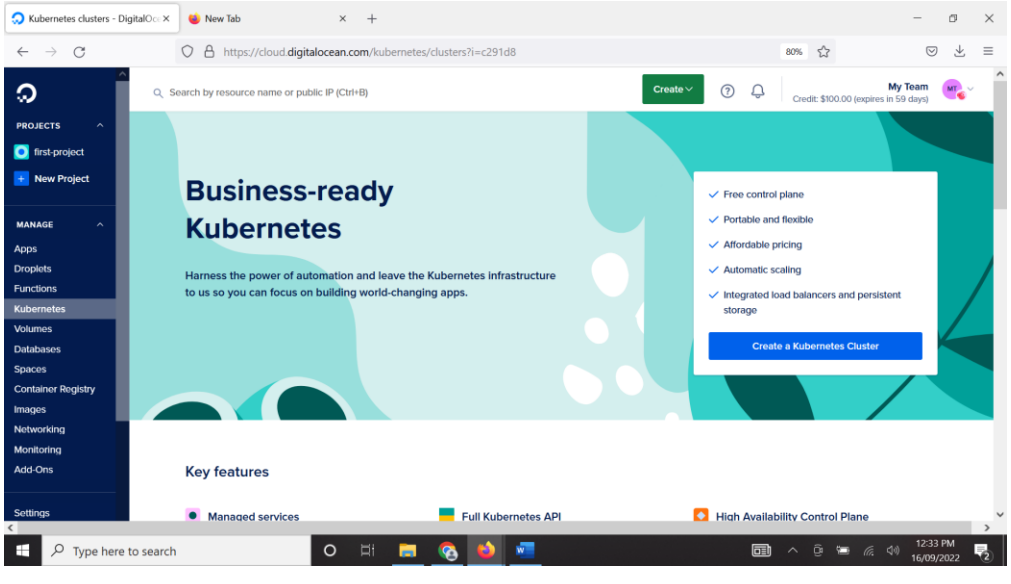
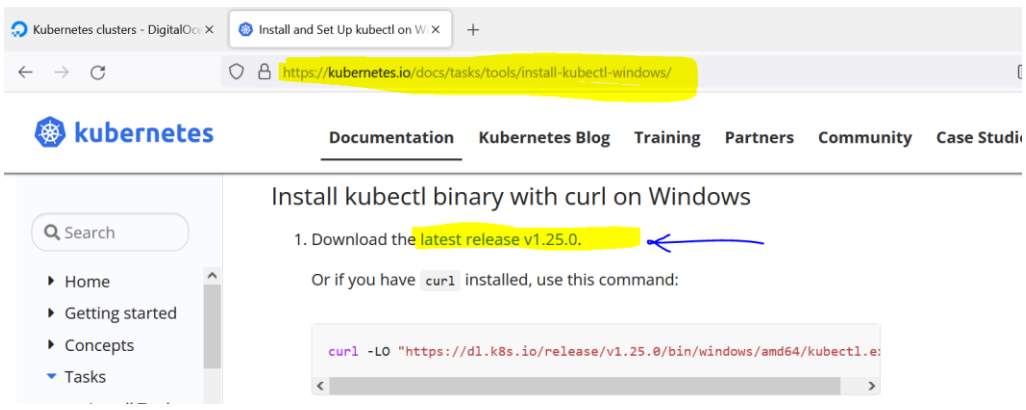
JENKINS PROJECT END TO END WITH KUBERNETES PODS

Step 1: First create a account in Digital Ocean:

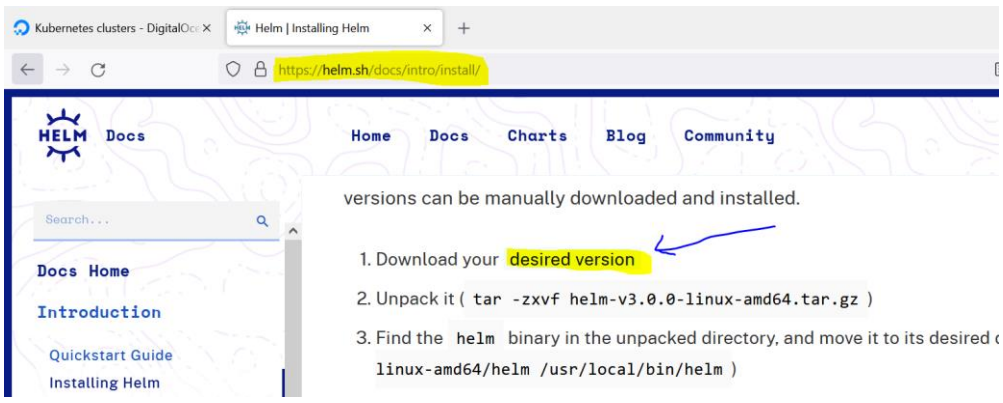


Step 2: Install Kubectl and Helm on Windows as we are using our Windows laptop

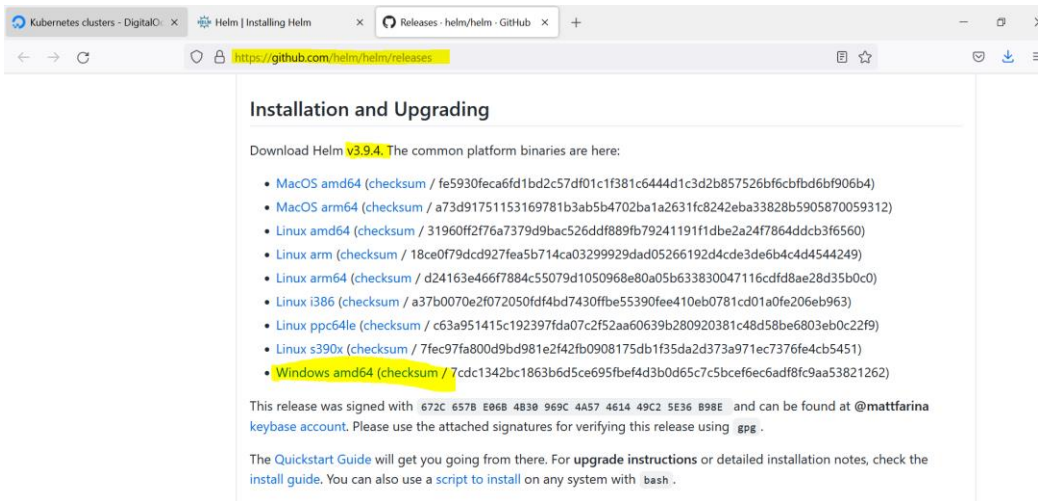
NOTE: You can use the official link to download



You can download the desired version of HELM, for that click on this link



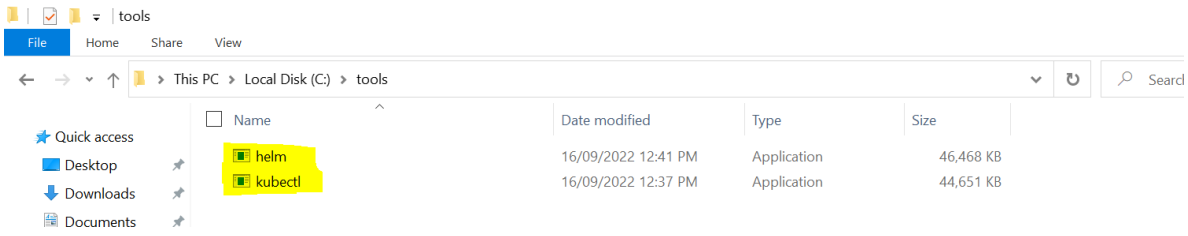
It will take you to this page, here we are downloading the v3.9.3 Windows amd64 file



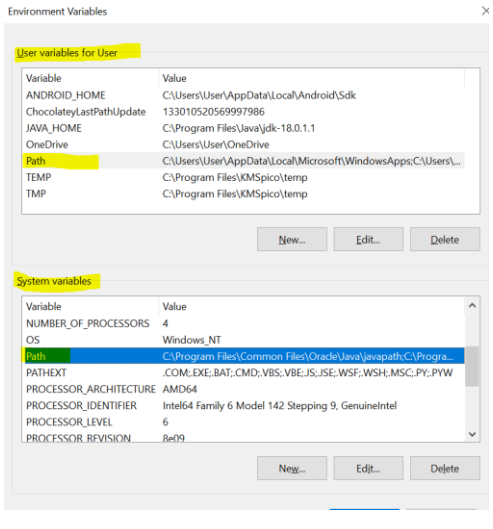
Now after downloading this ***kubectl*** and ***helm*** we need to move them to a folder and add that folder to a path so that from anywhere you can run ***kubectl*** and ***helm*** commands. This is just like adding the location of the binary to the system environment variable in case of linux.

Before moving helm you need to extract that file as it's in zip format.

I created a folder called ***tools*** and moved ***kubectl*** and ***helm*** exe files inside it.

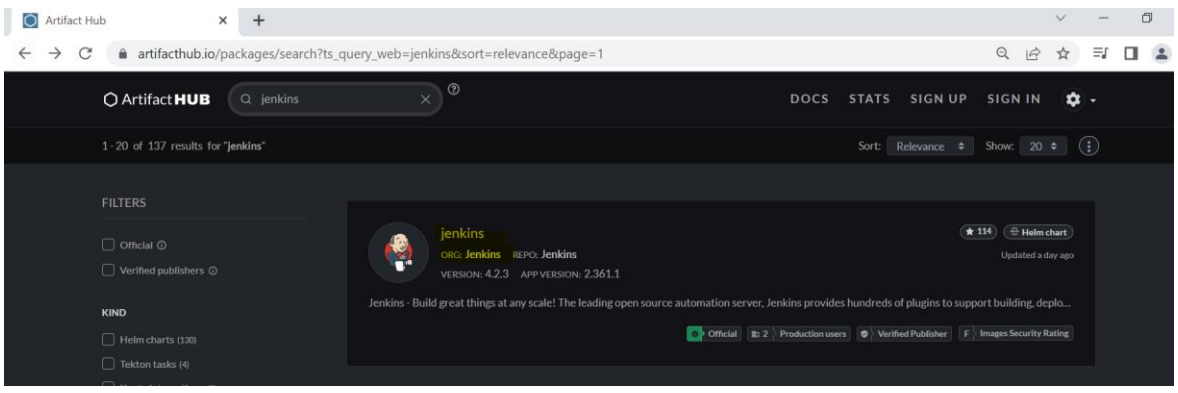


Now add this tools location to the ***path*** variable of your U***ser variable*** and ***System variable*** as highlighted.

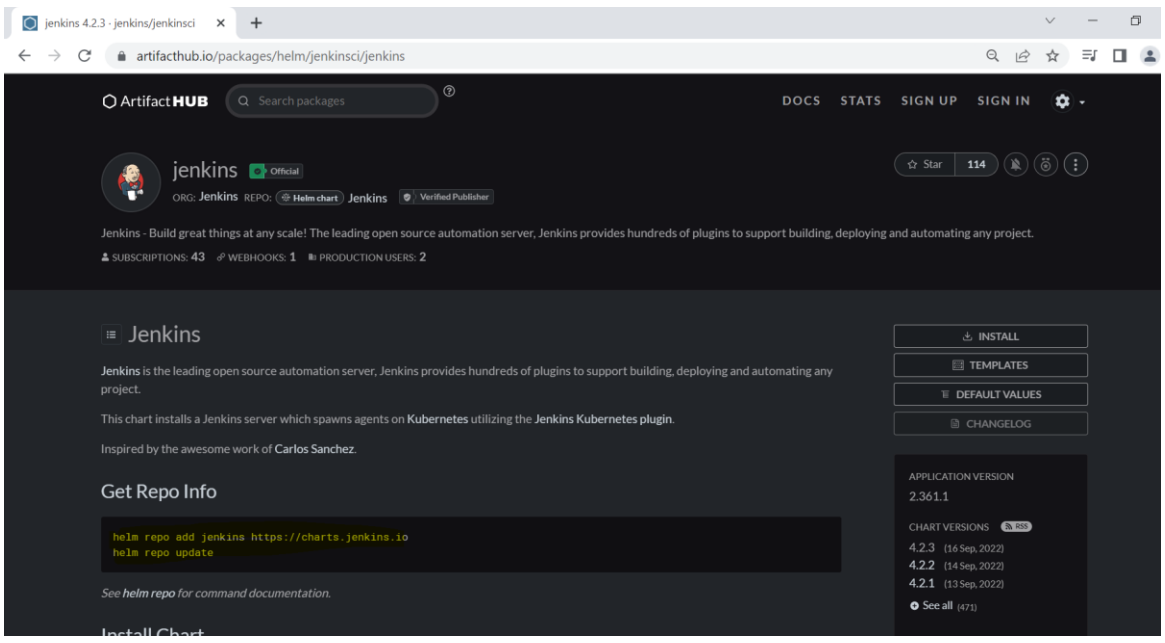


Step 3: Now we need to install Jenkins through HELM charts, for this we need to first check if our Kubernetes cluster is up and running.

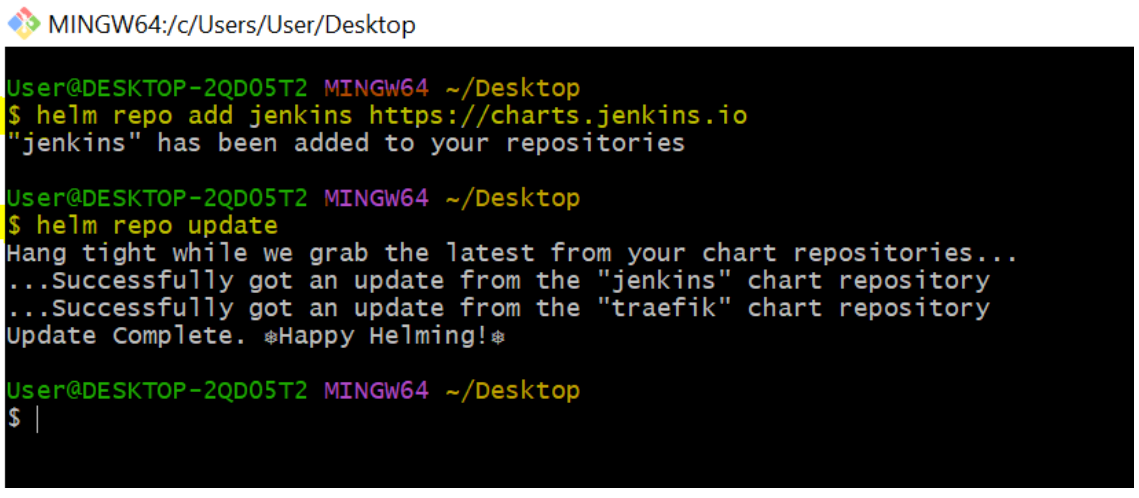
Now we search for a Jenkins HELM chart on artifacthub.io as shown below. We can take this copy as it's a official image.



We can run these commands to add the Jenkins helm chart.



Now the Jenkins HELM chart is added and its updated as shown below.



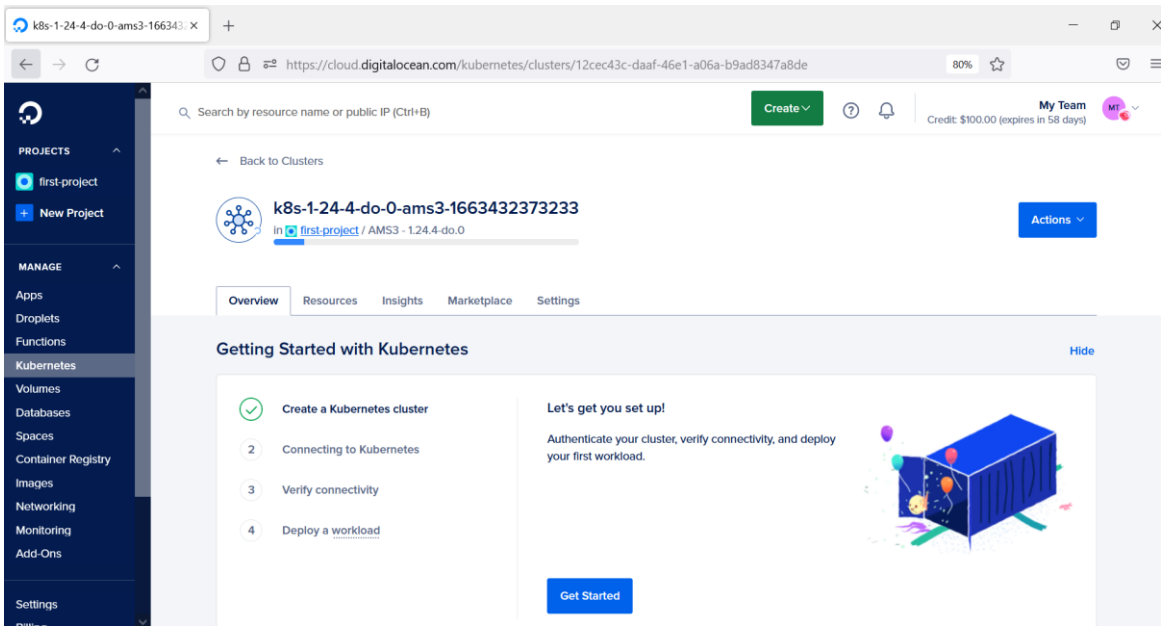
Now if you install the Jenkins helm chart then it will automatically pulled from the remote location and it will get installed in your Kubernetes cluster but you won't get a chance to configure the helm chart. Hence, to do custom configuration the helm chart, first we need to pull the Jenkins helm chart as shown below.

Here, I am pulling the Jenkins charts and using **--untar**, I am un-tarring it while pulling at the same time.

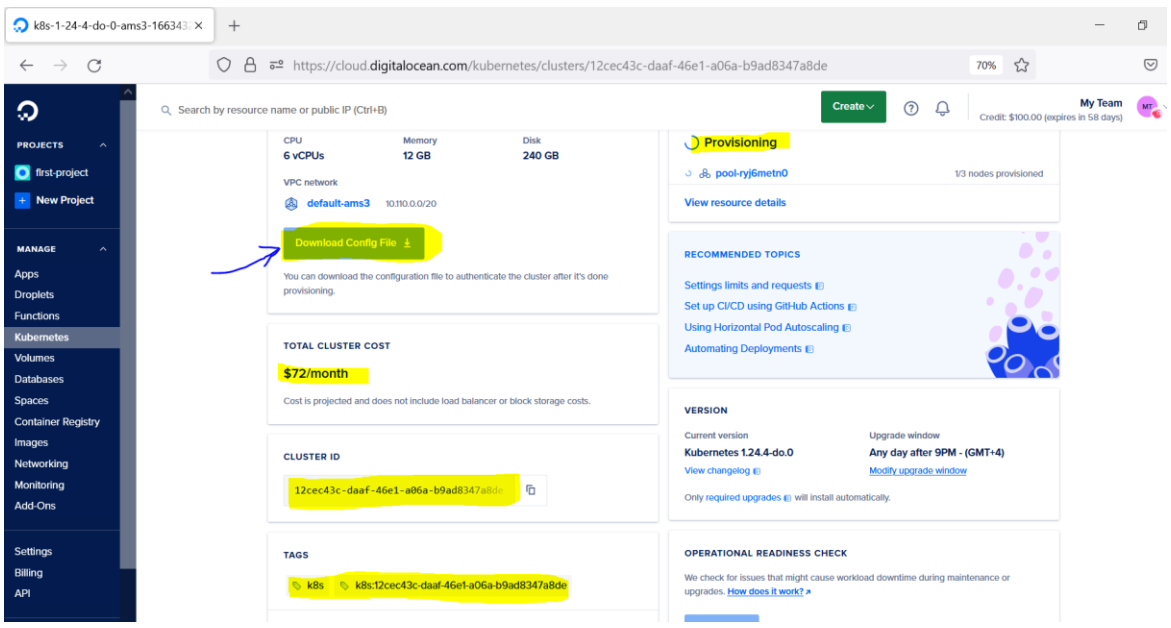
Syntax: ***helm pull RepositoryName/chartName --untar***

```
User@DESKTOP-2QDO5T2 MINGW64 ~/Desktop
$ helm pull jenkins/jenkins --untar

User@DESKTOP-2QDO5T2 MINGW64 ~/Desktop
$
```

Now, it's a time to create Kubernetes cluster, You can click on Kubernetes option on the side menu and click on Create a Cluster then you will see the below page.



Once the Kubernetes get installed completely then you can download the Config File for connecting the cluster.



Once the Config File is downloaded then if you do cat on that downloaded file then you can see the below details with cluster name, current-context info, user info etc which will be useful in future if it's needed.

```
User@DESKTOP-2QD05T2 MINGW64 ~/Downloads
$ cat k8s-1-24-4-do-0-ams3-1663432373233-kubeconfig.yaml
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURkekND
QWcrZOF3SUJBZOlDQm5VdORRWUpLb1pJaHZjTkFRRUxCUUF3TXpFVk1CTUdBMVVFQ2hNTVJHbG4KYVhS
aGJFOWpaV0Z1TVJvdOdBWURWUVFERXhGck9ITmhZWE1nUTJ4MWMzUmxjaUJEUVRBZUZ3MHlNakE1TVRj
eApOak0wTkRGYUZ3MDBNakE1TVRjeE5qTTBOREZhTURNeEZUQVRCZO5WQkFvVERFUnBaMmwwWVd4UFky
VmhiakVhCk1CZOdBMVVFQXhNUmF6aHpZVOZ6SUVOc2RYTjBaWElnUTBFd2dnRWlNQTBHQ1NxR1NJYjNE
UUVCQVFVQUE0SUIKRHdBd2dnRUtBb0lCQVFDOU5MMUdVdFN2YzNZZU1xMURhTFliWVZiMDh6R1NFQVFU
UWRTVHM1ekw3eWNZcFpvZgo4Sm9QQ1RrSmtzZU44VXNNVVN2TFRxZ21QdFlUVO15WHZnLzQyWE9zdzQ3
OXBSWjF4MFdCOWUvKzFHUVdoSDg1CnVRb2lPelNxVnVUOXBBTmtwT3hNQUk1RlRjTVBjRmQ5aDZReWlC
T0FMYWVpKzFRUmV2WElMQkFjUGU5UllPWWIKbkFHN1ZKYlVqRWQ3SnY0V0Uyci9qb1dieVZwTE9kaXRa
UmdmMjU1c1hERXg4NnB1amorRGt1ODlOd05EMXVCLwpuYTJOTElDdHhGNGZYeXhTeDVYdGJNQmczNHQr
ZVpPROhkSnhhdU9kL3VEUXVXaHNOemdOWDlQUThIblhPVkQyCk13R0pLNFU5cWI5N1MvWDlLdlhDclhU
VGdNK05IanBmSHNKZEFnTUJBQUdqUlRCRE1BNEdBMVVkRHdFQi93UUUKQXdJQmhqQVNCZO5WSFJNQkFm
OEVDREFHQVFIL0FnRUFNQjBHQTFVZERnUVdCQlFhRDd2V1BXakl0ZUZmSEhyWgp2d3NrdSsreG1qQU5C
Z2txaGtpRzl3MEJBUXNGQUFPQ0FRRUFNQTJkeEhqaFNMSXRqMlRIMFk5d2RvZFBlRm9lClM2dTBoWGJY
MHR5UE1lbUc4WU9wWE9Dc1MrbXFMTGNzRXc0a3dMZOZIS3luV3NwdnFuY1VKME1kWEU0M2dpTnkKcWZ4
R3QyQkOxNOJTNzJLZEgyQnM1czdmK2pDd1BvUW5Ud3gwZzJrUlRQbVFCbOdmbjVJT3lqWS9lcDd5MTVo
KwpJeHgxOThwT04OOXlqUmJLaktCN21BSXNMN1ZEUGdEQWpGRk5VSVNMbk1CLORzb2FkRHZmSUFnbDQr
VWJFSFVMCnFFajdGYzJ4TzRpN1lnRGNBcTJaN3V2ajJlR3llckFRcVJCdFdCZEN2bDluSjVObjFpZnNQ
Z1BLKOJJQmk2UEoKRlhuTkNteStaZG9KNEprSGdMZzZGaWlRTEoraFh1QXpySVNjc2JmWVlJajVVTnQx
cXFmKyt1ekFrZzO9Ci0tLS0tRU5EIENFUlRJRklDQVRFLS0tLS0K
    server: https://12cec43c-daaf-46e1-a06a-b9ad8347a8de.k8s.ondigitalocean.com
  name: do-ams3-k8s-1-24-4-do-0-ams3-1663432373233
contexts:
- context:
    cluster: do-ams3-k8s-1-24-4-do-0-ams3-1663432373233
    user: do-ams3-k8s-1-24-4-do-0-ams3-1663432373233-admin
  name: do-ams3-k8s-1-24-4-do-0-ams3-1663432373233
current-context: do-ams3-k8s-1-24-4-do-0-ams3-1663432373233
kind: Config
preferences: {}
users:
- name: do-ams3-k8s-1-24-4-do-0-ams3-1663432373233-admin
  user:
    token: dop_v1_df7e749f39c1121be8382b9c33faddec3248be72dcd0c18aaa2e2669a839e4
ee

User@DESKTOP-2QD05T2 MINGW64 ~/Downloads
$
```

Step 4: Now, it's time to access our cluster which we created on DigitalOcean platform, modify the Jenkins charts and install Jenkins using HELM.

Now, if you try to access your cluster, you won't be able to access it.

```
User@DESKTOP-2QD05T2 MINGW64 ~/Downloads
$ kubectl get sc
Unable to connect to the server: dial tcp [::1]:8080: connectex: No connection c
ould be made because the target machine actively refused it.
```

Because when ever we run kubectl command by default it look the config file in the location "***~/.kube/config***". Hence, we can move the above file to this location by following the below direction command.

```
User@DESKTOP-2QD05T2 MINGW64 ~/Downloads
$ cat k8s-1-24-4-do-0-ams3-1663432373233-kubeconfig.yaml > ~/.kube/config

User@DESKTOP-2QD05T2 MINGW64 ~/Downloads
$
```

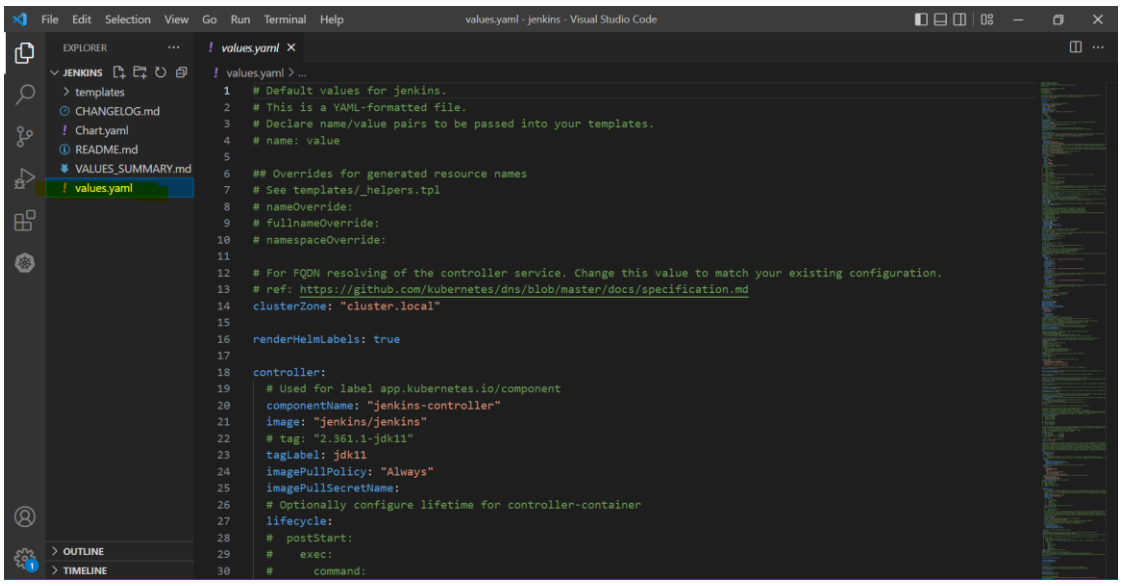
After moving the config file to ~/.kube/config location, you can execute and access your cluster successfully.

```
User@DESKTOP-2QD05T2 MINGW64 ~/Downloads
$ kubectl get sc
NAME                          PROVISIONER                 RECLAIMPOLICY   VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
do-block-storage (default)    dobs.csi.digitalocean.com   Delete          Immediate           true                   61m
do-block-storage-retain       dobs.csi.digitalocean.com   Retain          Immediate           true                   61m
do-block-storage-xfs          dobs.csi.digitalocean.com   Delete          Immediate           true                   61m
do-block-storage-xfs-retain   dobs.csi.digitalocean.com   Retain          Immediate           true                   61m

User@DESKTOP-2QD05T2 MINGW64 ~/Downloads
$ kubectl get pod
No resources found in default namespace.

User@DESKTOP-2QD05T2 MINGW64 ~/Downloads
$
```

Now, the Jenkins chart which I downloaded in Desktop, I moved it to Download folder and opened it using vscode as shown below.

```
MINGW64:/c/Users/User/Downloads

User@DESKTOP-2QD05T2 MINGW64 ~/Downloads
$ code jenkins/
```

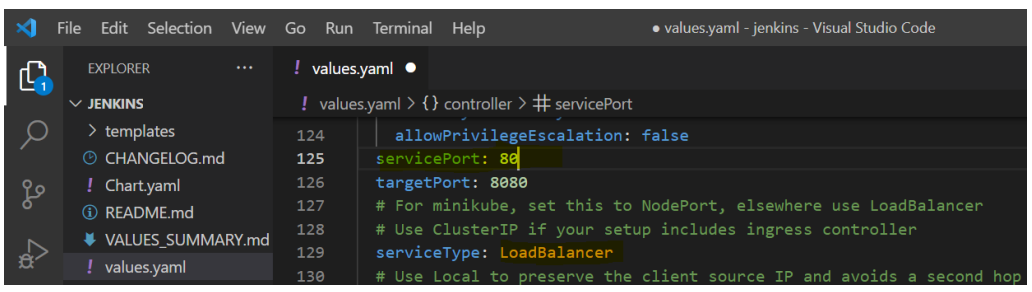
Now, I am going to this values.yaml file and performing some changes which I will mention in below slides.



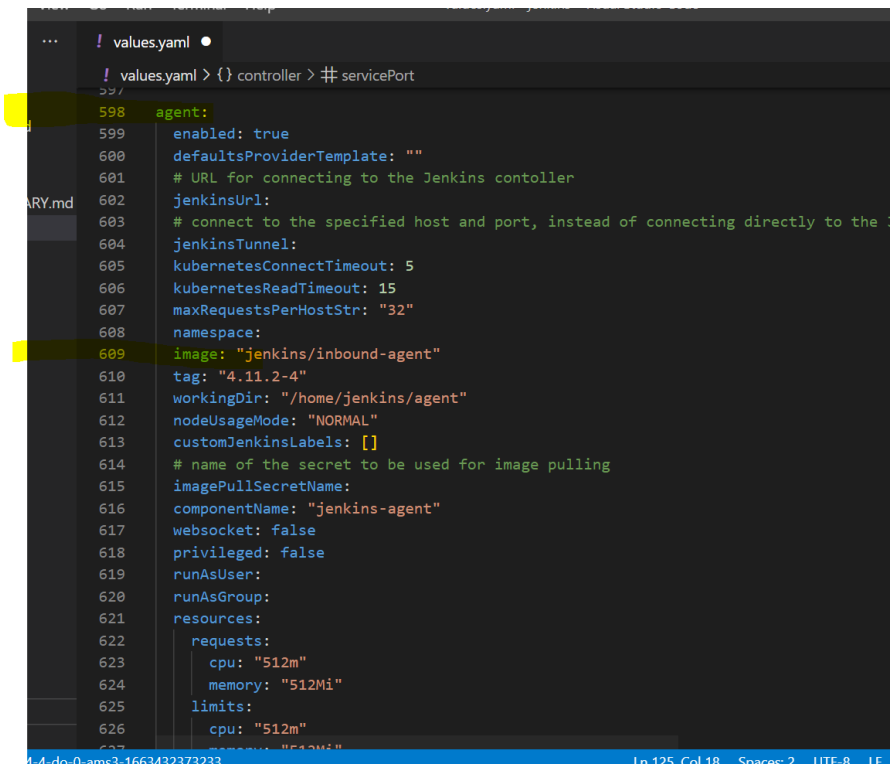
At this place I am doing two changes:

- The servicePort I am changing it from 8080 to 80 so that I can directly access it using the LoadBalancer IP I can access it directly or else if I don't change then I need to access it using *LoadBalancer:8080*
- Second change is, I changed the serviceType from ClusterIP to LoadBalancer.

NOTE: ***If you have Ingress Controller then you can make it as an internal service and use a Ingress resources to redirect the traffic but as we don't have Ingress Controller that's the reason we are redirecting it through the LoadBalancer service.***



Kubernetes by default create agent which will help us to communicate with Jenkins using the 50000 port which is a JNLP port this info you can find in this file. Here, in this screen we are not modifying anything.



Here, you can find that persistence is true and it's using 8GB which is completely fine, so no need to make any changes in this screen.

And here, we not mentioning anything for a storageClass that means it's taking the default storageClass which is provided by DigitalOcean.

values.yaml

values.yaml > { } persistence > enabled

```
#    args: "cat"
#    TTYEnabled: true

persistence:
  enabled: true
  ## A manually managed Persistent Volume and Claim
  ## Requires persistence.enabled: true
  ## If defined, PVC must be created manually before vol
  existingClaim:
  ## jenkins data Persistent Volume Storage Class
  ## If defined, storageClassName: <storageClass>
  ## If set to "-", storageClassName: "", which disables
  ## If undefined (the default) or set to null, no stora
  ##   set, choosing the default provisioner.  (gp2 on A
  ##   GKE, AWS & OpenStack)
  ##
  storageClass:
  annotations: {}
  labels: {}
  accessMode: "ReadWriteOnce"
  size: "8Gi"
  volumes:
  #  - name: nothing
  #    emptyDir: {}
  mounts:
  #  - mountPath: /var/nothing
  #    name: nothing
  #    readOnly: true

networkPolicy:
```

Now, to install this Jenkins chart we can use Install command but for upgrading it again we need to use Upgrade command, instead of it we can directly use Upgrade command because it can handle both the scenarios.

Syntax of the command: *helm upgrade --install releaseName whichFolder*

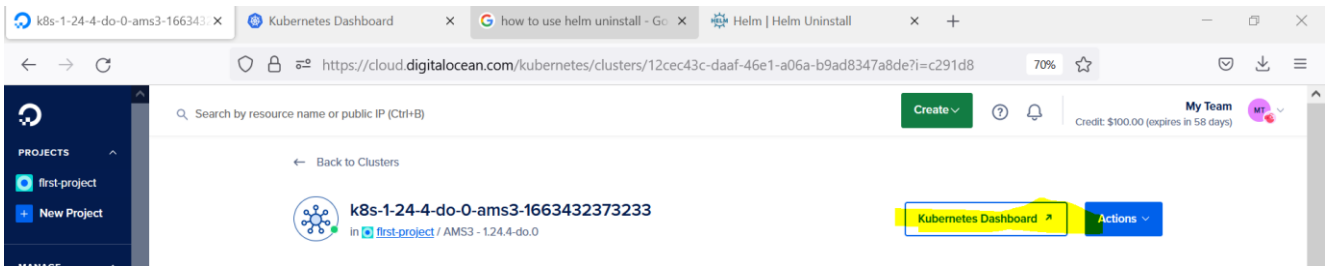
Command: *helm upgrade --install Jenkins jenkis/*

```
User@DESKTOP-2QD05T2 MINGW64 ~/Downloads
$ helm upgrade --install jenkins jenkins/
Release "jenkins" does not exist. Installing it now.
NAME: jenkins
LAST DEPLOYED: Sat Sep 17 21:55:57 2022
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
1. Get your 'admin' user password by running:
  kubectl exec --namespace default -it svc/jenkins -c jenkins -- /bin/cat /run/secrets/additional/chart-admin-password && echo
2. Get the Jenkins URL to visit by running these commands in the same shell:
  NOTE: It may take a few minutes for the LoadBalancer IP to be available.
        You can watch the status of by running 'kubectl get svc --namespace default -w jenkins'
  export SERVICE_IP=$(kubectl get svc --namespace default jenkins --template "{{ range (index .status.loadBalancer.ingress 0) }}{{ . }}{{ end }}")
  echo http://$SERVICE_IP:80/login

3. Login with the password from step 1 and the username: admin
4. Configure security realm and authorization strategy
5. Use Jenkins Configuration as Code by specifying configScripts in your values.yaml file, see documentation: http:///configuration-as-code and examples: https://gi
/jenkinsci/configuration-as-code-plugin/tree/master/demos

For more information on running Jenkins on Kubernetes, visit:
https://cloud.google.com/solutions/jenkins-on-container-engine

For more information about Jenkins Configuration as Code, visit:
https://jenkins.io/projects/jcasc/


NOTE: Consider using a custom image with pre-installed plugins

User@DESKTOP-2QD05T2 MINGW64 ~/Downloads
$
```

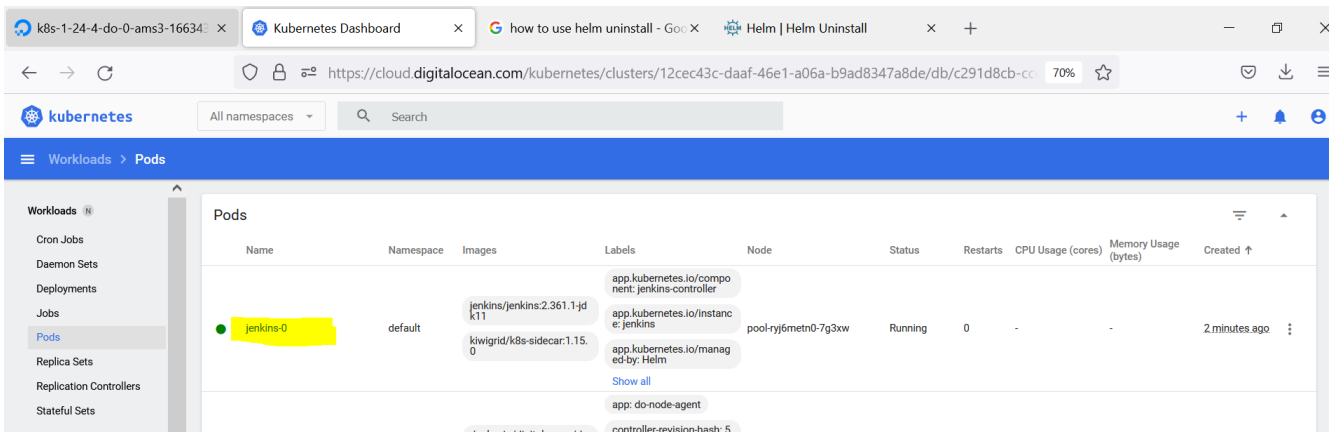
You can see that the jenkins pod is running on your cluster. And you can observe that it is installed as a Statefulset with only one replica

```
User@DESKTOP-2QD05T2 MINGW64 ~/Downloads
$ kubectl get po
NAME        READY   STATUS    RESTARTS   AGE
jenkins-0   2/2     Running   0          116s

User@DESKTOP-2QD05T2 MINGW64 ~/Downloads
$
```

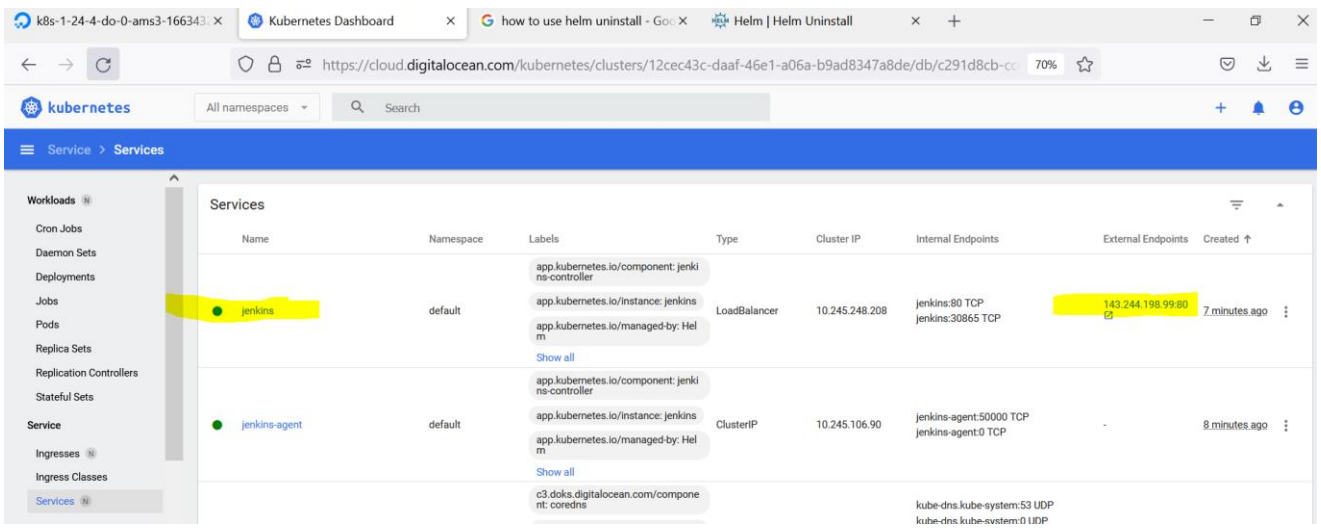
Now this information can be checked on your Kubernetes cluster dashboard by clicking on this button



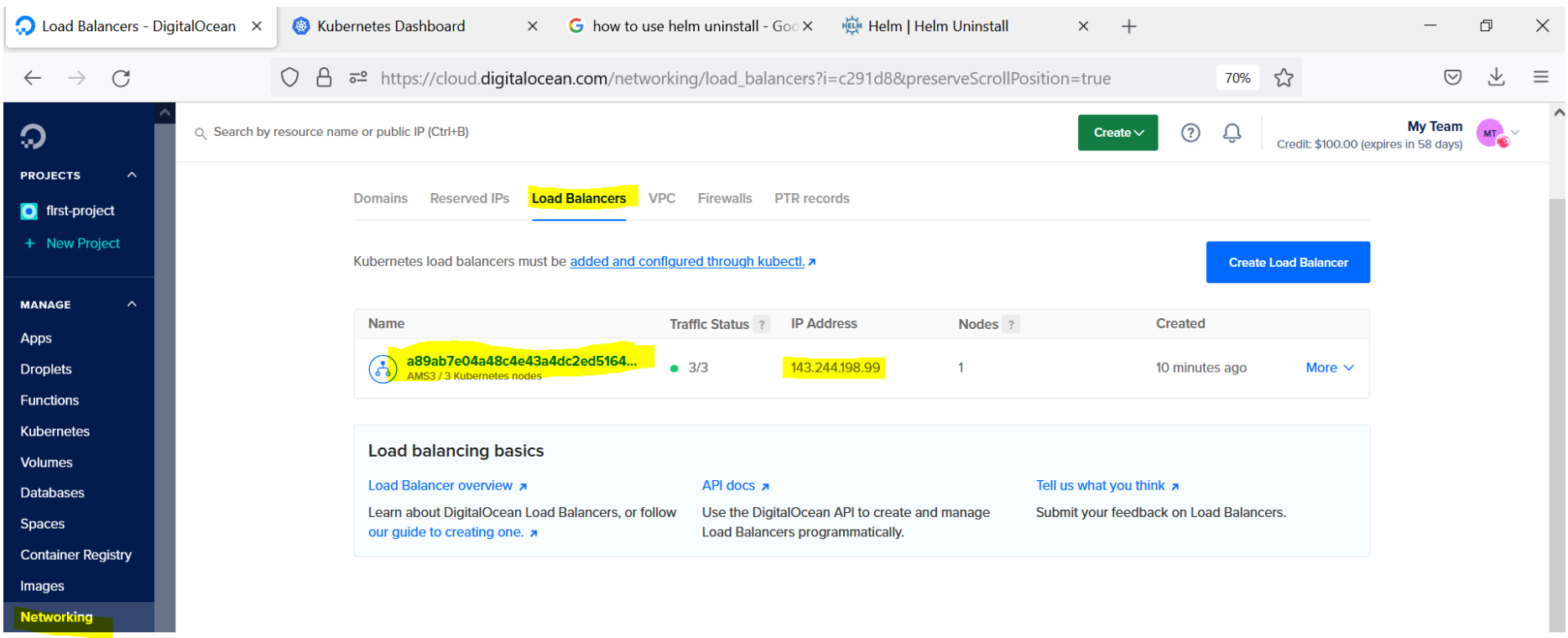
Here, we can observe that it is running with 2 images



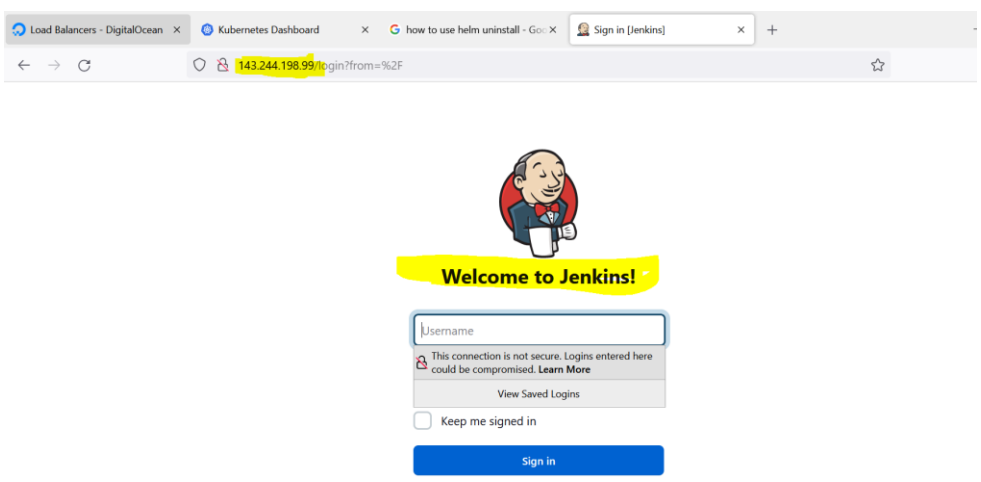
If you go to Services and check it you can observe that it got created as a LoadBalancer



And if you go to Networking and see the Load Balancer then you can observe that a new LoadBalancer is created for Jenkins.



Using this LoadBalancer IP you can open the Jenkins on the browser to cross check.



The LoadBalancer IP can also be found in cluster level

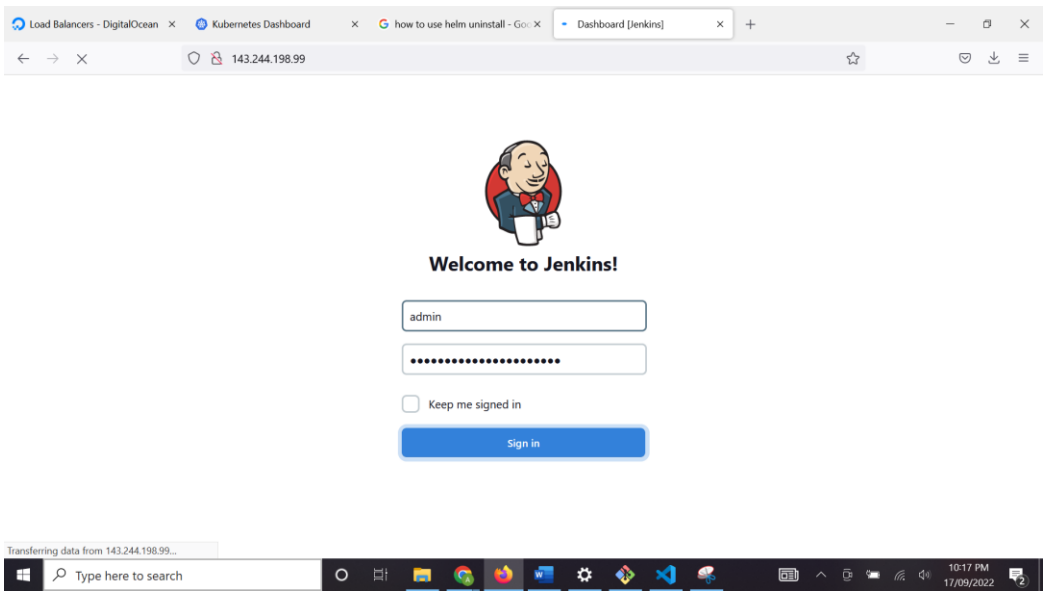
```
User@DESKTOP-2QD05T2 MINGW64 ~/Downloads
$ kubectl get svc
NAME            TYPE           CLUSTER-IP       EXTERNAL-IP      PORT(S)          AGE
jenkins         LoadBalancer   10.245.248.208   143.244.198.99   80:30865/TCP     13m
jenkins-agent   ClusterIP      10.245.106.90    <none>           50000/TCP        13m
kubernetes      ClusterIP      10.245.0.1       <none>           443/TCP          93m

User@DESKTOP-2QD05T2 MINGW64 ~/Downloads
$
```

Now, to login into your Jenkins, you need user name and password. User info we know that it's admin and to get a password we actually need to go inside the POD as shown below.

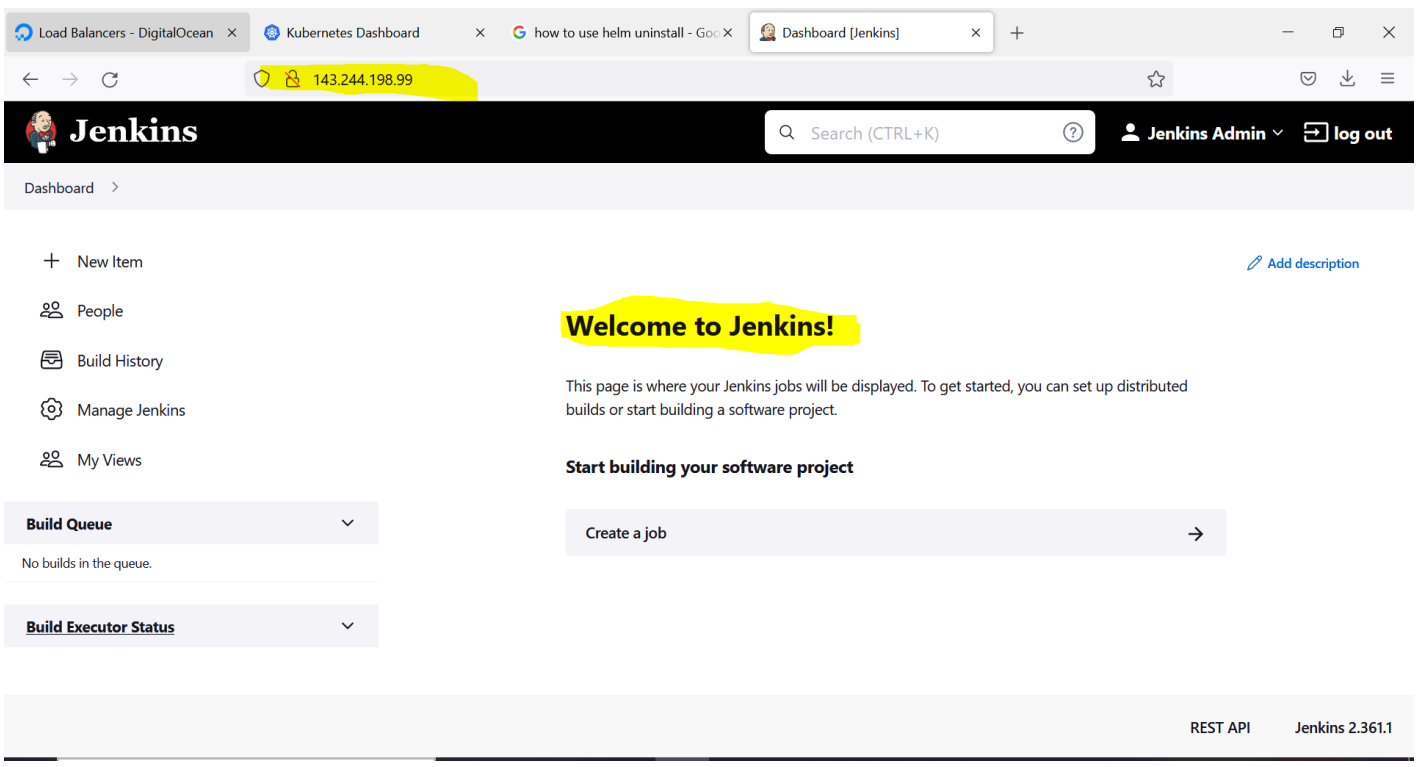
Here, password came along with jenkins@jenkins-0 which if you remove then the password is S9KND1Cj521s07sTni5GqQ

```
$ kubectl get pod
NAME        READY   STATUS    RESTARTS   AGE
jenkins-0   2/2     Running   0          23h

User@DESKTOP-2QD05T2 MINGW64 /c/DEVOPS/JENKINS/jenkins-end-to-end-pipeline1 (mas
ter)
$ kubectl exec -it jenkins-0 -- bash
Defaulted container "jenkins" out of: jenkins, config-reload, init (init)
jenkins@jenkins-0:/$ cat /run/secrets/additional/chart-admin-password
S9KND1Cj521s07sTni5GqQjenkins@jenkins-0:/$
```

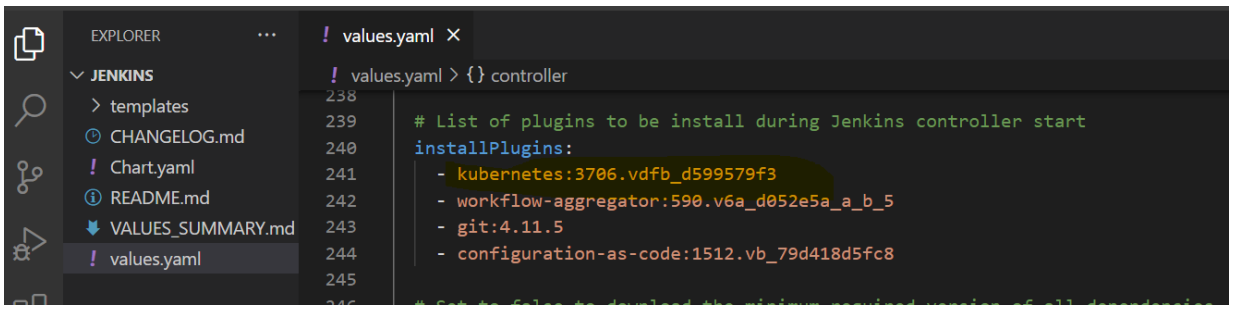
Now using the user name and password you can login to your Jenkins.



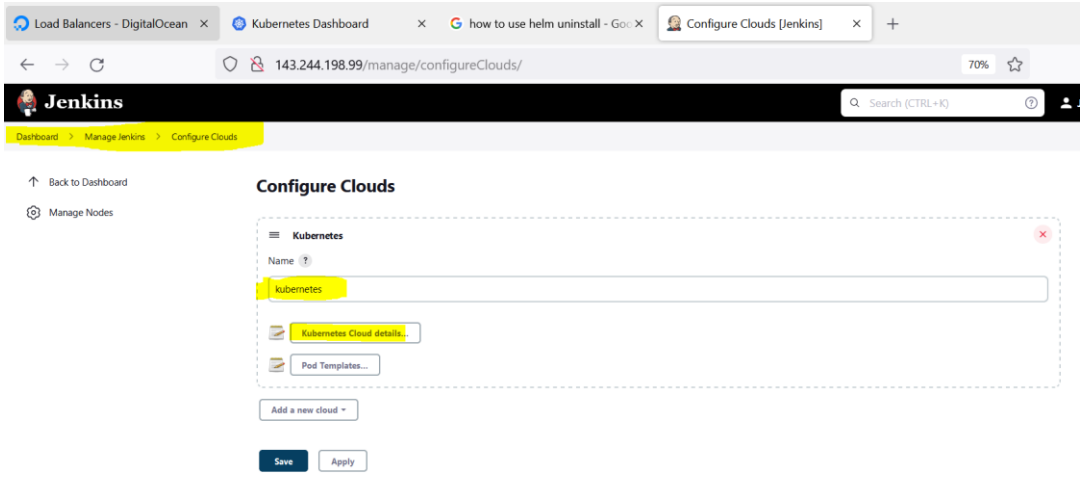
Now, we successfully logged in to Jenkins.



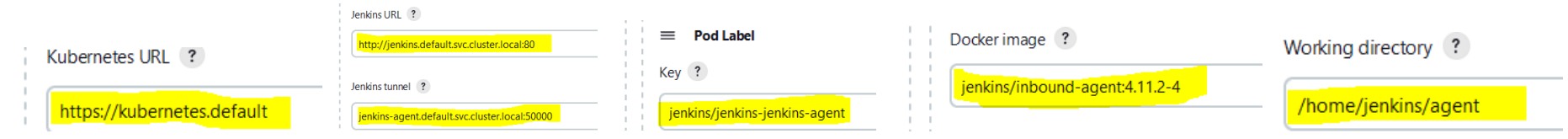
In Jenkins chart value file by default it installed few plugins as shown below.



If you want to use Kubernetes agents as pods then you can find that info in this location. You can get the details by clicking on that detail button.

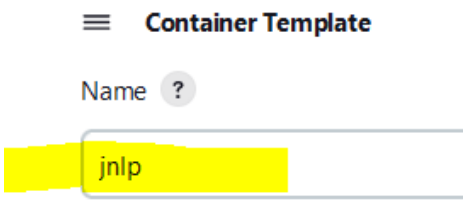


Few details as shown:



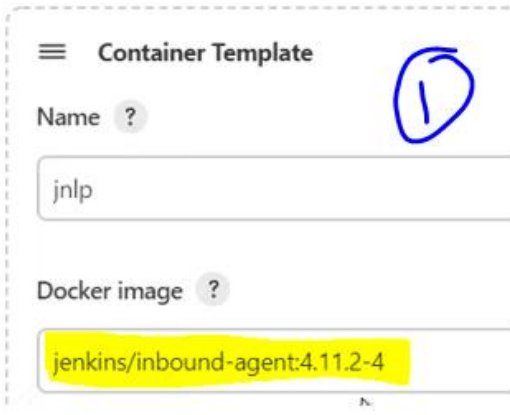
Container name used as jnlp as shown below:

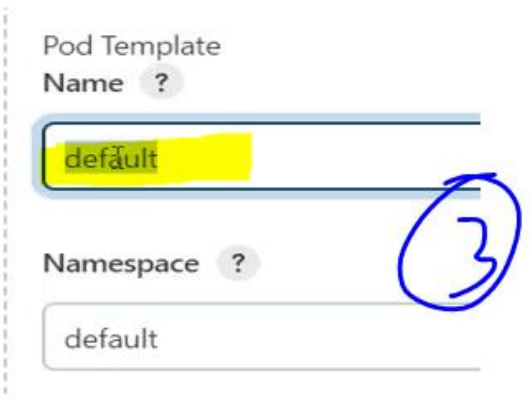
NOTE: At this stage only jnlp container is present but in our future exercise we will be creating few more containers.



NOTE:

- If you are using/having a private repository for your Docker Images then you need to replace that detail here at highlighted place in 1st diagram by removing ***jenkins/inbound-agent:4.11.2-4***. At present we are using this because we are not having a private repository for our images and we are using a public repository (***dockerhub***) for our images.
- If you are using your private repository then you need to use ***inheritFrom 'pod name'*** (Here, pod name can be replaced with pod template name which is default as shown in 3rd diagram but we are not doing it in this example) in your pipeline as shown in 2nd diagram, but at present we are not using it in our pipeline because we are not using a private repository.



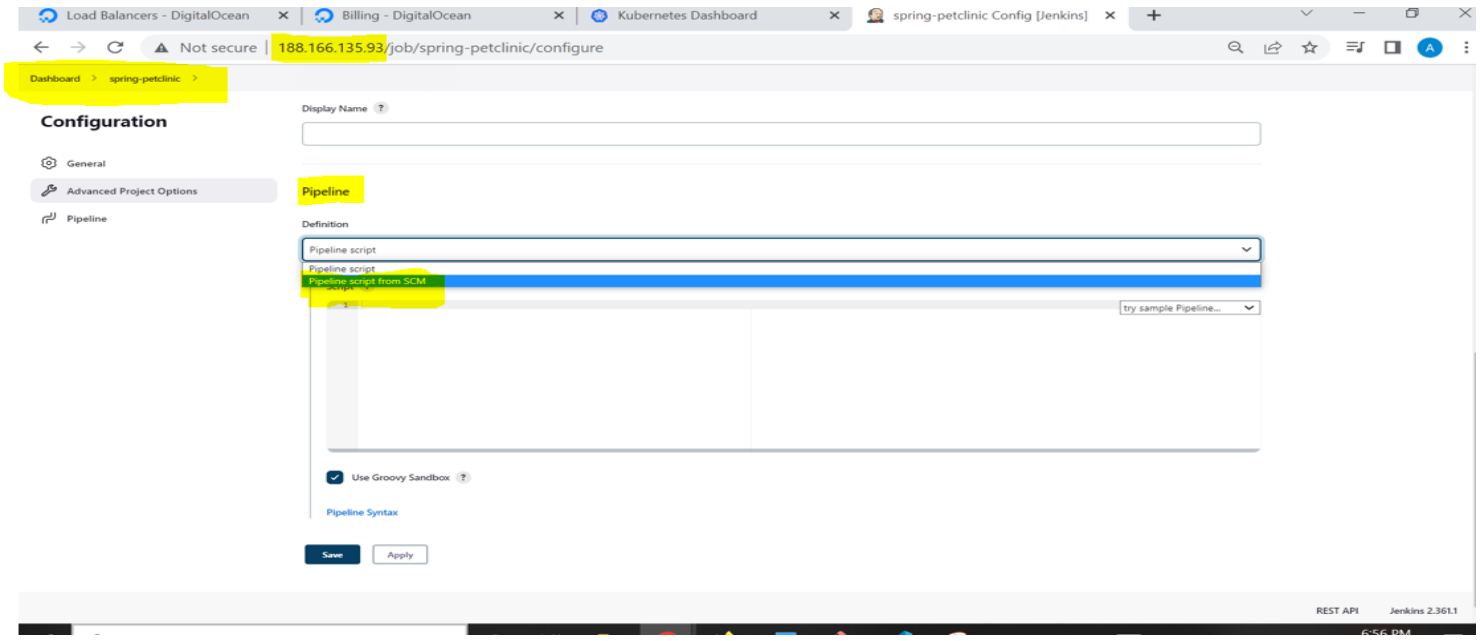


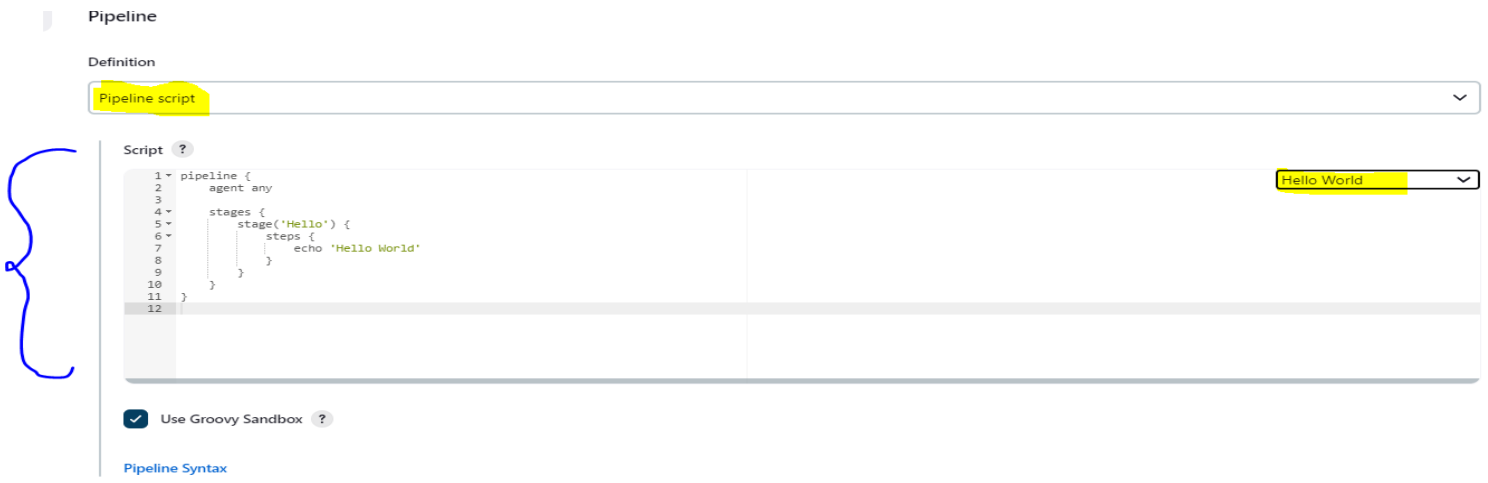
Step 5: Now that we created Jenkins using HELM chart, it's a time to create a sample Jenkins pipeline.

On Jenkins dashboard click on New Item → Enter your project name → Click on Pipeline based project → Click on OK

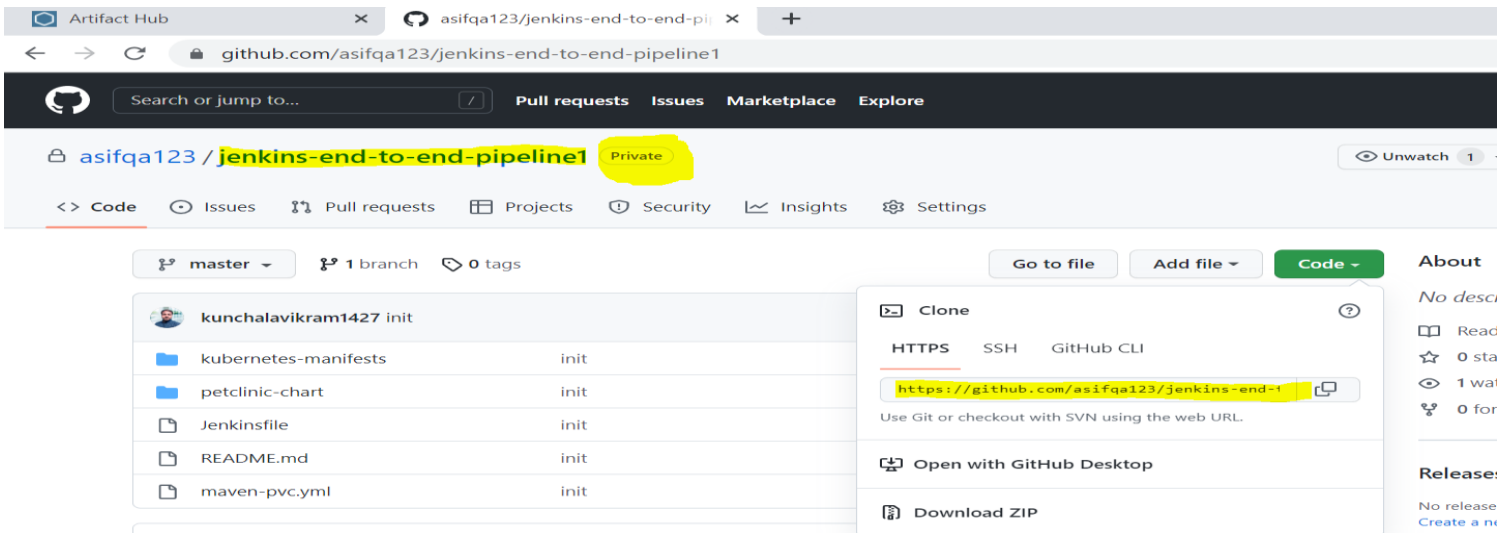
NOTE: ***Here, you can observe that the IP address of Jenkins LoadBalancer is changed that is because I deleted the previous LoadBalancer and created a new Jenkins LoadBalancer next day to save the cost of LoadBalancer*** 😊

If you already have Jenkins file in your project then you can select a option ***Pipeline script from SCM*** but here we are just practicing a sample pipeline so no need to choose this option at this point of time.





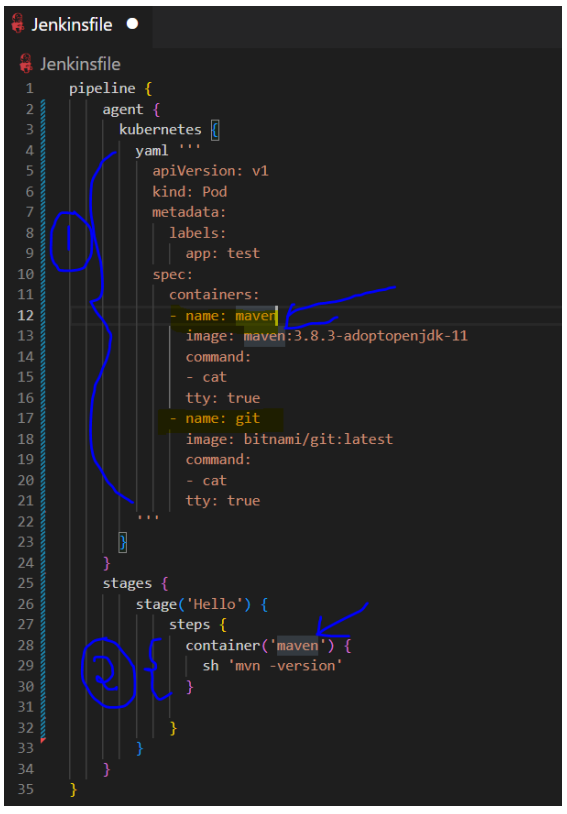
Now, go to the repository and clone it on your laptop before starting anything.

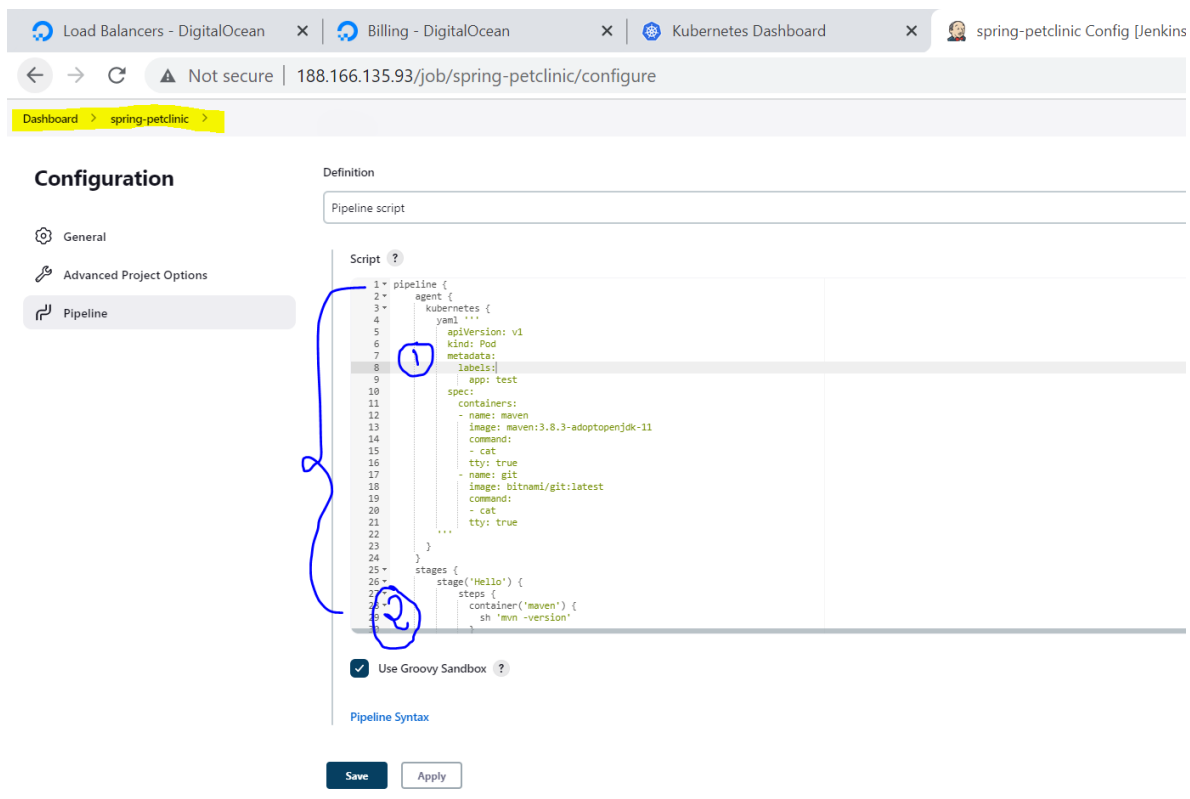


Once you clone, you can open it on Vscode and remove all the contents of that Jenkinsfile we can add the Hello World script from the above screen so that we can practice our steps from fresh.

NOTE: *We copied the Hello World script from the above screen but it needs few modifications as shown below. Here, we actually added 2 parts, in the 1st part we added Kubernetes agent and inside it included containers* ***maven*** *and* ***git*** *for performing the operations in related stages and in 2nd part we are using* ***maven*** *container so that we can execute maven commands and this stage will executed by utilizing* ***maven*** *container image from Kubernetes agents section.*

Now you can go to your Dashboard → your ***spring-petclinic*** project → Configure → and paste your Jenkinsfile code at this part → Save → Click on **Build Now**



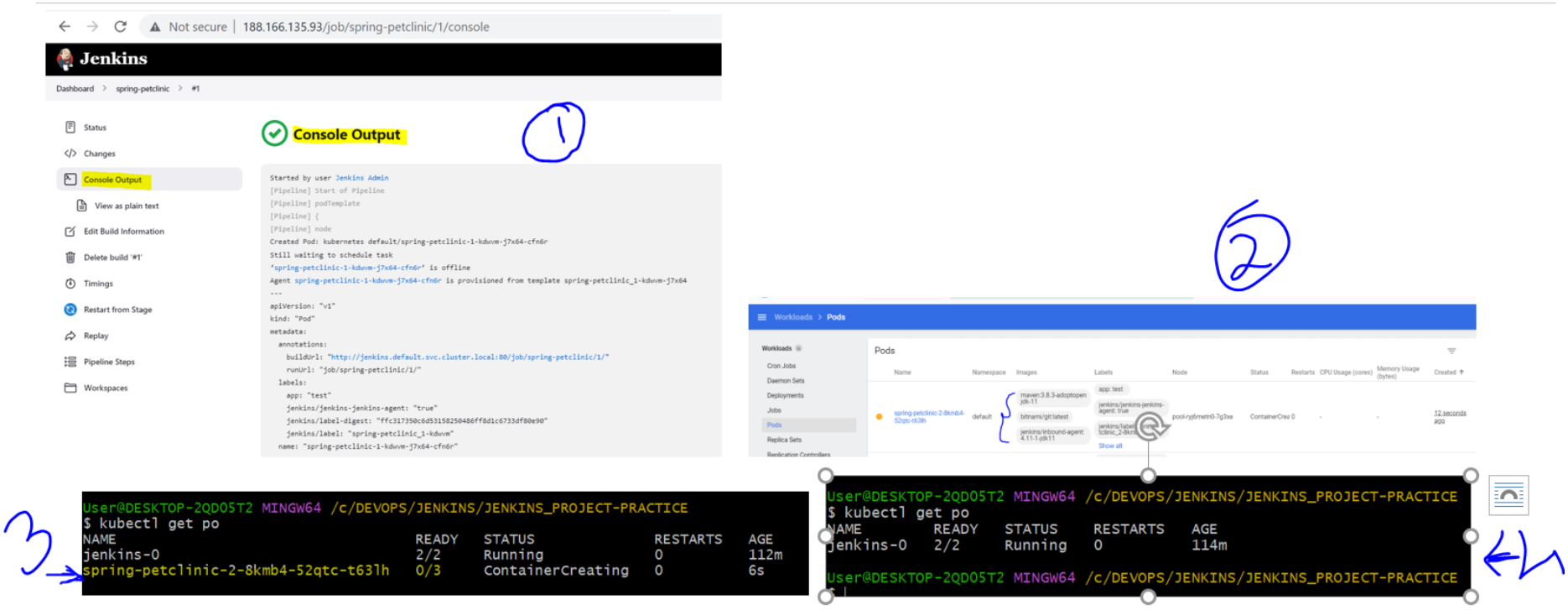


NOTE: *In the above diagrams we are setting command as* ***- cat*** *and* ***tty: true*** *while creating container images in stage 1 that is because it will run cat command, since we are not giving any input to this cat command so this container will keep on waiting for the command for ever which means these containers will be available till my job is finished after that these containers will automatically get deleted which you will see in below slides.*

After building the project we can observe 4 things:

1. 1st diagram is showing, that the Console output of the job is showing as successful.
2. 2nd diagram is showing that it's created 3 images (1st one is maven, 2nd one is git and 3rd one is inbound-agent which is started by the Kubernetes agent, that detail you can find if you go and check the Kubernetes agent details button).
3. 3rd diagram is showing that it's creating those 3 images.
4. 4th diagram is showing that after the execution all the 3 images are deleted as it is on Kubernetes environment, once the work of a POD is completed those pods will get deleted.

NOTE: Here, every stage is running in a separate container then how the cloned source is made available in the other container, for that Jenkins handle it in a different way by creating a emptyDir volume called workspace and it is going to mount that emptyDir volume into all the containers, you can see that details in 5th diagram.

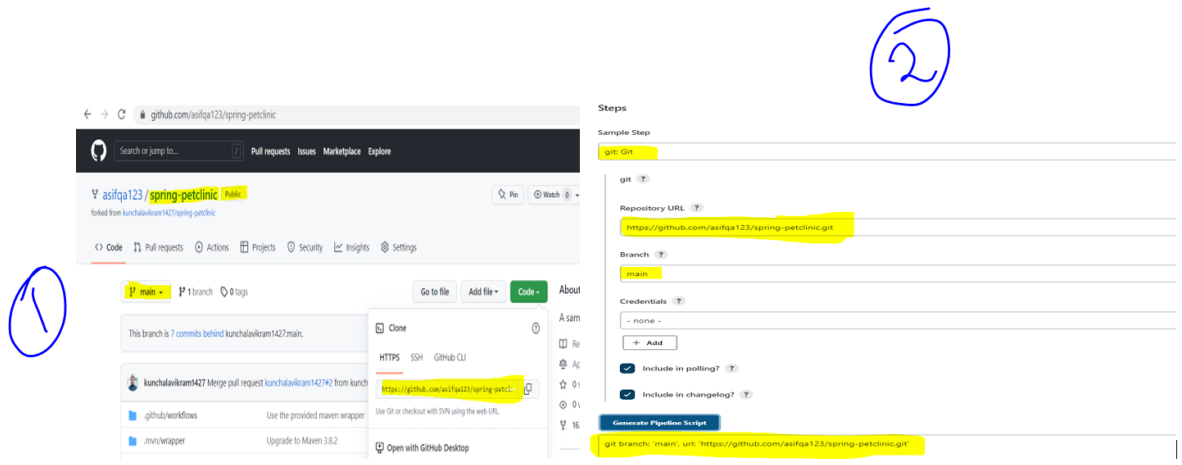


Here, you can observe that the ***emptyDir*** volume is mounted on ***maven*** and ***git*** containers on the same path. Hence, this way the containers can share the data among each other. And on 6th diagram you can see that the maven command is executed as per our pipeline and the output of the maven version is also printed.

Step 6: Now, it's time to clone our repository and building it through maven in our declarative pipeline.

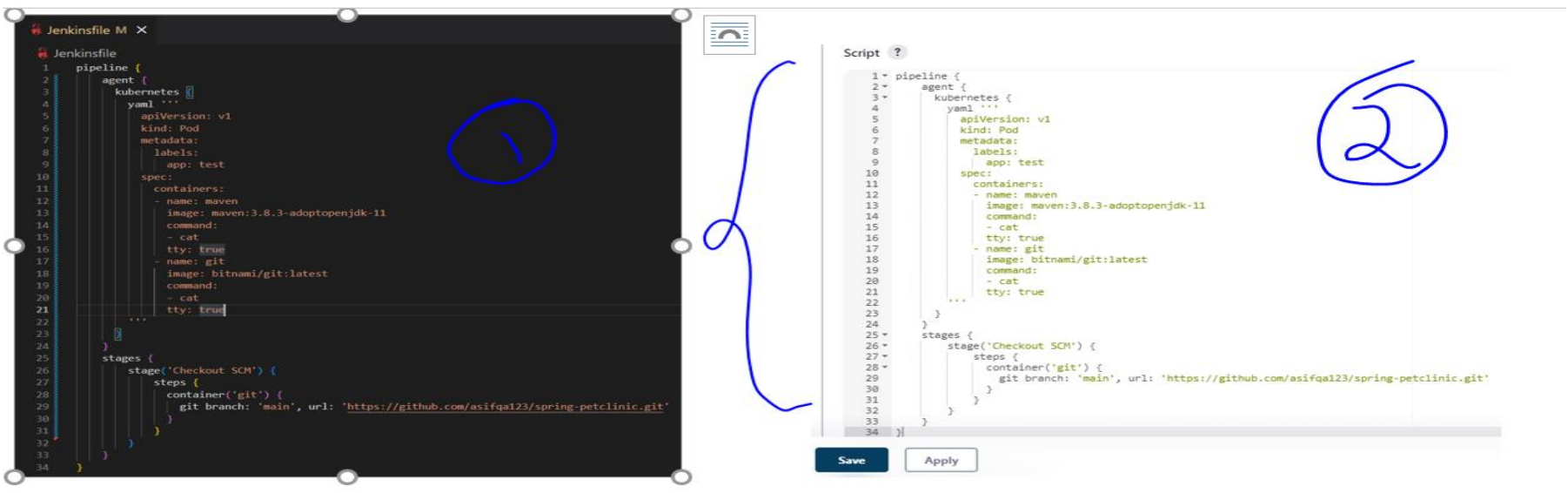
For cloning the git repository, we need to execute git commands for that either we can use the plugin related commands or we can directly use git commands. For now, we can use git plugin commands (This plugin you can find in Kubernetes agents section).

Let's generate the pipeline syntax but while generating the pipeline syntax for the ***spring-petclinic*** project. In 1st diagram we are showing that from where we are cloning and which branch we are cloning. And on 2nd diagram we are showing how to generate the pipeline syntax and generate the output, for this you need to go to your project → click on *Pipeline Syntax*



Now, on our Jenkins file let's modify the stage and name it as ***Checkout SCM*** and run it on ***git*** container as it's a git command.

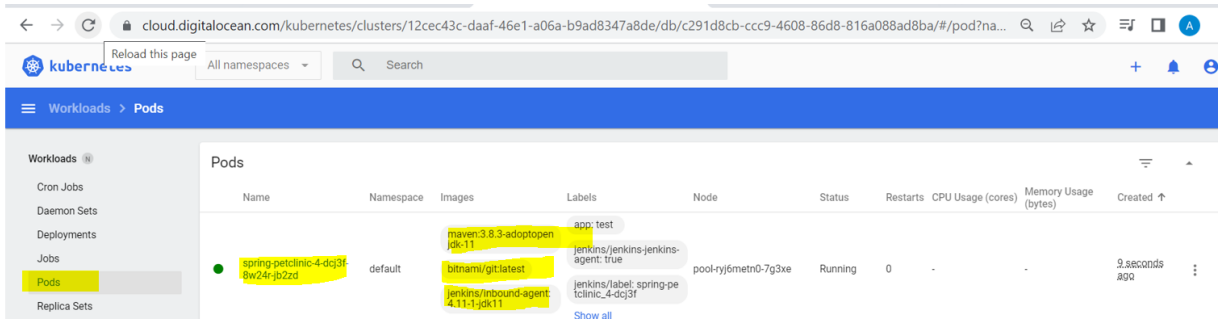
Here, in 1st diagram we modified our Jenkins file in our project ***Jenkins-end-to-end-pipeline1*** (this is our private repository) and the same code we copied on our Jenkins job spring-petclinic project which is shown in 2nd diagram.



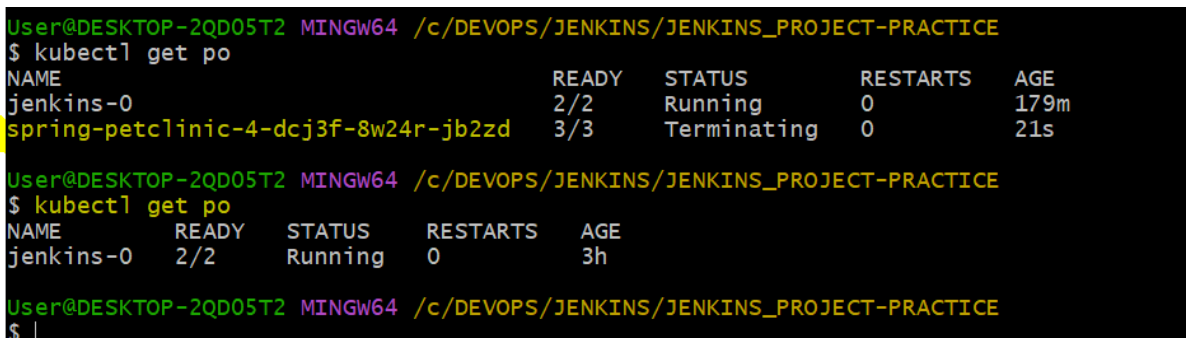
After running the above pipeline by clicking on Build Now button, we can see the following output.

1. In 1st diagram we can see that 3 container images have been created (maven, git, inbound)
2. In 2nd diagram we can see that 3 pods got created and got terminated once the pipeline is completed.
3. In 3rd diagram we can see that the spring-petclinic repository got cloned successfully.

1st diagram



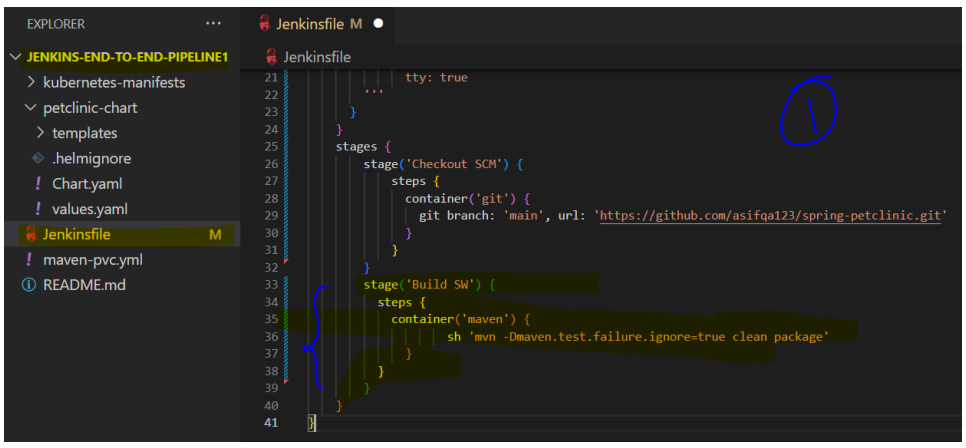
2nd diagram



3rd diagram

Step 7: Now, it's time to create a new stage in our declarative pipeline which is *Build SW* as shown below.

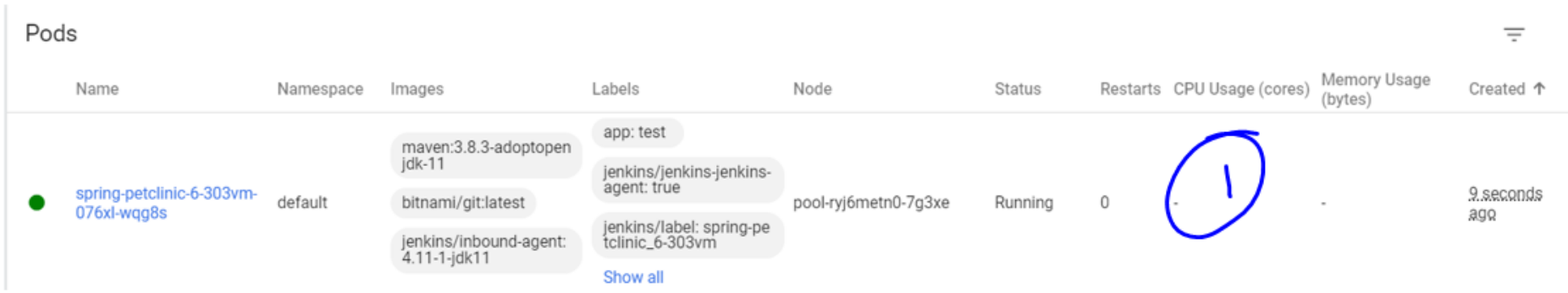
In 1st diagram we modified the code in Jenkinsfile of our ***Jenkins-end-to-end-pipeline1*** and in 2nd diagram we copied the same code and pasted on our Jenkins ***spring-petclinic*** job.

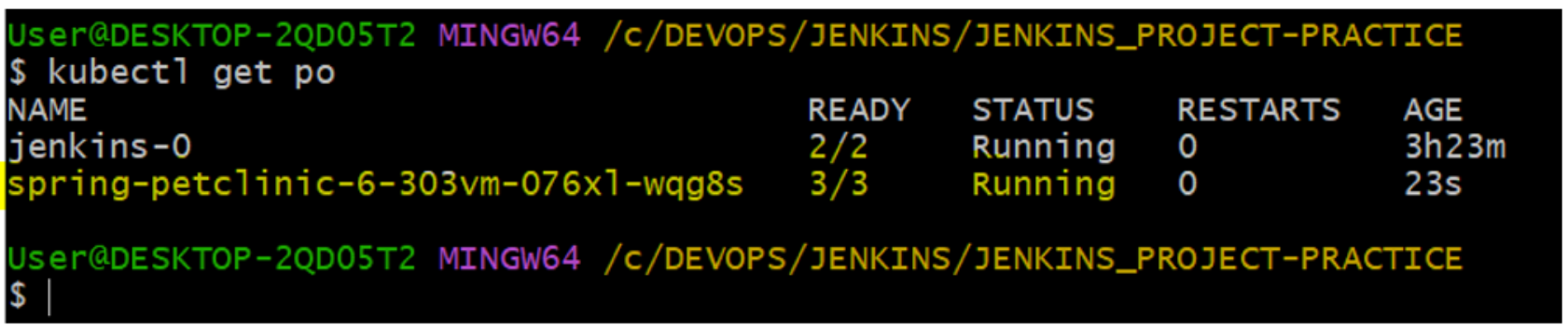


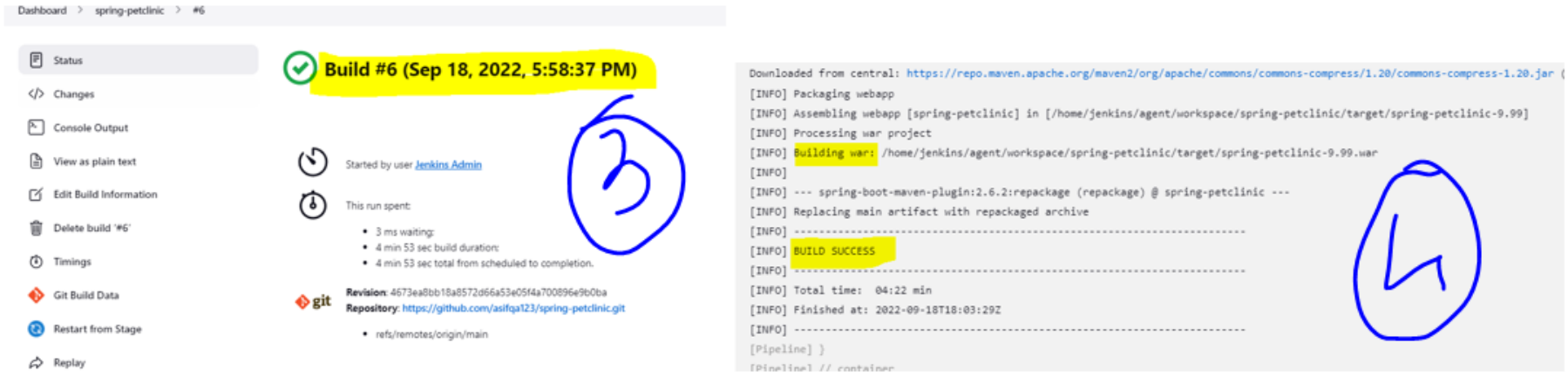
When we run the above job then we can see that those 3 images are created which is shown in 1st diagram and on 2nd diagram we can see that those 3 pods are created and those are in running state.

NOTE: Building this project will take more time (around 10 min) because it's a big project, so have some patience 😊

Now, here you can observe that 3 images are created in 1st diagram, 3 pods are created in 2nd diagram which will be destroyed once the job is done, 3rd diagram shows that the job is successful and the 4th diagram is showing that the repository is cloned in ***Checkout SCM*** stage and finally it got build in second stage which is ***Build SW*** and it's showing that where the WAR file is located.

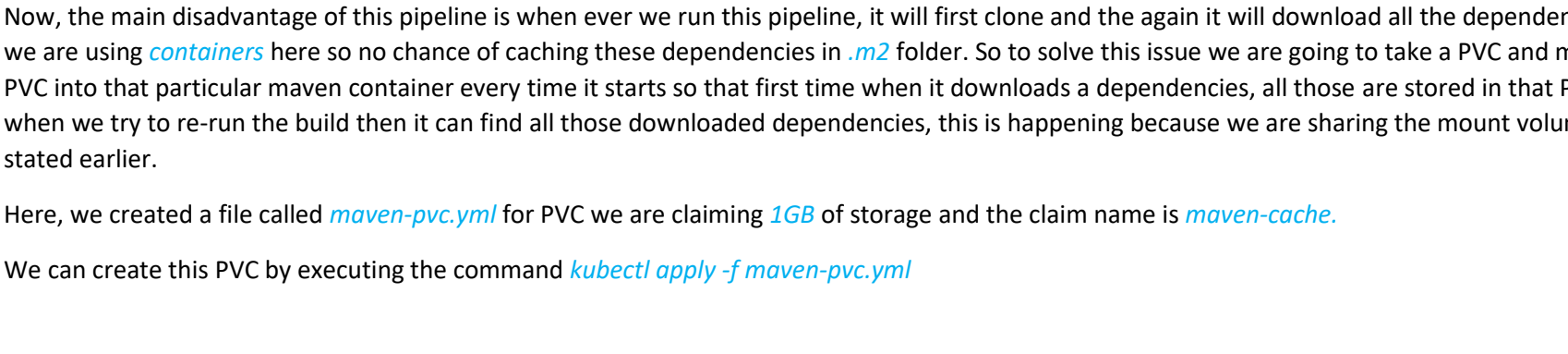






Now, it's time to create a *post build action* for ***Build SW*** stage to display the Junit test results. For this we need to install the Junit plugin in your Jenkins server as shown below.

Manage Jenkins → Manage Plugins → Available → Select junit box → Install without restart



Once, Junit is installed successfully, we can add this step in our post build action as shown below.

Here, we are using ***/target/surefire-reports/*.xml*, which means:

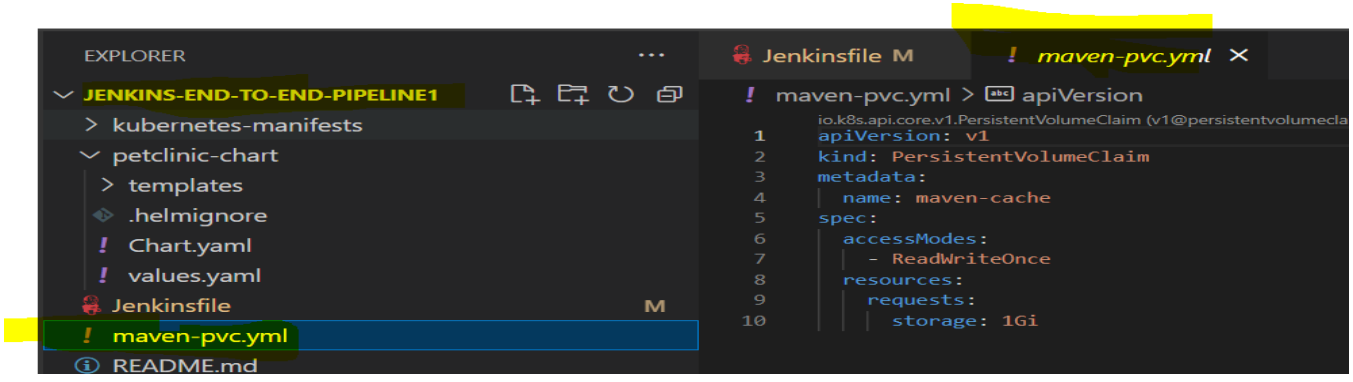
- Initial ** is there so that it will search for a ***target*** folder in any path.
- And at the end we used ****.xml*** which means inside the ***surefire-reports*** folder it will consider all the ***.xml*** files.

```
stages {
    stage('Checkout SCM') {
        steps {
            container('git') {
                git branch: 'main', url: 'https://github.com/asifqa123/spring-petclinic.git'
            }
        }
    }
    stage('Build SW') {
        steps {
            container('maven') {
                sh 'mvn -Dmaven.test.failure.ignore=true clean package'
            }
        }
        post{
            success {
                junit '**/target/surefire-report/*.xml'
            }
        }
    }
}
```

Now, the main disadvantage of this pipeline is when ever we run this pipeline, it will first clone and the again it will download all the dependencies because we are using *containers* here so no chance of caching these dependencies in *.m2* folder. So to solve this issue we are going to take a PVC and mount that PVC into that particular maven container every time it starts so that first time when it downloads a dependencies, all those are stored in that PVC and again when we try to re-run the build then it can find all those downloaded dependencies, this is happening because we are sharing the mount volumes as I stated earlier.

Here, we created a file called *maven-pvc.yml* for PVC we are claiming *1GB* of storage and the claim name is *maven-cache.*

We can create this PVC by executing the command *kubectl apply -f maven-pvc.yml*

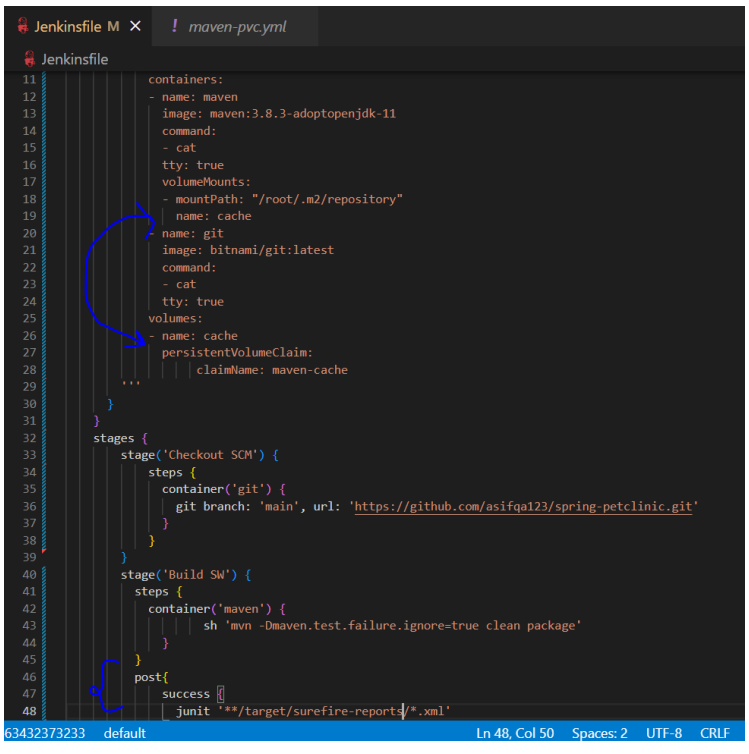


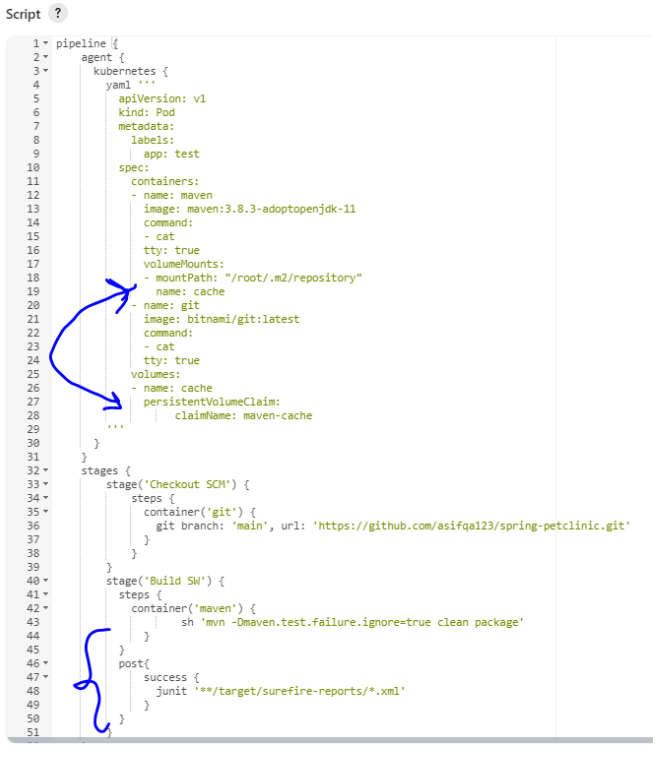
Go to your Vscode terminal and execute the below command to create a PVC of 1GB of size.

```
PS C:\DEVOPS\JENKINS\jenkins-end-to-end-pipeline1> kubectl apply -f maven-pvc.yml
persistentvolumeclaim/maven-cache created
PS C:\DEVOPS\JENKINS\jenkins-end-to-end-pipeline1>
```

Now, I am going to mount this ***PVC*** into the ***maven*** container as shown below. After modifying the Jenkins file, we will paste the same pipeline code in Jenkins job spring-petclinic.

NOTE: Here, we are using the PVC which we created above and using it in ***volumes*** section with a name ***cache*** and mounting it to ***maven*** container image with the same name ***cache***, this is to create a link between ***volumes*** and ***mountPath***. And the ***claimName*** should be matched with the above ***PVC*** which is ***maven-cache***.

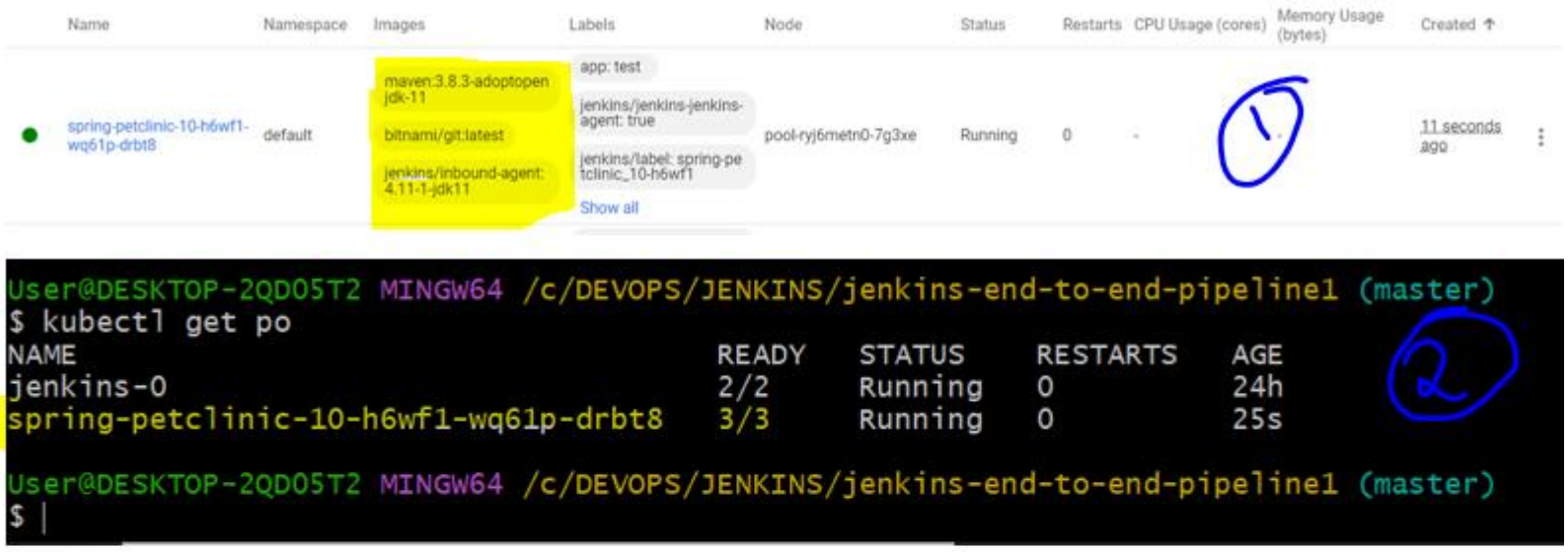




Now, again it will take more time to build the job because this is the first time, we are running the job after creating the PVC. Hence, if you run the job again then the job should use the PVC to get the dependencies and execute the job faster without downloading them again which you can see in 4th and 5th diagram.

NOTE:

- 1st diagram shows that it created 3 images.
- 2nd image shows that it created 3 pods.
- 3rd image displays the test results.
- 4th image is showing the time taken when we first executed the job after including PVC changes.
- 5th image is showing the time taken when we re-run the job for the second time. It proves that the dependencies are pulled from the PVC because of which it took less time to execute.



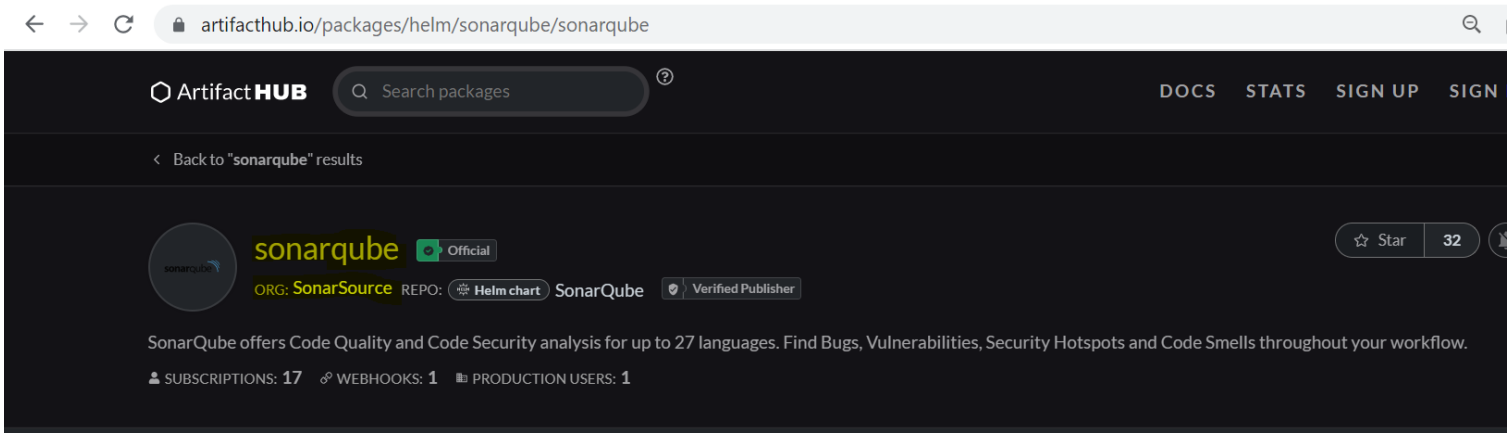
Step 8: Install the SonarQube for performing Static Code Analysis for your project

There are two parts in it one is Sonar Scanner and another is Sonar Server, Sonar Scanner will scan the source code and Sonar Server will perform analysis of those scanned reports.

Sonar Scanner is a local CLI tool, you can use direct binary or as a Maven plugin or you can use it in your pipeline as a container image.

NOTE: You can refer more details by going to the official document which is https://docs.sonarqube.org

For installing SonarQube we can pull the image from artifacthub.io portal, this is a official image from ***SonarSource*** organization.



Now, before installing SonarQube we can first execute these 2 steps (This details we got it from the above chart).

```
User@DESKTOP-2QDO5T2 MINGW64 /c/DEVOPS/JENKINS/jenkins-end-to-end-pipeline1 (master)
$ helm repo add sonarqube https://SonarSource.github.io/helm-chart-sonarqube
"sonarqube" has been added to your repositories

User@DESKTOP-2QDO5T2 MINGW64 /c/DEVOPS/JENKINS/jenkins-end-to-end-pipeline1 (master)
$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "sonarqube" chart repository
...Successfully got an update from the "traefik" chart repository
...Successfully got an update from the "jenkins" chart repository
Update Complete. ⎈Happy Helming!⎈

User@DESKTOP-2QDO5T2 MINGW64 /c/DEVOPS/JENKINS/jenkins-end-to-end-pipeline1 (master)
$
```

And pull the SonarQube chart using the below command so that we can update the required changes to the chart before installing. The syntax and use of untar command is explained while pulling the Jenkins chart, you can refer that explanation.

```
User@DESKTOP-2QDO5T2 MINGW64 /c/DEVOPS/JENKINS
$ helm pull sonarqube/sonarqube --untar

User@DESKTOP-2QDO5T2 MINGW64 /c/DEVOPS/JENKINS
$ code sonarqube/

User@DESKTOP-2QDO5T2 MINGW64 /c/DEVOPS/JENKINS
$
```

Once you open the code in Vscode, click on ***values.yaml*** file and perform the below changes.

NOTE:

- In 1st diagram we changed from ***ClusterIP*** to ***LoadBalancer***.
- In 2nd diagram we changed the ***memory*** from ***4Gi*** to ***2Gi*** and the ***persistence*** value we changed from ***false*** to ***true*** as we need persistent data.
- In 3rd diagram we changed ***size*** from ***20Gi*** to ***1Gi*** because for our testing we are ok with this size.

```
51  service:
52    type: LoadBalancer
53    externalPort: 9000
54    internalPort: 9000
55    labels:
56    annotations: {}
57    # May be used in example fo
```

```
316   resources:
317     limits:
318       cpu: 800m
319       memory: 2Gi
320     requests:
321       cpu: 400m
322       memory: 2Gi
323
324   persistence:
325     enabled: true
```

```
417     requests:
418       cpu: 100m
419       memory: 200Mi
420   persistence:
421     enabled: true
422     accessMode: ReadWriteOnce
423     size: 1Gi
424     storageClass:
```

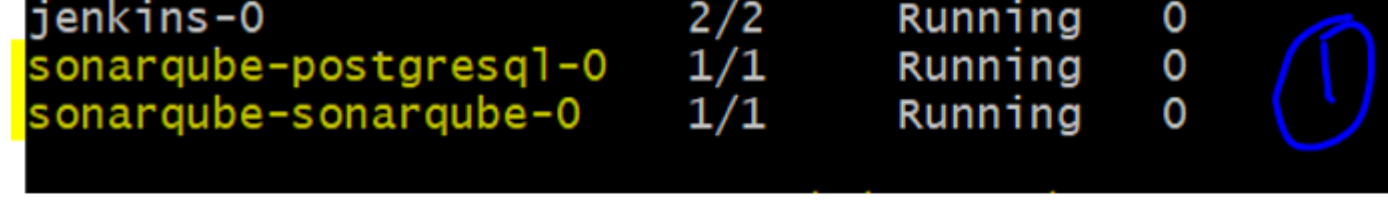
Now, after modification we can save the changes and run the below command to install SonarQube chart.

```
User@DESKTOP-2QDO5T2 MINGW64 /c/DEVOPS/JENKINS
$ helm install sonarqube sonarqube/
NAME: sonarqube
LAST DEPLOYED: Mon Sep 19 20:03:15 2022
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
1. Get the application URL by running these commands:
     NOTE: It may take a few minutes for the LoadBalancer IP to be available.
           You can watch the status of by running 'kubectl get svc -w sonarqube-sonarqube'
  export SERVICE_IP=$(kubectl get svc --namespace default sonarqube-sonarqube -o jsonpath='{.status.loadBalancer.ingress[0].ip}')
  echo http://$SERVICE_IP:9000

User@DESKTOP-2QDO5T2 MINGW64 /c/DEVOPS/JENKINS
$
```

After installing SonarQube we can see the below changes

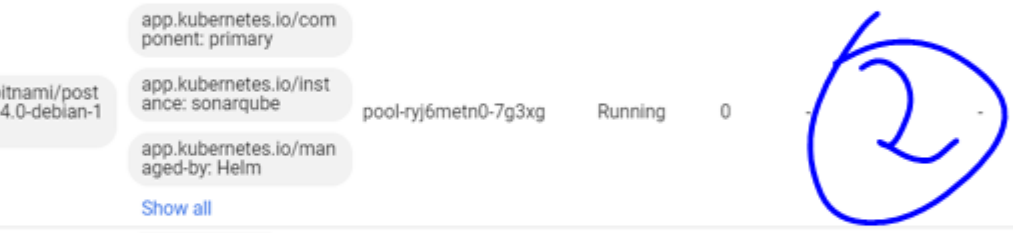
- 1st diagram shows that two more ***PODs*** are created along with Jenkins, those are ***sonarqube*** and ***sonarqube postgresql*** for database.
- 2nd diagram shows that two ***PODs*** are created one for ***sonarqube*** and another for ***sonar postgresql***.
- 3rd diagram shows that now total two ***LoadBalancers*** are created, one for ***Jenkins*** and one for ***SonarQube***

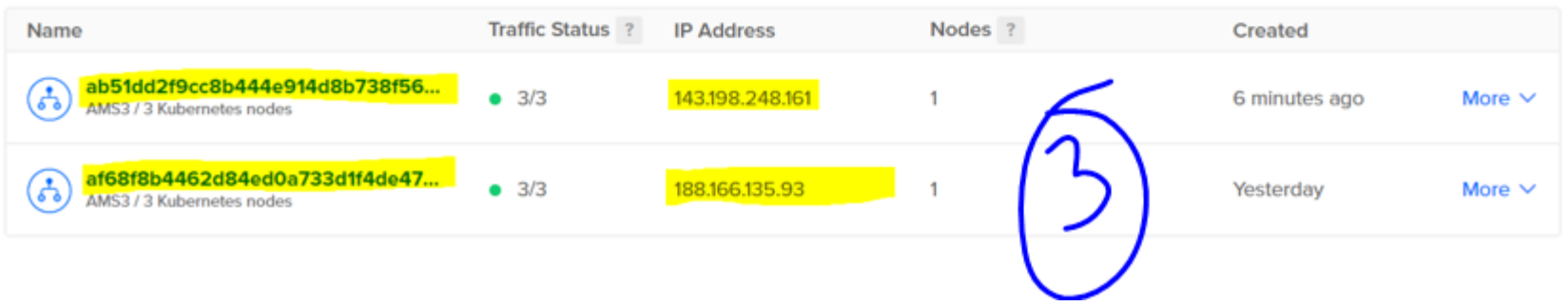
```
User@DESKTOP-2QD05T2 MINGW64 /c/DEVOPS/JENKINS
$ kubectl get po
NAME                        READY   STATUS    RESTARTS   AGE
jenkins-0                   2/2     Running   0          25h
sonarqube-postgresql-0      1/1     Running   0          12m
sonarqube-sonarqube-0       1/1     Running   0          12m
```



Pods

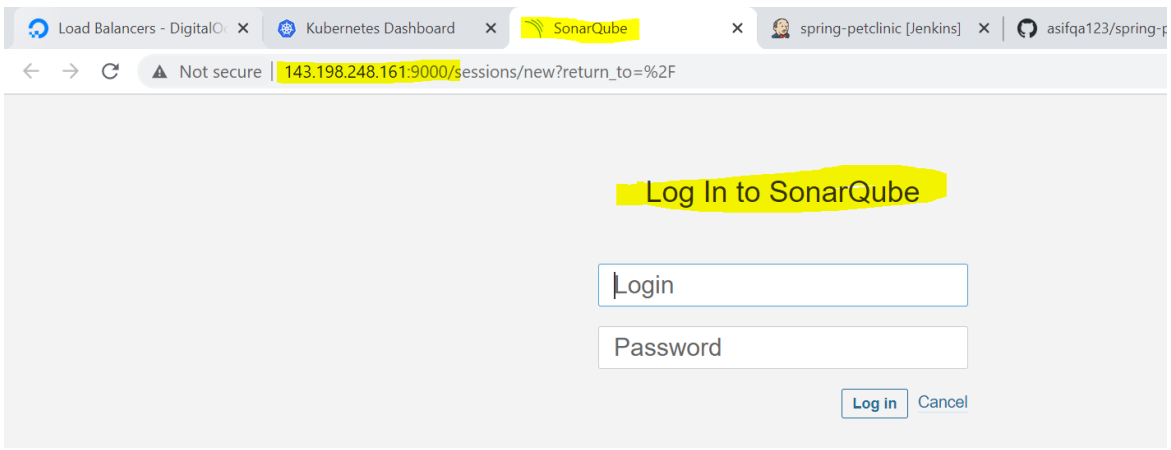
	Name	Namespace	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Us (bytes)
●	sonarqube-postgresql-0	default	docker.io/bitnami/postgresql:11.14.0-debian-10-r22	app.kubernetes.io/component: primary app.kubernetes.io/instance: sonarqube app.kubernetes.io/managed-by: Helm Show all	pool-ryj6metn0-7g3xg	Running	0	-	-
●	sonarqube-sonarqube-0	default	sonarqube:9.6.1-community	app: sonarqube controller-revision-hash: sonarqube-sonarqube-6685646659 release: sonarqube	pool-ryj6metn0-7g3xe	Running	0	-	-

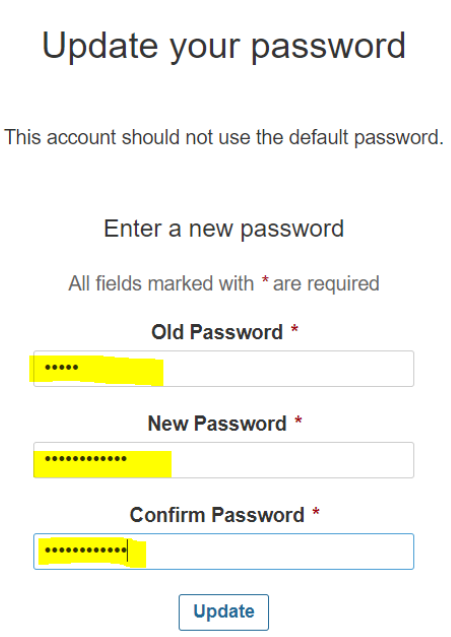




Now, you can cross check by opening the sonarqube using it's LoadBalancer and it's port which is 9000 as shown below.

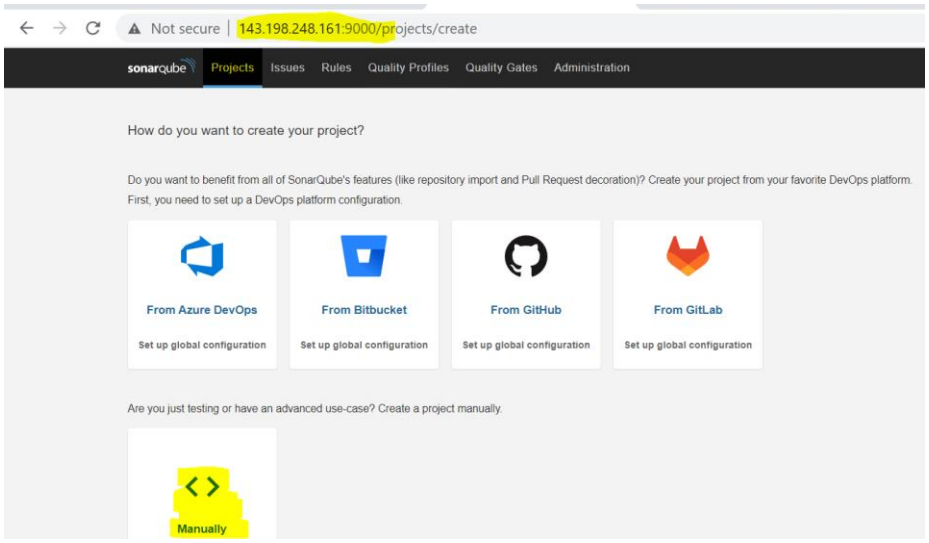
NOTE: The default ***user name*** and ***password*** of ***SonarQube*** is ***admin*** and ***admin***. And once you login it will ask you to update the password, I gave Sonarqube123 as a new password.

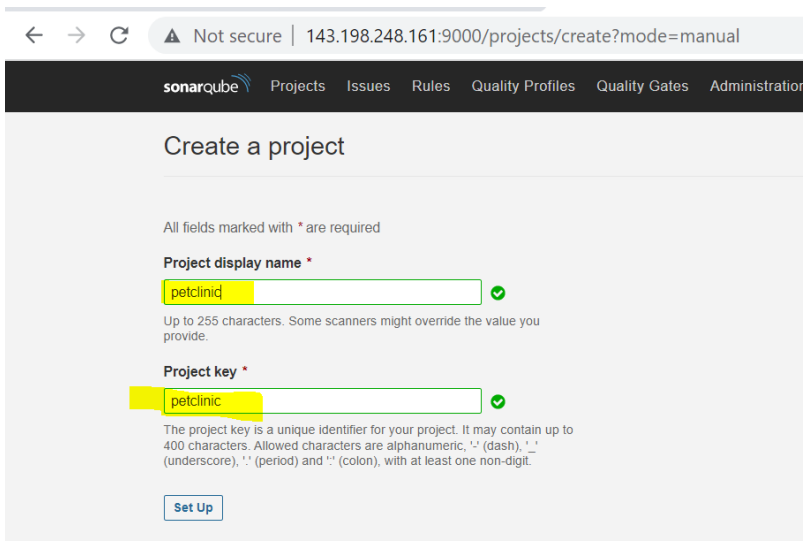




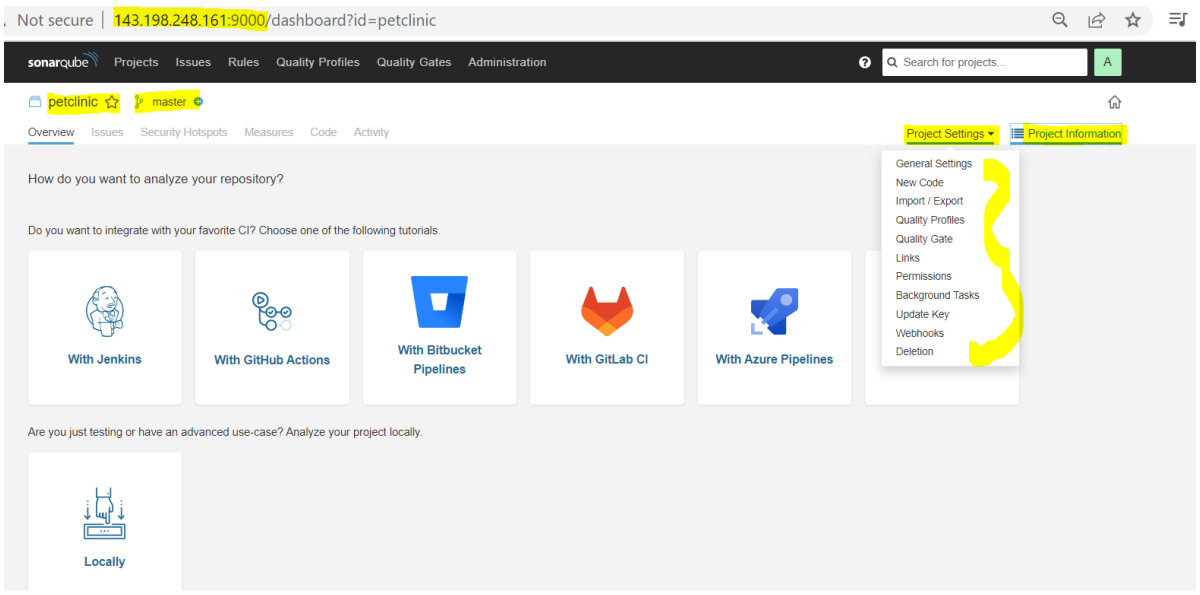
After updating the password, you can see this welcome screen where you can perform many configurations according to your project but we are creating the project using Manually option.

NOTE: Here, we are giving the petclinic as the name for our project.

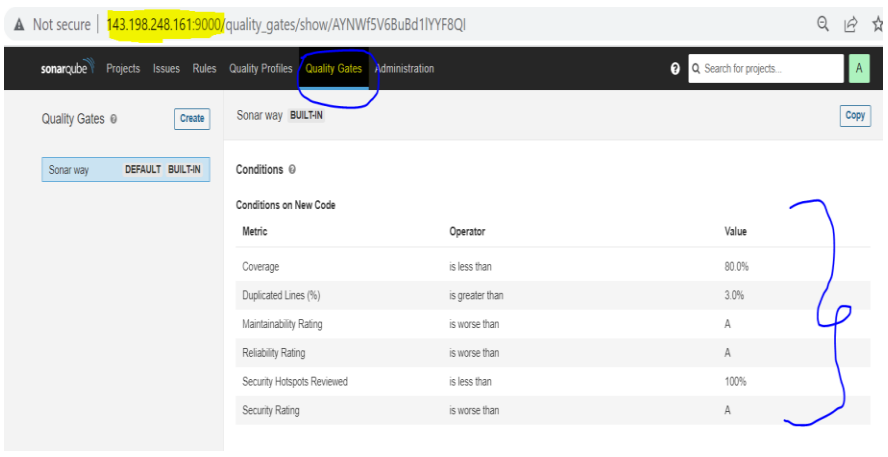


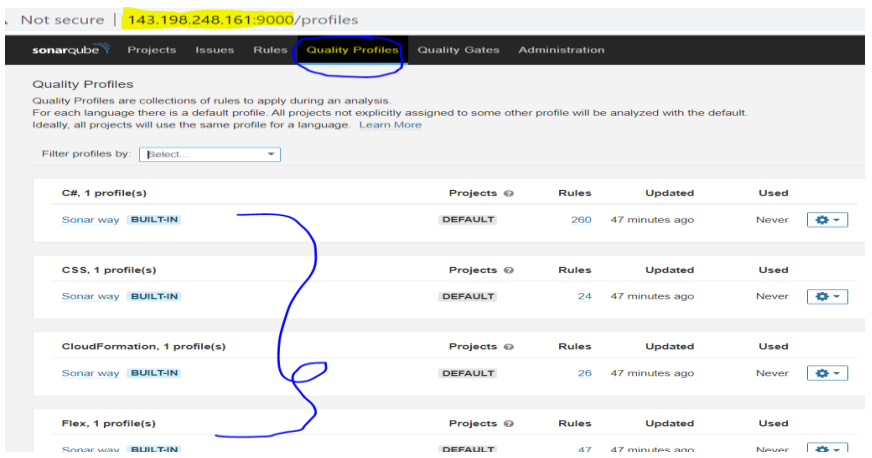


After clicking on Set Up button, you will be displayed with the below screen where you have various options. You can click on Project Information and see the info related to your project and by clicking on Project Settings, you will be able to see the settings configured for your project.

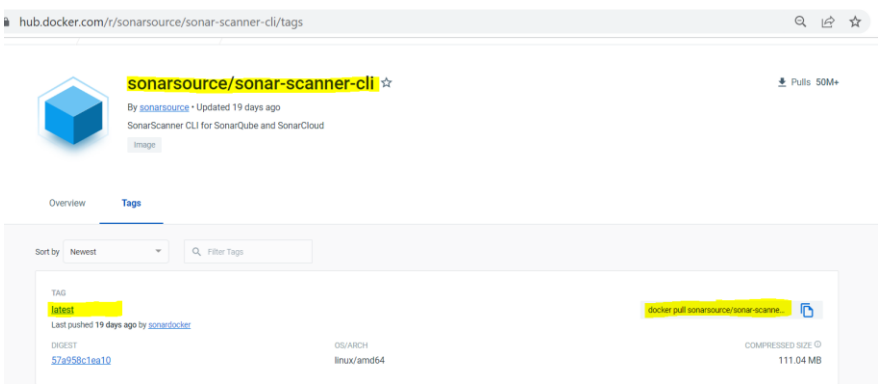


Here, we are showing some of the default ***Quality Gates*** and ***Quality Profiles*** which are associated to our project. If you want you can modify these settings according to your requirement.





Now that you created ***SonarQube*** through HELM chart, created a ***petclinic*** project and using default configuration for this project, it's time to write the pipeline code for SonarQube stage which is ***Sonar Scan***. So for that we need to create one more container in our Kubernetes ***agent***, you can get the image of SonarQube from ***dockerhub*** (You can search for *sonarsource/sonar-scanner-cli*)



Now, let's add the code in our pipeline as shown here, ***1st diagram*** shows that we have included ***sonarscanner*** container in our Kubernetes ***agent*** part and in ***2nd diagram*** we are running ***Sonar scan*** stage just to check the location of the ***sonar scanner*** for now.

And we are using the statement `when { expression { false} }` for ***Checkout SCM*** and ***Build SW*** stages because we don't want to run them now again just to get the location of ***sonar scanner*** from the 3rd stage ***Sonar scan*** by executing the `sh 'which sonar-scanner'` command.

```
      app: test
  spec:
    containers:
    - name: maven
      image: maven:3.8.3-adoptopenjdk-11
      command:
      - cat
      tty: true
      volumeMounts:
      - mountPath: "/root/.m2/repository"
        name: cache
    - name: git
      image: bitnami/git:latest
      command:
      - cat
      tty: true
    - name: sonarcli
      image: sonarsource/sonar-scanner-cli:latest
      command:
      - cat
      tty: true
    volumes:
    - name: cache
      persistentVolumeClaim:
        claimName: maven-cache
```

```
stages {
    stage('Checkout SCM') {
      when { expression { false} }
      steps {
        container('git') {
          git branch: 'main', url: 'https://github.com/asifqa123/spring-petclinic.git'
        }
      }
    }
    stage('Build SW') {
      when { expression { false} }
      steps {
        container('maven') {
          sh 'mvn -Dmaven.test.failure.ignore=true clean package'
        }
      }
      post{
        success {
          junit '**/target/surefire-reports/*.xml'
        }
      }
    }
    stage('Sonar scan') {
      when { expression { true } }
      steps {
        container('sonarcli') {
          sh 'which sonar-scanner'
        }
      }
```

After modifying the code in Jenkinsfile in Jenkins-end-to-end-project1, let's copy this code to our Jenkinsfile in the pipeline project spring-petclinic

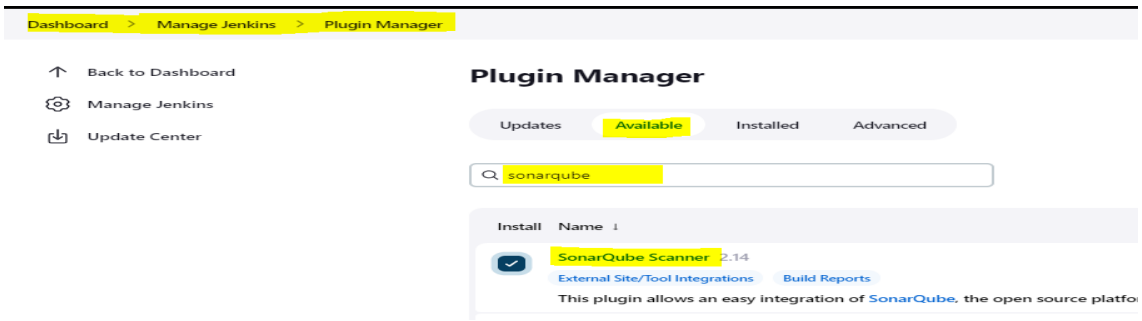
Script ?

```
pipeline {
    agent {
        kubernetes {
      yaml '''
        apiVersion: v1
        kind: Pod
        metadata:
          labels:
            app: test
        spec:
          containers:
          - name: maven
            image: maven:3.8.3-adoptopenjdk-11
            command:
            - cat
            tty: true
            volumeMounts:
            - mountPath: "/root/.m2/repository"
              name: cache
          - name: git
            image: bitnami/git:latest
            command:
            - cat
            tty: true
          - name: sonarcli
            image: sonarsource/sonar-scanner-cli:latest
            command:
            - cat
            tty: true
          volumes:
          - name: cache
            persistentVolumeClaim:
                claimName: maven-cache
      '''
    }
  }
  stages {
    stage('Checkout SCM') {
      when { expression { false} }
      steps {
        container('git') {
          git branch: 'main', url: 'https://github.com/asifqa123/spring-petclinic.git'
        }
      }
    }
    stage('Build SW') {
      when { expression { false} }
      steps {
        container('maven') {
          sh 'mvn -Dmaven.test.failure.ignore=true clean package'
        }
      }
      post{
        success {
          junit '**/target/surefire-reports/*.xml'
        }
      }
    }
    stage('Sonar scan') {
      when { expression { true } }
      steps {
        container('sonarcli') {
          sh 'which sonar-scanner'
        }
      }
    }
}
```

After executing the above job, we can see that we got the location of the sonar scanner.

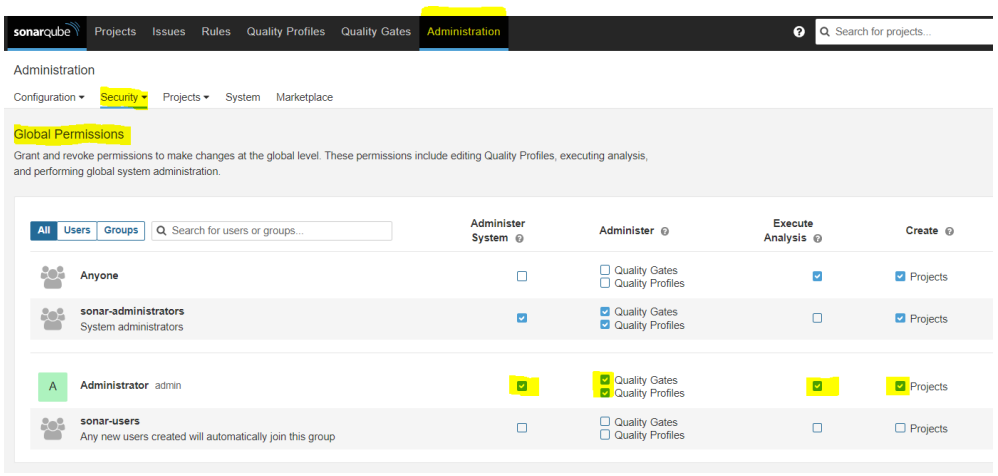
```
[Pipeline] sh
+ which sonar-scanner
/opt/sonar-scanner/bin/sonar-scanner
[Pipeline] }
```

Now, we can go to Manage Jenkins → Manage Plugins → Available → Search for sonarqube → install this plugin without restart, we are installing this plugin because it gives certain configuration which are mandatory.



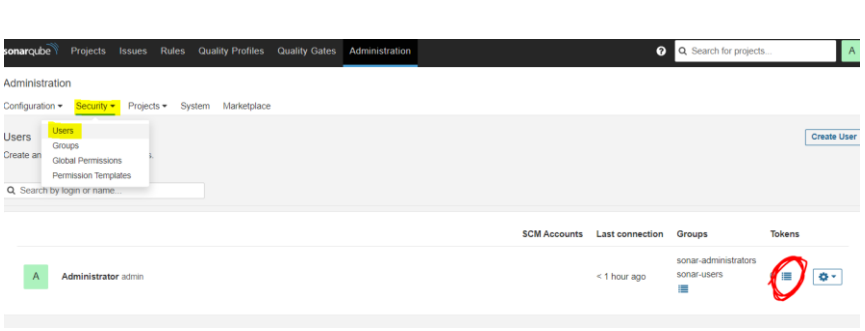
Now, we have to create token and set Global Permissions so that there can be a communication between SonarQube and Jenkins.

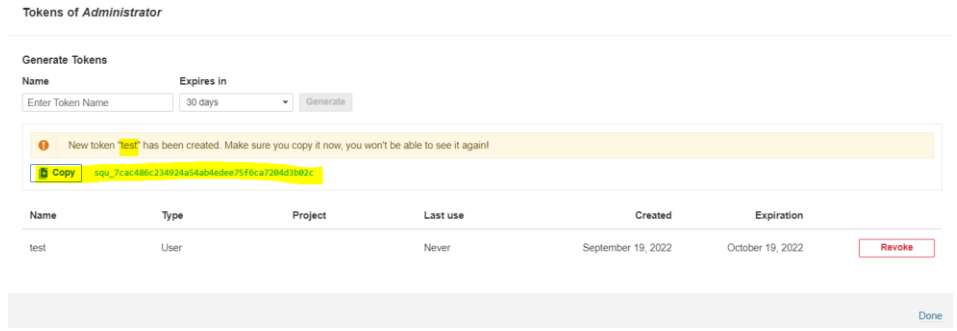
For that first check all the boxes which are highlighted for Administrator



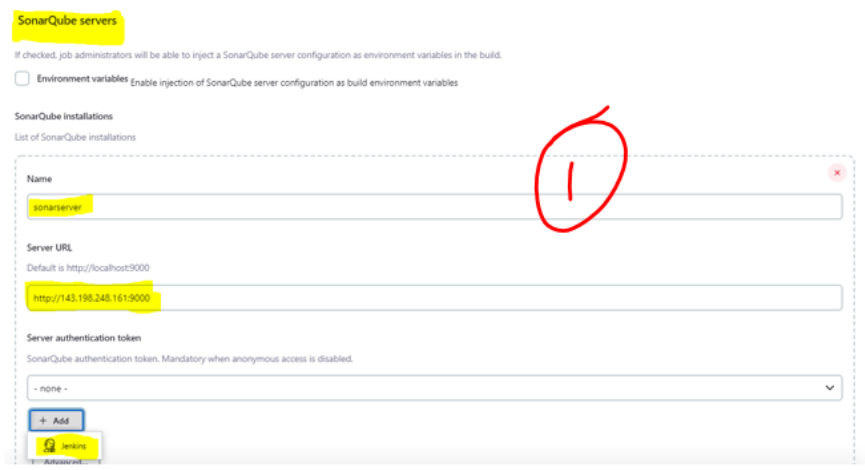
Then, generate the token by clicking on Security → Users → and click on the button which is highlighted in red colour. And in the next screen you can give Name as test and click on Generate button the you will get your token as highlighted.

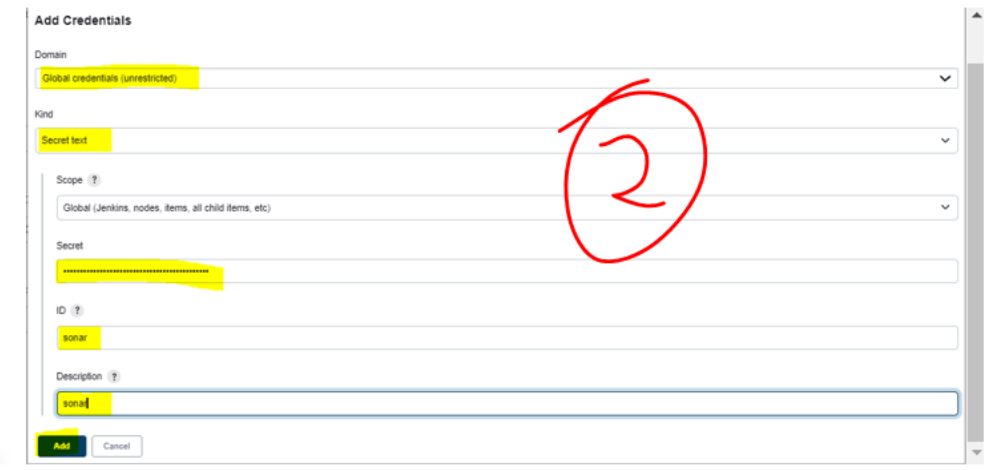
NOTE: The token is ***squ_7cac486c234924a54ab4edee75f6ca7204d3b02c*** but once you close it then you won't be able to access it again.





After generating the sonarqube token now go to Manage Jenkins → Configure System → Now you can find this setting related to SonarQube server as shown in 1st diagram fill the details and add click on Add button to create credentials. Now 2nd diagram will open, here fill the details of token and click on Add button, after adding the credential choose it.

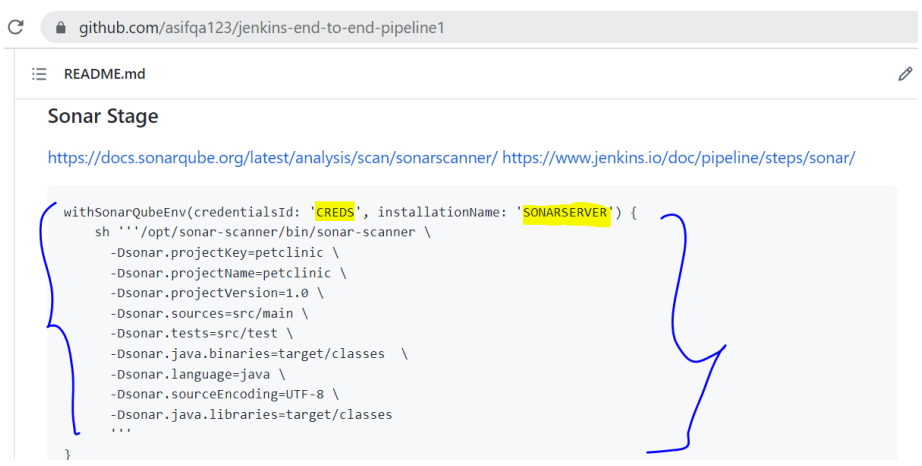




Now, you need to add the below sonar configuration details in your pipeline spring-petclinic by removing the line `sh 'which sonar-scanner'` and adding these below lines as shown below. The details which we mentioned here are as follows:

projectKey and ***projectName*** is petclinic in our ***petclinic*** project in ***SonarQube*** and it is going to scan the ***sources*** in src/main and the ***tests*** are in src/test folder, ***binaries*** are inside target/classes folder and the language in which this ***spring-petclinic*** is written in ***Java*** etc..

NOTE: The highlighted parts need to be changed.



NOTE: Here we are making the when condition true for all the stages and after removing the earlier shell line from Sonar scan stage and added the above lines which is already explained above.

```groovy
pipeline {
    agent {
        kubernetes {
            yaml '''
              apiVersion: v1
              kind: Pod
              metadata:
                labels:
                  app: test
              spec:
                containers:
                - name: maven
                  image: maven:3.8.3-adoptopenjdk-11
                  command:
                  - cat
                  tty: true
                  volumeMounts:
                  - mountPath: "/root/.m2/repository"
                    name: cache
                - name: git
                  image: bitnami/git:latest
                  command:
                  - cat
                  tty: true
                - name: sonarcli
                  image: sonarsource/sonar-scanner-cli:latest
                  command:
                  - cat
                  tty: true
                volumes:
                - name: cache
                  persistentVolumeClaim:
                    claimName: maven-cache
            '''
        }
```

```groovy
    stages {
        stage('Checkout SCM') {
            when { expression { true } }
            steps {
                container('git') {
                    git branch: 'main', url: 'https://github.com/asifqa123/spring-petclinic.git'
                }
            }
        }
        stage('Build SW') {
            when { expression { true } }
            steps {
                container('maven') {
                    sh 'mvn -Dmaven.test.failure.ignore=true clean package'
                }
            }
            post{
                success {
                    junit '**/target/surefire-reports/*.xml'
                }
            }
        }
        stage('Sonar scan') {
            when { expression { true } }
            steps {
                container('sonarcli') {
                    withSonarQubeEnv(credentialsId: 'sonar', installationName: 'sonarserver') {
                        sh '''/opt/sonar-scanner/bin/sonar-scanner \
                            -Dsonar.projectKey=petclinic \
                            -Dsonar.projectName=petclinic \
                            -Dsonar.projectVersion=1.0 \
                            -Dsonar.sources=src/main \
                            -Dsonar.tests=src/test \
                            -Dsonar.java.binaries=target/classes  \
                            -Dsonar.language=java \
                            -Dsonar.sourceEncoding=UTF-8 \
                            -Dsonar.java.libraries=target/classes
                            '''
                    }
                }
            }
        }
    }
}
```

Copy and paste the same code in your Jenkins pipeline job spring-petclinic and build the job to see the output.

Script ?

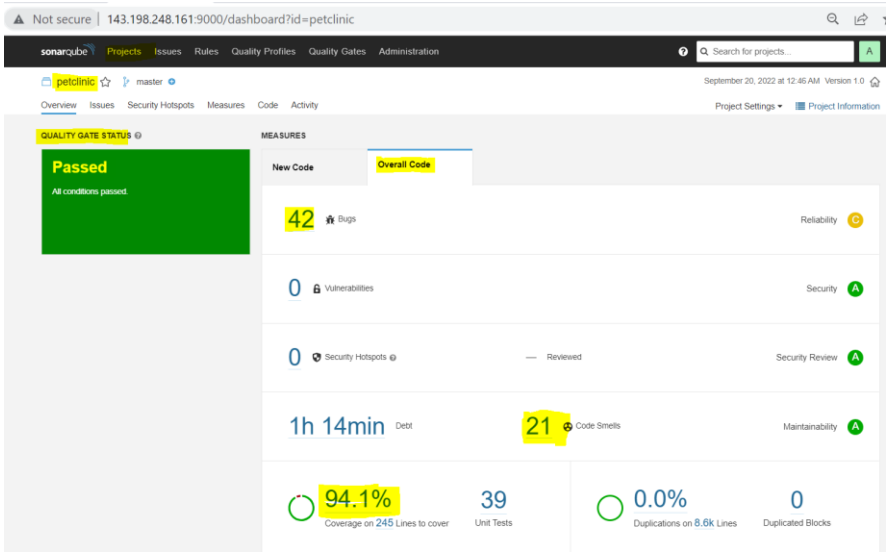
```groovy
stages {
    stage('Checkout SCM') {
        when { expression { true } }
        steps {
            container('git') {
                git branch: 'main', url: 'https://github.com/asifqa123/spring-petclinic.git'
            }
        }
    }
    stage('Build SW') {
        when { expression { true } }
        steps {
            container('maven') {
                    sh 'mvn -Dmaven.test.failure.ignore=true clean package'
                }
            }
            post{
                success {
                    junit '**/target/surefire-reports/*.xml'
                }
            }
        }
        stage('Sonar scan') {
            when { expression { true } }
            steps {
                container('sonarcli') {
                    withSonarQubeEnv(credentialsId: 'sonar', installationName: 'sonarserver') {
                        sh '''/opt/sonar-scanner/bin/sonar-scanner \
                            -Dsonar.projectKey=petclinic \
                            -Dsonar.projectName=petclinic \
                            -Dsonar.projectVersion=1.0 \
                            -Dsonar.sources=src/main \
                            -Dsonar.tests=src/test \
                            -Dsonar.java.binaries=target/classes  \
                            -Dsonar.language=java \
                            -Dsonar.sourceEncoding=UTF-8 \
                            -Dsonar.java.libraries=target/classes
                            '''
                    }
                }
            }
        }
    }
}
```

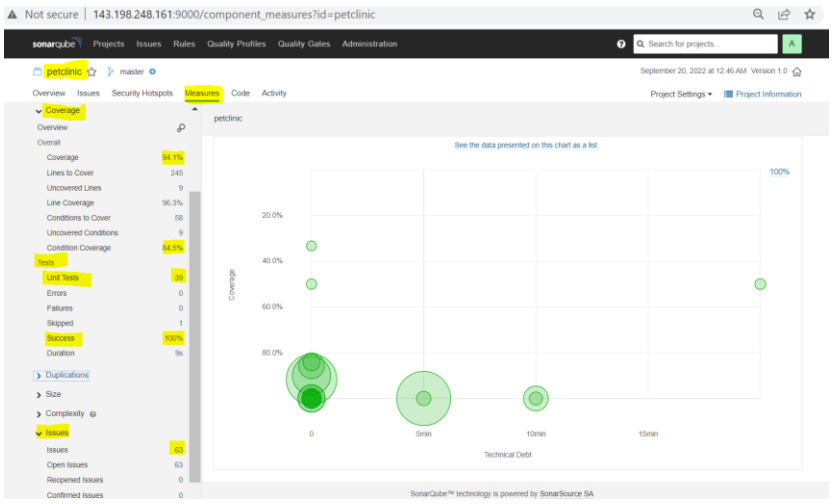
After you click on Build Now button, you can observe that 4 PODs are created in the spring-petclinic project.

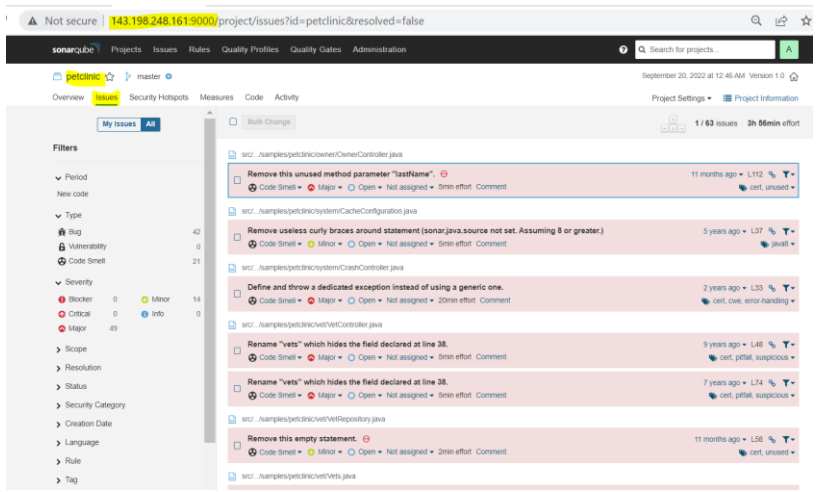
```
User@DESKTOP-2QD05T2 MINGW64 /c/DEVOPS/JENKINS
$ kubectl get po
NAME                                    READY   STATUS    RESTARTS   AGE
jenkins-0                               2/2     Running   0          29h
sonarqube-postgresql-0                  1/1     Running   0          4h29m
sonarqube-sonarqube-0                   1/1     Running   0          4h29m
spring-petclinic-14-3mfm2-9wr24-f39rm   4/4     Running   0          20s

User@DESKTOP-2QD05T2 MINGW64 /c/DEVOPS/JENKINS
$
```

The job has run successfully and your sonarqube results are also published as shown below.

```
INFO: CPD Executor 10 files had no CPD blocks
INFO: CPD Executor Calculating CPD for 25 files
INFO: CPD Executor CPD calculation finished (done) | time=37ms
INFO: Analysis report generated in 167ms, dir size=750.1 kB
INFO: Analysis report compressed in 225ms, zip size=255.2 kB
INFO: Analysis report uploaded in 148ms
INFO: ANALYSIS SUCCESSFUL, you can find the results at: http://143.198.248.161:9000/dashboard?id=petclinic
INFO: Note that you will be able to access the updated dashboard once the server has processed the submitted ana
INFO: More about the report processing at http://143.198.248.161:9000/api/ce/task?id=AYNXgMq3BuBd1lYYGF83
INFO: Analysis total time: 42.278 s
INFO: ------------------------------------------------------------------------
INFO: EXECUTION SUCCESS
INFO: ------------------------------------------------------------------------
INFO: Total time: 50.285s
INFO: Final Memory: 24M/104M
INFO: ------------------------------------------------------------------------
[Pipeline] }
[Pipeline] // withSonarQubeEnv
[Pipeline] }
[Pipeline] // container
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] }
[Pipeline] // podTemplate
[Pipeline] End of Pipeline
Finished: SUCCESS
```





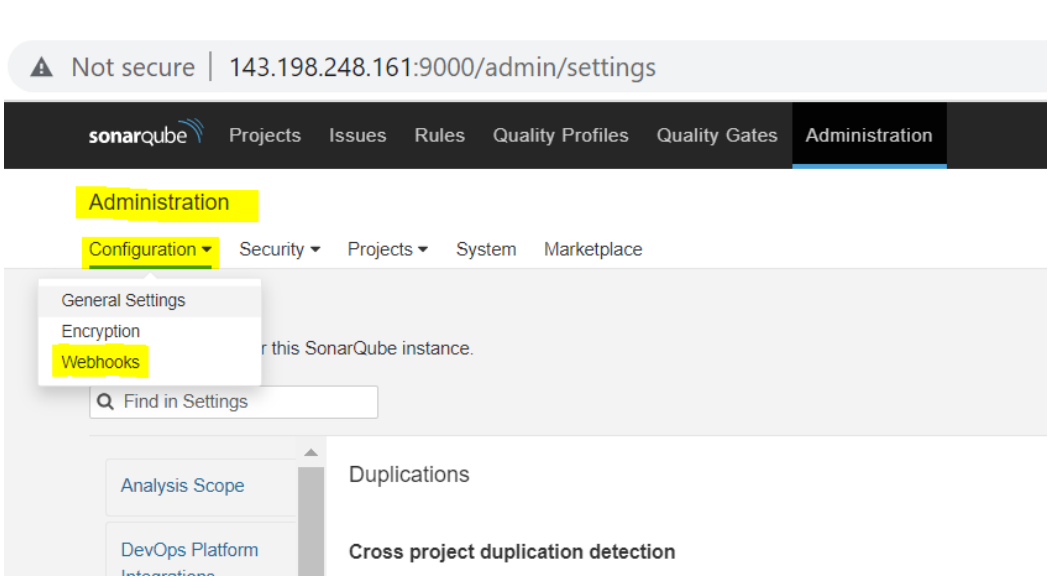


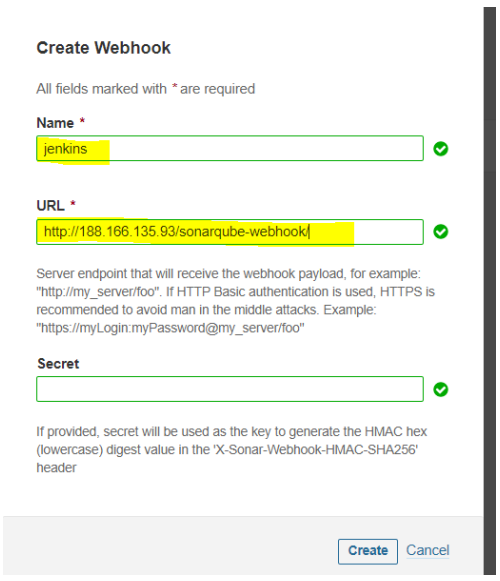
Now till now everything is fine but the main issue is I am not getting the Quality Gate results back to my Jenkins pipeline so that based on Quality Gate information I can decide if I can pass the pipeline or fail the pipeline. At present the job will be SUCCESS even if your Quality Gate result FAILS that is because their Quality Gate results are not sent back to your pipeline.

To get the Quality Gate results back to the pipeline we need to write one more stage as shown below. But after running the Quality Gates, sonarqube should send these results to Jenkins which is possible through web hooks. We need to create the web hook first then write the below stage for deciding if the Quality Gate is Pass or Fail.

NOTE: Here, URL syntax is *yourJenkinsIpAddress/sonarqube-webhook/*

Once this webhook is created then your pipeline can read the results and decide to PASS or FAIL





After creating the above webhook add this highlighted stage to your pipeline code.

```
                -Dsonar.projectKey=petclinic \
                -Dsonar.projectName=petclinic \
                -Dsonar.projectVersion=1.0 \
                -Dsonar.sources=src/main \
                -Dsonar.tests=src/test \
                -Dsonar.java.binaries=target/classes  \
                -Dsonar.language=java \
                -Dsonar.sourceEncoding=UTF-8 \
                -Dsonar.java.libraries=target/classes
                '''
            }
          }
        }
      } // abort the pipeline if the Quality Gate is FAILED
      stage('Wait For Quality Gate') {
        when { expression { true } }
        steps {
          container('sonarcli') {
            timeout(time: 1, unit: 'HOURS') {
                waitForQualityGate abortPipeline: true
            }
          }
        }
      }
    }
}
```

Script ?

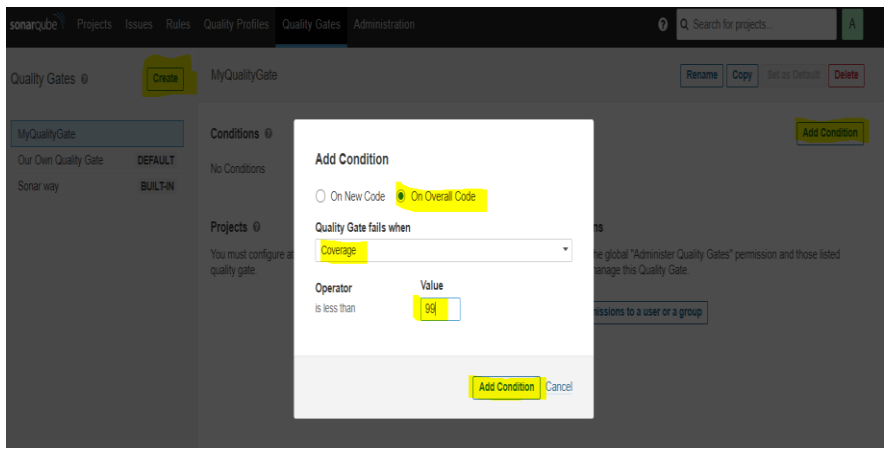
```
                withSonarQubeEnv(credentialsId: 'sonar', installationName: 'sonarserv
                  sh '''/opt/sonar-scanner/bin/sonar-scanner \
                    -Dsonar.projectKey=petclinic \
                    -Dsonar.projectName=petclinic \
                    -Dsonar.projectVersion=1.0 \
                    -Dsonar.sources=src/main \
                    -Dsonar.tests=src/test \
                    -Dsonar.java.binaries=target/classes  \
                    -Dsonar.language=java \
                    -Dsonar.sourceEncoding=UTF-8 \
                    -Dsonar.java.libraries=target/classes
                    '''
                }
              }
            }
        } // abort the pipeline if the Quality Gate is FAILED
        stage('Wait For Quality Gate') {
          when { expression { true } }
          steps {
            container('sonarcli') {
              timeout(time: 1, unit: 'HOURS') {
                  waitForQualityGate abortPipeline: true
              }
            }
          }
        }
    }
```

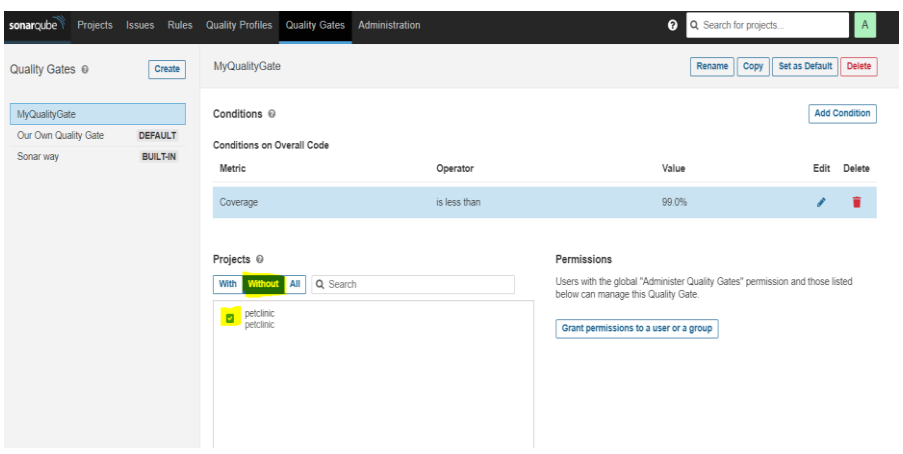
After creating the webhook and adding the stage Wait For Quality Gate, now the Quality Gate info is transferred back to our pipeline which we can see in the below screenshot.

```
[Pipeline] {
[Pipeline] waitForQualityGate
Checking status of SonarQube task 'AYNXpzmkBuBd1lYYGF86' on server 'sonarserver'
SonarQube task 'AYNXpzmkBuBd1lYYGF86' status is 'IN_PROGRESS'
SonarQube task 'AYNXpzmkBuBd1lYYGF86' status is 'SUCCESS'
SonarQube task 'AYNXpzmkBuBd1lYYGF86' completed. Quality gate is 'OK'
[Pipeline] }
[Pipeline] // timeout
[Pipeline] }
[Pipeline] // container
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] }
[Pipeline] // podTemplate
[Pipeline] End of Pipeline
Finished: SUCCESS
```

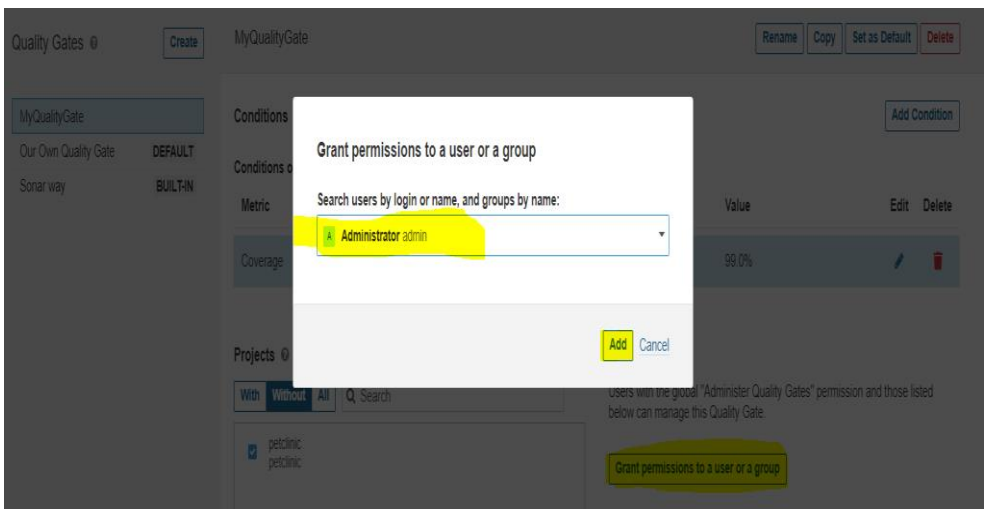
Now to prove that it is exactly picking our changes, we can click on Create button to create our own Quality Gate and (MyQualityGate) → Add Condition → Select On Overall Code → add the value 99 just to fail the test case.

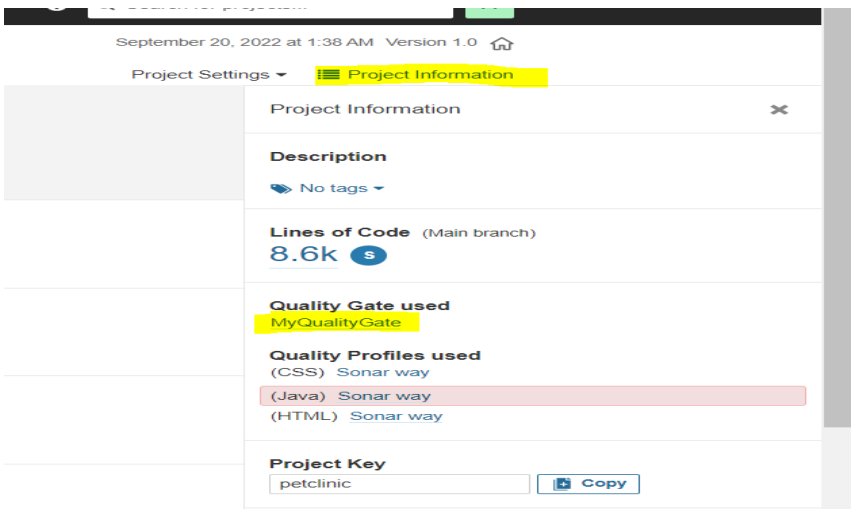
Then Select MyQualityGate → Click on Without → click the checkbox as shown below to add this Quality Gate to our petclinic project.





After that click on *Grant permissions to a user or a group* and select Administrator then click on Add button. If you go to your project petclinic on sonarqube and check the Project Information you will observe that ***MyQualityGate*** is being used as Quality Gate as shown below.

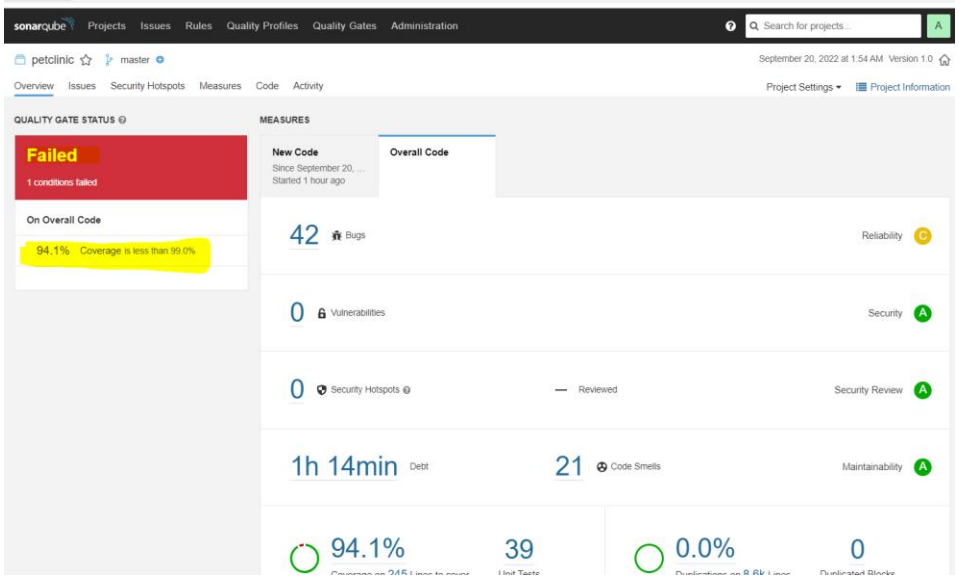




Now let's run the job and see if it's actually failing the test case or not because of coverage value in our Quality Gate (MyQualityGate).

Here, we can see that our ***Jenkins pipeline*** is FAILING the test case because the ***Quality Gate*** is FAILING in sonarqube.

```
Checking status of SonarQube task 'AYNXv5NDBuBd1lYYGF9J' on server 'sonarserver'
SonarQube task 'AYNXv5NDBuBd1lYYGF9J' status is 'IN_PROGRESS'
SonarQube task 'AYNXv5NDBuBd1lYYGF9J' status is 'SUCCESS'
SonarQube task 'AYNXv5NDBuBd1lYYGF9J' completed. Quality gate is 'ERROR'
[Pipeline] }
[Pipeline] // timeout
[Pipeline] }
[Pipeline] // container
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] }
[Pipeline] // podTemplate
[Pipeline] End of Pipeline
ERROR: Pipeline aborted due to quality gate failure: ERROR
Finished: FAILURE
```

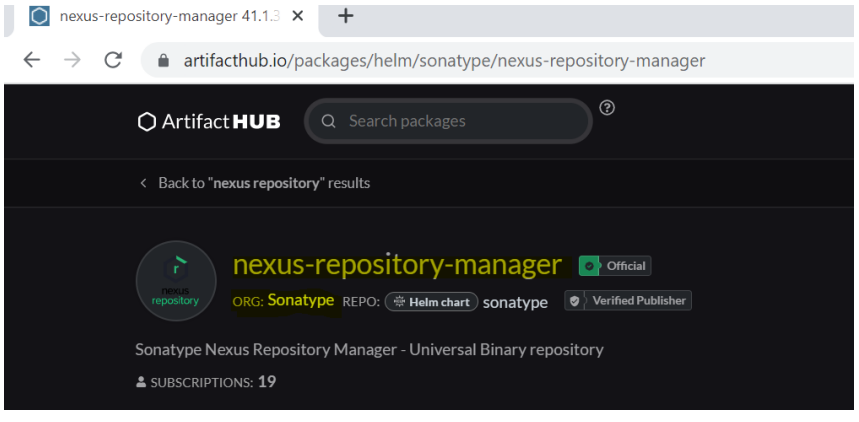


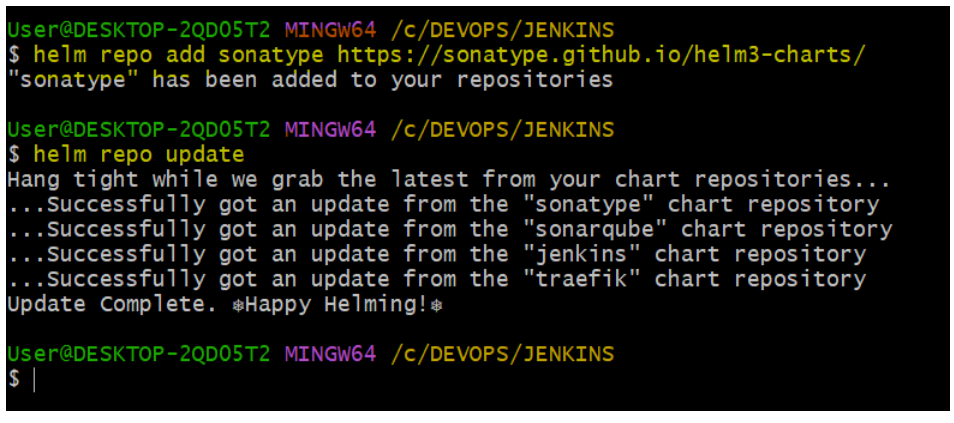
NOTE: *Now that we are getting a response back to Jenkins from SonarQube, it's time to store this generated artifact on NEXUS repository*

Step 9: Let's create a NEXUS repository to store the generated artifacts from our build.

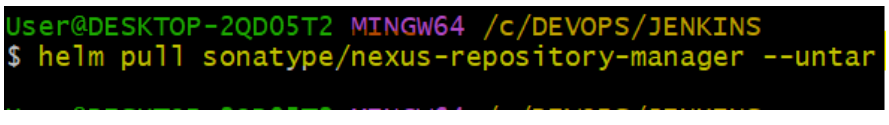
To install NEXUS repository, let's search it on artifacthub.io, we can use the below repository and first add the repository before installing it as shown below.

NOTE: We can use these two commands to add and update the repository as highlighted.

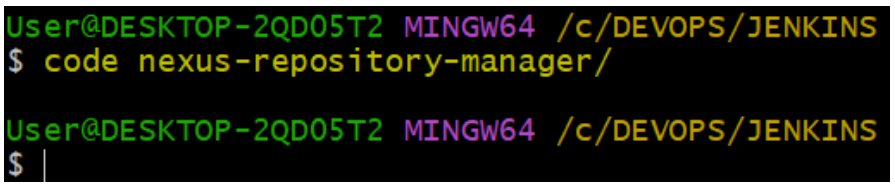




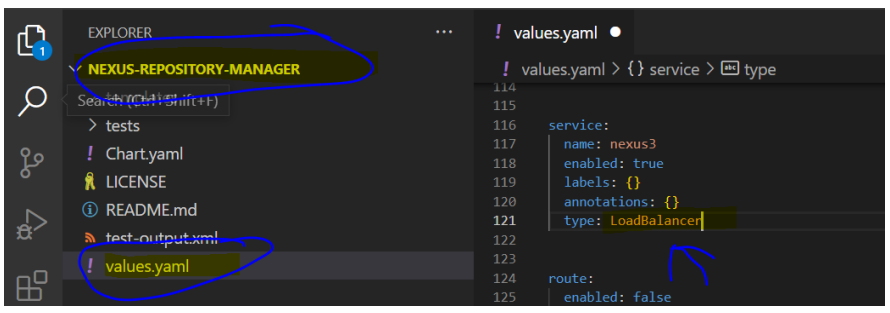
Now, pull the repository and untar it then makes the necessary changes as shown below, once it's updated then we can install it.



You can open this repository on Vscode editor by the below command.



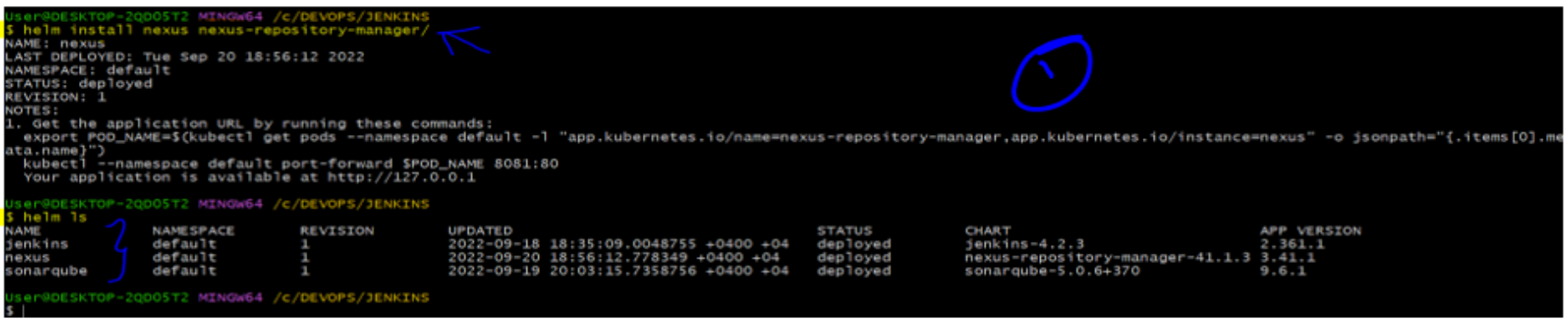
Once the file is opened make the necessary changes in ***value.yaml*** file. Here, in this file we are making only one change that is changing the type from ClusterIP to LoadBalancer as shown below.

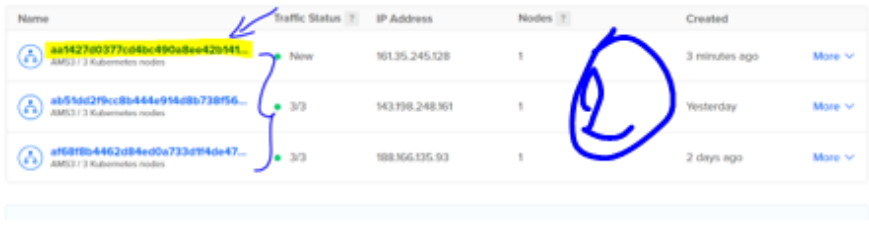


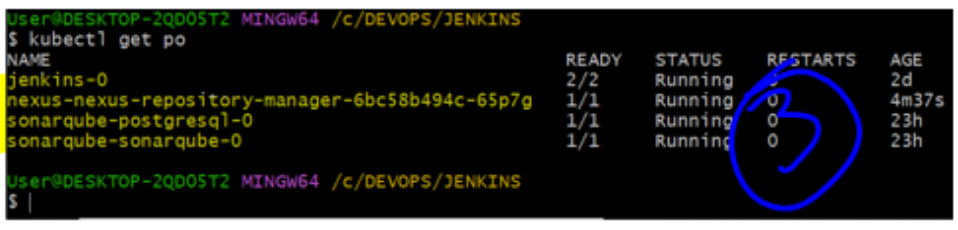
Now that you modified the required changes, you can install the nexus repository using HELM command as shown below.

NOTE:

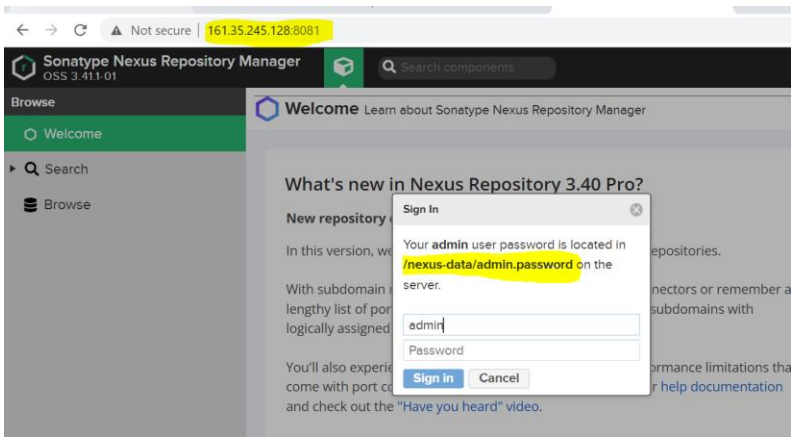
- *In 1st diagram we can see the command to install nexus and we can see that there are 3 repositories are installed on HELM now (Jenkins, SonarQube and Nexus).*
- *In 2nd diagram we can see that now total 3 LoadBalancers are created for each service (Jenkins/ SonarQube / Nexus)*
- *In 3rd diagram we can see that now total 4 PODs are created, 1 for Jenkins, 2 for SonarQube and Sonar Postgresql and 1 for Nexus.*
- *In 4th diagram we can see that these 4 PODs are shown on DigitalOcean as well.*

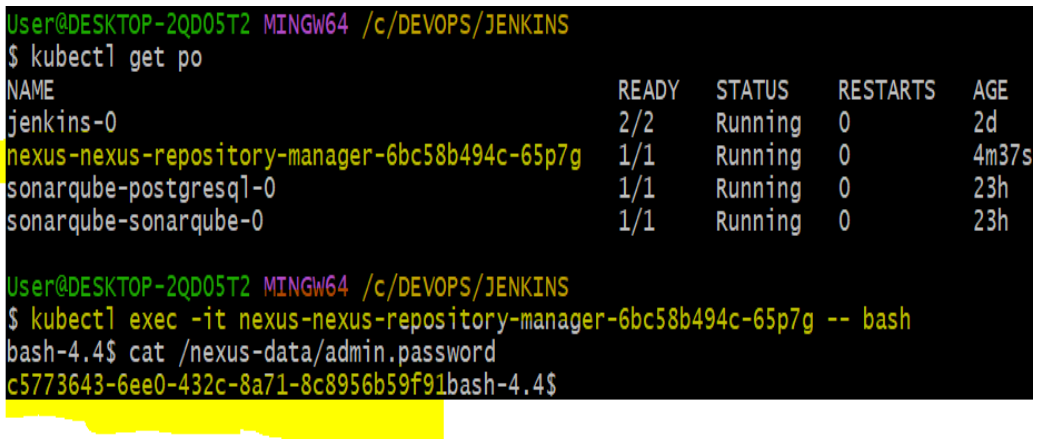






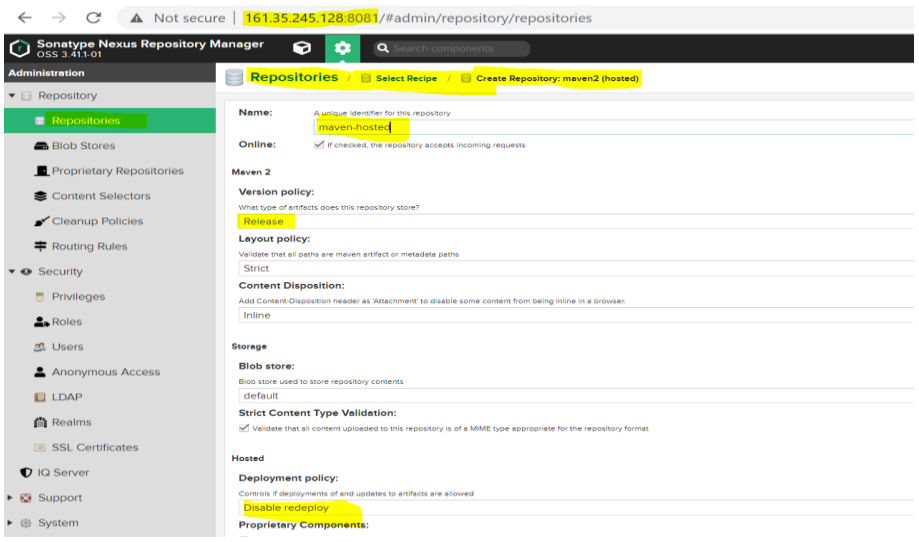
Now, it's time to login to our NEXUS and see but to get the password for your nexus repository you need to go inside your nexus POD and get the password in the highlighted location. It will prompt to change the password so I gave the password as ***Nexus123*** → Disable anonymous access → Finish

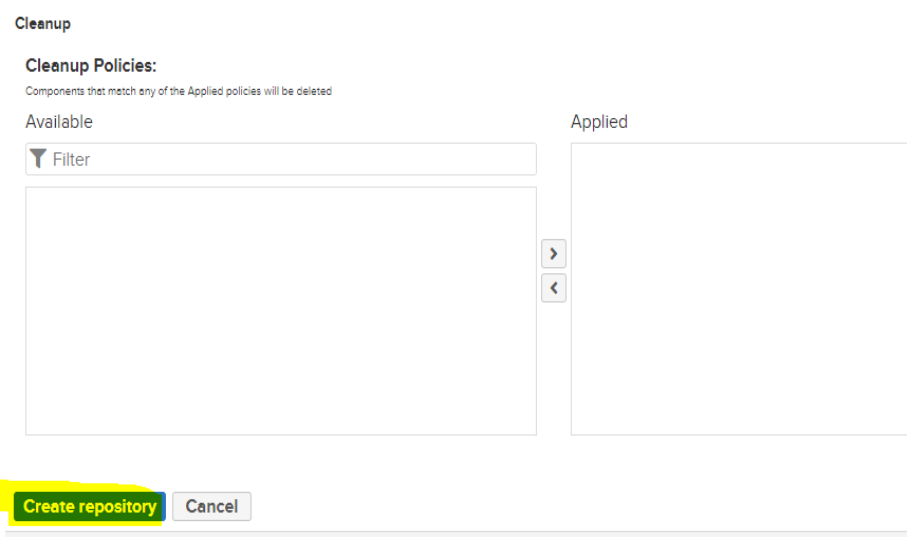




Now, to create a maven hosted repository we need to click on Settings button → click on Repositories → Create Repository → Select maven2 (hosted) → then update the following details.

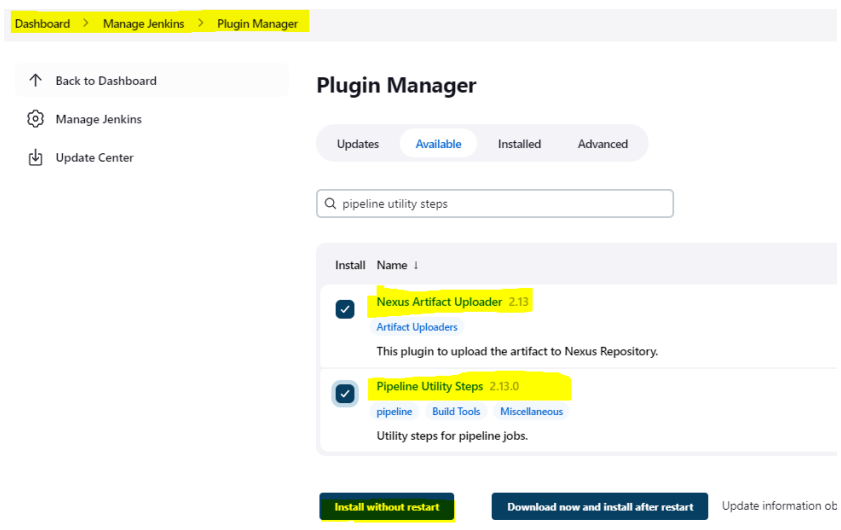
NOTE: *We are creating a Release Version policy, for that we have to choose Disable redeploy option because it will make sure that only the unique version number of the artifacts are accepted and we can not redeploy the artifact with the same version again and again.*

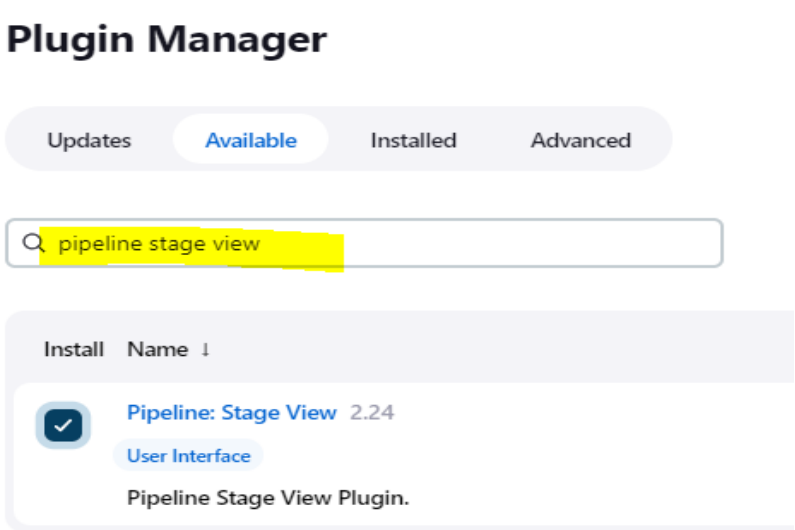




Now that we created maven2 (hosted) repository, now it's time install ***NEXUS*** plugin and ***Pipeline Utility Steps*** plugin onto your Jenkins along with credentials, then add the new stage for NEXUS artifacts. This ***Pipeline Utility Steps*** plugin is useful for finding the files, reading the CSV files and text documents, extracting ZIP files etc, which we commonly do in our pipeline.

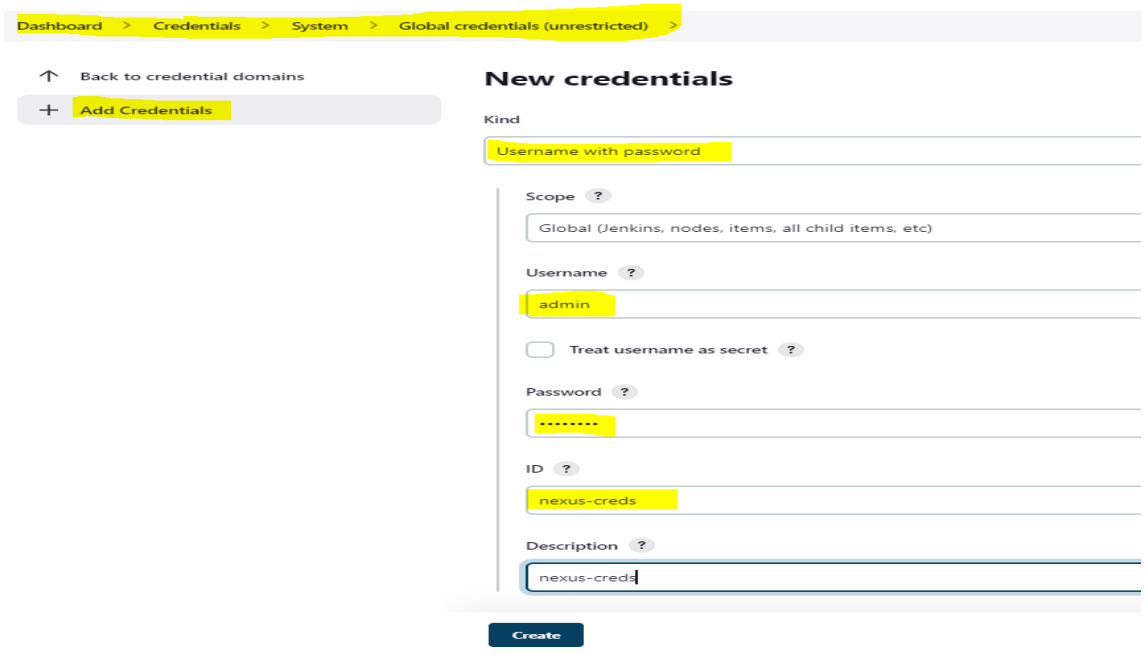
You can install one more plugin which will help to show the stage level views of our pipeline and the plugin name is ***Pipeline Stage View***.





Now, add the credentials in Jenkins for Maven.

Manage Jenkins → Manage Credentials → now click on ***Jenkins*** under ***Stores scope to Jenkins*** title → click on ***Global Credentials (unrestricted)*** → Add Credentials and mention the below details.



Now, let's write a pipeline code to upload the artifacts which is ***Publish Maven Artifacts To Nexus*** stage, at this point of time we are uploading the artifacts which are not containerized.

NOTE:

- In 1st diagram we are using JNLP container because this is just a API call to upload the artifacts to NEXUS repository.
- In 2nd diagram we are showing the code which will be part of this stage, this code will come inside the container part.
- In 3rd part we combined both the part and included in a stage and the highlighted environment variables we need to define.
- In 4th diagram we are showing that we are defining the environment variables which are highlighted in 3rd diagram. Here, nexus version is 3, protocol is http, url is nexus IpAddress and port without http and slash, credential_id is nexus-cred which we used while creating the credential for nexus in Jenkins in above steps.
- The artifact details will be pulled from the ***POM.xml*** file of your ***spring-petclinic*** project and it will be used.

```
                -Dsonar.sourceEncoding=UTF-8 \
                -Dsonar.java.libraries=target/classes
                '''
              }
            }
          }
        } // abort the pipeline if the Quality Gate is FAILED
        stage('Wait For Quality Gate') {
          when { expression { true } }
          steps {
            container('sonarcli') {
              timeout(time: 1, unit: 'HOURS') {
                waitForQualityGate abortPipeline: true
              }
            }
          }
        }
        stage('Publish Maven Artifacts To Nexus') {
          steps {
            container('jnlp') {

            }
          }
        }
      }
    }
```

github.com/asifqa123/jenkins-end-to-end-pipeline

README.md

Nexus Stage

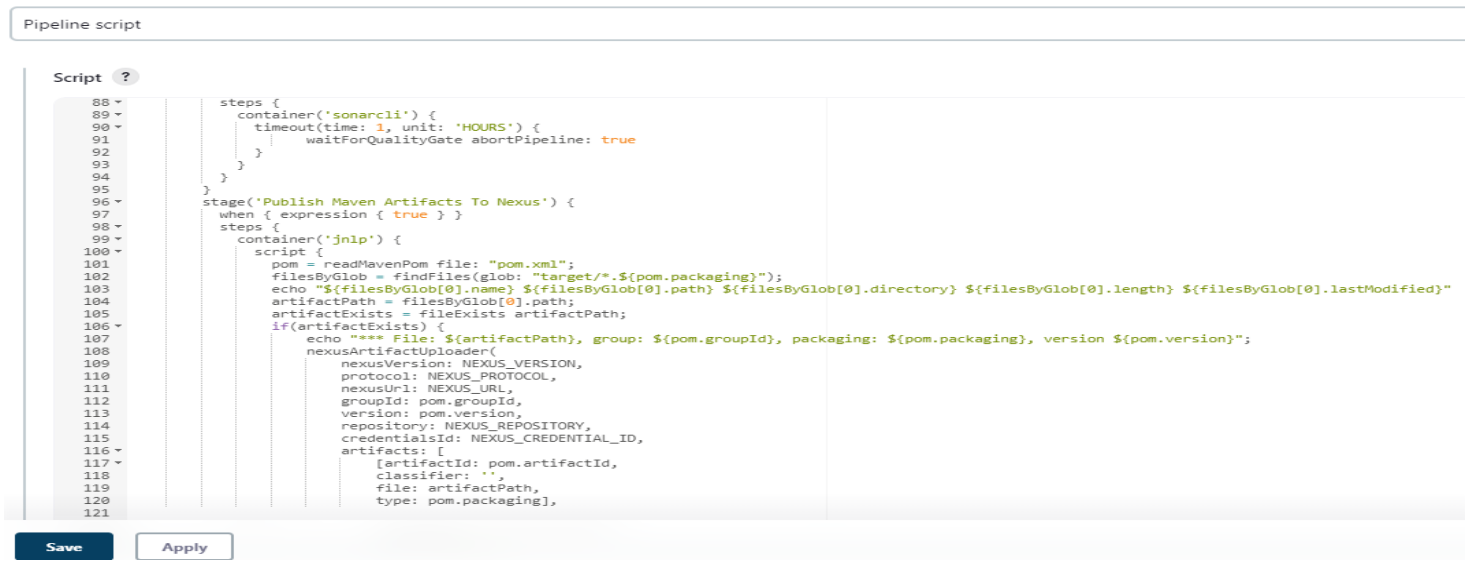
https://blog.sonatype.com/workflow-automation-publishing-artifacts-to-nexus-using-jenkins-pipelines

```
script {
    pom = readMavenPom file: "pom.xml";
    filesByGlob = findFiles(glob: "target/*.${pom.packaging}");
    echo "${filesByGlob[0].name} ${filesByGlob[0].path} ${filesByGlob[0].directory} ${filesByGlob[0].length}
    artifactPath = filesByGlob[0].path;
    artifactExists = fileExists artifactPath;
    if(artifactExists) {
        echo "*** File: ${artifactPath}, group: ${pom.groupId}, packaging: ${pom.packaging}, version ${pom.ve
        nexusArtifactUploader(
            nexusVersion: NEXUS_VERSION,
            protocol: NEXUS_PROTOCOL,
            nexusUrl: NEXUS_URL,
            groupId: pom.groupId,
            version: pom.version,
            repository: NEXUS_REPOSITORY,
            credentialsId: NEXUS_CREDENTIAL_ID,
            artifacts: [
                [artifactId: pom.artifactId,
                classifier: '',
                file: artifactPath,
                type: pom.packaging],

                [artifactId: pom.artifactId,
                classifier: '',
                file: "pom.xml",
                type: "pom"]
            ]
        );

    } else {
        error "*** File: ${artifactPath}, could not be found";
    }
}
```

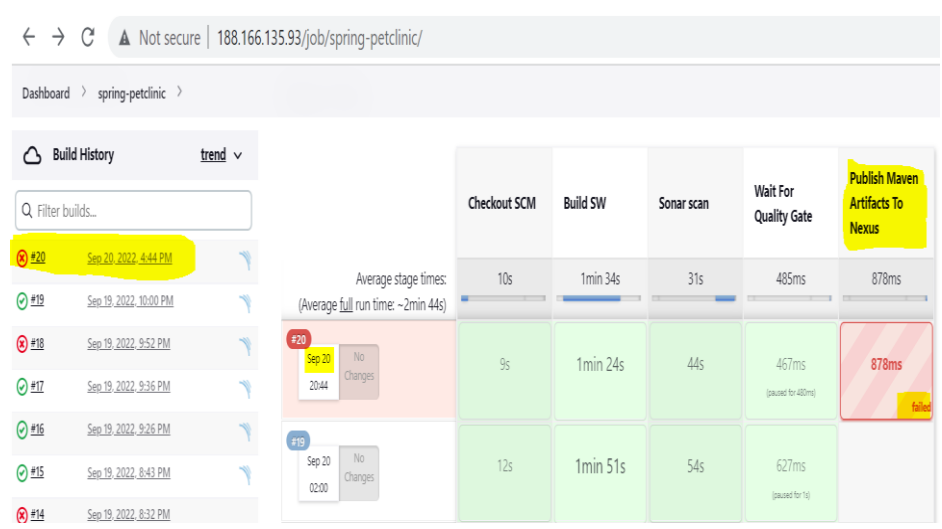
```
    stage('Publish Maven Artifacts To Nexus') {
      when { expression { true } }
      steps {
        container('jnlp') {
          script {
            pom = readMavenPom file: "pom.xml";
            filesByGlob = findFiles(glob: "target/*.${pom.packaging}");
            echo "${filesByGlob[0].name} ${filesByGlob[0].path} ${filesByGlob[0].directory} ${
            artifactPath = filesByGlob[0].path;
            artifactExists = fileExists artifactPath;
            if(artifactExists) {
                echo "*** File: ${artifactPath}, group: ${pom.groupId}, packaging: ${pom.packa
                nexusArtifactUploader(
                    nexusVersion: NEXUS_VERSION,
                    protocol: NEXUS_PROTOCOL,
                    nexusUrl: NEXUS_URL,
                    groupId: pom.groupId,
                    version: pom.version,
                    repository: NEXUS_REPOSITORY,
                    credentialsId: NEXUS_CREDENTIAL_ID,
                    artifacts: [
                        [artifactId: pom.artifactId,
                        classifier: '',
                        file: artifactPath,
                        type: pom.packaging],

                        [artifactId: pom.artifactId,
                        classifier: '',
                        file: "pom.xml",
                        type: "pom"]
                    ]
                );

            } else {
                error "*** File: ${artifactPath}, could not be found";
            }
        }
```

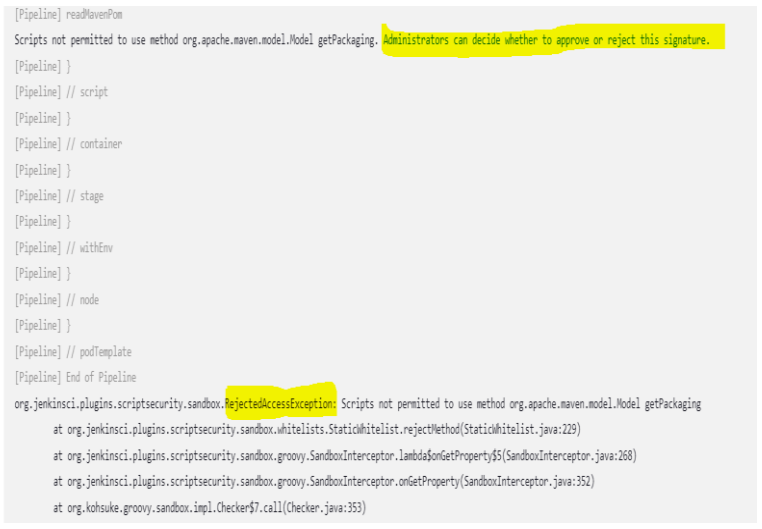
```
            image: sonarsource/sonar-scanner-cli:latest
            command:
            - cat
            tty: true
          volumes:
          - name: cache
            persistentVolumeClaim:
              claimName: maven-cache
        '''
      }
    }
    environment {
      NEXUS_VERSION = "nexus3"
      NEXUS_PROTOCOL = "http"
      NEXUS_URL = "161.35.245.128:8081"
      NEXUS_REPOSITORY = "maven-hosted"
      NEXUS_CREDENTIAL_ID = "nexus-creds"
    }
    stages {
      stage('Checkout SCM') {
        when { expression { true } }
        steps {
          container('git') {
            git branch: 'main', url: 'https://github.com/a
```

Now, copy this whole pipeline code and paste it on your Jenkins spring-petclinic job and run the job by clicking on Build Now button.



You will observe that the job is failing at the last stage which is Publish Maven Artifacts To Nexus that is because of a permission.

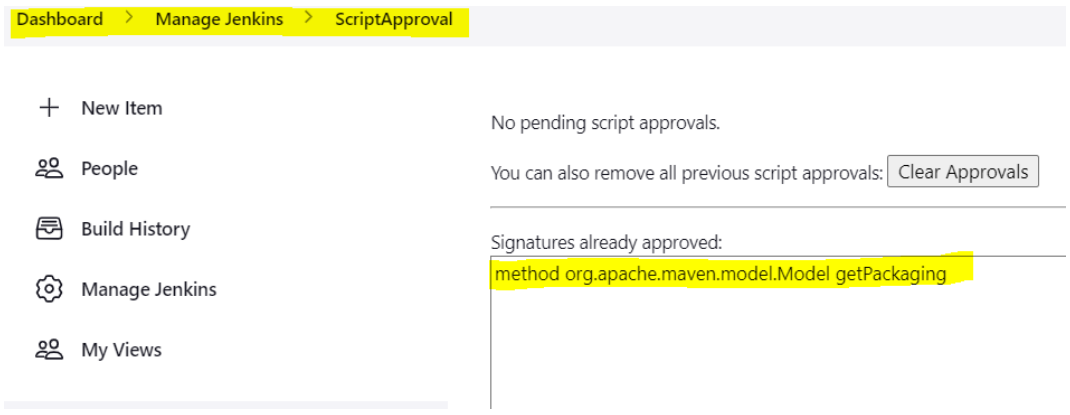




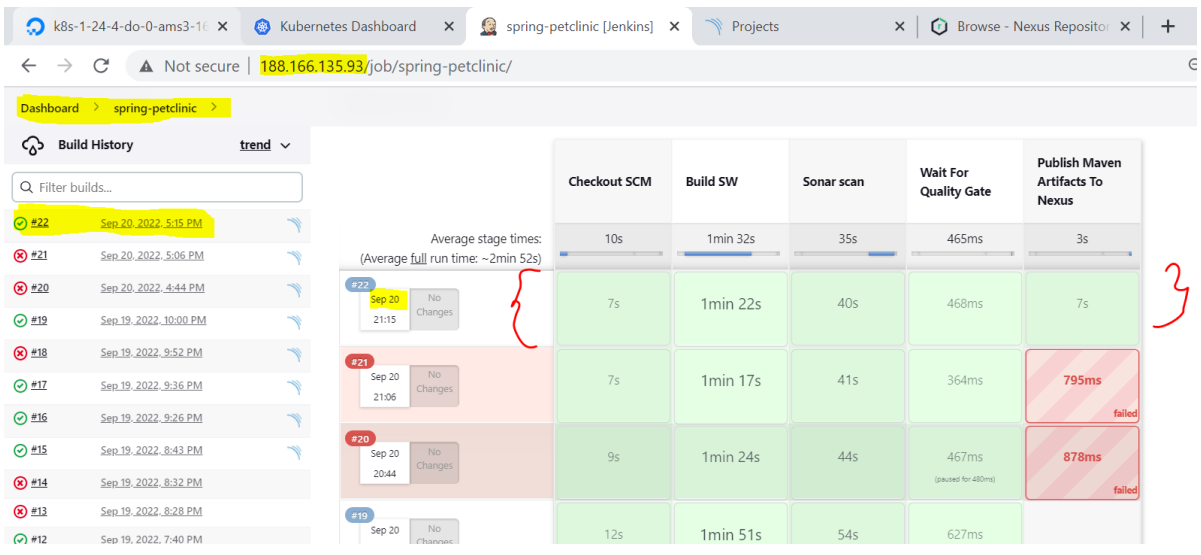
To solve this error we need to grant the permission at the below location.

Manage Jenkins → In-process Script Approval → click on Approve button → After this build the pipeline job again.

NOTE: You will see this signature after clicking on Approve button which is disappeared after clicking.



After performing the above changes, if you build the job again then you will observe that it's passing this time without any issues.



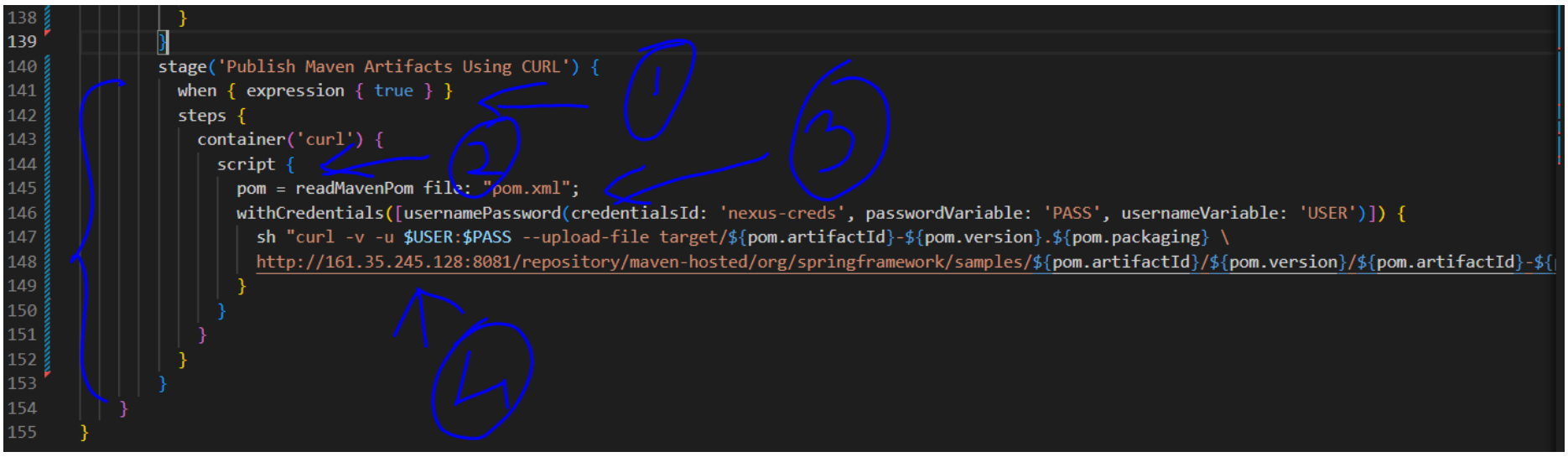
Now, let's see one more method which can be used to upload our artifacts to NEXUS which is using CURL command.

NOTE: You are fee to use any one of the methods, we are discussing about CURL command just for our knowledge. For that we have to pass false value to the ***when*** condition as shown below.

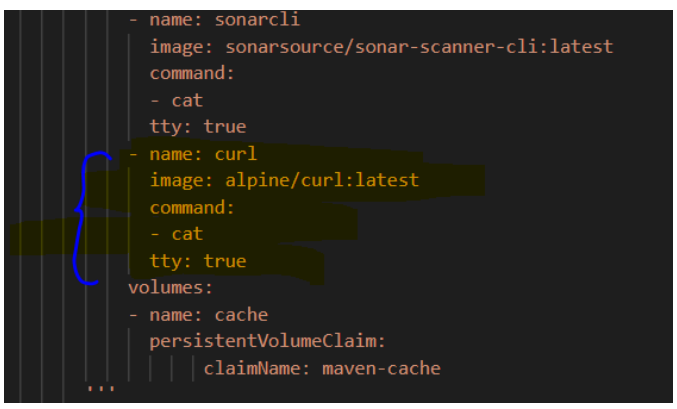
After passing the false value to the stage Publish Maven Artifacts To Nexus, we created a new stage Publish Maven Artifacts Using CURL as shown below.

NOTE:

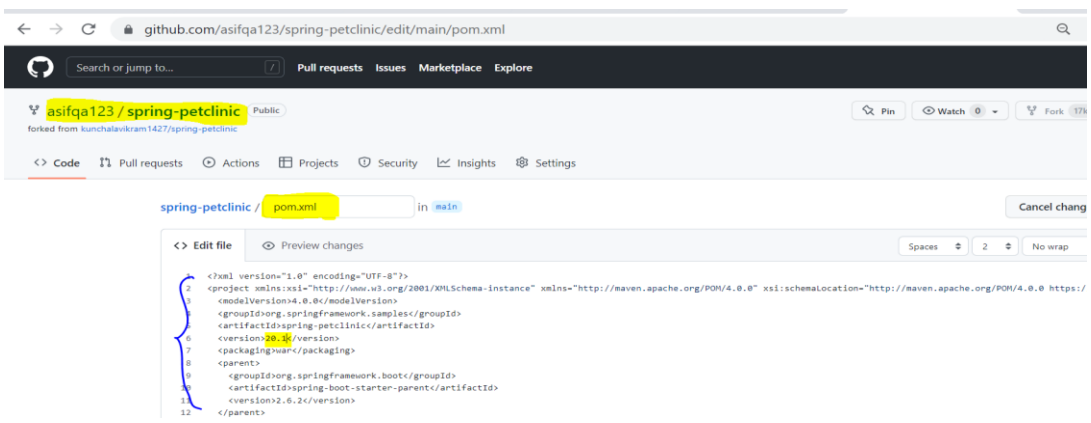
- In 1st point, we are showing that we add when expression and passing value as true to this stage.
- In 2th point, we are showing that this code is a part of SCRIPT block because this is a scripted syntax.
- In 3rd point, we are showing that we added this line to read the parameters from POM library of your project.
- In 4th point, we are showing that NEXUS url is being used while uploading.

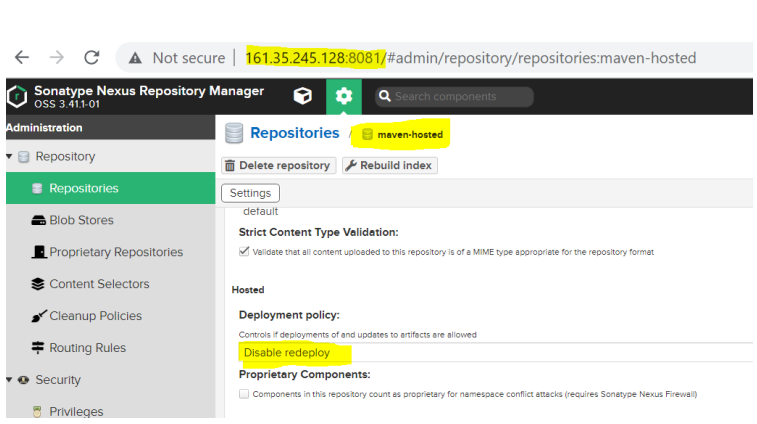


After this we have to add the below code to include CURL image in our Kubernetes agent section of the pipeline code.

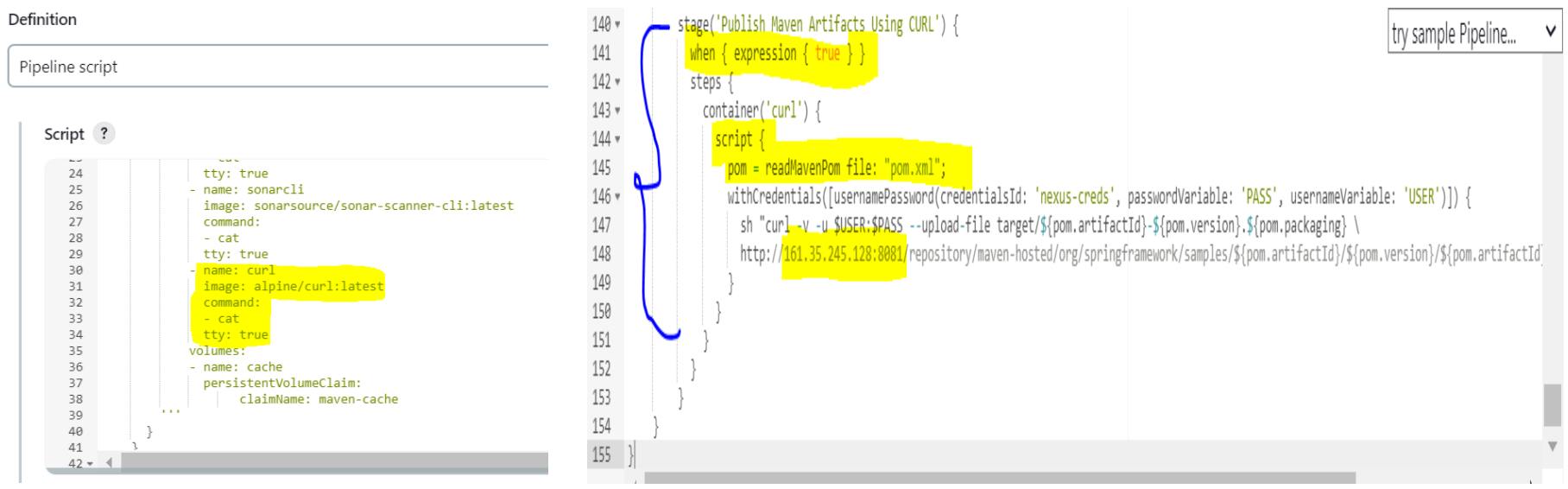


Before that we have to update the version number in the ***POM*** file of our ***spring-petclinic*** repository as shown below because we created a repository ***maven-hosted*** on our ***NEXUS*** server, which will always expect a new version to upload. If we pass the same version then it will reject it.



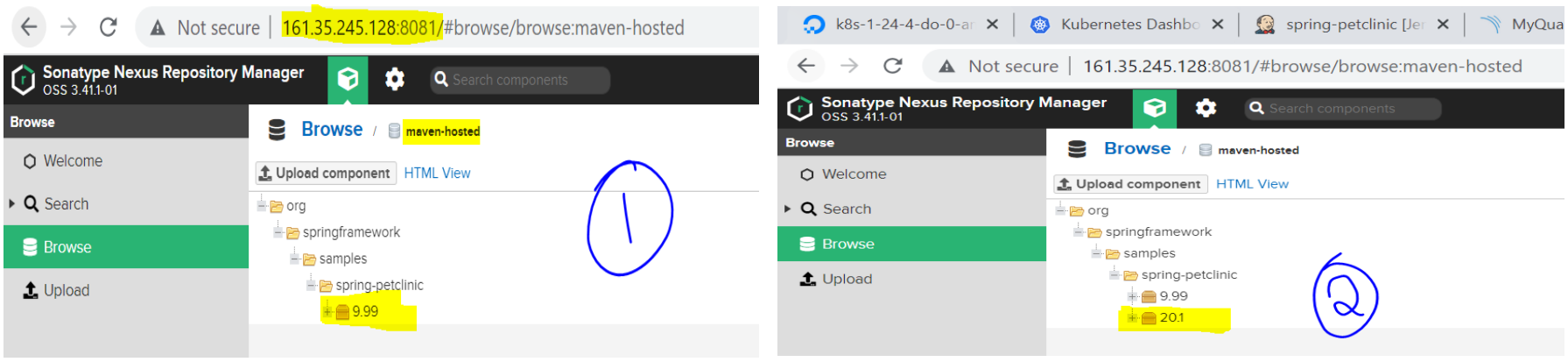


Now copy paste the complete code to your Jenkins pipeline code which you updated in the above steps.



Now let's run and see the output of our job.

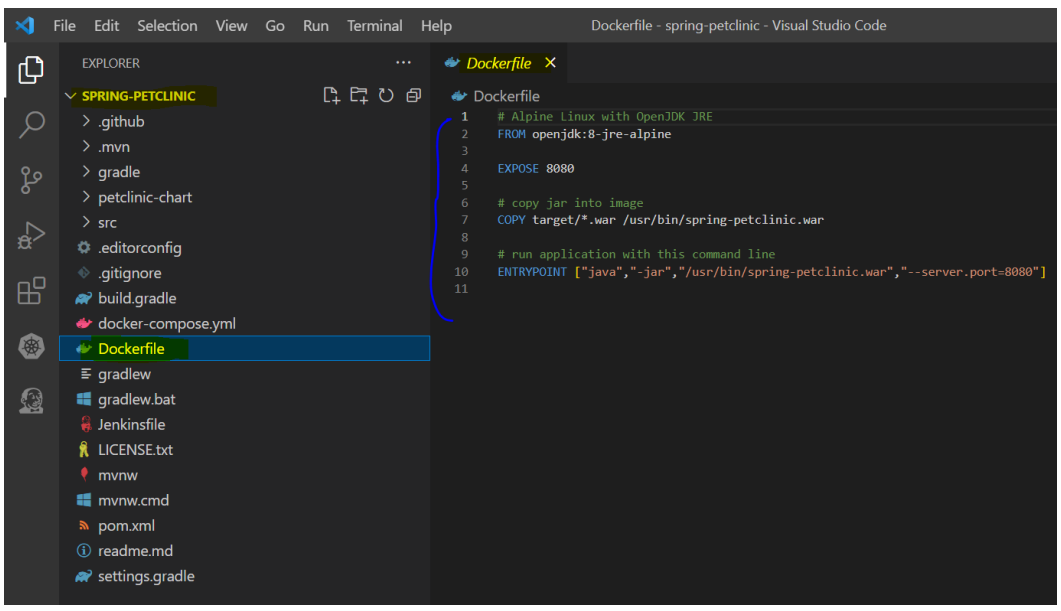
NOTE: Here, we can observe that before this change, NEXUS was having only 9.99 repo files but after uploading the new version 20.1 has been uploaded successfully.



Step 10: Now that we uploaded the WAR files to our NEXUS artifactory, it's for us to create a DOCKERFILE for our spring-petclinic project and upload it to DOCKER HUB or any server.

To containerize your spring-petclinic project, first create a Dockerfile as shown below.

NOTE: Here, we are using openjdk-8 as a base image, copying the .war file inside the target folder of the project to /usr/bin/spring-petclinic.war location and finally we are mentioning the commands which we want to execute when the container is built of this image.



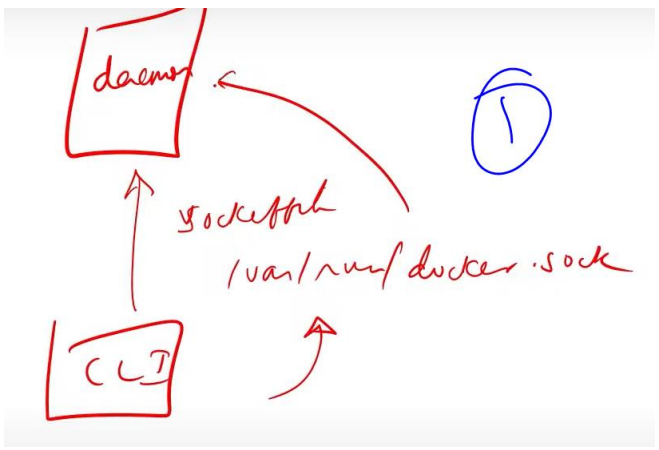
Now, to build an container image in Kubernetes environment we have two ways.

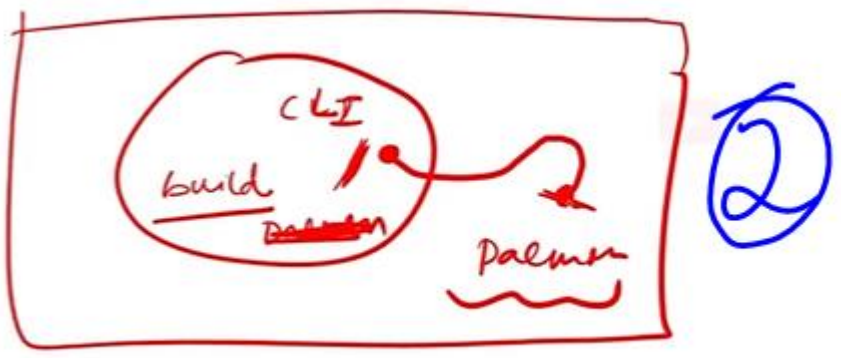
- 1st option is canico, using which we can build container images.
- 2nd option is to use Docker Daemon, which is running on the Kubernetes node

Very Important To Understand This Concept

*Now, if **Docker CLI** wants to communicate with **Docker Daemon**, then it needs socketfile which is located inside **/var/run/docker.sock**, which is shown in 1st pick.*
*So now, inside the Kubernetes environment we have various nodes and the POD can schedule onto any of the nodes, so every node will have Daemon installed and every node will have the socketfile which is **docker.sock**. If I have to build container images from within a POD, then I should have the CLI first then I need **Daemon**, since we can't have **Daemon** inside the **POD** for that I try to use the Daemon which is running **outside container** but in the same host.*
*Now, in order to communicate with **Daemon**, I will take that **/var/run/docker.sock** file from the host and I will mount it into the container, in the exact location. In this case when CLI invokes Docker build related commands, then it takes the help of that docker.sock file and it talks to the Daemon over that docker.sock file in this way **CLI** can communicate with the **Daemon** which is outside the container, this is shown in 2nd diagram.*

NOTE: If you have any private repository like NEXUS then you can directly upload your container images on it but Docker needs HTTPS connection to upload images to NEXUS and we don't have that setup with HTTPS. Hence, we are uploading our images to DOCKER HUB in this example.





Now, let's update our pipeline and run the job to upload the images to Docker Hub.

NOTE:

- In 1st point we are showing that we added docker container in Kubernetes agent.
- In 2nd point we are showing that we updated environment with DOCKER HUB details.
- In 3rd point we are showing that we build the images and tagged them, after login we are pushing the images then finally we are deleting them.

  We are doing this because we don't have HTTPS setup to our NEXUS repository or else we can directly upload the images to it instead of uploading it on DOCKER HUB.

```
33            - cat
34            tty: true
35          - name: docker
36            image: docker:latest
37            command:
38            - cat
39            tty: true
```

```
57      }
58      environment {
59          NEXUS_VERSION = "nexus3"
60          NEXUS_PROTOCOL = "http"
61          NEXUS_URL = "161.35.245.128:8081"
62          NEXUS_REPOSITORY = "maven-hosted"
63          NEXUS_CREDENTIAL_ID = "nexus-creds"
64          DOCKERHUB_USERNAME = "asifuae1983"
65          APP_NAME = "spring-petclinic"
66          IMAGE_NAME = "${DOCKERHUB_USERNAME}" + "/" + "${APP_NAME}"
67          IMAGE_TAG = "${BUILD_NUMBER}"
68      }
```

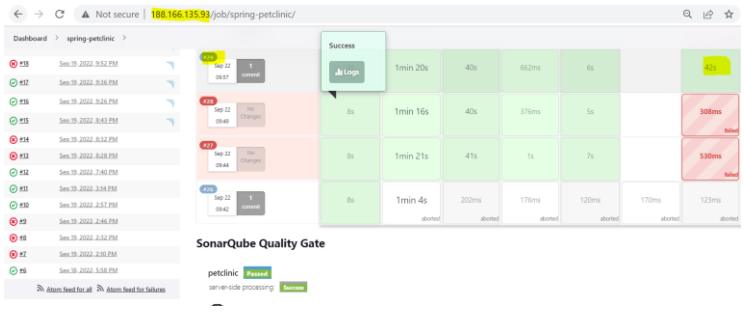
```
172         }
173       }
174       stage('Build Docker Image') {
175         when { expression { true }}
176         steps {
177           container('docker') {
178             sh "docker build -t $IMAGE_NAME:$IMAGE_TAG ."
179             sh "docker tag $IMAGE_NAME:$IMAGE_TAG $IMAGE_NAME:latest"
180             withCredentials([usernamePassword(credentialsId: 'docker', passwordVariable: 'PASS', usernameVariable: 'USER')]) {
181               sh "docker login -u $USER -p $PASS"
182               sh "docker push $IMAGE_NAME:$IMAGE_TAG"
183               sh "docker push $IMAGE_NAME:latest"
184             }
185             sh "docker rmi $IMAGE_NAME:$IMAGE_TAG"
186             sh "docker rmi $IMAGE_NAME:latest"
187           }
188         }
```

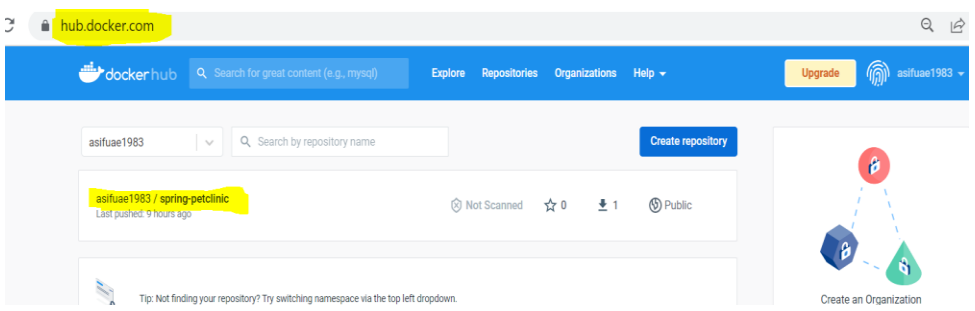
So now let's paste the complete pipeline code on our Jenkins job.

Script ?

```
170             }
171           }
172         }
173       }
174       stage('Build Docker Image') {
175         when { expression { false }}
176         steps {
177           container('docker') {
178             sh "docker build -t $IMAGE_NAME:$IMAGE_TAG ."
179             sh "docker tag $IMAGE_NAME:$IMAGE_TAG $IMAGE_NAME:latest"
180             withCredentials([usernamePassword(credentialsId: 'docker', passwordVariable: 'PASS', usernameVariable: 'USER')]) {
181               sh "docker login -u $USER -p $PASS"
182               sh "docker push $IMAGE_NAME:$IMAGE_TAG"
183               sh "docker push $IMAGE_NAME:latest"
184             }
185             sh "docker rmi $IMAGE_NAME:$IMAGE_TAG"
186             sh "docker rmi $IMAGE_NAME:latest"
187           }
188         }
189       }
```

After running the job with the new changes made to upload images to DOCKER HUB, we can see that the image got uploaded successfully to the DOCKER HUB.





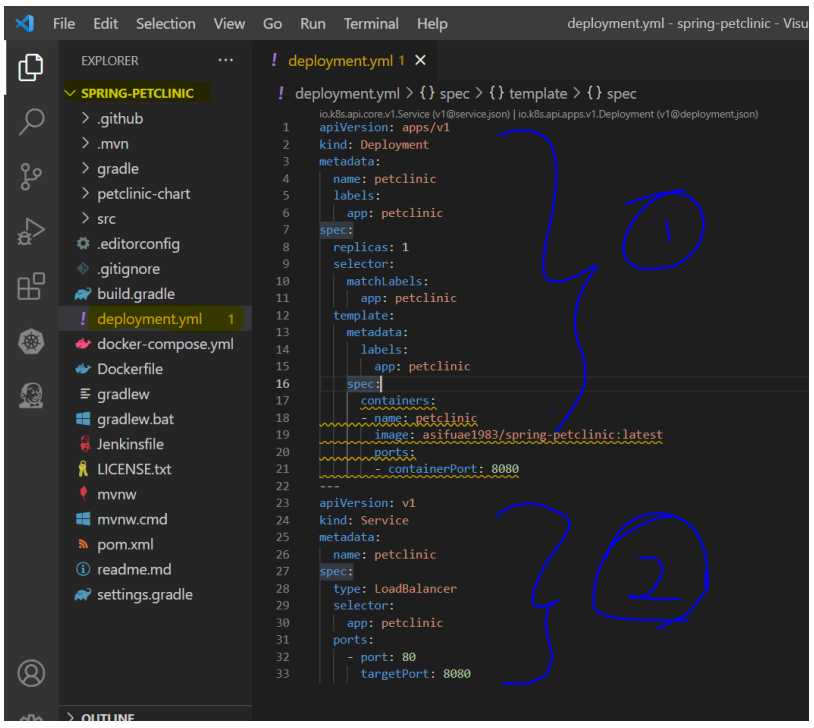
Step 10: Now that we pushed the docker image to our DOCKER HUB, it's time to deploy the app on Kubernetes.

For that first clone the repository to your laptop. (**Repository:** *https://github.com/asifqa123/spring-petclinic.git*)

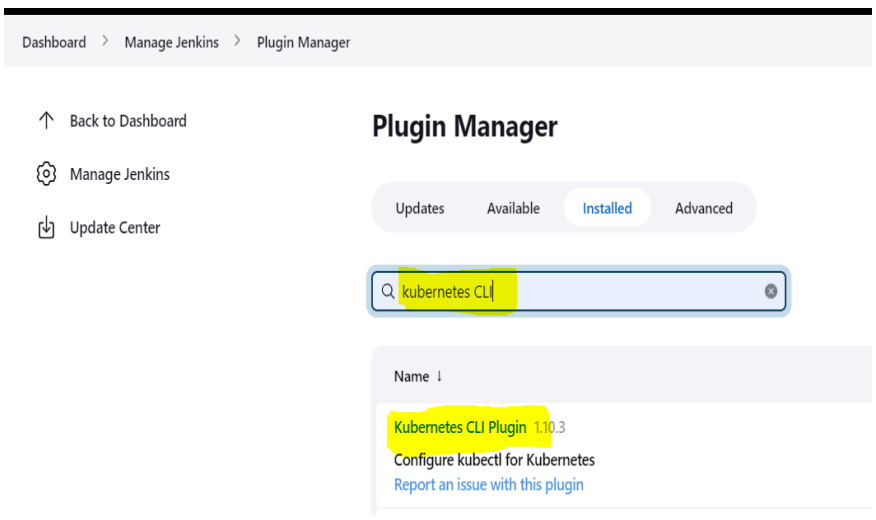
Once, the above repository is cloned on your system, you can add the Deployment and Service file related to your spring-petclinic project to deploy this application on Kubernetes.

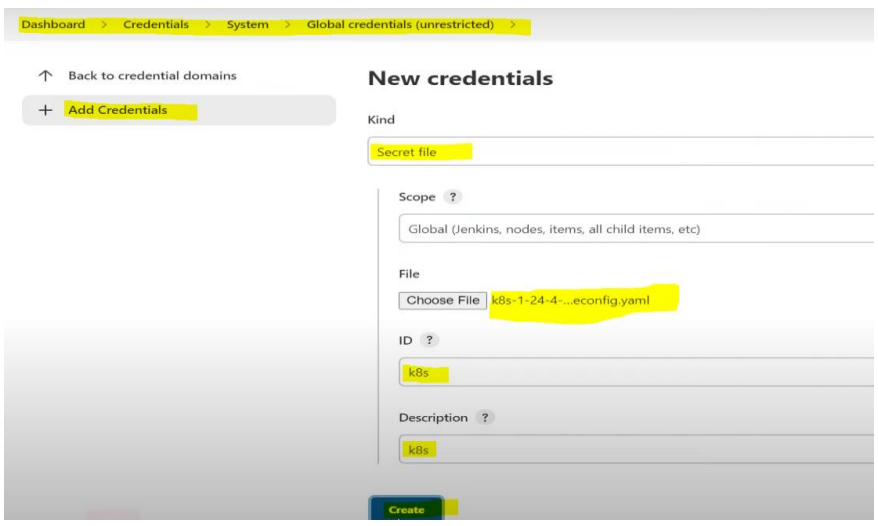
NOTE:

- In 1st point we are trying to show you the ***Deployment*** file for the image which we pushed to our ***DOCKER HUB*** account.
- In 2nd point we are trying to show you that the ***Service*** for this ***Deployment*** file is being created and the *service type* is ***LoadBalancer***.
- Now you should commit your changes and push it to your GitHub.

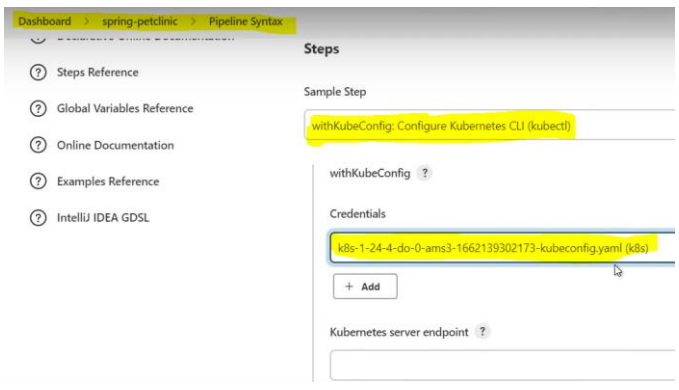


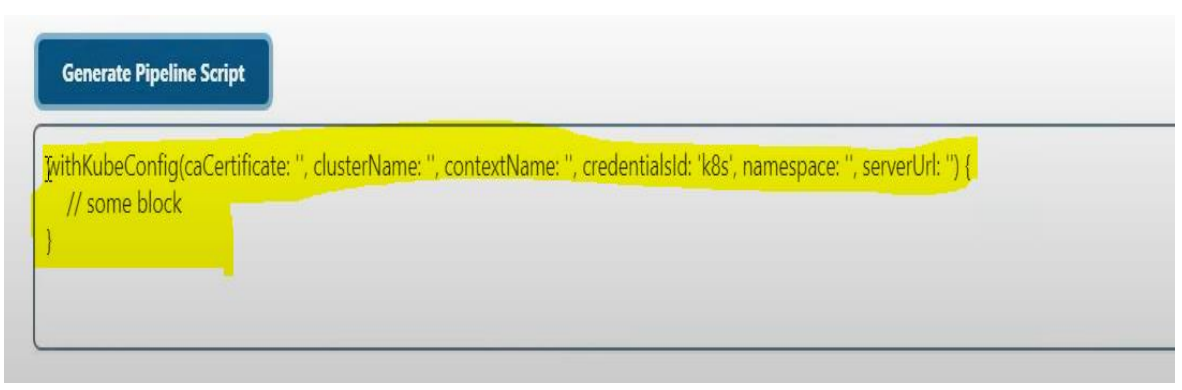
Now, you should install a plugin ***Kubernetes CLI Plugin*** on your Jenkins and add the credentials by providing the config file location which you downloaded from DigitalOcean.





Now, go to the pipeline syntax and select the shown Sample Step and select your config file in Credentials section, remaining details we don't have to fill because it will pick directly from our config file. After filling this two fields, if you click on Generate Pipeline Script you will get a code which you can use in your Jenkins pipeline.





We added a new stage in our Jenkins file for deploying our petclinic application on our Kubernetes cluster. We can copy paste the complete code on our Jenkins file.

NOTE: *Here, we used a custom image **kunchalavikram/kubectl_helm_cli** in which we are having command for downloading KUBECTL then we are moving it to /bin/kubectl and giving execute permissions.*

Dockerfile to build custom image with helm and kubectl cli tools

Image available in Dockerhub as: kunchalavikram/kubectl_helm_cli:latest

```
FROM alpine/helm
RUN curl -LO https://dl.k8s.io/release/v1.25.0/bin/linux/amd64/kubectl \
    && mv kubectl /bin/kubectl \
    && chmod a+x /bin/kubectl
```

```
    tty: true
  - name: kubectl-helm-cli
    image: kunchalavikram/kubectl_helm_cli
    command:
    - cat
    tty: true
    volumeMounts:
    - mountPath: /var/run/docker.sock
      name: docker-sock
  volumes:
  - name: cache
    persistentVolumeClaim:
        claimName: maven-cache
  - name: docker-sock
    hostPath:
        path: /var/run/docker.sock
''
}
}
```

```
    command:
    - cat
    tty: true
  - name: kubectl-helm-cli
    image: kunchalavikram/kubectl_helm_cli
    command:
    - cat
    tty: true
    volumeMounts:
    - mountPath: /var/run/docker.sock
      name: docker-sock
  volumes:
  - name: cache
    persistentVolumeClaim:
        claimName: maven-cache
  - name: docker-sock
    hostPath:
        path: /var/run/docker.sock
''
}
}
```

values.yaml
Jenkinsfile M
maven-pvc.yml
README.md

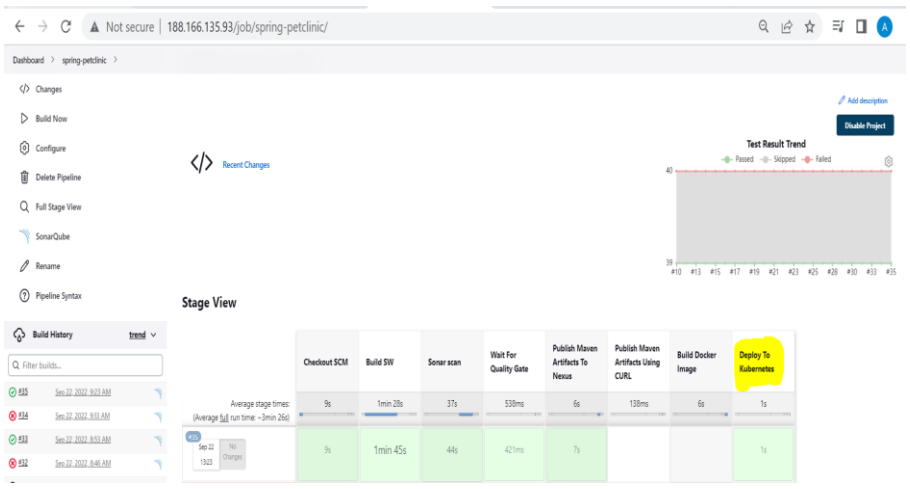
```
        }
      }
      stage('Deploy To Kubernetes') {
        when { expression { true }}
        steps {
          container('kubectl-helm-cli') {
            withKubeConfig(caCertificate: '', clusterName: '', contextName: '', credentialsId: 'k8s', namespace: '', serverUrl: '') {
                sh "kubectl apply -f deployment.yml"
            }
          }
        }
      }
    }
}
```

Pipeline script

Script ?

```
            sh "docker rmi $IMAGE_NAME:latest"
          }
        }
      }
      stage('Deploy To Kubernetes') {
        when { expression { true }}
        steps {
          container('kubectl-helm-cli') {
            withKubeConfig(caCertificate: '', clusterName: '', contextName: '', credentialsId: 'k8s', namespace: '', serverUrl: '') {
                sh "kubectl apply -f deployment.yml"
            }
          }
        }
      }
    }
}
```

Now, once we build this Jenkins job. We can see that the ***Deploy To Kubernetes*** is successful.

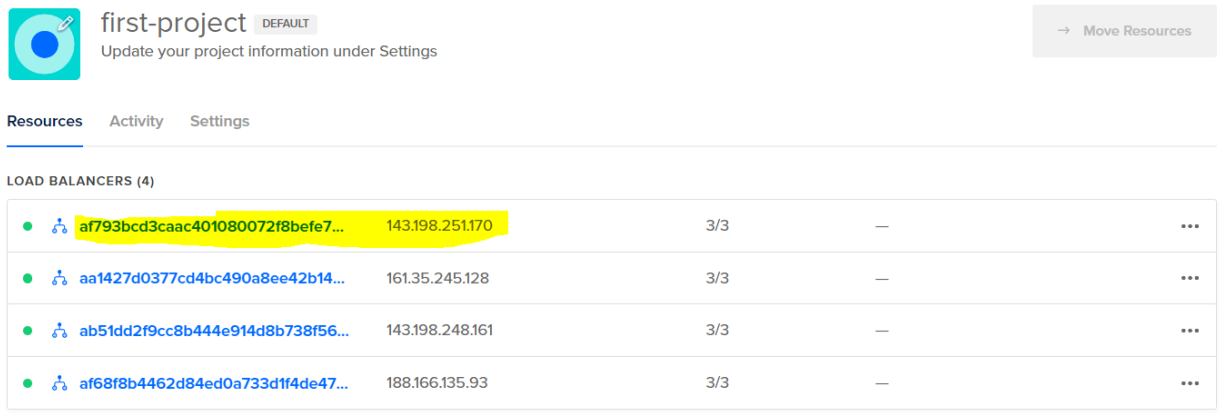



```
+ kubectl apply -f deployment.yml
deployment.apps/petclinic created
service/petclinic created
[Pipeline] }
[kubernetes-cli] kubectl configuration cleaned up
[Pipeline] // withKubeConfig
[Pipeline] }
[Pipeline] // container
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] }
[Pipeline] // podTemplate
[Pipeline] End of Pipeline
Finished: SUCCESS
```

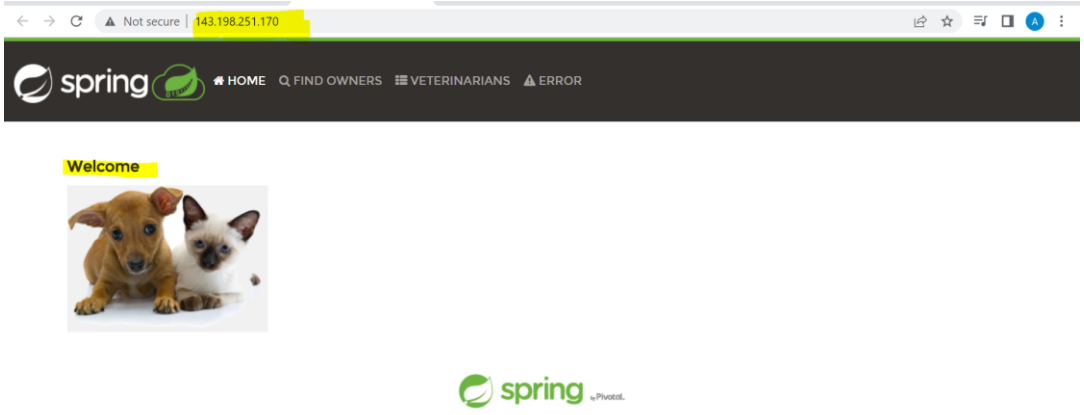
Now if you run the below command then you can observe that a new POD is created for spring-petclinic application.

```
User@DESKTOP-2QDO5T2 MINGW64 ~/Desktop
$ kubectl get po
NAME                                                READY   STATUS    RESTARTS   AGE
jenkins-0                                           2/2     Running   0          5d16h
nexus-nexus-repository-manager-6bc58b494c-65p7g     1/1     Running   0          3d16h
petclinic-9bb49576f-pjlnv                           1/1     Running   0          13m
sonarqube-postgresql-0                              1/1     Running   0          4d15h
sonarqube-sonarqube-0                               1/1     Running   0          4d15h

User@DESKTOP-2QDO5T2 MINGW64 ~/Desktop
$
```

Once, the POD is created for petclinic application, you can observe that a new LoadBalancer is also created on your DigitalOcean for your petclinic application.

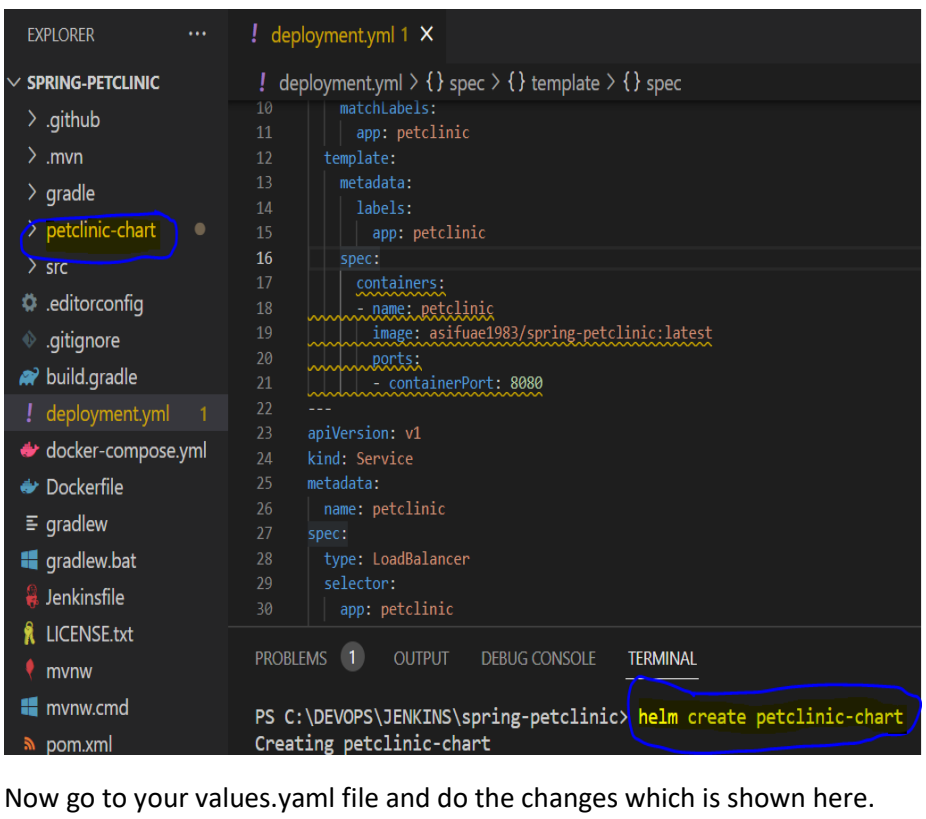


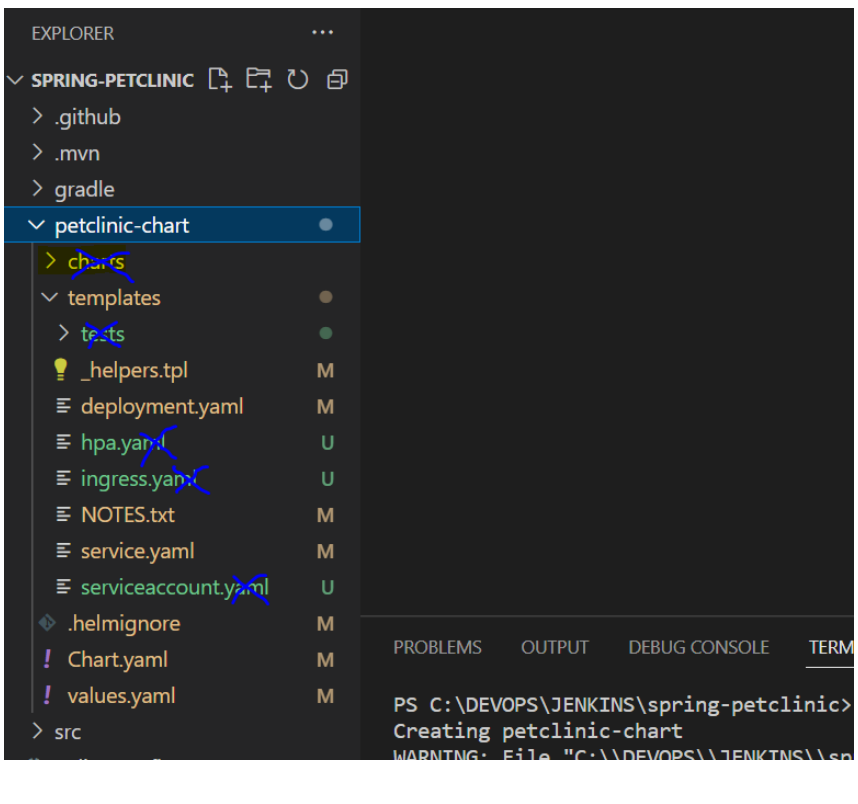
Now finally, you should see that your application is deployed successfully by opening the LoadBalancer IP on your browser.



Now, instead of deploying our application like this with a command, let's deploy our application using HELM charts. You can run the below command to create a petclinic-chart template.

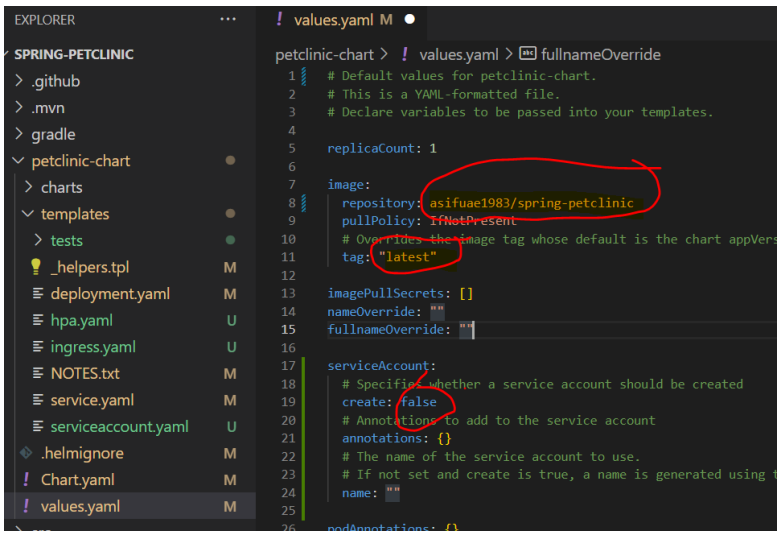
NOTE: We can remove the *charts folder, test folder, hpa.yaml, ingress.yaml* and *serviceaccount.yaml* because at present for this deployment we are using only *deployment.yaml, service.yaml* and *values.yaml* files.

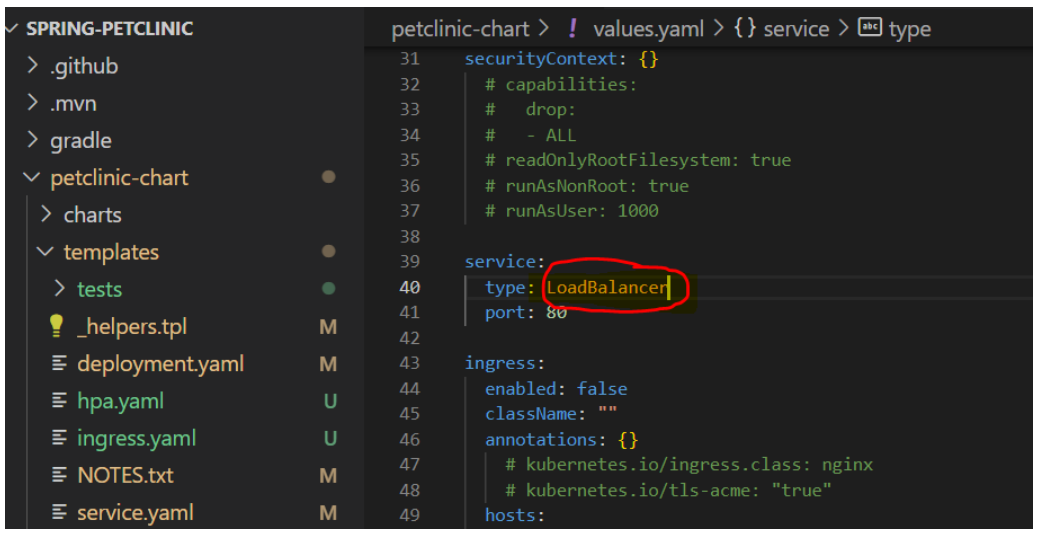




Now go to your values.yaml file and do the changes which is shown here.

NOTE: Change the value of ***repository*** from ***nginx*** to your docker image which you pushed to your DOCKER HUB (***asifuae1983/spring-petclinic***), give the ***tag*** value as ***latest*** and update the ***create*** value from ***true*** to ***false*** as we are not using ***serviceAccount*** at this point.

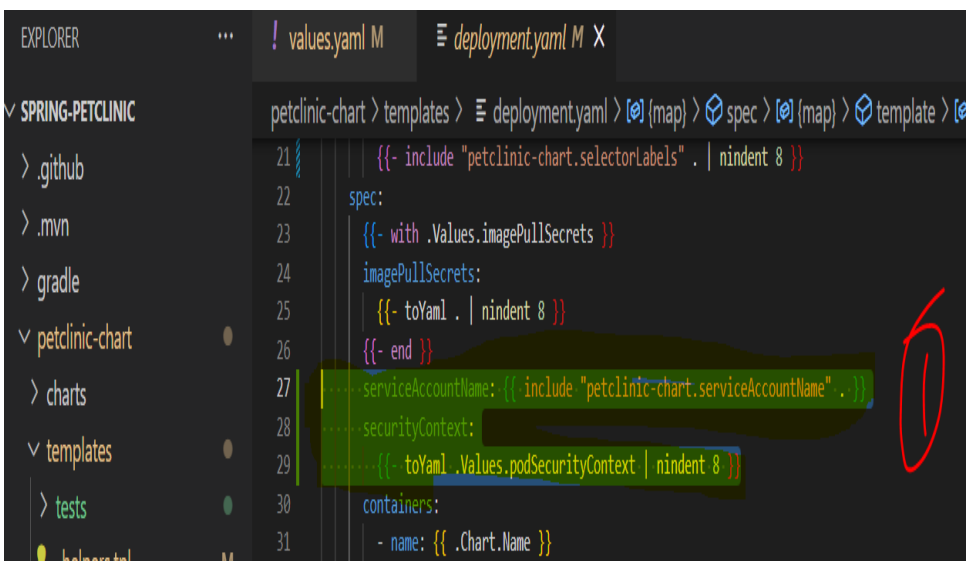




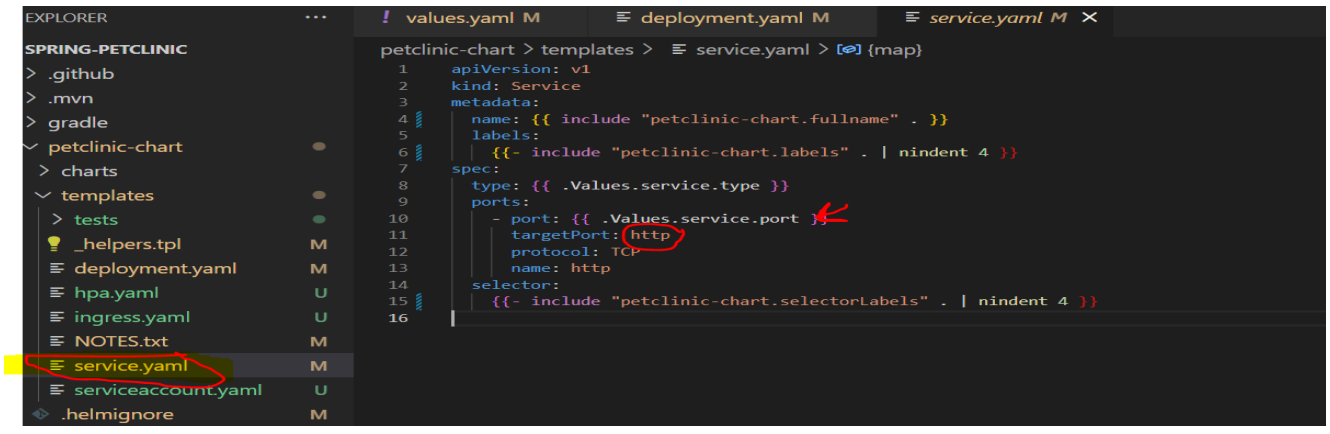
Now, from your ***deployment.yaml file*** inside the HELM chart which is ***petclinic-chart*** do the below modification.

- Remove these 3 lines which is shown in 1st diagram as we are not using ***serviceAccount*** and ***securityContext*** in this deployment.

- Remove these highlighted lines from ***LivenessProbes*** to the ***end*** as we don't need this in our deployment at this point and change the value of ***containerPort*** from 80 to 8080.

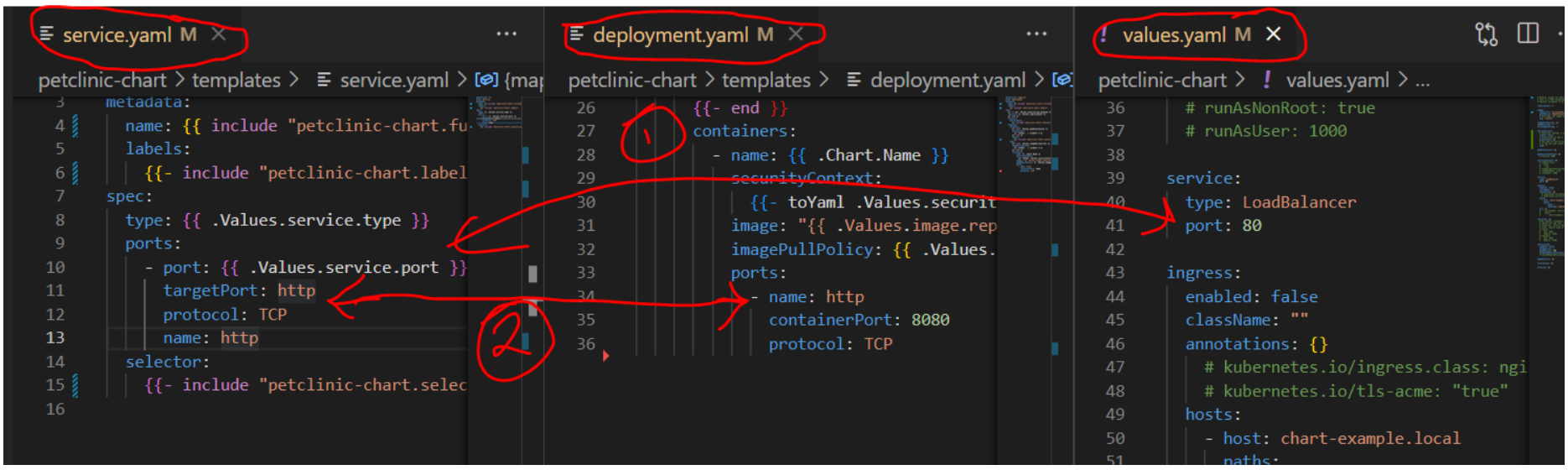


In our Service.yaml file everything looks good, so no need to do any modification.



Here, in 1st point we are showing that the port value of Service.yaml should be getting from values.yaml file which is 80 and the targetPort in Service.yaml should be matching with the ports name in deployment.yaml which is 8080.

So the ***Service.yaml*** file will get ***port*** value as ***80*** from ***values.yaml*** file and ***targetPort*** as ***8080*** from ***deployment.yaml*** file.



Now, let's add, commit and push our changes to our GitHub account with the following commands.

```
PS C:\DEVOPS\JENKINS\spring-petclinic> git add -A
```

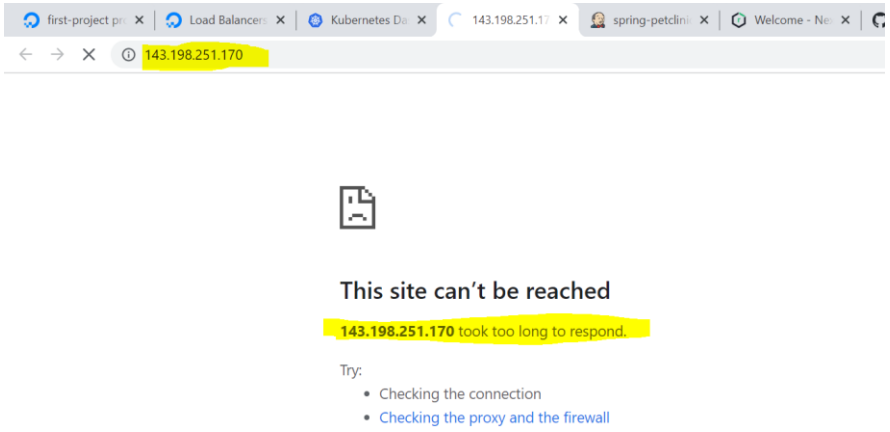
```
PS C:\DEVOPS\JENKINS\spring-petclinic> git commit -m "Added petclinic-chart file for HELM deployment"
[main 093e35c] Added petclinic-chart file for HELM deployment
 9 files changed, 149 insertions(+), 24 deletions(-)
 create mode 100644 petclinic-chart/templates/hpa.yaml
 create mode 100644 petclinic-chart/templates/ingress.yaml
 create mode 100644 petclinic-chart/templates/serviceaccount.yaml
 create mode 100644 petclinic-chart/templates/tests/test-connection.yaml
PS C:\DEVOPS\JENKINS\spring-petclinic>
```

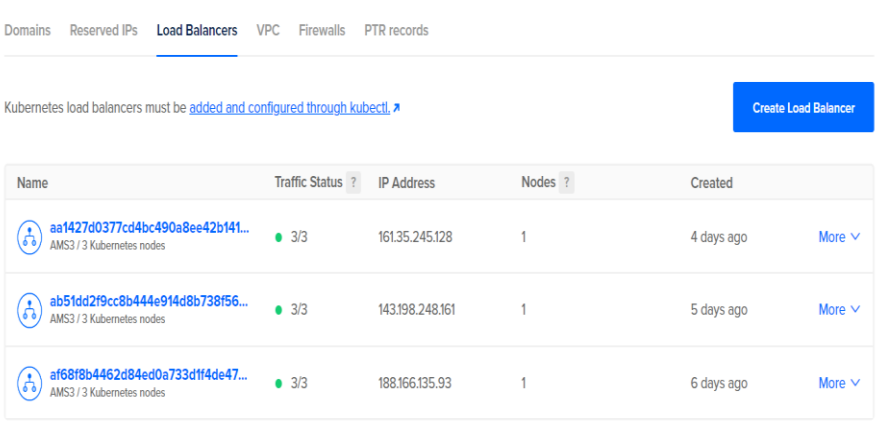
```
PS C:\DEVOPS\JENKINS\spring-petclinic> git push origin main
Enumerating objects: 22, done.
Counting objects: 100% (22/22), done.
Delta compression using up to 4 threads
Compressing objects: 100% (13/13), done.
Writing objects: 100% (14/14), 2.47 KiB | 632.00 KiB/s, done.
Total 14 (delta 7), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (7/7), completed with 7 local objects.
To https://github.com/asifqa123/spring-petclinic.git
   869457d..093e35c  main -> main
PS C:\DEVOPS\JENKINS\spring-petclinic>
```

Now, let's delete the deployment and service related to our application which we created using our Jenkins pipeline in the stage ***Deploy To Kubernetes*** as now we are planning to deploy them using our HELM charts which we created just now.

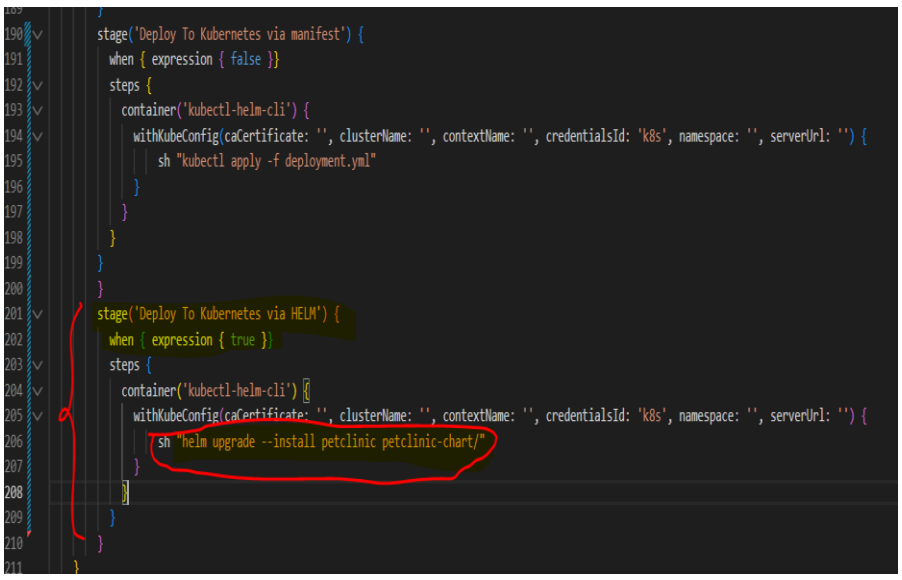
```
User@DESKTOP-2QD05T2 MINGW64 ~/Desktop
$ kubectl get deploy,svc
NAME                                          READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nexus-nexus-repository-manager   1/1     1            1           3d17h
deployment.apps/petclinic                        1/1     1            1           47h

NAME                                          TYPE           CLUSTER-IP       EXTERNAL-IP       PORT(S)          AGE
service/jenkins                               LoadBalancer   10.245.232.174   188.166.135.93    80:30276/TCP     5d18h
service/jenkins-agent                         ClusterIP      10.245.160.13    <none>            50000/TCP        5d18h
service/kubernetes                            ClusterIP      10.245.0.1       <none>            443/TCP          6d16h
service/nexus-nexus-repository-manager        LoadBalancer   10.245.225.225   161.35.245.128    8081:30754/TCP   3d17h
service/petclinic                             LoadBalancer   10.245.36.210    143.198.251.170   80:30744/TCP     47h
service/sonarqube-postgresql                  ClusterIP      10.245.69.77     <none>            5432/TCP         4d16h
service/sonarqube-postgresql-headless         ClusterIP      None             <none>            5432/TCP         4d16h
service/sonarqube-sonarqube                   LoadBalancer   10.245.64.140    143.198.248.161   9000:32413/TCP   4d16h

User@DESKTOP-2QD05T2 MINGW64 ~/Desktop
$ kubectl delete deploy/petclinic svc/petclinic
deployment.apps "petclinic" deleted
service "petclinic" deleted

User@DESKTOP-2QD05T2 MINGW64 ~/Desktop
$
```

After deleting the deployment and service related to our application you can observe that our application is down now and the related LoadBalancer is also got deleted, now only 3 LoadBalancers are there as compared to 4 earlier.





Now, let's add a new stage for deploying our application using HELM charts and run the command as highlighted, copy the complete pipeline and paste it in your Jenkins pipeline then click on Build Now.



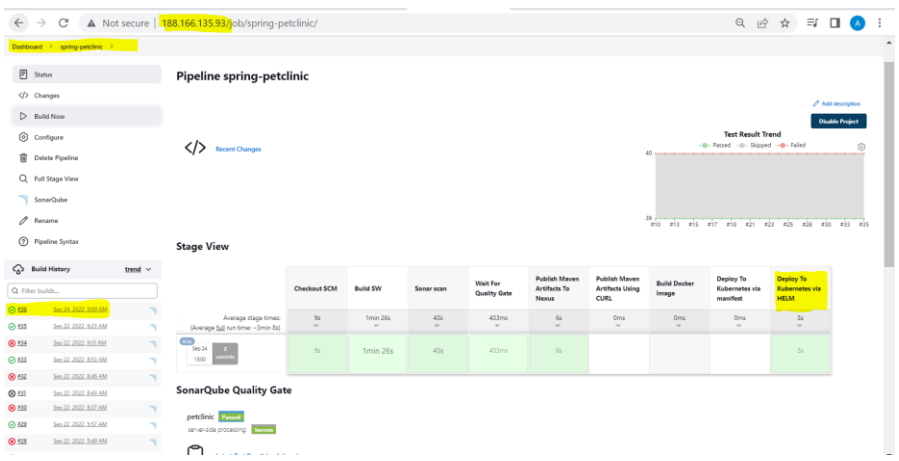
```
stage('Deploy To Kubernetes via manifest') {
  when { expression { false }}
  steps {
    container('kubectl-helm-cli') {
      withKubeConfig(caCertificate: '', clusterName: '', contextName: '', credentialsId: 'k8s', namespace: '', serverUrl: '') {
        sh "kubectl apply -f deployment.yml"
      }
    }
  }
}
stage('Deploy To Kubernetes via HELM') {
  when { expression { true }}
  steps {
    container('kubectl-helm-cli') {
      withKubeConfig(caCertificate: '', clusterName: '', contextName: '', credentialsId: 'k8s', namespace: '', serverUrl: '') {
        sh "helm upgrade --install petclinic petclinic-chart/"
      }
    }
  }
}
```

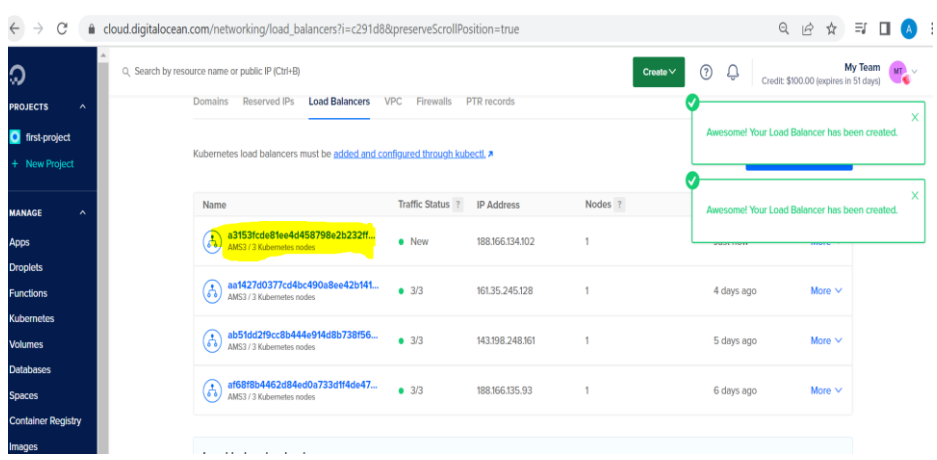
Script ?

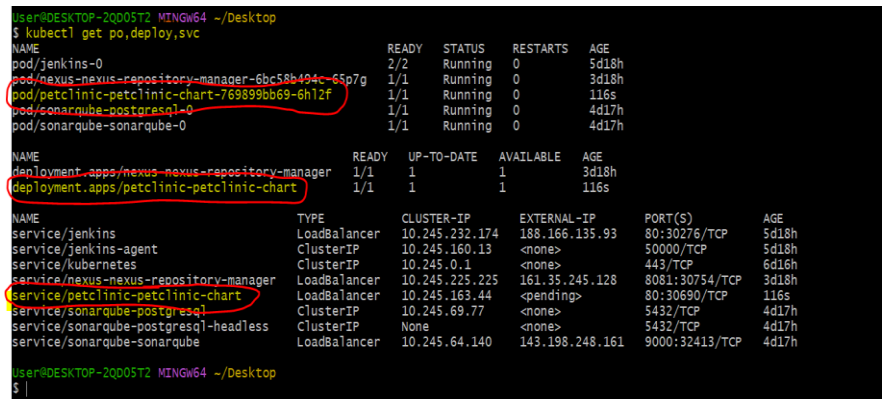
```
        sh "docker rmi $IMAGE_NAME:latest"
      }
    }
  }
  stage('Deploy To Kubernetes via manifest') {
    when { expression { false }}
    steps {
      container('kubectl-helm-cli') {
        withKubeConfig(caCertificate: '', clusterName: '', contextName: '', credentialsId: 'k8s', namespace: '', serverUrl: '') {
          sh "kubectl apply -f deployment.yml"
        }
      }
    }
  }
  stage('Deploy To Kubernetes via HELM') {
    when { expression { true }}
    steps {
      container('kubectl-helm-cli') {
        withKubeConfig(caCertificate: '', clusterName: '', contextName: '', credentialsId: 'k8s', namespace: '', serverUrl: '') {
          sh "helm upgrade --install petclinic petclinic-chart/"
        }
      }
    }
  }
}
```

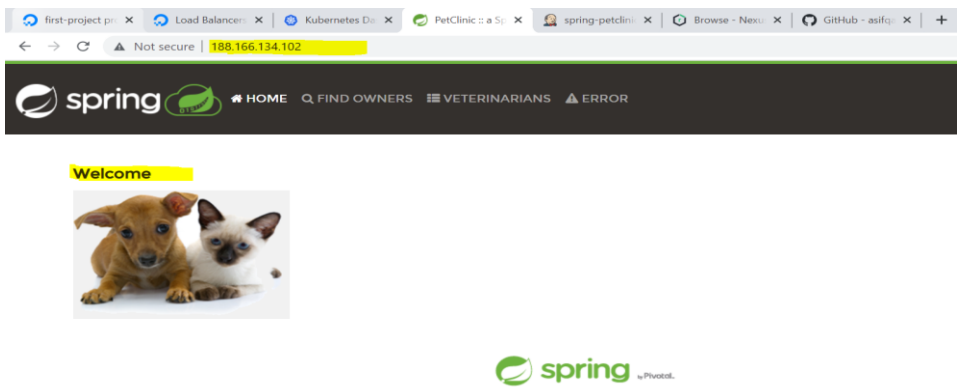
After successfully deploying the application using HELM chart, you can observe the following:

- One more LoadBalancer is created for our petclinic application.
- Pod, Deployment and Service related to our petclinic application are created.
- The petclinic application is working fine after deploying it via HELM.

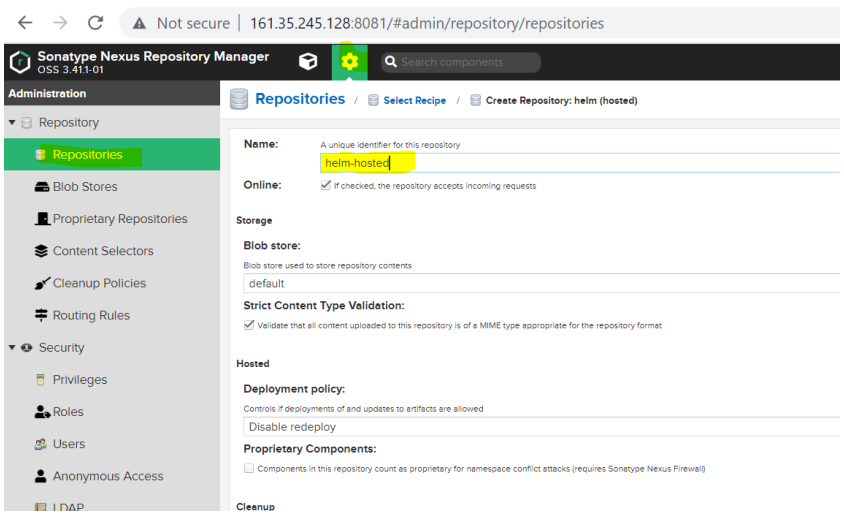








Now, let’s create a helm-hosted repository in our NEXUS and push the artifacts to our NEXUS.



Now, let’s add the helm repository with the following syntax:

Syntax: *helm repo add alias-name http://admin:password@ NexusIpAddress:8081/repository/Nexus-Repository-Name/*

```
User@DESKTOP-2QD05T2 MINGW64 ~/Desktop
$ helm repo add helm-hosted http://admin:Nexus123@161.35.245.128:8081/repository/helm-hosted/
"helm-hosted" has been added to your repositories
```

After that update your helm repo

```
User@DESKTOP-2QD05T2 MINGW64 ~/Desktop
$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "helm-hosted" chart repository
...Successfully got an update from the "sonarqube" chart repository
...Successfully got an update from the "sonatype" chart repository
...Successfully got an update from the "traefik" chart repository
...Successfully got an update from the "jenkins" chart repository
Update Complete. ⎈Happy Helming!⎈
```

Now, package your chart using the below command. You can observe that it created one tgz file in your project location.