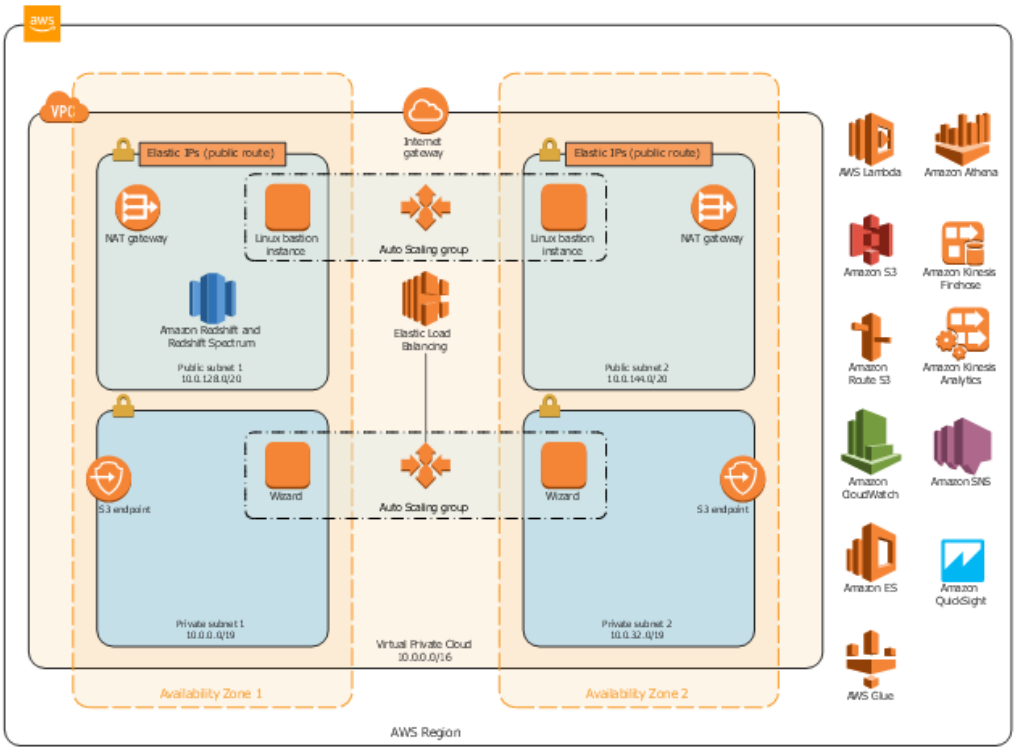
RMIT UNIVERSITY

System Deployment and Operations

Semester 2022B

Assignment 2

DATE: 28/08/2022



Author: Ngo My Quynh

ID: s3836322

Created date: 28/08/2022

Last modified: 28/08/2022

Acknowledgment: I acknowledge that all the work is done by me

Table of contents

Step 1 - AWS EC2 instance setup

- As a requirement, we will create two EC2 instances, one hosted on the Jerkins server and the Tomcat server hosted on the other. Two of the instance will have to satisfy the following requirements.
    - Having the same VPC on the same subnet and the same AZ
    - Having the same key
    - Attached to specific Elastic IP: We are doing this because, in the pipeline, we have the part that will ssh to the other container from the original one, and making the IP Address consistent will eliminate the changing of the IP, which leads to failing execution pipeline.
    - Having the same security group which contains the inbound and outbound rules that
        - Inbound rules: enable ssh, allow connection on HTTP from anywhere, and open port connection for 8080 from anywhere. We open the port so that the Jenkins and the web server will be exposed later.
        - Outbound rules: allow all traffic.

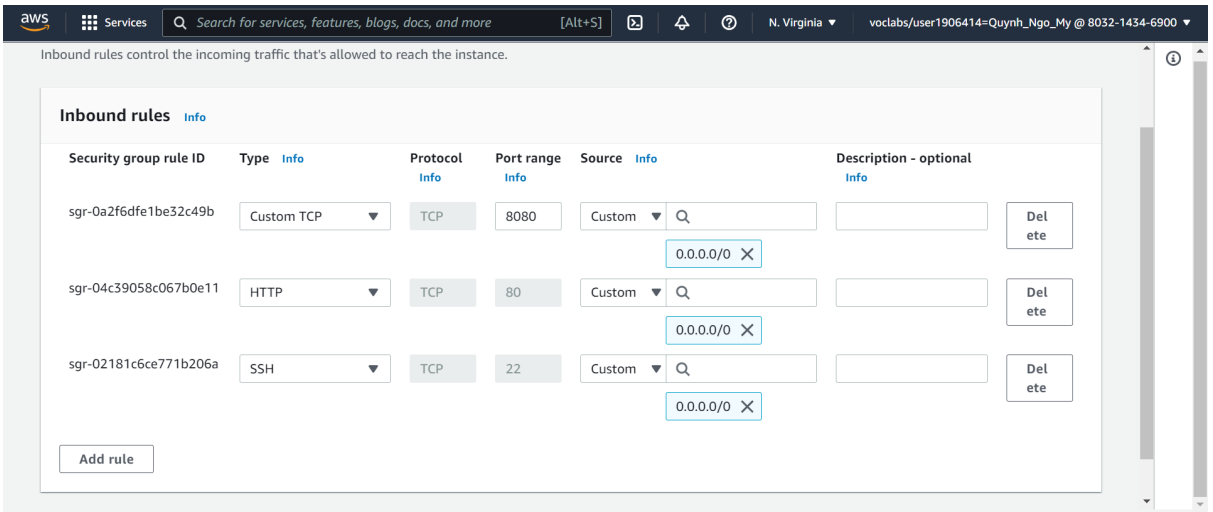


Figure 1.1: Inbound rules configuration

- Following all the requirements below, we will have two instances that have the configuration below

Name	Jenkins	Tomcat
AMI	Ubuntu Server 22.04 LTS (HVM)	Ubuntu Server 22.04 LTS (HVM)
Type	t2.micro	t2.micro

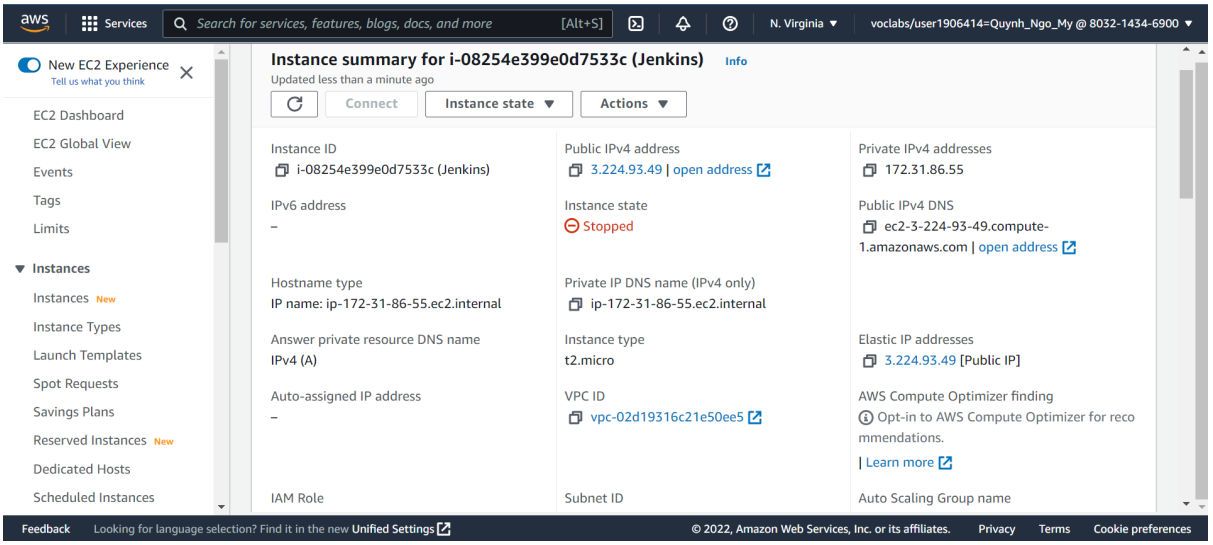


Figure 1.2.1: Overview of the instance Jenkins

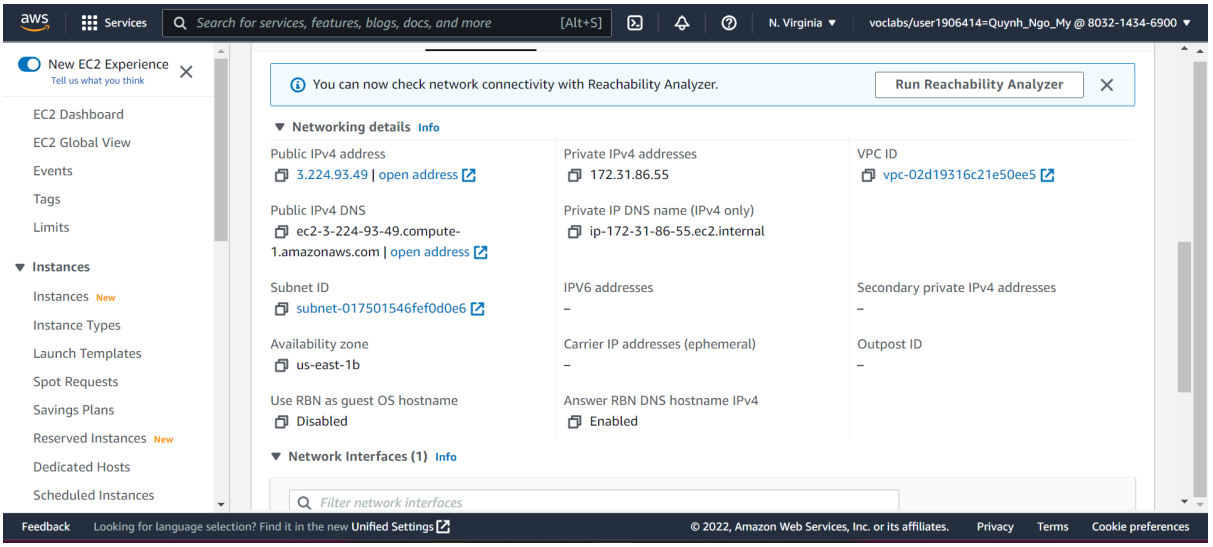


Figure 1.2.2: Overview of the instance Jenkins

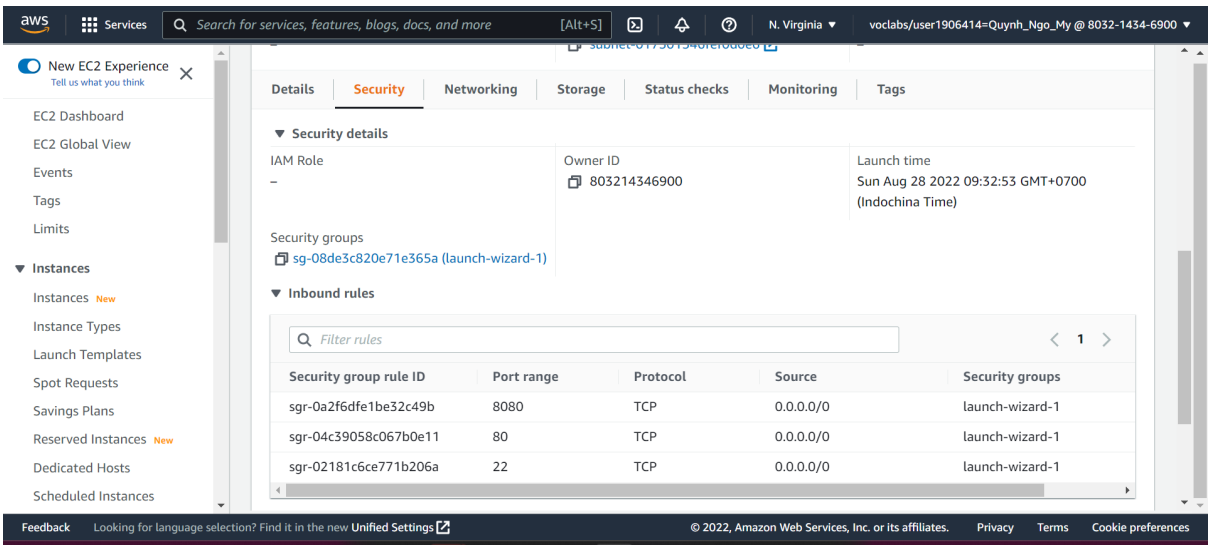


Figure 1.2.3: Overview of the instance Jenkins

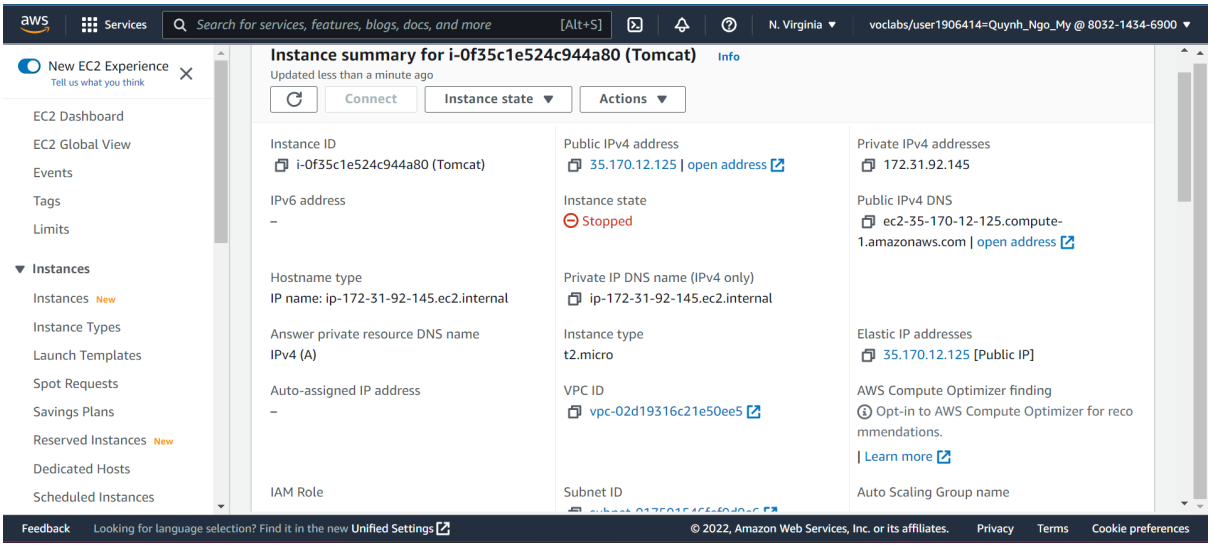


Figure 1.3.1: Overview of the instance Tomcat

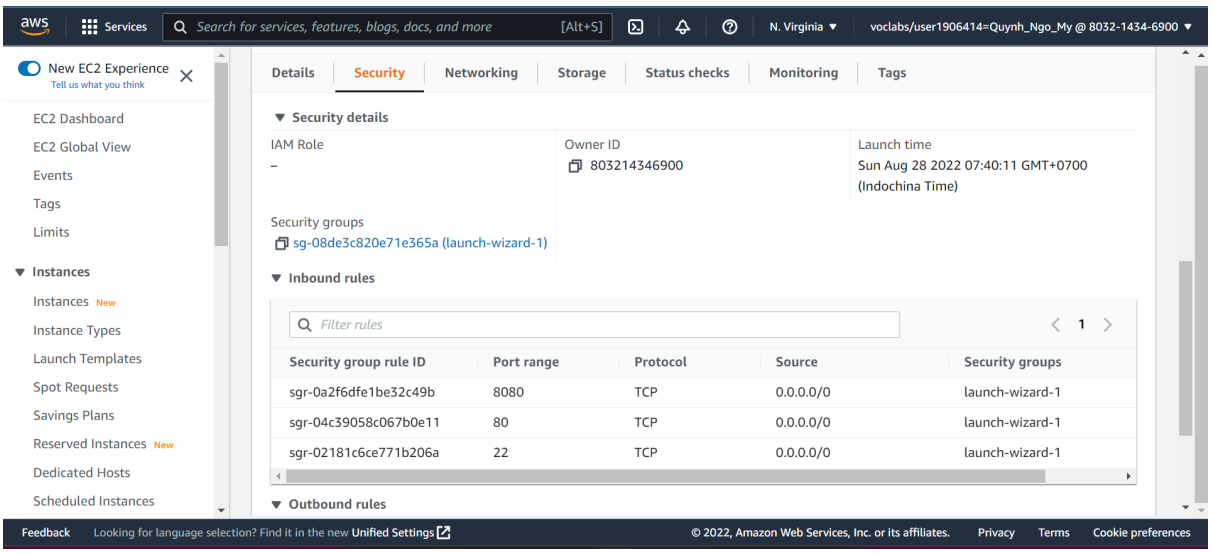


Figure 1.3.2: Overview of the instance Tomcat

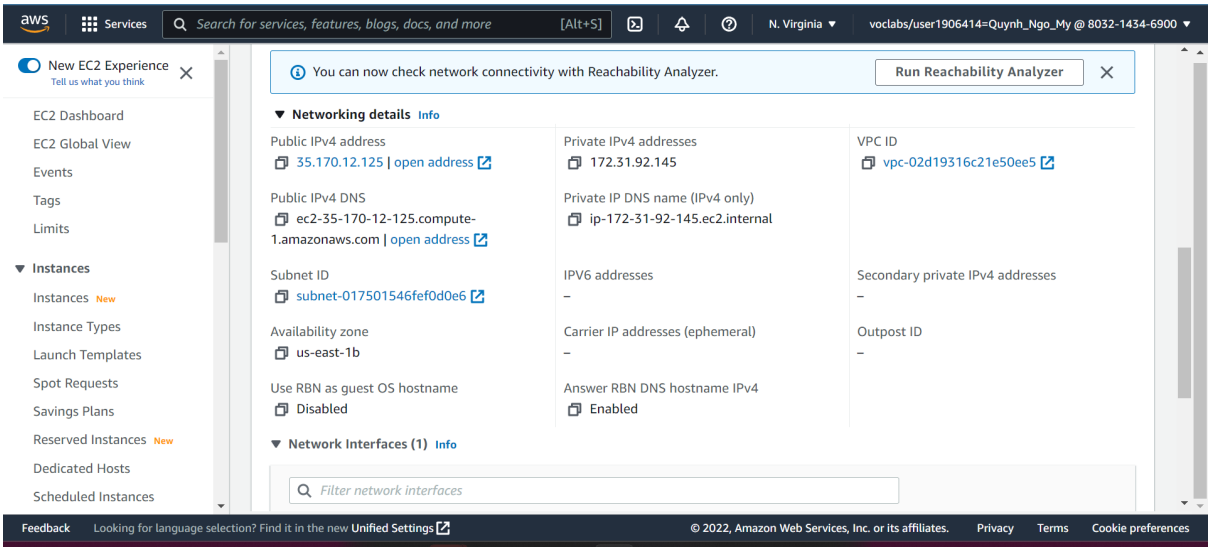


Figure 1.3.3: Overview of the instance Tomcat

- After creating the EC2 instance, we have to make sure that those two instances can ssh to each other since, in the CI/CD pipeline that we are building, we have the step that needs to ssh to the other one using the ssh key.
- First, we go into the EC2 Instance Jenkins to generate the key.

```
cd ~/.ssh && ssh-keygen
```

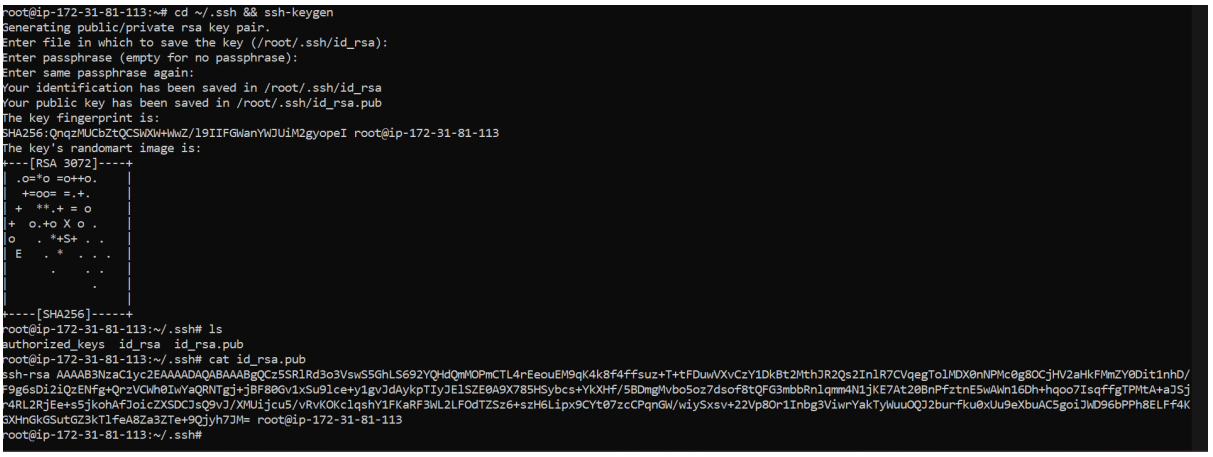


Figure 1.4: Generate the key pair

- After we have the key in the ssh folder, copy that key and put it in the **authorized_keys** file of the Tomcat instance. The file in the final should look like this

Figure 1.5: authorized_keys file of Tomcat server

- Doing the same steps as for Tomcat server so that both ec2 instances can have ssh to others. In the below example, we will try to ssh to the Tomcat server from the Jerkins instance.

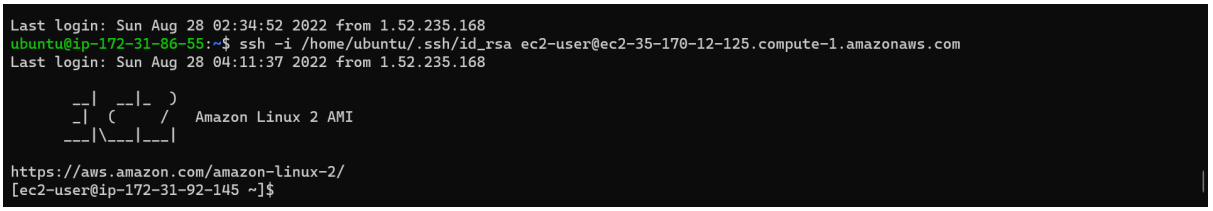


Figure 1.6: Result of ssh from Jerkins server to Tomcat using ssh key

- Since we are using git to pull our project and place it in the instances. We will also copy those keys and put them into our GitHub accounts

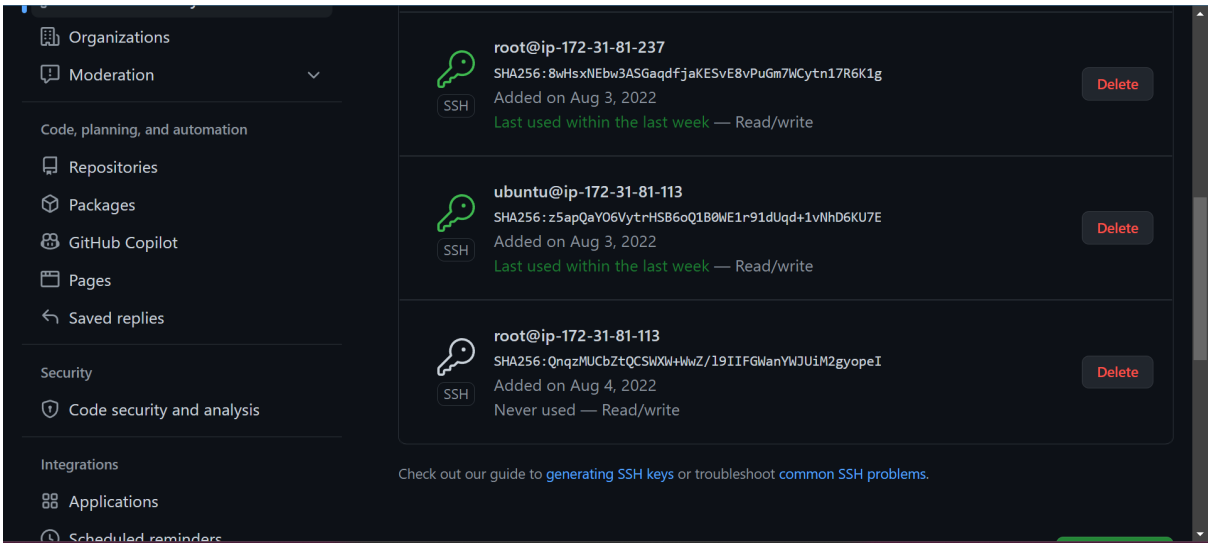


Figure 1.7: Put the SSH key into Github

- By doing that, we can clone our code without having any permission issue

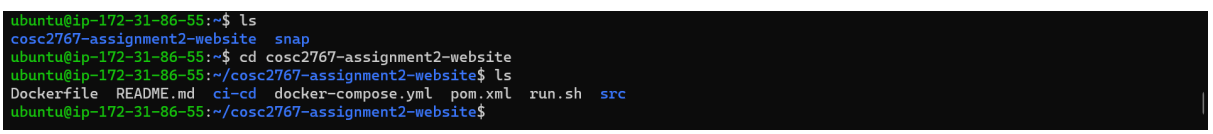


Figure 1.8: Checking if we can clone our code into our EC2 instance

- Next, we will need to configure the environment that we can run the docker in the instance by executing the docker.sh scripts that place in the ci-cd folder of the repository. The scripts will contain the following steps
    1. Install the docker package
    2. Start the docker daemon
    3. Set the permission for the user so that it can overwrite the docker.sock file to prevent permission issues when using the docker

```
sudo snap install docker
sudo snap start docker
sudo chmod 666 /var/run/docker.sock
```

Step 2: Setup the pipeline

- After setting up all the necessary things for our pipeline, we are going to build it using the docker-compose.yml files. This file will build the image based on the Dockerfile, exposing the port and declaring the environment variable so that we can later use it in our ci/cd pipeline.

```
docker-compose up -d --build
```

```
Step 5/11 : RUN jenkins-plugin-cli -f /usr/share/jenkins/ref/plugins.txt
 ---> Using cache
 ---> a67106bf970f
Step 6/11 : COPY casc.yaml /var/jenkins_home/casc.yaml
 ---> Using cache
 ---> ef898a692ee4
Step 7/11 : USER root
 ---> Using cache
 ---> ecbad3baee74
Step 8/11 : RUN apt-get update && apt-get install -y wget
 ---> Using cache
 ---> f2e0d157e1be
Step 9/11 : RUN cd /opt &&     wget https://dlcdn.apache.org/maven/maven-3/3.8.6/binaries/apache-maven-3.8.6-bin.tar.gz &&     tar -x
vzf apache-maven-3.8.6-bin.tar.gz &&     mv apache-maven-3.8.6 maven
 ---> Using cache
 ---> 4c4109bdb759
Step 10/11 : RUN curl -fsSLO https://get.docker.com/builds/Linux/x86_64/docker-17.04.0-ce.tgz   && tar xzvf docker-17.04.0-ce.tgz   &
& mv docker/docker /usr/local/bin   && rm -r docker docker-17.04.0-ce.tgz
 ---> Using cache
 ---> 6ed9c1df71f0
Step 11/11 : EXPOSE 8080
 ---> Using cache
 ---> 33d07c0c7c10
Successfully built 33d07c0c7c10
Successfully tagged ci-cd_web:latest
Starting ci-cd_web_1 ... done
ubuntu@ip-172-31-86-55:~/cosc2767-assignment2-website/ci-cd$ docker ps
CONTAINER ID   IMAGE       COMMAND                  CREATED       STATUS         PORTS
                                NAMES
9780b38e048a   ci-cd_web   "/usr/bin/tini -- /u…"   3 hours ago   Up 6 seconds   0.0.0.0:8080->8080/tcp, :::8080->8080/tcp, 0.0.0.0:5
0000->50000/tcp, :::50000->50000/tcp   ci-cd_web_1
ubuntu@ip-172-31-86-55:~/cosc2767-assignment2-website/ci-cd$
```

Figure 2.1: Docker container run on port 8080

- Then, we can access the Jenkins server through the port 8080 of the instance

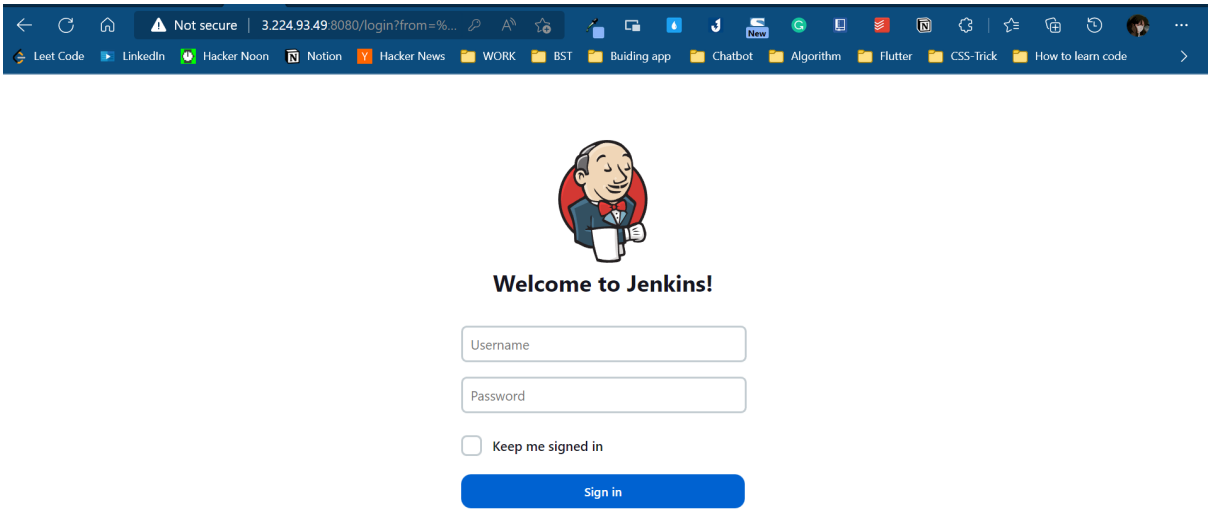


Figure 2.2: Jenkins server access on port 8080

- Before starting the pipeline, there are some explanations here. Instead of configuring Jenkins using UI, we will approach it through the concept: Configuration-as-code, provided by Jerkins. Overall, this concept introduces a way to help us spend less effort interacting with the UI by providing necessary files such as casc.yaml, plugins.txt, and Jenkinsfile so that everything we need is already set up and installed when we start the Jenkins server. There are many benefits, but the greatest one is that we can reuse these files if we host another Jenkins with the same configuration without doing it repeatedly.

- With this approach, we will look into 2 files to understand the set up

casc.yml	

`jenkins:`
`  securityRealm:`
`  local:`
`  allowsSignup: false`
`  users:`
`  - id: ${JENKINS_ADMIN_ID}`
`  password: ${JENKINS_ADMIN_PASSWORD}`
`  authorizationStrategy:`
`  globalMatrix:`
`  permissions:`
`  - "Overall/Administer:admin"`
`  - "Overall/Read:authenticated"`
`  remotingSecurity:`
`  enabled: true`
	The casc.yml file is the file that will declare all the configurations for Jenkins.
	
	We will divide this file into four parts to have a better understanding. The first section will set up the account, some policies, and permission for whether users can perform such action. Here, the user with the id admin can have the administrator role, which can perform anything in the Jenkins application, such as creating jobs, viewing other accounts, and building pipelines.
	
	The second part is the tool. Instead of configuring the tools through the UI in

```
security:
  queueItemAuthenticator:
    authenticators:
    - global:
        strategy:
triggeringUsersAuthorizationStrategy
unclassified:
  location:
    url: http://server_ip:8080/

tool:
  maven:
    installations:
      - home: "/opt/maven"
        name: "maven3"
  jdk:
    installations:
      - home: "/opt/java/openjdk"
        name: "jdk11"

credentials:
  system:
    domainCredentials:
    - credentials:
      - usernamePassword:
          scope: GLOBAL
          id: "tomcat_deployer"
          username: "s3836322"
          password: "s3836322"
          description: "Tomcat deployer"
      - usernamePassword:
          scope: GLOBAL
          id: "dockerhub_deployer"
          username: ${DOCKERHUB_ID}
          password: ${DOCKERHUB_PASSWORD}
          description: "Dockerhub deployer"
jobs:
  - script: >
      pipelineJob('default-agent') {
        definition {
          cps {
            script("""\
```

the “Global Tools Configuration” section, we can define it here. In this part, we will define the environment path for maven and JDK so that the ci/cd can later build and pack the maven project. We have the option to install it from the source, but we will hardly control where the path will place, and there is also some permissions issue. Therefore, in Dockerfile, it will help us set up this environment; we need to declare its environment path in this file.

The third part is credentials. Here we will have two credentials: the first from the tomcat server and the second from the docker hub. Since we will deploy the image to the docker in the later step, we need to provide the credential.

The fourth part will mainly be about our ci/cd pipeline. First, we declare the name for it and then config the job using the same script (The Jenkinsfile). First, the script will get the tools which is the global tool configuration that we declare above, so that the program can find the Maven path and the JDK path. Next, we will create the trigger to check if any commit is pushed in our central repository; it will trigger the ci/cd to build over again. The following part declares the process in the pipeline, which contains:

- Check out the branch we are going to build the war package.
- Pack the source code with the command line **mvn package** to produce the war file that is placed in the target folder
- Pack the whole project using a Dockerfile already defined in the source code. This file uses the latest image of tomcat and then copies the war file, places it in the /webapp/ folder, and opens port 8080 for the web application to be accessible.
- Tag the image with the mark

```
pipeline {
  agent any
  tools {
    maven 'maven3'
    jdk 'jdk11'
  }
  triggers {
    pollSCM ''
  }
  stages {
    stage('checkout') {
      steps {
        git branch: 'main', url:
'https://github.com/s3836322/cosc2767-assignme
nt2-website.git'
      }
    }
    stage('Execute Maven') {
      steps {
        sh 'mvn package'
      }
    }
    stage('Docker Build and
Tag') {
      steps {
        sh 'docker build -t
samplewebapp:latest .'
        sh 'docker tag
samplewebapp s3836322/samplewebapp:latest'
      }
    }
    stage('Publish image to
Docker Hub') {
      steps {
        withDockerRegistry([
credentialsId: "dockerhub_deployer", url: ""
]) {
          sh  'docker push
s3836322/samplewebapp:latest'
        }
      }
    }
```

"latest" and push the image in the Dockerhub with the already declared above credentials.

- Then we will ssh to the instance where we will host the webster and build the application based on the latest image on Dockerhub. We will remove the existing container if it has, creating the new container with the prefix "tmp" and renaming it to the original one.
- The "true" command is placed in the final to ignore errors from the execution in the shell. Since the first time, we do not have the container yet. Therefore, it will throw out the exception which makes our pipeline fails.

```
                    stage('Run Docker container
on Tomcat hosts') {
                      steps {

sh 'ssh -tt -i /home/ubuntu/.ssh/id_rsa
ec2-user@ec2-35-170-12-125.compute-1.amazonaws
.com -y "docker stop tomcat ; docker rm tomcat
; docker run --name tomcat_tmp -d -p 8080:8080
s3836322/samplewebapp:latest ; docker rename
tomcat_tmp tomcat || true"'



                      }
                    }
                  }
              }""".stripIndent())
          }
        }
      }
```

Dockerfile of jenkins	

`FROM jenkins/jenkins`
 	
`ENV JAVA_OPTS`
`-Djenkins.install.runSetupWizard=false`
`ENV CASC_JENKINS_CONFIG`
`/var/jenkins_home/casc.yaml`
`COPY plugins.txt`
`/usr/share/jenkins/ref/plugins.txt`
`RUN jenkins-plugin-cli -f`
`/usr/share/jenkins/ref/plugins.txt`
`COPY casc.yaml /var/jenkins_home/casc.yaml`
 	
`USER root`
 	
`RUN apt-get update && apt-get install -y wget`
`RUN cd /opt && \`
`wget`
`https://dlcdn.apache.org/maven/maven-3/3.8.6/bi`
	This file will be used for building up the container, which uses the official image that Jenkins provides.
	
	The first process will disable the setup wizard by passing the Jenkins.install.runSetupWizard to fail through the JAVA_OPTS environment variable. Using JCasC eliminates the need to show the setup wizard. Next, we will set the environment variable CASC_JENKINS_CONFIG to the path where we will place the casc.yml file so that, later on, the Configuration-as-code plugins can know where to read it. Then below, we will copy the casc.yml we define in the source code and replace it with the existing one that places in the CASE_JENKINS_CONFIG beside overwritten that; we have also overwritten the plugins.txt, which contains all the necessary plugins that

```
naries/apache-maven-3.8.6-bin.tar.gz && \
    tar -xvzf apache-maven-3.8.6-bin.tar.gz &&
\
    mv apache-maven-3.8.6 maven

RUN curl -fsSLO
https://get.docker.com/builds/Linux/x86_64/dock
er-17.04.0-ce.tgz \
  && tar xzvf docker-17.04.0-ce.tgz \
  && mv docker/docker /usr/local/bin \
  && rm -r docker docker-17.04.0-ce.tgz

EXPOSE 8080
```

we are going to use in the pipeline without installing it on UI.

Next, we will switch to the root permission to set up the maven and the java environment so that the latter can be used in the casc.yml. Besides that, we also install the docker package since it is unavailable in the EC2 instance.

Finally, we expose the web application on port 8080.

- After understanding the flow, let's go into the Jenkins server to see the configuration pipeline we declared and build it.

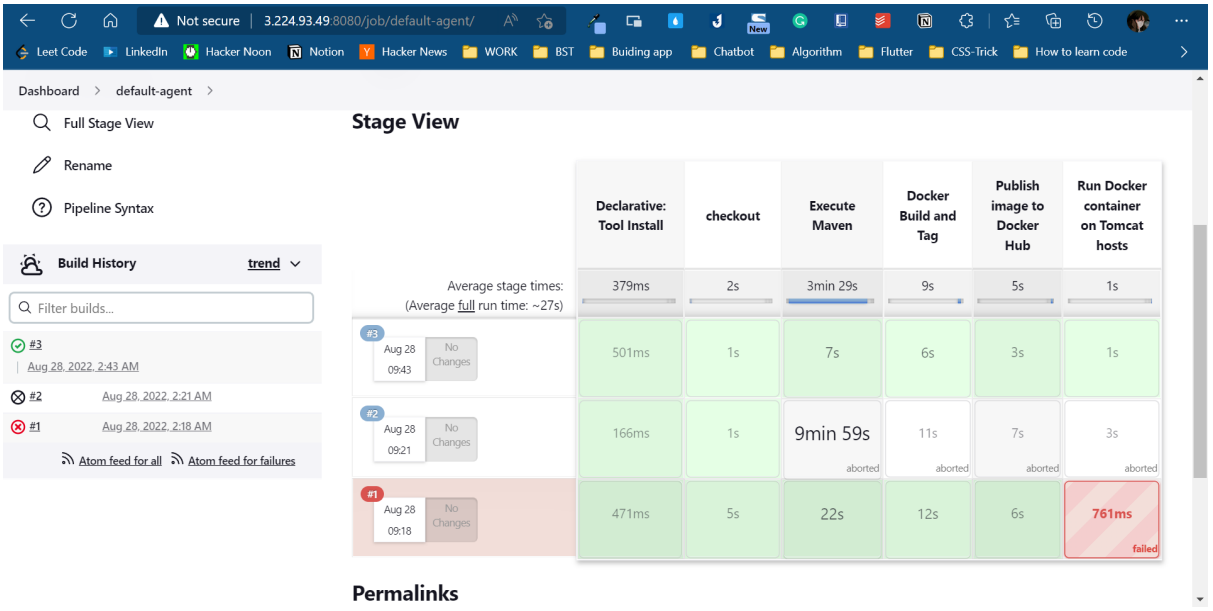


Figure 2.3: Pipeline process on the dashboard

- The pipeline build successfully, and the web application should be accessed on the other instance

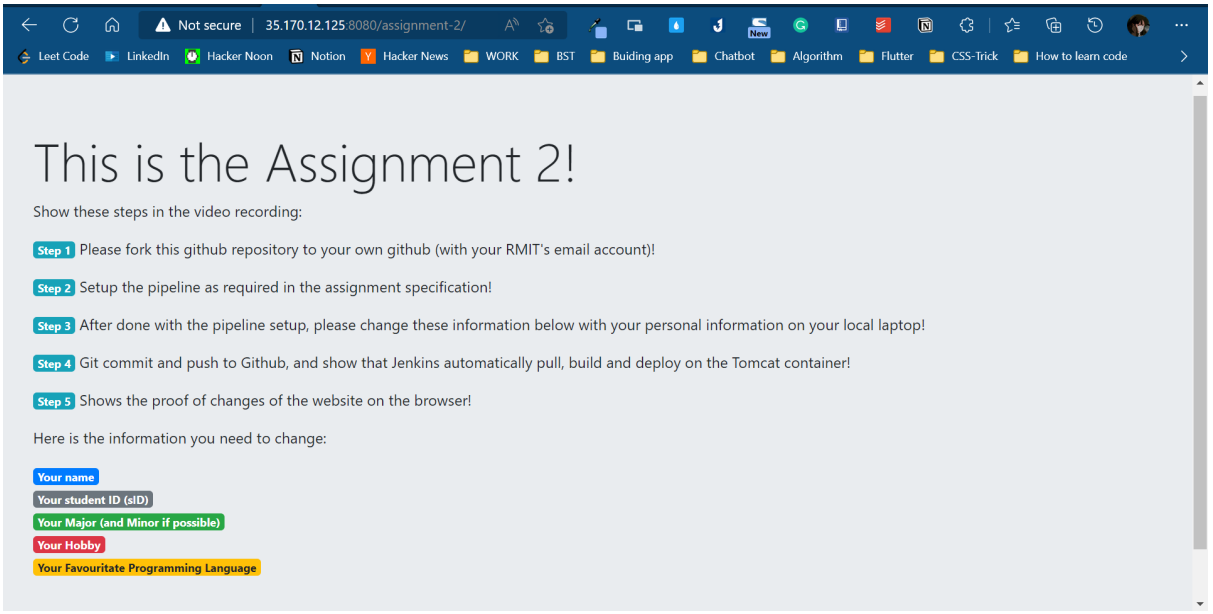


Figure 2.4: Web application on Tomcat server instance

- To check our trigger works correctly, let's change the source code by filling in some information on the website.

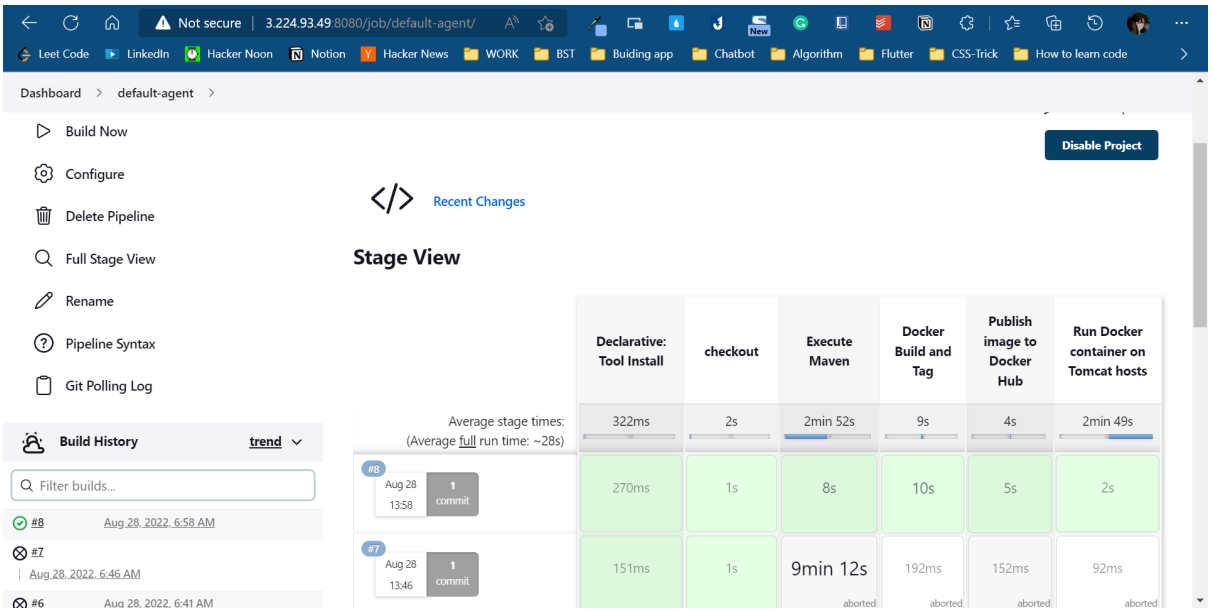


Figure 2.5: Pipeline process with the latest version

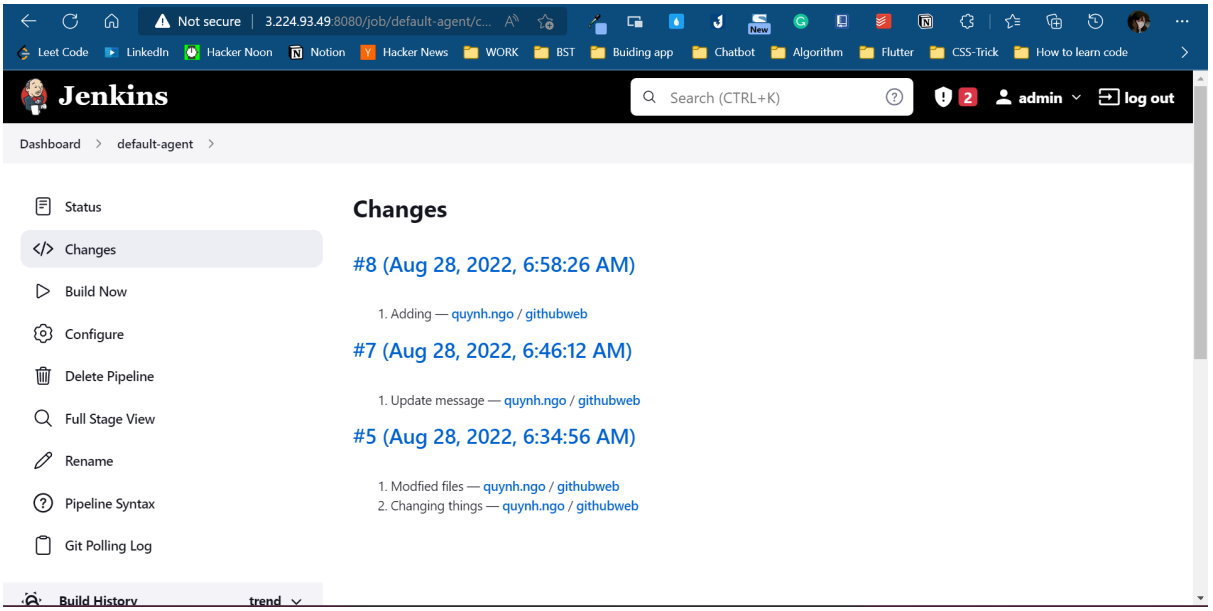


Figure 2.5: Changes dashboard

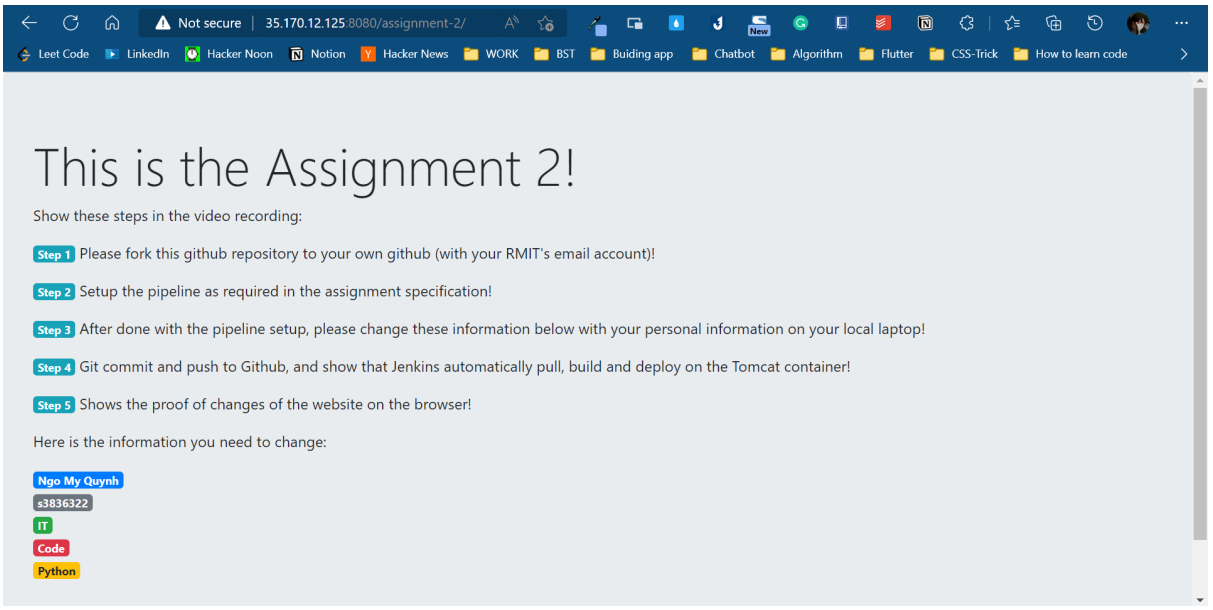


Figure 2.6: Application updating on Tomcat server