**Final DevOps Project By:**

**Ahmad Alkoshah**



Table of Contents

[HTU Environment Rebuild 4](#_Toc221290848)

[1.Server Rebuild Task 4](#_Toc221290849)

[2.OS Installation Requirements 4](#_Toc221290850)

[**3. Initial System Setup (Post-Installation)** 6](#_Toc221290851)

[4. **Storage Configuration** 7](#_Toc221290852)

[5. **Departmental Directory Structure** 9](#_Toc221290853)

[**6. User Groups and Accounts** 10](#_Toc221290854)

[**7. Backup Mechanism** 10](#_Toc221290855)

[**8. Server Optimization with TuneD** 11](#_Toc221290856)

[**9. System Identity** 11](#_Toc221290857)

[**10. YUM Repository Configuration** 12](#_Toc221290858)

[**11. Temporary HR Web Application Placeholder** 13](#_Toc221290859)

[12. SSH Service Hardening 15](#_Toc221290860)

[**HTU-Schedemy-Website** 17](#_Toc221290861)

[1.Back End Repository 17](#_Toc221290862)

[2.Front End Repository 31](#_Toc221290863)

[3. Setting up a CloudWatch Dashboard 33](#_Toc221290864)

[a. Traffic metrics 33](#_Toc221290865)

[b. Performance metric 34](#_Toc221290866)

[c. the infrastructure Health 35](#_Toc221290867)

[HTU-Schedemy-Website (CI/CD) 35](#_Toc221290868)

[1. Aws Setup 36](#_Toc221290869)

[2. Ansible YAML file 38](#_Toc221290870)

[3. GitHub Actions 41](#_Toc221290871)

[3.challenges and problems 46](#_Toc221290872)

[Pixel-Fed photo app 48](#_Toc221290873)

[1.Server Set up 48](#_Toc221290874)

# HTU Environment Rebuild

**Scenario Overview:**

For this scenario, we are going to create a RedHat system from scratch, doing the initial necessary set up from the file structure to the storage configuration, Backup Mechanisms and a small web server.

## 1.Server Rebuild Task

We start by setting up the RHEL 9.7 system using Oracle VirtualBox with the following specs:

* 4 vCPu
* 4 GB RAM
* 20 GB Disk
* 40 GB Disk

## 2.OS Installation Requirements

During the system installation set up we follow the provided specifications in the assignment brief as shown in the figure 2-1 below.

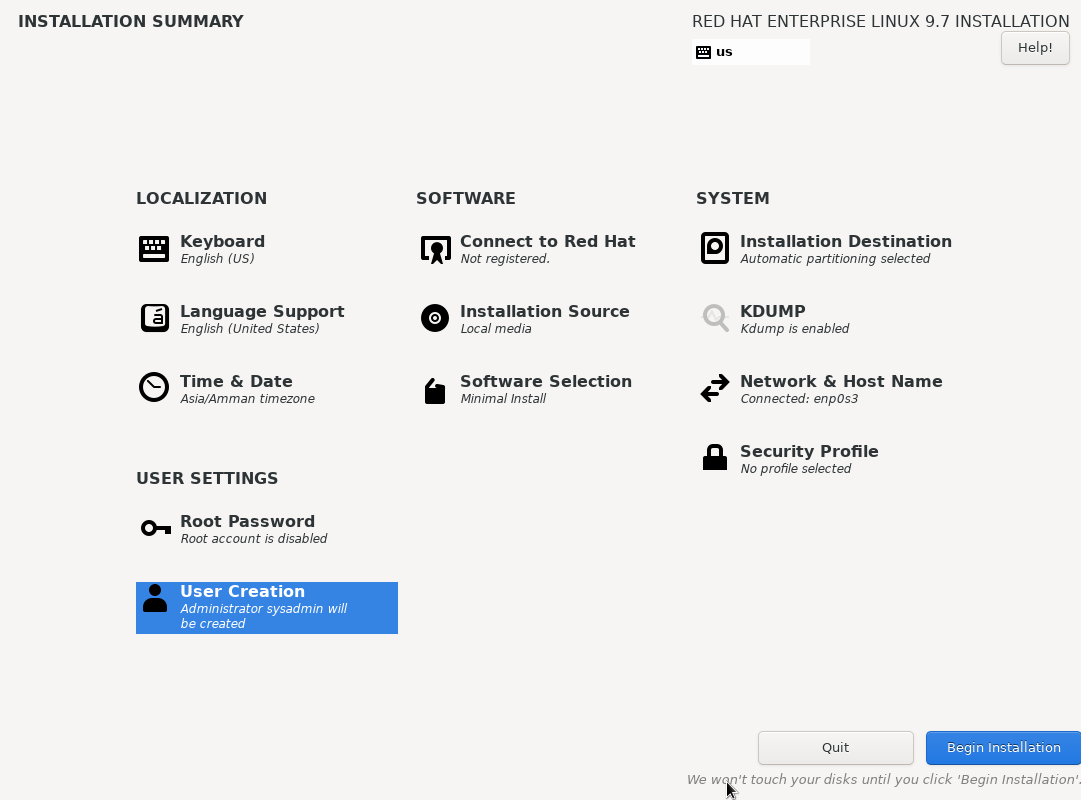


Figure 2-1

Now to achieve automation over this process of installation, we need to create the kickstart file.

Luckily for us, RedHat automatically generates a kickstart file for the configuration we have provided while installing the operating system, saved in the path **“/root/anaconda-ks.cfg**” check figure 2-2.

Now we change the network settings for the image to Bridged Adapter, we restart it then we obtain its ip address using the command “ ip a “, then we connect to the system via ssh.

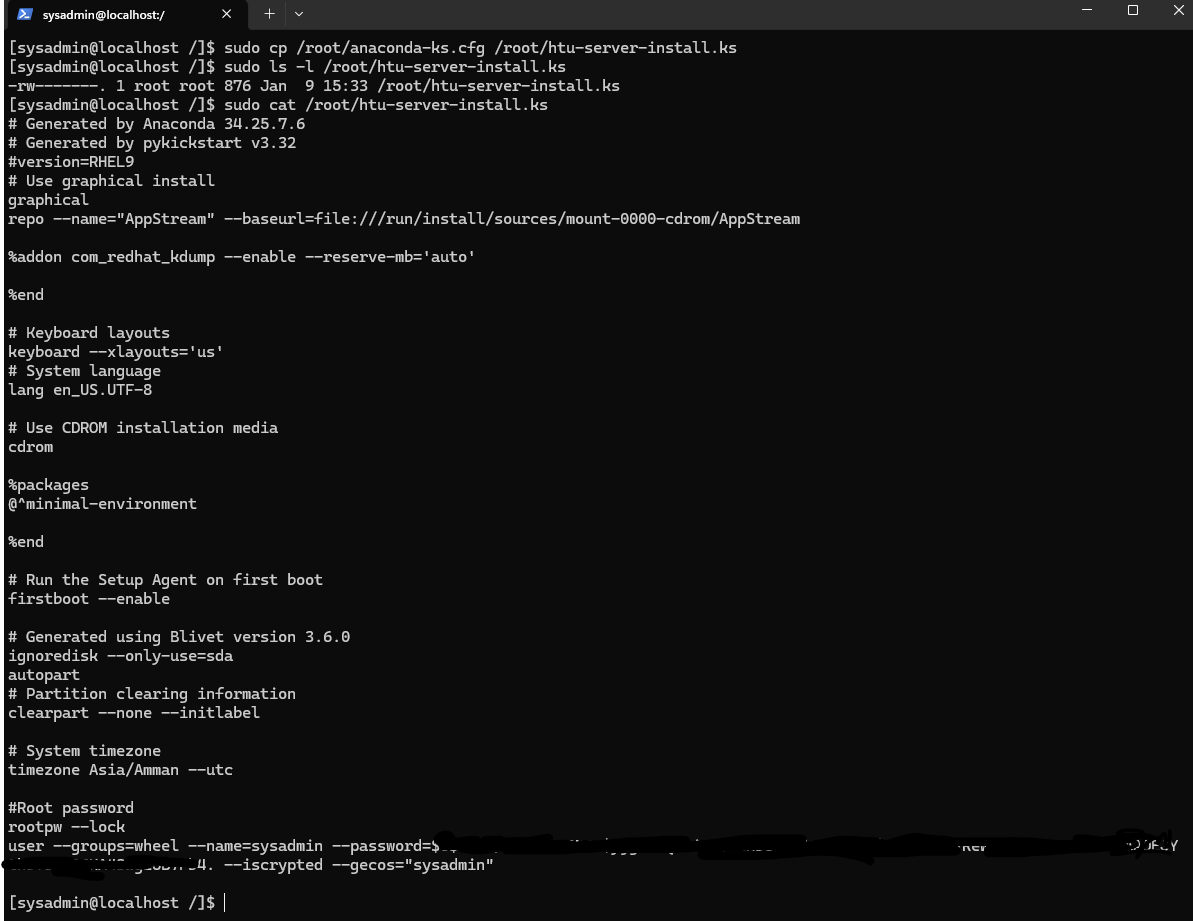


Figure 2-2

## **3. Initial System Setup (Post-Installation)**

After we’re done with the installation process, the first thing we do is to register with RedHat Subscription Management via the command “**rhc connect**” thus granting us the access to the DNF private repository, then we update all the packages using the command “**dnf update -y**”.

And we install the necessary package including but not limited to “**nano, VDO, kmod-kvdo, rsync, tuned, httpd/nginx, net-tools**” check figure 3-1 below.

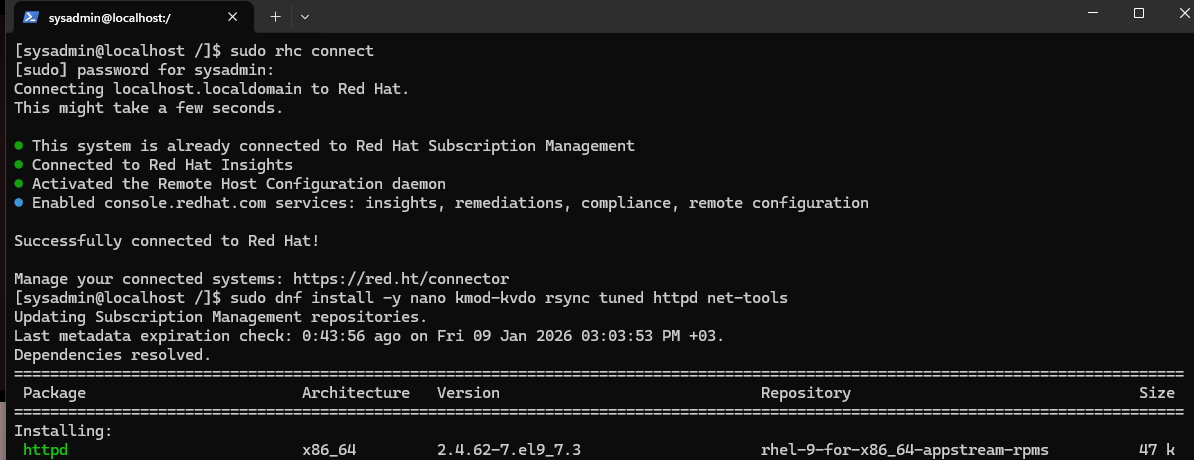


Figure 3-1

## 4. **Storage Configuration**

Now we first start by verifying that the 40G storage we created earlier is available using the command “**lsblk**” , we create a partition and put the labels then we create the physical volume and physical group to prepare for the logical volume creation, check figure 4-1.



Figure 4-1

This is followed by creating 3 logical volumes “home” and “company” as well as “swap” given 4G according to RedHat’s documentation for our 2-8 G ram and followed by formatting the file system for both logical volumes as “**XFS**”, check figure 4-2.

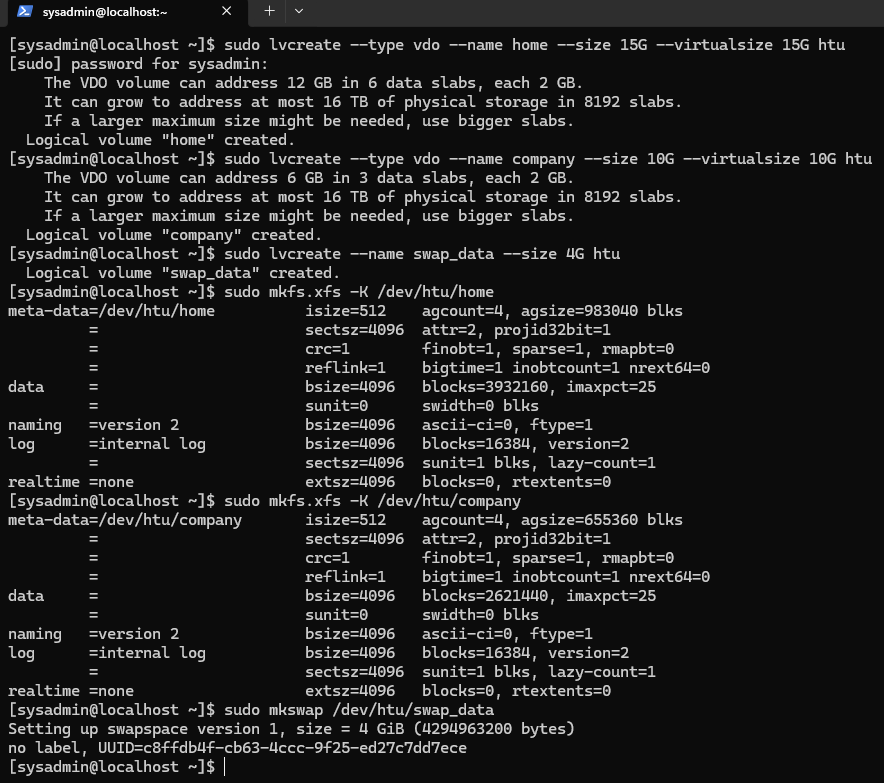


Figure 4-2

Now, we create a new home directory to mount to then we sync it with /home using “**rsync**”, so we don’t lose the data if we mount /home directly then we unmount it, check figure 4-3.

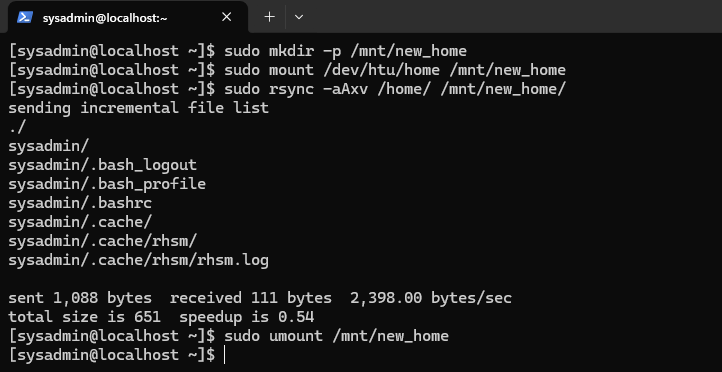


Figure 4-3

We add the fstab entries for persistence, make the new /home directory after renaming the old one then we use the command “**mount -a**”, but we are faced with an issue that the fstab is old we need to do a “**systemctl daemon-reload”** then we run the command again.

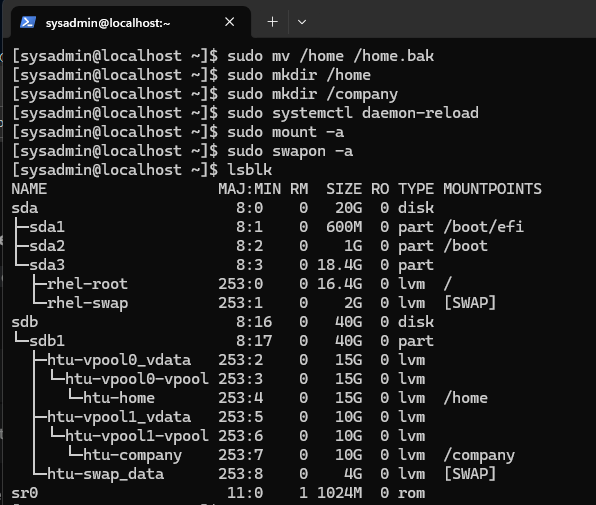


Figure 4-4

## 5. **Departmental Directory Structure**

For the creation process of the departmental files, we use environment variables to make future commands easier and more efficient, then we create the 4 groups and their respective departments, we make the group the owner of their respective directories, we give them “**3770**” thus achieving a few things:

* Department isolation and group collaboration.
* Deletion prevention for the members files by others (sticky bit).

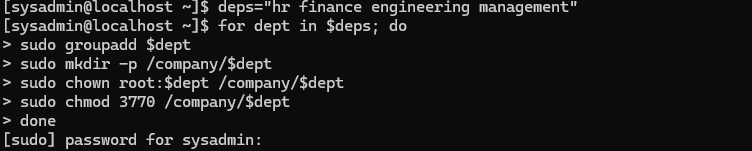


Figure 5-1

## **6. User Groups and Accounts**

Now we create the “IT” group first since it doesn’t have a shared directory, I created a small bash script that does the user creation process and add every user to their respective groups, check figures 6-1 and 6-2.

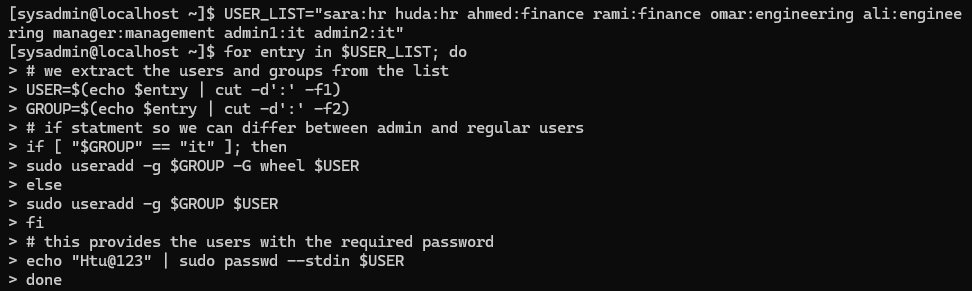


Figure 6-1

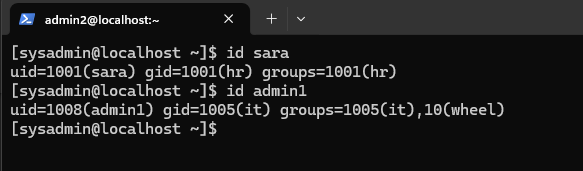


Figure 6-2

## **7. Backup Mechanism**

We start by creating a simple configuration file in “**/etc/cron.d**” named “**backup\_job**”, check figure 7-1.

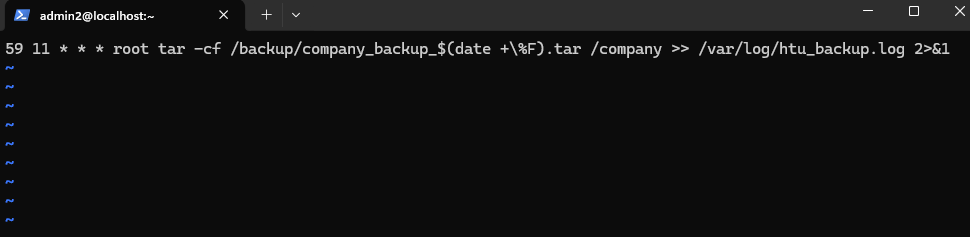


Figure 7-1

## **8. Server Optimization with TuneD**

We have already installed tuned in step 3, so we just make sure that the daemon works on boot and use the virtual-guest profile, check figure 8-1.

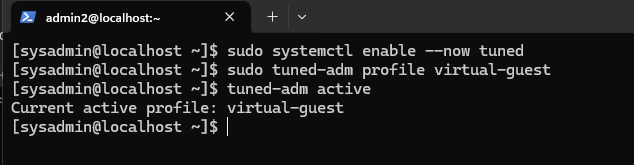


Figure 8-1

## **9. System Identity**

We start by setting the host’s name using the command “**hostnamectl set-hostname**”, and to ensure that the local hostname resolves without DNS we must add it to the file **“/etc/hosts**” check figure 9-1 and 9-2.

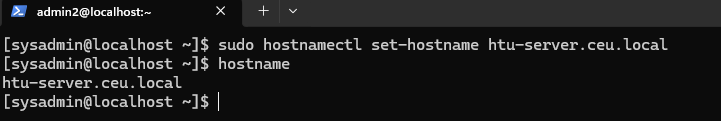


Figure 9-1

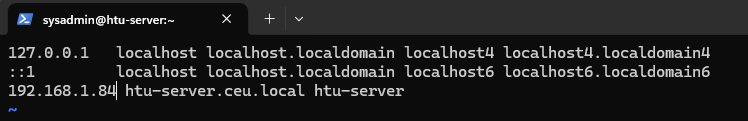


Figure 9-2

## **10. YUM Repository Configuration**

We start by executing the command “**sudo dnf install -y yum-utils**” so we can use the command “**yum-config-manager**” to add the Repositories, check figure 10-1.

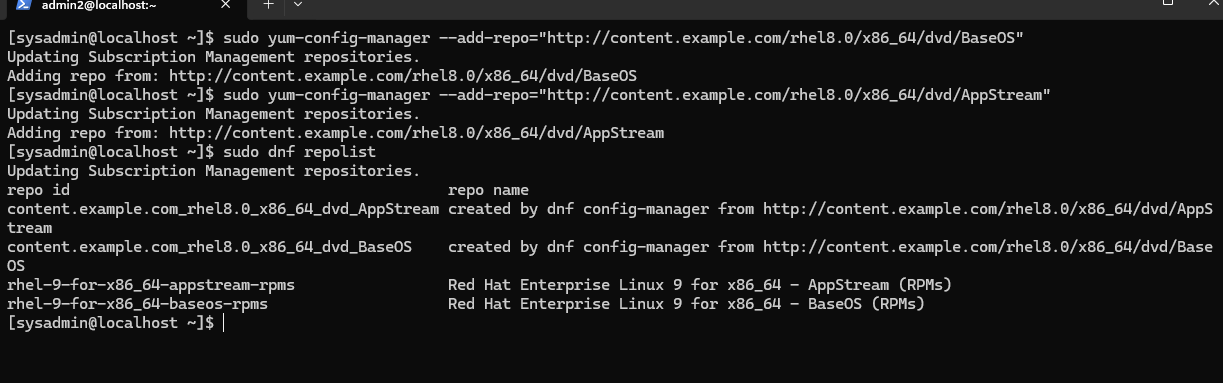
****

Figure 10-1

**Here I faced a problem that dnf install returns an error after adding the yum repos, check figure 10-2.**

****

Figure 10-2

After investigating, I found out that dnf by default checks the GPG keys for the URLs, but since the ones we used are dummy URLs this won’t work, we must remove the fake URLs manually for it to work again.

## **11. Temporary HR Web Application Placeholder**

we start by creating the html file for a simple “**coming soon** “web application and place it in **"/opt/hr\_placeholder**”, then we create a simple bash script that deploys that webpage, check figure 11-1.

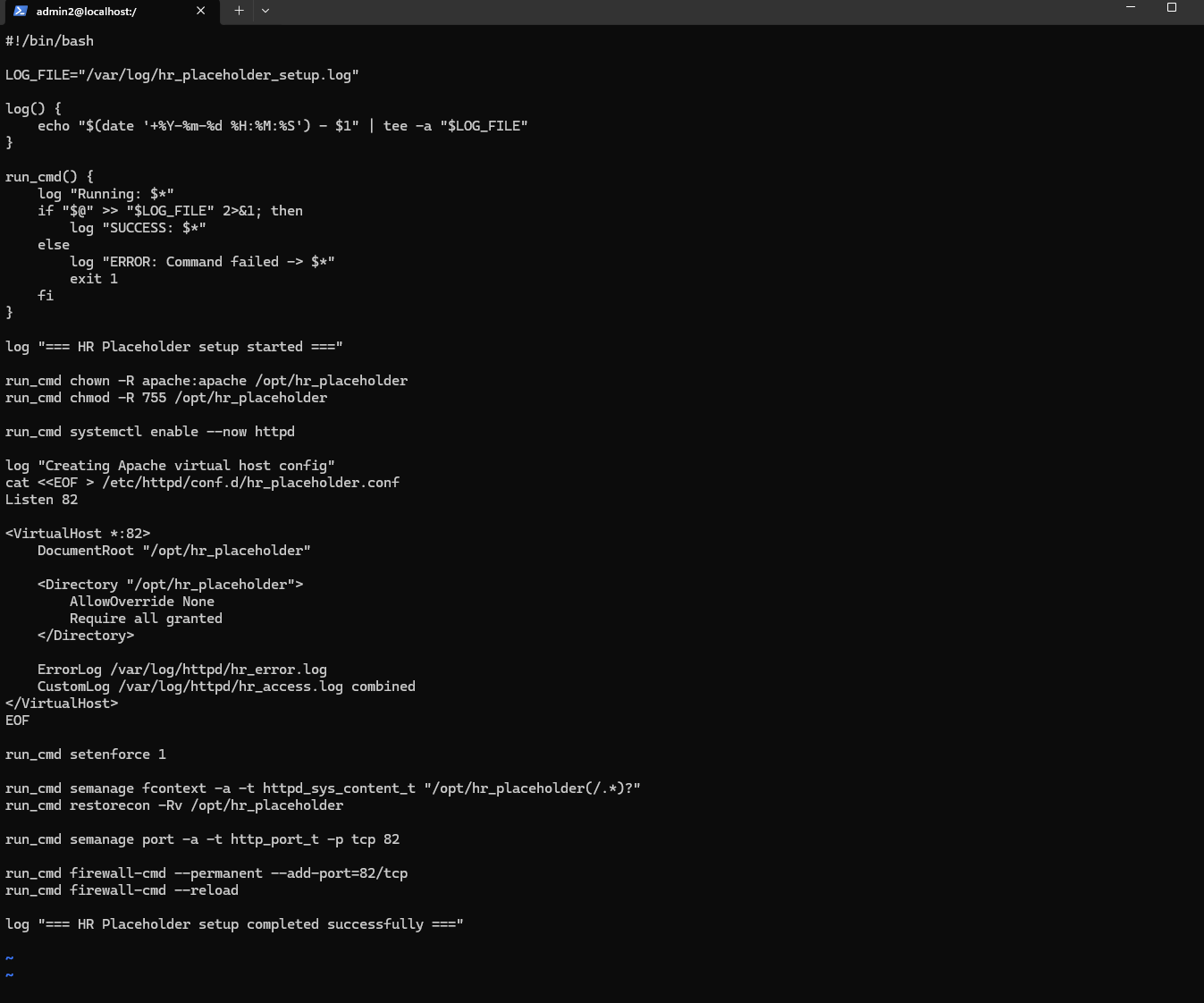


Figure 11-1

This bash script provides:

* Logging
* Failure tolerance: if a command fails it exists the script
* Gives owner permissions to Apache for the hr\_placeholder directory
* Make Apache a persistent daemon that loads on system start.
* Creates the .conf file so Apache can listen to the non-standard port
* Uses SElinux to allow Apache to access /opt and use port 82
* Adds a firewall rule for port 82

The webpage now works correctly, check figure 11-2.

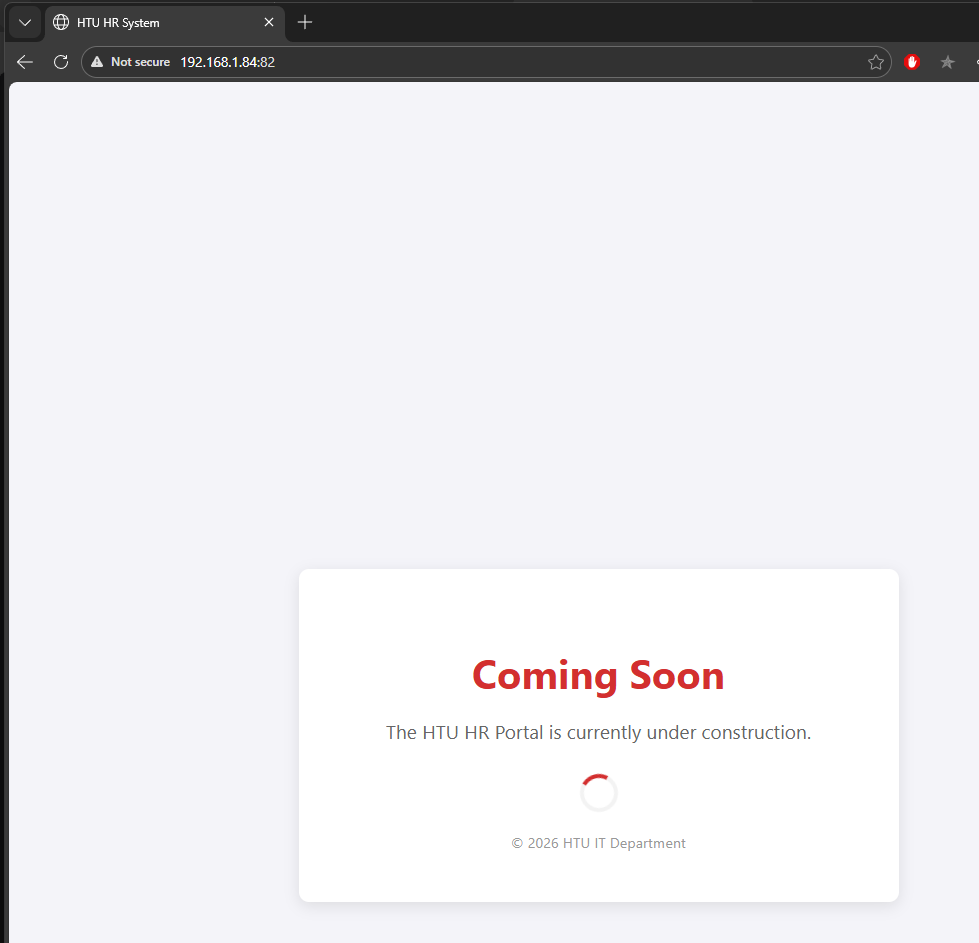


Figure 11-2

## 12. SSH Service Hardening

We start by creating the ssh key in our windows CMD using the command “**ssh-keygen”**, check figure 12-1.

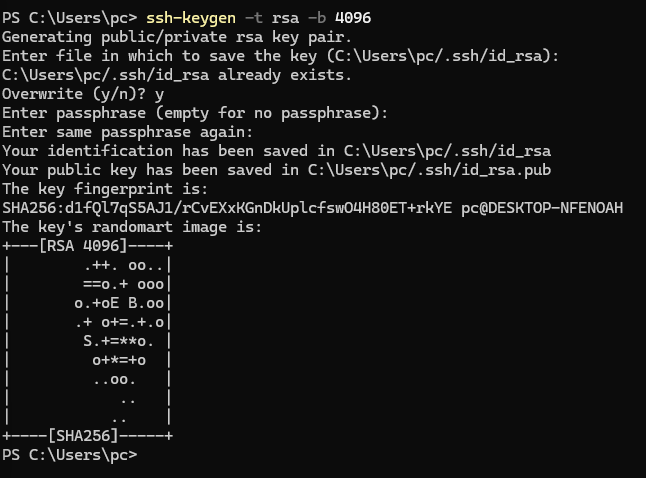


Figure 12-1

Now the ssh public key is stored in \.ssh in windows, we connect via sftp to the RedHat system and use the command “put C:\\Users\\pc\\.ssh\\id\_rsa.pub /home/sysadmin/.ssh/authorized\_keys”

To send the public key to the /.ssh that we created in the sysadmin home directory.

We give the /.ssh file “700” and the authorized\_keys “600” making it that only sysadmin can read write execute the .ssh and read/write the authorized\_keys.

Now we connect using ssh, it doesn’t ask for password since we use the key, check figure 12-2.

We also do the same process for admin1 and admin 2

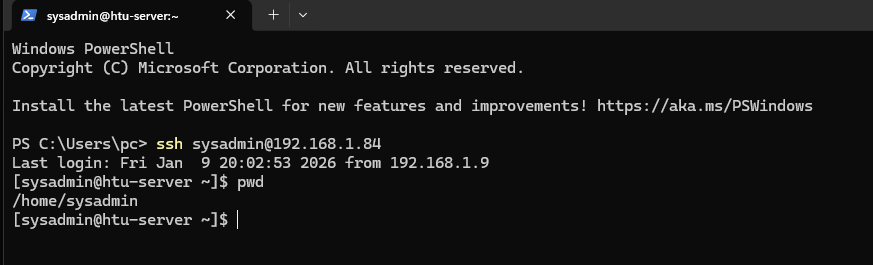


Figure 12-2

Now to disable password authentication for all users we can edit the “/etc/ssh/sshd\_config” and manually add the following lines:

* PermitRootLogin no
* PubkeyAuthentication yes
* PasswordAuthentication no

Then we save the file then run the command “**sudo systemctl reload sshd**”.

In a different CMD tab we use the “**ssh**” command using 2 different accounts to check if everything worked out, check figure 12-3.

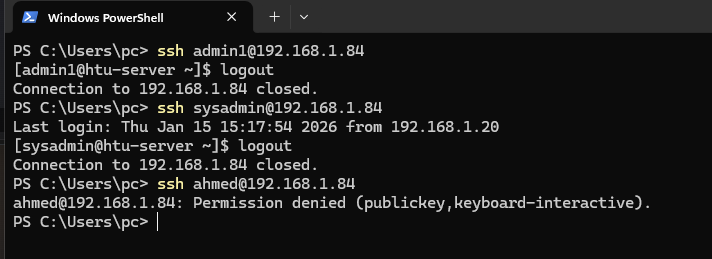


Figure 12-3

Only sysadmin/admin1/admin2 connected and the other accounts are not authorized which is the result we want.

# **HTU-Schedemy-Website**

**Scenario Overview:**

For this scenario, we have both backend and frontend repositories, we will be hosting the backend on a bunch of ec2 instances and deploy the frontend and connect them together.

## 1.Back End Repository

The first step is to fork the repository for the backend then clone it to our computer and open it using any IDE, in our case “**intellij IDEA 2025.3.1.1**”, check figure 1-0.

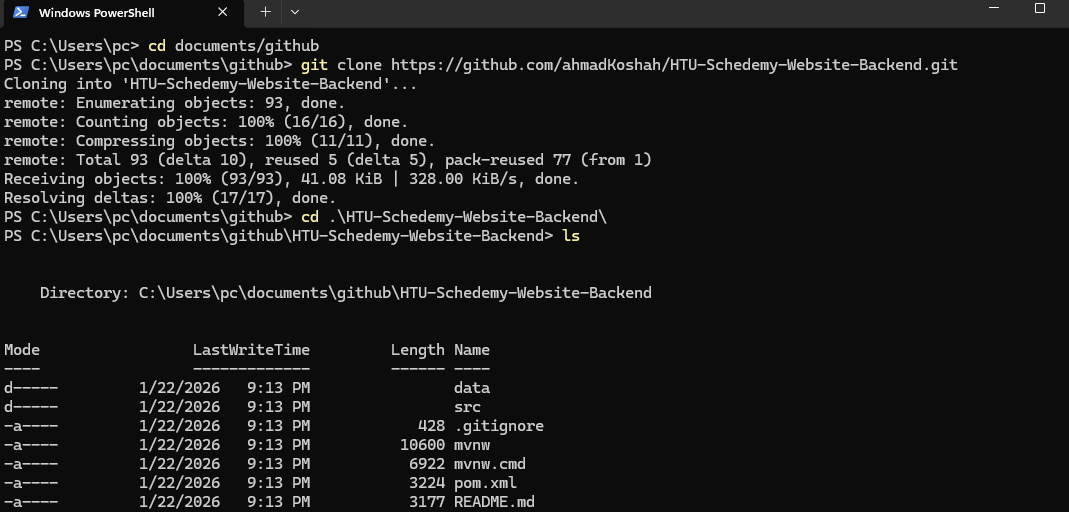


Figure 1-0

Then we open the project using the IDE and change the file “**application.properties**” and change the Database to be in memory so we won’t face problems later when we keep running the program.

Then we clean and package the code using maven and we get a “**.jar**” file, check figure 1-1.

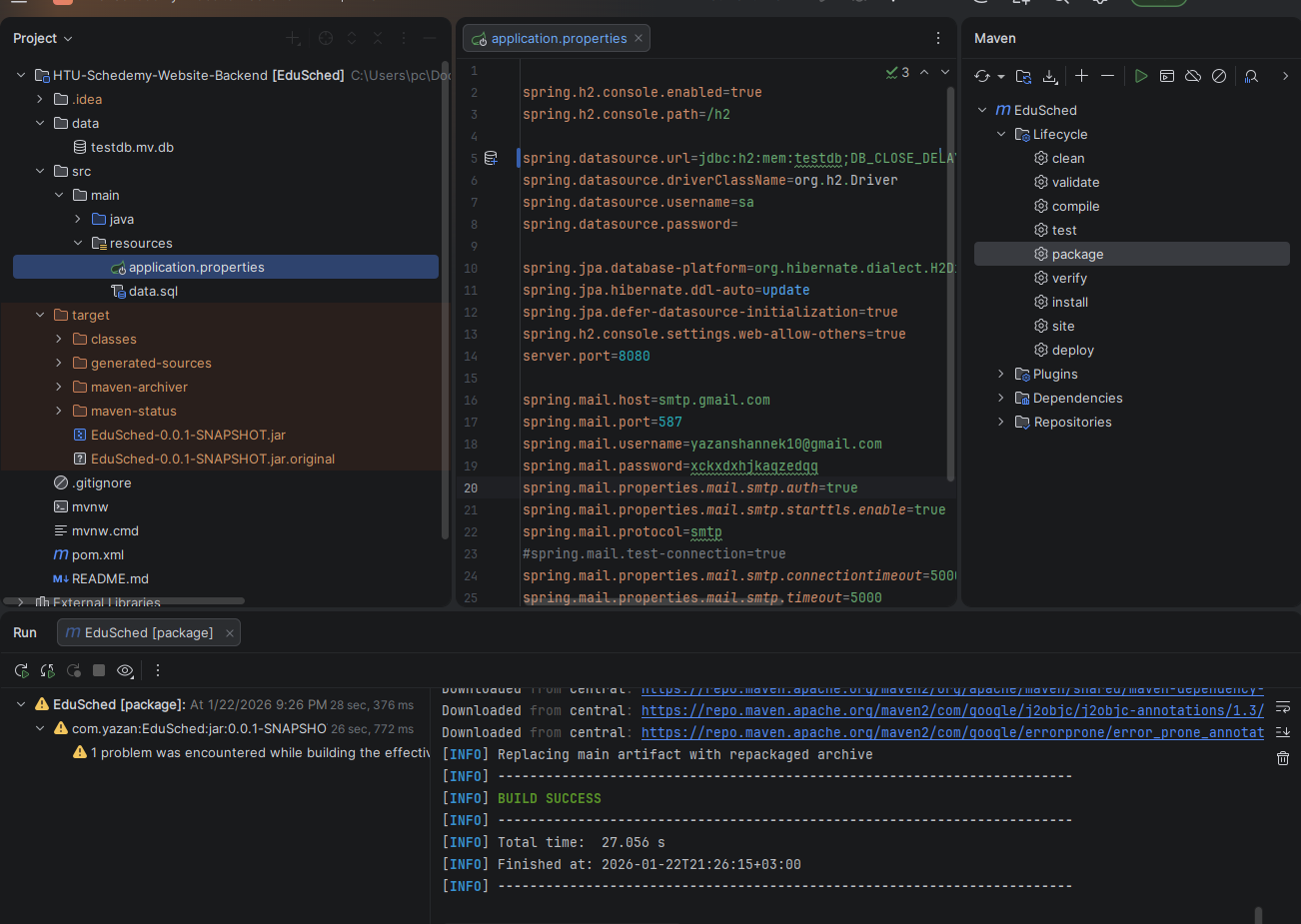


Figure 1-1

Now we stage, commit and push the code again to GitHub because we changed the “**application.properties**” , check figure 1-2.

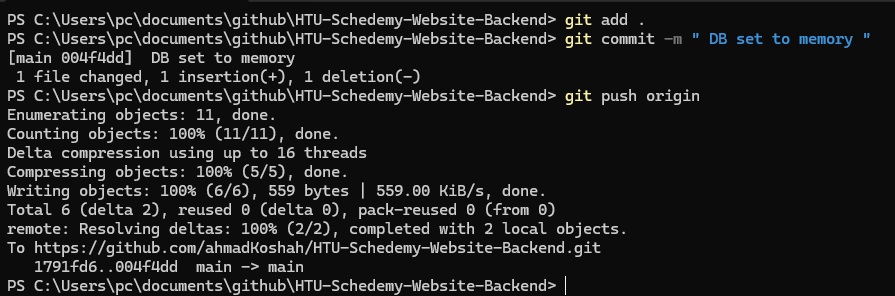


Figure 1-2

Now we head to AWS and create an instance and name it “**Backend\_schedemy**”, we set the instance to amazon Linux and use t3.small since it is general purpose and can be used for small applications and we can scale up if we need more CPU and ram for larger traffic, check figure 1-3.

We allow ssh and create a key pair and open a custom TCP port 8080.

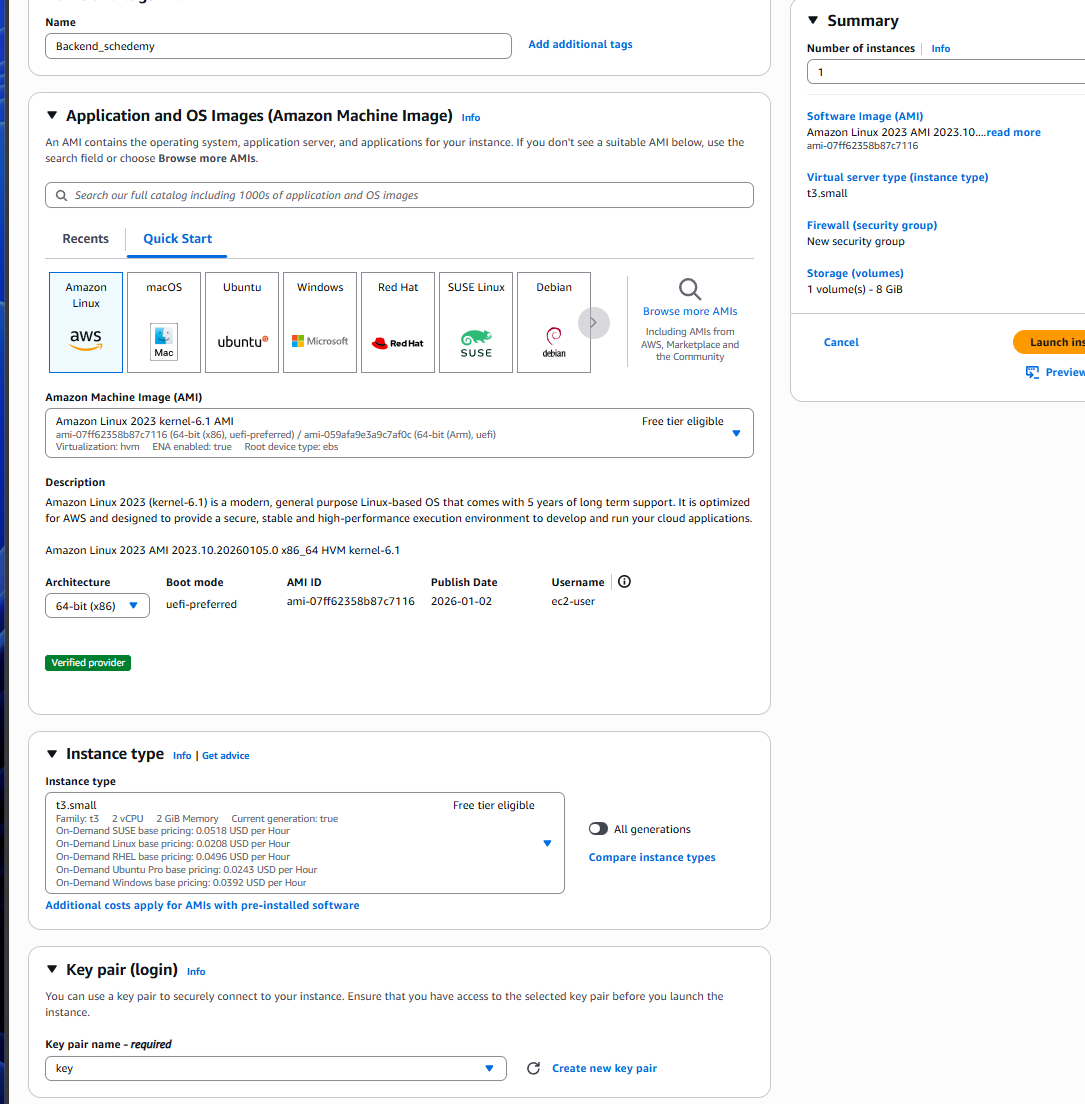


Figure 1-3

We connect to the instance via ssh, update the packages and then install java-17 for our project, check figure 1-4

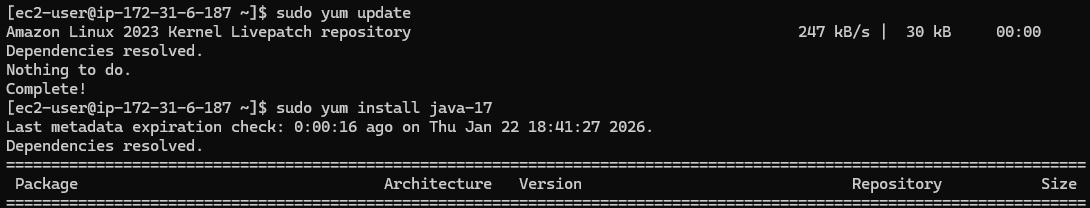


Figure 1-4

Now that our server is set up, we need the jar file to be in the ec2 instance, so we use sftp to put the file in the server, then we move it to **“/opt/myapp**” and give the file ownership to ec2-user.

Now we execute the file using “java -jar” and we make it run the background and persistent, so it doesn’t stop when we close the terminal, check figure 1-5.

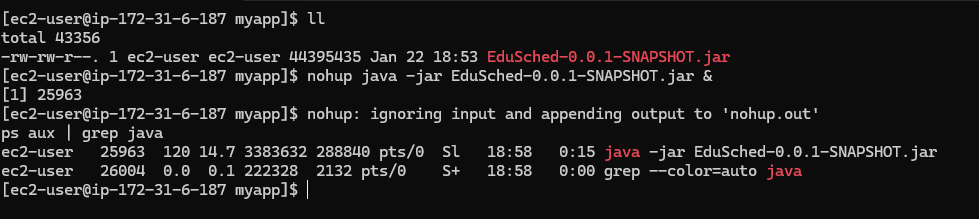


Figure 1-6

Now we access the ip with the port “8080” and hit one of the methods “rooms” to check if everything is correct, check figure 1-7.

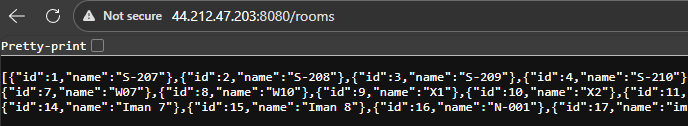


Figure 1-7

We can see that it works perfectly, now to make it run on every instance we create upon power on, we need to make it a system service.

We create the. service file in “**/etc/systemd/system/myapp.service**”.

This unit file executes the application upon power on and runs in the background as a system service, check figure 1-8.

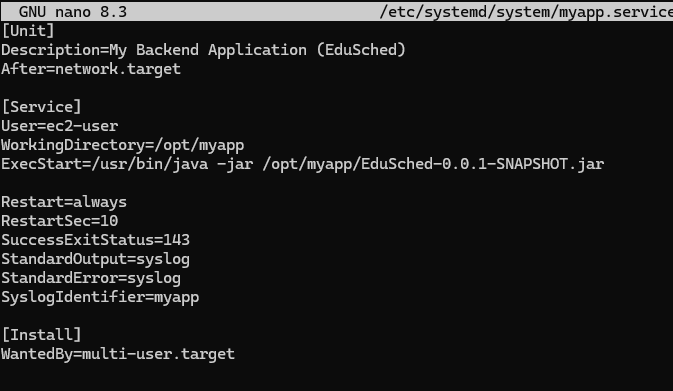


Figure 1-8

Now we reload the systemctl daemon then enable and start our service and reboot the device to check if it works, check figure 1-9.

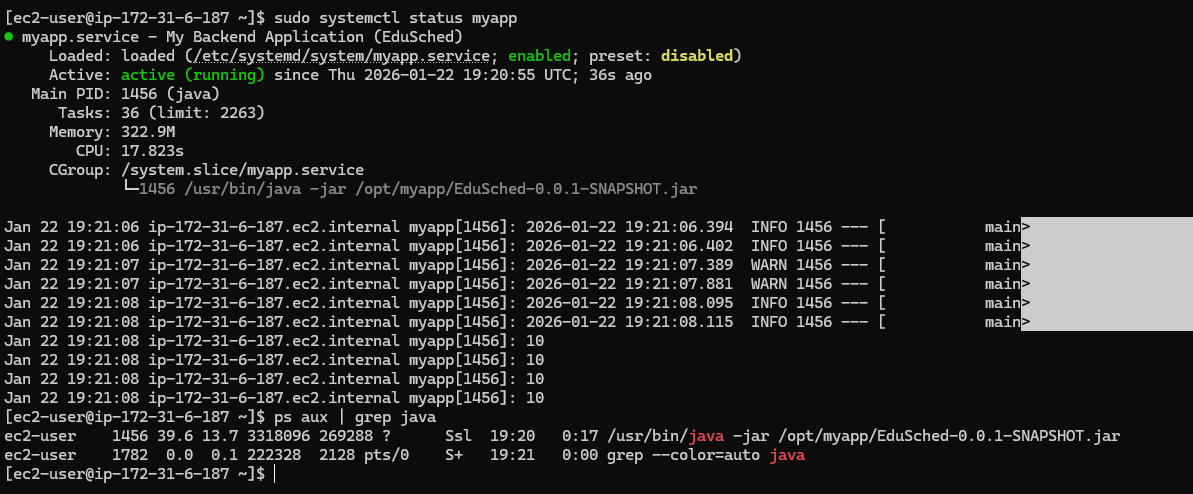


Figure 1-9

The service is enabled to work after reboot and runs perfectly.

Now that our instance is ready and running, we create an AMI so we can scale out later by going to AWS instances, actions then image and templates then we create the image.

We do the same thing and create a template and choose our image from “My AMIs”, check figure 1-10.

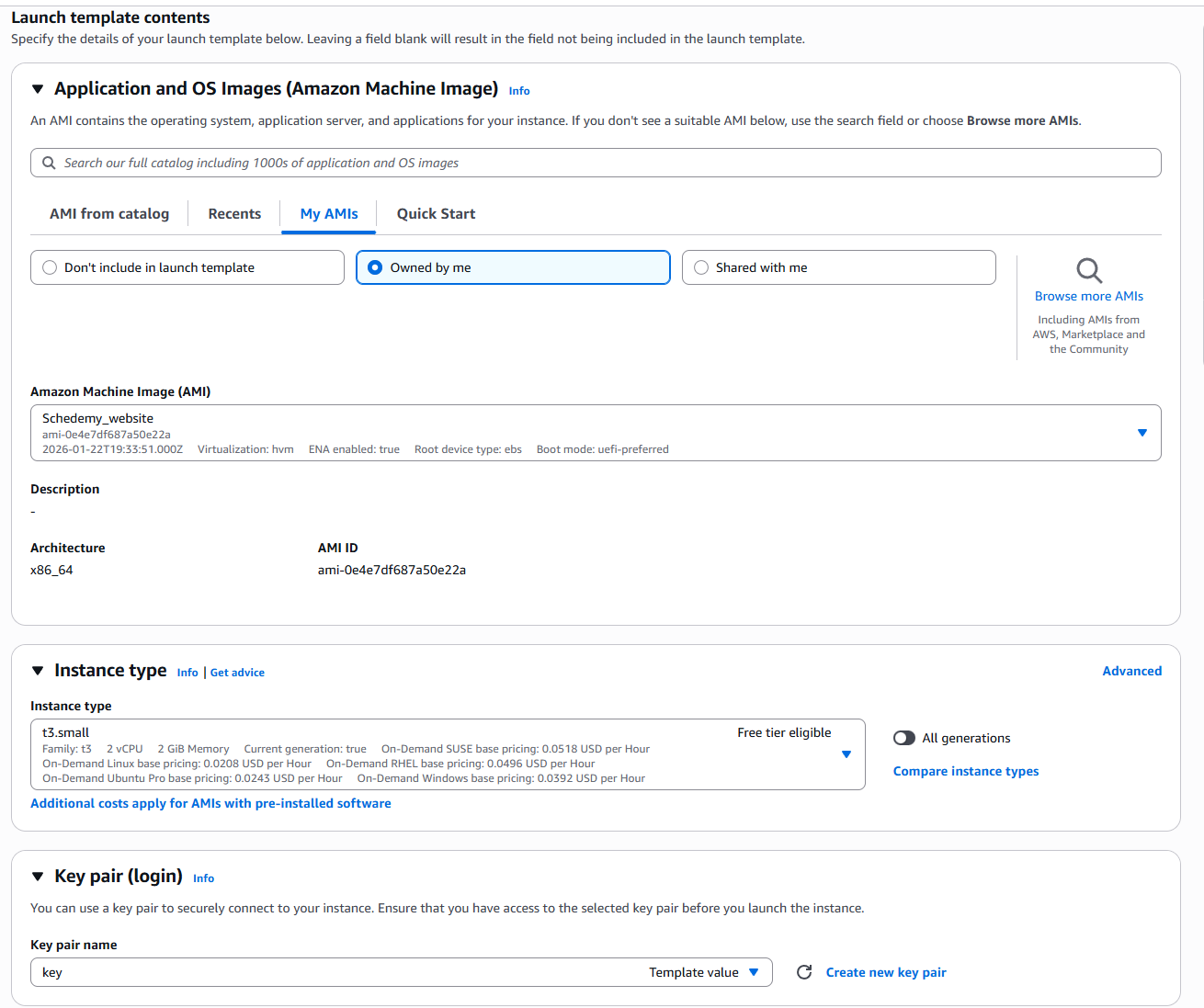


Figure 1-10

For now, we only allow ssh in the security group, but we will change that later then we create that template.

Now we create a target group and put the port to be 8080 , and for the health check we need to change the root / and put /instructor or any function because I have faced an issue with the Target group health because the root page returns 404 not 200 so it remains unhealthy, check figure 1-11.

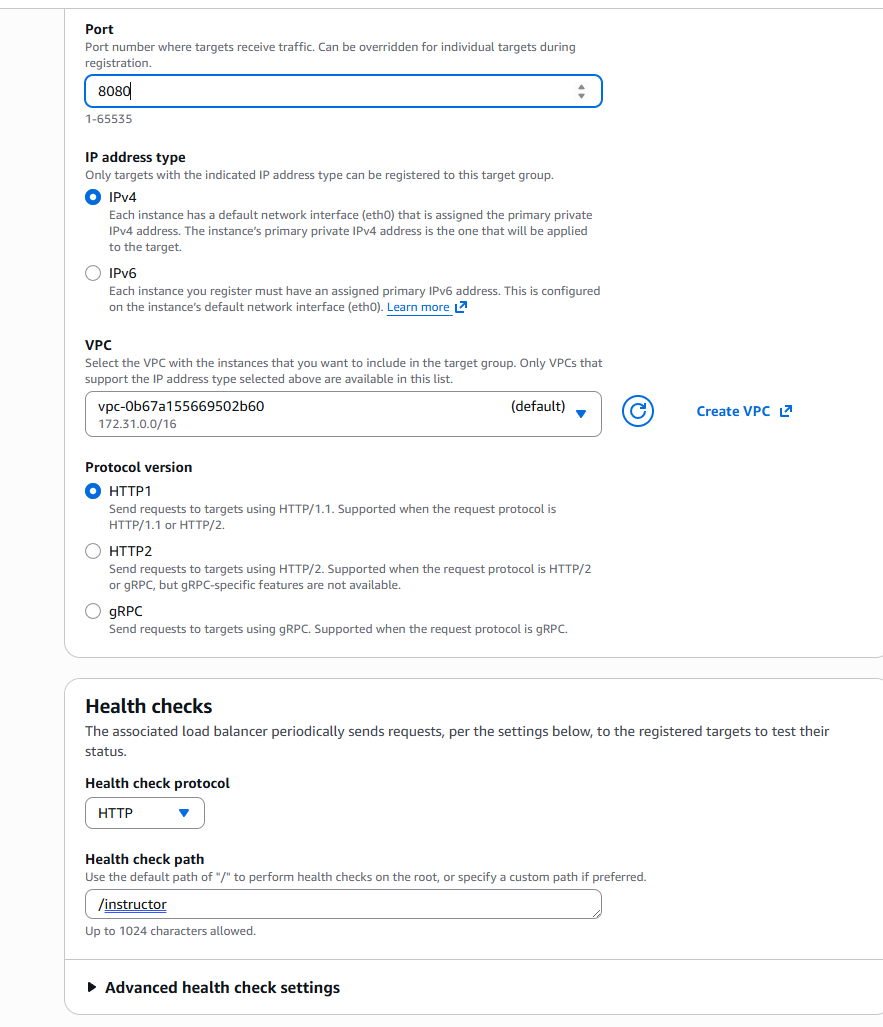


Figure 1-11

Then we add the instance we already created above to the target group and we create it.

Now we proceed with the process by creating an application load balancer that listens to port 80 not 8080 as it will be taking users http requests and we will forward it to the previous target group that we created, check figure 1-12.

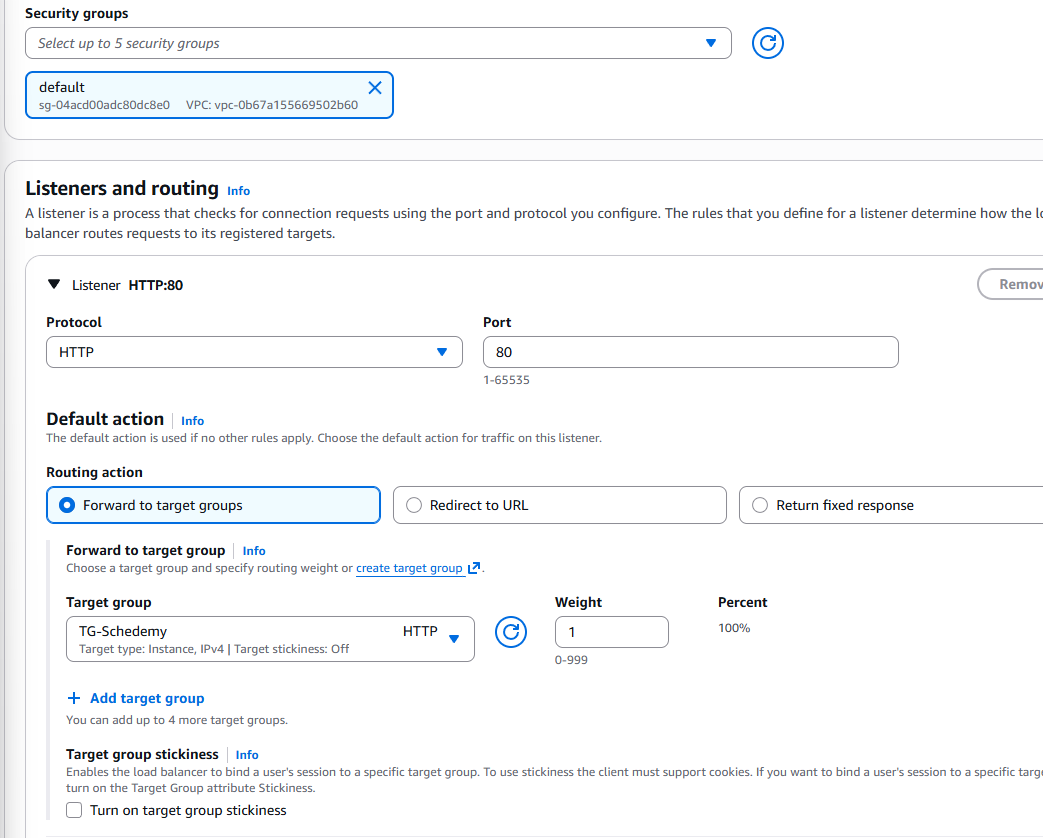


Figure 1-12

Then we create the load balancer and wait for it to be active.

Now we create the ASG, we attach the launch template we created earlier, add all the availability zones like we did in the ALB then we attach the ALB.

Now for the group size in the ASG we choose the desired to be 2 the min to be 2 and the max to be 4, this insures we don’t have a single point of failure and we can scale our with a max of 4 instances, and we make the scaling policy to be when the AVG CPU Utilization hits 80%, check figure 1-13.

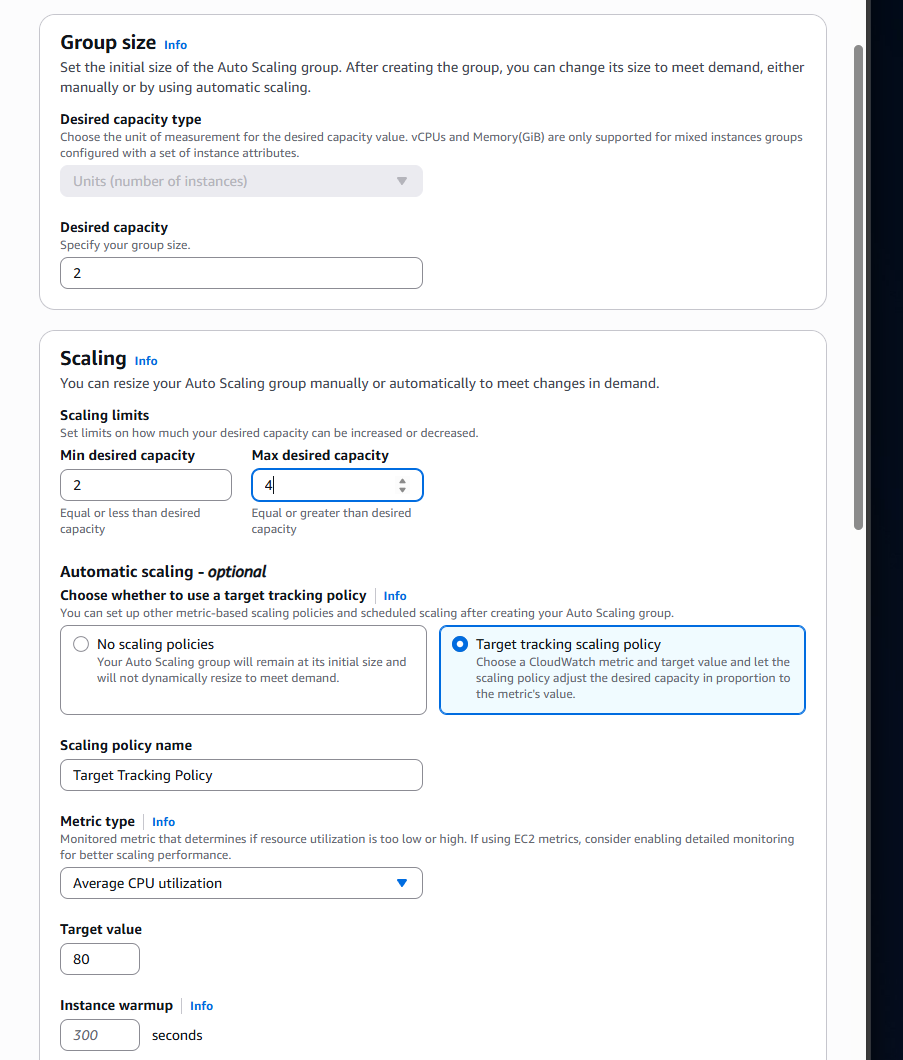


Figure 1-13

Then we add a notification topic via SNS to send us an email on the events shown in figure 1-14.

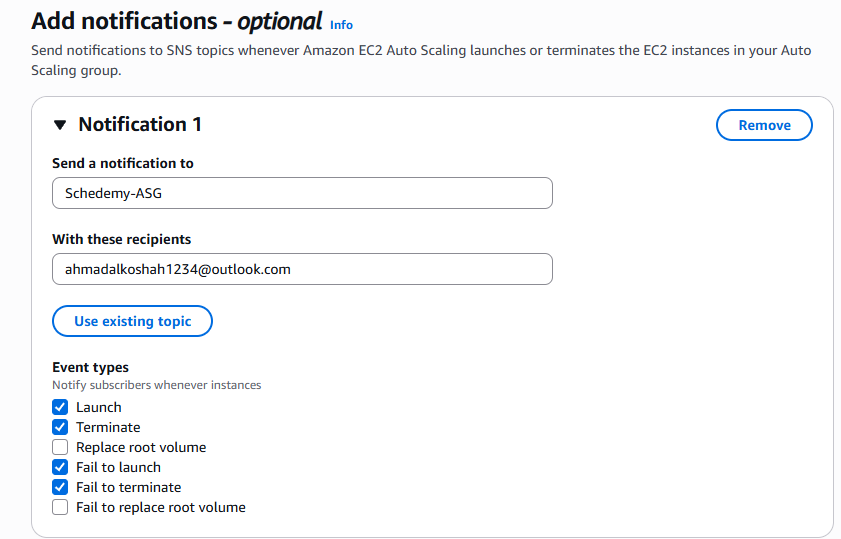


Figure 1-14

Now we have a small issue where all the instances are healthy and everything works out well, but the ALB DNS link doesn’t work, after some investigation it turns out the problem is that the ALB’S security group didn’t have a rule that allows http from anywhere.

Now we go back to the security group attached to the launch template and allow custom TCP port 8080 and set the source to be the security group of the ALB, check figure 1-15.

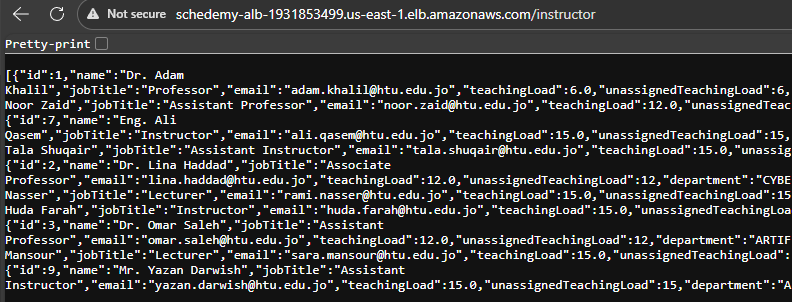


Figure 1-15

Now we want a public certificate to use with our domain name, so we go to AWS Certificate Manager, and we request a public certificate for a sub domain in the hosted zone in our instructor’s Yazan account, check figure 1-16.

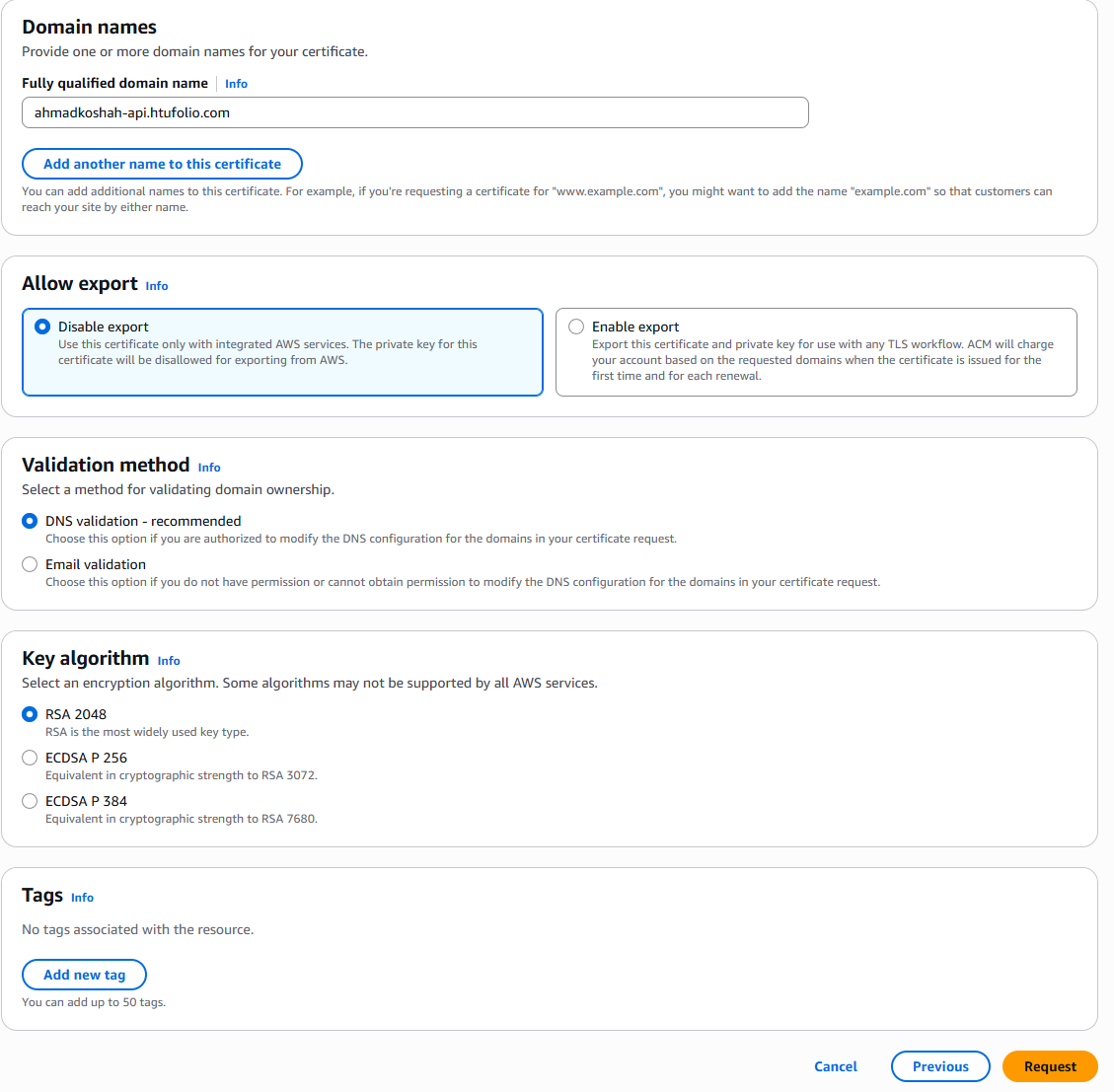


Figure 1-16

After creating , we copy the Cname name and the Cname value, then we head to Yazan’s account then route 53 hosted zones , we click on the “htufolio.com” then create a new record, then we add the cname name in the record name , record type set as cname  
and we paste the cname value in the value box, then we hit create and wait for a few minutes, check figure 1-17.

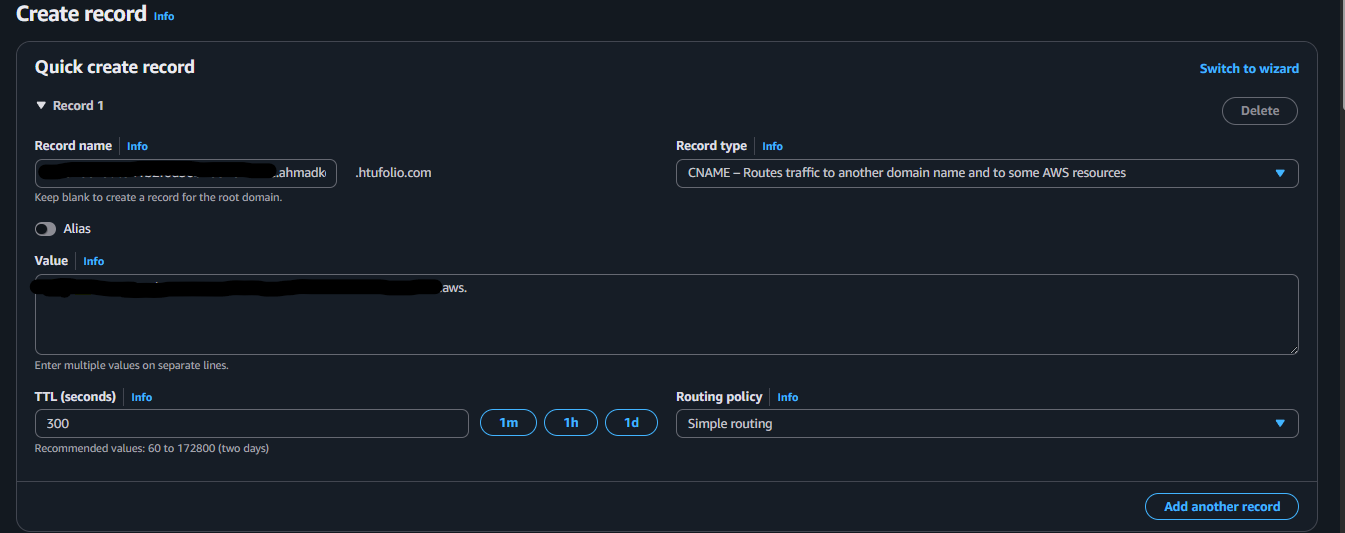


Figure 1-17

Now we go back to the first account and refresh, check figure 1-18.

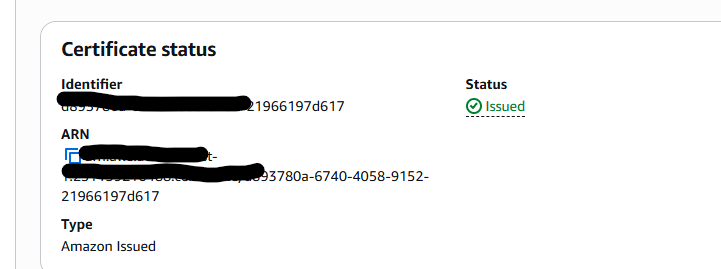


Figure 1-18

Now we use CloudFront to achieve fast access from different geographical locations, we create the distribution , choose ELB as the origin then load it, we set the cash settings to redirect http to https so we force https and for cache policy we choose Caching Disabled because we want CloudFront to always get the latest data from the backend, we can’t afford having old data with caching, check figure 1-19.

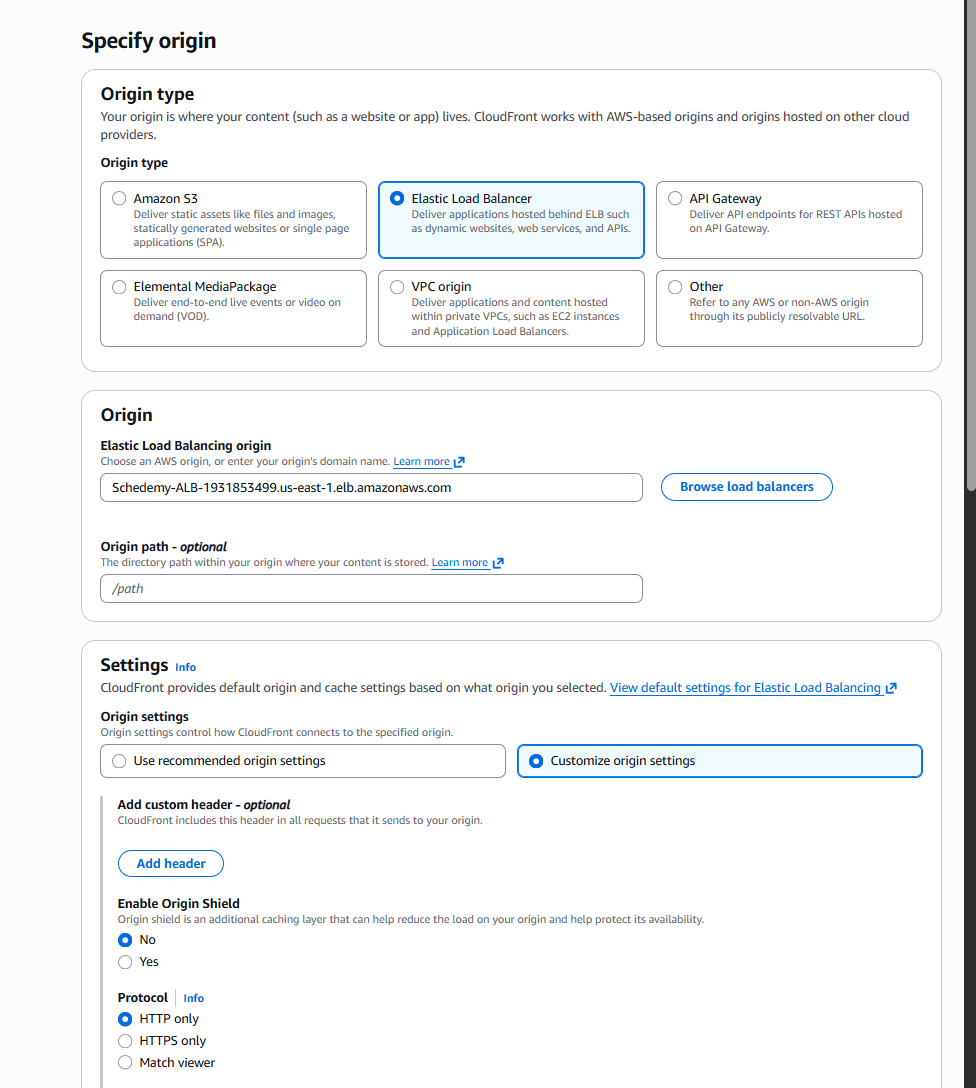


Figure 1-19

Then we hit next and we disable WAF as it is not needed currently, then we create the distribution.

We add the domain name in the distribution “ahmadkoshah-api.htufolio.com” and choose the certificate we created earlier then we add the domain, check figure 1-20.

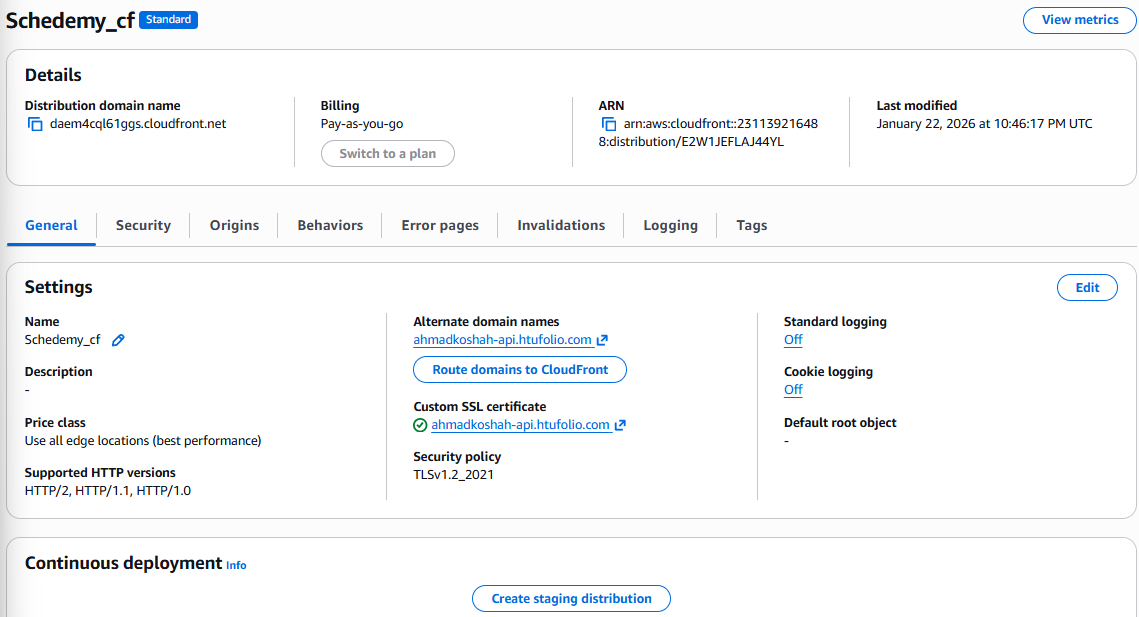


Figure 1-20

Now we take the distribution domain name, and we create a record in Yazan’s account, record type cname and we add our subdomain and the value as the distribution domain name, now we wait for a bit then check the domain, check figure 1-21.

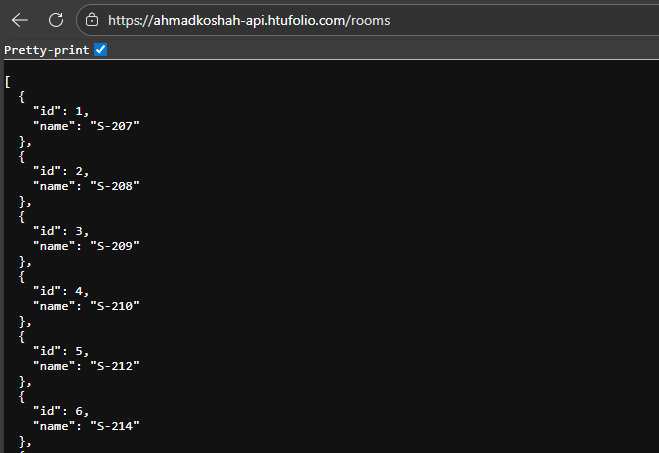


Figure 1-21

## 2.Front End Repository

Now that our backend API works perfectly, we will pull the front end code and reroute it to our API then deploy it on amplify.

We pull the repo and then open it with an IDE to reroute, check figure 2-0.

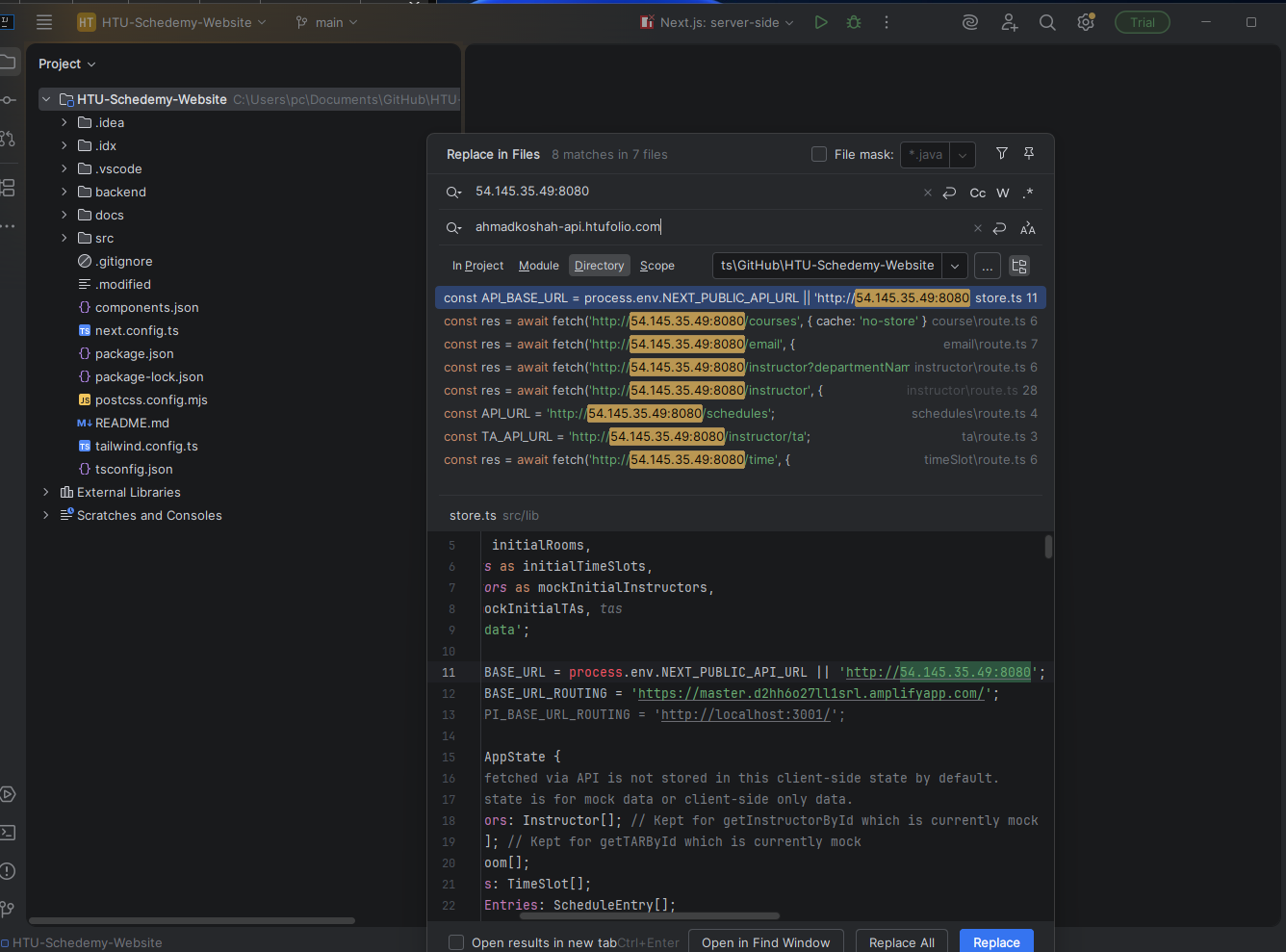


Figure 2-0

We then replace the previous API call with our new subdomain, then stage and commit and push the repo.

Now we head to AWS Amplify, create a new app then we select GitHub and connect to our GitHub account, and we choose the frontend repo, check figure 2-1

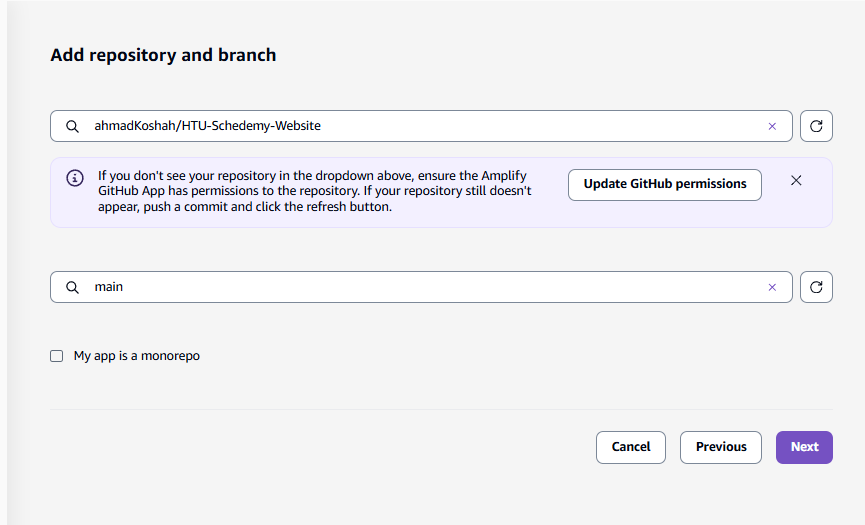


Figure 2-1

Then we hit next, next and create the app, wait a few minutes for it to be deployed then access the link, check figure 2-2.

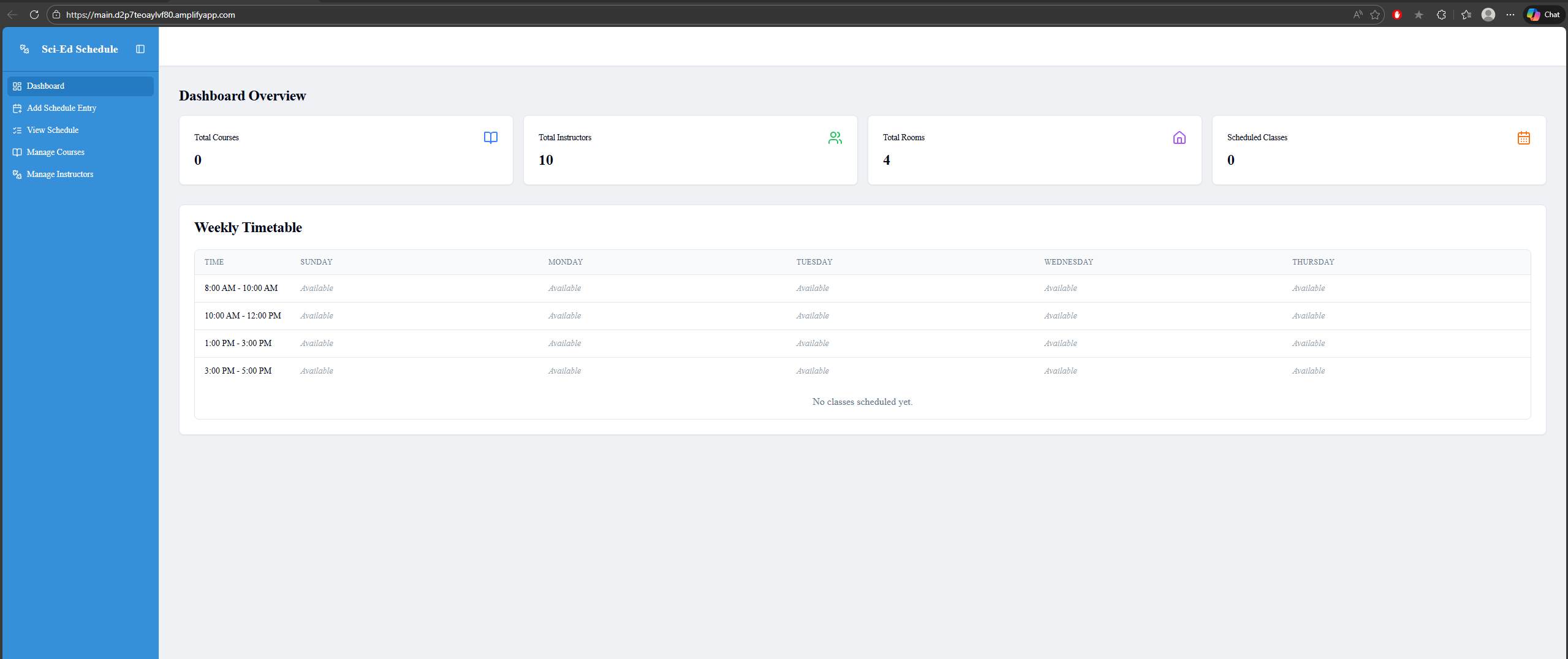


Figure 2-2

Creating a custom domain name for the front end is not a free tier service so we will avoid it

## 3. Setting up a CloudWatch Dashboard

Now that we deployed both the frontend and backend, we will set up some CloudWatch metrics so we can monitor our infrastructure.

We start by going to CloudWatch and creating a dashboard.

Now to monitor the important things, we will create 3 rows of metrics:

* Traffic
* Performance
* The Infrastructure Health

### a. Traffic metrics

We will add 3 widgets in this section, the first being Request Count, we create a new Line widget then choose Application ELB then Per App ELB Metrics then we choose Request Count for our ELB, and after that we change the statistic to sum and period to every minute thus achieving “**Requests/min**”, check figure 3-0.

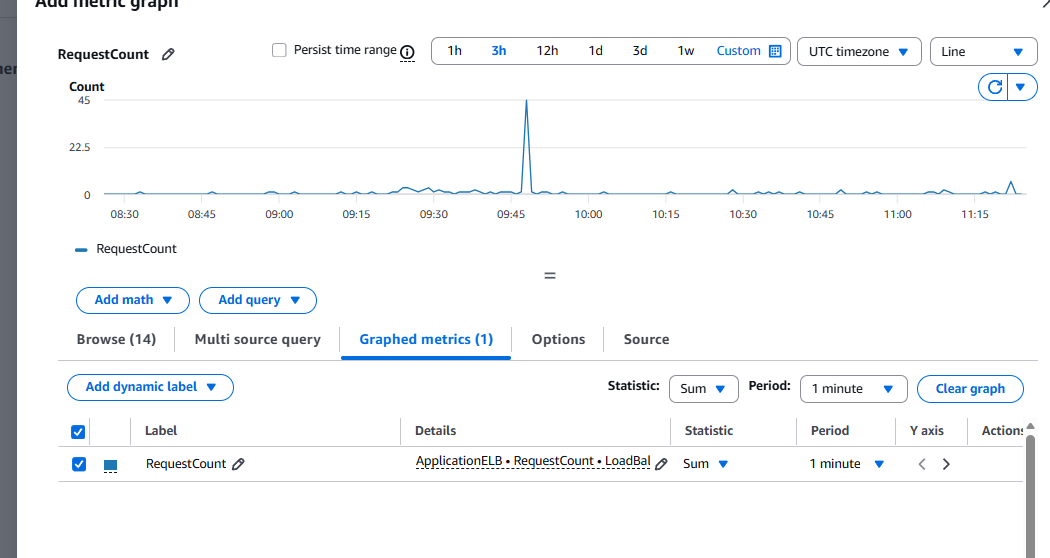


Figure 3-0

We do the same thing for widget 2 and 3 but with different metrics, 400 and 500 errors for bad requests and crashes, check figure 3-1.

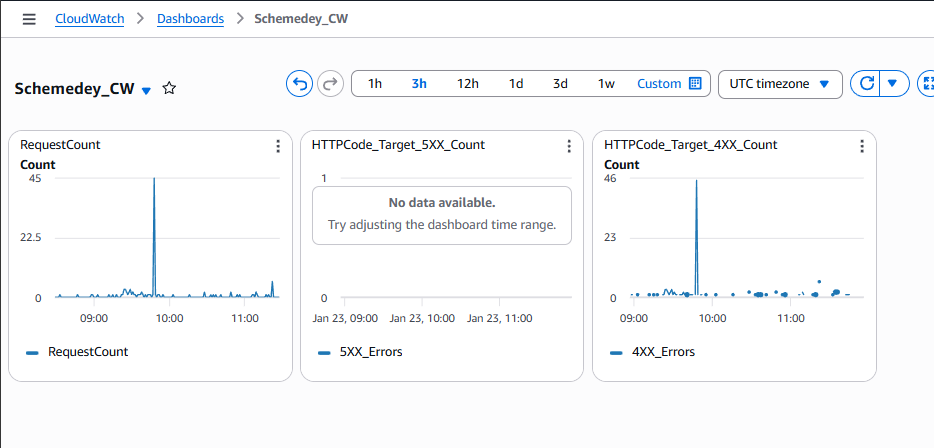


Figure 3-1

### b. Performance metric

This row has only one metric which is Application ELB 🡪 Load Balancer 🡪 Target Response Time “**avg**”, check figure 3-2.

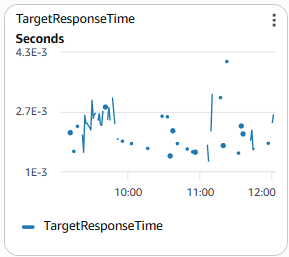


Figure 3-2

### c. the infrastructure Health

This row checks if the server is stressed by checking the CPU Utilization and the instance count.

Now we want the average CPU Utilization for every instance, but since we can’t pick them one by one because they might stop by the ASG when the traffic is low, we do this:

Ec2 🡪 By Auto Scaling Group 🡪 our ASG and the metric CPU Utilization.

And for the last one:

Auto Scaling 🡪 Group Metrics 🡪 Group in Service Instances for our ASG, check figure 3-3.

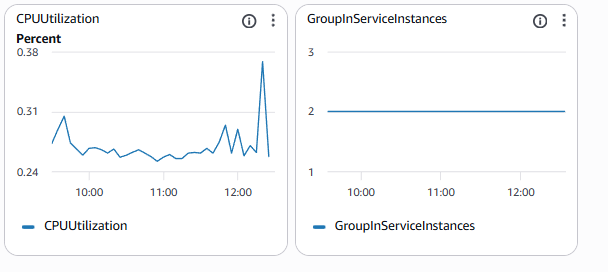


Figure 3-3

# HTU-Schedemy-Website (CI/CD)

**Scenario Overview:**

For this scenario we will modify scenario 2 and create a pipeline for the backend repo and its instance creation using GitHub actions and ansible, we want ansible to set up our server from scratch then we create an AMI from that server and replace the launch template AMI using github actions.

## Aws Setup

We’ll start by creating a key pair for GitHub actions to use we name it “pipeline-key”, check figure 1-0.

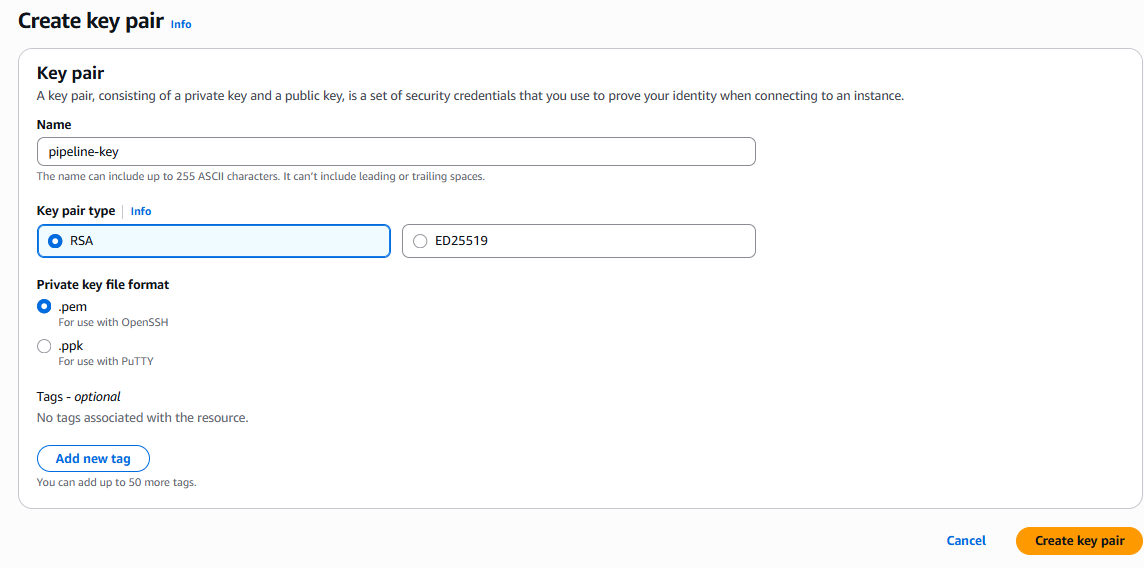


Figure 1-0

Then we copy the key content and put it in GitHub secrets.

Now we want to create a security group that GitHub actions will attach to the temporarily created ec2 instance, check figure 1-1.

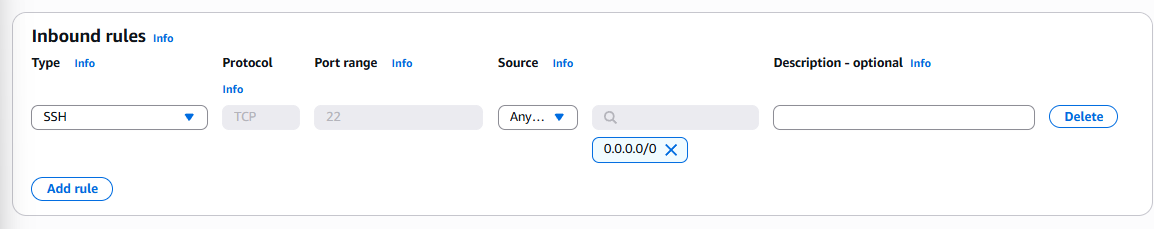


Figure 1-1

Now we copy the security group’s ID and put it in GitHub secrets.

We need to get the amazon Linux Ami ID as well from launch instance then we pick amazon Linux then we take the ID and put it in GitHub secrets, check figure 1-2.

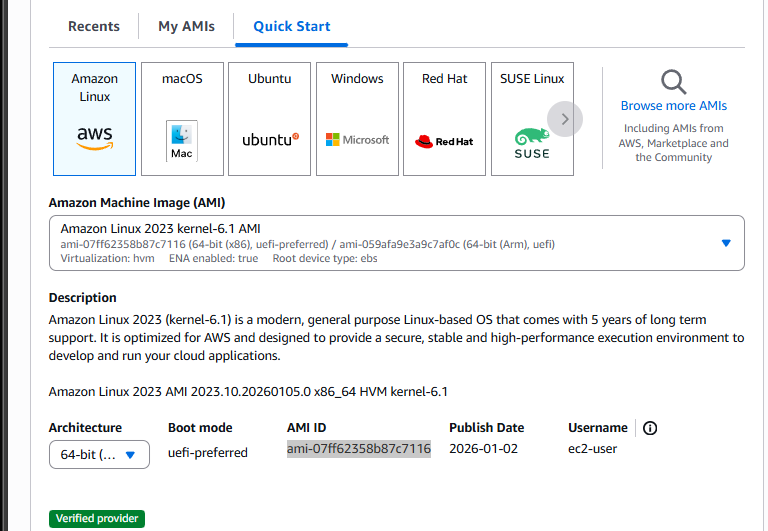


Figure 1-2

Now we will create an IAM user for GitHub actions named “**github-action**” and give it “**Amazon EC2 Full Access**”, check figure 1-3.

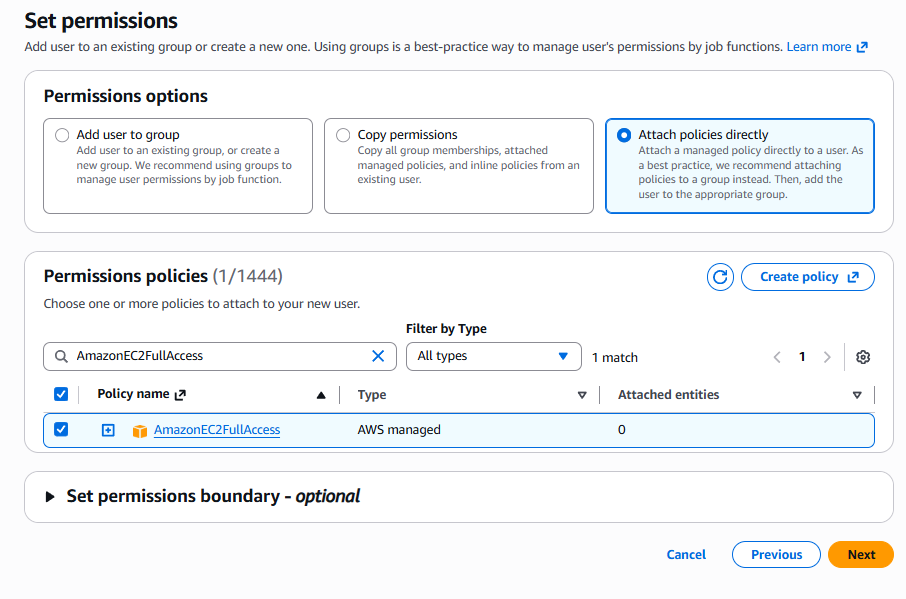


Figure 1-3

Now we want the access keys for that user, so we go to the user and then click on Security credentials then create an access key then we choose CLI, check figure 1-4.

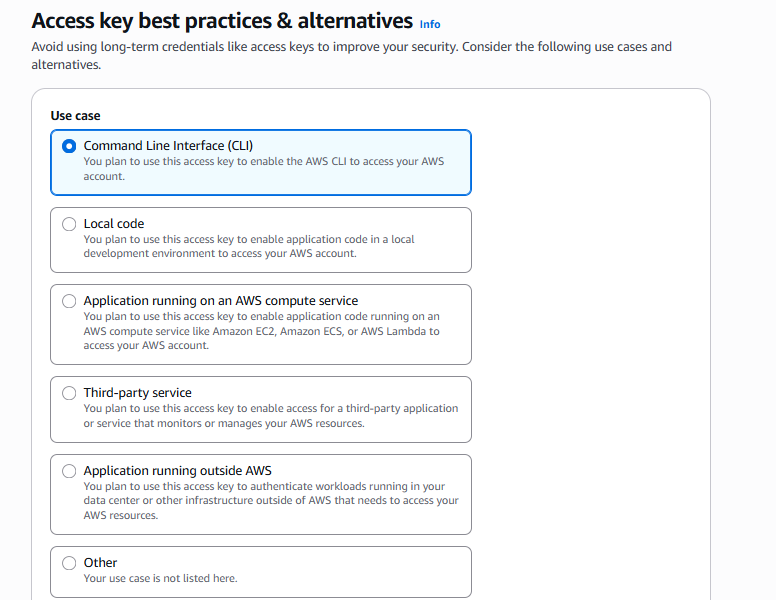


Figure 1-4

Now after creating the access keys, we copy them and paste them into GitHub secrets.

## Ansible YAML file

Ansible is going to be responsible for setting up the server from scratch.

Now we pull the backend repository to our computer then create ansible folder and we create the “**setup.yml**” file under it, this is where we will create our ansible YAML file.

Now for the hardest part, writing the YAML file we want to achieve a few things:

* Install java 17.
* Create the **“/opt/myapp**” directory for the application.
* Copy the Jar file to its destination
* Create a systemd Unit file.

For this scenario we will use an online YAML parser, we will start by putting a name for the set up then we make sure it always uses SUDO for our commands, then we set up our variables, check figure 2-0.

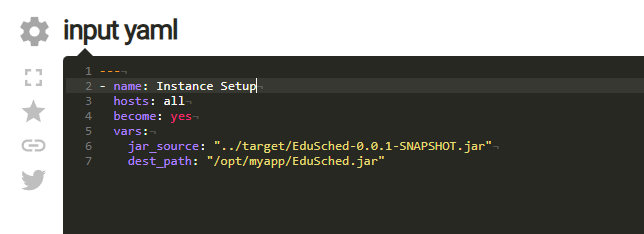


Figure 2-0

Then we will start with our first task which is installing java 17, check figure 2-1.

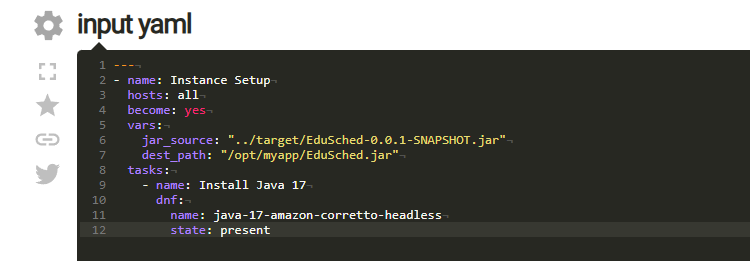


Figure 2-1

Now we want to create the **“myapp”** directory for the app give ec2-user it’s ownership and we give it permission “0700” thus only ec2-user can read, write and execute it.

Then we copy the jar file from the target directory under the main repo and put it in the server in the intended path we talked about above and give the same ownership permission we did above as well, check figure 2-2.



Figure 2-2

Now the last thing, we want to create the unit file so we can make the application run in the background as a systemd service, then we enable the service and not start it because we’re only creating the AMI here, the ASG will start it later and the service will run after it boots up, check figure 2-3

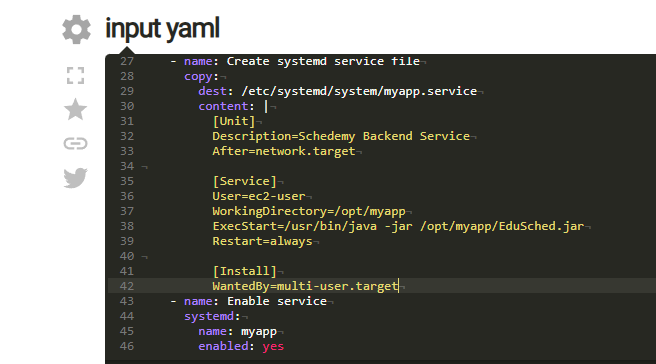


Figure 2-3

After that we will copy the YAML code and paste it in our “**setup.yml**” then push the code to the repo.

## GitHub Actions

Now we set up our GitHub workflow by clicking on GitHub actions in our backend repository, then we click on “set up a work flow yourself”.

This creates a “**main.yml**” file under workflows automatically, and this file is where we will write the YAML code for the brain of our CI/CD automation, check figure 2-4.



Figure 2-4

We will start by writing the YAML equivalent of “**start the github actions once a merge to the main branch occurs** “now we are assuming that any merge to the main branch is production ready, and we set the variables, check figure 2-5.

We create our main jobs, tell the system to run “ubuntu-latest” then we start steps for the job which is essentially tasks that we will run.

We start by using the action “**checkout** “to download the git repository to the runner “**the virtual machine hosted by github”,** then the action “**setup-java**” and choose java 17.

Then we compile and package the application using maven, check figure 2-6.

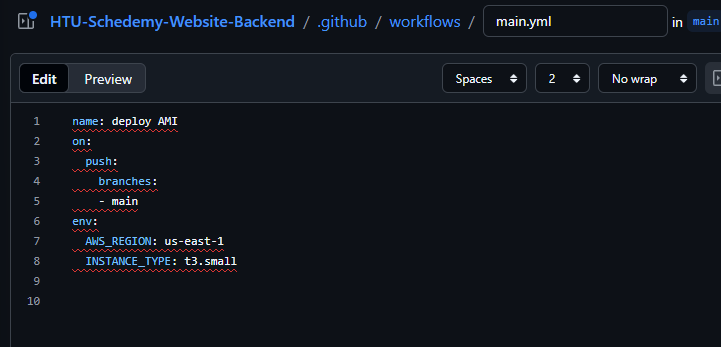


Figure 2-5

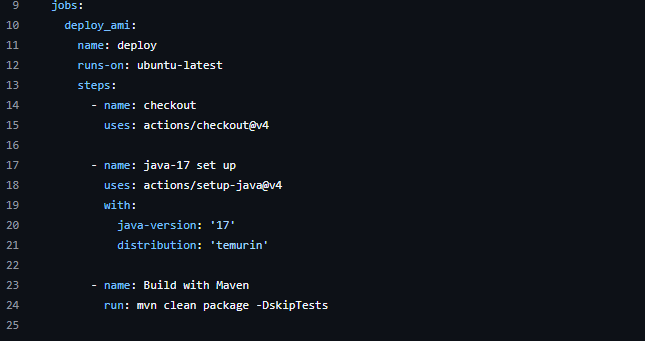


Figure 2-6

Now we use the action called “configure-aws-credentials” so we can use Aws commands using the IAM user we crated earlier.

Then we install Ansible on the runner, check figure 2-7.

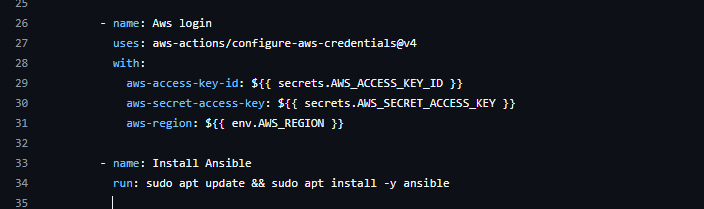


Figure 2-7

Now we set up the private key in the runner instance from GitHub secrets that we added earlier and then we give it “600” permissions so only the user can read and write.

Now we launch the instance using AWS cli command “**aws ec2 run-instances**” then we specify all the requirements for the instance such as the Amazon Linux id we put in GitHub secrets earlier as well as the security group id, and we put all of that in a variable so we can save the instance id and use it to get the public IP so we can ssh into the instance, check figure 2-8.

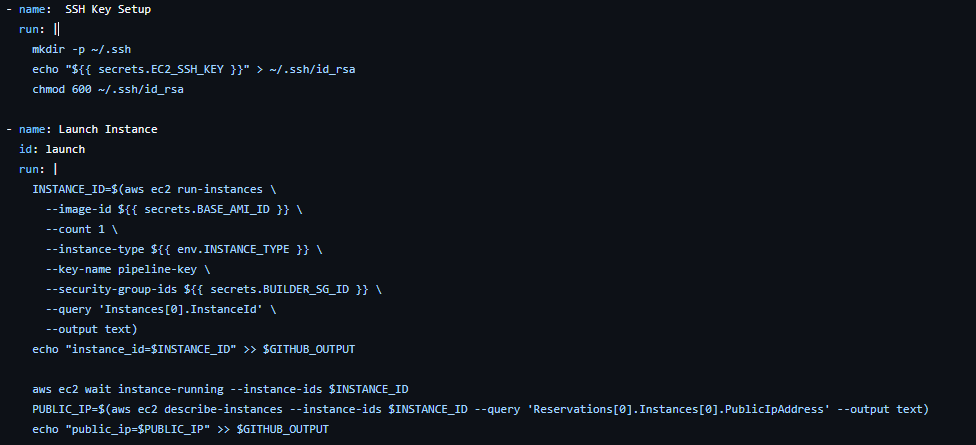


Figure 2-9

Now we want to run ansible, we create a loop to ssh into the server a few times until it boots up then the loop breaks, then we create the inventory.ini file with the server information.

Now since the server is set up, we want to create the AMI to we use the Aws command “**aws ec2 create-image**” then give it the requirements, and after that we terminate the instance, we created since we don’t need it anymore.

Lastly, we update the launch template with the new AMI and refresh the ASG, check figure 2-10.

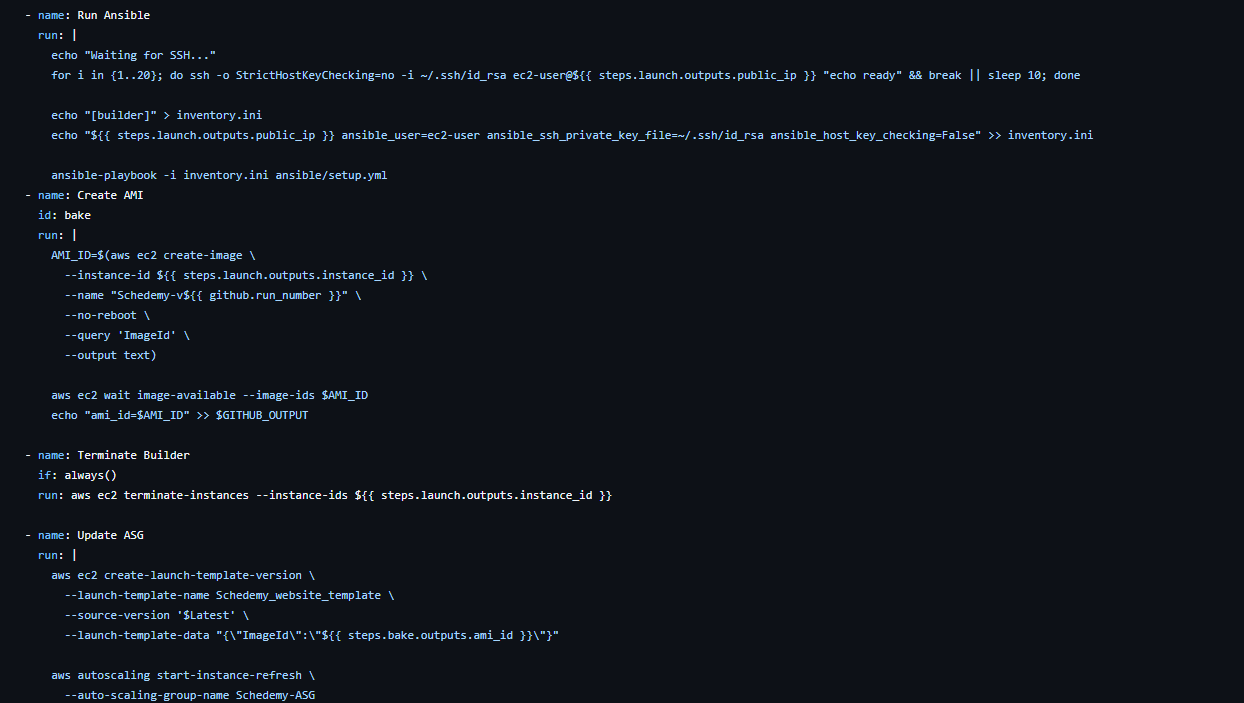


Figure 2-10

Now we commit the changes and then go to GitHub actions then “**main.yml**” then click on deploy, check figure 2-11.

After it completes the job we go check the instances in AWS , check figure 2-12.

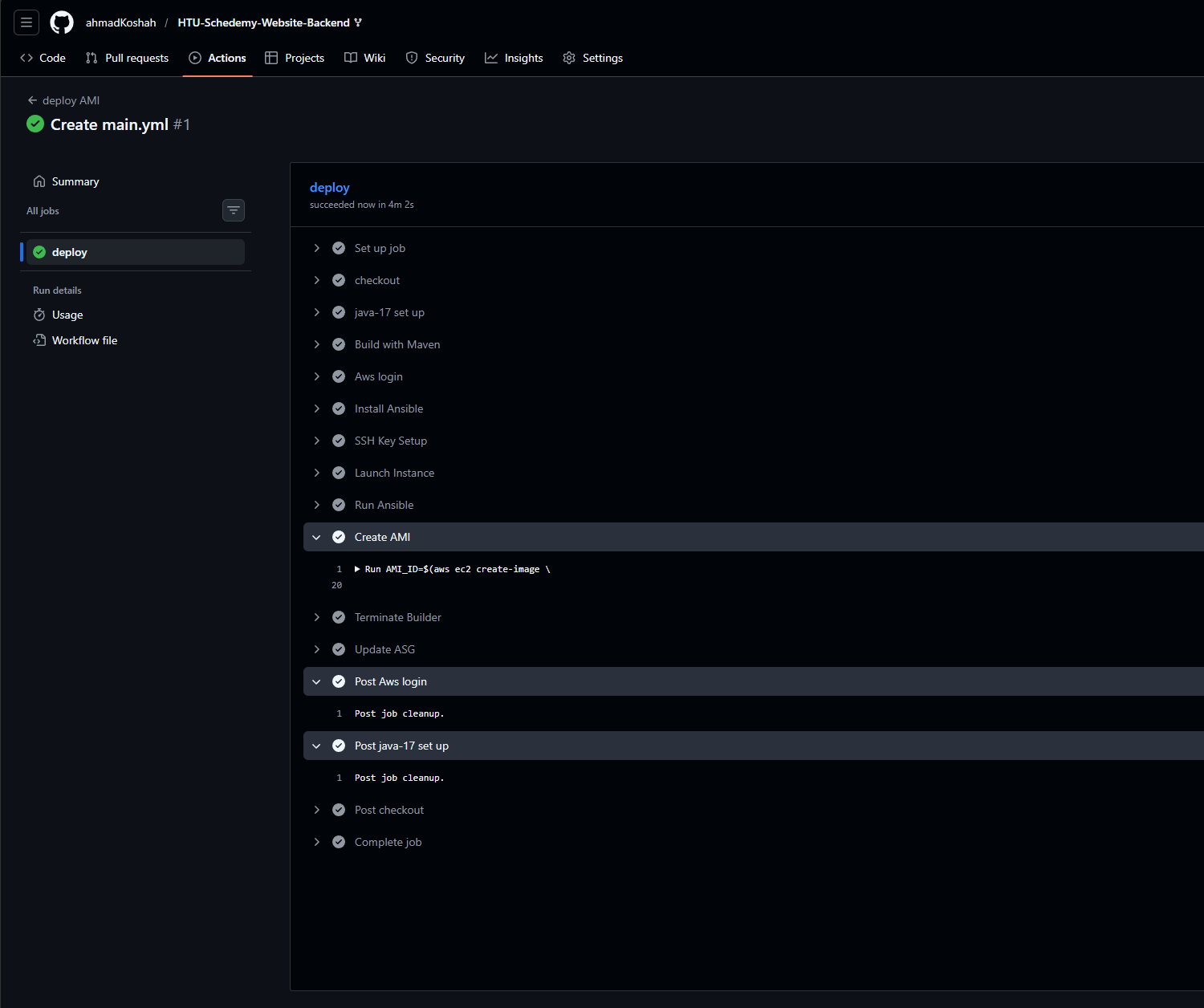


Figure 2-11

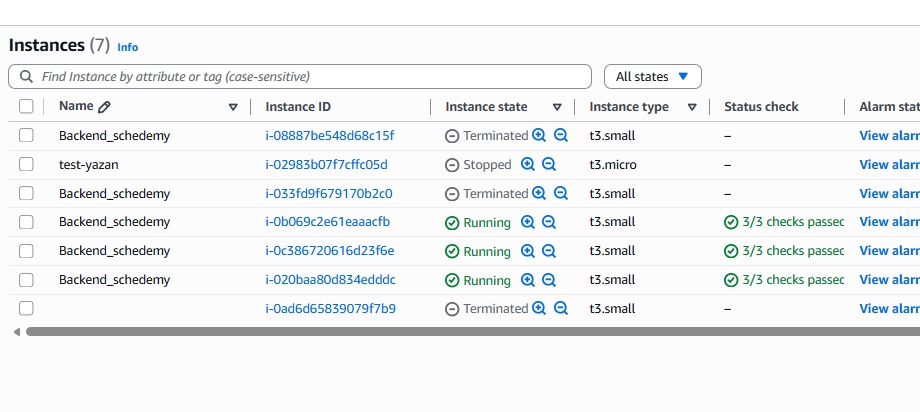


Figure 2-12

After that we change the readme file and then push it to GitHub, GitHub action deploys again then when it’s done, we check the ASG activity tab to see what happens, ASG launches two instances with the new application then after it passes the health checks it terminates the older instances, check figure 2-13.

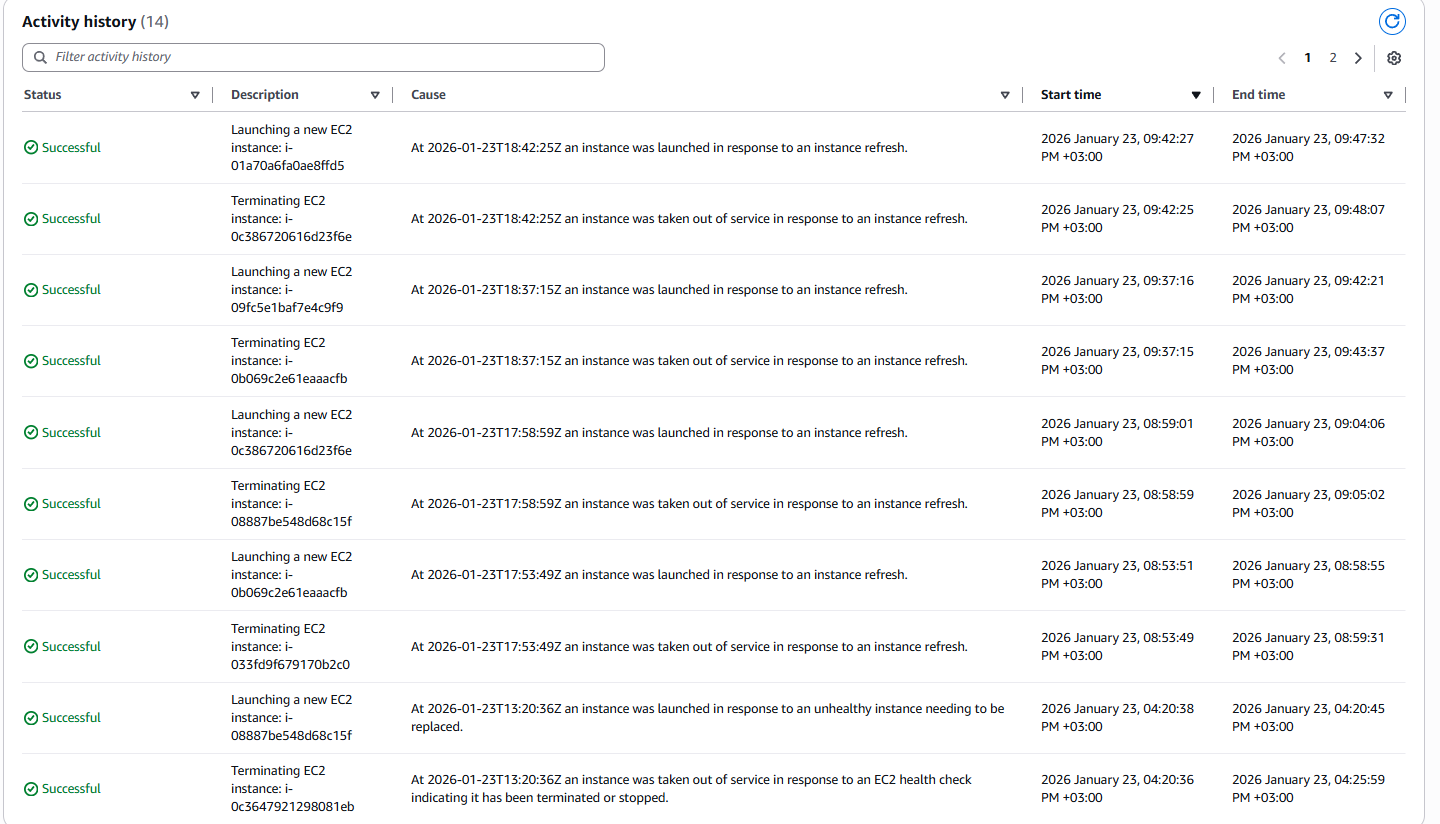


Figure 2-13

## 3.challenges and problems

Now I changed the test controller and added a /test/string that returns a string to check if the pipeline is working, and after all the process it didn’t work, so to diagnose the problem I connected to one of the instances using ssh and checked our application and noticed an issue.

The application name is not following the name scheme I used in the ansible YAML file which is “**EduSched.jar**” which means the ASG is still using the first version of the launch template AMI from scenario 2.

To solve this issue, we go to our ASG and then edit the launch template version to the latest instead of the default (1).

Another problem that happened after this is that the application on the new instances for the latest AMI didn’t work, the “**.service**” file was empty and the “**EduSched.jar**” file was empty with a size of 0 bytes.

To check the problem, I added log lines to debug the issue, and I found out that ansible sets up the server perfectly, but the issue occurs when the AMI is created.

After searching for a while and trying different methods, it turns out that we need to add a wait/sleep line after the instance is set up and before creating the AMI which solved the issue.

Also, I forgot to start the SYSTEMD service in the ansible YAML file, it only enabled the service but didn’t start it so I added the line “**state: started**”.

After successfully diagnosing and fixing those issues, the pipeline worked perfectly and the call to /test/string returned the string I added, check figure 3-0.



Figure 3-0

# Pixel-Fed photo app

**Scenario Overview:**

This scenario consists of deploying a dockerized on an AWS instance.

## 1.Server Set up

We’ll start by creating an AWS ec2 instance with the following requirements:

* Amazon Linux 2023
* C7i-flex.large 4GiB Ram 2vCPU
* Ports 22 and 8080 for ssh connection and the application port
* Ssh key

After creating the instance, we connect via ssh and initiate an update and install the necessary packages including (git, docker, curl), check figure 1.0.



Figure 1.0

Then we’ll enable and start the docker service using “systemctl” command.

Then we install docker-compose binary file from the official GitHub repo following their README file then we check the docker-compose version, check figure 1.1.

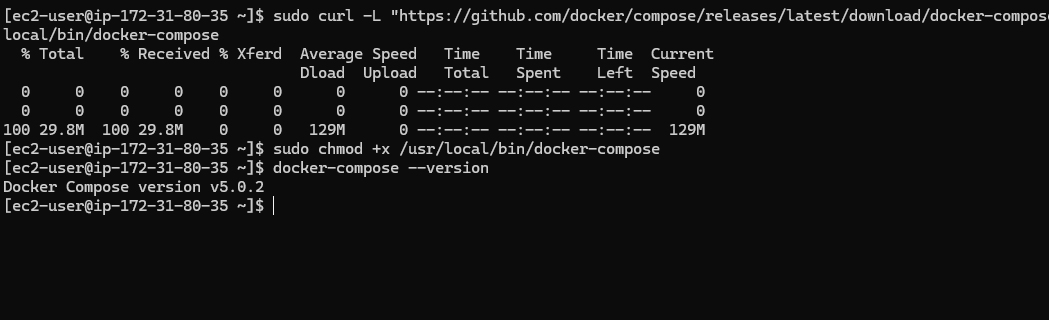


Figure 1.1

Now we clone the GitHub repository and check the files, check figure 1.2.

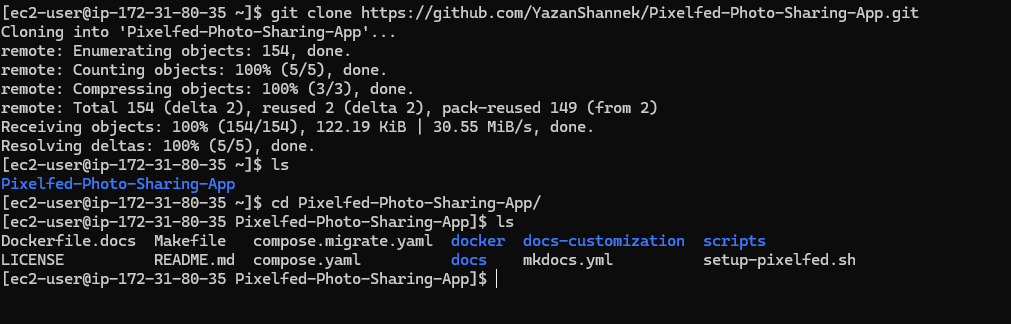


Figure 1.2

Now we’ll copy the “.**env.production**“file and rename it to “**.env**” then edit the variables like app domain and db password , check figure 1.3.

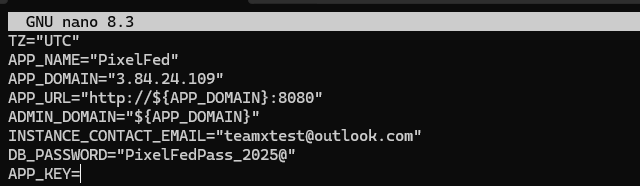


Figure 1.3

Now we run the docker compose up command and check , check figure 1.4.

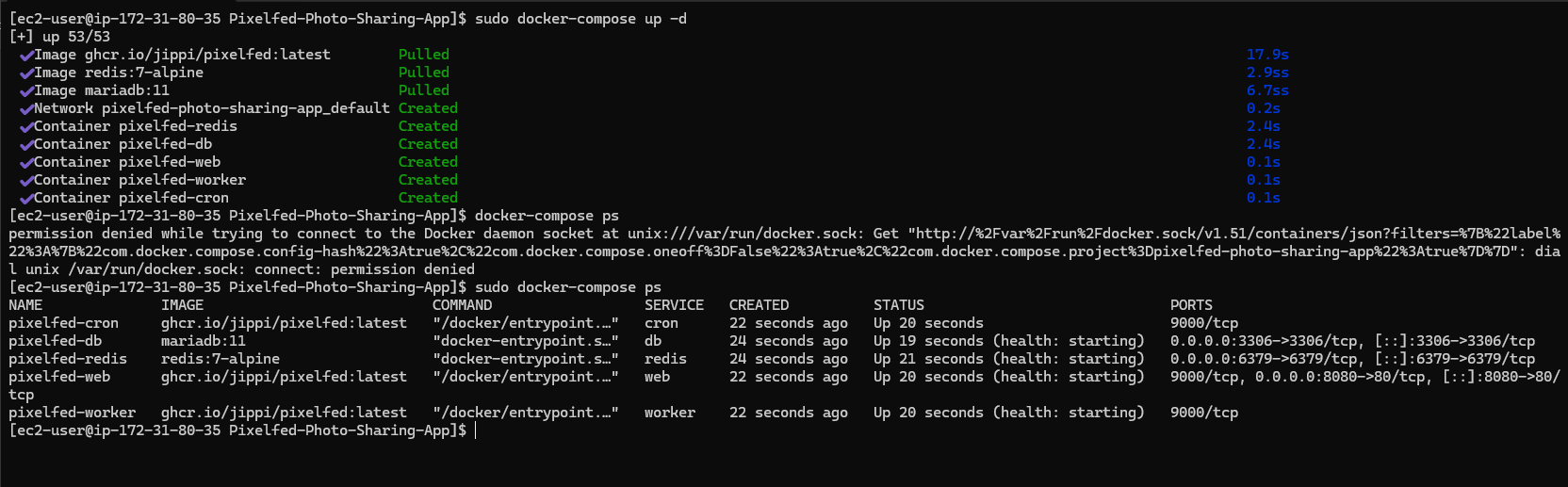


Figure 1.4

Now we run the setup file after giving it +x execute permission, check figure 1.5

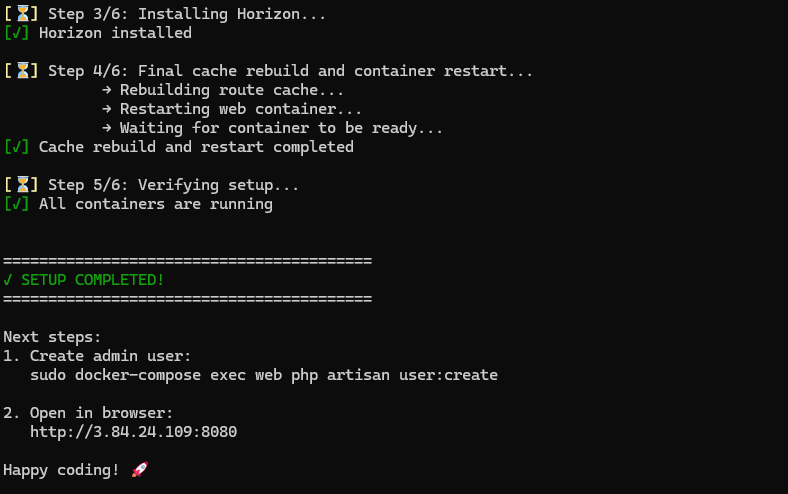


Figure 1.5

Now after that we will create an admin user , check figure 1.6

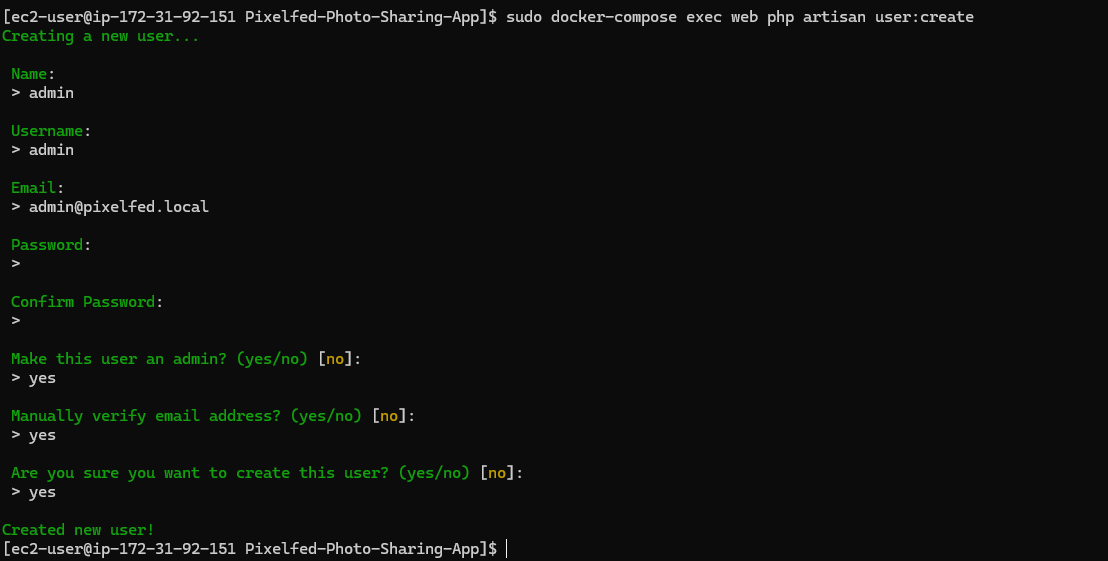


Figure 1.6

Now we check the website , check figure 1.7 and 1.8.

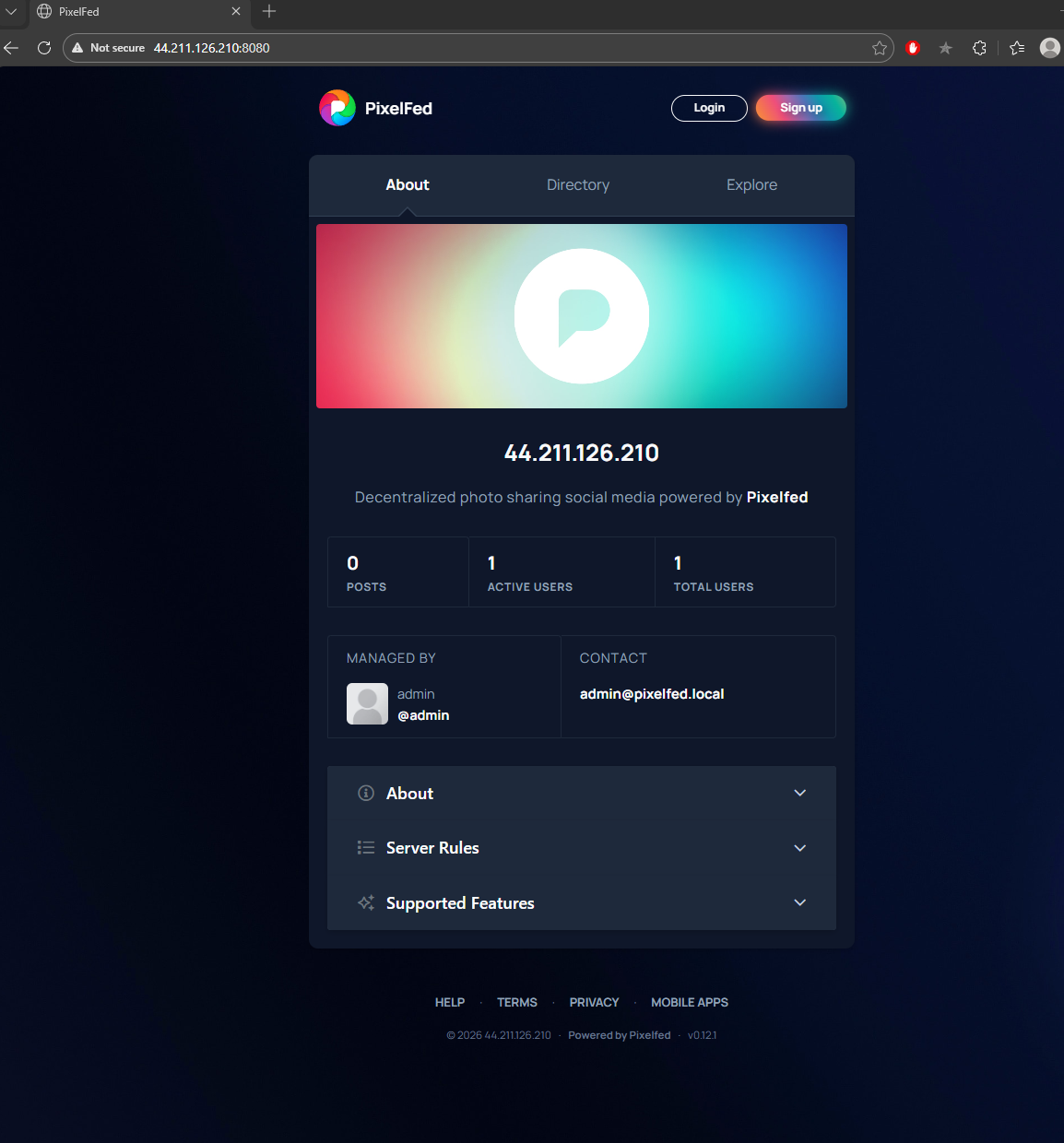


Figure 1.7

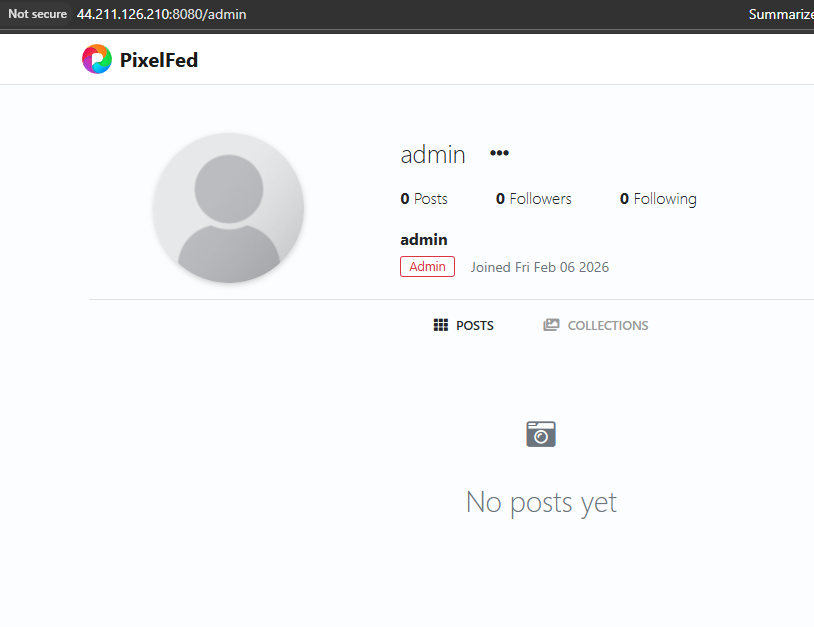


Figure 1.8