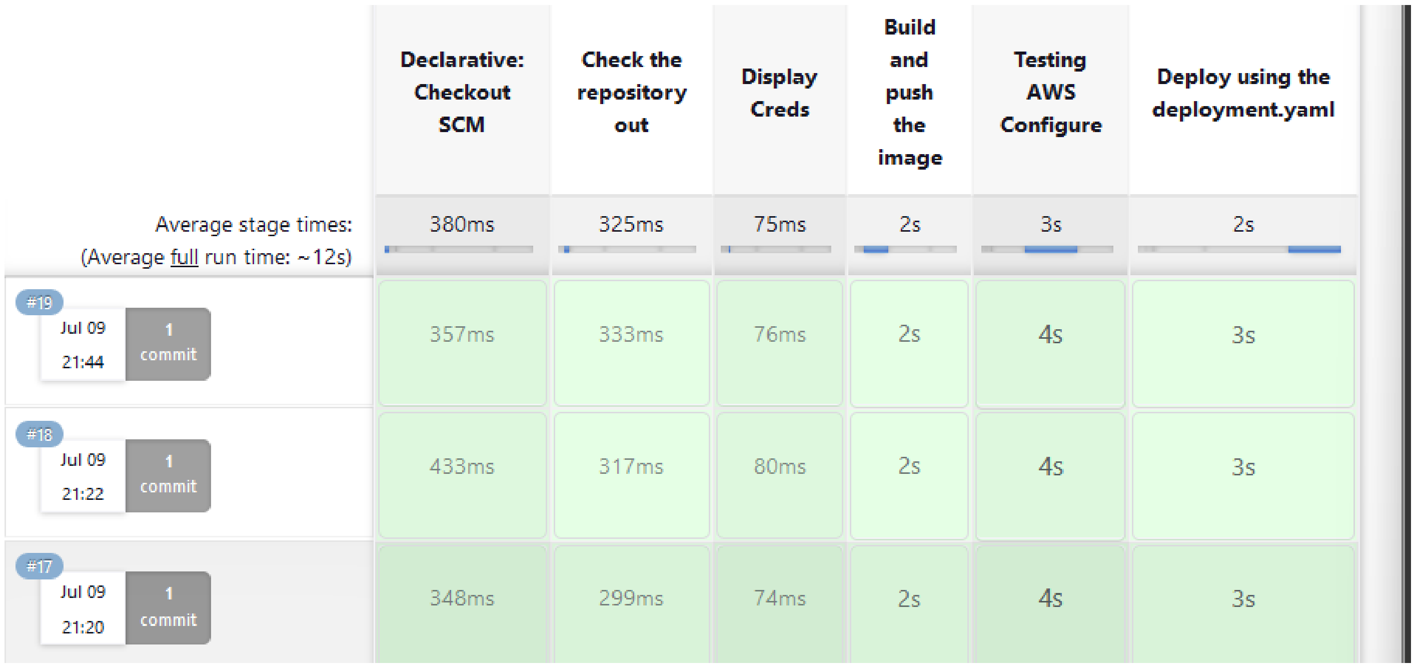


Global credentials (unrestricted)

Credentials that should be available irrespective of domain specification to requirements matching.

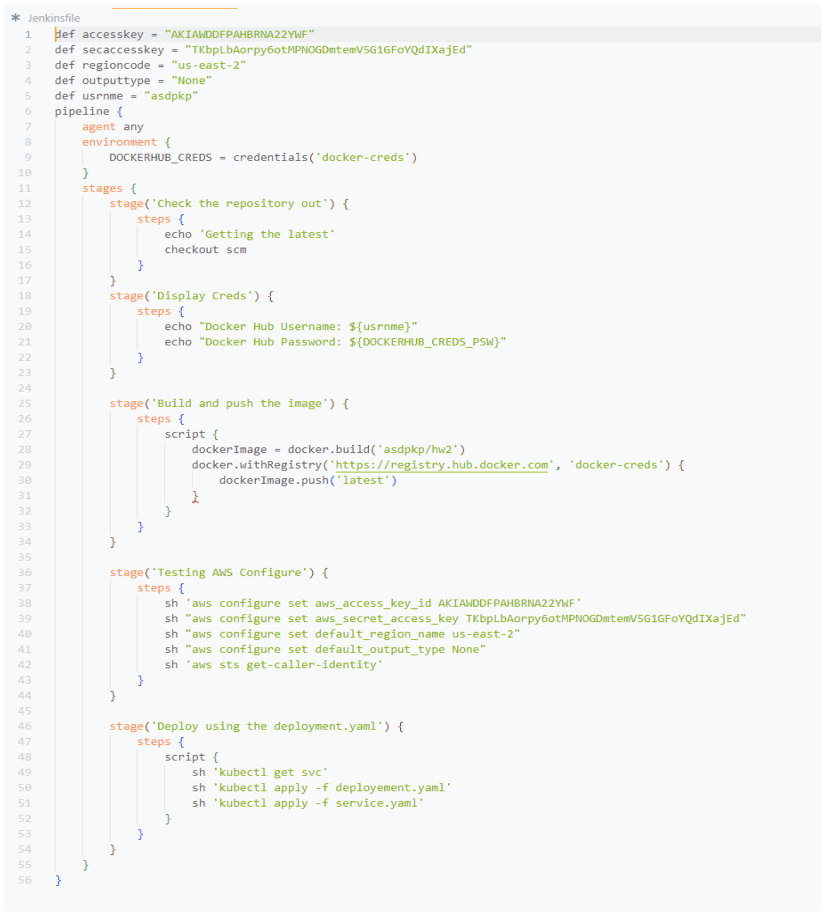
ID	Name	Kind	Description

- In the dropdown select “Username and Password” in the scope select “Global”. Enter your docker username and password and give a name to your credential set. And click create.
- Next on the dashboard select NewItem.
- Give a name to your pipeline and select Pipeline from the below options and click “Ok”.
- From the options under build triggers choose “Poll SCM” and in schedule type in “* * * * *”. This ensure that the docker push is updated every 1 minute.
- In the Pipeline section choose “Pipeline script from SCM” and in the SCM dropdown choose “GIT”.
- In the Repository URL paste your github repository URL and in credentials drop down choose the dockerhub credentials.
- And under the “branch specifier” change /master to /main.
- Finally click on create. This should create your pipeline and build your first pipeline. To check if your pipeline is working make a small change to any file in your github repository. As soon as the pipeline detects a change in your file, it should start an automatic build, which you can see on your stage view.



	Declarative: Checkout SCM	Check the repository out	Display Creds	Build and push the image	Testing AWS Configure	Deploy using the deployment.yaml
Average stage times: (Average full run time: ~12s)	380ms	325ms	75ms	2s	3s	2s
#19 Jul 09 21:44 1 commit	357ms	333ms	76ms	2s	4s	3s
#18 Jul 09 21:22 1 commit	433ms	317ms	80ms	2s	4s	3s
#17 Jul 09 21:20 1 commit	348ms	299ms	74ms	2s	4s	3s

This is what my Jenkins file looks like:



```
def accesskey = "AKIAWDDFPAHBRNA22YWF"
def secaccesskey = "TKbpLbAorpy6otMPNOGDmtemV5G1GFoYQdIXajEd"
def regioncode = "us-east-2"
def outputtype = "None"
def usrnme = "asdpkp"
pipeline {
    agent any
    environment {
        DOCKERHUB_CREDS = credentials('docker-creds')
    }
    stages {
        stage('Check the repository out') {
            steps {
                echo 'Getting the latest'
                checkout scm
            }
        }
        stage('Display Creds') {
            steps {
                echo "Docker Hub Username: ${usrnme}"
                echo "Docker Hub Password: ${DOCKERHUB_CREDS_PSW}"
            }
        }

        stage('Build and push the image') {
            steps {
                script {
                    dockerImage = docker.build('asdpkp/hw2')
                    docker.withRegistry('https://registry.hub.docker.com', 'docker-creds') {
                        dockerImage.push('latest')
                    }
                }
            }
        }

        stage('Testing AWS Configure') {
            steps {
                sh 'aws configure set aws_access_key_id AKIAWDDFPAHBRNA22YWF'
                sh "aws configure set aws_secret_access_key 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"
                sh "aws configure set default_region_name us-east-2"
                sh "aws configure set default_output_type None"
                sh 'aws sts get-caller-identity'
            }
        }

        stage('Deploy using the deployment.yaml') {
            steps {
                script {
                    sh 'kubectl get svc'
                    sh 'kubectl apply -f deployement.yaml'
                    sh 'kubectl apply -f service.yaml'
                }
            }
        }
    }
}
```

- I first initialized my variables that I will use in the below statements.
- The first stage is used to assign the docker credentials that we received from the pipeline into a local variable.
- The next stage is used to login into docker.
- The following stage builds and pushes the updated docker image into your repository.
- My next stage is a precautionary measure in case the system throws and authentication error. The stage configures the current aws directory using aws configure and passing the aws account access keys and regions.
- Finally, I will deploy using deployment and service .yaml files.

What results to expect?

1) In docker the image should you pushed must appear in your repository. Running the image container should display your web-application.
2) After creating and deploying your cluster, three instances must be created on your EC2 dashboard.
3) After deploying your pipeline, the build must run without any errors and any changes made to your github repository must trigger a new build.
4) After the initial build and load balancer must be created which can be seen in the EC2 page of AWS.
5) Clicking on the aws link at the top should open the student survey form.
6) The dns of the loadn balancer followed by ss645 must open the web application.

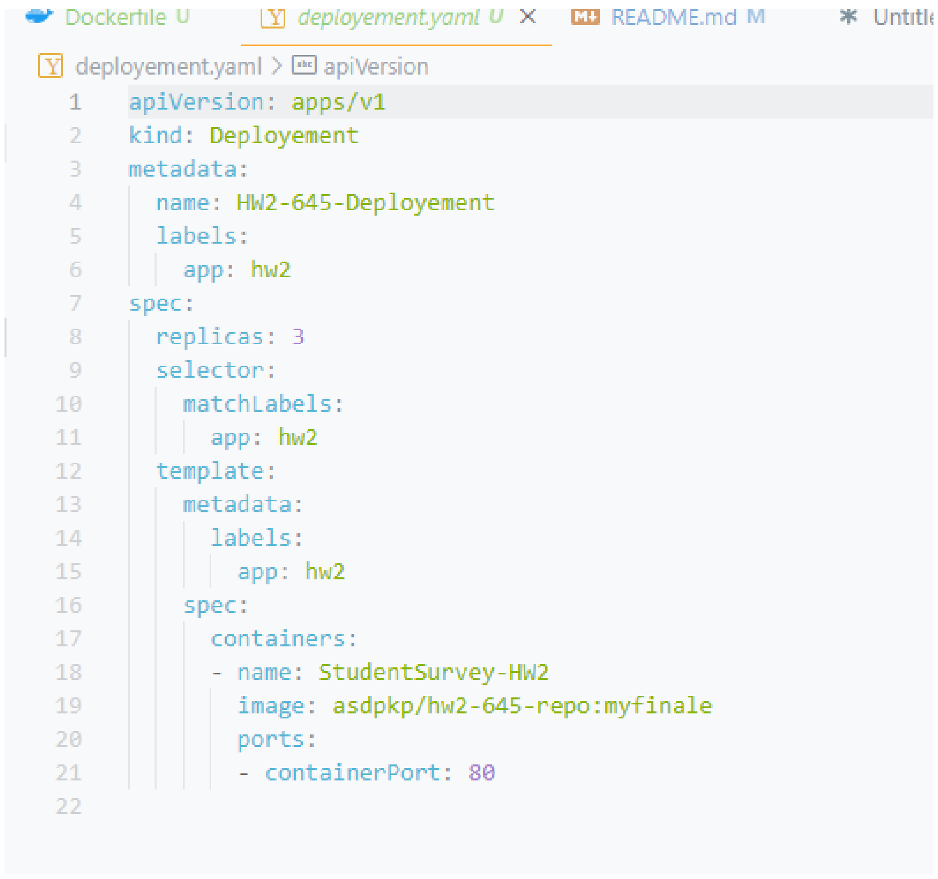
Who did what?

Thanuj Pathapati: Installation of Docker, Creation of Dockerfile, creation of war file, deploying the file to the EC2 instance and building and pushing the image into docker hub.

Navaneeth Krishna: Creation of EKS cluster, EKS role, node group and node group role.

Abhishek Samuel: Setup of Jenkins on instance, Creation of Jenkinsfile, Creation of deployment and service yaml files, Creating and deploying the pipeline.

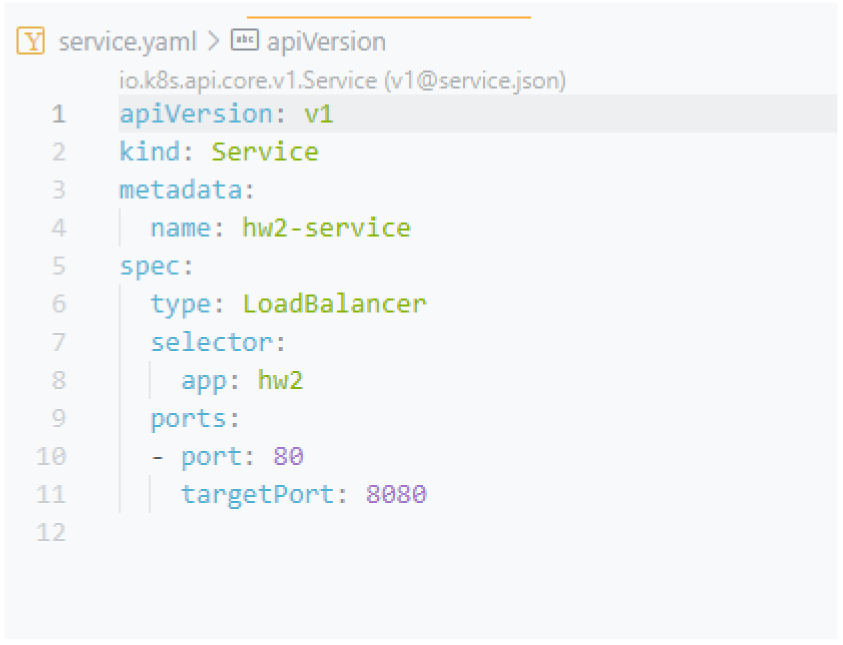
Keerthan Srinivas: Creation of github, creation of git repository, committing and pushing of initial files. Wrote the readme file.



```yaml
apiVersion: apps/v1
kind: Deployement
metadata:
  name: HW2-645-Deployement
  labels:
    app: hw2
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hw2
  template:
    metadata:
      labels:
        app: hw2
    spec:
      containers:
      - name: StudentSurvey-HW2
        image: asdpkp/hw2-645-repo:myfinale
        ports:
        - containerPort: 80

```

Figure 2Deployementyaml


service.yaml > apiVersion
io.k8s.api.core.v1.Service (v1@service.json)

```yaml
apiVersion: v1
kind: Service
metadata:
  name: hw2-service
spec:
  type: LoadBalancer
  selector:
    app: hw2
  ports:
  - port: 80
    targetPort: 8080

```

Figure 1Service.yaml