**Hand-on Lab Continuous deploy using Jenkins Pipeline**

Contents

[Logon to Jenkins 2](#_Toc22130959)

[Create a new pipeline job 2](#_Toc22130960)

[Adding ad-hoc parameters to pipeline 3](#_Toc22130961)

[Adding username as parameter 3](#_Toc22130962)

[Generating sonarqube for build token 4](#_Toc22130963)

[Adding Sonarqube CLI as parameter in pipeline 6](#_Toc22130964)

[Configuring Pipeline 7](#_Toc22130965)

[Jenkinsfile - Explained 8](#_Toc22130966)

[Triggering the Jenkins pipeline 9](#_Toc22130967)

[Verify the Docker image in registry 12](#_Toc22130968)

[Pipeline stages - Explained 13](#_Toc22130969)

[Declarative Checkout SCM and Preparation 13](#_Toc22130970)

[Update user references 14](#_Toc22130971)

[Build 14](#_Toc22130972)

[SonarQube 15](#_Toc22130973)

[Build Image 15](#_Toc22130974)

[Push Image to repo 16](#_Toc22130975)

[Deploy the application 16](#_Toc22130976)

[Verifying the Deployed application 17](#_Toc22130977)

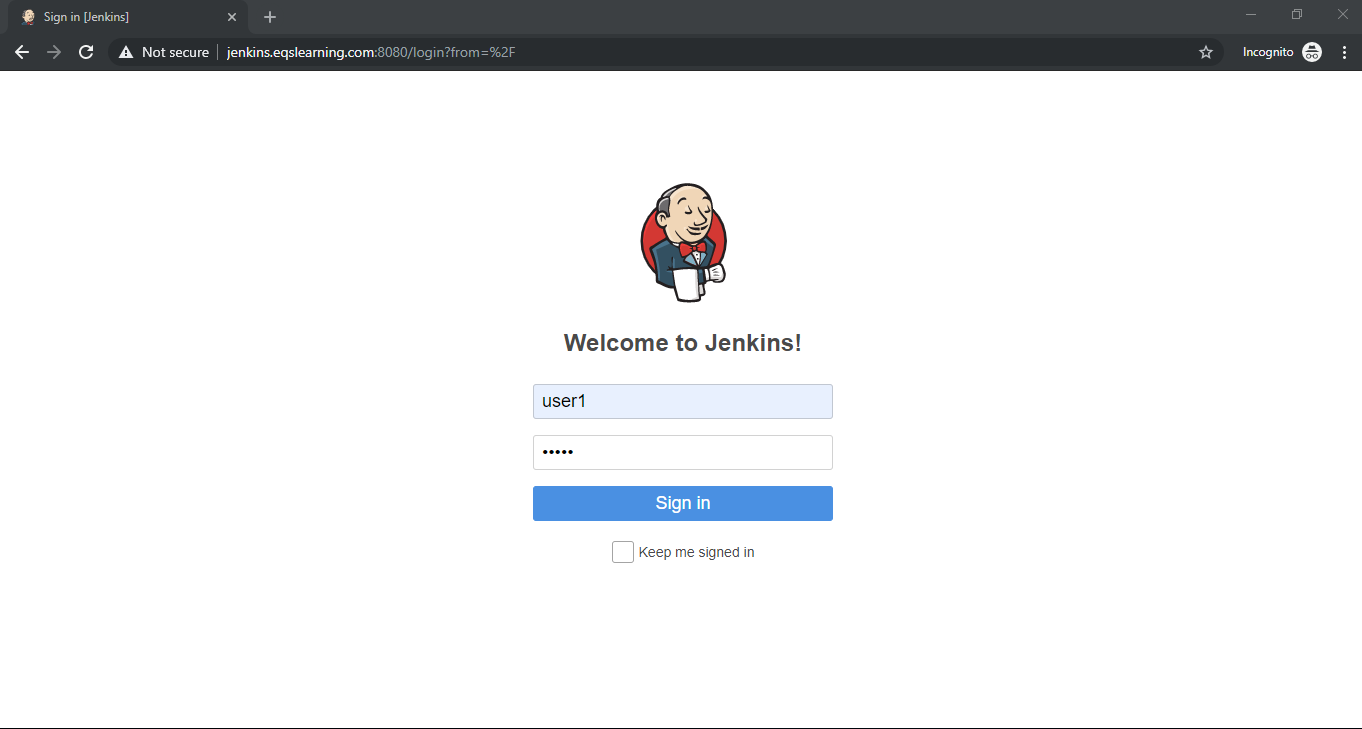
[Accessing Traefik dashboard 17](#_Toc22130978)

[Accessing the external Domain address 18](#_Toc22130979)

# Logon to Jenkins

Logon to Jenkins with below URL address with user id shared to you. E.g : user1. Password would same as username.

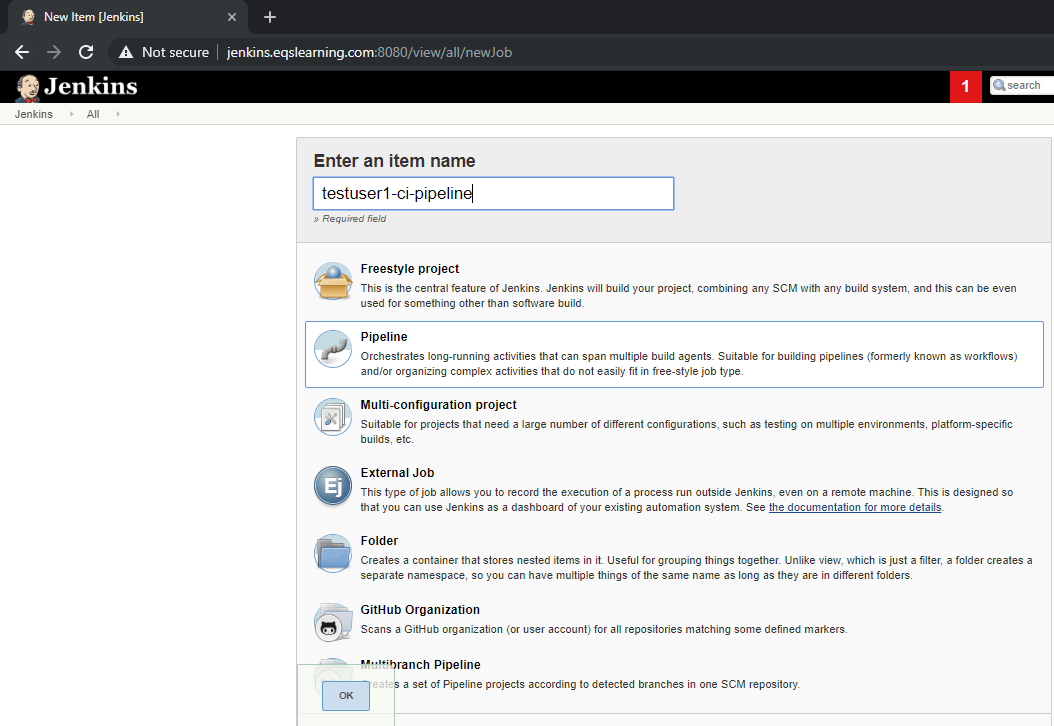
[http://jenkins.eqslearning.com:8080](http://jenkins.eqslearning.com:8080/)



# Create a new pipeline job

Click on **New Item** option on the left menu bar to create new Jenkins Job. Select job type as **Pipeline** andEnter the job name and click **OK** button on the bottom of the page.

Note: For convenience of others, kindly use the naming convention as **<username>-ci-handson**



# Adding ad-hoc parameters to pipeline

Since we are using common git repo to create multiple pipeline, we need to separate the sonarqube setup for each user. Hence we are passing the username and sonarqube cli as parameters.

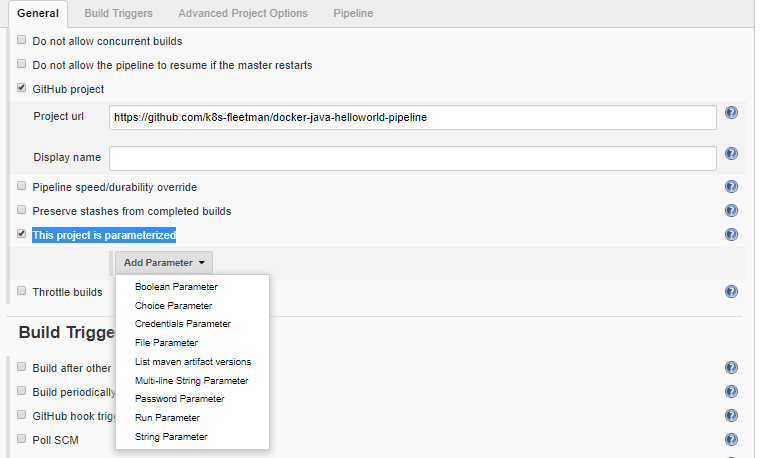
On newly created pipeline, update the below GitHub project URL.

<https://github.com/k8s-fleetman/docker-java-helloworld-cd-pipeline>

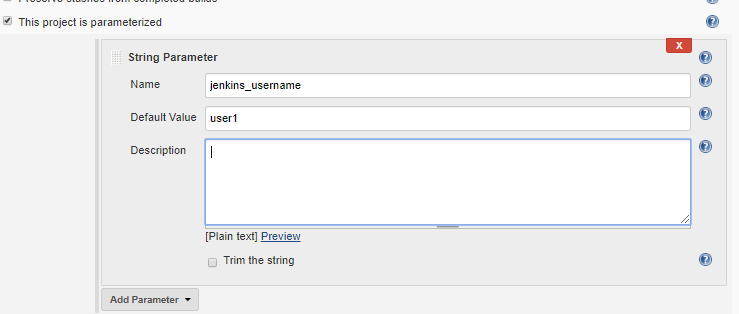
## Adding username as parameter

To build separate Docker image for each user, we will be creating a parameter which will be used in pipeline for building the Docker container.

On General section of the newly created pipeline, check the option “**This project is parameterized**” and click on option **Add parameter** and select **String parameter**.

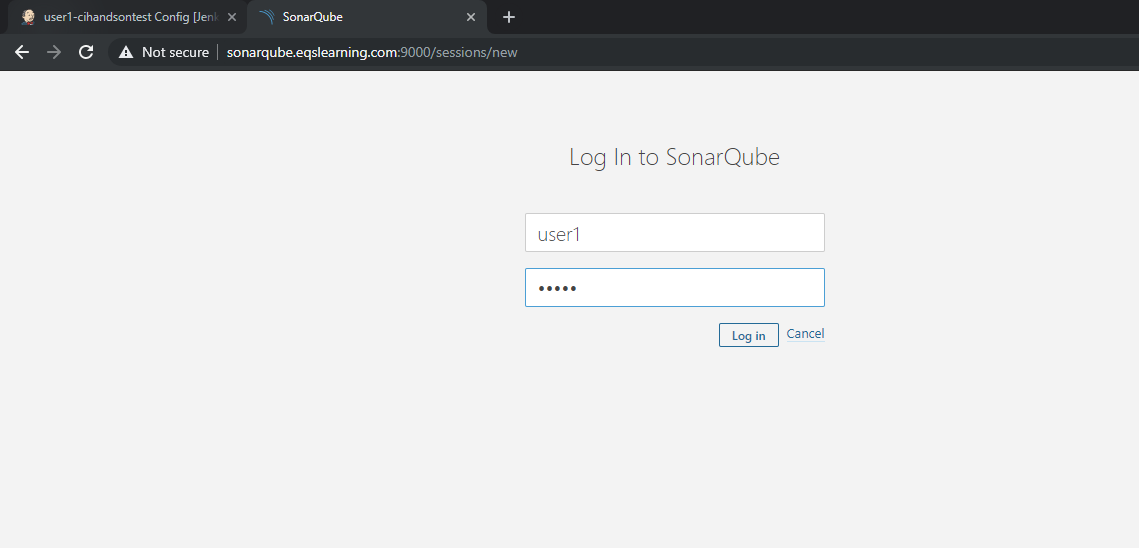


Enter the parameter name as **jenkins\_username** and enter the default value as your username.

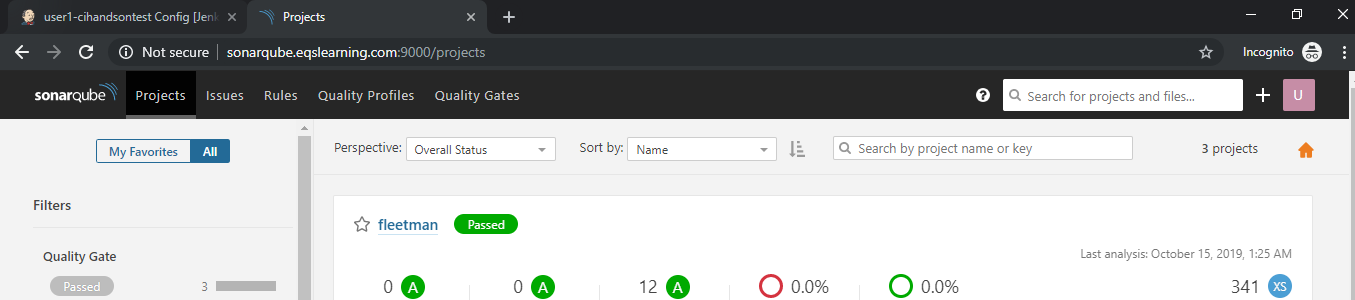


# Generating sonarqube for build token

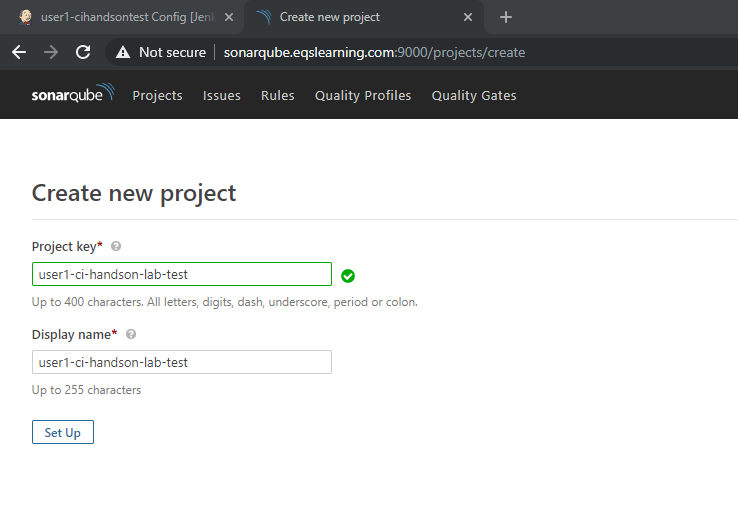
Logon to sonarqube in a seprate tab/page using URL [http://sonarqube.eqslearning.com:9000](http://sonarqube.eqslearning.com:9000/) with your user name and credential which was provided for Jenkins.



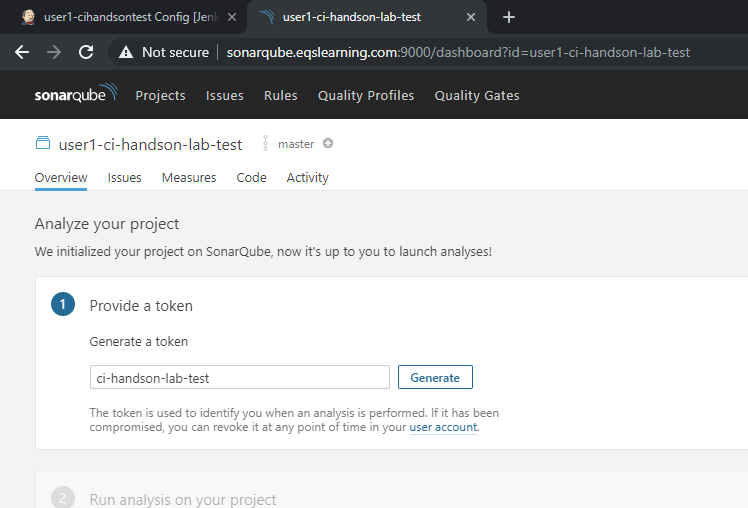
After logon, click on + symbol on the right top corner and select the option “**create new project**”.



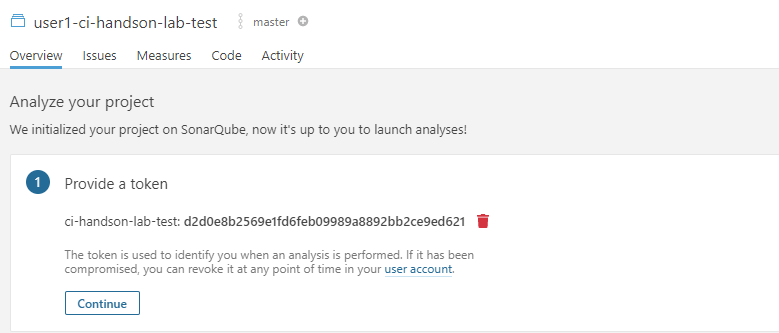
Enter the appropriate name for the project with naming convention and add a description for the project. Click on Setup to get the access token.



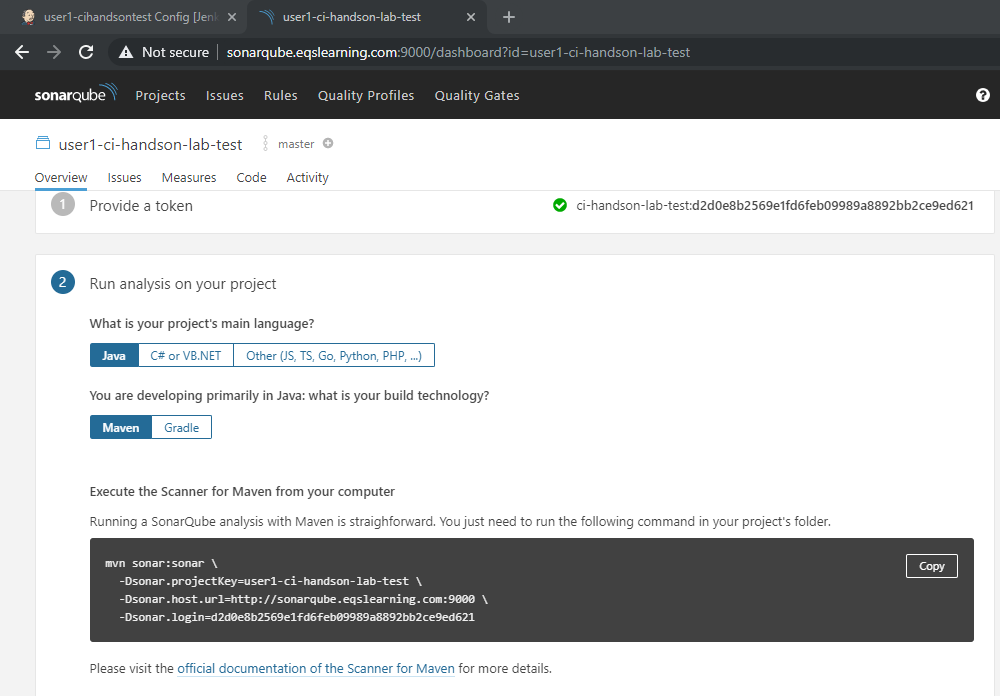
Enter a description to get a new token for the project and click on generate button.



Upon new key generated, click on Continue to get command which we need to execute in our Jenkins build.



Copy the code generated in sonarqube for maven based build on Java language as shown below.

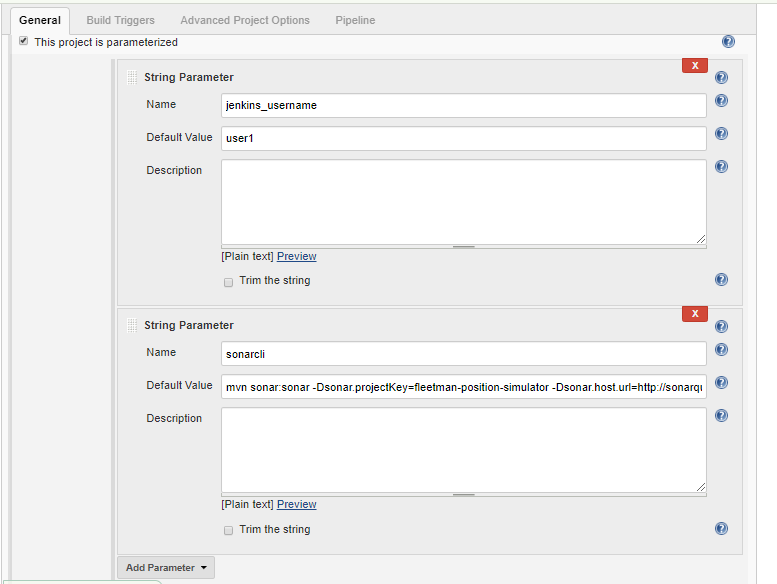


Copy the sonarqube command with toke details and save it for updating in Jenkins job.

## Adding Sonarqube CLI as parameter in pipeline

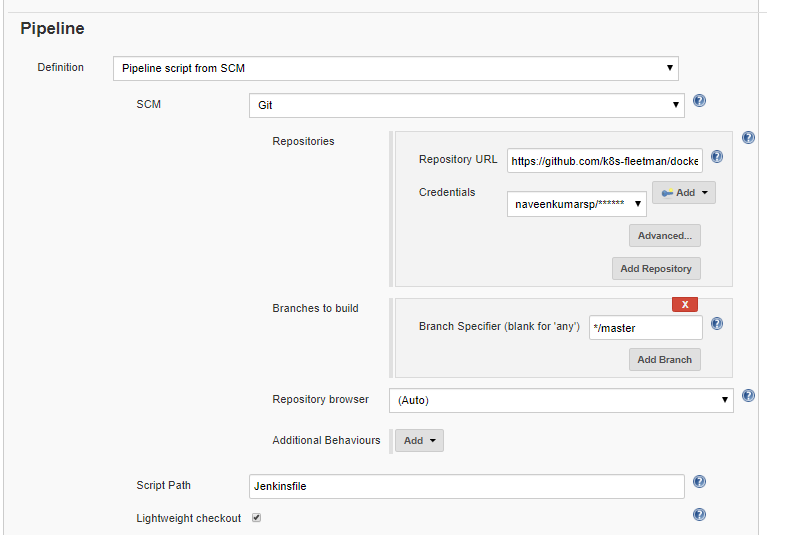
In Jenkins, Click on Add parameters once again and enter the newly generated CLI which was created as mentioned in previous section. Enter the name of the parameter as ‘**sonarcli’** and copy the cli which was copied from sonarqube to default value field.

**Note: make sure you remove the multiline separator i.e ‘\’ from cli before updating to default value field.**



# Configuring Pipeline

On newly created pipeline, go to Pipeline section and update the Definition field to “**Pipeline Scripte from SCM**”. Select the SCM as ‘**Git**’ and update the Repository URL as listed below as well as credentials. Up on all updates listed above, click on **Apply/save** button.

<https://github.com/k8s-fleetman/docker-java-helloworld-cd-pipeline>

# Jenkinsfile - Explained

For versioning purpose, Typically Jenkins file will be kept in code repository. We have written a simple Jenkinsfile which will define the multiple build stages/workflows which get executed as part of build setup. Here is the pipeline which is listed in our GitHub repo which will trigger the jobs in sequential order.

pipeline {

agent any

environment {

// You must set the following environment variables

// ORGANIZATION\_NAME

// YOUR\_DOCKERHUB\_USERNAME (it doesn't matter if you don't have one)

SERVICE\_NAME = "docker-java-helloworld-cd-pipeline"

IMAGE\_NAME = "ci-cd-demo-${jenkins\_username}"

REPOSITORY\_TAG="${DOCKERHUB\_URL}/${IMAGE\_NAME}:${BUILD\_ID}"

}

stages {

stage('Preparation') {

steps {

cleanWs()

git credentialsId: 'GitHub', url: "https://github.com/${ORGANIZATION\_NAME}/${SERVICE\_NAME}"

}

}

stage('Update user references') {

steps {

sh 'cat ./src/main/webapp/index.jsp'

sh """sed -i 's+Admin+'"${jenkins\_username}"'+' ./src/main/webapp/index.jsp"""

sh 'cat ./src/main/webapp/index.jsp'

sh """sed -i 's+jenkins\_username+'"${jenkins\_username}"'+' deploy.yaml"""

sh """sed -i 's+REPOSITORY\_TAG+'"${REPOSITORY\_TAG}"'+' deploy.yaml"""

sh """sed -i 's+jenkins\_username+'"${jenkins\_username}"'+' ingress-routing.yaml"""

}

}

stage('Build') {

steps {

sh '''mvn clean install package'''

}

}

stage('SonarQube') {

steps {

sh '${sonarcli}'

}

}

stage('Build Image') {

steps {

sh 'scp -r ${WORKSPACE} jenkins@${DOCKER\_HOST\_IP}:/home/jenkins/docker/${BUILD\_ID}'

sh 'ssh jenkins@${DOCKER\_HOST\_IP} docker image build -t ${REPOSITORY\_TAG} /home/jenkins/docker/${BUILD\_ID}'

sh 'ssh jenkins@${DOCKER\_HOST\_IP} docker image ls'

sh 'ssh jenkins@${DOCKER\_HOST\_IP} rm -rf /home/jenkins/docker/${BUILD\_ID}'

}

}

stage('Push Image to repo') {

steps {

sh 'ssh jenkins@${DOCKER\_HOST\_IP} docker push ${REPOSITORY\_TAG}'

}

}

stage('Deploy the application') {

steps {

sh 'cat deploy.yaml'

sh 'kubectl apply -f deploy.yaml'

sh 'cat ingress-routing.yaml'

sh 'kubectl apply -f ingress-routing.yaml'

}

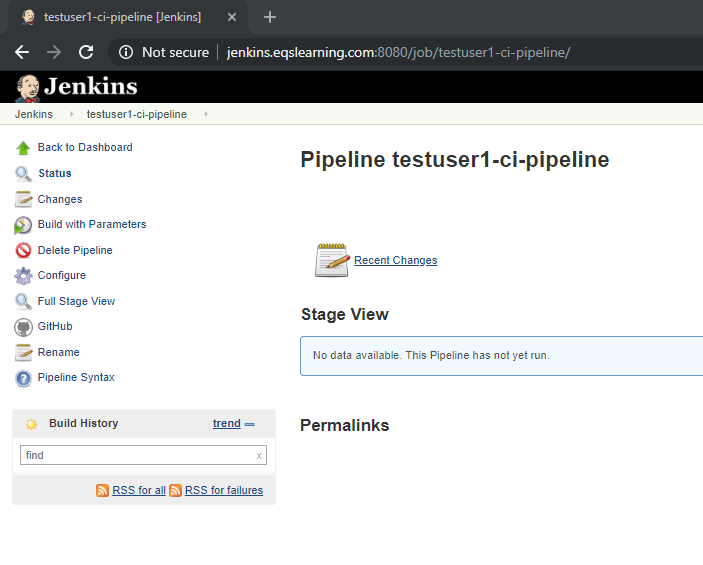
}

}

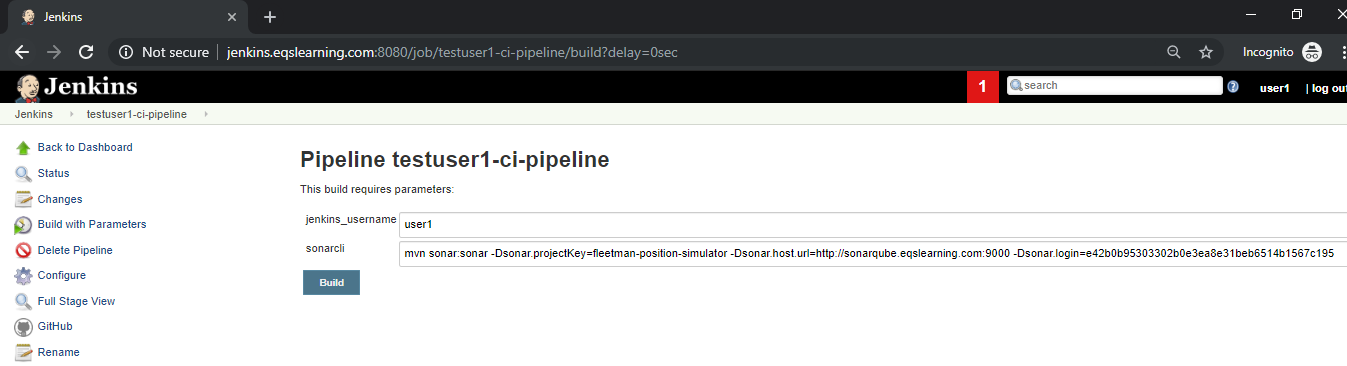
}

# Triggering the Jenkins pipeline

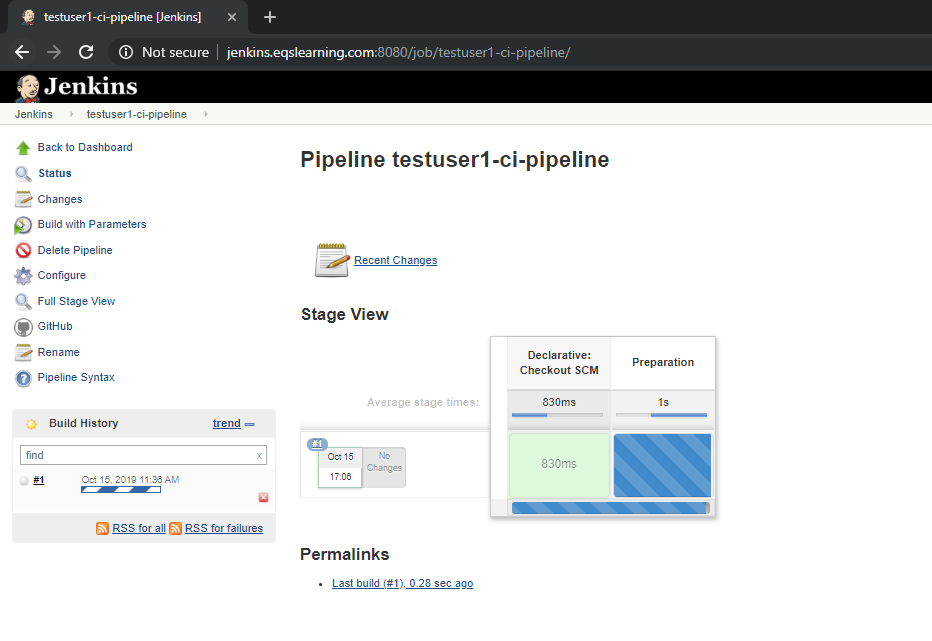
In order to trigger the pipeline, navigate to your pipeline and click on “**Build with parameters**” listed on left side menu of the page.



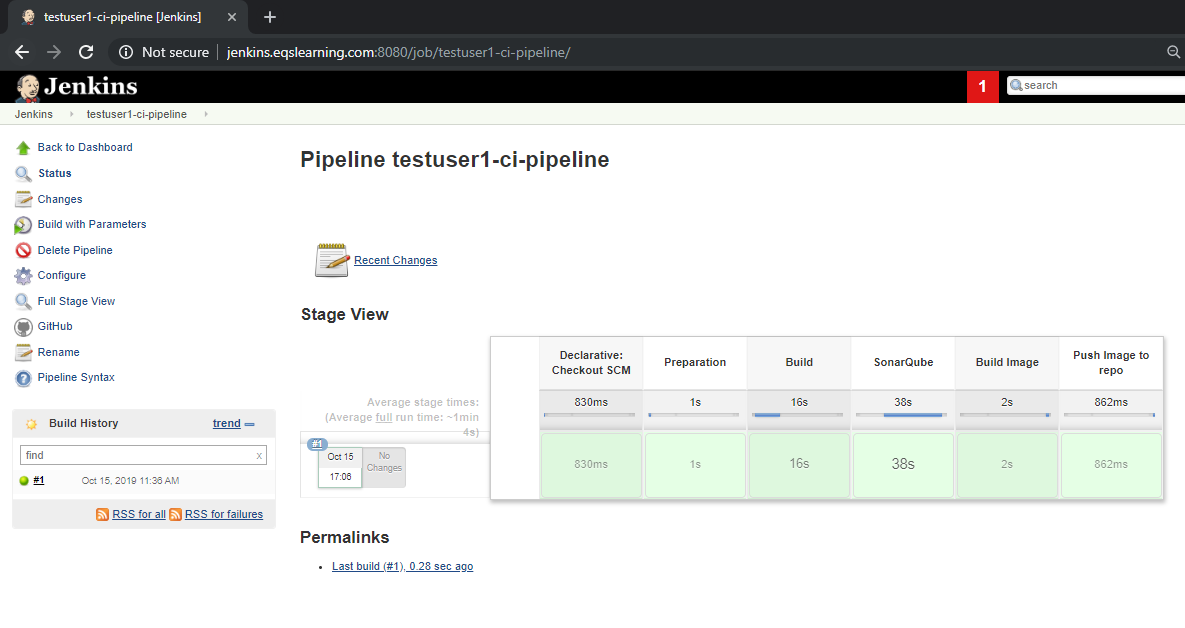
Parameter page lists all the default value which you have updated. Validate the value and click on build to trigger the pipeline.



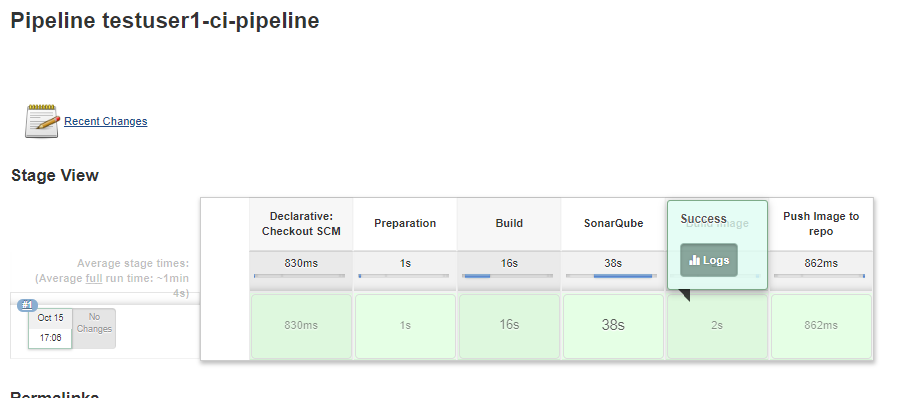
After couple of seconds, you will be able to see the pipeline stage view get updated and you will be able to see the progress. For the first time you may notice the build will create the stages one after the another.

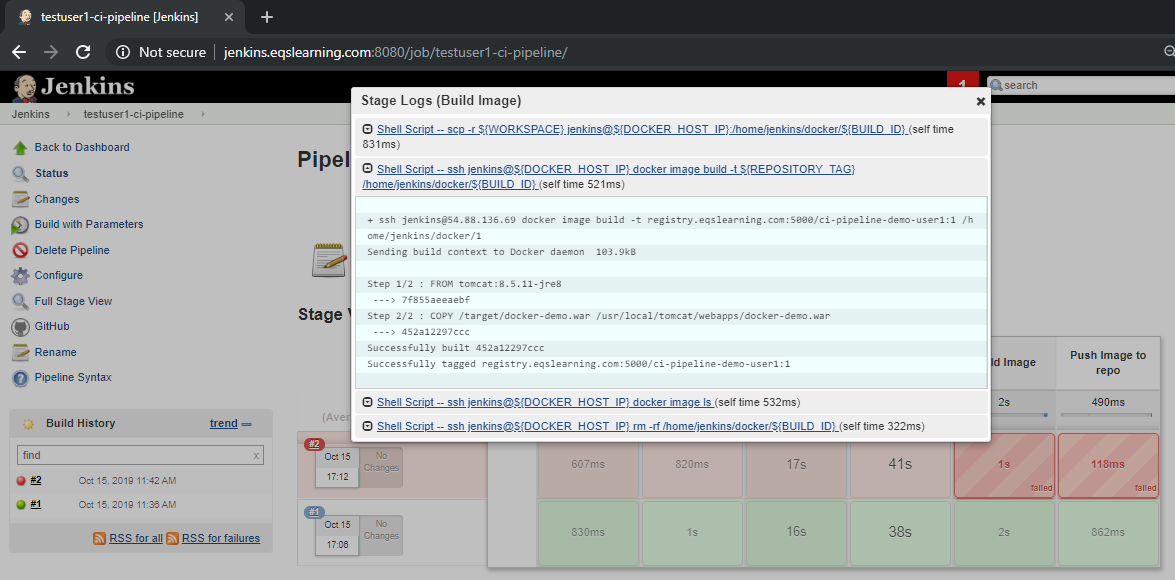


Up on completion of the Jobs, you may see the end to end pipeline setup.



You may review the logs on each stage and verify the entire process. It will be really helpful to review failures and validate the build workflow.

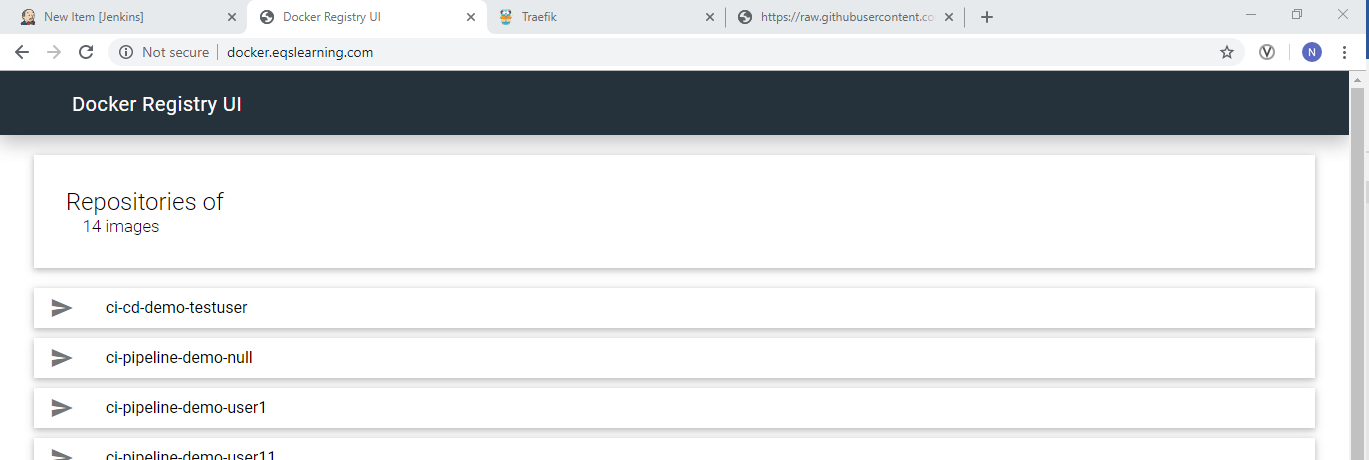




# Verify the Docker image in registry

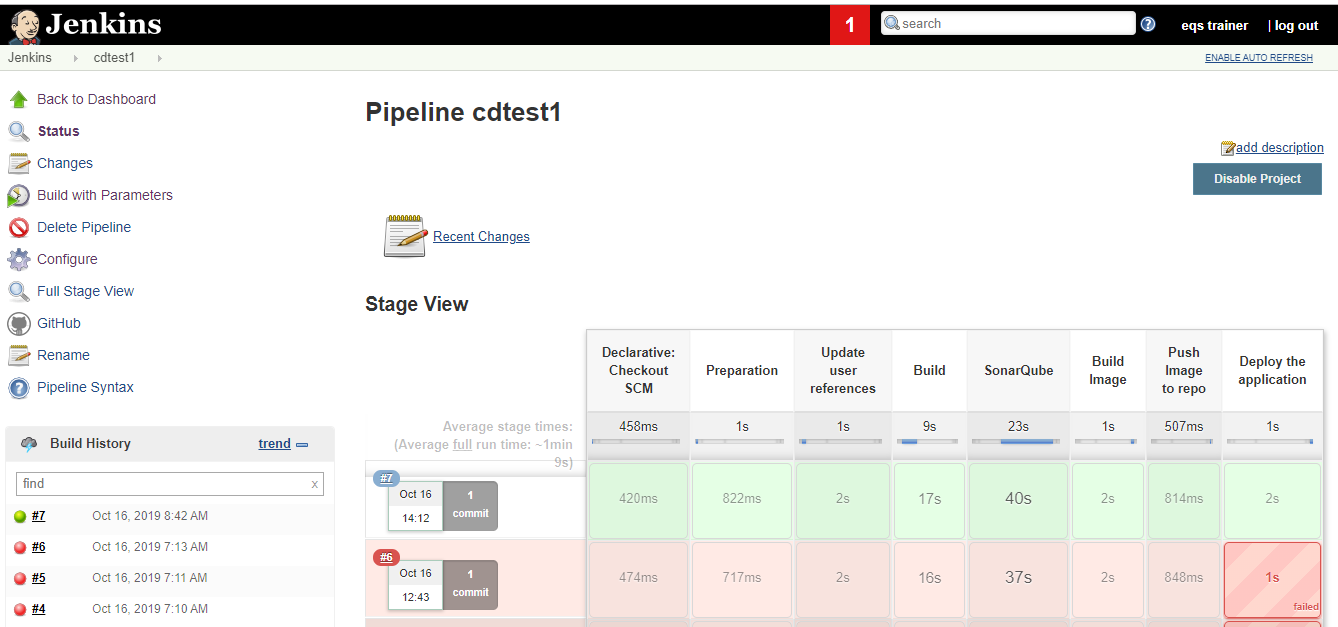
As part of our pipeline, we will be building the application and validating the code quality. If the build completes these two stages successfully, Jenkins job builds a Docker image and pushes it to private registry for deployment. i.e registry.eqslearning.com:5000. You may verify the docker image listed in UI of the registry by accessing the URL [http://docker.eqslearning.com](http://docker.eqslearning.com/).

You may verify the newly built docker by locating **ci-cd-demo-<userid>.** e.g: user named ‘**testuser’** has trigged the CD Pipeline, hence Jenkins pipeline has created a Docker image named ‘**ci-cd-demo-testuser**’.



# Pipeline stages - Explained

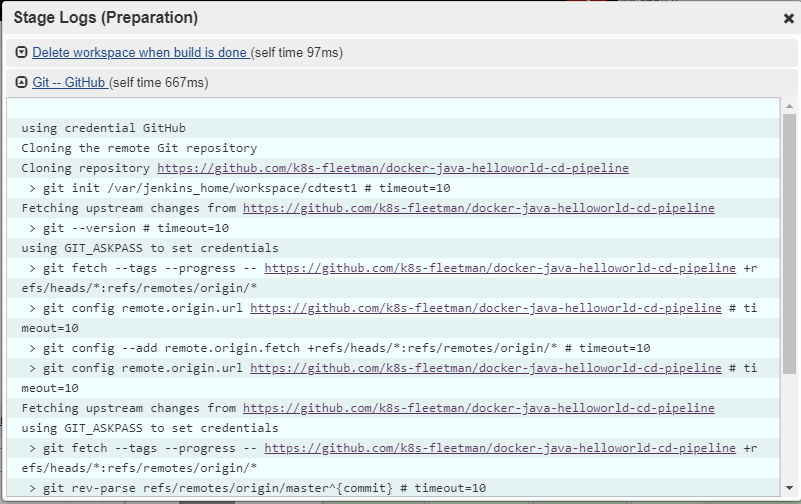
Application which we have deployed is a simple Java application display hello <username>! In the webpage.



## Declarative Checkout SCM and Preparation

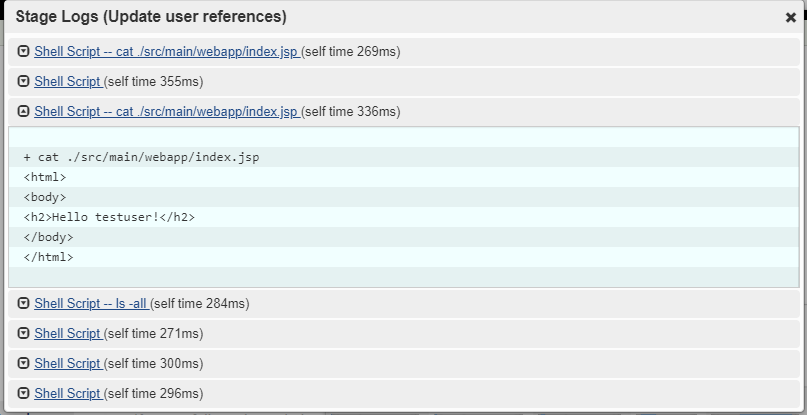
Declarative checkout SCM is inbuilt step which pulls our Jenkinsfile where we have defined the stages to be executed for our build job. However in preparation we are pulling the SCM code. This is one apt example where we can keep Jenkinsfile in a repository and the source code can be anywhere else.

We are deleting the Jenkin workspace of the build and cloning the source code using the Jenkins Credentails storage in the name **GitHub.**



## Update user references

Since we are using common repository, we are replacing the user reference in web application and deployment scripts. We will be replacing the keyword Admin from Index.jsp to username which you have updated as parameters while creating the pipeline.



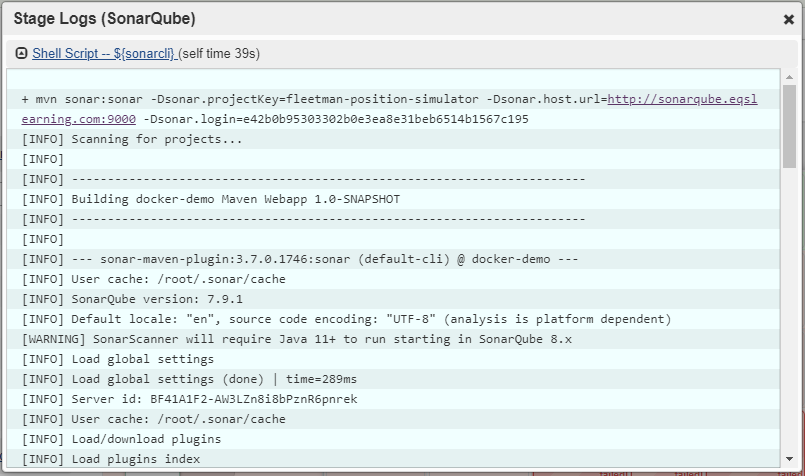
## Build

Using maven, we have built the application which has generated the docker-demo.war file under directory /target.



## SonarQube

Code quality check was performed using sonarqube. The command is passed on via parameter named ‘sonarcli’ which was declared while creating the pipeline.

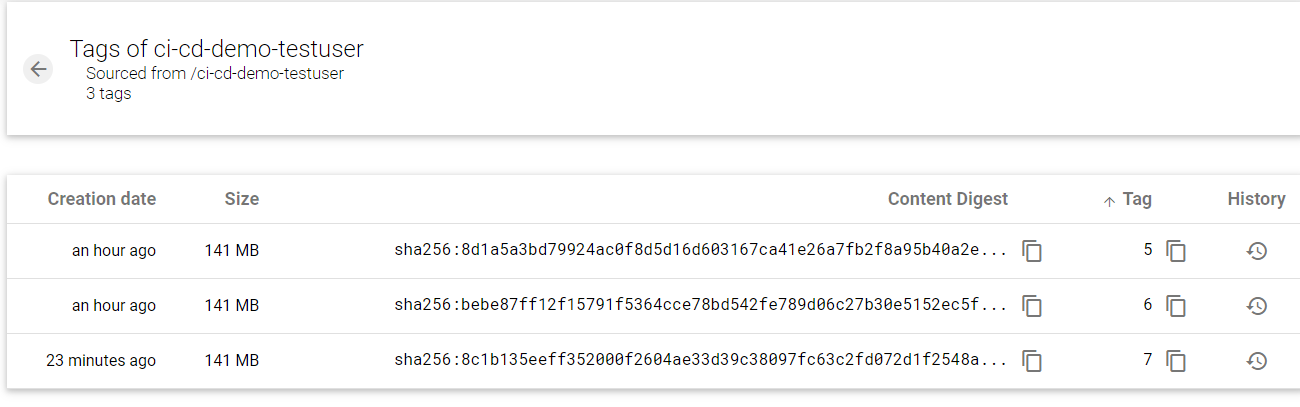


## Build Image

Since our Jenkins is running on a container, we cannot build the Docker image. Hence we are copying the entire workspace to a Docker host and building the docker image.

In build Build image stage, based on the Dockerfile specification, we have pulled a public image of tomcat:8.5.11-jre8 and pushed the docker-demo.war file to /usr/local/tomcat/webapps/docker-demo.war in the newly created container image.

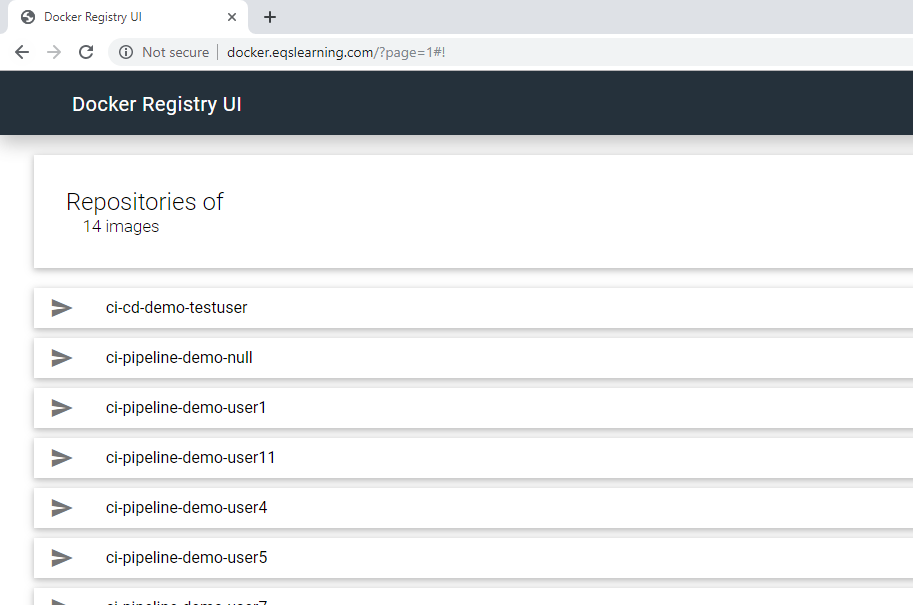
Note: While building the Docker image, we have tagged the version number as the build ID of the Jenkins job. E.g : If the current Jenkins job of the pipeline is 5, then our docker image tag will be ci-cd-demo-testuser:5.



## Push Image to repo

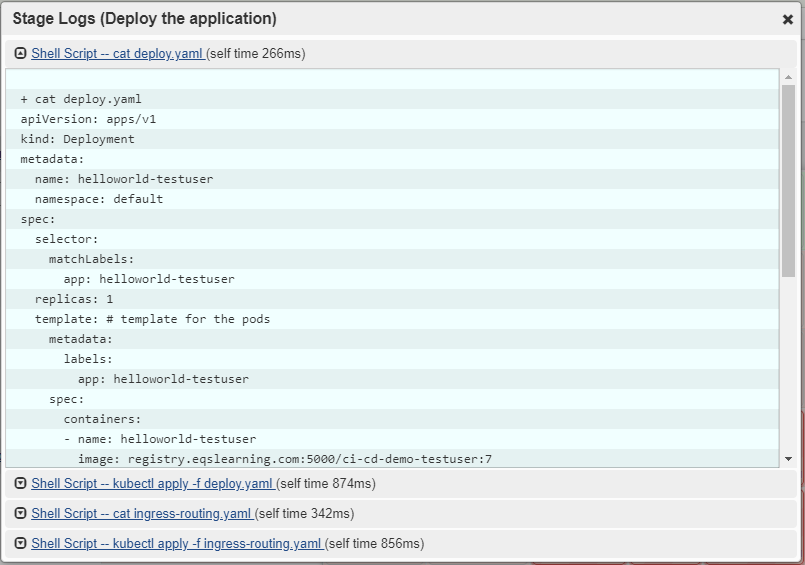
In push image stage, we have pushed the image to our private registry hosted internally on ‘registry.eqslearning.com:5000’. Registry is can only be accessible only via API, hence we have deployed a Registry UI container which displays the images.

Note: The pull command which you get from the UI does not work as it is just used for visualization.



## Deploy the application

During Deploy stage of the pipeline, we have used the same Docker image which will be pulled from our private registry and deploys the application in Kubernetes cluster and exposes the traffic to external DNS address.

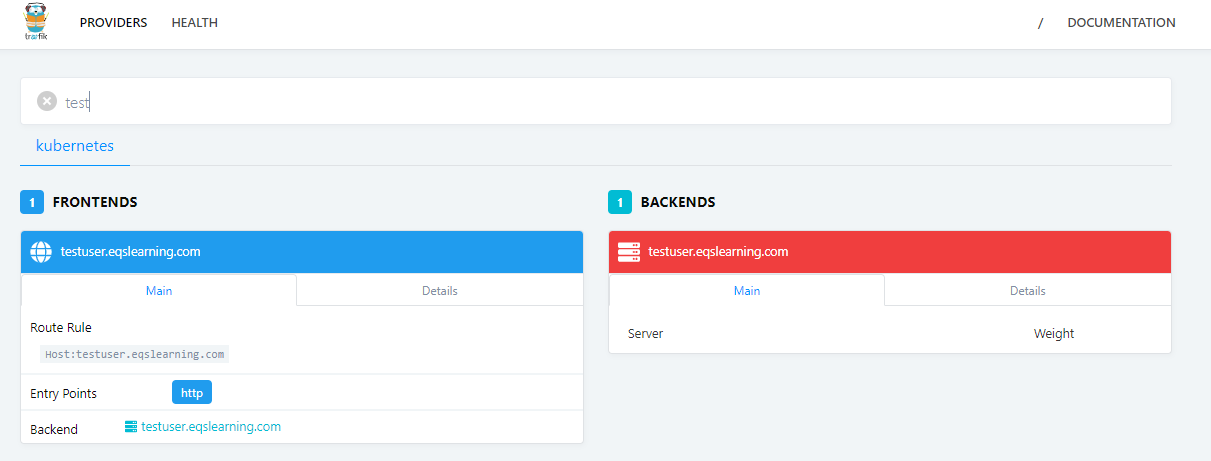


# Verifying the Deployed application

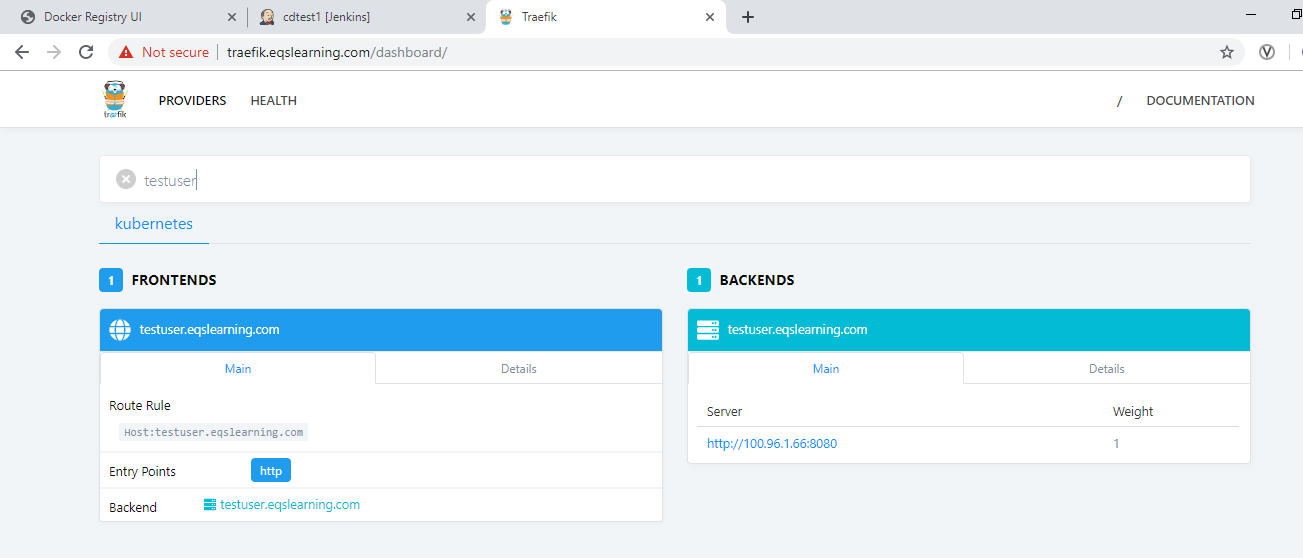
Once the deploy is successful, we can validate the application on public DNS address. For convenience, we have pre-setup the URL which points to our application. The naming convention for the DNS address is <username>.eqslearning.com. e.g. for ‘testuser’ the DNS address would be testuser.eqslearning.com.

## Accessing Traefik dashboard

Since the deploy takes roughly about a minute to pull and run container. However, the Ingress service which maps the traffic take couple of minutes. Hence Validate your application service on traefik dash board. <http://traefik.eqslearning.com/dashboard/>. Here in below screenshot which was capture depicts how Frontend URL from the external source is being mapped to backend service. Since backend service is not ready, you will be seeing the service ribbon is red color.

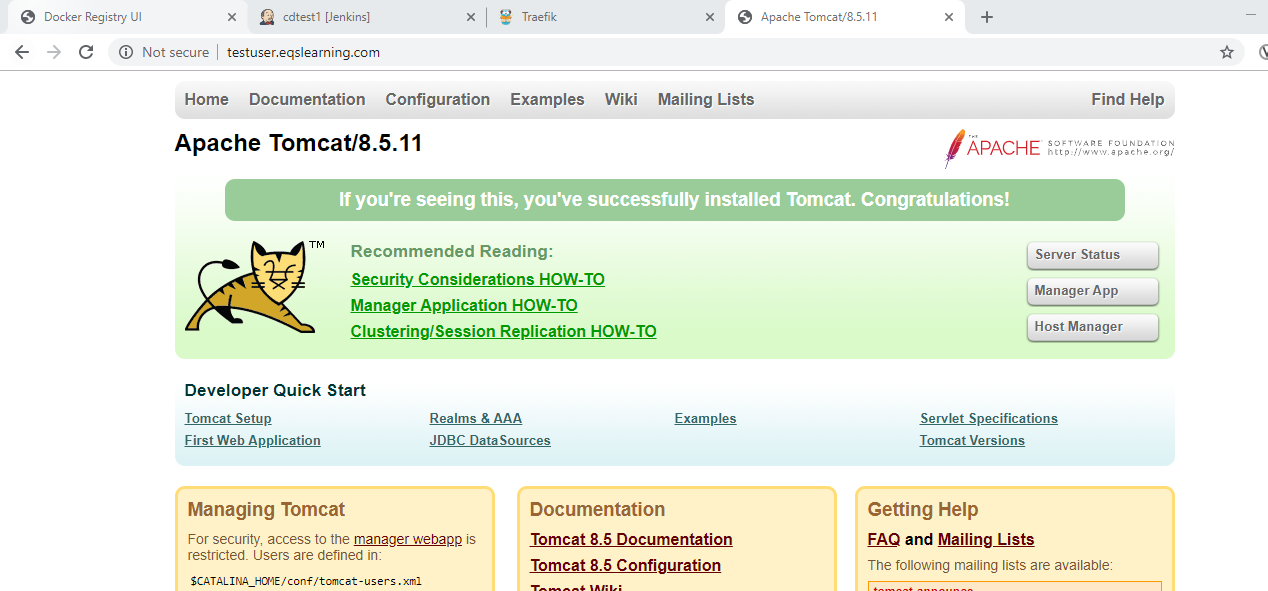


Once the backend service(container) is ready, you may access the external domain address/URL.



## Accessing the external Domain address

Default page of the mapped URL shows default the apache tomcat page of the container which was deployed.



Kindly access the URL address <username>.eqslearning.com\docker-demo\ to validate deployed web application. E.g : <http://testuser.eqslearning.com/docker-demo/>

