DevOps Toolchain Implementation

**1. Introduction**

In this project, we have tried to simulate a CI/CD pipeline for a node.js application using the Jenkins integration tool. For this we have used different applications/software’s and set of tools which are listed and discussed below:

* **AWS EC2 Cloud Instance** We have used AWS EC2 Cloud instance to deploy our node.js source code. You can get further details about this from aws.amazon.com.

The same instance is used to install the Jenkins Integration tool, Docker Engine, Docker-Compose for running the CI/CD from Jenkins to deploy the node.js application over the same instance.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Instance Type** | **Public IP** | **Instance OS** | **Inbound Ports Allowed** | **Outbound Ports Allowed** |
| T3.Medium | 52.11.199.59 | Ubuntu 20.04 | 8080,8888,22,80,443 | All |

* **Ubuntu OS on EC2 Instance** The unix based Ubuntu OS is chosen to deploy the application and the tools to be used for the CI/CD. The AWS EC2 Cloud instance is loaded with Ubuntu image at the time of spinning the EC2 instance.
* **Jenkins Integration Tool** The Jenkins is most common widely used integration tool for deploying CI/CD pipelines for setting up DevOps lifecycle. We have installed this Jenkins on the Ubuntu 20.04 instance to setup Continuous Integration/Continuous Delivery pipelines for our node.js application automated deployment.

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Website** | **Version** | **Plugins Used** |
| Jenkins | <https://www.jenkins.io/> | 2.303.3 | Node.js, Github Publisher, Git |

* **Docker Engine** Docker is the most commonly used containerization tool these days. As you all know about containerization so we wont need to discus it here, you will find several introductory lectures about this over internet. In short containerization enables us to run our application in isolated environment and is very helpful in micro services architecture. Here we have installed the docker-engine on Ubuntu instance to deploy our application in a containerized environment.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Website** | **Version** | **DevOps Lifecycle/Pipeline** | **Stage** |
| Docker Engine | https://www.docker.com/ | 20.10.7 | Continuous Deployment | 2, 3 (Build, Push) |

* **Docker-Compose** Docker-compose is the tool to deploy multi container applications using the docker-compose yaml files. These files have yaml extensions and contains the JSON format for defining multi services which initiates the docker containers for those services. We have used this docker-compose file to spin up Database and Node.js App container to run our application in CD pipeline.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Website** | **Version** | **DevOps Lifecycle/Pipeline** | **Stage** |
| Docker-Compose | https://www.docker.com/ | 1.29.2 | Continuous Deployment | 4 (App Deployment) |

* **Github SCM Tool** Git SCM tool is used to host our Application repository locally which is responsible to trigger the pipelines upon push event to central repo. Git SCM is the local repo software which is used to being sync to Github web central repository. This tool is installed on our windows machine from where the updates are being pushed to central repo.
* **Github Web** Github web is the central online repo which is integrated with the jenkins pipeline to trigger the pipeline whenever there is a push (update to that repo). There we have already created a repository named vhesara/loginapp.

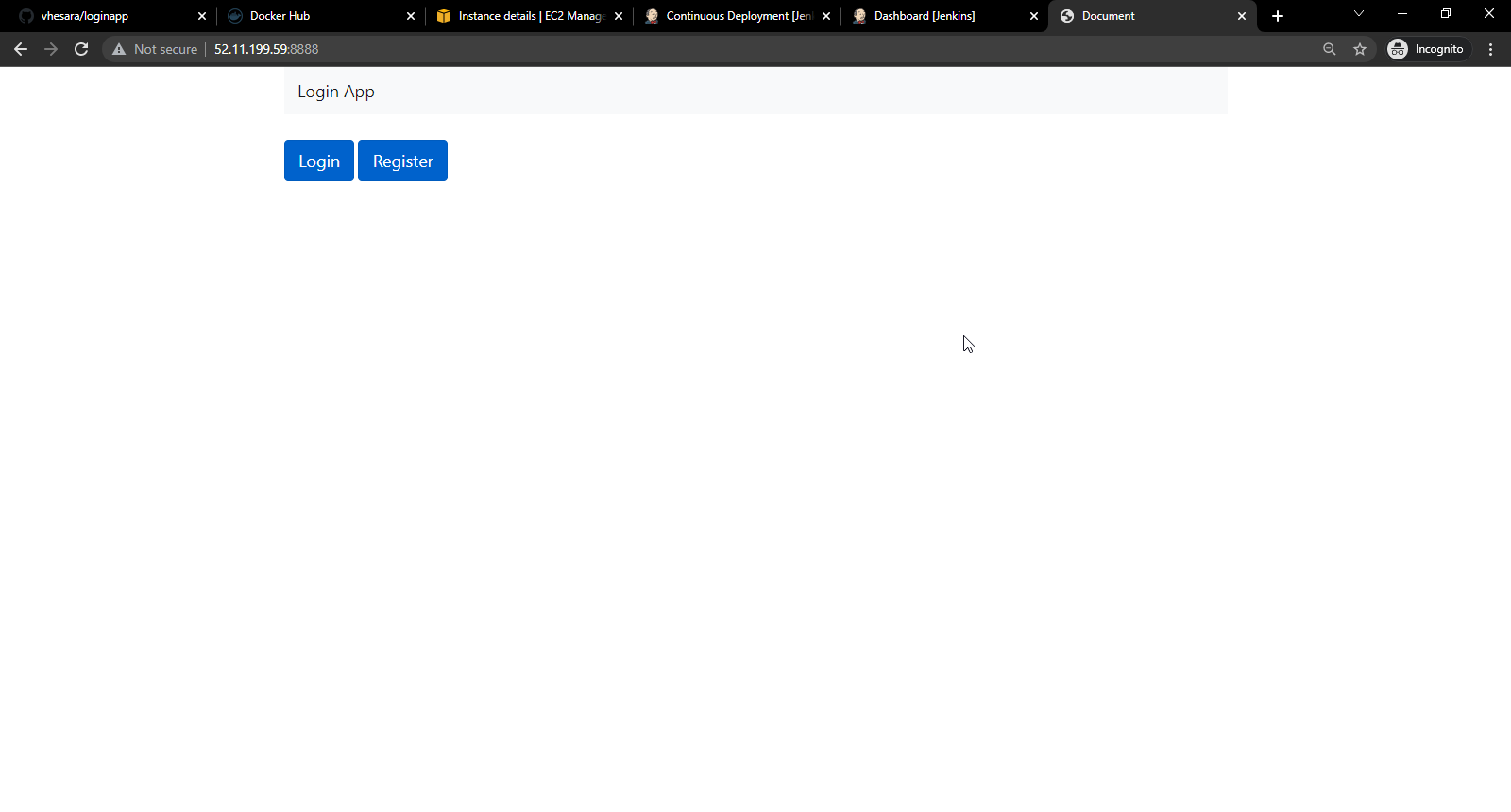
|  |  |  |
| --- | --- | --- |
| **Name** | **Repo Name** | **Repository Link** |
| Github | Loginapp | https://github.com/vhesara/loginapp/tree/development |

**2. Terminology**

This section is about to discuss the terminology used for this project, what components have been used and what the procedure is adopted to setup the CI/CD pipeline.

**2.1 Sample Application**

The sample application used in this example is node.js application which is simple login app with mysql database. The sample node.js applictaion is written in package.json file and is able to perform two function, one is login and another is registering the user. The application deployed image is shown below.



We have deployed this application using the Docker containers, there are two services deployed in separate containers for this node.js application, one is MySQL database to register the users and authenticate the user login and the other is the node.js application itself which is using the node package manager running.

**2.2 Continuous Integration**

As we all know that Continuous Integration is the development practice where developers integrate code into a shared repository and then each integration can then be verified by using automated build and test tools.

We have used the same approach to test the build whenever there is update in the central repository development branch and if the build goes to be successful then merging the development branch to the main branch which in result causes triggering the continuous deployment pipeline.

To implement this we have used the Jenkin integration tool where we have create a freestyle project as CI pipeline. This freestyle project pipeline has the following characteristics:

* This pipeline is integrated with github central repo (development branch) using the web hook.
* Whenever there is update on central repo development branch this pipeline is executed.
* Upon every execution of this pipeline, the following steps would be performed:
  + **Fetch the latest code from Github** Repo (development branch)- Source Code Management stage.
  + **Test the build** using the build environment of node.js plugin and executing commands in “Execute Shell” in Build stage.
  + If the **build is successful**, the development **branch is merged to the main repo**, which caused the Continuous Deployment pipeline to trigger.

**2.3 Continuous Deployment**

The continuous deployment is the pipeline setup in Jenkins where we are going to deploy our application in staging or production environment once the build is verified to be successful by CI pipeline.

This CD pipeline is of type Jenkins Pipeline, where we have copied the pipeline as a code/script. This pipeline is also integrated with central github repository main branch. As soon as the new build is tested and verified by the CI pipeline, the code is merged from development branch to main branch and in a result that update in main branch also trigger this CD pipeline which deploys the application in docker containers on the same jenkins instance by using the ssh commands. The docker and docker-compose have already been installed on the ubuntu instance. This pipeline has the following stages:

* **Git Clone:** This stage is to clone the updated repository from github main branch to local Jenkins instance.
* **Docker Build Image:** Once the updates files are there in local directory, a new docker image is built by running the below command over ssh.

*#docker build -t vahid-loginapp .*

This image would be build by below dockerfile contents.

*#base image*

*FROM node:alpine*

*#install dependancies*

*WORKDIR /usr/loginapp*

*COPY ./package.json ./*

*RUN npm install*

*COPY ./ ./*

*## Add the wait script to the image*

*ADD* [*https://github.com/ufoscout/docker-compose-wait/releases/download/2.7.3/wait*](https://github.com/ufoscout/docker-compose-wait/releases/download/2.7.3/wait) */wait*

*RUN chmod +x /wait*

*#start-up command*

*CMD /wait && npm start*

* **Push Image to Docker Hub:** This stage is used to push the image from local repo to docker-hub repo (owned by Vahid) by using the below command, so that it could be accessible globally.

*#docker push vahidhesara/loginapp:latest*

* **App Deploy using Docker-Compose**: This stage is used to run the application by deploying the mysql databas and node.js service by running the docker-compose file.

#*docker-compose up –d*

The docker-compose.yaml file contains the following content.

*version: '3'*

*services:*

*mysql-server:*

*image: mysql:5.7*

*environment:*

*MYSQL\_ROOT\_PASSWORD: password*

*MYSQL\_DATABASE: logindb*

*loginapp:*

*build:*

*dockerfile: Dockerfile*

*context: ./*

*environment:*

*MSHOST: mysql-server*

*MSUSER: root*

*MSPASSWORD: password*

*MSDATABASE: logindb*

*SERVERPORT: 8888*

*WAIT\_HOSTS: mysql-server:3306*

*WAIT\_HOSTS\_TIMEOUT: 90*

*ports:*

*- "8888:8888"*

*depends\_on:*

*- mysql-server*

**3. Step by Step Implementation**

To implement this project, we should have the Ubuntu instance ready with public IP(required by github webhook). Once the Ubuntu Instance ready and public IP is attached, you can go ahead with following steps:

* 1. **Setup & Installation:**

**3.1.1 Provision EC2 Instance**

**Step#1** Go to EC2 Service Console from AWS Management Console.

**Step#2** On EC2 console, click on Launch Instance

**Step#3** Select the AMI as Ubuntu 20.04 64 bit

**Step#4** Choose Instance Type as t2.micro, click on Next: Configure Instance Details

**Step#5** Select the parameters values as below:

* Number of instances: 1
* Network: Default VPC
* Auto-Assign Public IP: Yes

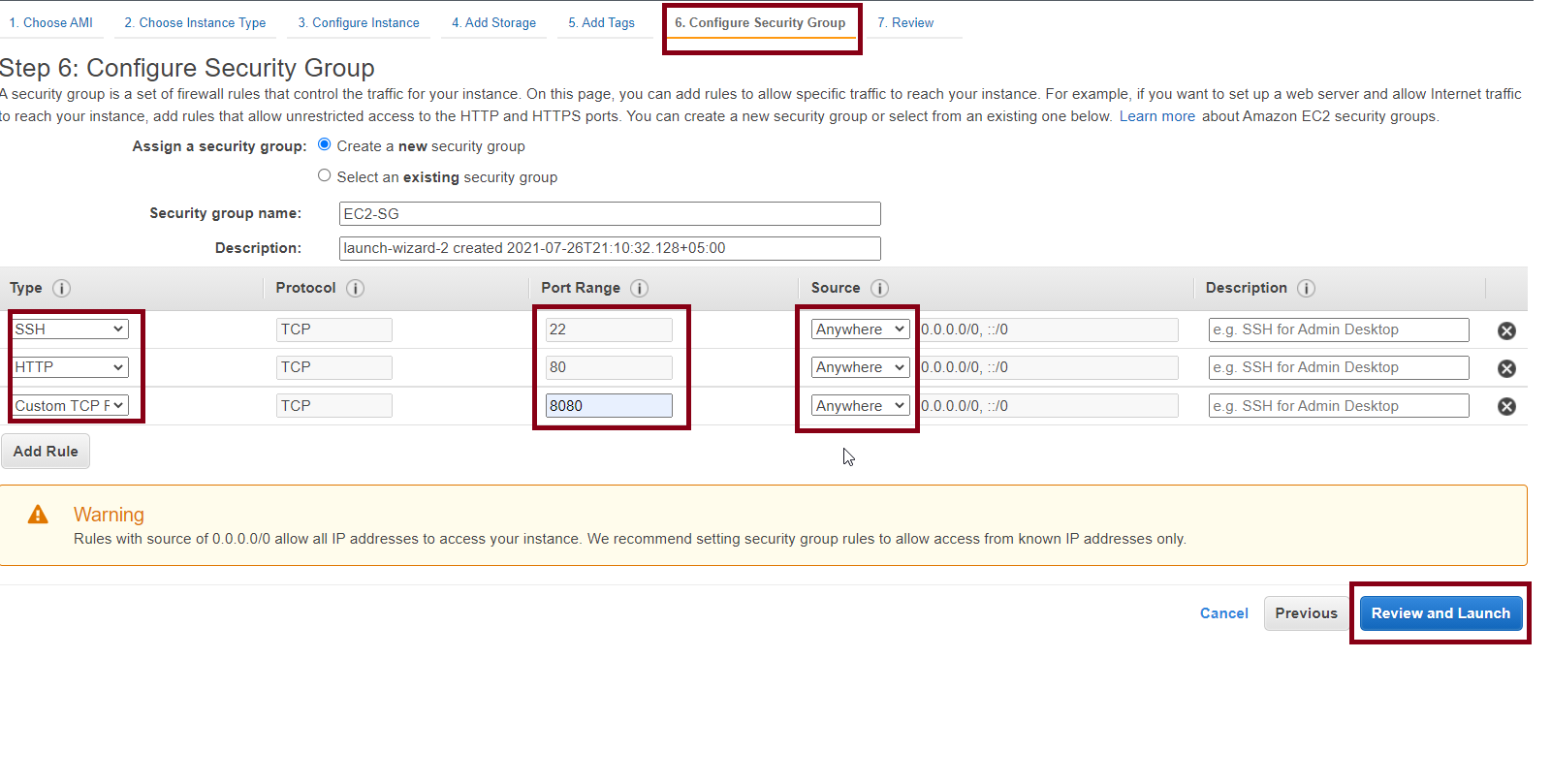
**Step#6** Leave other options as default and click on Next:Add Storage

**Step#7** Leave the storage size to default 8 GB, if your image is under this size, if your app is large then change it to desired. For our example app, 8 GB is enough.

**Step#8** Click on Next:Add tags.

**Step#9** Add the tags as you want or leave them blank, and click on Next Configure Security Group.

**Step#10** On Security Group, you can use either existing SG or create new one if you do not have it.Click on create new and make sure to add the port 22 and port 80 and port 8080 allow as inbound from source anywhere.



**Step#11** Once you added the SG, click on Review and Launch.

**Step#12** Review the settings and click on Launch, it will ask you to provide the Keypair if you already have one or create the new keypair for this instance. Remember that you would need this keypair(private key which would be generated from this public key) to ssh to this instance. If you do not have the keypair then create a new one, and public key would auto be downloaded to your system, you can use this public key to generate the private key using the Putty KeyGen later. And that private key would be used to ssh this ec2 instance in later section.

* + 1. **Connecting to your Ubuntu EC2 Instance**

**Step#1** Launch Putty Gen

**Step#2** Load the public key that have been downloaded to your system when creating the instance.

**Step#3** Once loaded the public key successfully, click on save private key.

**Step#4** Name your key same as public key and save it. (type should be ppk) and close Putty Gen.

**Step#5** Note the public ip of Instance from EC2 console.

**Step#6** Open putty software, and paste public IP in Hostname/IP Address and on left section; click on Connection🡪SSH🡪Auth🡪 Browse and provide that private key that you have generated above.

**Step#7** Once copied public IP into hostname textbox with port 22, and uploaded the key, click on open.

**Step#8** New terminal windows would be opened there, asking for the username, provide username as “ubuntu” and click enter, and you would have the ssh terminal access of your Ubuntu EC2 instance.

* + 1. **Jenkins Setup**

In this tutorial, we have used Jenkins as the automation server on which we setup the CI/CD pipeline, to setup the Jenkins Server we follow the below steps.

**Step#1** SSH to your Ubuntu access if you have it on cloud, or open terminal if you have physical access.

**Step#2** Once on terminal, run the below commands to setup the Jenkins.

Note: #if you get error on any below command, try typing by your own hand rather than pasting it. Or copy the commands from below link : https://www.jenkins.io/doc/book/installing/linux/#debianubuntu

sudo apt-get update

sudo apt install openjdk-8-jdk

wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -

sudo sh -c 'echo deb https://pkg.jenkins.io/debian-stable binary/ >

/etc/apt/sources.list.d/jenkins.list'

sudo apt-get update

sudo apt-get install Jenkins

sudo systemctl enable jenkins

sudo systemctl start jenkins

sudo systemctl status jenkins

sudo ufw allow 8080

sudo ufw status

sudo ufw enable

**Step#3** Now open the below link on your client machine

http://ip\_address\_or\_domain:8080

**Step#4** The above link will prompt you with Administrator password, you can know the administrator password by running below command in Jenkins instance, and input that into console and later change the password.

sudo cat /var/lib/jenkins/secrets/initialAdminPassword

5d35dab7e0984406ad5660f20b333a6f

**Step#5** Click on “Install Suggested Plugins”, this will install the basic required plugins.

**Step#6** Once the plugins have been installed, you can create the first user and set the password and other details on next screen.

**Step#7** After that you can configure the Jenkins URL, leave it to default if you do not want any changes.

* + 1. **Installing the Docker & Docker-Compose**

To run/deploy the node.js app in Docker container using CI/CD pipeline on Ubuntu EC2 instance, you would need to build the image from the dockerfile and then pushing your image to dockerhub and running it on Ubuntu instance in doc. But for this we need to setup/install the docker and docker-compose software on them, to achieve this follow below steps.

**Step#1** To install Docker Engine, rune below commands.

sudo yum update –y

sudo amazon-linux-extras install docker

sudo yum install docker

sudo systemctl enable docker

sudo systemctl start docker

sudo usermod -a -G docker ec2-user

sudo chmod 666 /var/run/docker.sock

docker -version

**Step#2** To install Docker-Compose, run below commands.

sudo curl -L https://github.com/docker/compose/releases/download/1.22.0/docker-compose-$(uname -s)-$(uname -m) -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose

docker-compose version

* + 1. **Git SCM Installation**

To setup the CI/CD pipeline, we must have to push the code to central repo(github, gitlab), and for this to achieve we have to first setup the local repository(distributed repo) and to connect that to central repo, so every time when there is update in change, the user must commit the change and push the changes to Github.

**Step#1** Download the Git for windows. <https://git-scm.com/download>

**Step#2** Run the executable

**Step#3** At the time of installation, choose default options for all except one to run git using both cmd and gitbash.

**Step#4** Once installed successfully, you can run it by opening Gitbash, or cmd or by going to that folder which you want to initialize as local repository (Version Control System) and right click that folder to go to gitbash from there.

**Step#5** To initialize your code repository, go to that folder on explorer and right click there and click on GitBash here.

**Step#6** On gitbash console, type the following command to make your code folder as local git repo and push that into central repository.

$ *git init*

$ *git status*  # Will give u status of edited files not commit yet,but 1st have 2 add d files 4 d commit,after d files committed nothing shown

$ *git commit -m "first commit"* .

$ *git config --global user.email* [*yourGitHub@email.com*](mailto:yourGitHub@email.com)#to add your credentials for pushing without prompt for password.

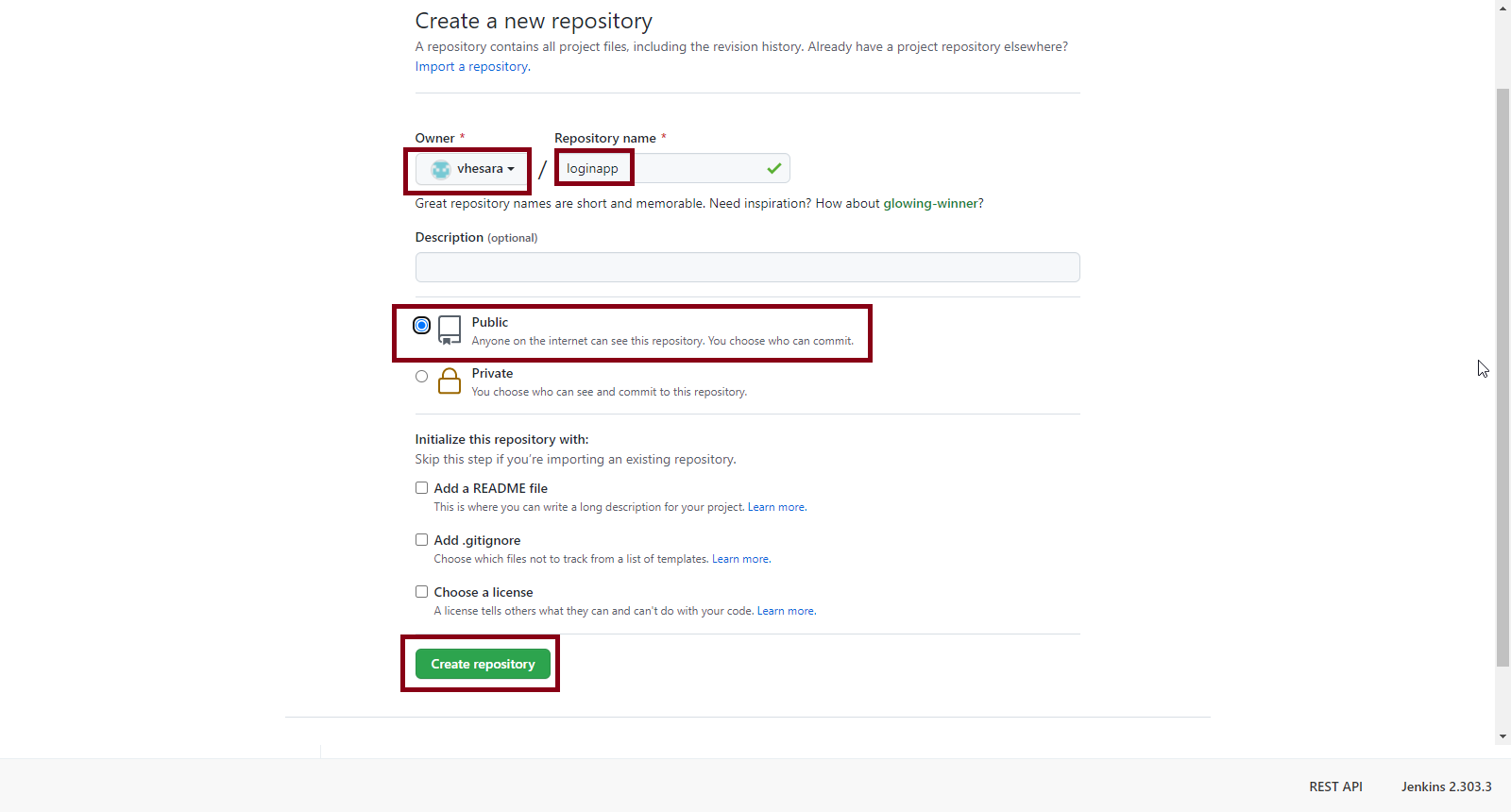
$ *git remote add origin* [*https://github.com/username/repositoryt.git*](https://github.com/username/repositoryt.git)

$ *git push -u origin master*

**Step#7** Once the changes have been pushed to central repo, you could see all of your code files and folders there.

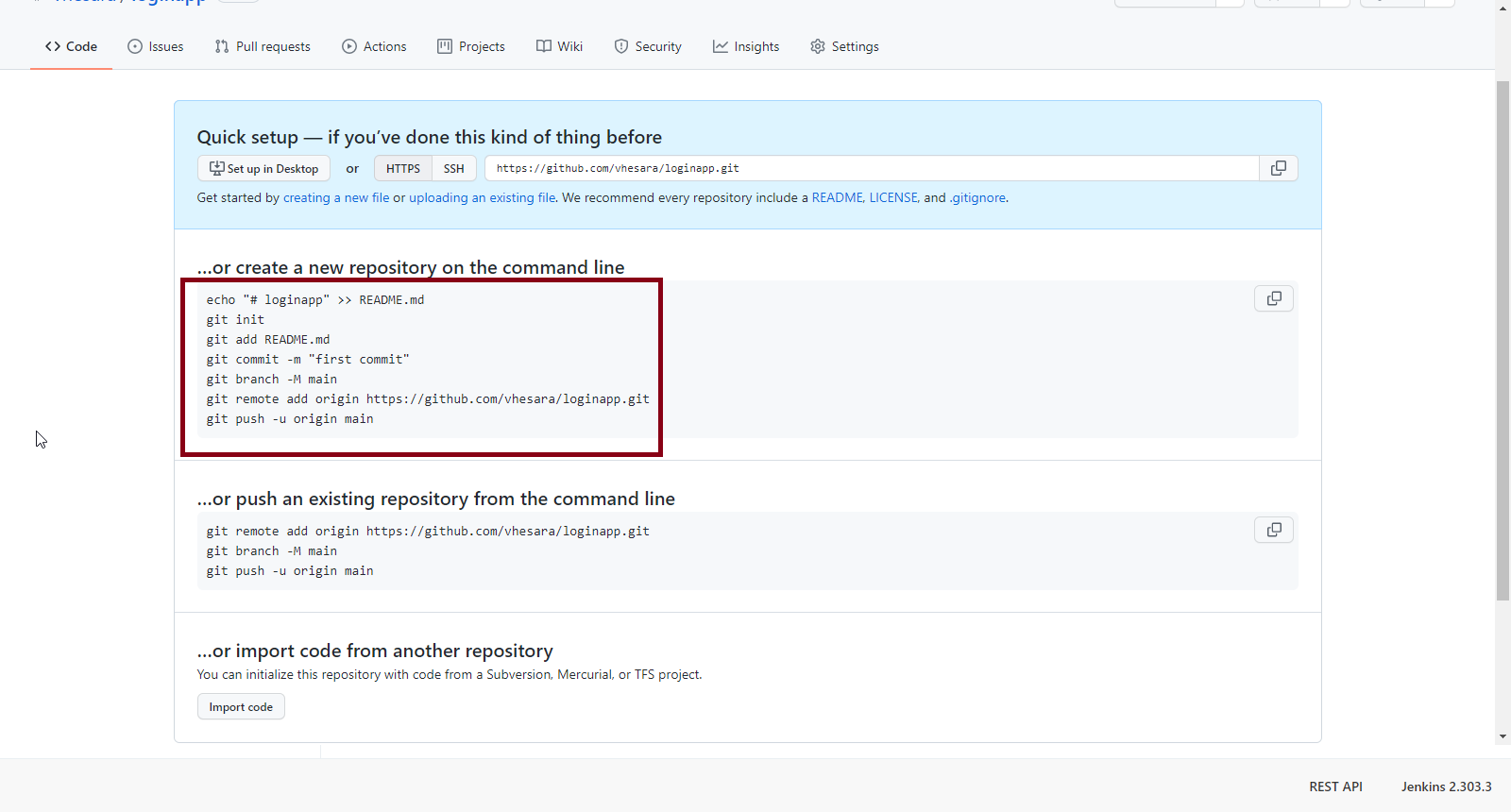
* 1. **Software Configuration**
     1. **Setting up Central Github Repo & Creating Access Token**

To implement this project, we would have to create a central repo on github, in our example we have already created one with name username/repository. After that we should have access token in hand as it would be needed in CI pipeline.



* + 1. **Initializing Local Repo with Github**

Once the central github repo has been created, we can initialize our project directory on local machine to sync that directory to the central repository. For this Git SCM tool is to be used which is required to be setup on the machine where the app code is resided. We have already done this in above step#6 of section 4.1.5.



**3.2.3 Creating the Development Branch and Push that to Central Repo**

Once you have successfully initialize the local app directory with github and sync that, you will next create a development branch using the GIT Cli tool and push that changes to github central repo. In final we would have two branches on Central Repo: Main & Development.

Follow below steps to create the development branch on github repo.

**Step#1** On gitbash (command line terminal), go to your local repo directory with below command and create and push the branch to github repo.

$cd nodeapp\_repository

$git checkout -b development

$git push origin development

* + 1. **Creating Repo on Dockerhub**

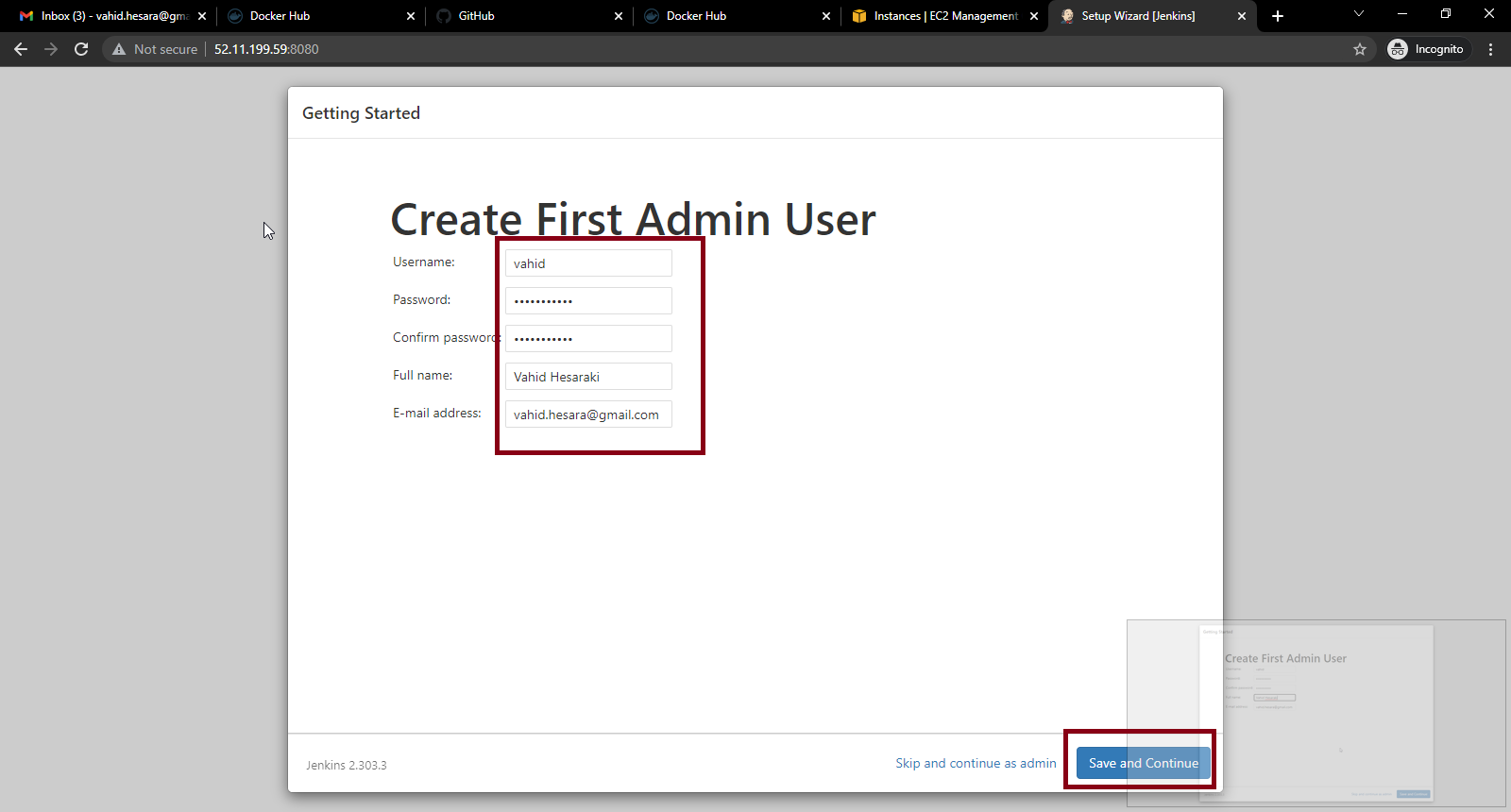
To make our application docker image available globally, we have to push the image on docker hub repository, for this we have created the repository on dockerhub also.

To create the repo on dockerhub, follow below steps:

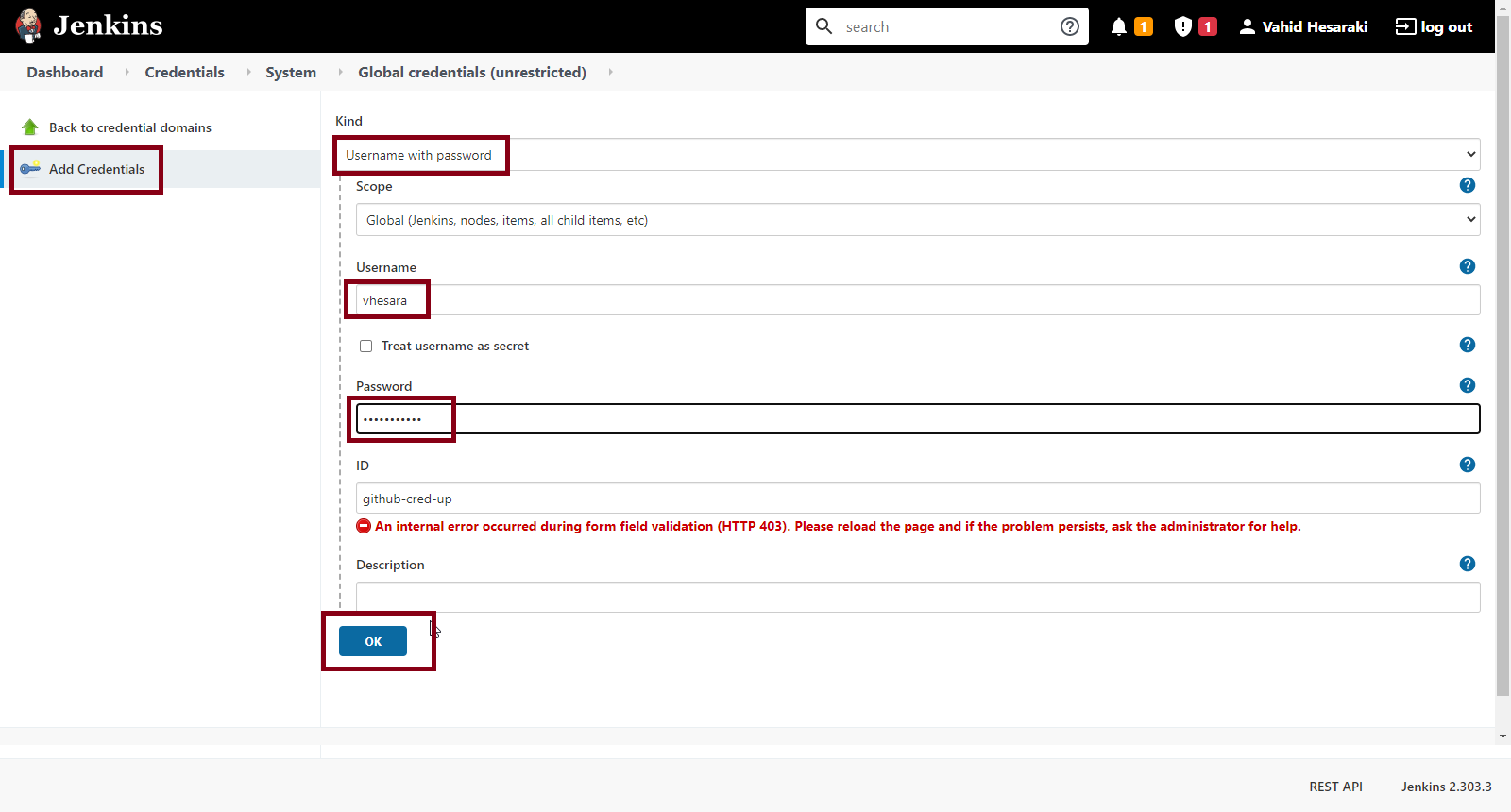
**Step#1** Sign-in/up to dockerhub account on web console. <https://hub.docker.com>

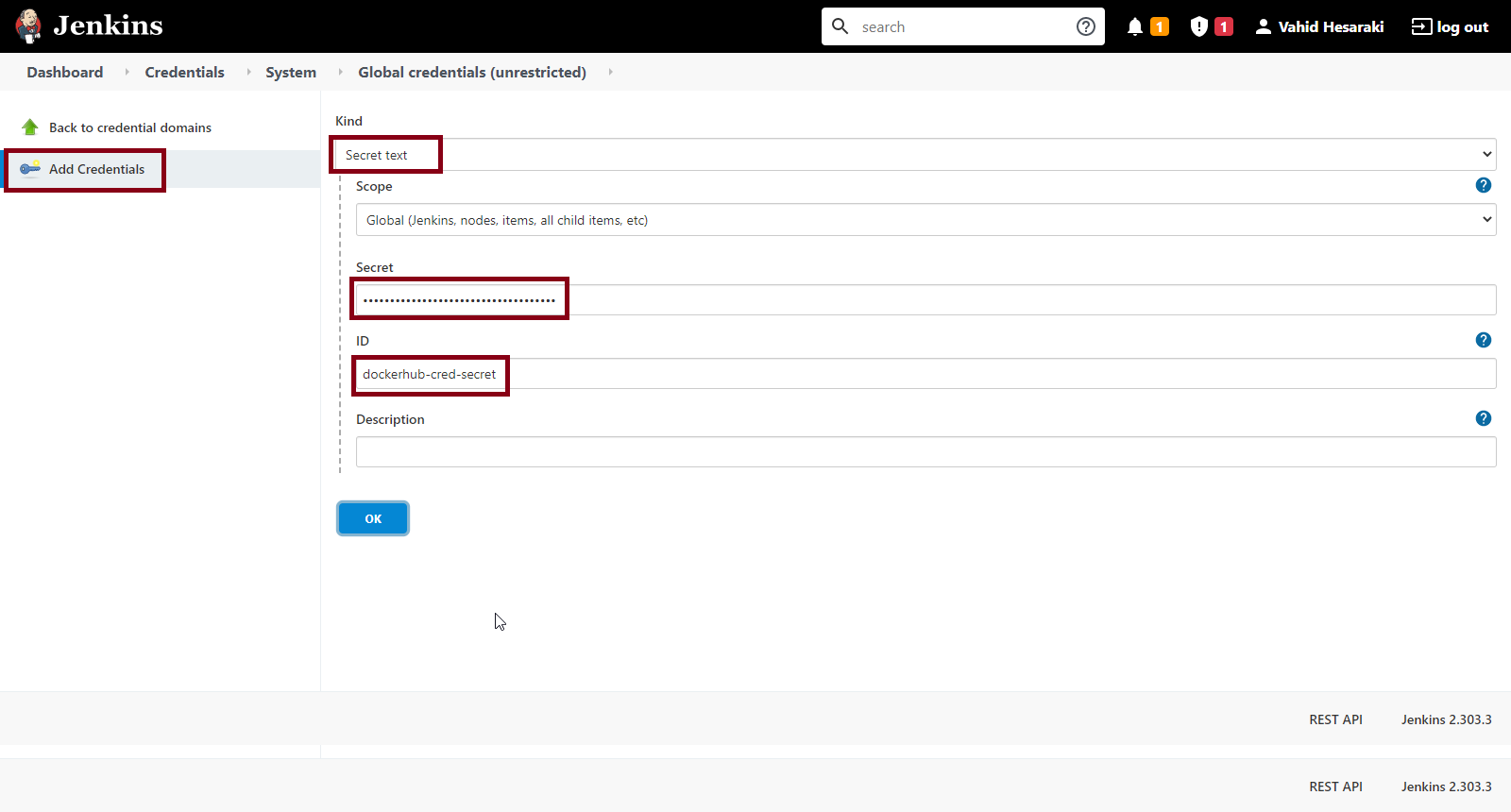
**Step#2** Create a public repo.

* + 1. **Jenkins Configuration**

**Step#1** Once the Jenkins has been installed, you can login to the jenkins on any web browser, you would be prompted to **setup the username and password** and **install the recommended plugins**.

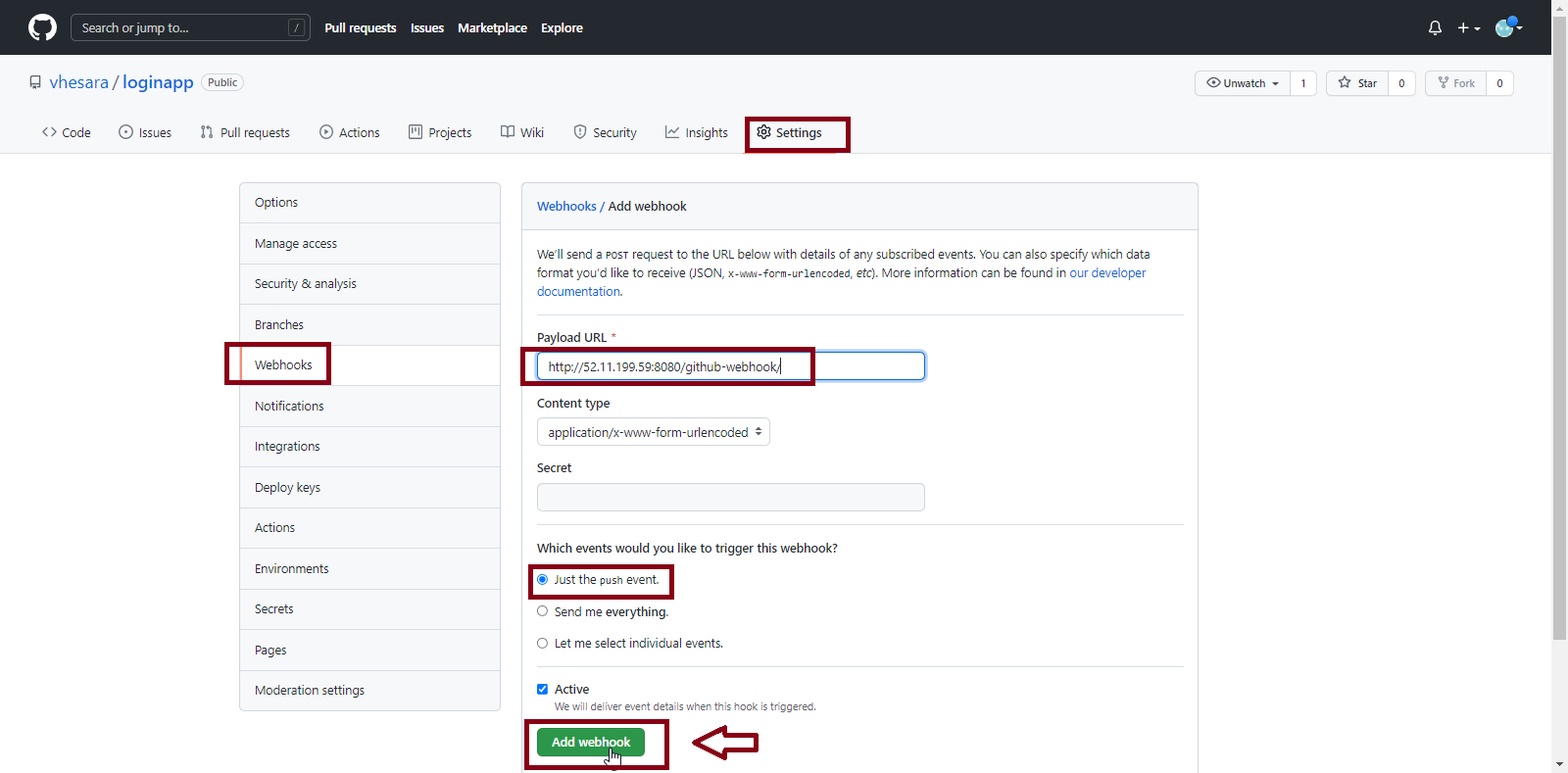
**Step#2** Next is to **setup the Github and docker-hub credentials**, for this go to Manage Jenkins and Manage Credentials, there you have to add the Github credentials with Username & Password type while the Docker-Hub credentials would be Secret type.





**Step#3** Once you have setup the github and dockerhub credential in jenkins, next is to **install the node.js plugin** required by pipeline, and **enabling** that from Global Tool Configuration in Manage Jenkins Section.

**Step#4** After the credentials have been added on Jenkins, next is to **enable the webhook in Jenkins System Configuration** and **add that webhook** address to the repository in Github repo settings.



**3.3 Creating the Continuous Integration Pipeline**

After installing all the required softwares and configuring them, we will setup our Continuous Integration pipeline on jenkins, to do this follow below steps:

**Step#1** Click Create Item on Jenkins Dashboad.

**Step#2** Name your project and select it as freestyle project.

**Step#3** Fill the valueas as below and save:

* 1. Discard Old Build -> Checked
  2. Days to keep build -> 5
  3. Max Number of builds ->5
  4. Github project -> <https://github.com/vhesara/loginapp/>
  5. Source Code Management -> Git
  6. Repo URL-> <https://access_token@github.com/vhesara/loginapp>
  7. Branches to Build -> \*/development
  8. Build Trigger -> GitHub hook trigger for GITScm polling (Checked)
  9. Build Environment -> Specify NodeJS Installation
  10. Build Section -> Execute Shell
  11. Execute Shell Command -> npm install

npm test

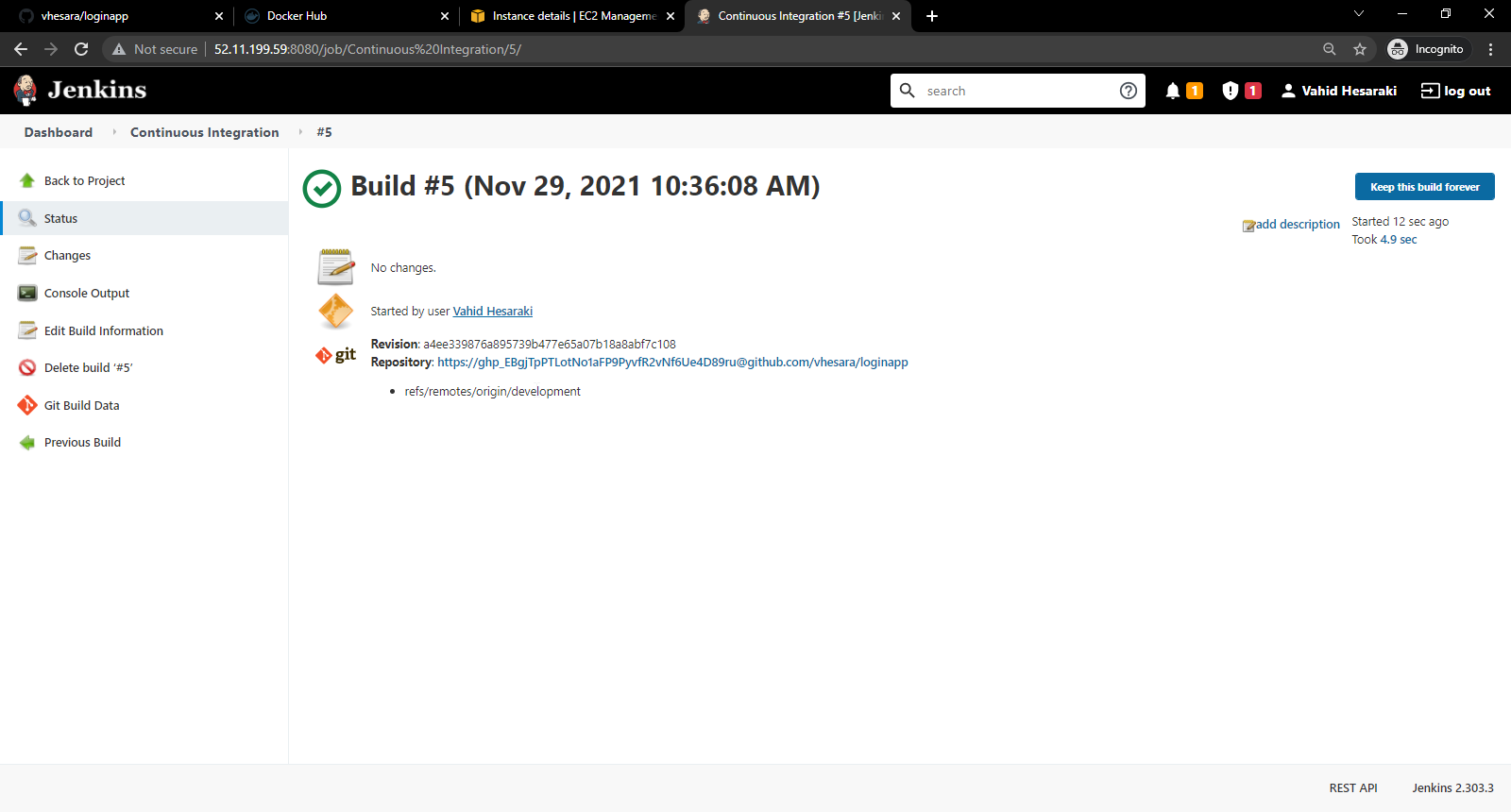
* 1. Post Build Section -> Git Publisher

Check on Push only if Build Succeed and Merge results.

Branches to Push: Main

Target Remote Name: Origin

Once you have saved the pipeline, you can build this pipeline and you would get the success tag with green status once the pipeline has been successfully build, if any error is encounter you can check to console output and identify the cause.



**3.4 Creating the CD Pipelines**

After CI pipeline has been successfully executed, we can continue with setting up the Continuous Deployment/Delivery pipeline, which will deploy the application using docker containers in staging environment. To setup the CD pipeline follow below steps:

**Step#1** Click **Create Item** on Jenkins Dashboad.

**Step#2** Name your project and select it as “**pipeline**” project.

**Step#3** In **Build Trigger**, put check on “GitHub hook trigger for GITScm polling”

**Step#4** In Pipeline, select pipeline script and paste the below code in that pipeline and save the pipeline.

node {

stage("Git Clone"){

git credentialsId: 'github-cred-up', url: 'https://github.com/vhesara/loginapp', branch: 'main'

}

stage("Docker build"){

sh 'docker version'

sh 'docker build -t vahid-loginapp .'

sh 'docker image list'

sh 'docker tag vahid-loginapp vahidhesara/loginapp'

}

withCredentials([string(credentialsId: 'dockerhub-cred-secret', variable: 'PASSWORD')]) {

sh 'docker login -u vahidhesara -p $PASSWORD'

}

stage("Push Image to Docker Hub"){

sh 'docker push vahidhesara/loginapp:latest'

}

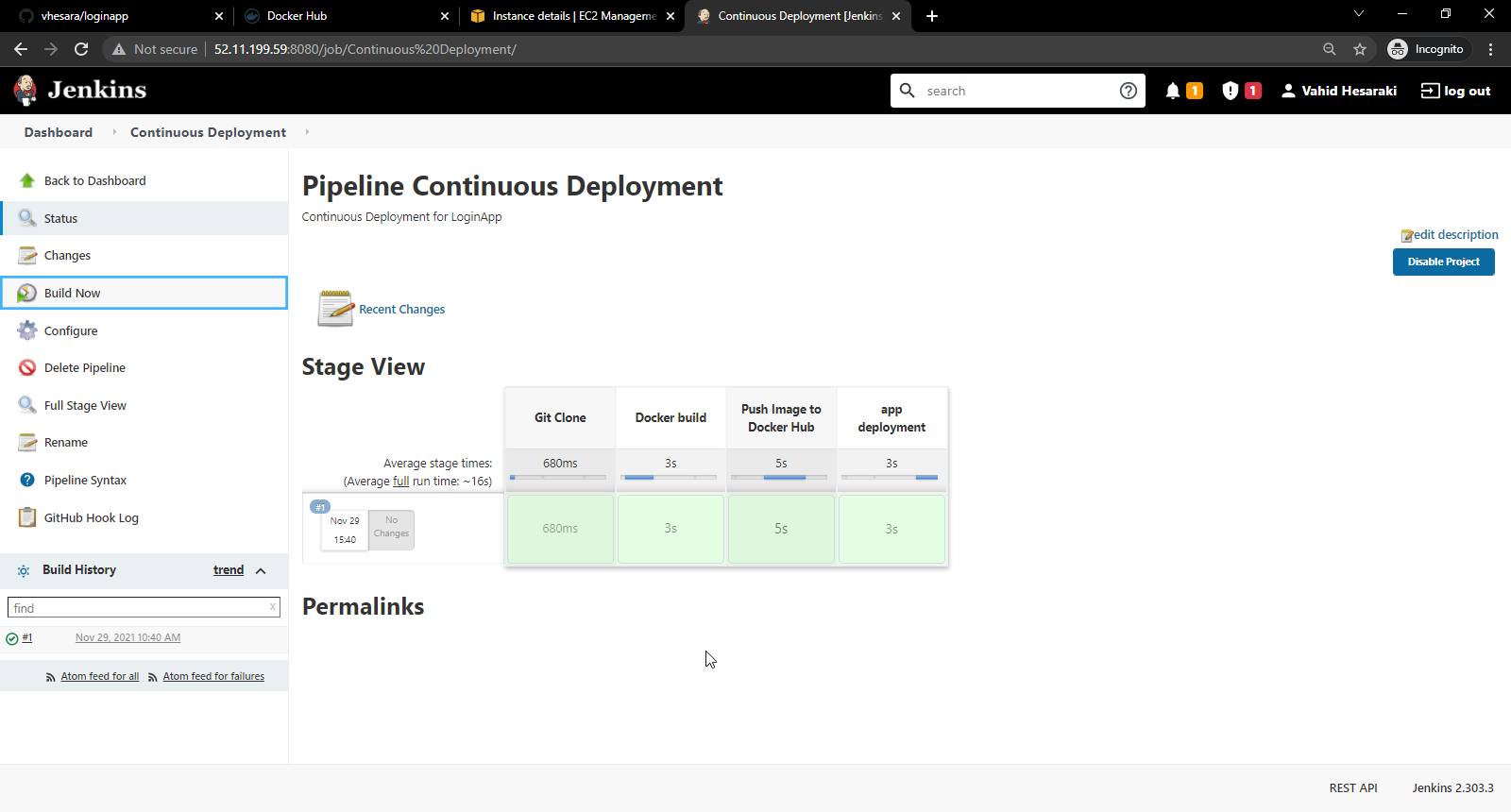
stage("app deployment"){

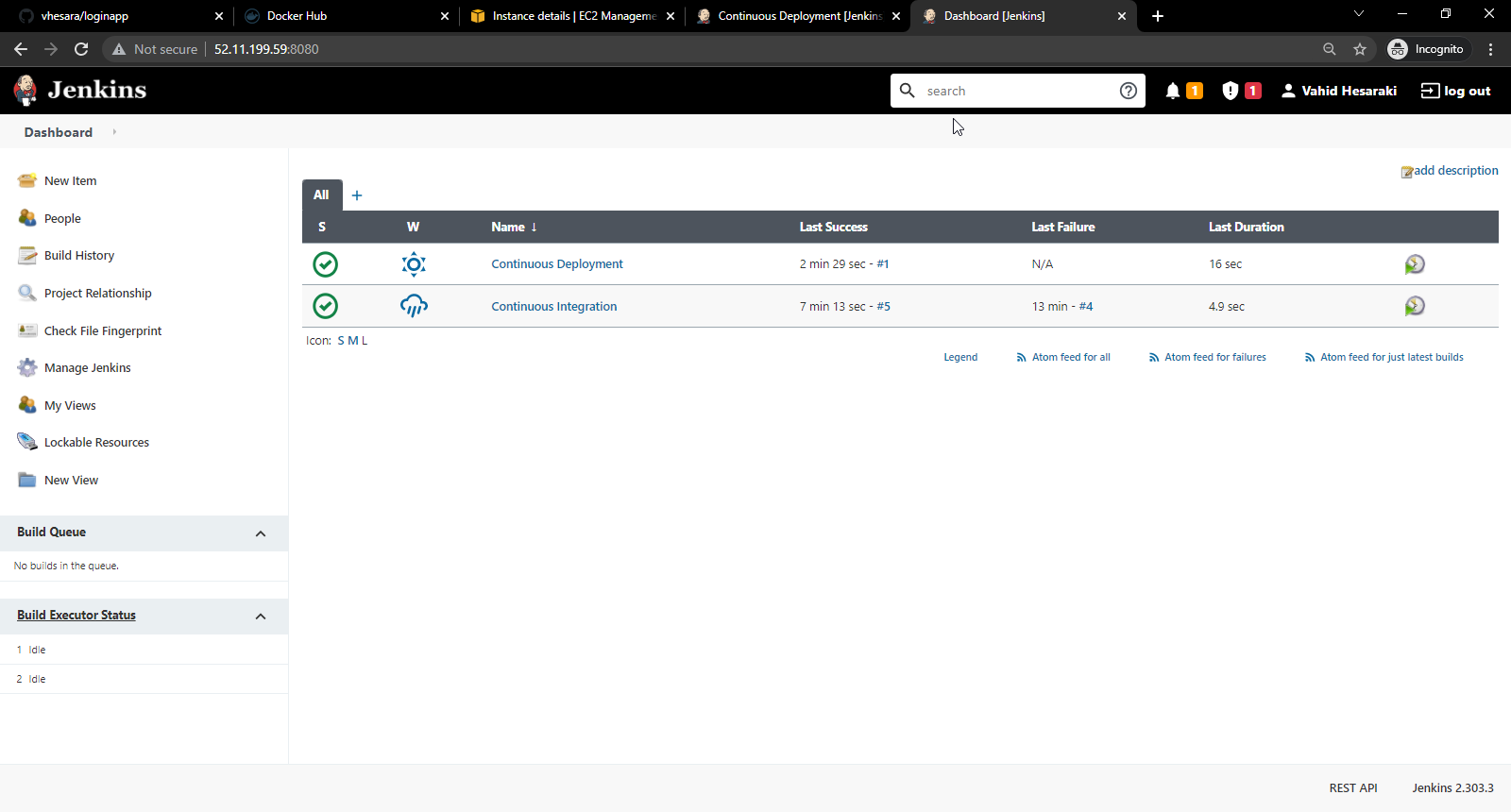
sh 'docker-compose up -d'

}

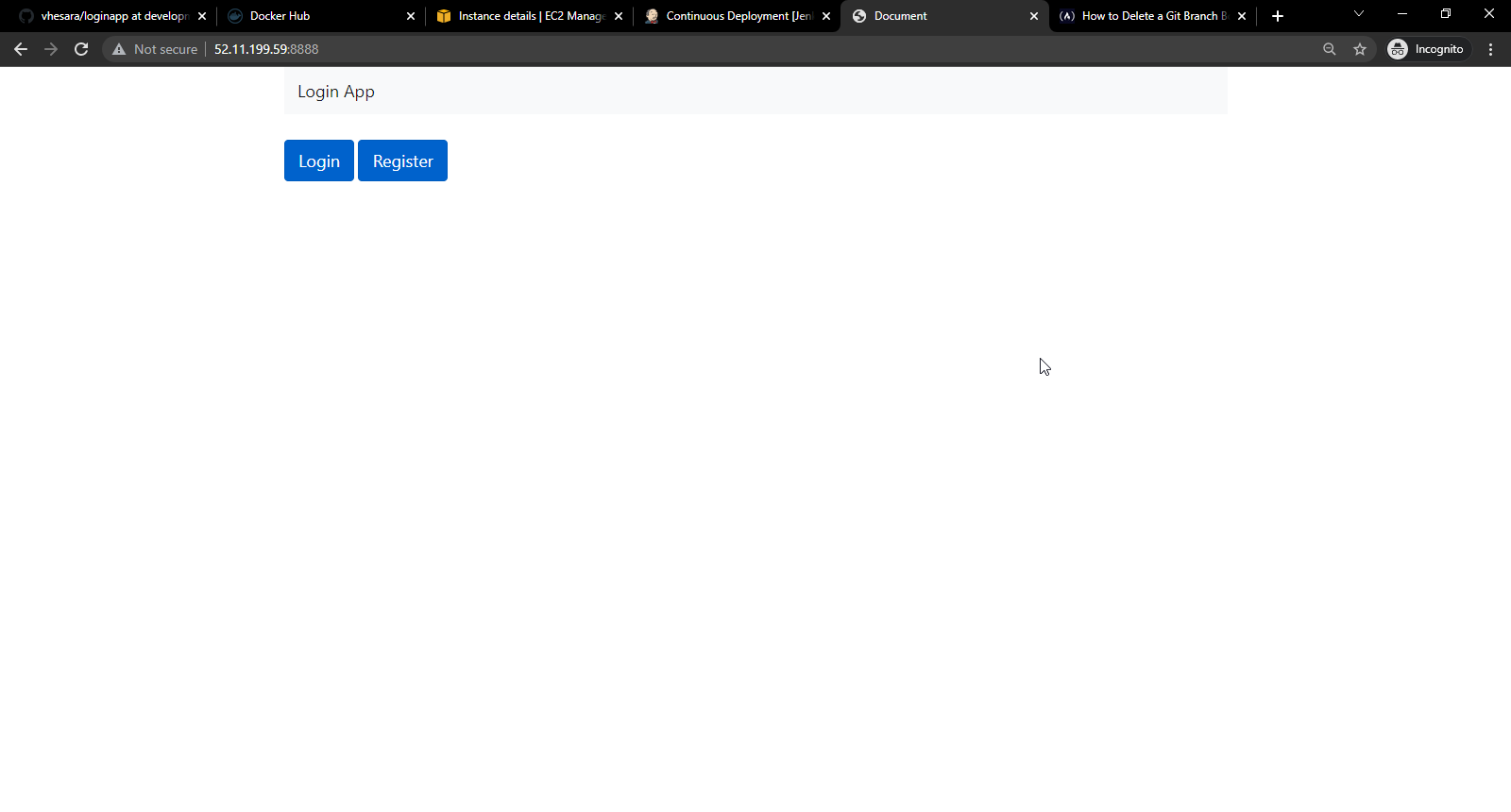
}

Once the pipeline is finished, you can execute this manually or by pushing any changes to development branch, which in turn merges the changes to main branch, and the webhook in that github repo would trigger the CD pipeline and deploys the node.js login application using docker containers.





Once the CD pipelines has been successfully executed, you could open your application with same public IP address followed by 8888 port in web browser.



1. **Summary / Conclusion**

In this project, we have simulated the implemention of CI/CD pipeline on jenkins for deploying our node.js application in docker container. We have demonstrated the following scripts in this solution.

* Package.json – This is the node.js application file.
* Dockerfile – To containerize the node.js application into image and running the container from that image.
* Docker-compose File – As our application requires the node.js tool and the mysql database services, this docker-compose file has been used to run MySQL DB Container and Node.JS app container.
* Jenkins CD Pipeline Script – This script is not included in directory but directly written in CD pipeline script section. This script has many stages of git clone, build the docker image(containerize the node.js app), pushing the image to dockerhub, and running the app using docker-compose file.

In this solution, we have demonstrated how the push event in central repo trigger the test build and upon successful test results, the build is deployed in staging environment from where the build could finally be deployed in production environment.

**The End.**