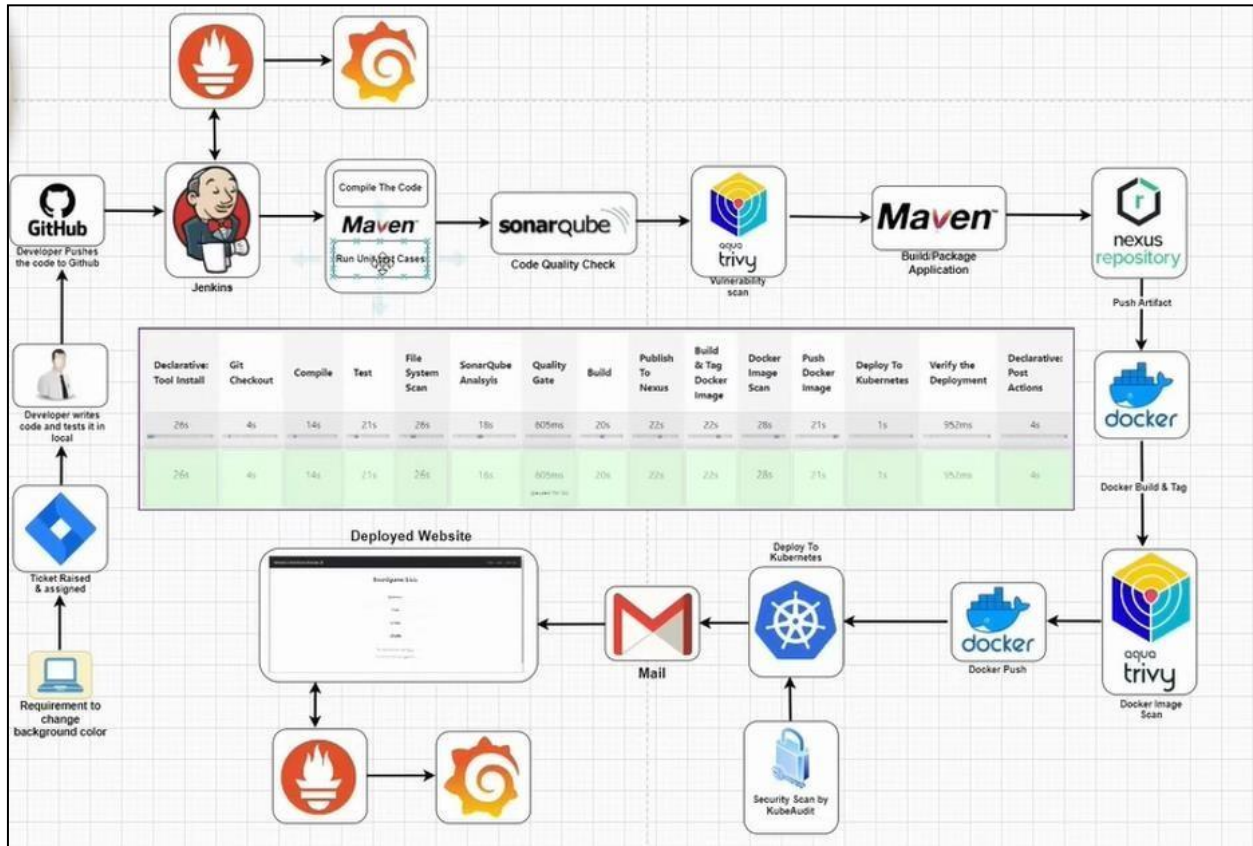


Boardgame Project



Introduction:

1. Client Request:

- The client requests a new feature for the application, such as changing the background color.
- They create a Jira ticket detailing the changes or new features required.

2. Development Stage:

- The assigned developer writes the source code.
- The developer tests the code on their local machine.

3. Code Integration:

- Once the code runs successfully, it's pushed to the GitHub repository containing the application's source code.

4. Pipeline Setup:

- As a DevOps engineer, the next step is setting up the pipeline using Jenkins.

The pipeline consists of multiple stages:

- **Compilation:** Checking for syntax errors.
- **Unit Testing:** Ensuring functionality.
- **SonarQube Code Quality Check:** Identifying bugs, code smells, or vulnerabilities.
- **Vulnerability Scan:** Checking for sensitive data and dependency vulnerabilities.
- **Report Generation:** Creating a formatted report for analysis.
- **Artifact Packaging:** Building and publishing the application artifact to Nexus repository.
- **Docker Image Creation:** Building and tagging Docker images.
- **Docker Image Scan:** Using Trivy to scan for container vulnerabilities.
- **Docker Image Deployment:** Pushing the image to Docker Hub repository.
- **Kubernetes Cluster Security:** Scanning using kube audit tool for cluster issues.
- **Application Deployment:** Deploying the application to the Kubernetes cluster.
- **Mail Notification:** Receiving notifications on pipeline success or failure.

5. Monitoring:

- **Website Level Monitoring:** Using a blackbox exporter to monitor website traffic and status.
- **System Level Monitoring:** Monitoring CPU and RAM usage, for example, using node exporter for Jenkins.

6. Documentation and Resources:

- All scripts, commands, and documentation are shared in a Telegram group for reference and learning purposes.

Phases for the Project:

1. Phase One: Setting Up Infrastructure

- Establish a separate network environment for privacy and security. Setting up **VPC**.
- Deploy a Kubernetes or similar cluster for application deployment. Setup Kubernetes **Master & Slave**.
- Set up servers (e.g., SonarQube, Nexus) and tools (e.g., Jenkins).
- Install monitoring tools for application monitoring.

2. Phase Two: Source Code Management

- Create a private Git repository for source code management.
- Push the source code to the repository.
- Ensure visibility of the pushed code in the repository.

3. Phase Three: CI/CD Pipeline Setup

- Implement CI/CD pipeline adhering to best practices and security measures.
- Configure mail notifications for pipeline status (success or failure).
- Perform vulnerability scanning of the Kubernetes cluster.

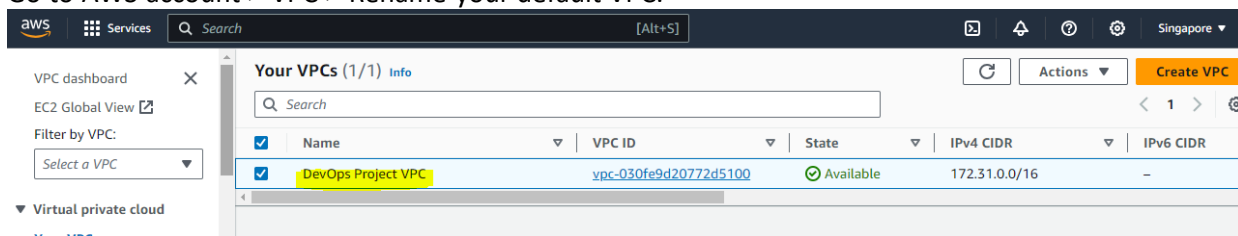
4. Phase Four: Monitoring

- Set up monitoring tools to monitor the application.
- Monitor systemlevel metrics such as CPU and RAM.
- Monitor websitelevel metrics such as traffic.

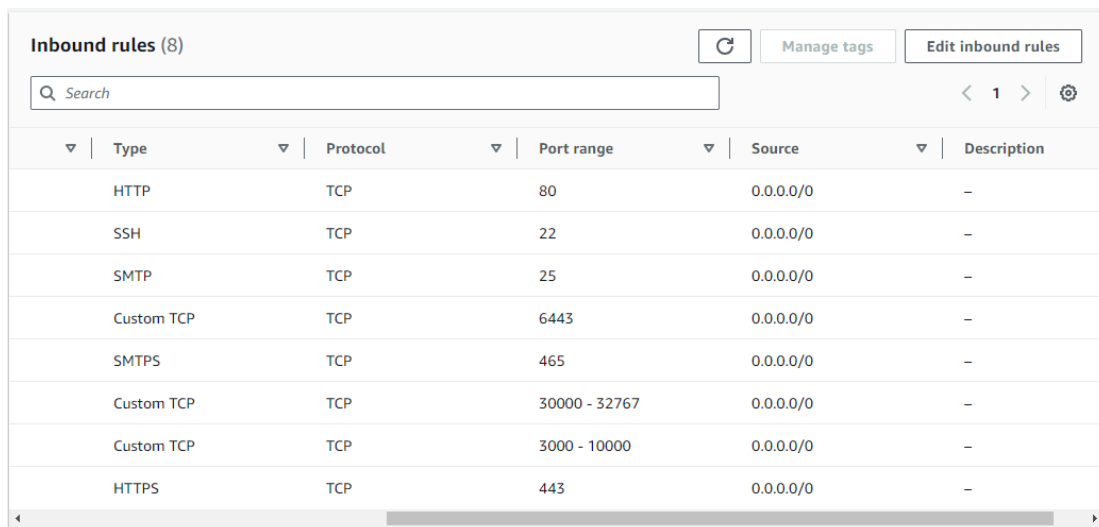
This structured approach ensures a systematic setup of infrastructure, source code management, CI/CD pipeline, and monitoring for effective DevOps implementation.

Phase 1:

- Go to AWS account > VPC > Rename your default VPC.



- Go to EC2 > Security Group > Add the below ports in the inbound rule.



Type	Protocol	Port range	Source	Description
HTTP	TCP	80	0.0.0.0/0	-
SSH	TCP	22	0.0.0.0/0	-
SMTP	TCP	25	0.0.0.0/0	-
Custom TCP	TCP	6443	0.0.0.0/0	-
SMTPS	TCP	465	0.0.0.0/0	-
Custom TCP	TCP	30000 - 32767	0.0.0.0/0	-
Custom TCP	TCP	3000 - 10000	0.0.0.0/0	-
HTTPS	TCP	443	0.0.0.0/0	-

Here are the ports that need to be opened in the Security Group:

1. Ports 30,000 to 32,767:

- Used for deploying applications when using virtual machines as a Kubernetes cluster.

2. Port 465:

- Used for sending mail notifications from Jenkins pipeline to Gmail addresses.

3. Port 6443:

- Required during the setup of the Kubernetes cluster.

4. Port 22 (SSH):

- Used for accessing virtual machines.

5. Ports 443 (HTTPS) and 80 (HTTP):

- Used for web traffic.

6. Ports 3,000 to 10,000:

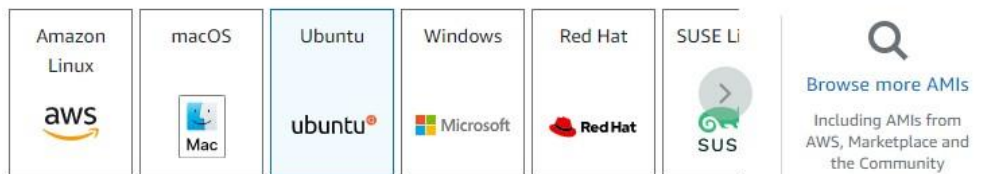
- Range for deploying applications.

7. Port 25 (SMTP):

- Typically used for SMTP servers for sending email notifications over Gmail, but not utilized in this context.

Ensure these ports are opened in the Security Group for the necessary functionalities in the infrastructure setup.

Now create the 3 EC2 instances.



Amazon Machine Image (AMI)

Ubuntu Server 20.04 LTS (HVM), SSD Volume Type Free tier eligible

ami-08e4b984abde34a4f (64-bit (x86)) / ami-09e5beccfea17b5ca (64-bit (Arm))

Virtualization: hvm ENA enabled: true Root device type: ebs

▼ Instance type [Info](#) | [Get advice](#)

Instance type

t2.medium

Family: t2 2 vCPU 4 GiB Memory Current generation: true

On-Demand RHEL base pricing: 0.1184 USD per Hour

On-Demand Windows base pricing: 0.0764 USD per Hour

On-Demand SUSE base pricing: 0.1584 USD per Hour

On-Demand Linux base pricing: 0.0584 USD per Hour

[Additional costs apply for AMIs with pre-installed software](#)

Select the security group which we have updated storage extends to 25 GB.

Firewall (security groups) | [Info](#)
A security group is a set of firewall rules that control the traffic for your instance. Add rules to allow specific traffic to and from your instance.

☐ Create security group ☒ **Select existing security group**

Common security groups [Info](#)

Select security groups

launch-wizard-1_sg-0c6eaf4da8221eebf ✕
VPC: vpc-030fe9d20772d5100

Security groups that you add or remove here will be added to or removed from all your network interfaces.

▼ **Configure storage** [Info](#)

1x GiB Root volume (Not encrypted)

[?](#) Free tier eligible customers can get up to 30 GB of EBS General Purpose (SSD) or Magnetic storage.

Instances (1/3) [Info](#) [Refresh](#) [Connect](#) [Instance state ▼](#) [Actions ▼](#) [Launch instances](#)

[Running ▼](#) [<](#)

<input type="checkbox"/>	Name ✎ ▼	Instance ID	Instance state ▼	Instance type ▼	Status check	Alarm status
<input type="checkbox"/>	Master	i-04b5de34dc06958b2	Running 🔍 🔍	t2.medium	Initializing	View alarms +
<input type="checkbox"/>	Slave1	i-0a55be8872f91f9d1	Running 🔍 🔍	t2.medium	Initializing	View alarms +
<input checked="" type="checkbox"/>	Slave2 ✎	i-0736c181b9739587b	Running 🔍 🔍	t2.medium	Initializing	View alarms +

Kubernetes Setup:

Once all 3 instances are created, open the instances via MobaXterm.

Setup K8-Cluster using kubeadm [K8 Version-->1.28.1]

1. Update System Packages [On Master & Worker Node]

```
sudo apt-get update
```

2. Install Docker[On Master & Worker Node]

```
sudo apt install docker.io -y  
sudo chmod 666 /var/run/docker.sock
```

3. Install Required Dependencies for Kubernetes[On Master & Worker Node]

```
sudo apt-get install -y apt-transport-https ca-certificates curl gnupg
sudo mkdir -p -m 755 /etc/apt/keyrings
```

4. Add Kubernetes Repository and GPG Key[On Master & Worker Node]

```
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.28/deb/Release.key | sudo gpg --dearmor -o
/etc/apt/keyrings/kubernetes-apt-keyring.gpg
echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg]
https://pkgs.k8s.io/core:/stable:/v1.28/deb/ /' | sudo tee /etc/apt/sources.list.d/kubernetes.list
```

5. Update Package List[On Master & Worker Node]

```
sudo apt update
```

6. Install Kubernetes Components[On Master & Worker Node]

```
sudo apt install -y kubeadm=1.28.1-1.1 kubelet=1.28.1-1.1 kubectl=1.28.1-1.1
```

7. Initialize Kubernetes Master Node [On MasterNode]

```
sudo kubeadm init --pod-network-cidr=10.244.0.0/16
```

Here we're defining a range here for deployment purposes. This range encompasses over 65,000 addresses available for deployment. Depending on your requirements, you can adjust the network range accordingly.

This command will facilitate the connection between the master node and the slave nodes (Slave 1 and Slave 2), effectively configuring them as Kubernetes worker nodes. Subsequently, our deployments will be executed on these worker nodes.

```
export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 172.31.34.119:6443 --token 8g0jbe.u21ha0fk8k239d8h \
--discovery-token-ca-cert-hash sha256:831968c5d8981cc7ec90c075b930eebae33b57454a2c2334c75a8e62d2e35e83
ubuntu@ip-172-31-34-119:~$
```

This command generates a token that we'll use to join the worker nodes to the Kubernetes cluster. The command starts with "kubeadm join" which needs to be executed on each worker node (**Slave 1 and Slave 2**). These nodes are virtual machines that we want to add as Kubernetes worker nodes, connected to the master node.

Copy this kubeadm join command and execute it on slave 1 and slave 2. Once you execute this command on slave you will see the slave is connected to master.

```
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiservert and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.

root@ip-172-31-45-124:/home/ubuntu#
```

8. Configure Kubernetes Cluster [On MasterNode]

```
mkdir -p $HOME/.kube
```

We're creating a folder to store a configuration file for Kubernetes. This configuration file, typically named "config," holds all the essential information about the Kubernetes cluster. It includes details such as cluster settings, authentication parameters, and connection endpoints. This file is crucial for managing and interacting with the Kubernetes cluster effectively.

```
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

9. Deploy Networking Solution (Calico) [On MasterNode]

```
kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
```

10. Deploy Ingress Controller (NGINX) [On MasterNode]

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v0.49.0/deploy/static/provider/baremetal/deploy.yaml
```

These YAML files contain network-related configurations required for managing the network aspects of the Kubernetes cluster. They specify settings and parameters for networking components within the cluster, such as networking policies, service networking, and network plugins. Executing these YAML files will apply the specified configurations to the Kubernetes cluster, ensuring proper network management and communication between cluster components.

To streamline the process, instead of running multiple commands individually, I'll group them and execute them together. I'll create a shell script file named "1.sh" and paste all the commands into it. Then, I'll save the file and ensure it's executable by changing its permissions using the "chmod +x" command. Once done, I can execute the script file by typing "./filename" and pressing enter, which will run all the commands sequentially.

This approach saves time and effort, especially when dealing with multiple commands.

Create these files in Master and Worker nodes.

vi 1.sh --#Copy the steps 1 - 4.

```
sudo apt-get update
sudo apt install docker.io -y
sudo chmod 666 /var/run/docker.sock
sudo apt-get install -y apt-transport-https ca-certificates curl gnupg
sudo mkdir -p -m 755 /etc/apt/keyrings
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.28/deb/Release.key | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.28/deb/ /' | sudo tee /etc/apt/sources.list.d/kubernetes.list
```

chmod +x 1.sh

./1.sh

sudo apt update --#Execute on all 3 machines.

Once this is done next we are going to install tools **Kubeadm, Kubelet, Kubectl**.

- Kubeadm is going to set up the Kubernetes cluster.
- Kubelet will be responsible for creating the Pods to which we are going to deploy applications.
- Kubectl will be the one working as a CLI for us to interact with the Kubernetes cluster.

- Execute **Step 6** Command on all 3 machines.

- Execute **Step 7 - Step 10** on Master node only.

Now, once all these steps are done, check on master node if the worker nodes are visible or not.

kubectl get nodes

```
root@ip-172-31-34-119:/home/ubuntu# kubectl get nodes
NAME                 STATUS    ROLES    AGE   VERSION
ip-172-31-34-119     Ready    control-plane   17m   v1.28.1
ip-172-31-41-203     Ready    <none>         11m   v1.28.1
ip-172-31-45-124     Ready    <none>         11m   v1.28.1
root@ip-172-31-34-119:/home/ubuntu#
```

After setting up your Kubernetes cluster, it's essential to conduct a security scan to identify any potential issues. One tool for this purpose is **KubeAudit**. Although there are other tools like Trivy, but KubeAudit is chosen for its reliability.

To begin, you'll navigate to the **KubeAudit** releases and download the appropriate version for your system architecture, such as **AMD 64**. After downloading, you'll extract the files and move the **KubeAudit** executable to a designated location.

[On Master]

wget

https://github.com/Shopify/kubeaudit/releases/download/v0.22.1/kubeaudit_0.22.1_linux_amd64.tar.gz

tar -xvzf kubeaudit_0.22.1_linux_amd64.tar.gz

sudo mv kubeaudit /usr/local/bin/

Once the KubeAudit executable is in place, you can run the "kubeaudit all" command. This command initiates a comprehensive scan of the entire Kubernetes cluster, detecting various errors and vulnerabilities.

kubeaudit all

The generated report provides valuable insights that can be analyzed by the infrastructure team. It's important to note that the initial scan may reveal numerous issues, especially if certain security measures like role-based access control (RBAC) and service accounts haven't been set up yet.

By using KubeAudit for security scans, you can proactively address potential vulnerabilities and ensure the overall security of your Kubernetes cluster.

Till now, we have setup the Kubernetes. Now let's setup other tools like, SonarQube & Nexus.

Use the same configuration that we used in Master and Slave of Kubernetes.

<input checked="" type="checkbox"/>	SonarQube	i-0609d348a0a2082b4	Running	t2.medium	2/2 checks passed
<input type="checkbox"/>	Nexus	i-0a245e01b7204bcaf	Running	t2.medium	2/2 checks passed

Now create Jenkins server. For this instance, take bigger resources.

Amazon Linux

macOS

Ubuntu

Windows

Red Hat

SUSE Linux

Browse more AMIs
Including AMIs from AWS, Marketplace and the Community

Amazon Machine Image (AMI)
Ubuntu Server 20.04 LTS (HVM), SSD Volume Type
ami-08e4b984abde34a4f (64-bit (x86)) / ami-09e5beccfca17b5ca (64-bit (Arm))
Virtualization: hvm ENA enabled: true Root device type: ebs
Free tier eligible

▼ Instance type Info | Get advice

Instance type
t2.large
Family: t2 2 vCPU 8 GiB Memory Current generation: true
On-Demand RHEL base pricing: 0.1768 USD per Hour
On-Demand Windows base pricing: 0.1448 USD per Hour
On-Demand Linux base pricing: 0.1168 USD per Hour
On-Demand SUSE base pricing: 0.2168 USD per Hour
Additional costs apply for AMIs with pre-installed software

▼ **Configure storage** [Info](#)

1x GiB Root volume (Not encrypted)

[i](#) Free tier eligible customers can get up to 30 GB of EBS General Purpose (SSD) or t

Now, we have created 3 new instances i.e. SonarQube,

<input type="checkbox"/>	Jenkins	i-051b2477a280e789b	Running	t2.large	2/2 checks
<input type="checkbox"/>	SonarQube	i-0609d348a0a2082b4	Running	t2.medium	2/2 checks
<input type="checkbox"/>	Nexus	i-0a245e01b7204bcaf	Running	t2.medium	2/2 checks

Setup SonarQube & Nexus:

Install **Docker** in both SonarQube & Nexus server.

vi dock.sh

Add Docker's official GPG key:

sudo apt-get update

sudo apt-get install ca-certificates curl

sudo install -m 0755 -d /etc/apt/keyrings

sudo curl -fsSL <https://download.docker.com/linux/ubuntu/gpg> -o /etc/apt/keyrings/docker.asc

sudo chmod a+r /etc/apt/keyrings/docker.asc

Add the repository to Apt sources:

**echo **

"deb [arch=\$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]

**<https://download.docker.com/linux/ubuntu> **

**\$(. /etc/os-release && echo "\$VERSION_CODENAME") stable" | **

sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/ubuntu \
$(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

chmod +x dock.sh

./dock.sh

sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin

Once Docker is installed, it's important to ensure that all users have permissions to execute Docker commands, not just the root user. To achieve this, a specific command needs to be executed to grant permissions to other users.

chmod 666 /var/run/docker.sock

The next step is creating Docker containers for SonarQube and Nexus. This is accomplished by running the "docker run" command with various parameters, such as specifying the container name, ports, and Docker image name. Once the containers are created, Docker PS can be used to verify that they are running.

[On SonarQube server]

docker run -d --name sonar -p 9000:9000 sonarqube:lts-community

docker run: This is the command to run a Docker container.

-d: This flag specifies that the container should run in detached mode, meaning it runs in the background.

--name sonar: This option assigns the name "sonar" to the container.

-p 9000:9000: This option maps port 9000 on the host machine to port 9000 in the container. This is necessary to access the SonarQube web interface.

sonarqube:lts-community: This is the Docker image used to create the container. In this case, it's pulling the SonarQube Community Edition with LTS (Long-Term Support) tag.

docker ps

```
root@ip-172-31-35-125:/home/ubuntu# docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED        STATUS        PORTS
2b7f4d7c2070   sonarqube:lts-commu... "/opt/sonarqube/dock... 57 seconds ago Up 51 seconds 0.0.0.0:9000->9000/tcp, :::9000->9000/
tcp_sonar
root@ip-172-31-35-125:/home/ubuntu#
```

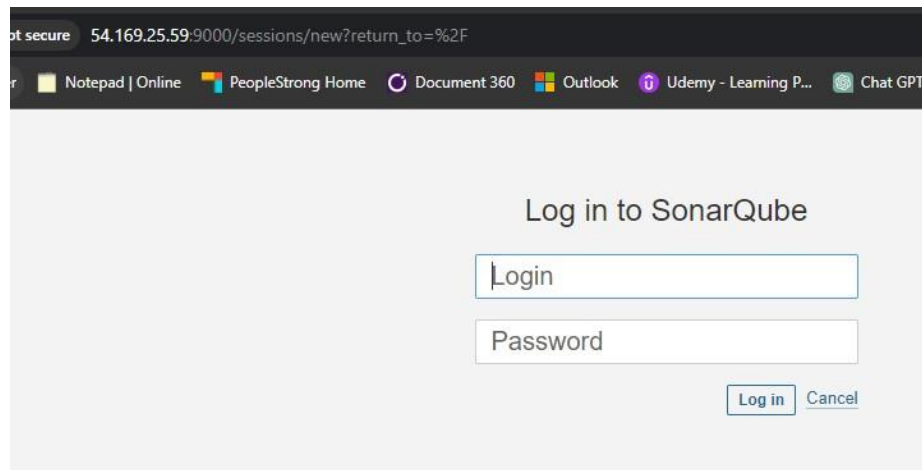
[On Nexus server]

docker run -d --name Nexus -p 8081:8081 sonatype/nexus3

Now to access the **SonarQube**, go to browser and type,

<Public-IP of SonarQube server>:9000

Ex- 54.169.25.59:9000



Default Username: admin

Password: admin

Change the password after login.

Now to access the **Nexus**, go to browser and type,

<Public-IP of Nexus server>:8081

Ex: 13.212.205.240:8081

Default Username: admin

Password: For password, get inside the Nexus container, since the Nexus application is running inside the container.

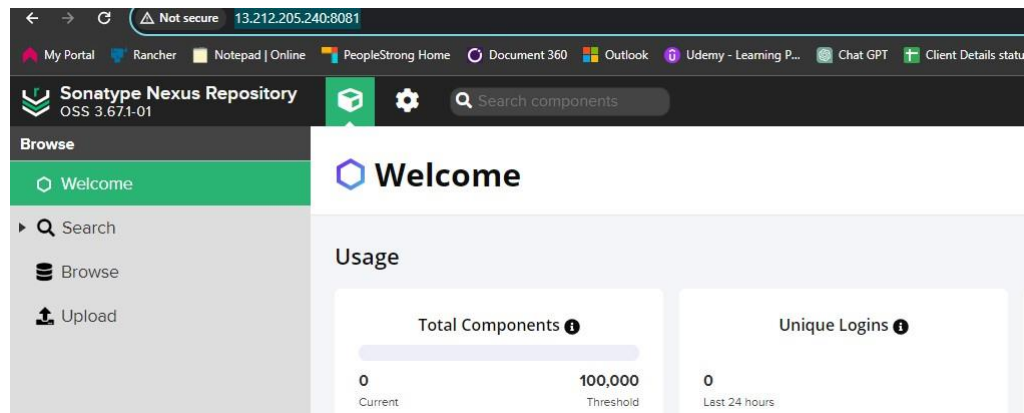
docker exec -it <Container-ID> /bin/bash

Ex: docker exec -it 1d5ae49f82d8 /bin/bash

cd sonatype-work/nexus3

cat admin.password --#You will see the password, just copy the password before **bash-4.4\$** since this is the username.

```
bash-4.4$ cat admin.password
9d3b2989-b54b-4218-8006-5dfae5462b5fbash-4.4$
```



Setup Jenkins:

With the SonarQube and Nexus containers set up, the next step is to install Jenkins. The installation process involves installing Java, as Jenkins requires it to run, and then running the installation commands provided by the official Jenkins documentation. Once Jenkins is installed, the initial password is retrieved to complete the setup process.

vi jen.sh

```
sudo apt install openjdk-17-jre-headless -y
```

```
sudo wget -O /usr/share/keyrings/jenkins-keyring.asc \
https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key
echo "deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] \
https://pkg.jenkins.io/debian-stable/binary/ | sudo tee \
/etc/apt/sources.list.d/jenkins.list > /dev/null
sudo apt-get update
sudo apt-get install jenkins -y
```

```
chmod +x jen.sh
./jen.sh
```

Now install docker.

vi dock.sh

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc
```

```
# Add the repository to Apt sources:
echo \
```

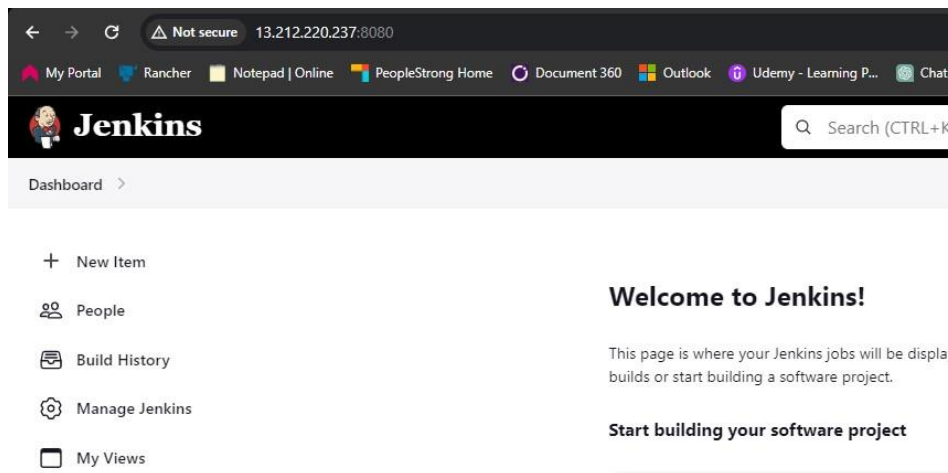
```
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/ubuntu \
$(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

```
chmod +x dock.sh
./dock.sh
```

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
y
chmod 666 /var/run/docker.sock
sudo usermod -aG docker jenkins
```

Now to access **Jenkins**.
<Public-IP of Jenkins server>:8080
Ex: 13.212.220.237:8080

Get the password from
`cat /var/lib/jenkins/secrets/initialAdminPassword`



Till here Phase 1 is completed. Now starting Phase 2.

Phase 2:

Create a private Git repository for source code management.

Go to Git hub account > Repository > New Repository.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk ().*

Owner * amanduggal001 / Repository name * Boardgame

✔ Boardgame is available.

Great repository names are short and memorable. Need inspiration? How about **redesigned-umbrella** ?

Description (optional)

☐ **Public**
Anyone on the internet can see this repository. You choose who can commit.

☒ **Private**
You choose who can see and commit to this repository.

Now clone this private repository to the local and push the source code to this repository.

Get the source code:

git clone https://github.com/amanduggal001/aditya_repo_boardgame.git

Create one folder in local system and clone the our private repository.

git clone <https://github.com/amanduggal001/Boardgame.git>

Now copy the source code and paste it in the Boardgame folder in local system.

The screenshot shows a Windows File Explorer window with the address bar set to 'DevopsProject > Boardgame > Boardgame'. The window displays a list of files and folders. The columns are 'Name', 'Date modified', 'Type', and 'Size'.

Name	Date modified	Type	Size
.github	4/13/2024 1:04 AM	File folder	
.mvn	4/13/2024 1:04 AM	File folder	
src	4/13/2024 1:04 AM	File folder	
.gitignore	4/13/2024 12:53 AM	Text Document	1 KB
deployment-service	4/13/2024 12:53 AM	Yaml Source File	2 KB
Dockerfile	4/13/2024 12:53 AM	File	1 KB
mvnw	4/13/2024 12:53 AM	File	11 KB
mvnw	4/13/2024 12:53 AM	Windows Comma...	7 KB
pom	4/13/2024 12:53 AM	Microsoft Edge H...	4 KB
README	4/13/2024 12:53 AM	Markdown Source...	3 KB
sonar-project	4/13/2024 12:53 AM	Properties Source ...	1 KB

Now, open the git bash and push the code in our private repository.

```
git status
git add .
git commit -m "New source code"
git push origin main
```

Till here Phase 2 is completed.

In **Phase 3** where we are going to start with CD pipelines and everything, so before configuring Jenkins and before we start writing the pipelines we need to install certain plugins that will be required.

Manage Jenkins > Plugins > Available Plugins

Install these 3 plugins.

- **Eclipse Temurin installer** - When you have multiple Jenkins jobs which require different versions of JDK then you can use this plugin to set up multiple versions of JDK for usage.
- **Config File Provider** - It allows users to store configuration files centrally and use them in Jenkins jobs. It helps manage settings, credentials, and other parameters needed for job execution.
- **Pipeline Maven Integration** - This plugin provides integration with Pipeline, configures maven environment to use within a pipeline job by calling sh mvn or bat mvn.
- **Maven Integration.**
- **SonarQube Scanner.**
- **Docker.**
- **Docker Pipeline.**
- **Kubernetes Credentials.**
- **Kubernetes.**
- **Kubernetes CLI.**

Note:

SonarQube Scanner and SonarQube Server are two distinct components of the SonarQube platform.

1. SonarQube Scanner: It is the tool that performs code analysis on projects and generates reports based on predefined rules and metrics. It scans the source code, identifies issues, measures code quality, and produces detailed reports highlighting areas for improvement, and publishes the report on the SonarQube server.

2. SonarQube Server: This is the central component of the SonarQube platform where analysis results are stored, managed, and displayed.

Now plugins are installed and now we are going to configure it.

Manage Jenkins > Tools

JDK installations

Add JDK

≡

JDK

Name

jdk17

☒

Install automatically ?

≡

Install from adoptium.net ?

Version ?

jdk-17.0.9+9 ▼

SonarQube Scanner installations

Add SonarQube Scanner

≡

SonarQube Scanner

Name

sonar-scanner

☒

Install automatically ?

≡

Install from Maven Central

Version

SonarQube Scanner 5.0.1.3006

Maven installations

Add Maven

≡

Maven

Name

maven3

☒ Install automatically ?

≡

Install from Apache

Version

3.6.1

Docker installations

Add Docker

≡

Docker

Name

docker

☒ Install automatically ?

≡

Download from docker.com

Docker version ?

latest

Add Installer ▾

Apply > Save

Till now, we configured the plugins and now setting up the credentials.

Now, Setup Credentials:

Manage Jenkins > Credentials > System > Add Credentials

- **Create GitHub credentials in Jenkins.**

Take Password of thee GitHub by generating token.

Dashboard > Manage Jenkins > Credentials > System > Global credentials (unrestricted) >

New credentials

Kind

Username with password

Scope ?

Global (Jenkins, nodes, items, all child items, etc)

Username ?

amanduggal001

☐ Treat username as secret ?

Password ?

.....

ID ?

git-credentials

Description ?

git-credentials

Create

- **Creating the SonarQube credentials on Jenkins.**

- Before creating the cred on Jenkins, go to the **SonarQube > Security > Tokens.**
- Generate the token and copy it.

sonarqube Projects Issues Rules Quality Profiles Quality Gates Administration ? Search for projects...

A Administrator Profile Security Notifications Projects

Tokens

If you want to enforce security by not providing credentials of a real SonarQube user to run your code scan or to invoke web services, you can provide a User Token as a replacement of the user login. This will increase the security of your installation by not letting your analysis user's password going through your network.

Generate Tokens

Name	Type	Expires in	
Enter Token Name	Select Token Type	30 days	Generate

Message: New token "sonar-cred" has been created. Make sure you copy it now, you won't be able to see it again!

Copy `sqa_aa9f90d12cb1d95ada85a7f4a0fc3963acd86299`

- Now in Jenkins, create a credentials using the secret text.

Dashboard > Manage Jenkins > Credentials > System > Global credentials (unrestricted) >

New credentials

Kind

Secret text

Scope ?

Global (Jenkins, nodes, items, all child items, etc)

Secret

.....

ID ?

sonar-cred

Description ?

sonar-cred

Create

Configure SonarQube Server in Jenkins.

Manage Jenkins > System > SonarQube Servers.

Dashboard > Manage Jenkins > System >

SonarQube servers

If checked, job administrators will be able to inject a SonarQube server configuration as environment variables in the build.

☐ Environment variables

SonarQube installations

List of SonarQube installations

Name

sonar

Server URL

Default is http://localhost:9000

http://18.143.156.216:9000

Server authentication token

SonarQube authentication token. Mandatory when anonymous access is disabled.

sonar-cred

+ Add

Next we are going to create the pipeline.

Dashboard > New Item > Select Pipeline Job > Name = Boardgame

As a best practice, we keep only 2 builds during the execution and old builds will be discarded, so that it do not occupy too much space.

☒ Discard old builds ?

Strategy

Log Rotation

Days to keep builds

if not empty, build records are only kept up to this number of days

Max # of builds to keep

if not empty, only up to this number of build records are kept

2

Advanced ▾

Pipeline Script:

Whenever you install a plugin for Jenkins and you want to use it in your pipeline, you need to define it inside your pipeline in one way or another. So, for doing that, what we can do is write tools in this format. I'm going to define two tools here: Java and Maven.

```
pipeline {
  agent any

  tools {
    jdk 'jdk17'
    maven 'maven3'
  }
}
```

- **First stage** is getting connected with the GitHub account where our source code is present. Use the pipeline syntax option for generating the script.

```
stages {
  stage('Git Checkout') {
    steps {
      git branch: 'main', credentialsId: 'git-cred', url:
'https://github.com/amanduggal001/Boardgame.git'
    }
  }
}
```

- In the **second stage** that we want to perform is to compile this source code. Now, compilation is done to find out if there are any syntax-based errors in our source code, right?

```

stage('Compile') {
    steps {
        sh "mvn compile"
    }
}

```

- In the **third stage**, we want to test the cases.

```

stage('Test') {
    steps {
        sh "mvn test"
    }
}

```

- In the **fourth stage**, we scan the whole source code to find out the vulnerabilities that may exist in the dependencies in the `pom.xml` file. Since, all the dependencies that are being used by this project are mentioned in this file.

Also, if we want to find if there is any kind of sensitive data that is being stored in my repository. For that, we can use this third-party tool known as **Trivy**.

For **Trivy**, the default plugin is not available. So we are going to install Trivy on this Jenkins server itself.

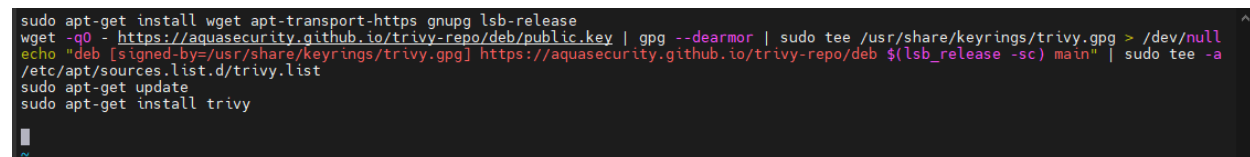
[On Jenkins Server]

vi trivy.sh

```

sudo apt-get install wget apt-transport-https gnupg lsb-release
wget -qO - https://aquasecurity.github.io/trivy-repo/deb/public.key | gpg --dearmor | sudo tee
/usr/share/keyrings/trivy.gpg > /dev/null
echo "deb [signed-by=/usr/share/keyrings/trivy.gpg] https://aquasecurity.github.io/trivy-repo/deb
$(lsb_release -sc) main" | sudo tee -a /etc/apt/sources.list.d/trivy.list
sudo apt-get update
sudo apt-get install trivy

```



```

sudo apt-get install wget apt-transport-https gnupg lsb-release
wget -qO - https://aquasecurity.github.io/trivy-repo/deb/public.key | gpg --dearmor | sudo tee /usr/share/keyrings/trivy.gpg > /dev/null
echo "deb [signed-by=/usr/share/keyrings/trivy.gpg] https://aquasecurity.github.io/trivy-repo/deb $(lsb_release -sc) main" | sudo tee -a /etc/apt/sources.list.d/trivy.list
sudo apt-get update
sudo apt-get install trivy

```

chmod +x trivy.sh

./trivy.sh

trivy --version

```

stage('File System Scan') {
    steps {
        sh "trivy fs --format table -o trivy-fs-report.html ."
    }
}

```

```
}
```

- **sh:** This is a shell step in Jenkins. It allows you to execute shell commands within your Jenkins pipeline.
- **"trivy fs --format table -o trivy-fs-report.html .":** This is the command being executed. Let's dissect it:
- **trivy fs:** This is invoking the Trivy tool with the fs command, which is used for scanning the file system.
- **--format table:** This option specifies the format in which the scan results will be displayed. In this case, it's set to table, indicating that the results will be formatted in a tabular structure.
- **-o trivy-fs-report.html:** This option specifies the output file name and format. Here, it's set to trivy-fs-report.html, indicating that the scan results will be saved in an HTML file named trivy-fs-report.html.
- **..:** This specifies the directory to be scanned. In this case, . represents the current directory, meaning Trivy will scan the current directory and its subdirectories for vulnerabilities.
- In the **fifth stage**, we are adding the SonarQube Analysis. The 'sonar' we have configured in the system where the URL and credentials are available, we are just calling it in the pipeline.

To define any third-party tools, such as the Sonar Scanner tool we have to define an environment variable, let's call it scanner_home, and assign it the path to the Sonar Scanner tool. Since we've already configured Sonar Scanner in our tool section, we can simply refer to it as "**sonar-scanner**".

```
environment {  
    SCANNER_HOME= tool 'sonar-scanner'  
}
```

Execute the analysis in the Pipeline:

- We enclose the commands within triple quotes because we have multiple lines of code.
- First, we call the scanner_home variable to locate the executable file of the Sonar Scanner tool inside the bin folder.
- Then, we add arguments to the command. For example:
 - **Dsonar.projectName:** Specifies the project name.
 - **Dsonar.projectKey:** Specifies the project key.
 - **Dsonar.java.binaries:** Specifies the location of Java binary files. Typically, it's inside the target folder.
- If there's any confusion, we can simply use a dot (.) to specify the current directory.

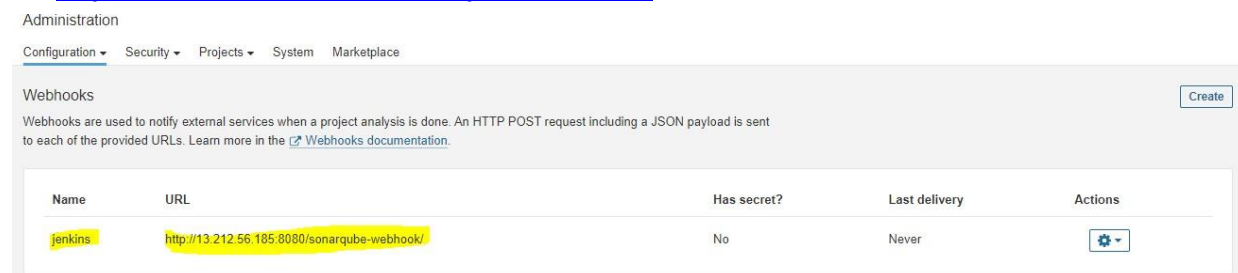
```
stage('SonarQube Analysis') {  
    steps {  
        withSonarQubeEnv('sonar') {  
            sh "'$SCANNER_HOME/bin/sonar-scanner -Dsonar.projectName=BoardGame -  
Dsonar.projectKey=BoardGame \  
-Dsonar.java.binaries=. '"  
        }  
    }  
}
```


- In **Sixth stage**, once the code analysis is done, we also check the **quality gate**.

The quality gate in SonarQube acts as a set of conditions for assessing code quality. These conditions are configured in the Quality Gate section of SonarQube. If the code meets these conditions, it's deemed to be of good quality.

To set up the quality gate, we first navigate to the **Administration > Configuration > Web Hook** section. Here, we create a new web hook. We provide a name for the web hook, and for the URL, we use a specific format, such as "**Jenkins public IP/sonar-webhook/**". This ensures that the quality gate can be executed properly.

Ex- <http://13.212.56.185:8080/sonarqube-webhook/>



Once the web hook is created, we proceed to write a stage in our Jenkins pipeline to perform the SonarQube analysis. This stage ensures that the quality gate check is integrated into our pipeline workflow seamlessly.

```
stage('Quality Gate') {
    steps {
        script {
            waitForQualityGate abortPipeline: false, credentialsId: 'sonar-cred'
        }
    }
}
```

- In **Seventh stage**, after the SonarQube analysis is done next step is, we can **build** the application so for building we can write build stage.

```
stage('Build') {
    steps {
        sh "mvn package"
    }
}
```

- In **Eighth stage**, we are going to **publish artifacts to Nexus**.

For that, we need to configure Nexus Repository URLs in our `pom.xml` file.

- Open the **POM.xml** file and paste the below tags in the xml file.

```
<distributionManagement>
  <repository>
    <id>maven-releases</id>
    <url>http://54.254.190.235:8081/repository/maven-releases/</url>
  </repository>
  <snapshotRepository>
    <id>maven-snapshots</id>
    <url>http://54.254.190.235:8081/repository/maven-snapshots/</url>
  </snapshotRepository>
</distributionManagement>
```

```

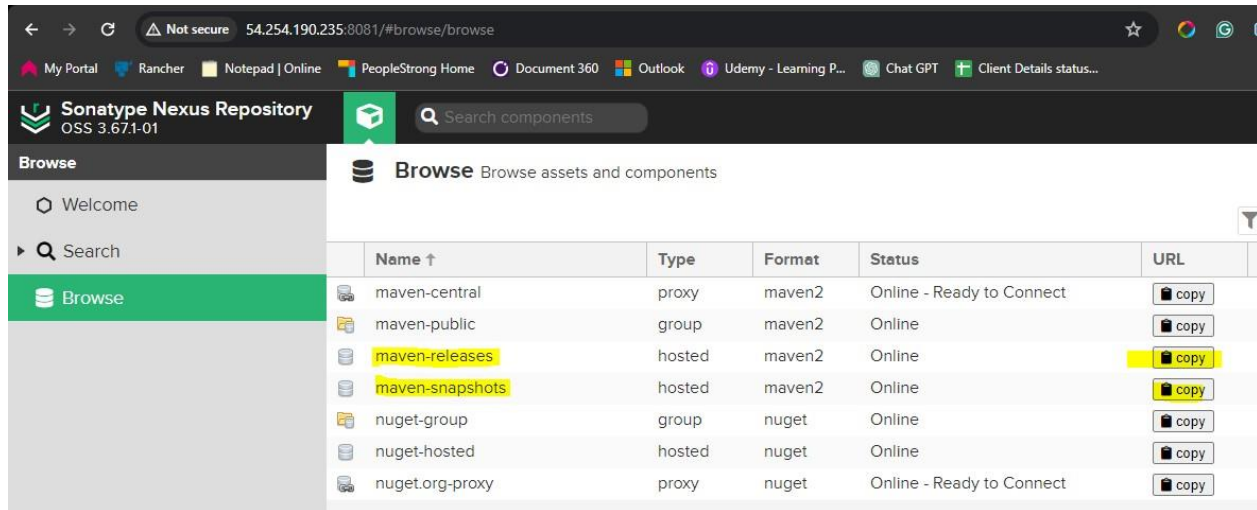
    </execution>
    <execution>
      <id>jacoco-site</id>
      <phase>package</phase>
      <goals>
        <goal>report</goal>
      </goals>
    </execution>
  </executions>
</plugin>
</plugins>
</build>

<distributionManagement>
  <repository>
    <id>maven-releases</id>
    <url>http://54.254.190.235:8081/repository/maven-releases/</url>
  </repository>
  <snapshotRepository>
    <id>maven-snapshots</id>
    <url>http://54.254.190.235:8081/repository/maven-snapshots/</url>
  </snapshotRepository>
</distributionManagement>
</project>
```

In this snippet:

- ``<id>``: This is a unique identifier for the repository.
- ``<url>``: This is the URL of your Nexus repository.

Copy the **ID** and **URL** from the Nexus UI dashboard.



- Once changes are done commit and push to the git hub.

git status

git add .

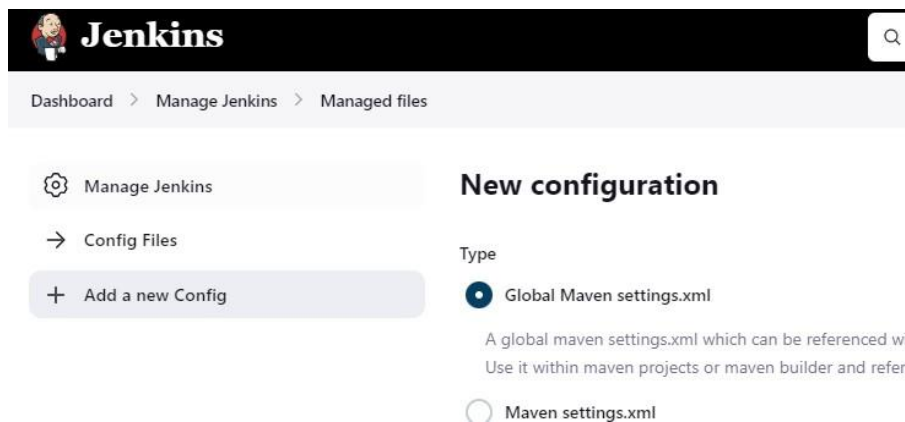
git commit -m "New Commit"

git push origin main

Still, we have not yet added the credential to access this repository. How do we do that so?

Since we have installed the plugin "**Config File Provider**" so when we go to Manage Jenkins we will see the option "**Managed Files**".

- Select "**Add a new Config**" > **Global Maven settings.xml**.
- Edit the ID and enter the name like "**global-settings**"

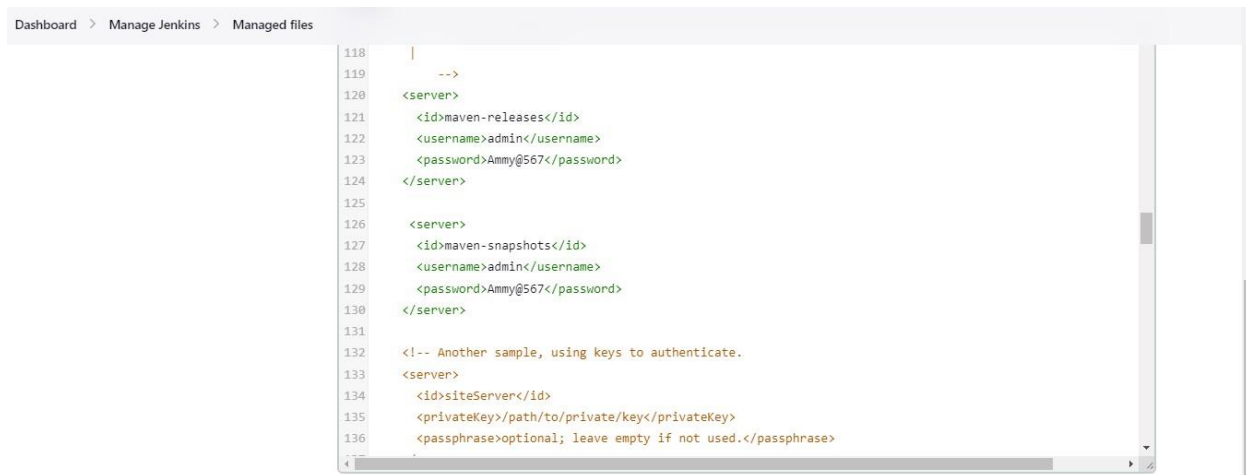


Now this file is known as the settings.xml file where we are going to provide the credentials for accessing Nexus.

- Under the Content section, uncomment the server block as shown in the screenshot.

```
<server>
  <id>maven-releases</id>
  <username>admin</username>
  <password>Ammy@567</password>
</server>

<server>
  <id>maven-snapshots</id>
  <username>admin</username>
  <password>Ammy@567</password>
</server>
```



- To write the pipeline command, use pipeline syntax.

Sample Step

withMaven: Provide Maven environment

withMaven ?

Maven ?

maven3

JDK ?

jdk17

Global Maven Settings Config ?

MyGlobalSettings

Global Maven Settings File Path ?

Generate Pipeline Script

```

withMaven(globalMavenSettingsConfig: 'global-settings', jdk: 'jdk17', maven: 'maven3', mavenSettingsConfig: '', traceability: true) {
    // some block
}

```

To deploy the application artifact to Nexus, you can use the following command in your pipeline:
sh 'mvn deploy'

This Maven command will deploy the artifacts of your application to Nexus, as configured in the <distributionManagement> section of your pom.xml file.

```

stage('Publish To Nexus') {
    steps {
        withMaven(globalMavenSettingsConfig: 'global-settings', jdk: 'jdk17', maven: 'maven3',
mavenSettingsConfig: '', traceability: true) {
            sh "mvn deploy"
        }
    }
}

```

- In the **Nineth stage**, we will execute the docker build command to build and tag the Docker image.
- Use the pipeline syntax to generating the docker block.

Pipeline Syntax

e

Sample Step

withDockerRegistry: Sets up Docker registry endpoint

- Add the Docker credentials, instead of going to the Manage Jenkins > Credentials, we can directly add it under the pipeline syntax. Click on Add button and enter the details.

Jenkins Credentials Provider: Jenkins

Add Credentials

Domain

Global credentials (unrestricted)

Kind

Username with password

Scope ?

Global (Jenkins, nodes, items, all child items, etc)

Username ?

amanduggal001

☐ Treat username as secret ?

Password ?

ID ?

docker-cred

Description ?

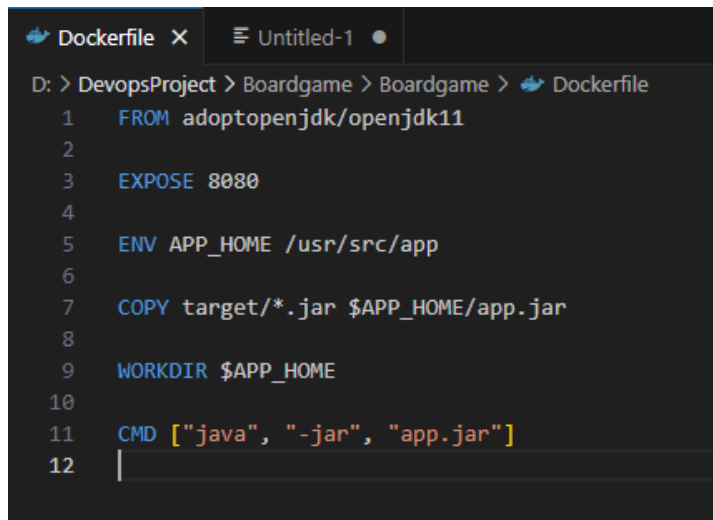
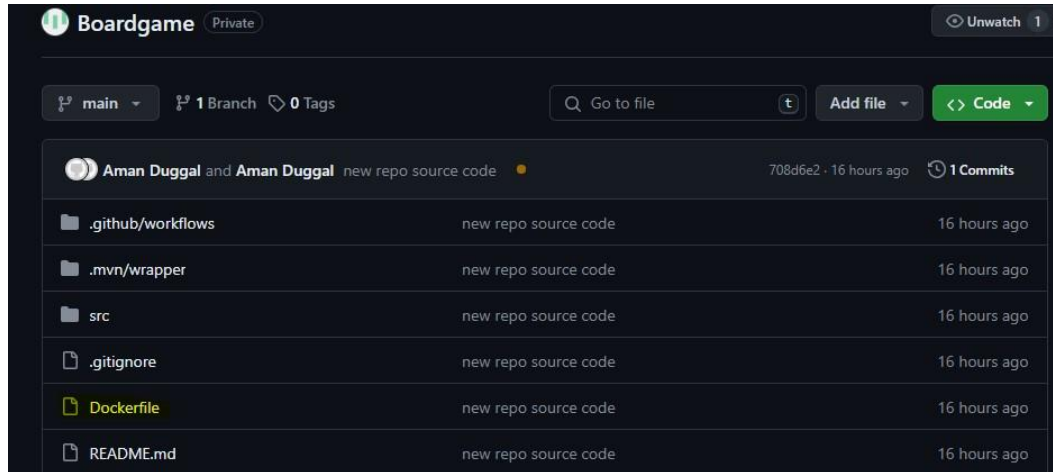
docker-cred

- Once created enter the Registry credentials of the docker which we created.

```
stage('Build & Tag Docker Image') {  
  steps {  
    script {  
      withDockerRegistry(credentialsId: 'docker-cred') {  
        sh "docker build -t amanduggal001/boardshack:latest ."  
      }  
    }  
  }  
}
```

This executes the Docker build command to build the **Docker image**. The -t option is used to tag the image with a name and tag. In this case, the image is tagged as amanduggal001/boardshack with the latest tag.

Note: Docker image is already present in the Git along with the source code.



- The next step in the **Tenth stage** we scan the Docker image before pushing it to the **DockerHub** repository. We can utilize **Trivy** for this task, which is advantageous because Trivy serves multiple purposes.

```
stage('Docker Image Scan') {  
    steps {  
        sh "trivy image --format table -o trivy-image-report.html amanduggal001/boardshack:latest"  
    }  
}
```

- In the **Eleventh stage**, once the scan of the docker images are done, then we push the image to docker hub.

```
stage('Push Docker Image') {
  steps {
    script {
      withDockerRegistry(credentialsId: 'docker-cred') {
        sh "docker push amanduggal001/boardshack:latest"
      }
    }
  }
}
```

- In **Twelfth stage**, once the image is pushed, the next step involves deploying the application to the Kubernetes Cluster.

To ensure this deployment is done correctly, we can implement **role-based access control (RBAC)**. RBAC is essential for managing permissions effectively within Kubernetes. Let me explain it visually, which might help clarify things.

Imagine we have three users in our project: User One, User Two, and User Three. User One is the architect with comprehensive knowledge, User Two is at a medium level, and User Three is an intern or a fresher.

Now, in Kubernetes, it's crucial not to grant complete access to inexperienced users. Hence, we create roles: Role One, Role Two, and Role Three.

- Role One grants cluster admin access, suitable for User One.
- Role Two provides a good level of permissions, appropriate for User Two.
- Role Three offers read-only access, ideal for User Three, who can only view but not modify anything.

This approach is what we mean by role-based access control (RBAC). It's a security measure to assign specific roles to users based on their level of expertise, ensuring security and preventing unauthorized modifications.

Now, let see how we can implement this. To create a service account, we first need to create a separate user.

[On Kubernetes Master Node]

Creating Service Account

vi svc.yml

```
apiVersion: v1
kind: ServiceAccount
metadata:
```



```
name: jenkins
namespace: webapps
```

This YAML file creates a service account named "jenkins" specifically for performing deployments. These deployments will be isolated within a separate project or namespace, which we'll name "webapps."

```
kubect! create ns webapps --#To create the namespace 'webapps'.
kubect! apply -f svc.yml
```

Creating Role

```
vi role.yml
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: app-role
namespace: webapps
rules:
- apiGroups:
  - ""
  - apps
  - autoscaling
  - batch
  - extensions
  - policy
  - rbac.authorization.k8s.io
resources:
- pods
- componentstatuses
- configmaps
- daemonsets
- deployments
- events
- endpoints
- horizontalpodautoscalers
- ingress
- jobs
- limitranges
- namespaces
- nodes
- pods
- persistentvolumes
- persistentvolumeclaims
- resourcequotas
- replicaset
- replicationcontrollers
- serviceaccounts
```

```
- services
verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

This Yaml file defines a Role resource in Kubernetes for RBAC. The Role (app-role) grants comprehensive permissions within the "webapps" namespace, allowing actions like creating, updating, and deleting various resources across multiple API groups. This Role can be bound to users, groups, or service accounts within the namespace to grant them the specified permissions.

kubectI apply -f role.yml

Till now, we've created the service account (or user) and created a corresponding role. Our next step is to link this role with the service account. To do this, we utilize a **binding mechanism**.

Bind the role to service account

vi bind.yml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: app-rolebinding
namespace: webapps
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: app-role
subjects:
- namespace: webapps
  kind: ServiceAccount
  name: jenkins
```

kubectI apply -f bind.yml

We've successfully granted permissions to the user (or service account) named "jenkins" to perform deployments and other operations. Now, we want our Jenkins to connect to the Kubernetes cluster. To achieve this, we'll create a token that Jenkins can use for authentication.

Generate token using service account in the namespace

vi sec.yml

```
apiVersion: v1
kind: Secret
type: kubernetes.io/service-account-token
metadata:
  name: mysecretname
  annotations:
    kubernetes.io/service-account.name: jenkins
```

Since in yml file, we didn't define the namespace. So we can apply this file by defining the namespace.

Now, when describe the secret in K8s master node you will see the token which we will use to authenticate with jenkins.

kubectl describe secrets -n webapps

```
root@ip-172-31-34-119:/home/ubuntu# kubectl describe secret -n webapps
Name:         mysecretname
Namespace:    webapps
Labels:       <none>
Annotations:  kubernetes.io/service-account.name: jenkins
              kubernetes.io/service-account.uid: 4ccf0690-366a-414e-b048-fdddf77a923f

Type: kubernetes.io/service-account-token

Data
====
token: eyJhbGciOiJSUzI1NiIsImtpZCI6Im02RjE1FERFLcDlMTlGLWGHNSITRSRJNjZjd2I1NHX3E1LVFLSkx6M2E4WEIfq.eyJpc3MiOiJrdWJlcml5dGVzL3NlcncpY2VhY2tvdW50Iiwia3ViZXJzXRRLcySpby9zXJZ2aWNLYWNjb3VudC9zZWlyZXQubmFtZSI6Im15c2Vjc2VmbmFtZSIsIntIYmVmYm9kZXMuYW8vc2VydmllZWFiY291bnQvc2VydmllZS1hY2tvdW50LmShbwIoIQJZw5raW5zIiwia3ViZXJzXRRLcySpby9zXJZ2aWNLYWNjb3VudC9zXJZ2aWNLLWFiY291bnQvdWJlcml5IjoINGNjZjA2OTAtMzYyZS00MTRLLWlnWmdgtZmRkZGY3NE2EMjNmIiwic3ViIjoic3lzdGV0bnNlc2pY2VhY2tvdW50bnRlYmFwcHlw6mVua2LucyJ9.nrjBdyTbVM618RGdcrcv5vsxb7t8k-MFP23Apevh-EqxijqAaePb_Kj8oJQ55xcL-ftjIIPwUGtahvIo82WjaNMeg9fvQERxLSqZE_DVP4mA7ktPl2xHD2SeyyyinLspb_gOE9nLpdy69qQ4ZdxNGgdgV51ew0jnFnFDYdxaMFdhDScYATAISIZD15XCMRxpixuywrgKUOHaoXjkp3JuqT54FY4Rphw9SBRCvrqbVAet4j5g29UGLFs7Iwt2M_YDSYxXPx21mOKBU7UZbNB1ggoIk0rFHVP_punATwilbfM6m3r8g4ER0CcSIPEj33hLEd5OIE32ActKMxjBw
ca.crt: 1187 bytes
namespace: 7 bytes
root@ip-172-31-34-119:/home/ubuntu#
```

Now, create a credentials in the Jenkins UI with this token.

Dashboard > Manage Jenkins > Credentials > System > Global credentials (unrestricted) >

New credentials

Kind

Secret text

Scope ?

Global (Jenkins, nodes, items, all child items, etc)

Secret

.....

ID ?

k8-cred

Description ?









k8-cred

Create

Global credentials (unrestricted)

+ Add Credentials

Credentials that should be available irrespective of domain specification to requirements matching.

ID	Name	Kind	Description	
 git-cred	amanduggal001/***** (git-cred)	Username with password	git-cred	
 sonar-cred	sonar-cred	Secret text	sonar-cred	
 docker-cred	amanduggal001/***** (docker-cred)	Username with password	docker-cred	
 k8-cred	k8-cred	Secret text	k8-cred	

Now, create the pipeline. Use the pipeline script to get the block.

ce

Sample Step

withKubeConfig: Configure Kubernetes CLI (kubectl)

withKubeConfig ?

Credentials

k8s-cred

+ Add

Kubernetes server endpoint ?

https://172.31.34.119:6443

Cluster name ?

kubernetes

We will get the 'Kubernetes server endpoint' & 'Cluster Name' from the worker node of k8s.

```
cd ~/.kube
```

```
cat config
```

```
root@ip-172-31-34-119:/home/ubuntu# cd ~/.kube
root@ip-172-31-34-119:~/.kube# ls
cache  config
root@ip-172-31-34-119:~/.kube# cat config
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSB0tLS0tck1
    XdGVEVUTUJFR0ExVUUKQXhNS2EzVmIawEp1wIhSbGN6QWVGdzB5TkRBME1USXh0akF5TWpa
    BaWE13Z2dFaU1BMEdDU3FHU0l iM0RRRUJBUVVBQTRJQkR3QXdnZ0VLckFvSUJBUURJSDI3c
    rOXkKckZmdW00RnhjNUhnbWpYNG8zSTdNQkl5VkIwVmNlZSsxduRoQVdsVnY3S0MwR0JJa0
    SudiiejVCVFVjVmdGQlY1L2x5STNNU2ZJMkk3RWVsCmorYitRTGZRdkRmU3NJSVdhOHVmaE0
    GN2akNFZDBYOWZmOHpGNW5JazZDc2tScGt0SnJPNXQ4UXplMFhNEl10TYyN3lSYWtadTI2
    ExVWREd0VCL3dRRUF3SUNwREFQckJnTlZIuk1CQWY4RUJUQURBUUgvTUiwR0ExVWREZ1FXQ
    XSmxjbTVsZEdwek1BMEdDU3FHU0l iM0RRRUJDD1VBQTRJQkFRRENYUEJGSWJXbGpzZWl2a2
    VGdBUU5jRWRNClhVcUxQdWNEdnhLeXNleDlEUTFjYk12aFg5ODdaUk93bGdCUHNLZHd2cGh
    kxSL0RTWFdXWG9EaFlsL2Fnaw1pYkIvbnJsN3dnWGttramdNMwpaMGJUMjhUTW5ZdjByZnk3
    RQaFVkbjU1aTFcbThJOVFLRlQrY29kNVJxR2RYR3lxRUR00ENN0WhxRlR6Q3BwVEdoNlc1S
    K
    server: https://172.31.34.119:6443
    name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
    name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
```

Create a manifest file and save it in the GitHub repository.

deployment-service.yaml

```
apiVersion: apps/v1
kind: Deployment # Kubernetes resource kind we are creating
metadata:
  name: boardgame-deployment
spec:
  selector:
    matchLabels:
      app: boardgame
  replicas: 2 # Number of replicas that will be created for this deployment
  template:
    metadata:
      labels:
        app: boardgame
    spec:
      containers:
        - name: boardgame
          image: amanduggal001/boardshack:latest # Image that will be used to
containers in the cluster
          imagePullPolicy: Always
          ports:
            - containerPort: 8080 # The port that the container is running on in
the cluster

---
apiVersion: v1 # Kubernetes API version
kind: Service # Kubernetes resource kind we are creating
metadata: # Metadata of the resource kind we are creating
  name: boardgame-ssvc
spec:
  selector:
    app: boardgame
  ports:
    - protocol: "TCP"
      port: 8080 # The port that the service is running on in the cluster, since
we define this port in docker file.
      targetPort: 8080 # The port exposed by the service
      type: LoadBalancer # type of the service.
```

This YAML script defines two Kubernetes resources:

1. Deployment:

- Creates a Deployment named "boardgame-deployment".
- Specifies to create 2 replicas (copies) of the pod.
- Defines a pod template with a container named "boardgame" using the Docker image "amanduggal001/boardshack:latest", listening on port 8080.

2. Service:

- Creates a Service named "boardgame-ssvc".
- Routes traffic to pods labeled with "app: boardgame".
- Exposes port 8080 on the Service, forwarding it to the pods' port 8080.
- Configures the Service as a LoadBalancer type, allowing external access to the pods.

Now the problem is we didn't install the kubectl in the Jenkins server, so how this manifest file will be executed.

For that we need to install the kubectl in the Jenkins server.

[On Jenkins Node]

```
vi kube.sh
```

```
curl -o kubectl https://amazon-eks.s3.us-west-2.amazonaws.com/1.19.6/2021-01-05/bin/linux/amd64/kubectl
chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin
kubectl version --short --client
```

```
chmod +x kube.sh
./kube.sh
```

Pipeline script for 'Deploy To Kubernetes' stage:

```
stage('Deploy To Kubernetes') {
    steps {
        withKubeConfig(caCertificate: '', clusterName: 'kubernetes', contextName: '', credentialsId: 'k8-cred', namespace: 'webapps', restrictKubeConfigAccess: false, serverUrl: 'https://172.31.34.119:6443') {
            sh "kubectl apply -f deployment-service.yaml"
        }
    }
}
```

- In the **Thirteenth stage**, we will verify if the deployment has been completed successfully.

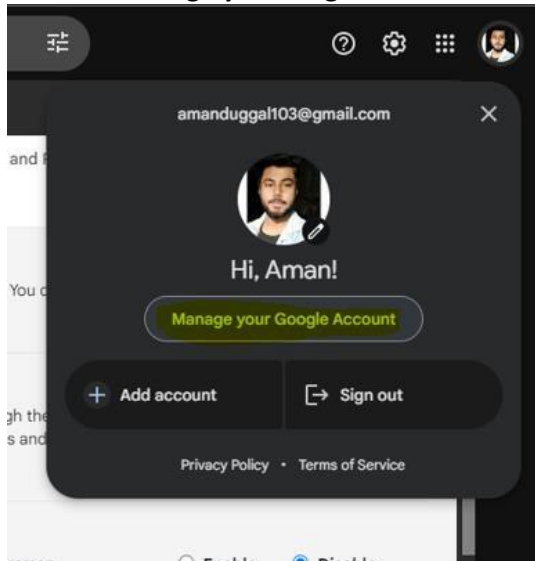
```
stage('Verify the Deployment') {
    steps {
        withKubeConfig(caCertificate: '', clusterName: 'kubernetes', contextName: '', credentialsId: 'k8-cred', namespace: 'webapps', restrictKubeConfigAccess: false, serverUrl: 'https://172.31.34.119:6443') {
            sh "kubectl get pods -n webapps"
            sh "kubectl get svc -n webapps"
        }
    }
}
```

- In the next step, we need to configure **email notifications**.

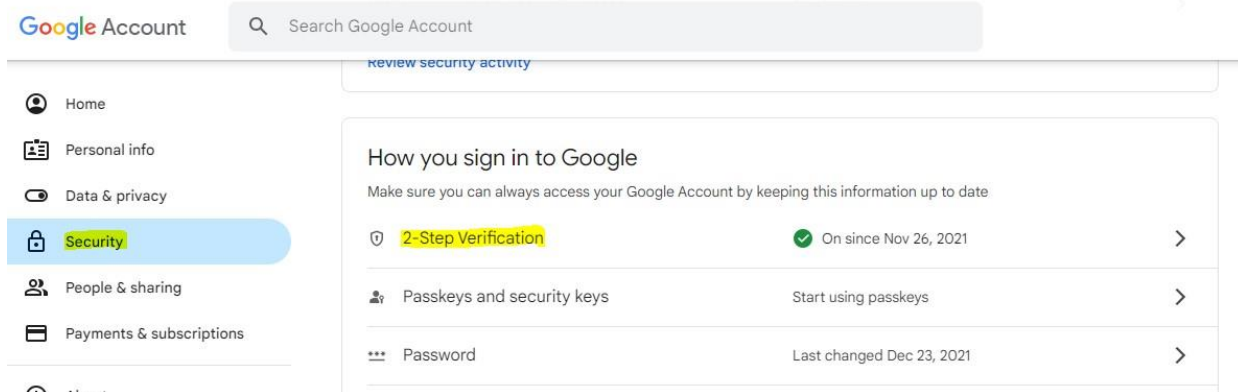
To do this, we need to set up an application-specific password for Gmail by enabling **two-step verification** and generating the password in Google account settings.

Once we have the password, we can configure email notifications in Jenkins by navigating to the Jenkins dashboard, accessing the system settings, and configuring email notifications under the "Manage Jenkins" section.

- Click on '**Manage your Google Account**'.



- Select **Security > Two step verification**.



- Select **App Password**.

App passwords

App Passwords aren't recommended and are unnecessary in most cases. To help keep your account secure, use "Sign in with Google" to connect apps to your Google Account.

App passwords

None

>

- Enter the **App name** = jenkins. Once you create it you will see the App password, copy this password.

← App passwords

... into your Google Account on older apps and services that don't support modern security standards.

App passwords are less secure than using up-to-date apps and services that use modern security standards. Before you create an app password, you should check to see if your app needs this in order to sign in.

[Learn more](#)

You don't have any app passwords.

To create a new app specific password, type a name for it below...

App name

jenkins

Create

Generated app password

Your app password for your device

~~xxxxxxxxxxxx~~vmjk dejj avsg

How to use it

Go to the settings for your Google Account in the application or device you are trying to set up. Replace your password with the 16-character password shown above.

Just like your normal password, this app password grants complete access to your Google Account. You won't need to remember it, so don't write it down or share it with anyone.

Done

- Now configure the Email in the Jenkins. Go to **Manage Jenkins > System > Extended E-mail Notification**.
- **SMTP server:** smtp.gmail.com
- **SMTP Port:** 465
- **Credentials:** Select Use SSL and Add credentials.
- **Username:** amanduggal103@gmail.com
- **Password:** Enter the mail App password which we created.
- **ID:** gmail-cred
- **Description:** gmail-cred

Jenkins Credentials Provider: Jenkins

Kind

Username with password

Scope ?

Global (Jenkins, nodes, items, all child items, etc)

Username ?

amanduggal103@gmail.com

☐ Treat username as secret ?

Password ?

ID ?

gmail-cred

Description ?

Manage Jenkins > System >

Extended E-mail Notification

SMTP server

smtp.gmail.com

SMTP Port

465

Advanced ^

 Edited

Credentials

amanduggal103@gmail.com/***** (gmail-cred)

+ Add ▾



Use SSL

- Now, go to **Email Notification**.
- **SMTP server:** smtp.gmail.com
- Select '**Use SSL**' and '**Use SMTP Authentication**'.
- **Username:** amanduggal103@gmail.com
- **Password:** Enter the mail App password which we created.
- **SMTP Port:** 465

E-mail Notification

SMTP server

smtp.gmail.com

Default user e-mail suffix ?

Advanced ^

✎ Edited

☒ Use SMTP Authentication ?

User Name

amanduggal103@gmail.com

Password

☒ Use SSL ?

☐ Use TLS

SMTP Port ?

465

Save

Apply

- Now test the configuration by sending the email from Jenkins.

☒ Test configuration by sending test e-mail

Test e-mail recipient

amanduggal103@gmail.com

Email was successfully sent

Test email #1 > Inbox x



address not configured yet <amanduggal103@gmail.com>
to me ▾

This is test email #1 sent from Jenkins

↩ Reply

➦ Forward



- Now, we can integrate email notifications into our pipeline by adding a post-notification section in the stages and specifying the recipient email address, sender details, and any attachments, such as reports.

Pipeline Script for gmail configuration:

```
post {
  always {
    script {
      def jobName = env.JOB_NAME
      def buildNumber = env.BUILD_NUMBER
      def pipelineStatus = currentBuild.result ?: 'UNKNOWN'
      def bannerColor = pipelineStatus.toUpperCase() == 'SUCCESS' ? 'green' : 'red'

      def body = """
        <html>
        <body>
        <div style="border: 4px solid ${bannerColor}; padding: 10px;">
        <h2>${jobName} - Build ${buildNumber}</h2>
        <div style="background-color: ${bannerColor}; padding: 10px;">
        <h3 style="color: white;">Pipeline Status: ${pipelineStatus.toUpperCase()}</h3>
        </div>
        <p>Check the <a href="${BUILD_URL}">console output</a>.</p>
        </div>
        </body>
        </html>
        """

      emailtext (
        subject: "${jobName} - Build ${buildNumber} - ${pipelineStatus.toUpperCase()}",
        body: body,
        to: 'amanduggal103@gmail.com',
        from: 'jenkins@example.com',
        replyTo: 'jenkins@example.com',
        mimeType: 'text/html',
        attachmentsPattern: 'trivy-image-report.html'
      )
    }
  }
}
```

This script sends an email notification with detailed information about the Jenkins job's build status, including a link to the console output, and optionally attaches a report file.

- Finally, we'll save the configuration and ensure that it's correctly applied to every stage of our pipeline.

Final Pipeline script:

```
pipeline {
  agent any

  tools {
    jdk 'jdk17'
    maven 'maven3'
  }

  environment {
    SCANNER_HOME= tool 'sonar-scanner'
  }

  stages {
    stage('Git Checkout') {
      steps {
        git branch: 'main', credentialsId: 'git-cred', url:
'https://github.com/amanduggal001/Boardgame.git'
      }
    }

    stage('Compile') {
      steps {
        sh "mvn compile"
      }
    }

    stage('Test') {
      steps {
        sh "mvn test"
      }
    }

    stage('File System Scan') {
      steps {
        sh "trivy fs --format table -o trivy-fs-report.html ."
      }
    }

    stage('SonarQube Analysis') {
      steps {
        withSonarQubeEnv('sonar') {
          sh "' $SCANNER_HOME/bin/sonar-scanner -Dsonar.projectName=BoardGame -
Dsonar.projectKey=BoardGame \
-Dsonar.java.binaries=. '"
        }
      }
    }
  }
}
```

```

    }

    stage('Quality Gate') {
        steps {
            script {
                waitForQualityGate abortPipeline: false, credentialsId: 'sonar-token'
            }
        }
    }

    stage('Publish To Nexus') {
        steps {
            withMaven(globalMavenSettingsConfig: 'global-settings', jdk: 'jdk17', maven: 'maven3',
mavenSettingsConfig: '', traceability: true) {
                sh "mvn deploy"
            }
        }
    }

    stage('Build & Tag Docker Image') {
        steps {
            script {
                withDockerRegistry(credentialsId: 'docker-cred') {
                    sh "docker build -t amanduggal001/boardshack:latest ."
                }
            }
        }
    }

    stage('Docker Image Scan') {
        steps {
            sh "trivy image --format table -o trivy-image-report.html amanduggal001/boardshack:latest"
        }
    }

    stage('Push Docker Image') {
        steps {
            script {
                withDockerRegistry(credentialsId: 'docker-cred') {
                    sh "docker push amanduggal001/boardshack:latest"
                }
            }
        }
    }

    stage('Deploy To Kubernetes') {
        steps {

```

```

        withKubeConfig(caCertificate: "", clusterName: 'kubernetes', contextName: "", credentialsId: 'k8-cred', namespace: 'webapps', restrictKubeConfigAccess: false, serverUrl: 'https://172.31.34.119:6443') {
            sh "kubectl apply -f deployment-service.yaml"
        }
    }
}

stage('Verify the Deployment') {
    steps {
        withKubeConfig(caCertificate: "", clusterName: 'kubernetes', contextName: "", credentialsId: 'k8-cred', namespace: 'webapps', restrictKubeConfigAccess: false, serverUrl: 'https://172.31.34.119:6443') {
            sh "kubectl get pods -n webapps"
            sh "kubectl get svc -n webapps"
        }
    }
}

post {
    always {
        script {
            def jobName = env.JOB_NAME
            def buildNumber = env.BUILD_NUMBER
            def pipelineStatus = currentBuild.result ?: 'UNKNOWN'
            def bannerColor = pipelineStatus.toUpperCase() == 'SUCCESS' ? 'green' : 'red'

            def body = """
            <html>
            <body>
            <div style="border: 4px solid ${bannerColor}; padding: 10px;">
            <h2>${jobName} - Build ${buildNumber}</h2>
            <div style="background-color: ${bannerColor}; padding: 10px;">
            <h3 style="color: white;">Pipeline Status: ${pipelineStatus.toUpperCase()}</h3>
            </div>
            <p>Check the <a href="${BUILD_URL}">console output</a>.</p>
            </div>
            </body>
            </html>
            """

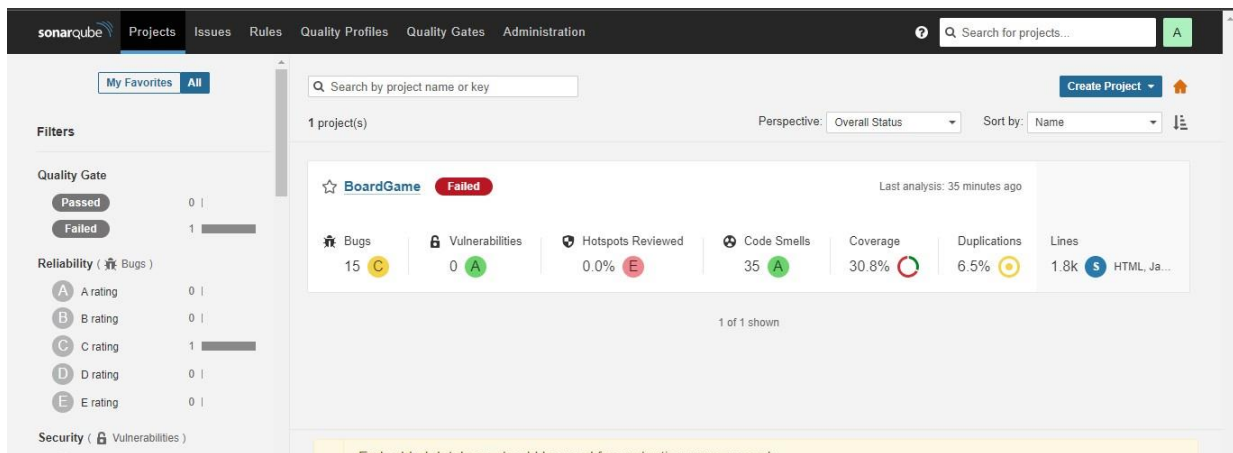
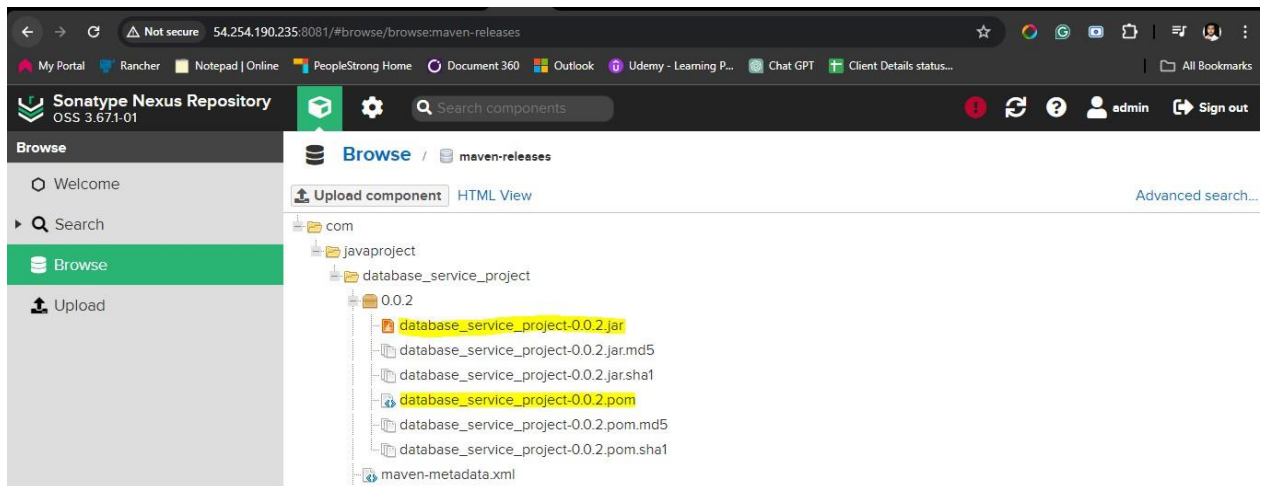
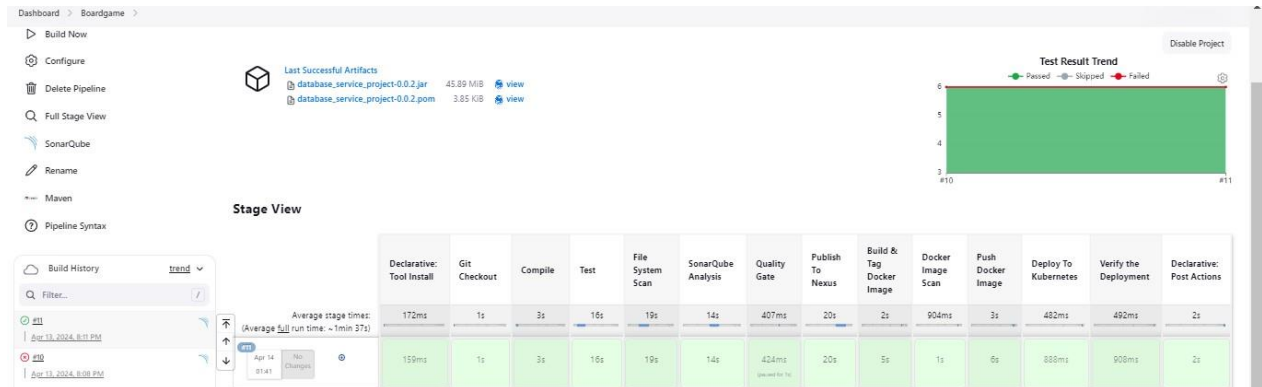
            emailx (
                subject: "${jobName} - Build ${buildNumber} - ${pipelineStatus.toUpperCase()}",
                body: body,
                to: 'amanduggal103@gmail.com',
                from: 'jenkins@example.com',
                replyTo: 'jenkins@example.com',
                mimeType: 'text/html',
            )
        }
    }
}

```

```

    attachmentsPattern: 'trivy-image-report.html'
  }
}
}
}

```



amanduggal001

Search by repository name

All Content

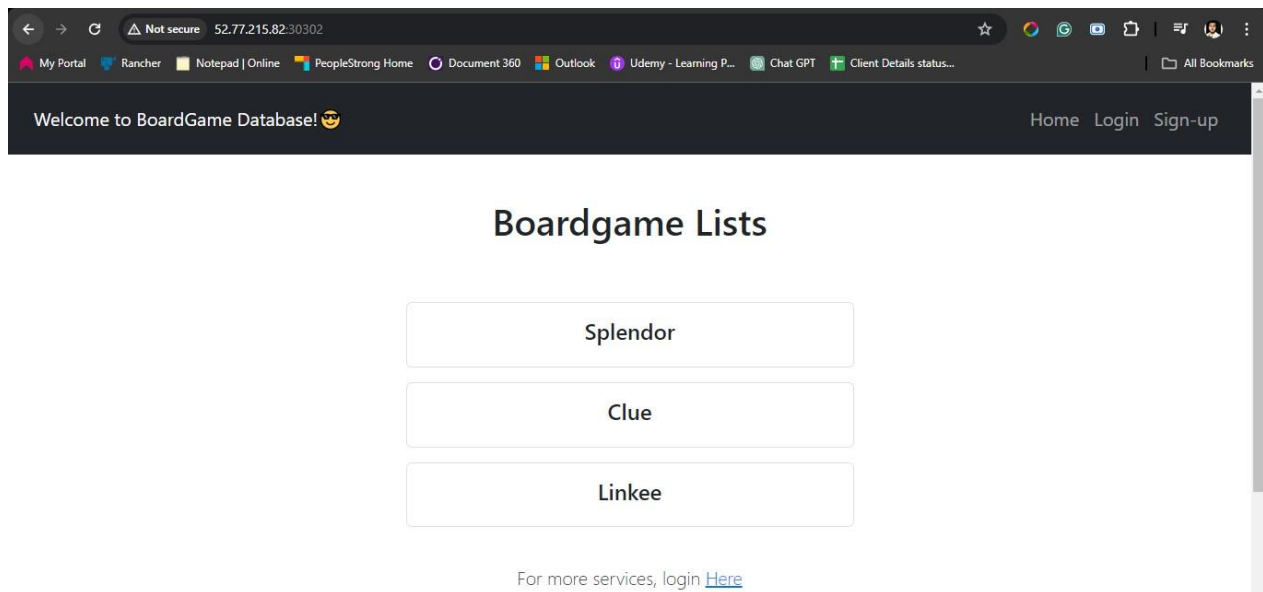
Create repository

amanduggal001 / boardshack

Security unknown
0
3
Public

Contains: Image • Last pushed: 35 minutes ago

```
+ kubectl get pods -n webapps
NAME                                READY   STATUS             RESTARTS   AGE
boardgame-deployment-796cf5f76d-zgb2s 0/1     ContainerCreating   0          1s
boardgame-deployment-796cf5f76d-zxgxz 0/1     ContainerCreating   0          1s
[Pipeline] sh
+ kubectl get svc -n webapps
NAME            TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
boardgame-ssvc  LoadBalancer  10.106.116.216 <pending>      8080:30302/TCP  1s
[Pipeline] }
[kubernetes-cli] kubectl configuration cleaned up
[Pipeline] // withKubeConfig
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Declarative: Post Actions)
[Pipeline] script
[Pipeline] {
[Pipeline] emailx
Sending email to: amanduggal103@gmail.com
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```



Till now Phase 3 is completed, now we perform the Monitoring in Phase 4.

Phase 4: Monitoring

- Create the EC2 instance with the same configuration which we done for other instances.
- Open the terminal.

sudo apt update

- Let us start by installation of Grafana, Prometheus and BlackBox.

Prometheus Installation:

wget <https://github.com/prometheus/prometheus/releases/download/v2.51.1/prometheus-2.51.1.linux-amd64.tar.gz>

tar -xvf prometheus-2.51.1.linux-amd64.tar.gz

ls

The prometheus file present here is the executable file, so as soon as we execute this file the Prometheus will get started.

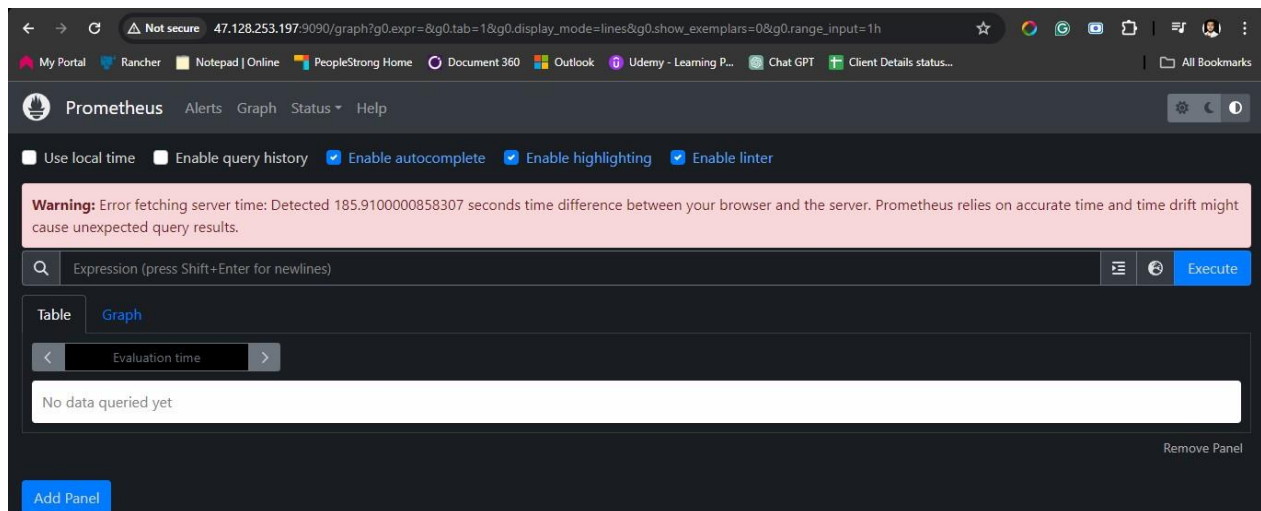
```
root@ip-172-31-41-208:/home/ubuntu/prometheus-2.51.1.linux-amd64# ls
LICENSE  NOTICE  console_libraries  consoles  prometheus  prometheus.yml  promtool
root@ip-172-31-41-208:/home/ubuntu/prometheus-2.51.1.linux-amd64#
```

./prometheus & --#To execute the file. We add &, because we want it to run this in background.

By default Prometheus is running on port **9090**. To access,

<Public-IP of the monitoring server>:9090

Ex- 47.128.253.197:9090



Grafana Installation:

```
cd ~
```

```
sudo apt-get install -y adduser libfontconfig1 musl
```

```
wget https://dl.grafana.com/enterprise/release/grafana-enterprise_10.4.2_amd64.deb
```

```
sudo dpkg -i grafana-enterprise_10.4.2_amd64.deb
```

```
sudo /bin/systemctl start grafana-server --#To start the grafana.
```

By default Grafana is running on port **3000**. To access,

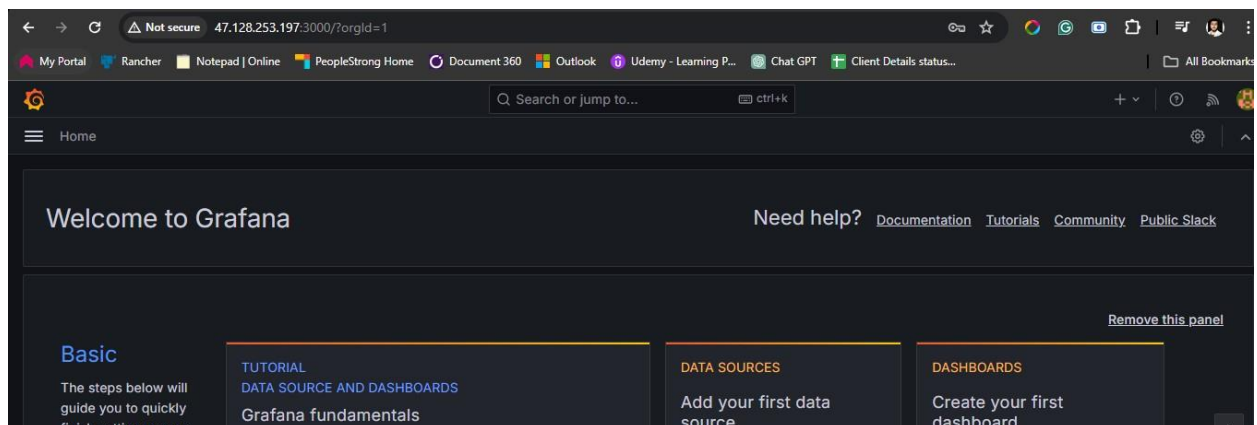
<Public-IP of the monitoring server>:3000

Ex- 47.128.253.197:3000

Default Cred-

Username: admin

Password: admin



Blackbox Installation:

Now, installing the Blackbox Exporter, which will assist us in monitoring websites.

wget

https://github.com/prometheus/blackbox_exporter/releases/download/v0.25.0/blackbox_exporter-0.25.0.linux-amd64.tar.gz

tar -xvf blackbox_exporter-0.25.0.linux-amd64.tar.gz

cd blackbox_exporter-0.25.0.linux-amd64

ls

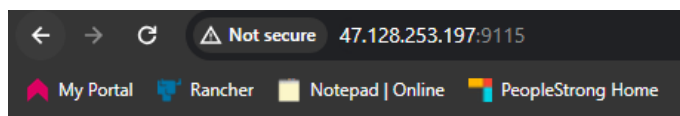
```
root@ip-172-31-41-208:~/blackbox_exporter-0.25.0.linux-amd64# ls
LICENSE  NOTICE  blackbox.yml  blackbox_exporter
```

./blackbox_exporter & --#To execute the file. We add &, because we want it to run this in background.

By default Blackbox is running on port **9115**. To access,

<Public-IP of the monitoring server>:9115

Ex- 47.128.253.197:9115



Blackbox Exporter

[Probe prometheus.io for http_2xx](#)

[Debug probe prometheus.io for http_2xx](#)

[Metrics](#)

[Configuration](#)

Recent Probes

Module	Target	Result	Debug
--------	--------	--------	-------

Now edit the prometheus.yaml file and add the content.

```
- job_name: 'blackbox'
  metrics_path: /probe
  params:
    module: [http_2xx] # Look for a HTTP 200 response.
  static_configs:
    - targets:
        # In the "Target" section, we can define exactly what we want to monitor
        - http://prometheus.io
        - http://52.77.215.82:30302 # URL in which our application is running.
  relabel_configs:
    - source_labels: [__address__]
      target_label: __param_target
```

- source_labels: [__param_target]
- target_label: instance
- target_label: __address__
- replacement: 47.128.253.197:9115 # Enter the <Public-IP of the monitoring server>:9115

```
scrape_configs:
  # The job name is added as a label 'job=<job_name>' to any timeseries scraped from this config.
  - job_name: "prometheus"

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
      - targets: ["localhost:9090"]

  - job_name: 'blackbox'
    metrics_path: /probe
    params:
      module: [http_2xx] # Look for a HTTP 200 response.
    static_configs:
      - targets:
          - http://prometheus.io # Target to probe with http.
          - http://52.77.215.82:30302
    relabel_configs:
      - source_labels: [__address__]
        target_label: __param_target
      - source_labels: [__param_target]
        target_label: instance
      - target_label: __address__
        replacement: 47.128.253.197:9115 # The blackbox exporter's real hostname:port.

~
-- INSERT --
```

Now restart the Prometheus.

pgrep prometheus --#By this we get the Process id and then we kill it.

kill 1951

```
root@ip-172-31-41-208:/home/ubuntu/prometheus-2.51.1.linux-amd64# pgrep prometheus
1951
root@ip-172-31-41-208:/home/ubuntu/prometheus-2.51.1.linux-amd64# kill 1951
root@ip-172-31-41-208:/home/ubuntu/prometheus-2.51.1.linux-amd64# ts=2024-04-13T21:58:48.972Z caller=main.go:964 level=info msg="Stopping notify discovery manager..."
SIGTERM, exiting gracefully...
ts=2024-04-13T21:58:48.972Z caller=main.go:988 level=info msg="Stopping scrape discovery manager..."
ts=2024-04-13T21:58:48.972Z caller=main.go:1002 level=info msg="Stopping notify discovery manager..."
ts=2024-04-13T21:58:48.972Z caller=manager.go:177 level=info component="rule manager" msg="Stopping rule manager..."
ts=2024-04-13T21:58:48.972Z caller=manager.go:187 level=info component="rule manager" msg="Rule manager stopped"
```

./prometheus & --#To restart the service.

Now, go to Prometheus Dashboard > Status > Targets.

So, this is our application's target URL. We're also seeing some metrics scraped by the blackbox exporter.

Targets

All scrape pools ▾ All Unhealthy Collapse All 🔍 Filter by endpoint or labels

Unknown Unhealthy Healthy

blackbox (2/2 up) show less

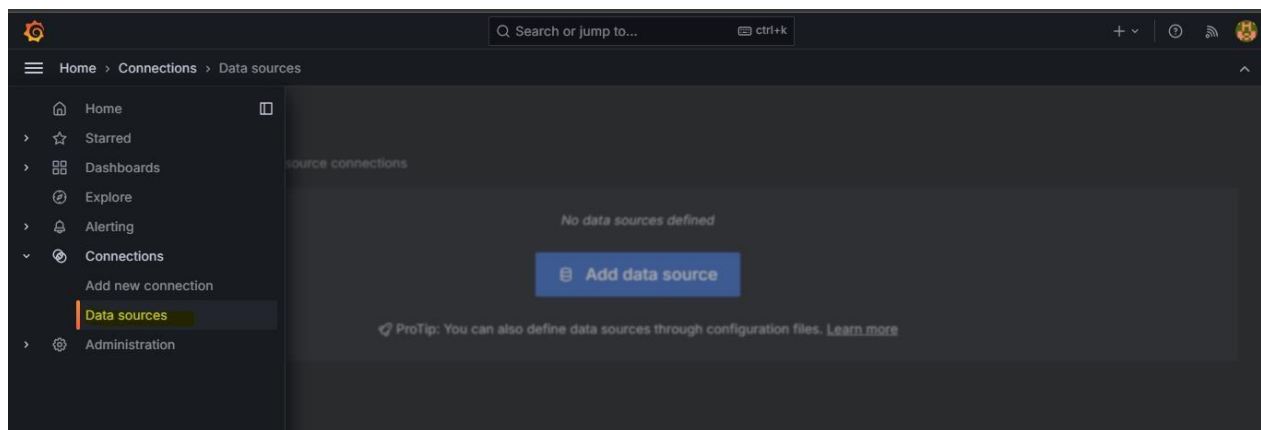
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://47.128.253.197:9115/probe module="http_2xx" target="http://prometheus.io"	UP	instance="http://prometheus.io" job="blackbox"	3m 20s ago	41.350ms	
http://47.128.253.197:9115/probe module="http_2xx" target="http://52.77.215.82:30302"	UP	instance="http://52.77.215.82:30302" job="blackbox"	3m 26s ago	9.502s	

prometheus (1/1 up) show less

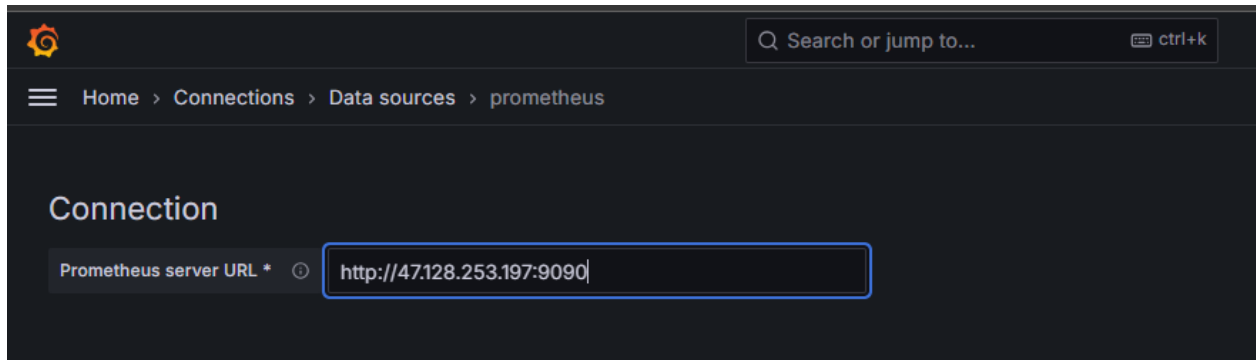
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:9090/metrics	UP	instance="localhost:9090" job="prometheus"	3m 9s ago	5.802ms	

The next thing that we need we want to add is Prometheus as a data source inside our Grafana.

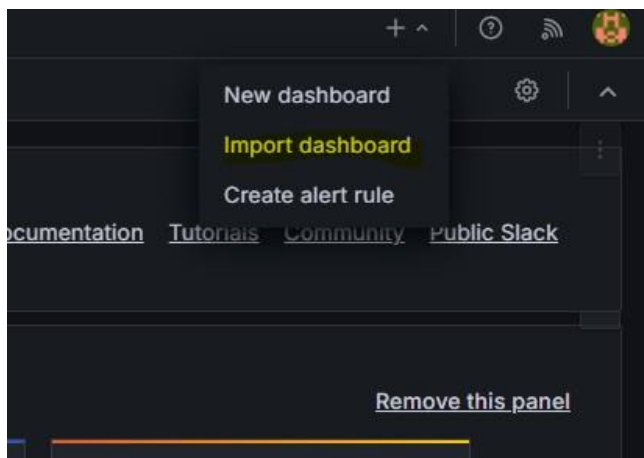
- Go to **Grafana Dashboard > Connections > Data Source > Add data source**.



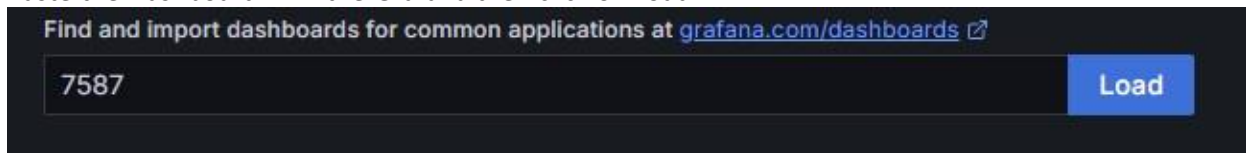
- Select Prometheus.
- Add the Prometheus URL.



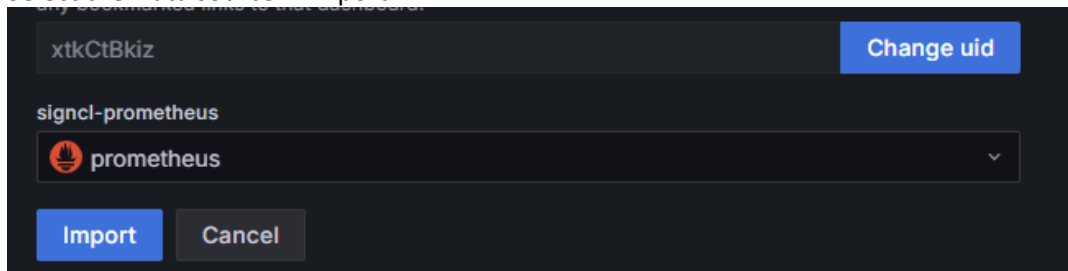
- Now Select the '**Import Dashboard**'.



- Copy the dashboard ID from - <https://grafana.com/grafana/dashboards/7587-prometheus-blackbox-exporter/>
- Paste the Dashboard ID in the Grafana then click on Load.



- Select the Data source > Import.



- Now the Dashboard is setup completely.



So this is the website monitoring for HTTP. Now to check the system level monitoring, let's say we are monitoring the system metrics of Jenkins.

For that install the below plugin.

- Manage Jenkins > Plugins > Available Plugins.
- Install '**Prometheus metrics**' Plugin.
- Restart the Jenkins once the installation is completed.

[On Jenkins Node]

Install the **Node_exporter**

wget https://github.com/prometheus/node_exporter/releases/download/v1.7.0/node_exporter-1.7.0.linux-amd64.tar.gz

tar -xvf node_exporter-1.7.0.linux-amd64.tar.gz

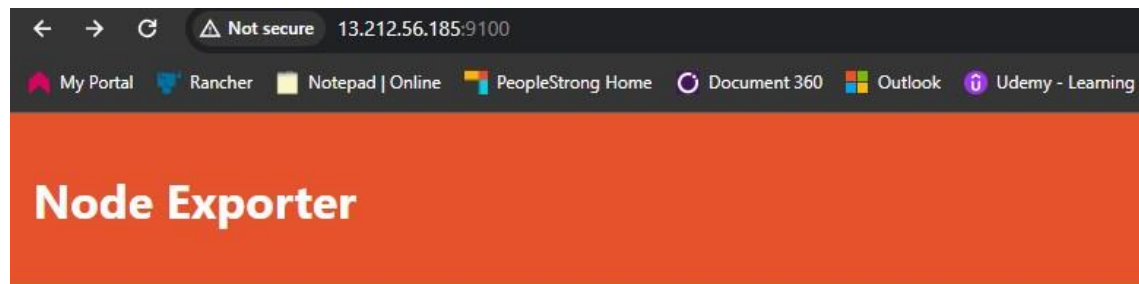
cd node_exporter-1.7.0.linux-amd64

./node_exporter & **--#To execute the file. We add &, because we want it to run this in background.**

By default **Node_exporter** is running on port **9100**. To access,

<Public-IP of the Jenkins server>:9100

Ex- 13.212.56.185:9100



Prometheus Node Exporter

Version: (version=1.7.0, branch=HEAD, revision=7333465abf9efba81876303bb57e6fadb946041b)

- [Metrics](#)

[On Monitoring Server]

Now, add the Node_exporter job in the prometheus.yml file.

vi prometheus.yml

```
- job_name: 'node_exporter'
  static_configs:
    - targets: ['13.212.56.185:9100'] #Node_exporter Url.

- job_name: 'jenkins'
  metrics_path: /prometheus
  static_configs:
    - targets: ['13.212.56.185:8080'] #Jenkins Url.
```

```

# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label 'job=<job_name>' to any timeseries scraped from this config.
  - job_name: "prometheus"

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
      - targets: ["localhost:9090"]

  - job_name: 'node_exporter'
    static_configs:
      - targets: ['13.212.56.185:9100']

  - job_name: 'jenkins'
    metrics_path: /prometheus
    static_configs:
      - targets: ['13.212.56.185:8080']

  - job_name: 'blackbox'
    metrics_path: /probe
    params:
      module: [http_2xx] # Look for a HTTP 200 response.
    static_configs:
      - targets:
          - http://prometheus.io # Target to probe with http.
          - http://52.77.215.82:30302
    relabel_configs:
      - source_labels: [__address__]
        target_label: __param_target
      - source_labels: [__param_target]
        target_label: instance
      - target_label: __address__
        replacement: 47.128.253.197:9115 # The blackbox exporter's real hostname:port.

```

pgrep prometheus

kill 13587


./prometheus &

Now Add new dashboard, i.e. for **Node_exporter** on **Grafana**.

- Copy the dashboard ID from - <https://grafana.com/grafana/dashboards/1860-node-exporter-full/>
- Paste the Dashboard ID in the Grafana then click on Load.

Find and import dashboards for common applications at grafana.com/dashboards

- Select the Data source > Import.



Home > Dashboards > Import dashboard

Published by: rrm0z

Updated on: 2024-03-06 12:03:51

Options

Name

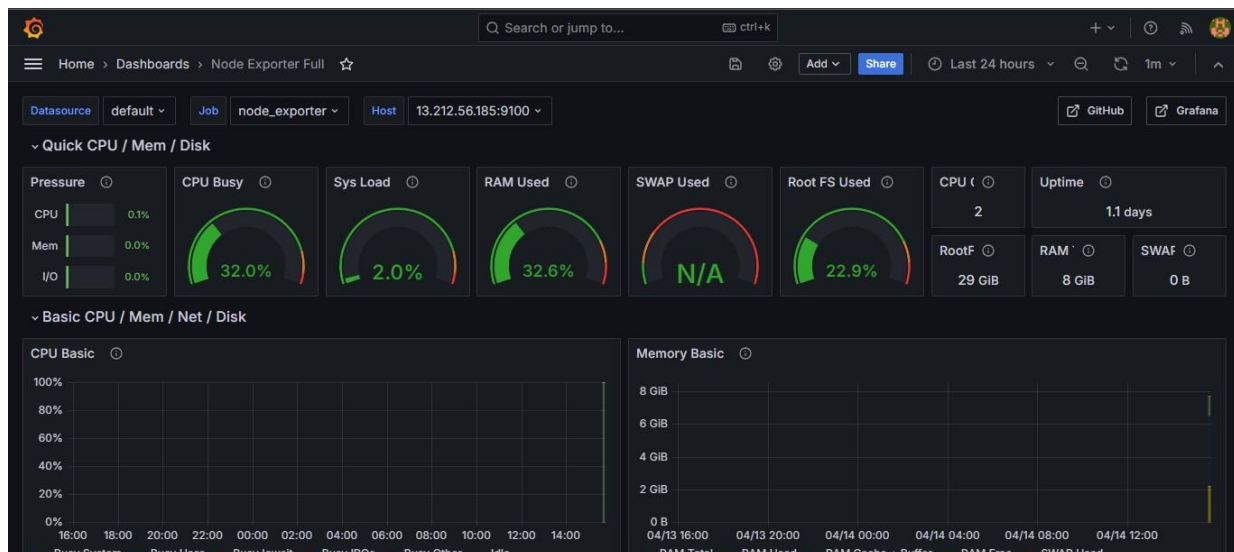
Folder

Unique identifier (UID)

The unique identifier (UID) of a dashboard can be used for uniquely identify a dashboard between multiple Grafana installs. The UID allows having consistent URLs for accessing dashboards so changing the title of a dashboard will not break any bookmarked links to that dashboard.

Prometheus

- Now the Dashboard is setup completely.



These details we are observing here are all sourced from our Jenkins machine. Now, everything we see in the dashboard, including RAM usage, system load, and more, originates from Jenkins.

Additionally, we can also access other details like network traffic and memory information, including VM stats. This comprehensive monitoring setup allows us to track various aspects effectively.

At this juncture, we've created two dashboards: one utilizes the **node exporter for monitoring Jenkins' system-level metrics**, while the other utilizes **Prometheus's blackbox exporter to monitor the website itself**.