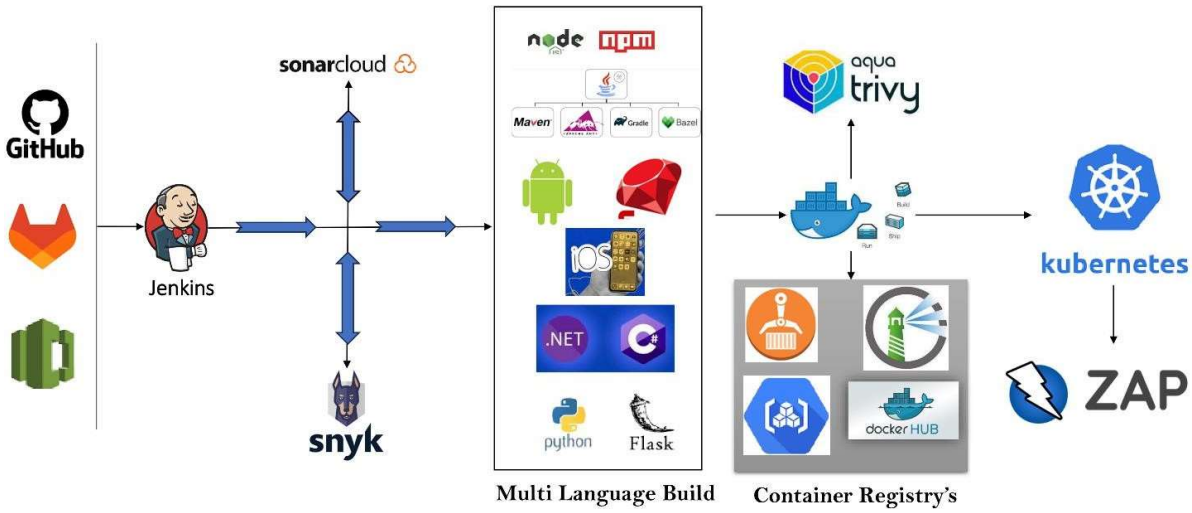
DEVSECOPS PIPELINE: AUTOMATING CI/CD PIPELINE FOR SECURE MULTI-LANGUAGE APPLICATIONS USING JENKINS



Prerequisites:

1. Git
2. Jenkins
3. Sonar-Scanner
4. Snyk
5. Application - Python, Java, Maven, Node etc... (The language you choose for your project depends on which one needs to be installed.)
6. Docker
7. Aqua Trivy
8. Kubernetes
9. Zaproxy

Gitlab: https://gitlab.com/ganesharavind124/AnaCart

Introduction:

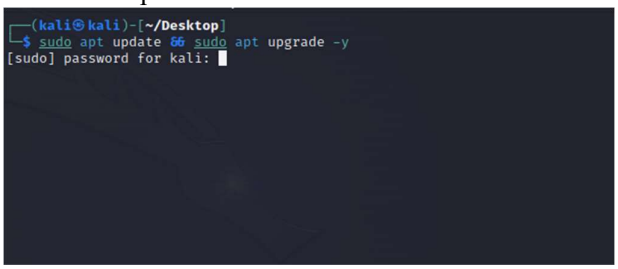
In today's fast-paced software development landscape, implementing efficient CI/CD pipelines is essential. This blog outlines the journey of building a robust CI/CD pipeline using Jenkins, integrating various tools to achieve seamless automation, security, and deployment for multi-language applications.

Setting the Stage:

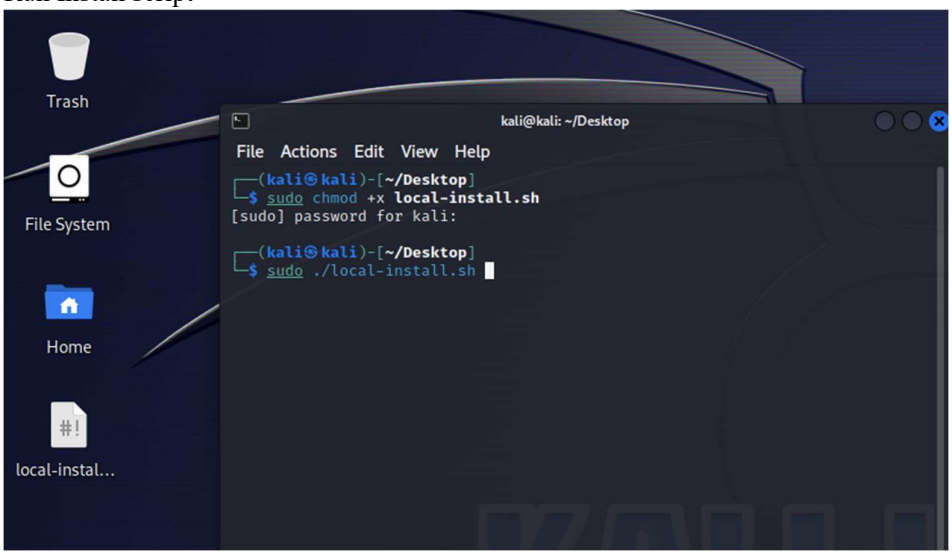
This project involved orchestrating a CI/CD pipeline that encompassed Git, SonarCloud, Synk, multi-language build automation, Docker, Aqua Trivy, Kubernetes, and ZAP Proxy. Leveraging Jenkins' flexibility and Groovy scripting capabilities, these tools are streamlined into a cohesive pipeline.

Local Linux Installation:

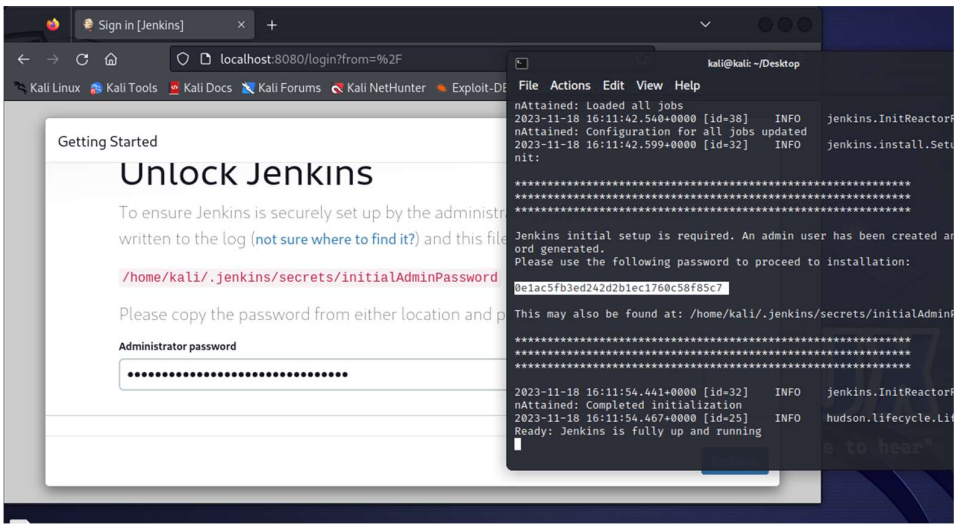
1. Install and update Linux machine. For this demonstration, I am using Kali Linux



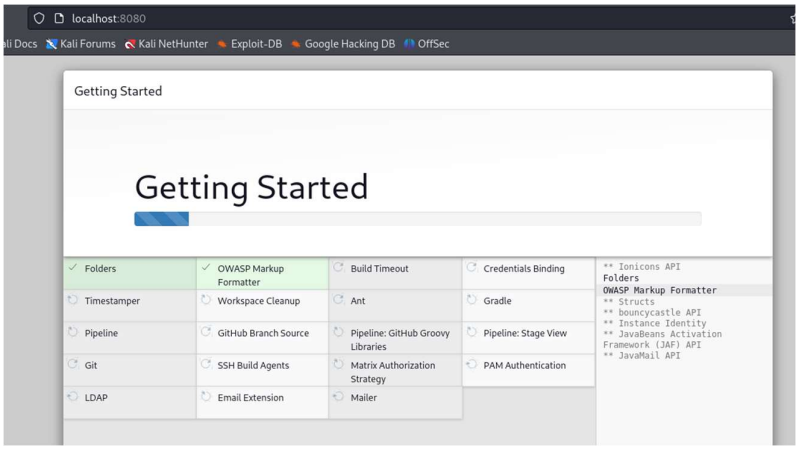
2. Download local-install.sh to Linux machine and place on Desktop
3. Make script executable
4. Run Install script



5. Run command ‘jenkins’ to start Jenkins and retrieve initial password
6. Log in to Jenkins with password that appears on screen by visiting http://localhost:8080
    a. Keep this window open! The script simply executes the command Jenkins at the end which starts the Jenkins server process
    b. If password does not appear or you close window by mistake just run cat for the location given after running the the ‘jenkins’ command
    c. Paste password into ‘Administrator Password’ field



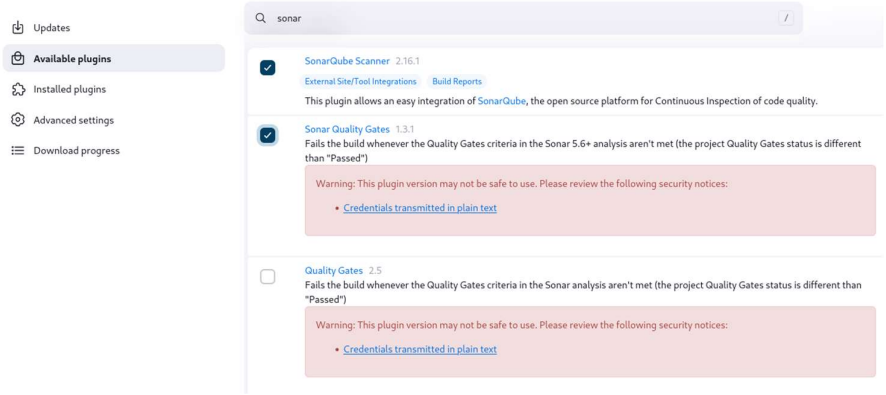
7. Click install suggested plugins and wait for installations to complete:



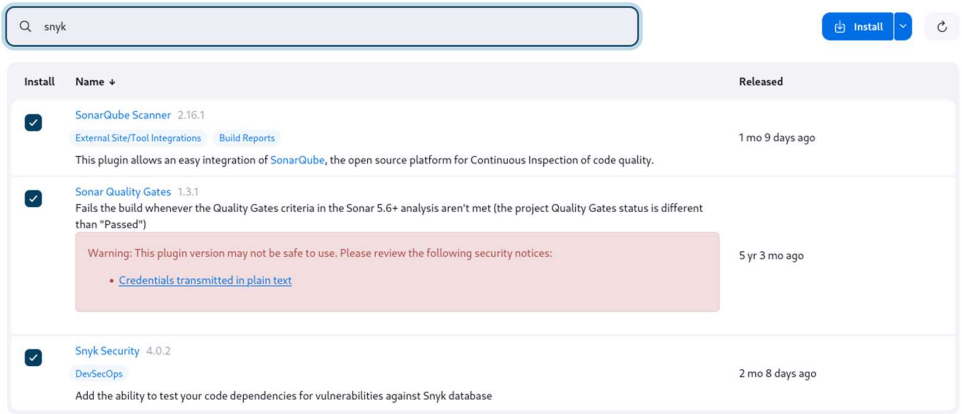
8. Create admin user credentials or use defaults for testing

Installing Plugins for the pipeline

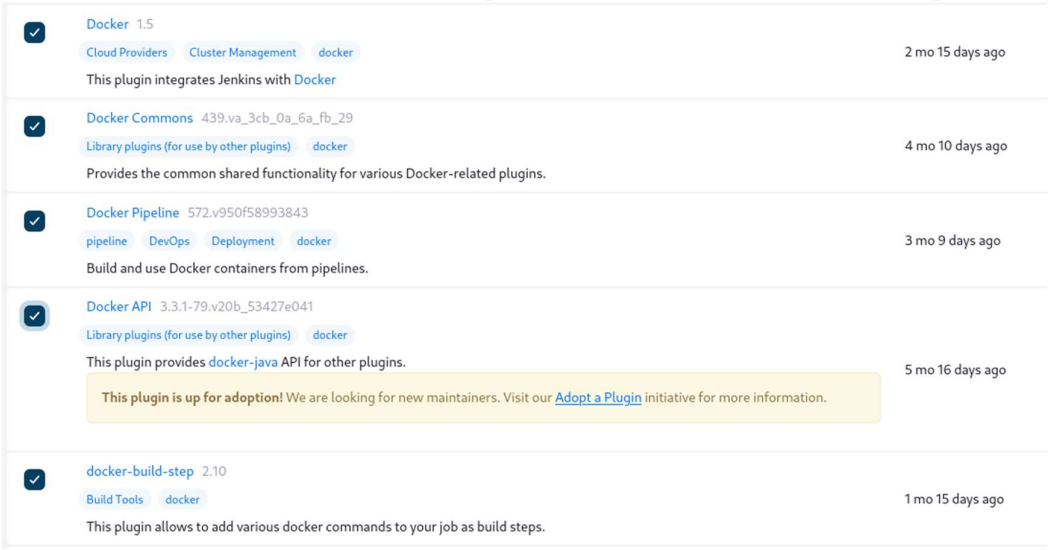
1. From the dashboard go to Manage Jenkins> Plugins> Available Plugins
2. Select SonarQube Scanner and Sonar Quality Gates for installation



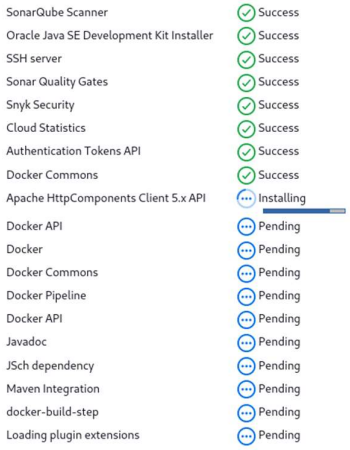
3. Select Snyk Security for installation



4. Select Docker, Docker commons, Docker Pipeline, Docker API and docker-build-step for installation



5. Ensure that all of the selected Sonar, Docker, and Snyk dependencies are installed (Success) before moving on to next step

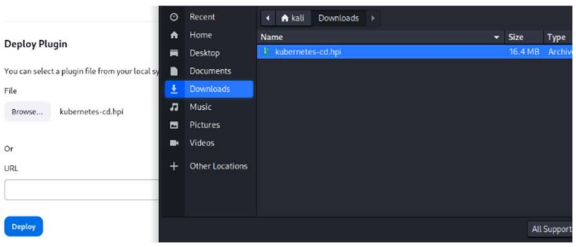


6. Install Kubernetes-cd manually for Kubernetes (this is missing from Jenkins plugin list)
    a. You can use the plugin file from the repo at https://github.com/Martian1337/BugFlow

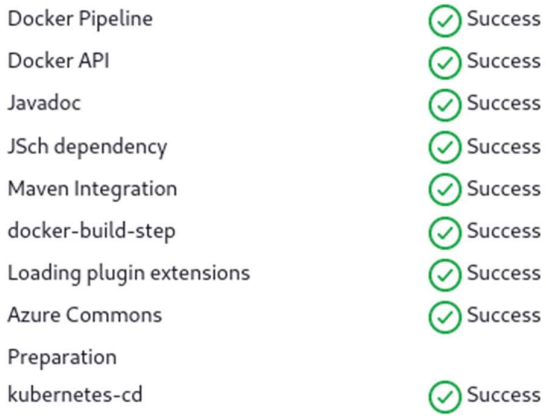
    OR

    b. You can also Download from https://updates.jenkins.io/download/plugins/kubernetes-cd/1.0.0/kubernetes-cd.hpi

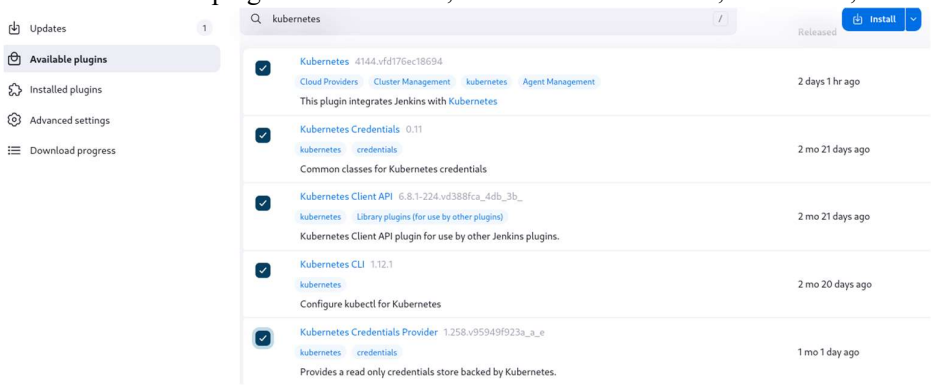
    c.
    d. Go to Manage Jenkins>Plugins>advanced settings
    e. Navigate to Deploy plugin section
    f. Click browse and select downloaded Kubernetes-cd .hpi file
    g. Click Deploy



h. Confirm plugin installed successfully



7. Install Kubernetes plugins – Kubernetes, Kubernetes Credentials, Client API, Kubernetes CLI

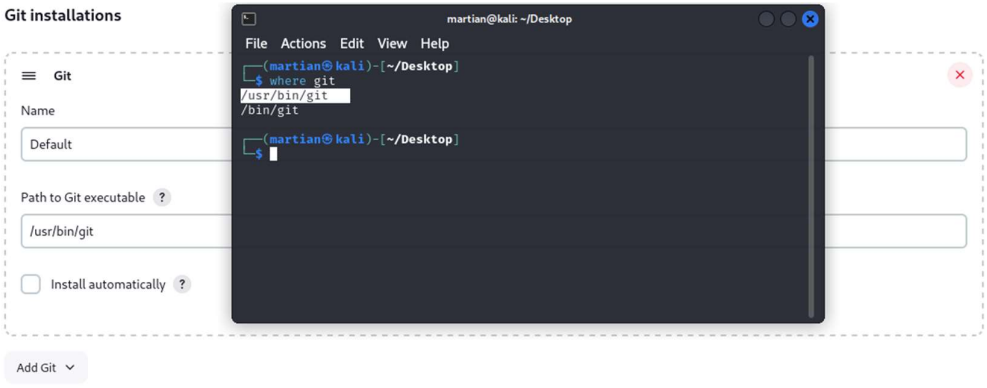


8. Ensure Kubernetes plugins are installed successfully

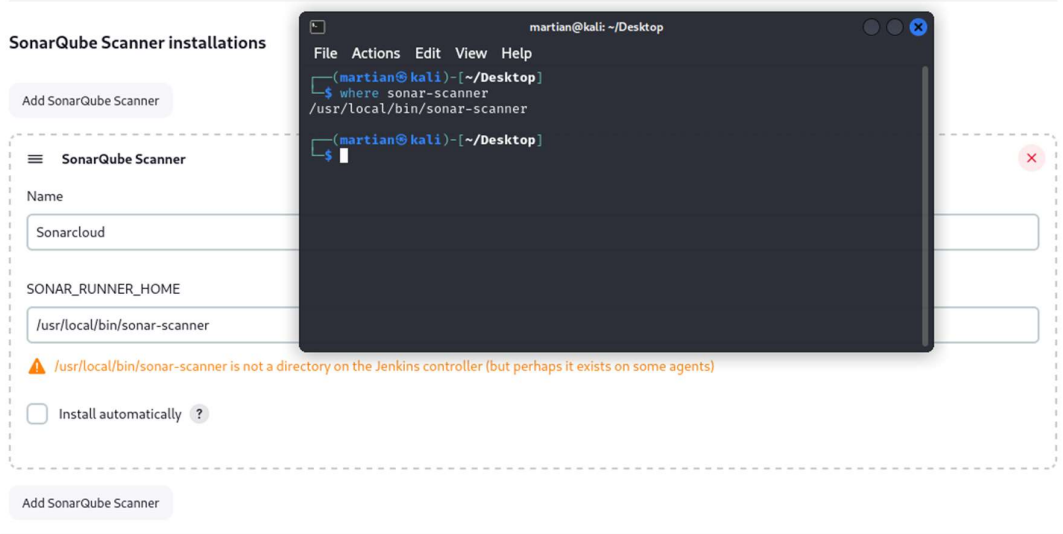
Configure Jenkins Tools

1. Navigate to Manage>Tools to configure the tools installed via the install script
2. Configure the name and path for:
    a. JDK

    b. Git

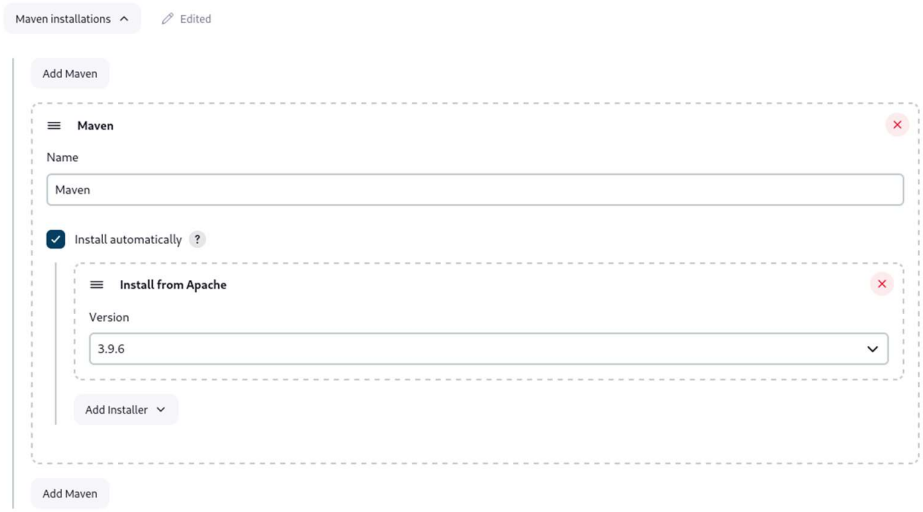


    c. SonarQube scanner (sonar-scanner)



d. Snyk

e. Maven



f. Docker

Note: Versions may vary so ensure that the tool and environment variable match accordingly

Clone/fork the target repo

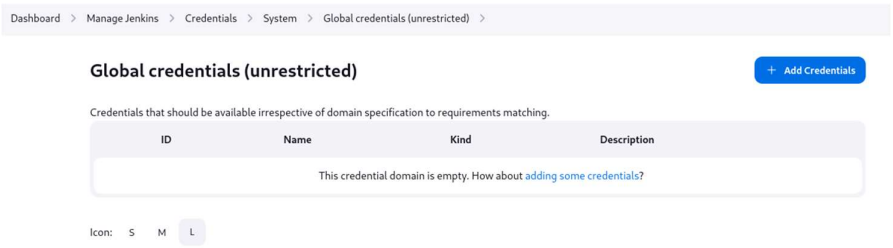
1. Go to the repo with the source code to be tested
2. Clone/fork the repo into your own Github/Gitlab

3. Add/configure a deployment.yaml file for the docker instance to be deployed

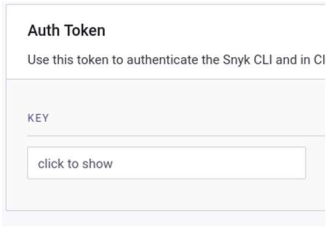
   Note: For this demo I am using an app from the OWASP Vulnerable Applications Directory

Configure Credentials

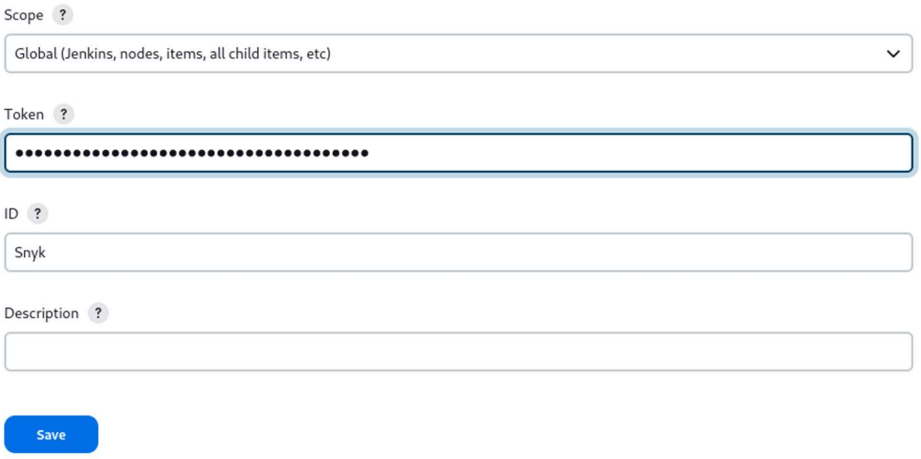
1. Go to Manage Jenkins>Credentials>Click Global under domains for System store
2. Click Add credentials



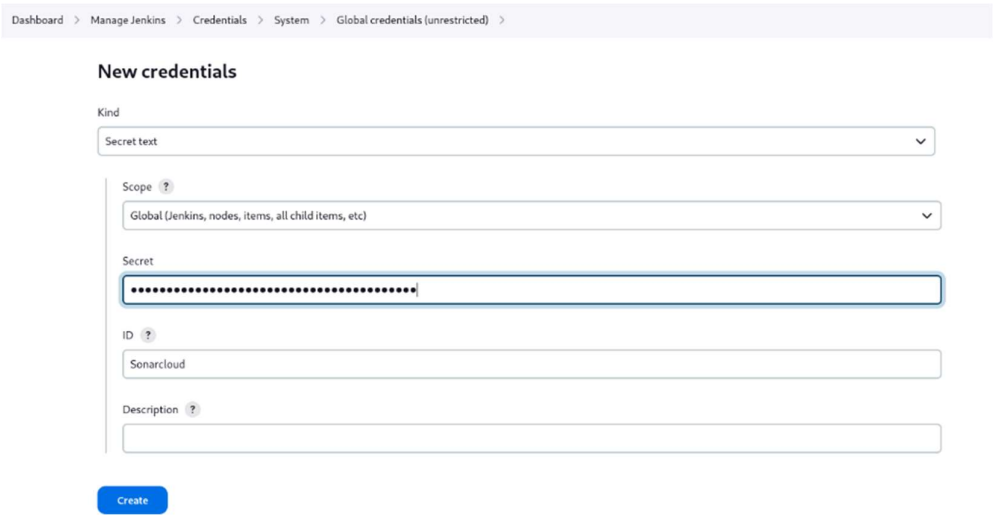
3. Add Snyk credentials
   a. Login to snyk account https://snyk.io and click skip for now
   b. Go to Snyk account settings and generate auth token
   c. Copy auth token



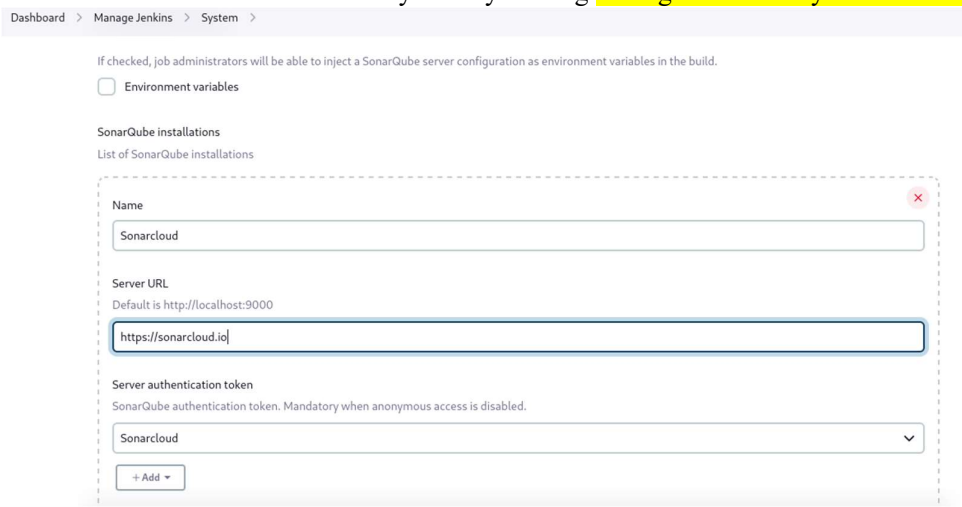
   d. In Jenkins, select dropdown and look for Snyk API token on the menu
   e. Paste auth token into Token field
   f. Name the token “Snyk” as written in the pipeline script
   g. Click Create



4. Add Sonarcloud credentials
    a. Login to Sonarcloud and start a new project
    b. Copy generated token
    c. Select ‘Secret text’ from dropdown menu
    d. Paste Sonarcloud token into ‘Secret field
    e. Name this token Sonarcloud



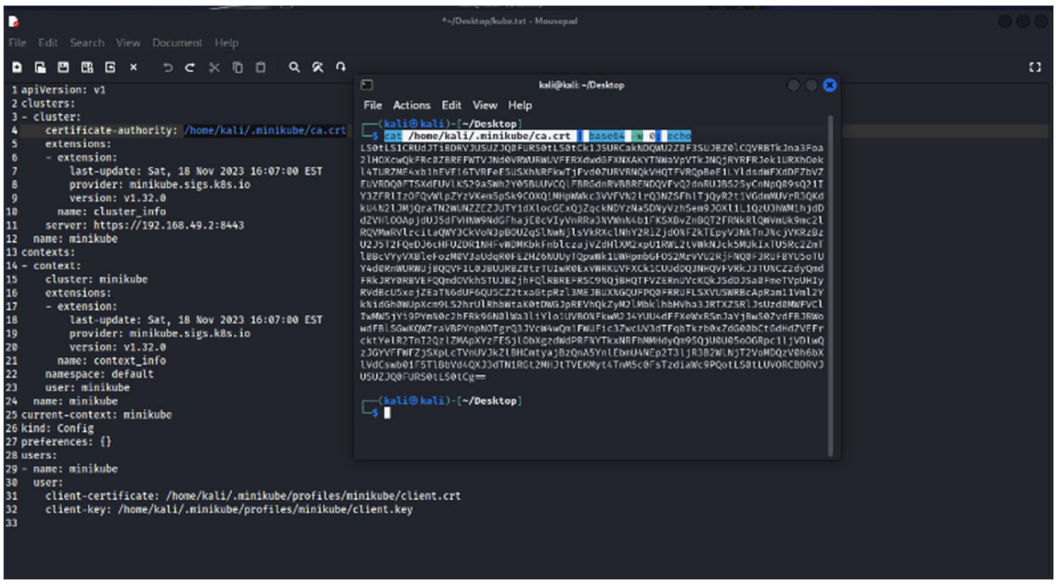
    f. Add Sonarcloud URL as Server to system by visiting Manage Jenkins> System> SonarQube Servers



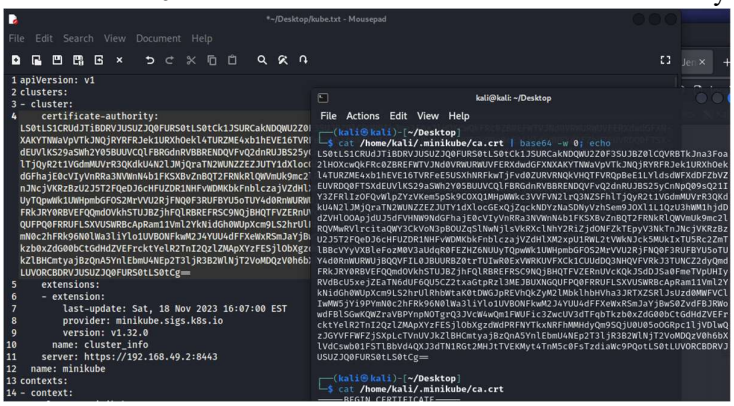
5. Add credentials for Docker
    a. Copy Docker username/password used for docker account
    b. In Jenkins, user/pass credentials into appropriate fields
    c. Name the credentials “Docker_Server” in the ID field as written in the pipeline script
    d. Click Create

Add Kubernetes kubeconfig to Jenkins

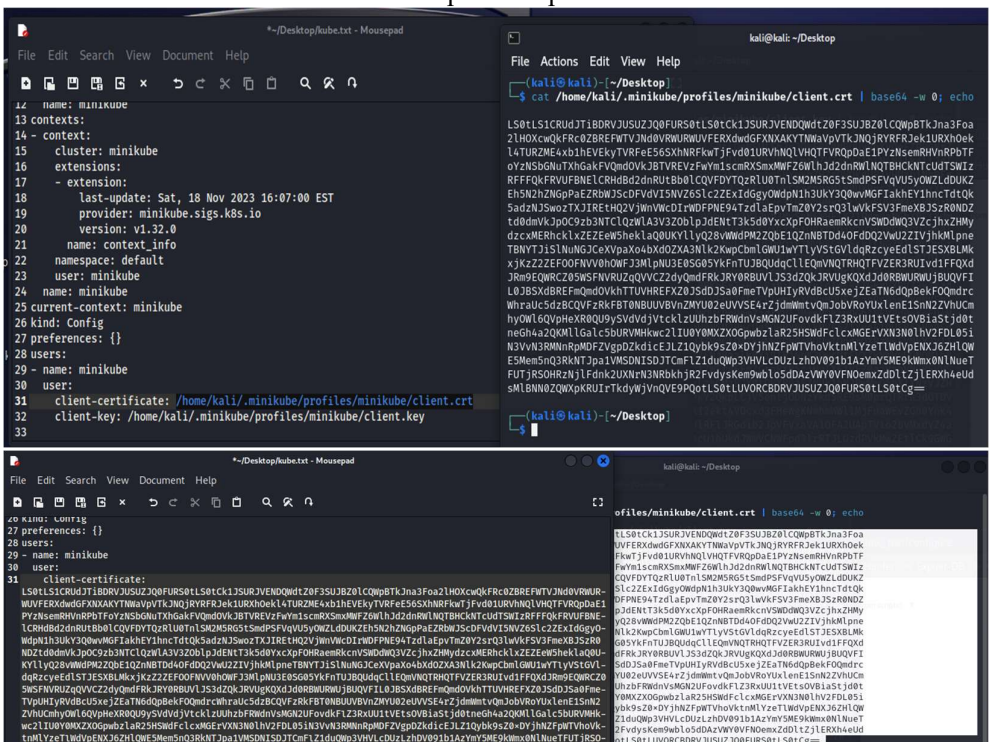
    e. Start minikube with ‘sudo usermod -aG docker $USER && newgrp docker && minikube start’ so that the config file can be retrieved
    f. Run command ‘cat ~/.kube/config > kubeconfig.txt’ from the Linux Desktop
    g. Open kubeconfig.txt with text editor
    h. Display the certificate in the path in the ‘certificate-authority’ field and turn the certificate into base64.
    You can do this in one line by typing as ‘cat /path/to/certificate | base64 -w 0; echo’ shown here:



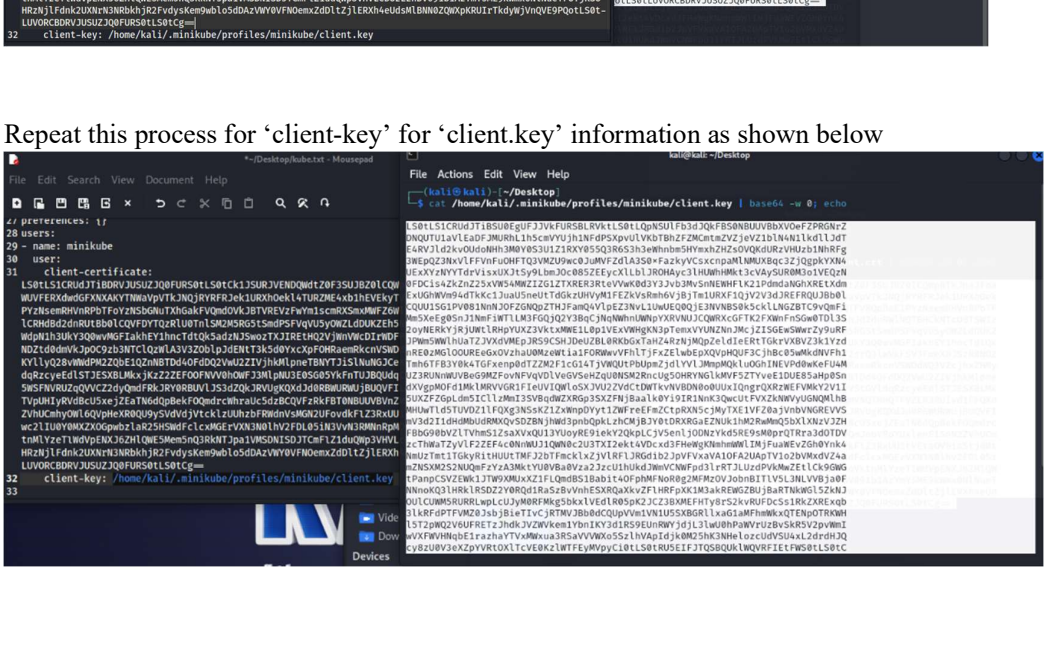
i. Paste the base64 encoded certificate into the ‘certificate-authority’ kubeconfig.txt as shown below

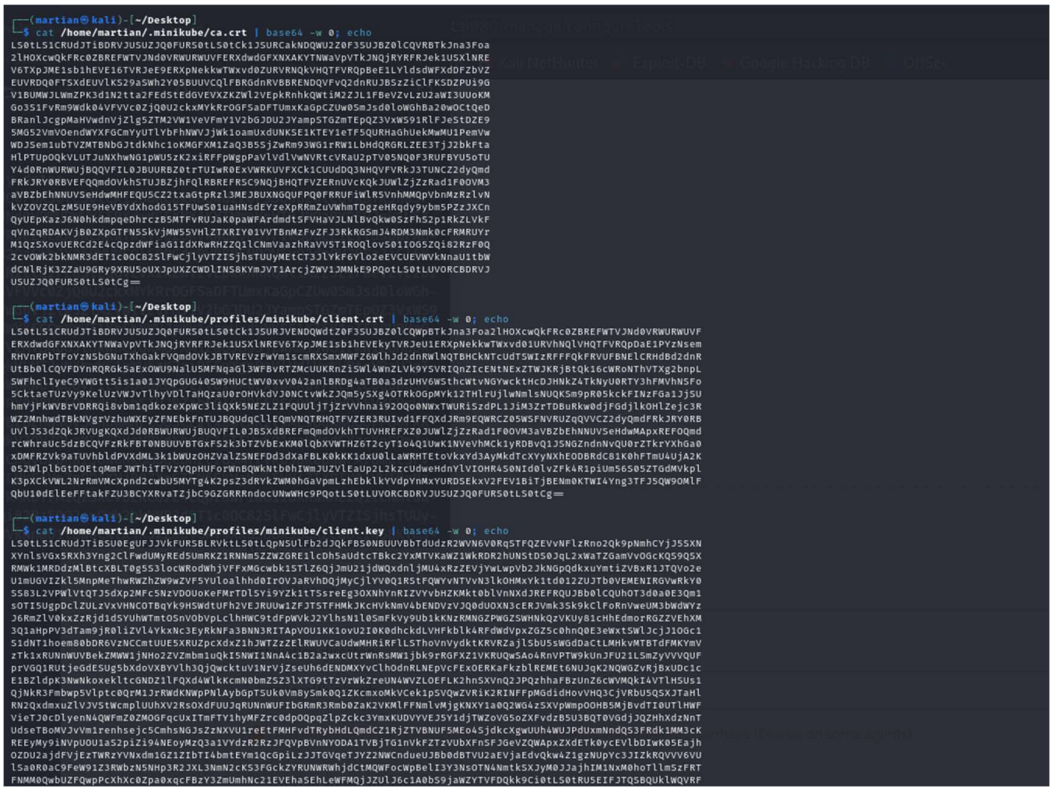


j. Scroll down to ‘client-certificate’ and repeat that process for ‘client.crt’ information as shown below

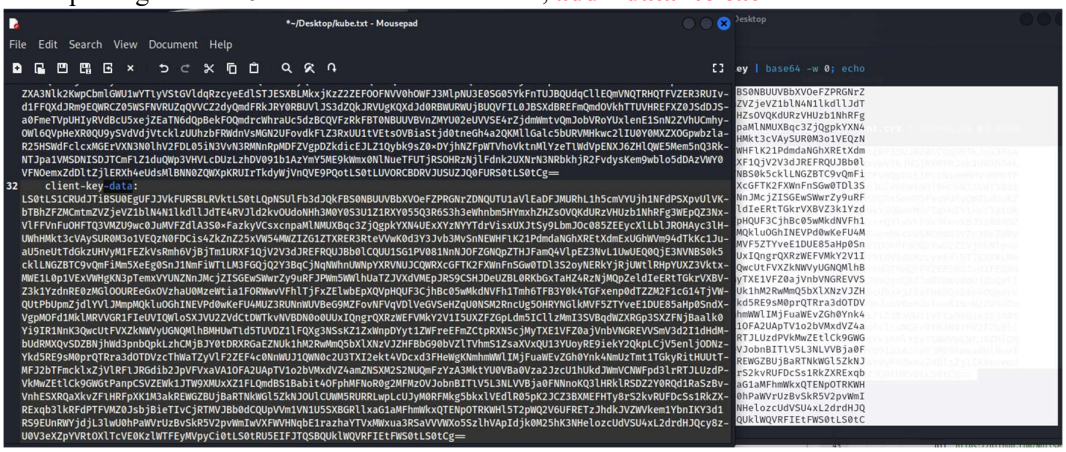


k. Repeat this process for ‘client-key’ for ‘client.key’ information as shown below





l. After pasting the base64 encoded certificate data, **add '-data' to each** field like below:



m. Now go to Jenkins, create credential for 'kubernetes'
n. Paste the new kubeconfig contents directly into the kubeconfig
o. add '-data' to the client-authority, client-certificate, and client-key fields
p. Name it 'kubernetes' and click create

The final kube contents in Jenkins should look like the following:

Certificate-authority-data:

Client-certificate-data and client-key-data:

6. Confirm that all credentials are added to match the picture below:

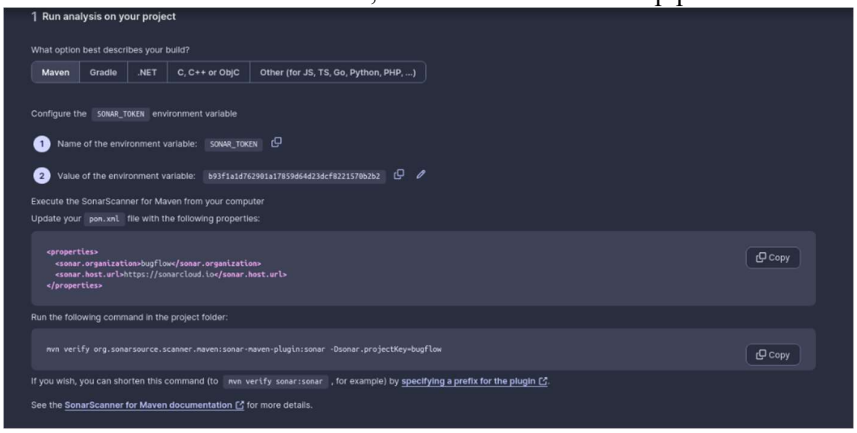
Global credentials (unrestricted)

+ Add Credentials

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

ID	Name	Kind	Description
Docker_Server	martian1337/******	Username with password	
Snyk	Snyk	Snyk API token	
kubernetes	kubernetes	Kubernetes configuration (kubeconfig)	
Sonarcloud	Sonarcloud	Secret text	

7. Add SonarCloud API key to project
    a. Log in to Sonarcloud and go to “My Account”
    b. In Sonarcloud, go to “My Projects” page and configure the project
    c. In the “Choose your analysis method” section, select your CI tool. For this application, Maven is being used.
    d. Follow the instructions on screen, to add Sonar scan to the pipeline



    e. Add the generated token to the pipeline script for continuity

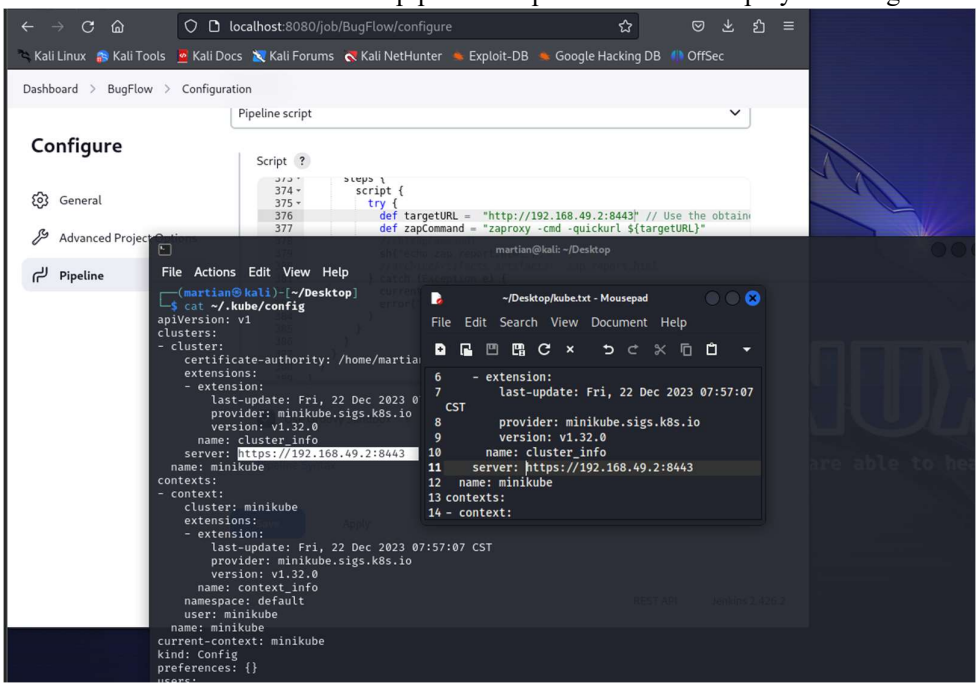
```
stage('Compile and Run Sonar Analysis') {
    steps {
        script {
            withSonarQubeEnv(credentialsId: SONARCLOUD, installationName: 'Sonarcloud') {
                try {
                    if (fileExists('pom.xml')) {
                        sh 'mvn verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectkey=bugflow'
                    } else if (fileExists('package.json')) {
                        sh "${sonarscanner} -Dsonar.organization=jenkeen -Dsonar.projectKey=jenkeen_testjs -Dsonar.sources=. -Dsonar.host.url=https://sonarcloud.io -Dsonar.login=b93f1a1d762901a17859d64d23dcf8221570b2b2"
                    } else if (fileExists('go.mod')) {
                        sh "${sonarscanner} -Dsonar.organization=jenkeen -Dsonar.projectKey=jenkeen_go -Dsonar.sources=. -Dsonar.host.url=https://sonarcloud.io -Dsonar.login=b93f1a1d762901a17859d64d23dcf8221570b2b2"
                    } else if (fileExists('Gemfile')) {
                        sh "${sonarscanner} -Dsonar.organization=jenkeen -Dsonar.projectKey=jenkeen_ruby -Dsonar.sources=. -Dsonar.host.url=https://sonarcloud.io -Dsonar.login=b93f1a1d762901a17859d64d23dcf8221570b2b2"
                    } else if (fileExists('requirements.txt')) {
                        sh "${sonarscanner} -Dsonar.organization=jenkeen -Dsonar.projectKey=jenkeen_python -Dsonar.sources=. -Dsonar.host.url=https://sonarcloud.io -Dsonar.login=b93f1a1d762901a17859d64d23dcf8221570b2b2"
                    } else {
                        currentBuild.result = 'FAILURE'
                        pipelineError = true
                        error("Unsupported application type: No compatible build steps available.")
                    }
                } catch (Exception e) {
                    currentBuild.result = 'FAILURE'
                    pipelineError = true
                    error("Error during Sonar analysis: ${e.message}")
                }
            }
        }
    }
}
```

f. After updating pom.xml (Maven), it should now look like this in the repo when complete:

```
<properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <easybuggy-port>8080</easybuggy-port>
        <!-- default properties (JDK 1.7 or earlier) -->
        <jvm.args.perm.size>-XX:MaxPermSize=128m</jvm.args.perm.size>
        <jvm.args.gc.log.path>-Xloggc:logs/gc.log</jvm.args.gc.log.path>
        <jvm.args.print.heap.at.gc>-XX:+PrintHeapAtGC</jvm.args.print.heap.at.gc>
        <jvm.args.print.gc.details>-XX:+PrintGCDetails</jvm.args.print.gc.details>
        <jvm.args.print.gc.date.stamps>-XX:+PrintGCDateStamps</jvm.args.print.gc.date.stamps>
        <jvm.args.gc.log.file.rotation>-XX:+UseGCLogFileRotation</jvm.args.gc.log.file.rotation>
        <jvm.args.number.of.gc.log.files>-XX:NumberOfGCLogFiles=5</jvm.args.number.of.gc.log.files>
        <jvm.args.gc.log.file.size>-XX:GCLogFileSize=10M</jvm.args.gc.log.file.size>
        <sonar.organization>bugflow</sonar.organization>
        <sonar.host.url>https://sonarcloud.io</sonar.host.url>
</properties>
```

Local Kubernetes Deployment config

1. Run ‘cat ~/.kube/config’ or open the created kubeconfig.txt to see the server URL
2. Paste the service URL into pipeline script’s Kubernetes Deployment stage



Pipeline Config Methods:

Go to the configuration page in the pipeline job. This page will open. Add your Jenkins pipeline script path there. There are two options available.

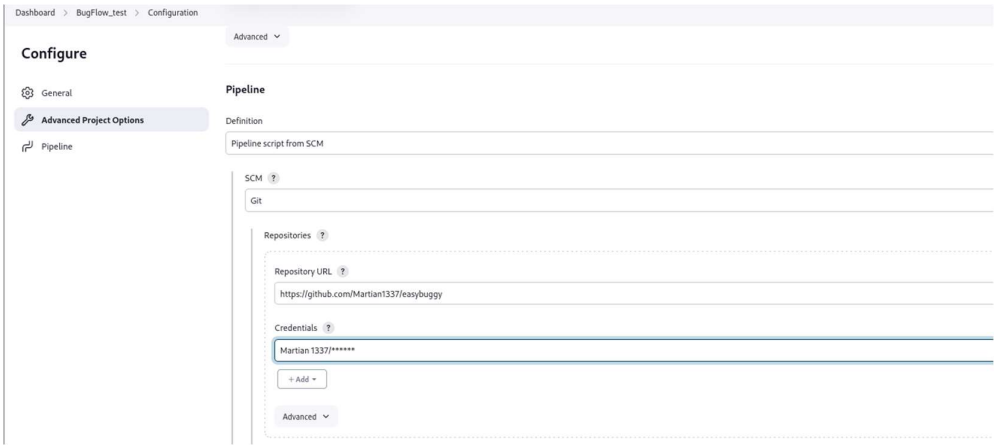
1. Script for pipeline: Here is where you can write your own script.
2. Pipeline from SCM: it will use your SCM repository's Jenkins file.

Note: Method 2 requires that you have the Jenkins file in the same repo as

the app.

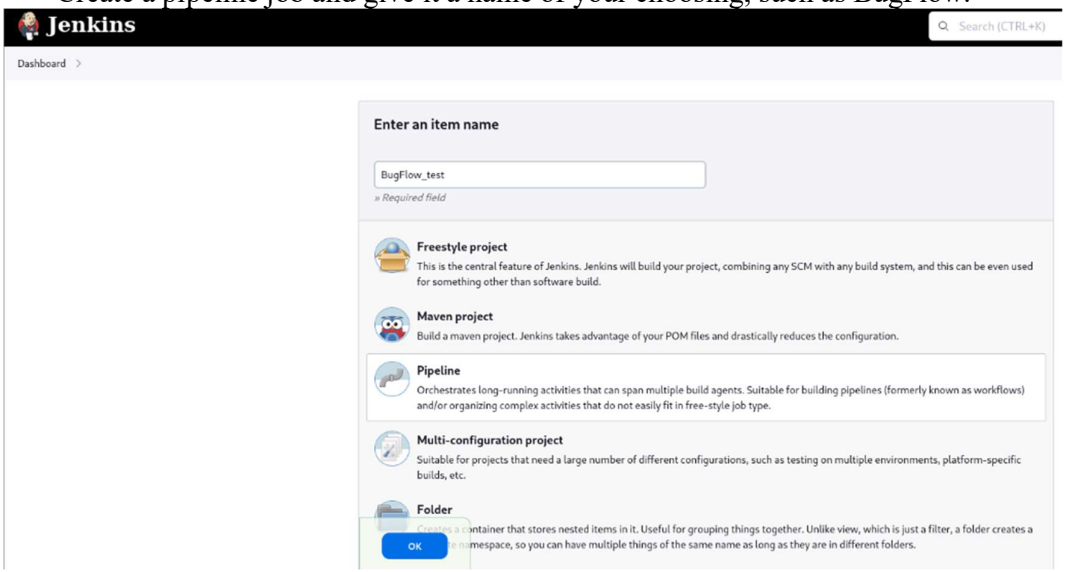
Here is an example of the first option pasted directly pipeline:

Here is an example of the second option:

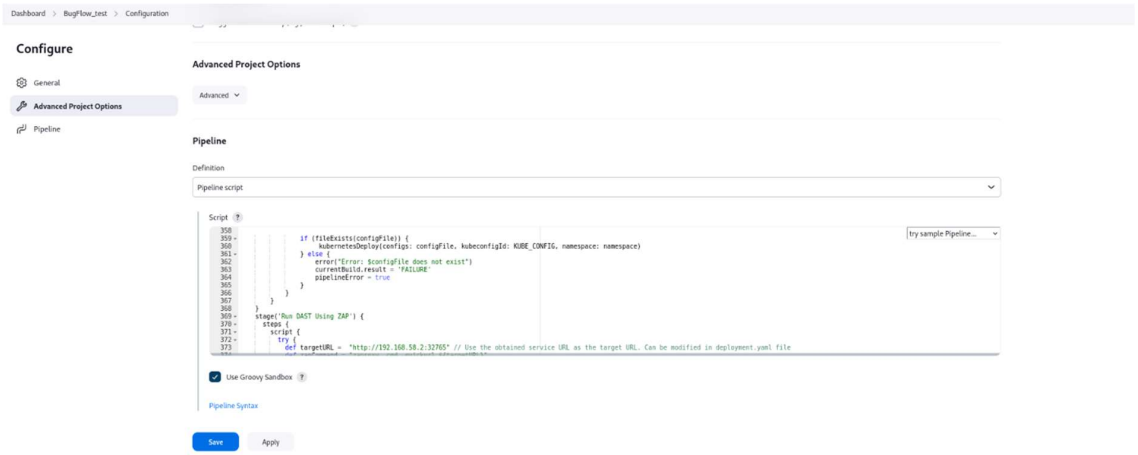


So, choose your SCM and give your branch and URL of your repository and also mention your Jenkinsfile in script path

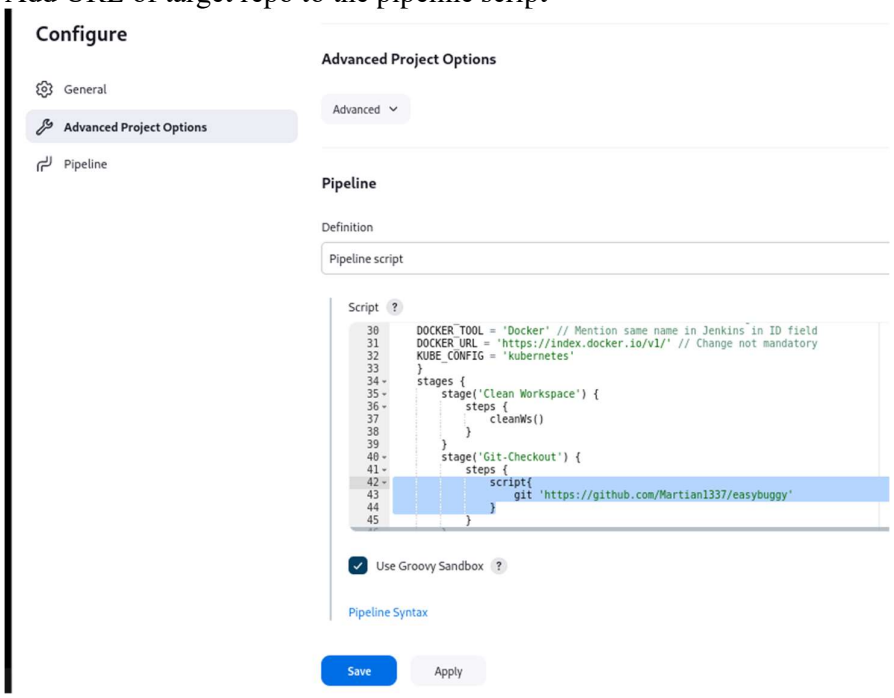
Create a pipeline job and give it a name of your choosing, such as BugFlow.



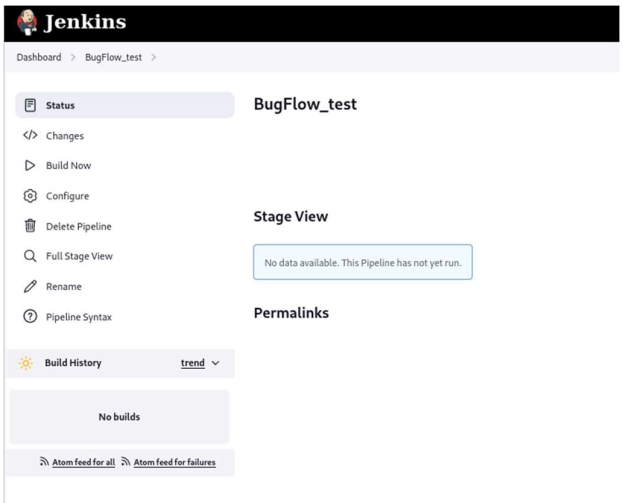
After clicking Ok, this is where the pipeline method can be selected; Script or SCM. Here I pasted the contents of the Jenkins file directly into the text box for “pipeline script”



Add URL of target repo to the pipeline script



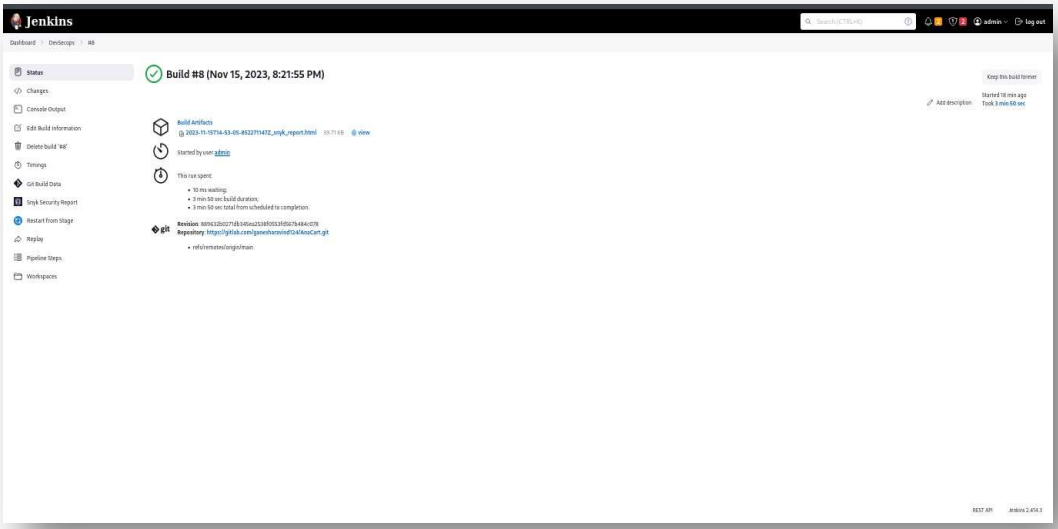
After creating the pipeline job will look like this:



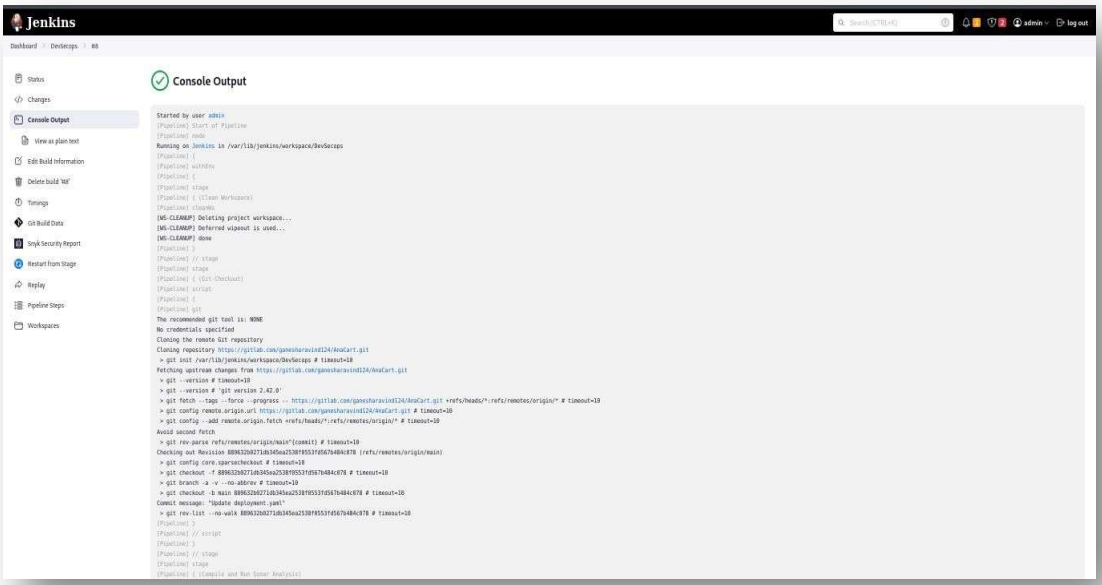
Check all of the lines, curl braces, and credentials before saving and applying. You should

also make sure that the variable names in the environment and the stages are the same, as many people make mistakes in this specific area. Next, click "Apply." If you run into any problems, an X will appear in that line. If you change "Save," the page will redirect to the main site.

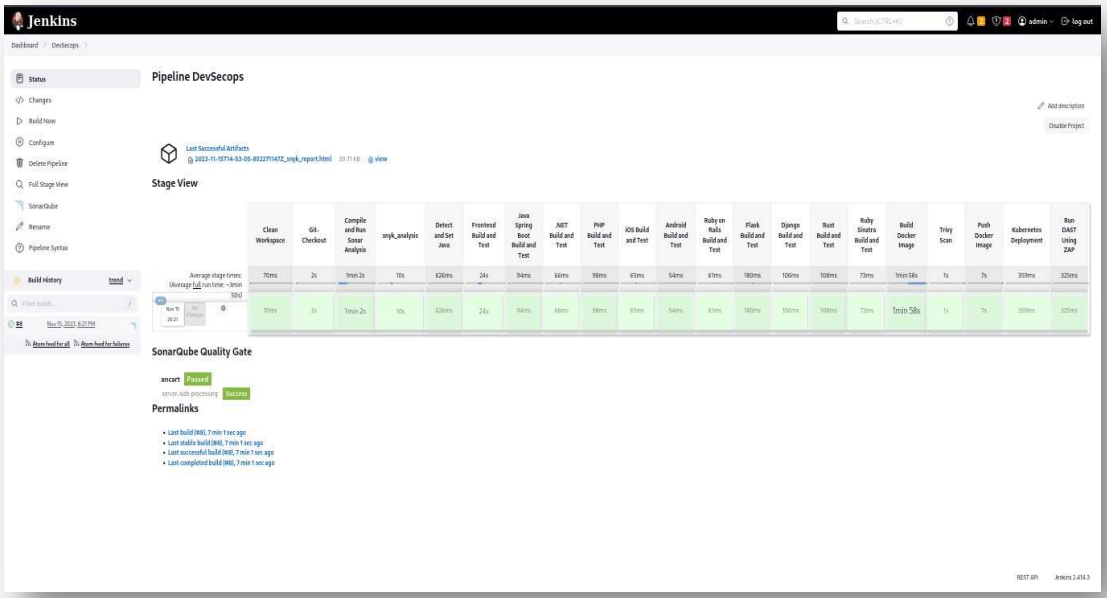
After that, click the "Build Now" button.



The job will begin to execute. You can check your job outcomes at the console to see if there are any problems.



This is where you can see output of the job has succeeded.



STAGE 1 (Cleanup Workspace)

In this stage, we going to cleanup our workspace where files and documents that are previously deployed before, after this stage done the git will pull newly updated files and run everything newly.

```
stages {
  stage('Clean Workspace') {
    steps {
      cleanWs()
    }
  }
```

Stage 2 (Git Checkout)

Git:

Give your Git details in the SCM above; therefore, update those with the URL and branch names of your git details in from SCM.

```
stage('Git-Checkout') {
    steps {
        checkout scm
    }
}
```

Note: If your git repository is private, you should give your Jenkins account your Gitlab personal access token or git credentials.

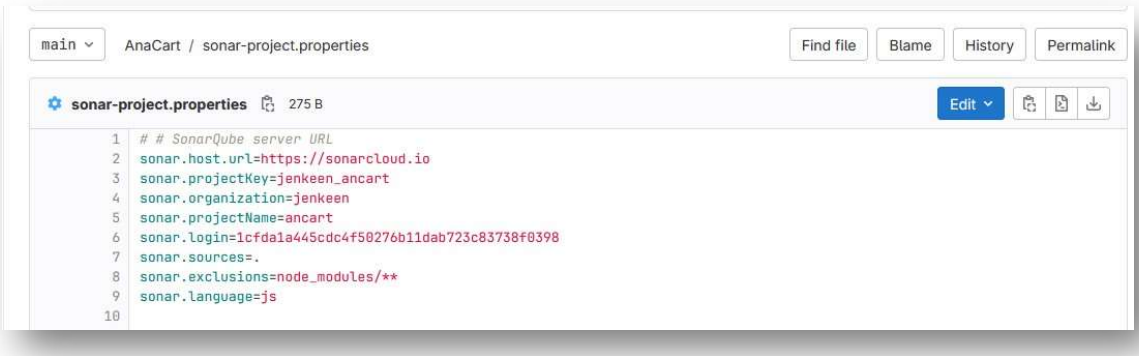
Stage 3 (Sonarcloud):

SonarCloud is used to perform SAST code Quality scans, so integrate it with Jenkins by giving it a personal access token or authentication token. You may also call your Sonar scanner tool as sonar-scanner, or anything else you choose, and don't forget to include it in your pipeline.

```
SONARCLOUD = 'Sonarcloud'
```

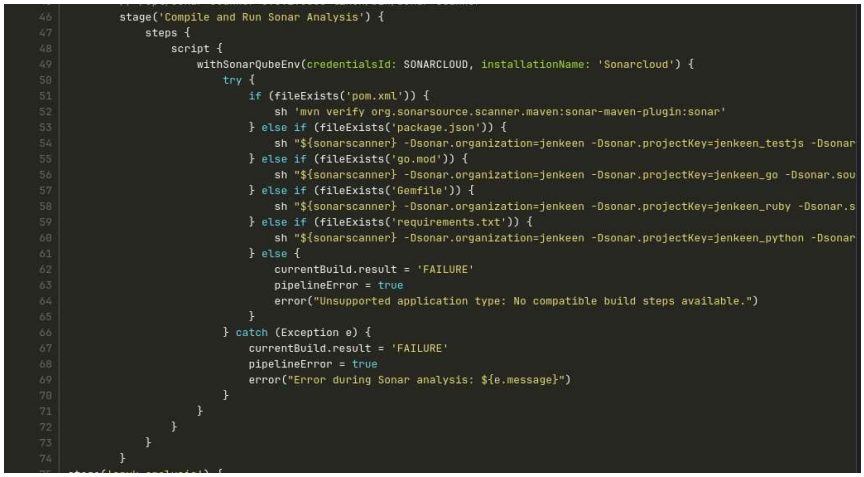
There are two options to run sonarcloud :

1) Create sonar-project-properties file in your git repository and give the sonarcloud details like give below:



Here, adds your sonar-scanner path to the Jenkins pipeline script along with your pom.xml, csproj, solution file, package. Json, Gem file, requirement.txt, etc.

2) You can directly mention your sonar cloud script in the Jenkins file.



3) If Sonar throws compile errors, ensure to edit your versioning according for a completed compile of the app. For example, maven plugin errors may require an edit of the pom.xml file that is used in the repo.

Stage 4 (Snyk):

Synk is used to perform security vulnerability scans, so integrate it with Jenkins by giving it a personal access token or authentication token. You may also call your synk Installation tool as Snyk@latest, or anything else you choose, and don't forget to include it in your pipeline.

```
SNYK_INSTALLATION = 'snyk@latest'
SNYK_TOKEN = 'Snyk'
```

In your pipeline, now mention your installation and the name of your Snyk token so that it knows which API you are attempting to access.

```
stage('snyk_analysis') {
  steps {
    script {
      echo 'Testing...'
      try {
        snykSecurity(
          snykInstallation: SNYK_INSTALLATION,
          snykTokenId: SNYK_TOKEN,
          failOnIssues: false,
          monitorProjectOnBuild: true,
          additionalArguments: '--all-projects --d'
        )
      } catch (Exception e) {
        currentBuild.result = 'FAILURE'
        pipelineError = true
        error("Error during snyk_analysis: ${e.message}")
      }
    }
  }
}
```

Stage 5 (Java Detection):

Java may be automatically detected, and you will be able to see whether it is supported or not. Ensure to setup the correct JDK in Jenkins tools before doing it.

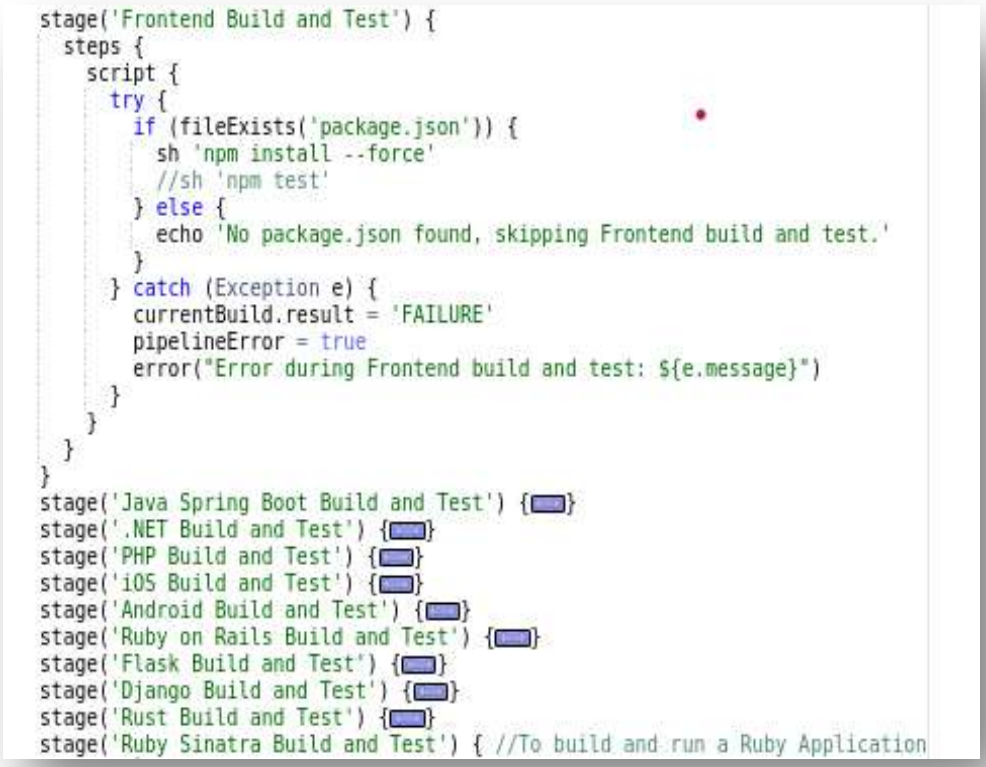
```
// Define the detectJavaVersion function outside of the pipeline block
def detectJavaVersion() {
    def javaVersionOutput = sh(script: 'java -version 2>&1', returnStatus: false, returnStdout: true).trim()
    def javaVersionMatch = javaVersionOutput =~ /openjdk version "(\d+\.\d+)/

    if (javaVersionMatch) {
        def javaVersion = javaVersionMatch[0][1]

        if (javaVersion.startsWith("1.8")) {
            return '8'
        } else if (javaVersion.startsWith("11")) {
            return '11'
        } else if (javaVersion.startsWith("17")) {
            return '17'
        } else {
            error("Unsupported Java version detected: ${javaVersion}")
        }
    } else {
        error("Java version information not found in output.")
    }
```

So here the java detects and set java pipeline script looks like:

```
stage('Detect and Set Java') {
  steps {
    script {
      try {
        def javaVersion = detectJavaVersion()
        tool name: "Java_${javaVersion}", type: 'jdk'
        sh 'java --version'
      } catch (Exception e) {
        currentBuild.result = 'FAILURE'
        pipelineError = true
        error("Error during Java version detection: ${e.message}")
      }
    }
  }
}
```

Stage 6 (Multi-Language Build and Deployment):

During this phase, there is a variety of programming languages, including front-end, back-end, iOS, Android, Ruby, Flask, and so forth. Based on the languages I provided, the system will identify the source code from your repository and install, build, and execute the test in accordance with the pipeline script that we previously discussed.



```
stage('Frontend Build and Test') {
  steps {
    script {
      try {
        if (fileExists('package.json')) {
          sh 'npm install --force'
          //sh 'npm test'
        } else {
          echo 'No package.json found, skipping Frontend build and test.'
        }
      } catch (Exception e) {
        currentBuild.result = 'FAILURE'
        pipelineError = true
        error("Error during Frontend build and test: ${e.message}")
      }
    }
  }
}
stage('Java Spring Boot Build and Test') {...}
stage('.NET Build and Test') {...}
stage('PHP Build and Test') {...}
stage('iOS Build and Test') {...}
stage('Android Build and Test') {...}
stage('Ruby on Rails Build and Test') {...}
stage('Flask Build and Test') {...}
stage('Django Build and Test') {...}
stage('Rust Build and Test') {...}
stage('Ruby Sinatra Build and Test') { //To build and run a Ruby Application
```

The pipeline script for the multi-lang build in the image above.

Stage 7 (Docker Build and Push):

At this Stage we are going to dockerize our project after building our source code. Our pipeline script will automatically identify whether a dockerfile exists and generate the dockerfile if not, else it will display the dockerfile not found.

Note: Make sure you specify the name of your Docker image correctly in the environment stage (the variable name will automatically identify and take the image name). The Docker file name is case-sensitive, add docker tool and docker API in your Jenkins.

```
stage('Build and Push Docker Image') {
    steps {
        script {
            try {
                if (fileExists('Dockerfile')) {
                    withDockerRegistry(credentialsId: DOCKER_REGISTRY_CREDENTIALS, toolName: DOCKER_TOOL, url: DOCKER_URL) {
                        def dockerImage = docker.build(DOCKER_IMAGE, ".")
                        // Push the built Docker image
                        dockerImage.push()
                    }
                } else {
                    echo "Dockerfile not found. Skipping Docker image build and push."
                }
            } catch (Exception e) {
                currentBuild.result = 'FAILURE'
                pipelineError = true
                echo "Error during Docker image build and push: ${e.message}"
            }
        }
    }
}
```

We will push and store our image on container registries such as Docker Hub, AWS ECR, GCP GCR, Harbour, and others at this stage. I used Docker Hub in this instance by supplying my credentials and indicating the Docker API that I am pushing to my hub repository. Don't forget to set up a repository on Docker Hub before that.

To link with your container registries, make sure you provide your credentials or personal access token to Jenkins. Mention your credentials in the environment stage of the pipeline.

```
environment {
SONARCLOUD = 'Sonarcloud'
SNYK_INSTALLATION = 'snyk@latest'
SNYK_TOKEN = 'Snyk'
DOCKER_REGISTRY_CREDENTIALS = 'Docker_Server'
DOCKER_IMAGE = '[redacted]:latest'
DOCKER_TOOL = 'Docker'
DOCKER_URL = 'https://index.docker.io/v1/'
KUBE_CONFIG = 'kubernetes'
```

Note: By using the docker run command locally, you can verify whether your Docker image is up and running or not.

Stage 8 (Aqua Trivy Scan):

Now that the Docker build has finished and our image has been successfully produced, it's time to detect any vulnerabilities by scanning it. We're going to use Aqua Trivy Scan for image Scanning.

Verify that Aqua Trivy has been installed on your local system.If you don't already have trivy installed on your system, get it from docker and run trivy image. Once it's finished, try using docker trivy image to scan your image. Put the name of your image after the image command using the following docker trivy command:

docker run ghcr.io/aquasecurity/trivy:latest image DOCKER_IMAGE

```
stage('Trivy Scan') {
    steps {
        script {
            def trivyInstalled = sh(script: 'command -v trivy', returnStatus: true) == 0
            def imageName = DOCKER_IMAGE
            if (trivyInstalled) {
                sh "trivy image --format table ${imageName}"
            } else {
                // Run trivy using Docker
                sh "docker run --rm -v /var/run/docker.sock:/var/run/docker.sock aquasec/trivy image --format table ${imageName}"
            }

        }

    }
}
```

Mention your docker image here, it will scan and detect the vulnerabilities.

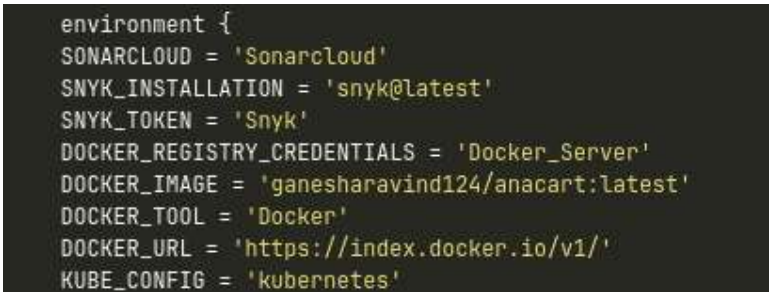
Stage 9 (Kubernetes):
It is the main stage we are now in. Up to this point, everything went according to plan as we built, deployed, and dockerized our image and pushing it to the hub. However, we must host our program in runtime. What is the process? By applying Kubernetes is the way forward.

Make sure you have installed clusters before integrating Kubernetes with Jenkins; it doesn't matter if they are minikube, kind, or kubeadm. If you're using a loadbalancer, install kubeadm and construct your master and worker nodes. If you're using nodeport, install your minikube or kind cluster on your Jenkins slave machine.

Note: You can integrate your Jenkins with your Kubernetes cluster using the kube config file.

```
stage('Kubernetes Deployment') {
  steps {
    script {
      def configFile = 'deployment.yaml'
      if (fileExists(configFile)) {
        kubernetesDeploy(configs: configFile, kubeconfigId: KUBE_CONFIG)
      } else {
        error("Error: $configFile does not exist")
        currentBuild.result = 'FAILURE'
        pipelineError = true
      }
    }
  }
}
```

In the environment stage, provide your kube configuration credentials and add the name of your deployment.yaml file in place of the config file.



```
environment {
SONARCLOUD = 'Sonarcloud'
SNYK_INSTALLATION = 'snyk@latest'
SNYK_TOKEN = 'Snyk'
DOCKER_REGISTRY_CREDENTIALS = 'Docker_Server'
DOCKER_IMAGE = 'ganesharavind124/anacart:latest'
DOCKER_TOOL = 'Docker'
DOCKER_URL = 'https://index.docker.io/v1/'
KUBE_CONFIG = 'kubernetes'
```

The application will now be operational on your pods after the deployment was successfully created there. You can test by running (kubectl get svc) with the service name. If you used a loadbalancer, you would receive your external IP and be able to access your application through it.

If you ran (minikube service MY-SERVICE-NAME) with minikube, you will receive your IP and port number and be able to access your application through it.

Stage 10 (Zaproxy Testing):

We have already performed SAST Scan and Application Testing; going forward, we will perform DAST, which is intended to assist in detecting security vulnerabilities in web applications throughout the software development and testing stages.

ZAP testing will involve using that URL to test an application hosted in PROD or DEV. We'll use a variety of scan methods, including spidering, active, passive, fuzzer, proxy interception, and scripting attacks. However, for now, I'm just doing a basic zap test that generates and provides us with a report.

Make sure that ZAPROXY has installed on your Local or Instance or server systems.

With the use of a local minikube, provided the URL can be provided directly in Jenkins

based on deployment.yaml configuration in the pipeline.

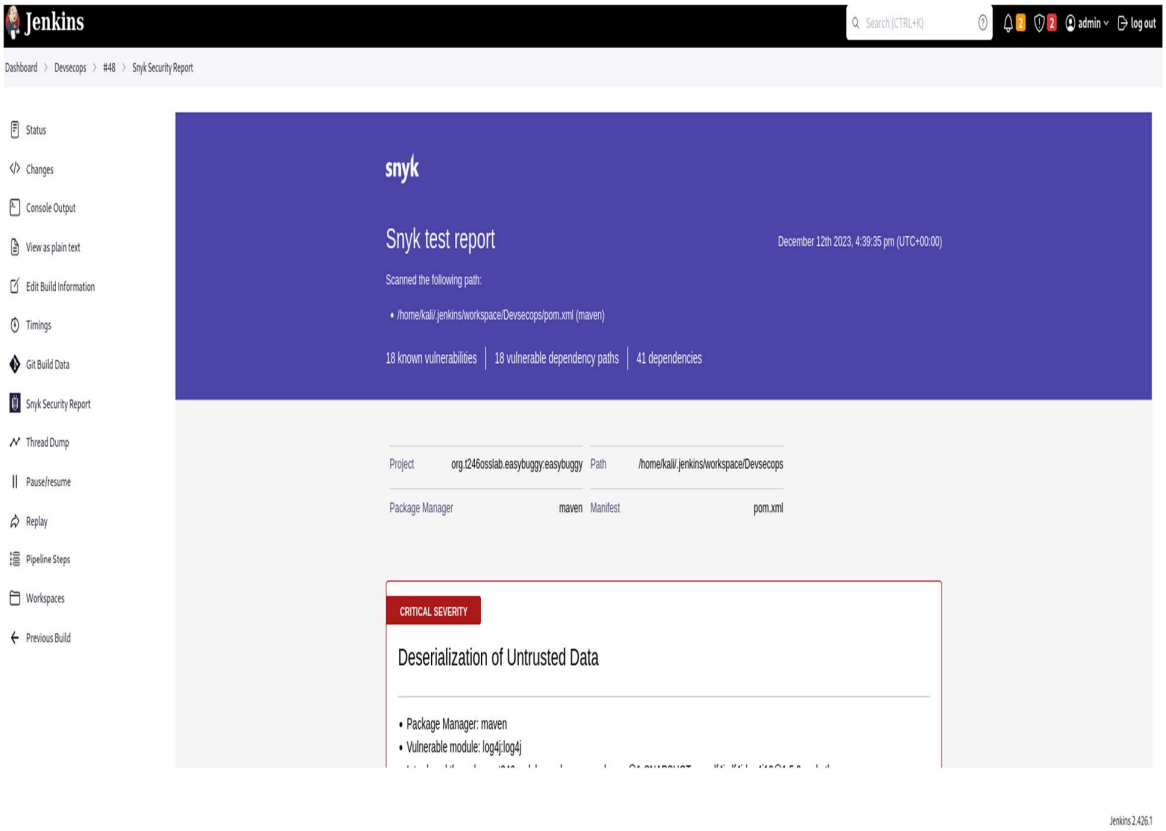
```
    stage('Run DAST Using ZAP') {
      steps {
        script {
          try {
            def targetURL = "http://192.168.49.2:32767/" // Replace with your actual target URL
            def zapCommand = "zaproxy -cmd -quickurl ${targetURL}"
            sh(zapCommand)
            archiveArtifacts artifacts: 'zap_report.html'
          } catch (Exception e) {
            currentBuild.result = 'FAILURE'
            error("Error during ZAP DAST: ${e.message}")
          }
        }
      }
    }
  }
}
```

When using Loadbalancer (Cloud), the zap command will be executed automatically without the need for manual entry, and the IP and port will be generated automatically. Use the below script to detect URL automatically.

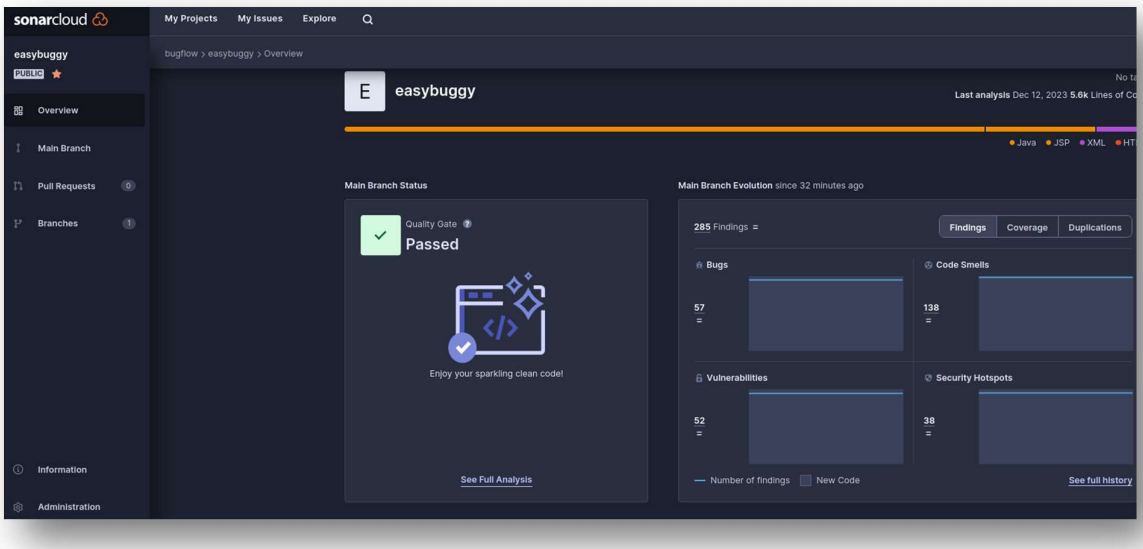
```
stage('Run DAST Using ZAP') {
  steps {
    script {
      try {
      def targetURL = sh(script: 'kubectl get service/' + serviceName + ' -o json | jq -r ".status.loadBalancer.ingress[0].hostname"', returnStatus: true).trim()
      def zapCommand = "zap.sh -cmd -quickurl http://${targetURL} -quickprogress -quickout ${ZAP_WORKSPACE}/zap_report.html"
        sh(zapCommand)
        archiveArtifacts artifacts: 'zap_report.html'
      } catch (Exception e) {
        currentBuild.result = 'FAILURE'
        error("Error during ZAP DAST: ${e.message}")
      }
    }
  }
}
```

Sample Results:

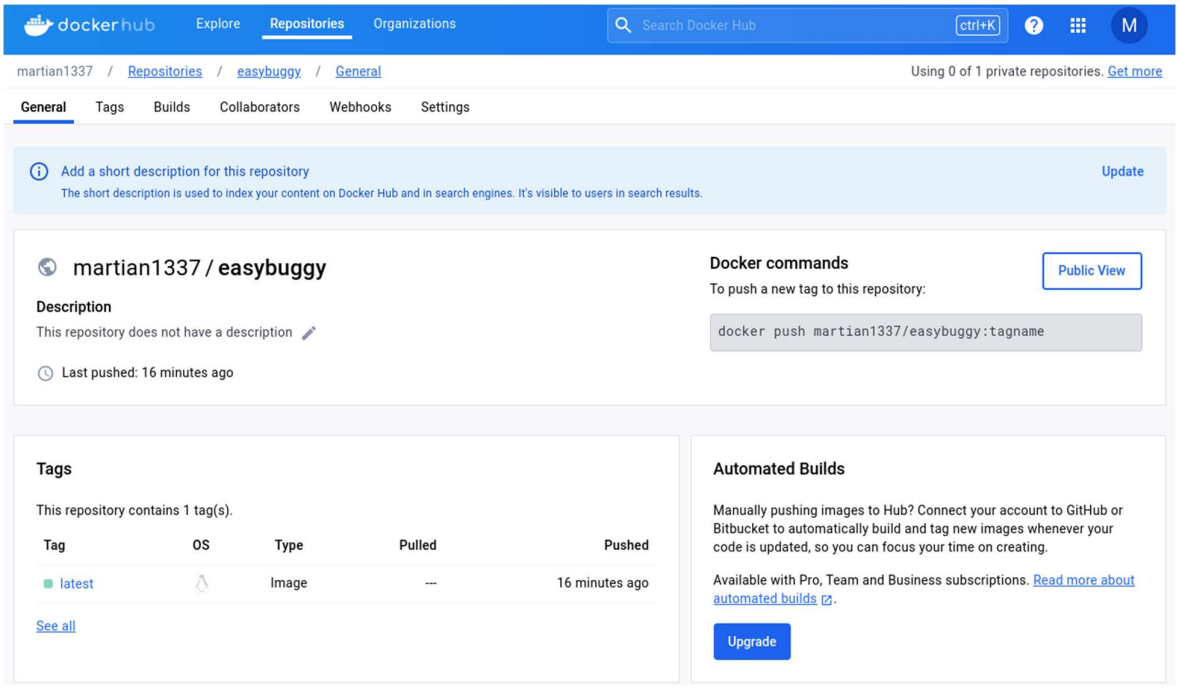
SNYK:



SONARCLOUD:



DOCKER HUB:



TRIVY:

```
martian1337/easybuggy:latest (debian 11.6)
==========================================
Total: 127 (UNKNOWN: 0, LOW: 73, MEDIUM: 31, HIGH: 21, CRITICAL: 2)

┌──────────────┬────────────────┬──────────┬────────────┬───────────────────┬───────────────┬──────────────────────────────────────────────────────────────┐
│   Library    │ Vulnerability  │ Severity │   Status   │ Installed Version │ Fixed Version │                            Title                             │
├──────────────┼────────────────┼──────────┼────────────┼───────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ apt          │ CVE-2011-3374  │ LOW      │ affected   │ 2.2.4             │               │ It was found that apt-key in apt, all versions, do not       │
│              │                │          │            │                   │               │ correctly...                                                 │
│              │                │          │            │                   │               │ https://avd.aquasec.com/nvd/cve-2011-3374                    │
├──────────────┼────────────────┼──────────┼────────────┼───────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ bash         │ CVE-2022-3715  │ HIGH     │            │ 5.1-2+deb11u1     │               │ a heap-buffer-overflow in valid_parameter_transform          │
│              │                │          │            │                   │               │ https://avd.aquasec.com/nvd/cve-2022-3715                    │
│              ├────────────────┼──────────┤            │                   │               ├──────────────────────────────────────────────────────────────┤
│              │ TEMP-0841856-B18BAF │ LOW      │            │                   │               │ [Privilege escalation possible to other user than root]      │
│              │                │          │            │                   │               │ https://security-tracker.debian.org/tracker/TEMP-0841856-B1- │
│              │                │          │            │                   │               │ 8BAF                                                         │
├──────────────┼────────────────┼──────────┼────────────┼───────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ bsdutils     │ CVE-2022-0563  │          │            │ 1:2.36.1-8+deb11u1│               │ util-linux: partial disclosure of arbitrary files in chfn    │
│              │                │          │            │                   │               │ and chsh when compiled...                                    │
│              │                │          │            │                   │               │ https://avd.aquasec.com/nvd/cve-2022-0563                    │
├──────────────┼────────────────┼──────────┼────────────┼───────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ coreutils    │ CVE-2016-2781  │          │ will_not_fix │ 8.32-4+b1         │               │ coreutils: Non-privileged session can escape to the parent   │
│              │                │          │            │                   │               │ session in chroot                                            │
│              │                │          │            │                   │               │ https://avd.aquasec.com/nvd/cve-2016-2781                    │
│              ├────────────────┼──────────┼────────────┼───────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│              │ CVE-2017-18018 │          │ affected   │                   │               │ coreutils: race condition vulnerability in chown and chgrp   │
│              │                │          │            │                   │               │ https://avd.aquasec.com/nvd/cve-2017-18018                   │
├──────────────┼────────────────┼──────────┼────────────┼───────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ e2fsprogs    │ CVE-2022-1304  │ HIGH     │            │ 1.46.2-2          │               │ e2fsprogs: out-of-bounds read/write via crafted filesystem   │
│              │                │          │            │                   │               │ https://avd.aquasec.com/nvd/cve-2022-1304                    │
├──────────────┼────────────────┼──────────┼────────────┼───────────────────┼───────────────┼──────────────────────────────────────────────────────────────┤
│ gcc-10-base  │ CVE-2023-4039  │ MEDIUM   │            │ 10.2.1-6          │               │ gcc: -fstack-protector fails to guard dynamic stack          │
│              │                │          │            │                   │               │ allocations on ARM64                                         │
│              │                │          │            │                   │               │ https://avd.aquasec.com/nvd/cve-2023-4039                    │
├──────────────┤                │          │            │                   │               │                                                              │
│ gcc-9-base   │                │          │            │ 9.3.0-22          │               │                                                              │
```

Troubleshooting Trivy + Docker deployment:

You may encounter a permissions issue with accessing the Docker daemon like this:

docker: permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Post "http://%2Fvar%2Frun%2Fdocker.sock/v1.24/containers/create": dial unix /var/run/docker.sock: connect: permission denied.

This usually happens when the user running the Docker command does not have the necessary permissions to access the Docker daemon. Here are a few ways to resolve this issue:

Run as Root User: The simplest way is to run the Docker command as the root user using sudo. This gives you the necessary permissions to interact with the Docker daemon. Try running:

sudo docker run --rm -v /var/run/docker.sock:/var/run/docker.sock aquasec/trivy image --format table docker:latest

Be cautious when using sudo as it grants high-level access to your system.

Add Your User to the Docker Group: A more secure and commonly used method is to add your user to the docker group. This allows any user in the docker group to run Docker commands without needing root access.

First, add your user to the Docker group with: sudo usermod -aG docker $USER

Then, log out and log back in for this change to take effect. Alternatively, you can use newgrp docker to activate the changes in the current session.

After this, try running your Docker command again without sudo.

Check Docker Daemon Status: Ensure that the Docker daemon is running. You can check this with:

sudo systemctl status docker

If it's not running, start it with: sudo systemctl start docker

Verify Socket Permissions: Check the permissions of the Docker socket file: ls -l /var/run/docker.sock

The output should show that the file is owned by root and the docker group. If it's not, you can change the group with:

sudo chgrp docker /var/run/docker.sock

1. Verify Group Membership: Ensure that the user running your pipeline is indeed a member of the Docker group. You can check this with the groups command:

groups [username]

Replace [username] with the username of the pipeline's user. Ensure docker is listed in the output.

Restart the Docker Service: After modifying group memberships, it can sometimes be necessary to restart the Docker service for the changes to take effect. Use:

sudo systemctl restart docker

By following these steps, you should be able to resolve the permission issues and successfully run your Docker command.

Permissions to run Docker

1. Run the below command to add user and Jenkins to the appropriate group for permissions
    a. ‘sudo usermod -aG docker $USER && sudo usermod -aG docker jenkins && newgrp docker’
2. Restart docker service with command below
    a. ‘ sudo systemctl restart docker ‘
3. Set permissions on Docker socket
    a. ‘ sudo chmod 666 /var/run/docker.sock ‘

Lost Jenkins Password

1. If you don’t change the initial password and lose it for any reason, you can always retrieve it with the command: sudo cat /var/lib/jenkins/secrets/initialAdminPassword

Too many Kubernetes pods/reset

1. If you have too many pods and want to remove them, you can run the below command to delete them all within a namespace or all namespaces with the commands here:
   kubectl delete pods --all --namespace <name> or kubectl delete pods --all --all-namespaces