**Phase-4**

**Project Title: *Establishing CI/CD PIPELINE FOR AUTOMATED DEPLOYMENT***

**College Name:** MARATHA MANDAL ENGINEERING COLLEGE BELGAVI

**Group Members:**

**Name:** Chandrasen D Bhosle **Name:** Varsha P Lohar

CAN ID Number: 33790524 CAN ID Number: 33829156

**Name:** Sonali S Takawade **Name:** Nikita T Vanjeri

CAN ID Number: 33835085 CAN ID Number:33833650

**1. Overview of Containerized Application Deployment**

Containerized applications are deployed in a structured manner using **CI/CD pipelines** to ensure automation, scalability, and reliability. In this process:

* **Containers** are used to package applications with all dependencies.
* **IBM Cloud Kubernetes Service (IKS)** orchestrates container deployment.
* **IBM Cloud Container Registry (ICR)** stores and manages container images.
* **CI/CD Pipelines** automate the entire deployment process, reducing manual effort and ensuring continuous integration and delivery.

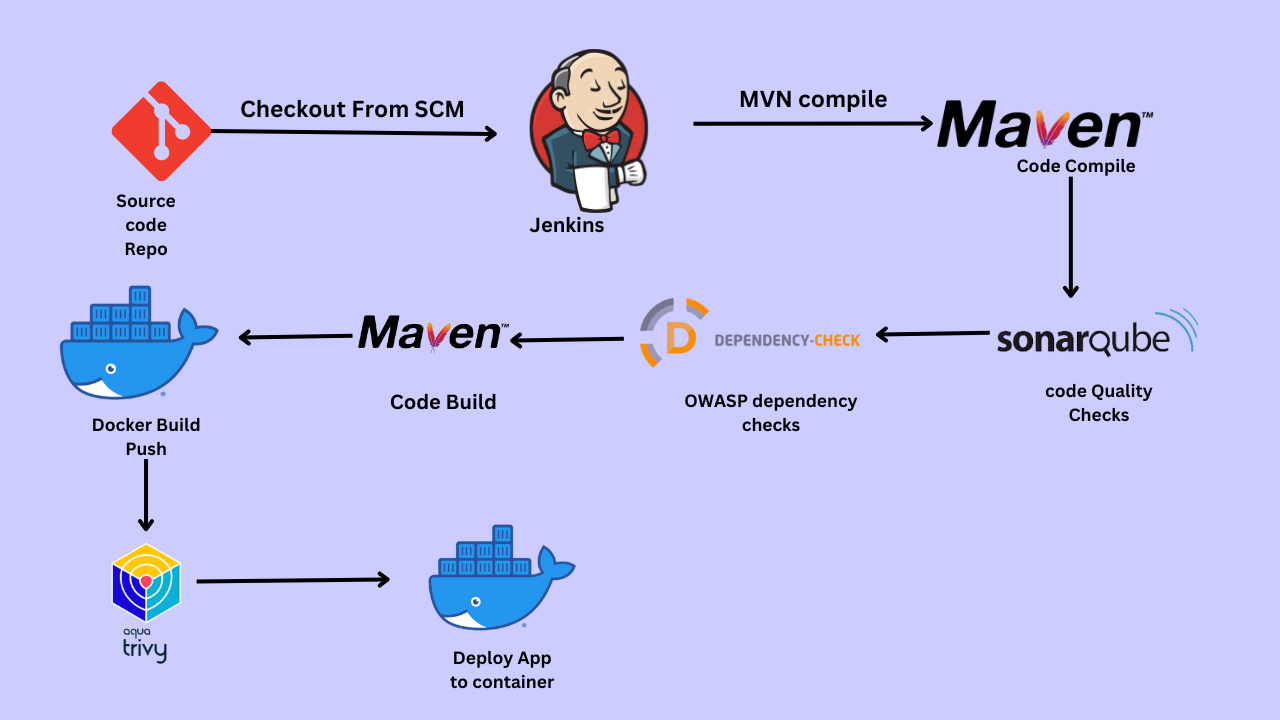
**Key Components**:

The deployment pipeline for a containerized application consists of multiple automated stages:

| **Stage** | **Description** |
| --- | --- |
| **Source Code Management** | Developers commit code changes to a repository (GitHub, GitLab, Bitbucket). |
| **Build Stage** | The CI system builds a **Docker image** using a Dockerfile. |
| **Testing Stage** | Automated tests validate the application functionality inside the container. |
| **Push to Container Registry** | The Docker image is tagged and pushed to **IBM Cloud Container Registry (ICR)**. |
| **Deployment Stage** | The containerized application is deployed to **IBM Cloud Kubernetes (IKS)**. |
| **Monitoring & Logging** | Tools like **IBM Cloud Monitoring, Prometheus, and Grafana** track application health and performance. |

∙ **Automation & CI/CD**: Automate the build, push, and deployment pipeline for rapid and consistent application deployment.

Hello friends, we will be deploying a Pet Clinic Java Based Application. This is an everyday use case scenario used by several organizations. We will be using Jenkins as a CICD tool and deploying our application on Docker container.



We will be deploying our application in two ways, using Docker Container and other is using Tomcat Server.

**Steps:-**

Step 1 — Create an Ubuntu T2 Large Instance

Step 2 — Install Jenkins, Docker and Trivy. Create a Sonarqube Container using Docker.

Step 3 — Install Plugins like JDK, Sonarqube Scanner, Maven, OWASP Dependency Check,

Step 4 — Create a Pipeline Project in Jenkins using Declarative Pipeline

Step 5 — Install OWASP Dependency Check Plugins

Step 6 — Docker Image Build and Push

Step 7 — Deploy image using Docker

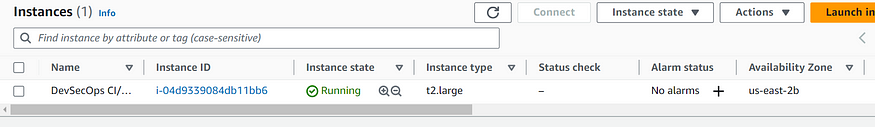
Step 8 — Access the Real World Application

Step 9 — Terminate the AWS EC2 Instance

**References**

**Now, lets get started and dig deeper into each of these steps :-**

**Step 1** — Launch an AWS T2 Large Instance. Use the image as Ubuntu. You can create a new key pair or use an existing one. Enable HTTP and HTTPS settings in the Security Group.



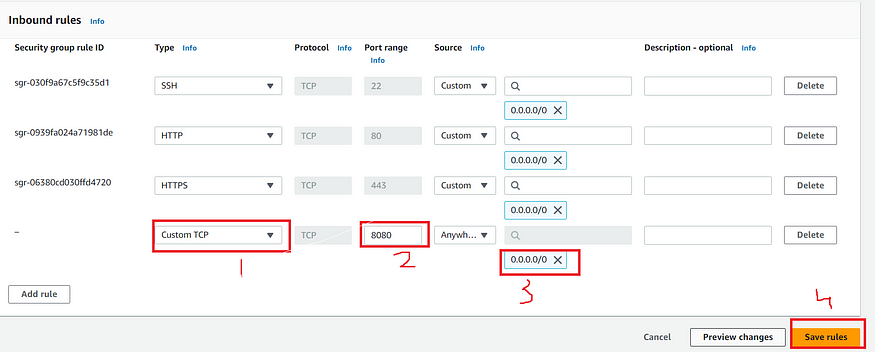
**Step 2** — Install Jenkins, Docker and Trivy

2A — To Install Jenkins

Connect to your console, and enter these commands to Install Jenkins

sudo apt-get update  
  
curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key | sudo tee \  
 /usr/share/keyrings/jenkins-keyring.asc > /dev/null  
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 update  
sudo apt install openjdk-17-jdk  
sudo apt install openjdk-17-jre  
  
sudo systemctl enable jenkins  
sudo systemctl start jenkins  
sudo systemctl status jenkins  
  
sudo cat /var/lib/jenkins/secrets/initialAdminPassword

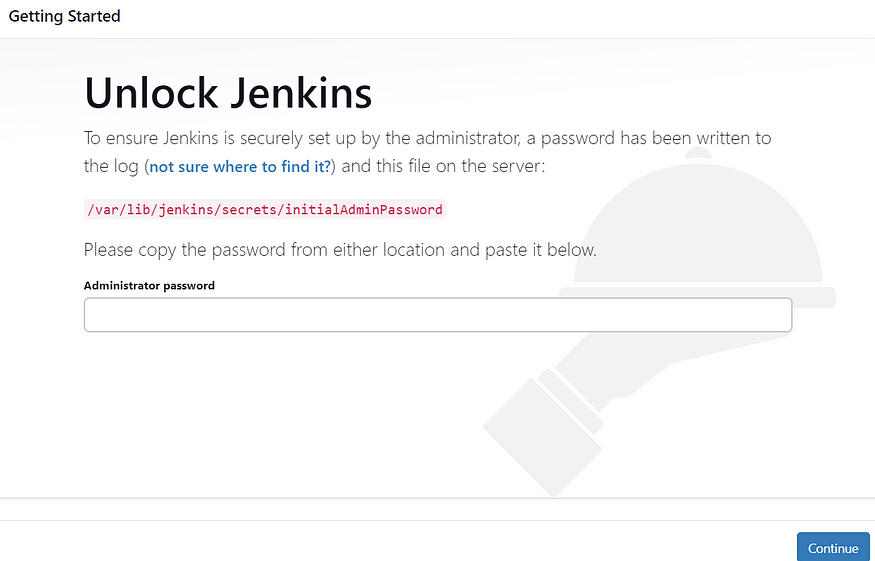
Once Jenkins is installed, you will need to go to your AWS EC2 Security Group and open Inbound Port 8080, since Jenkins works on Port 8080.



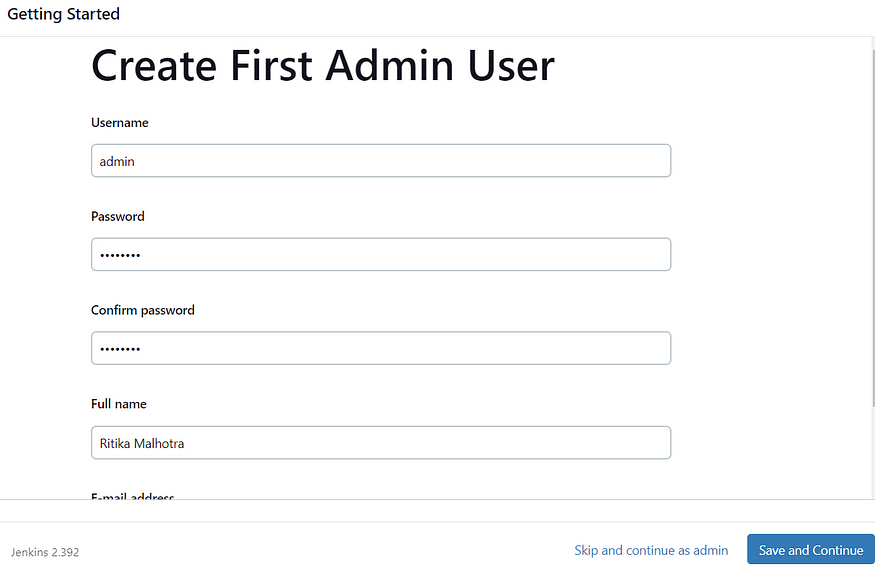
Now, grab your Public IP Address

<EC2 Public IP Address:8080>  
sudo cat /var/lib/jenkins/secrets/initialAdminPassword

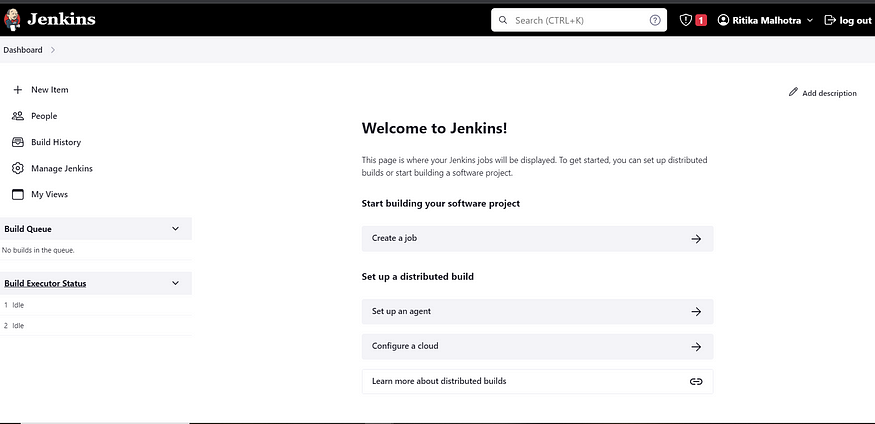
Unlock Jenkins using an administrative password and install the required plugins.



Jenkins will now get installed and install all the libraries.



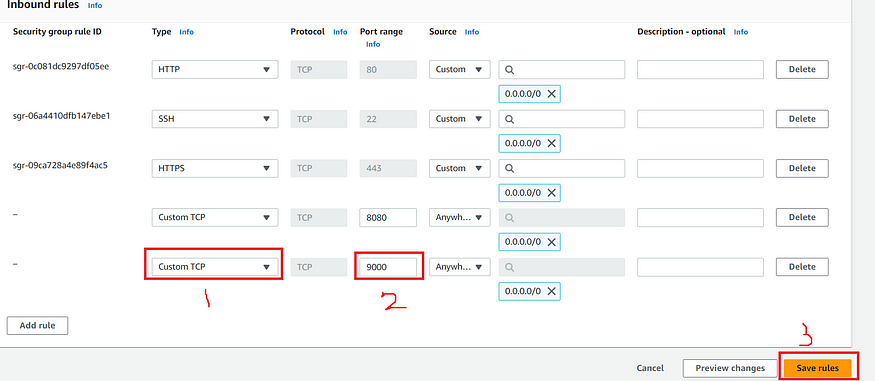
Jenkins Getting Started Screen



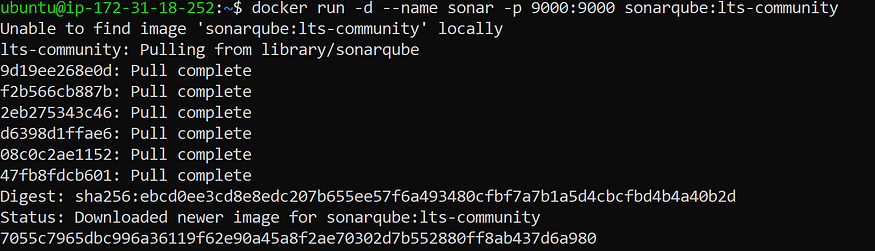
2B — Install Docker

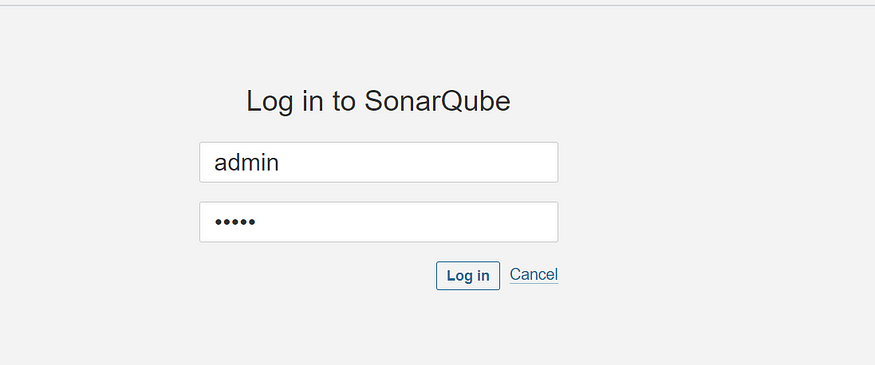
sudo apt-get update  
sudo apt-get install docker.io -y  
sudo usermod -aG docker $USER  
sudo chmod 777 /var/run/docker.sock   
sudo docker ps

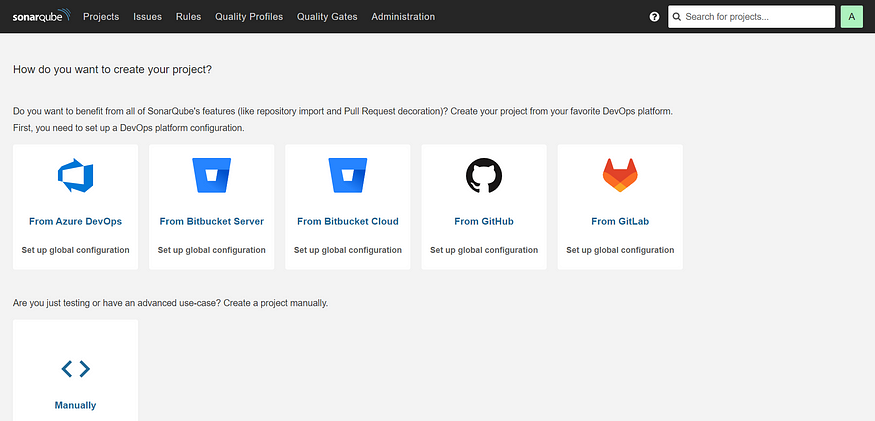
After the docker installation, we create a sonarqube container (Remember added 9000 port in the security group)



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







2C — Install Trivy

sudo apt-get install wget apt-transport-https gnupg lsb-release -y  
  
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 -y

Next, we will login to Jenkins and start to configure our Pipeline in Jenkins

**Step 3**— Install Plugins like JDK, Sonarqube Scanner, Maven, OWASP Dependency Check,

**3A — Install Plugin**

Goto Manage Jenkins →Plugins → Available Plugins →

Install below plugins

1 → Eclipse Temurin Installer (Install without restart)

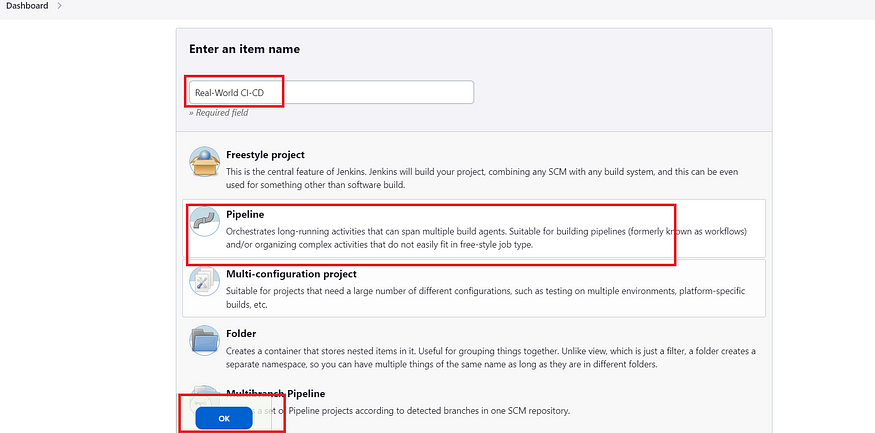
2 → SonarQube Scanner (Install without restart)

**3B — Configure Java and Maven in Global Tool Configuration**

Goto Manage Jenkins → Tools → Install JDK and Maven3 → Click on Apply and Save

**3C — Create a Job**

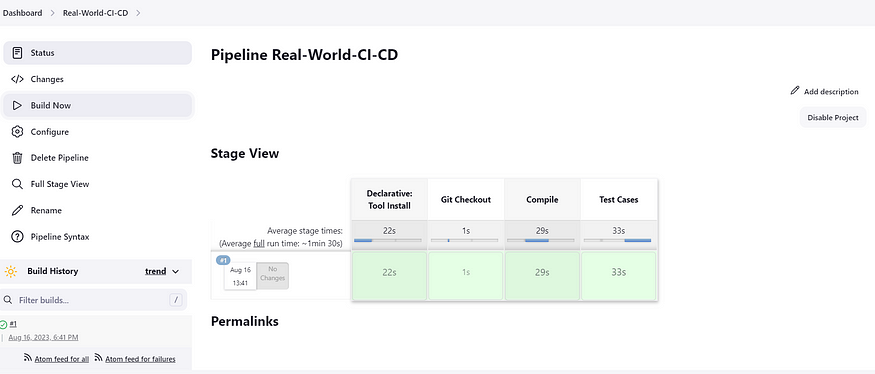
Label it as Real-World CI-CD, click on Pipeline and Ok.



Enter this in Pipeline Script,

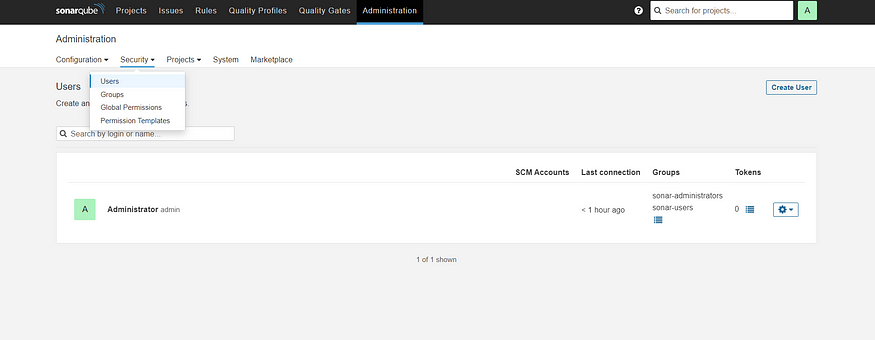
pipeline {  
 agent any   
   
 tools{  
 jdk 'jdk17'  
 maven 'maven3'  
 }  
   
 stages{  
   
 stage("Git Checkout"){  
 steps{  
 git branch: 'main', changelog: false, poll: false, url: 'https://github.com/Aj7Ay/Petclinic.git'  
 }  
 }  
   
 stage("Compile"){  
 steps{  
 sh "mvn clean compile"  
 }  
 }  
   
 }  
}

The stage view would look like this,

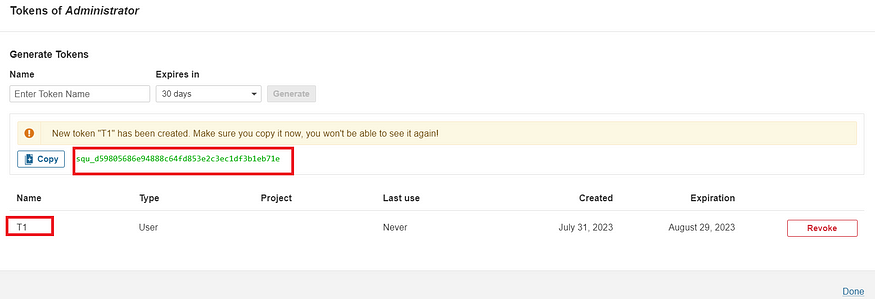


**Step 4 — Configure Sonar Server in Manage Jenkins**

Grab the Public IP Address of your EC2 Instance, Sonarqube works on Port 9000 , sp <Public IP>:9000. Goto your Sonarqube Server. Click on Administration → Security → Users → Click on Tokens and Update Token → Give it a name → and click on Generate Token

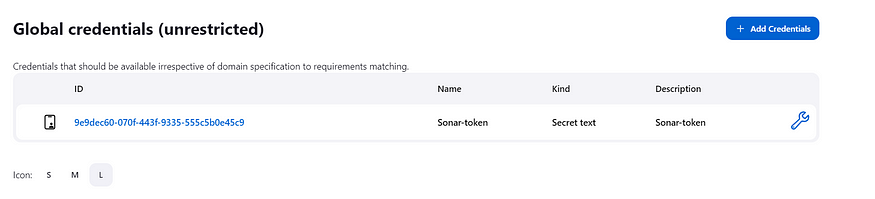


Click on Update Token



Copy this Token squ\_1fe3f7207ffb6a4860398475013b1c37a3177b53

Goto Dashboard → Manage Jenkins → Credentials → Add Secret Text. It should look like this



Now, goto Dashboard → Manage Jenkins → Configure System

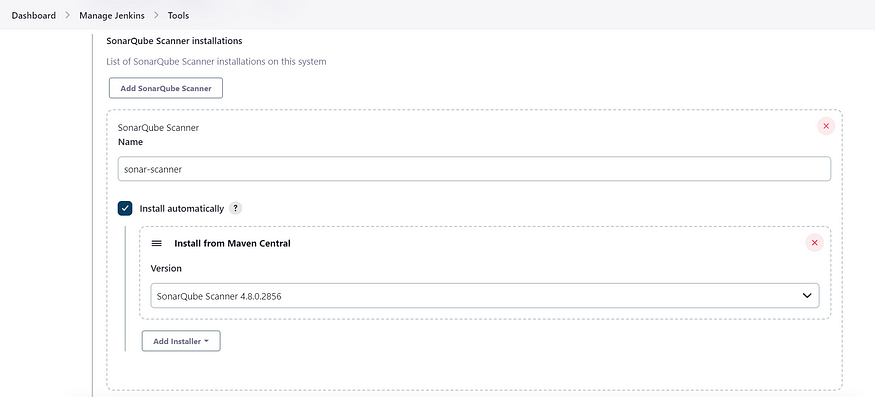


Click on Apply and Save

**Configure System option** is used in Jenkins to configure different server

**Global Tool Configuration** is used to configure different tools that we install using Plugins

We will install sonar-scanner in tools.



Lets goto our Pipeline and add Sonar-qube Stage in our Pipeline Script

pipeline {  
 agent any   
   
 tools{  
 jdk 'jdk17'  
 maven 'maven3'  
 }   
 stages{  
   
 stage("Git Checkout"){  
 steps{  
 git branch: 'main', changelog: false, poll: false, url: 'https://github.com/Aj7Ay/Petclinic.git'  
 }  
 }  
   
 stage("Compile"){  
 steps{  
 sh "mvn clean compile"  
 }  
 }

stage("Sonarqube Analysis "){

steps{

script {

withSonarQubeEnv(credentialsId: 'Sonar-token') {

sh 'mvn sonar:sonar'

}

}

}

}

stage("quality gate"){

            steps {

                script {

                  waitForQualityGate abortPipeline: false, credentialsId: 'Sonar-token'

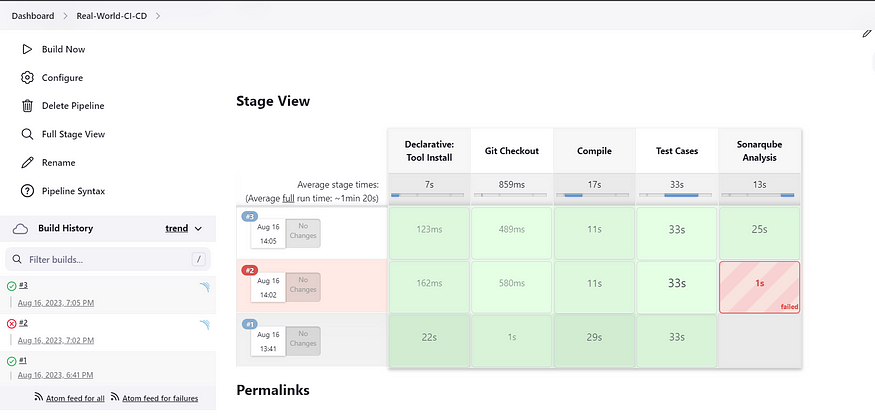
                }

           }

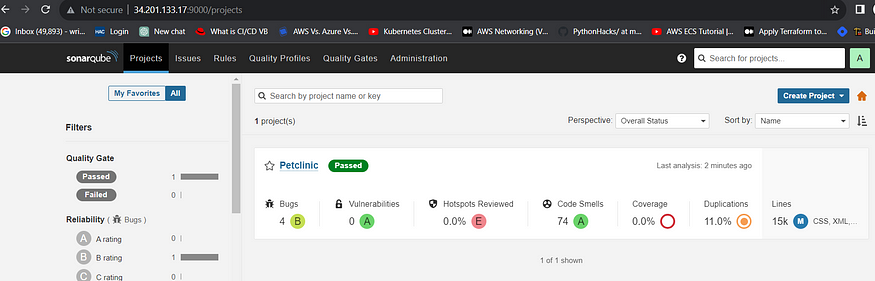
        }

}  
 }  
}

Click on Build now, you will see the stage view like this



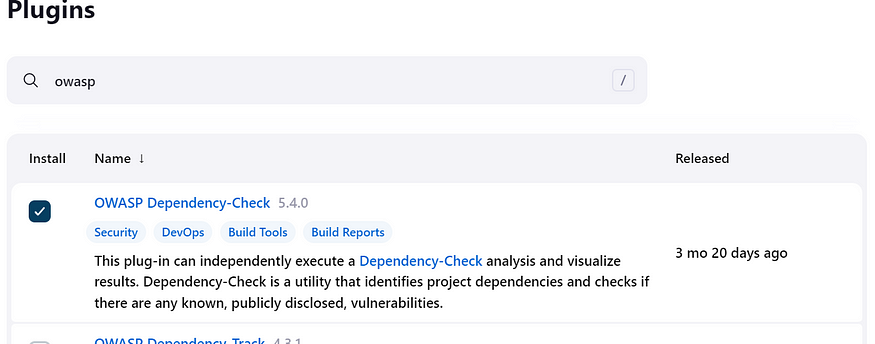
To see the report, you can goto Sonarqube Server and goto Projects.



You can see the report has been generated and the status shows as passed. You can see that there are 15K lines. To see detailed report, you can go to issues.

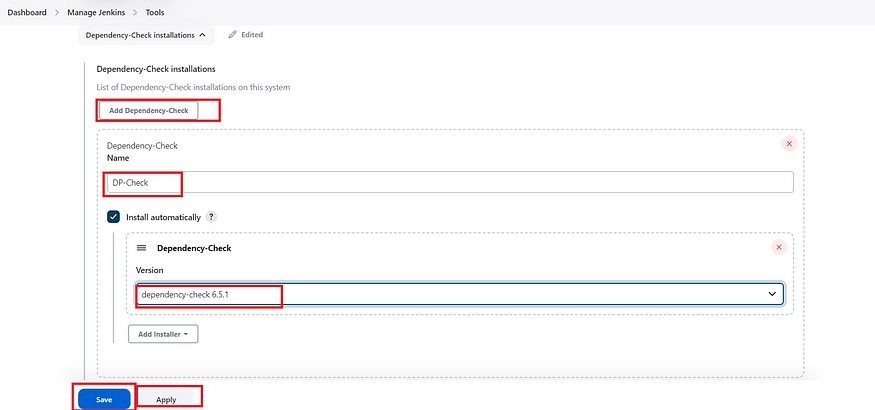
**Step 5**— Install OWASP Dependency Check Plugins

GotoDashboard → Manage Jenkins → Plugins → OWASP Dependency-Check. Click on it and install without restart.



First, we configured Plugin and next we have to configure Tool

Goto Dashboard → Manage Jenkins → Tools →



Click on apply and Save here.

Now goto configure → Pipeline and add this stage to your pipeline

stage("OWASP Dependency Check"){  
 steps{  
 dependencyCheck additionalArguments: '--scan ./ --format HTML ', odcInstallation: 'DP-Check'  
 dependencyCheckPublisher pattern: '\*\*/dependency-check-report.xml'  
 }  
 }  
stage("Build"){  
 steps{  
 sh " mvn clean install"  
 }  
 }

The final pipeline would look like this,

pipeline {  
 agent any   
   
 tools{  
 jdk 'jdk17'  
 maven 'maven3'  
 }  
   
 environment {  
 SCANNER\_HOME=tool 'sonar-scanner'  
 }  
  
 stages{  
   
 stage("Git Checkout"){  
 steps{  
 git branch: 'main', changelog: false, poll: false, url: 'https://github.com/Aj7Ay/Petclinic.git'  
 }  
 }  
   
 stage("Compile"){  
 steps{  
 sh "mvn clean compile"  
 }  
 }  
   
 stage("Test Cases"){  
 steps{  
 sh "mvn test"  
 }  
 }  
 stage("Sonarqube Analysis "){

steps{

script {

withSonarQubeEnv(credentialsId: 'Sonar-token') {

sh 'mvn sonar:sonar'

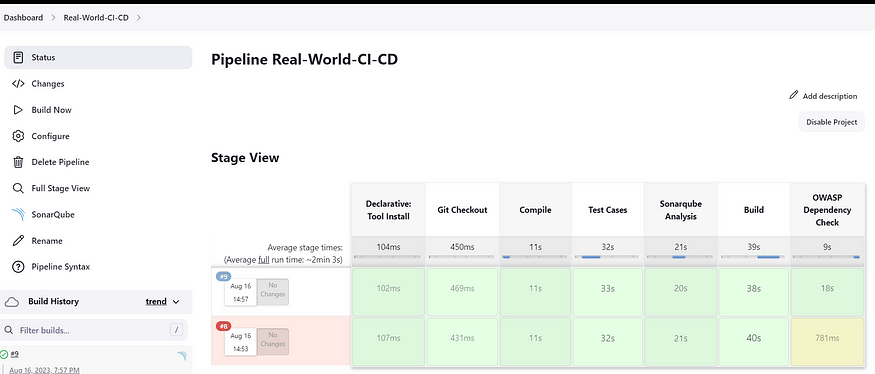
}

}

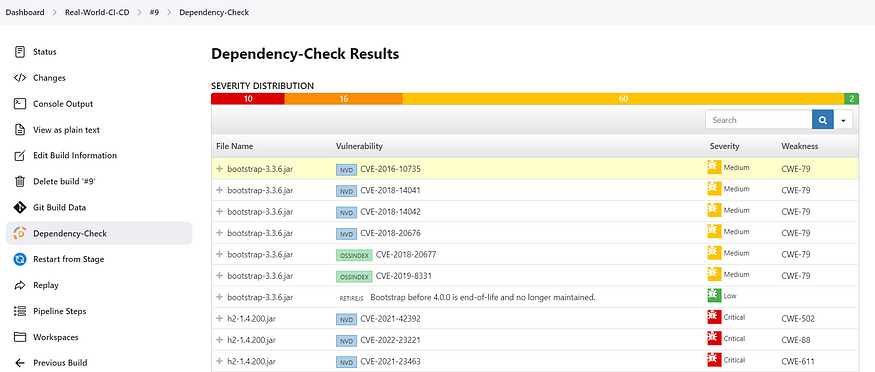
}

}  
   
 stage("Build"){  
 steps{  
 sh " mvn clean install"  
 }  
 }  
   
 stage("OWASP Dependency Check"){  
 steps{  
 dependencyCheck additionalArguments: '--scan ./ ' , odcInstallation: 'DP-Check'  
 dependencyCheckPublisher pattern: '\*\*/dependency-check-report.xml'  
 }  
 }  
   
 }  
}

The stage view would look like this,



You will see that in status, a graph will also be generated



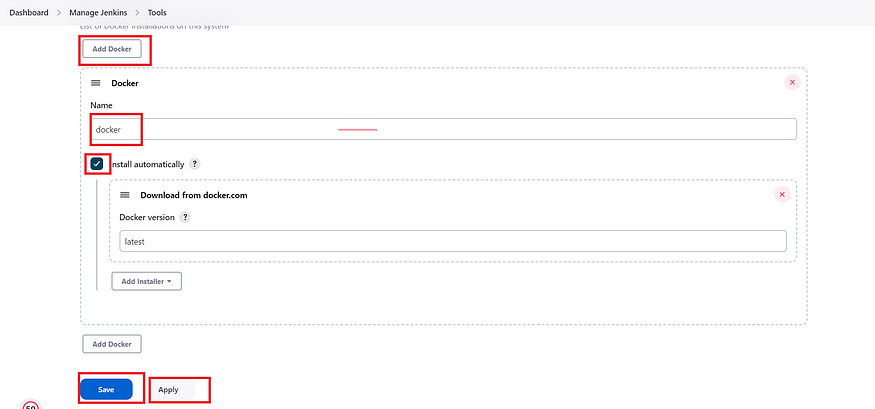
**Step 6** — Docker Image Build and Push

We need to install Docker tool in our system, Goto Dashboard → Manage Plugins → Available plugins → Search for Docker and install these plugins

* Docker
* Docker Commons
* Docker Pipeline
* Docker API
* docker-build-step

and click on install without restart

Now, goto Dashboard → Manage Jenkins → Tools →



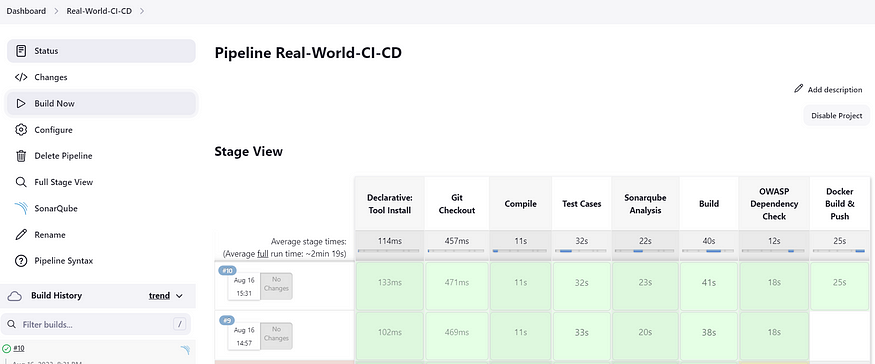
Add DockerHub Username and Password under Global Credentials



Add this stage in Pipeline Script

stage("Docker Build & Push"){  
 steps{  
 script{  
 withDockerRegistry(credentialsId: ''bc86df08-bacf-4695-99cb-8cefb3406235', toolName: 'docker') {  
   
 sh "docker build -t petclinic1 ."  
 sh "docker tag petclinic1 sevenajay/pet-clinic123:latest "  
 sh "docker push sevenajay/pet-clinic123:latest "  
   
 }  
 }  
 }  
 }

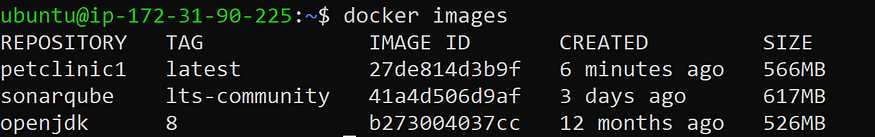
You will see the output like below,



Now, when you do

docker images

You will see this output



When you log in to Dockerhub, you will see a new image is created



**Step 7** — Deploy image using Docker

Add this stage to your pipeline syntax

stage("TRIVY"){

steps{

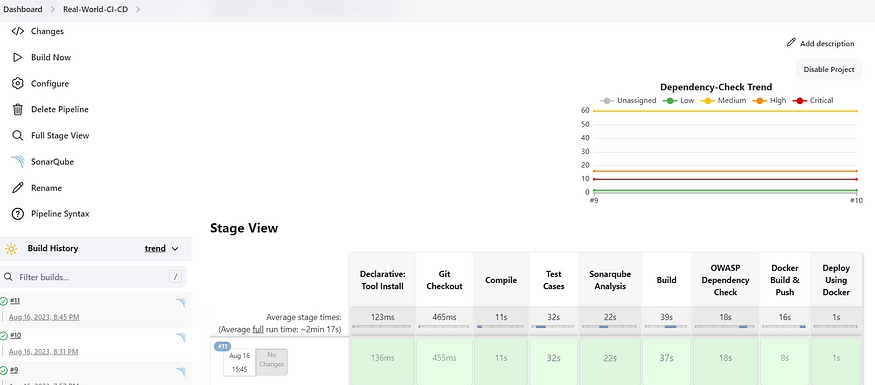
sh "trivy image sevenajay/pet-clinic123:latest"

}

}

stage("Deploy Using Docker"){  
 steps{  
 sh " docker run -d --name pet1 -p 8082:8080 sevenajay/pet-clinic123:latest "  
 }  
 }

You will see the Stage View like this,



**Step 8**— Terminate the AWS EC2 Instance

Lastly, do not forget to terminate the AWS EC2 Instance.

Complete pipeline

pipeline{

    agent any

    tools {

        jdk 'jdk11'

        maven 'maven3'

    }

    stages{

        stage ('clean Workspace'){

            steps{

                cleanWs()

            }

        }

        stage ('checkout scm') {

            steps {

                checkout scmGit(branches: [[name: '\*/master']], extensions: [], userRemoteConfigs: [[url: 'https://github.com/Aj7Ay/amazon-eks-jenkins-terraform-aj7.git']])

            }

        }

        stage ('maven compile') {

            steps {

                sh 'mvn clean compile'

            }

        }

        stage ('sonarqube Analysis'){

            steps{

                script{

                    withSonarQubeEnv(credentialsId: 'Sonar-token') {

                      sh 'mvn sonar:sonar'

                    }

                }

            }

        }

        stage("quality gate"){

            steps {

                script {

                  waitForQualityGate abortPipeline: false, credentialsId: 'Sonar-token'

                }

           }

        }

        stage("OWASP Dependency Check"){

            steps{

                dependencyCheck additionalArguments: '--scan ./ --format HTML ', odcInstallation: 'DP-Check'

                dependencyCheckPublisher pattern: '\*\*/dependency-check-report.xml'

            }

        }

        stage ('Build war file'){

            steps{

                sh 'mvn clean install package'

            }

        }

        stage ('Build and push to docker hub'){

            steps{

                script{

                    withDockerRegistry(credentialsId: 'docker', toolName: 'docker') {

                        sh "docker build -t petclinic1 ."

                        sh "docker tag petclinic1 sevenajay/pet-clinic123:latest"

                        sh "docker push sevenajay/pet-clinic123:latest"

                   }

                }

            }

        }

        stage("TRIVY"){

            steps{

                sh "trivy image sevenajay/pet-clinic123:latest"

            }

        }

        stage ('Deploy to container'){

            steps{

                sh 'docker run -d --name pet1 -p 8082:8080 sevenajay/pet-clinic123:latest'

            }

        }

    }

}

**Conclusion**

Establishing a CI/CD pipeline for automated deployment streamlines the software development process by ensuring faster, more reliable, and efficient code delivery. It automates testing, integration, and deployment, reducing manual errors and enhancing software quality. By implementing CI/CD, teams can achieve continuous feedback, quicker iterations, and improved collaboration, ultimately leading to a more robust and scalable deployment process.

**Further Enhancements**

1. Security Integration (DevSecOps) – Implement security checks within the pipeline to detect vulnerabilities early.
2. Infrastructure as Code (IaC) – Automate infrastructure provisioning using tools like Terraform or AWS CloudFormation.
3. Performance Testing – Incorporate performance and load testing to ensure the system handles real-world scenarios effectively.
4. AI-Powered Automation – Use AI/ML to optimize deployment strategies, detect anomalies, and predict failures.
5. Multi-Cloud Deployment – Expand the pipeline to support deployment across multiple cloud providers for better availability.
6. Monitoring and Logging – Enhance observability by integrating monitoring tools (Prometheus, Grafana) and logging systems (ELK Stack).
7. Canary Deployments & Blue-Green Deployments – Implement advanced deployment strategies to minimize downtime and risk.
8. Self-Healing Pipelines – Automate rollback mechanisms and failure recovery to maintain system stability.

**YOUR GITHUB LINK:**