Table of Contents

[Introduction: 1](#_Toc147693753)

[Pipeline Overview: 2](#_Toc147693754)

[Tools used: 2](#_Toc147693755)

[CICD Pipeline Flowchart: 3](#_Toc147693756)

[Setup Instructions 5](#_Toc147693757)

[Integrating Sonarqube with Jenkins 5](#_Toc147693758)

[Creating Docker hosted repository in Nexus and pushing the docker image through Jenkins 9](#_Toc147693759)

[Creating Helm hosted repository in Nexus and Pushing the helm charts 11](#_Toc147693760)

[Configuring mail server in Jenkins (Gmail) 13](#_Toc147693761)

[Configuring PR based trigger in Jenkins 16](#_Toc147693762)

[How to pull the images from private repository (nexus) to k8s kubeadm cluster 24](#_Toc147693763)

[Connecting jenkins with kubernetes cluster 24](#_Toc147693764)

[Conclusion 25](#_Toc147693765)

# Introduction:

This is the comprehensive documentation for our CI/CD pipeline, a pivotal component in our DevOps ecosystem. This documentation has been created to guide you through the setup, configuration, and usage of our CI/CD pipeline, which automates the building, testing, and deployment of our software applications.

**Purpose:** At the heart of our development and deployment strategy lies the CI/CD pipeline. It serves the critical purpose of expediting our software delivery process, enhancing code quality, and ensuring that our applications are reliable and performant. By automating repetitive tasks, our CI/CD pipeline empowers our development and operations teams to collaborate seamlessly, iterate rapidly, and deliver value to our users more efficiently.

**Significance:** The significance of this CI/CD pipeline cannot be overstated. It enables us to:

* Achieve faster release cycles, allowing us to respond to user feedback and market demands with agility.
* Maintain consistent and predictable software quality through automated testing and validation.
* Reduce the manual overhead of deployment, minimizing the risk of human errors.
* Foster a culture of collaboration and continuous improvement across development and operations teams.

**Scope:** In this documentation, you will find a comprehensive guide covering every aspect of our CI/CD pipeline implementation. We will walk you through:

* The initial setup of essential tools such as Jenkins, Docker, Kubernetes, Helm, and GitHub.
* Configuration details to customize the pipeline according to your project's requirements.
* Usage instructions for triggering builds, monitoring deployments, and handling rollbacks.

Troubleshooting guidance to address common issues and challenges that may arise during the pipeline's operation.

**Audience:** This documentation is tailored to meet the needs of developers, DevOps engineers, system administrators, and anyone responsible for managing and automating our software development and deployment processes. While it assumes a basic understanding of DevOps principles and tools, it is designed to cater to both experienced professionals and those new to the field.

**Encouragement:** We strongly encourage you to make use of this documentation as your primary resource while working with our CI/CD pipeline. Whether you are looking to configure the pipeline for a new project, troubleshoot issues, or simply enhance your knowledge of CI/CD best practices, this documentation is here to support you every step of the way.

# Pipeline Overview:

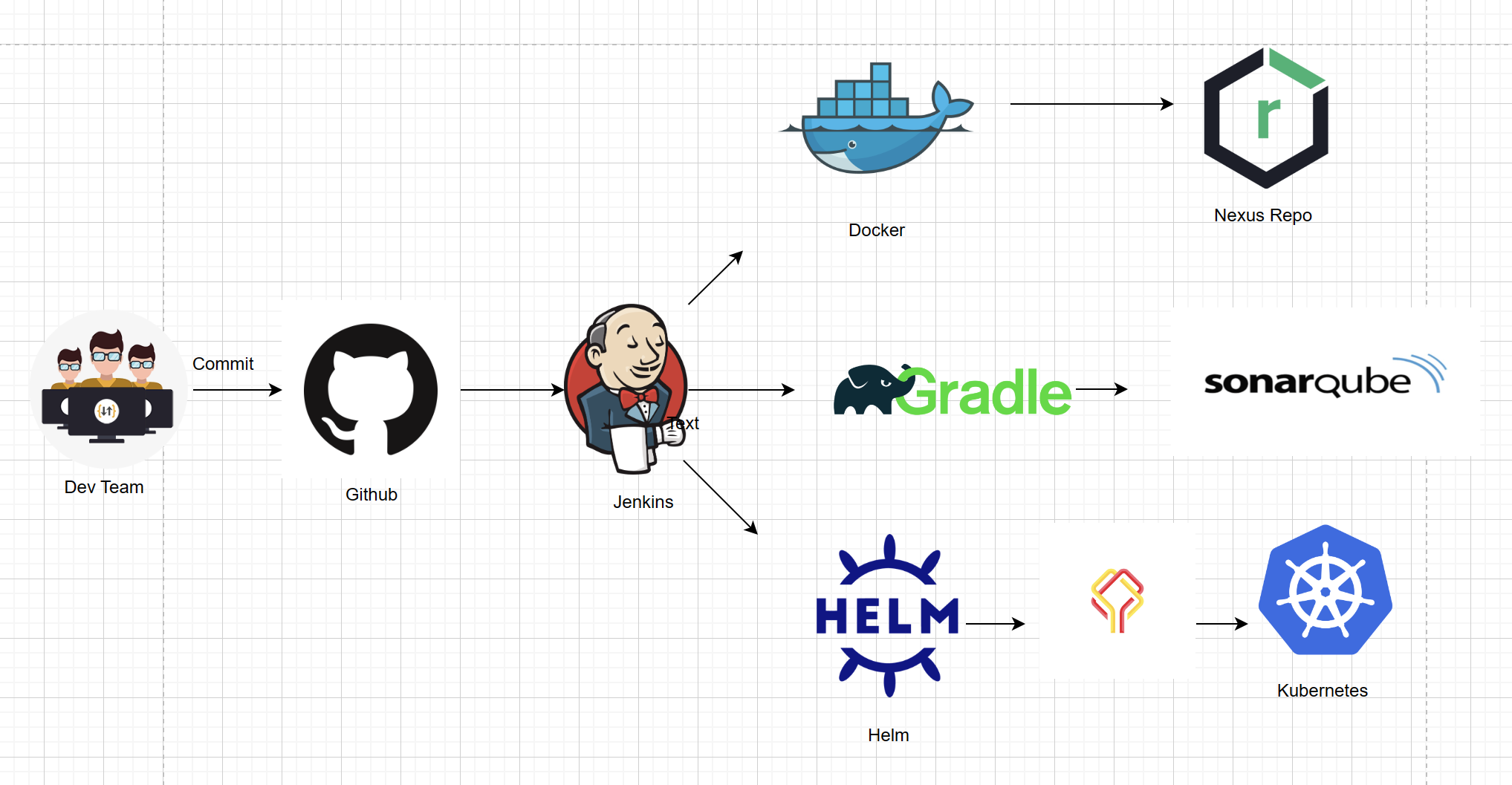


Figure 1- CICD tools

## Tools used:

The first tool in this pipeline is **Git and GitHub**. Git is a distributed version control system installed on local machine for cloning and pushing code changes. GitHub, on the other hand, is a Source Code Management (SCM) tool where we store code, manage branches, and collaborate. Other SCM tools include Bitbucket, Azure Repos, and GitLab.  
  
In this CI/CD pipeline, GitHub stores application code, Dockerfiles, Kubernetes (K8s) manifest files, Helm charts, Jenkinsfiles, and more. Gradle is the next tool used for build automation, supporting multiple languages like Java, C, C++, and JavaScript. It's known for its speed compared to tools like Maven and Ant. In this pipeline, Gradle is used for static code analysis and building artifacts.  
  
**SonarQube** is another tool integrated into the pipeline for universal static code analysis. It supports various languages and enforces code quality rules. Failing the pipeline for code quality issues helps maintain code quality standards.  
  
**Jenkins** serves as the central tool to integrate all others. It's an open-source CI/CD tool that can automate building, testing, and deploying software across different stages of development. Jenkins integrates with tools like GitHub, Gradle, SonarQube, Docker, Kubernetes, and more.  
  
**Docker** is a containerization tool used to package applications with dependencies into containers for consistent deployment. Docker images are created, pushed to Nexus (a repository manager), and pulled from Nexus for deployment in a Kubernetes cluster.  
  
**Kubernetes** is a container orchestration tool to manage containerized applications at scale. Helm, a package manager for Kubernetes, simplifies deploying applications by packaging Kubernetes manifest files into charts.

**Helm** helps in managing Kubernetes applications. Helm Charts helps to define, install and upgrade even the most complex Kubernetes application. Here, helm charts will deploy applications on K8s cluster.  
  
Lastly, **Datree.io** is introduced as a tool to identify misconfigurations, syntax issues, and schema problems in Kubernetes manifest files or Helm charts. It enforces rules early in the development lifecycle to reduce rework.  
  
This CI/CD pipeline streamlines building, testing, and deploying applications while maintaining code quality and consistency.

# CICD Pipeline Flowchart:

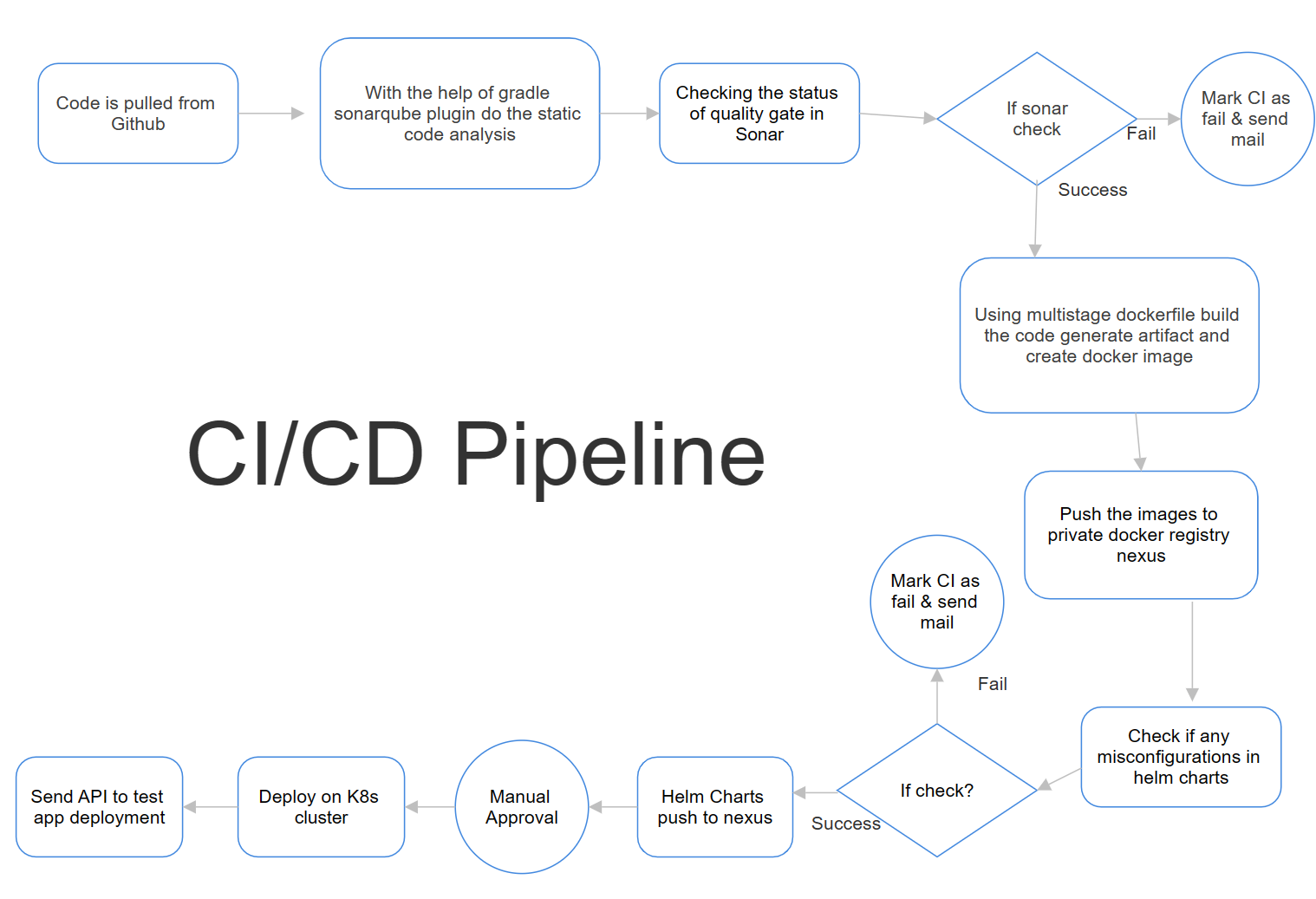


Figure 2- CICD pipeline Flowchart

Let's discuss my CI/CD workflow. To set up a pipeline job, I place my Jenkinsfile in GitHub and configure job details like the GitHub repository, branch, etc. The Jenkinsfile defines various stages.  
  
1. Static Code Analysis**:** I use the SonarQ plugin within Gradle to perform static code analysis. If the code quality check passes, I proceed. If it fails, I mark the CI as failed.  
  
2. Docker Stage: I use a multi-stage Dockerfile to copy code, generate an artifact, and create a Docker image. Instead of building on the Jenkins host, I copy code to the Docker image, run the code there, and create an image. I push this image to my private Docker registry, Nexus.  
3. Helm Chart Verification: In this stage, I verify Helm charts for misconfigurations. If there are no issues, I push the Helm chart to Nexus. Violations trigger an email notification.  
  
4. Manual Approval: After Helm chart pushes, I send an email for manual approval. I use this step to ensure someone reviews the deployment before proceeding.  
  
5. Kubernetes Deployment: Upon approval, I deploy Helm charts to the Kubernetes cluster. I wait for a brief period and then trigger one of the application's APIs. A successful response indicates a correct deployment, while no response leads to CI failure.  
  
This workflow includes email configuration, PR-based triggers, and more, which we'll discuss in detail in the CI pipeline.

**Prerequisites and Server Setup:**

1. Ensure that you have a Jenkins server and a SonarQube server running. Jenkins typically runs on port 8080, and SonarQube runs on port 9000.
2. Additionally, set up a Nexus server alongside Jenkins. By default, Jenkins runs on port 8080, and Nexus runs on port 8081.
3. Install Docker on the Jenkins host machine.

**SonarQube Configuration:**

* Verify that your SonarQube server is up and running.

**Nexus Configuration:**

* Verify that your Nexus server is up and running.

**Jenkins Server Configuration:**

* Ensure that you have a running Jenkins server.

**Gmail Account Setup:**

* To enable email notifications in Jenkins, you need a Gmail account. Make sure this Gmail account allows access to insecure apps.

**GitHub Repository Setup:**

* You will also need a Jenkins server and a GitHub repository for your CI/CD pipeline setup.

# Setup Instructions

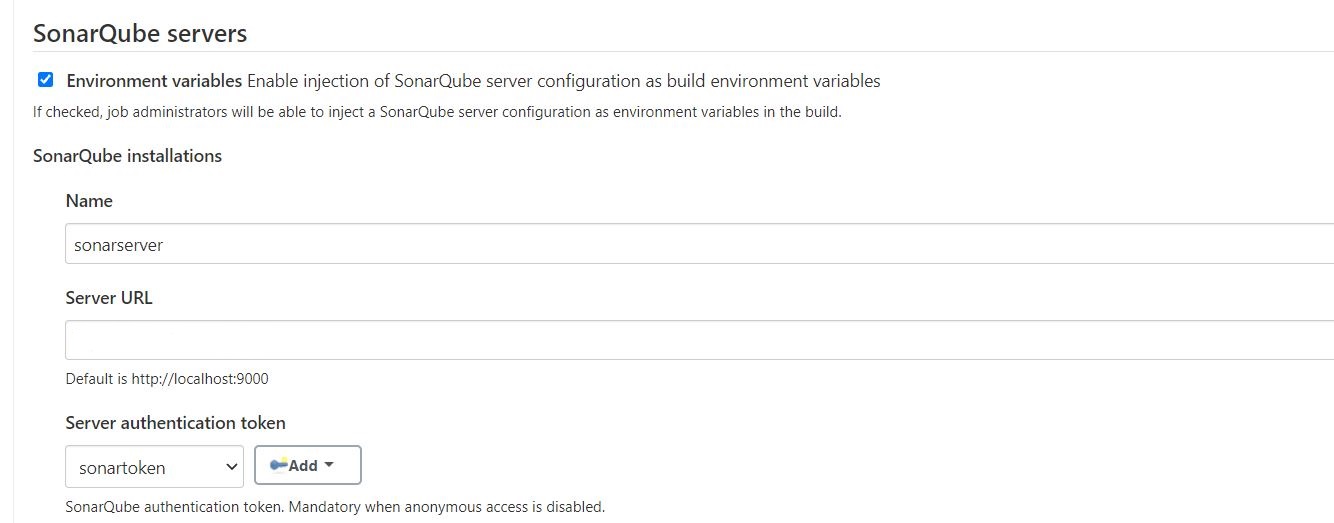
## Integrating SonarQube with Jenkins

**Prerequisite:**

Make sure you have Jenkins and SonarQube servers up and running. Jenkins typically runs on port 8080, and SonarQube on port 9000.

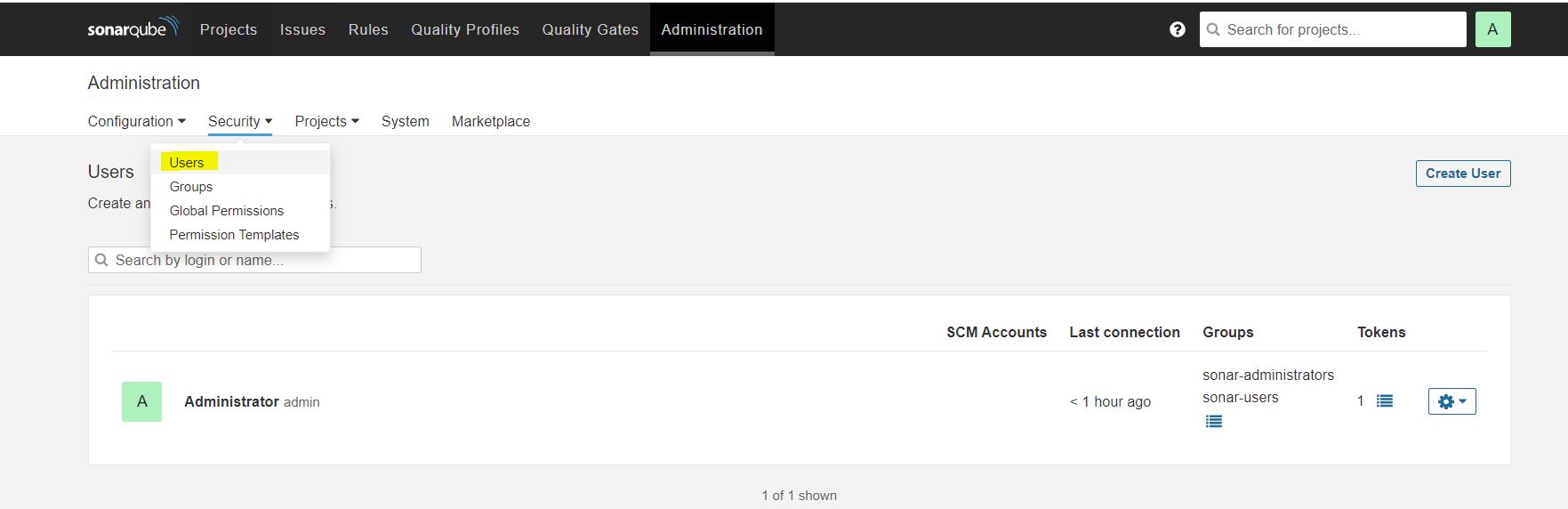
**Initial Setup:**

1. In Jenkins, install the required plugins: Docker, SonarQube, and Sonar Quality Gate Plugin. You can do this by navigating to "Manage Jenkins" > "Manage Plugins" > "Available" tab and searching for these plugins.

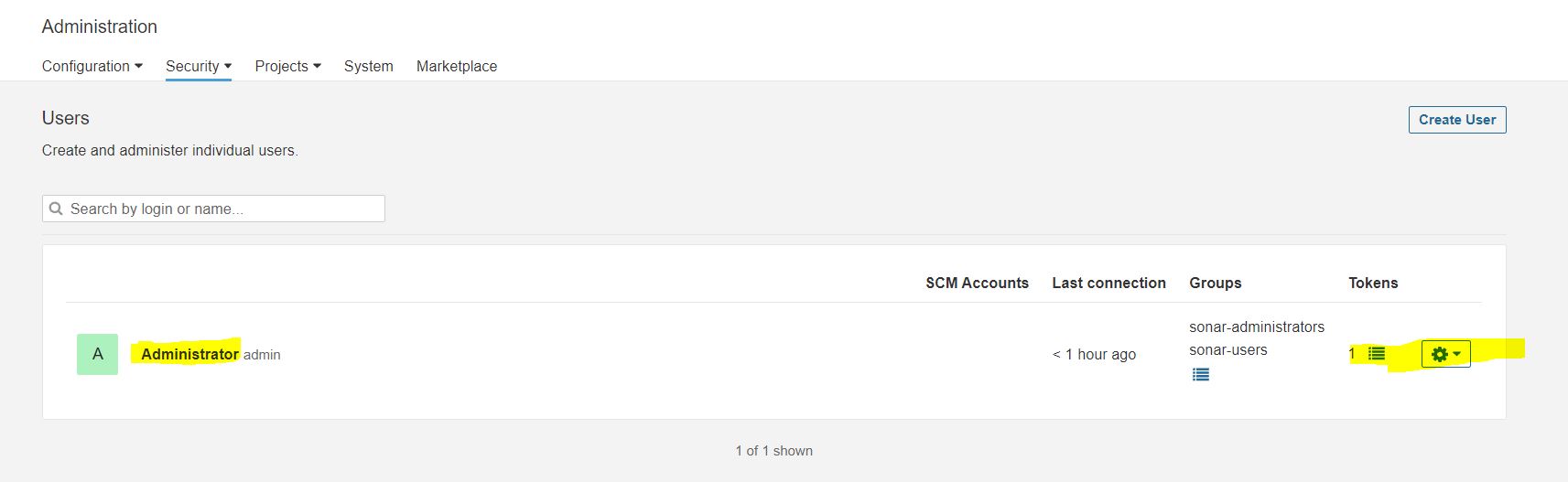


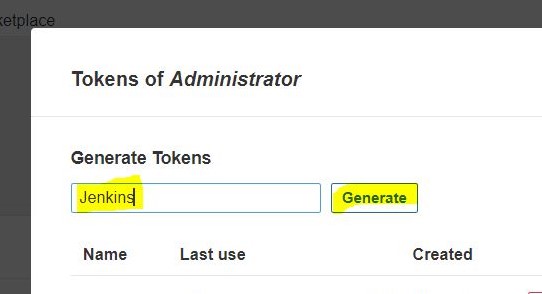
1. After installing the SonarQube plugin, go to the plugin's configuration settings in Jenkins by navigating to "Manage Jenkins" > "Configure System." In this section, provide the following information:

* Name: Give a meaningful name for reference in your Jenkins pipeline.
* Server URL: Enter the URL of your SonarQube server.
* Server Authentication Token: To get this token, follow these steps:
  + In SonarQube, go to "Administration" > "Security" > "Users."

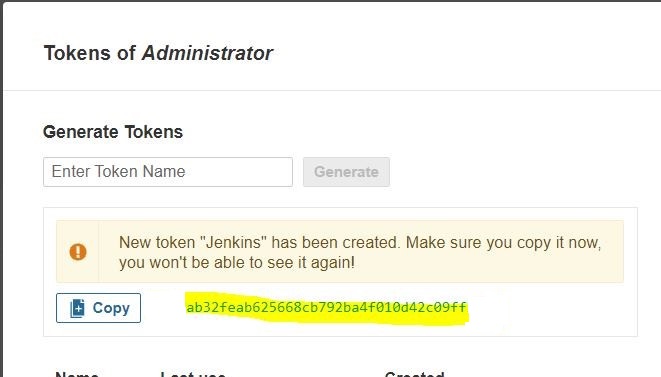


* + Click on the "Tokens" button, then "Generate" to create a new token with a meaningful name.

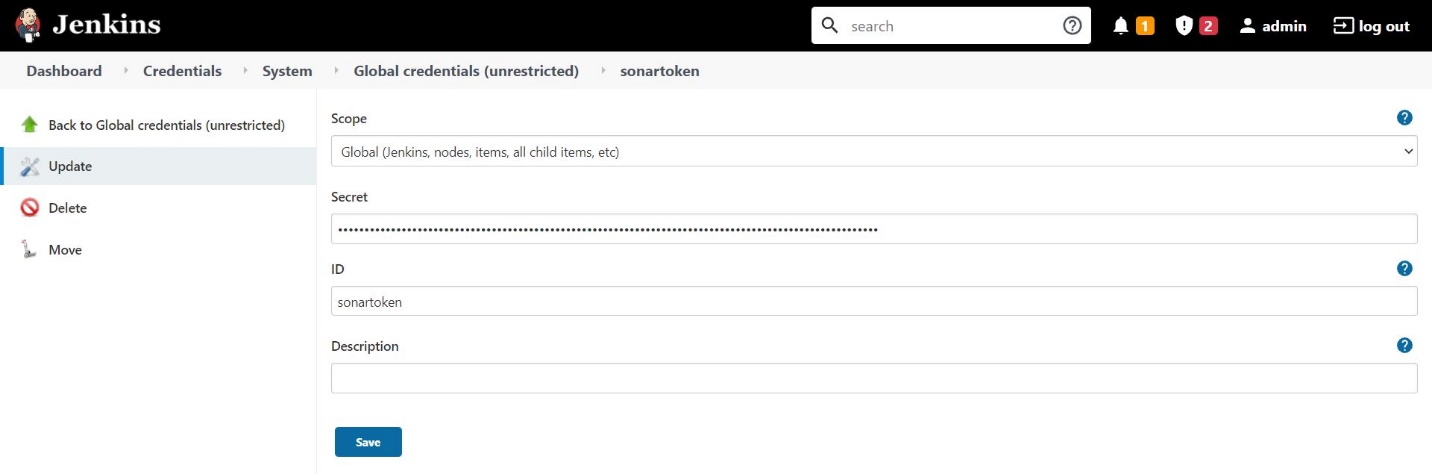




* + Copy the generated token.



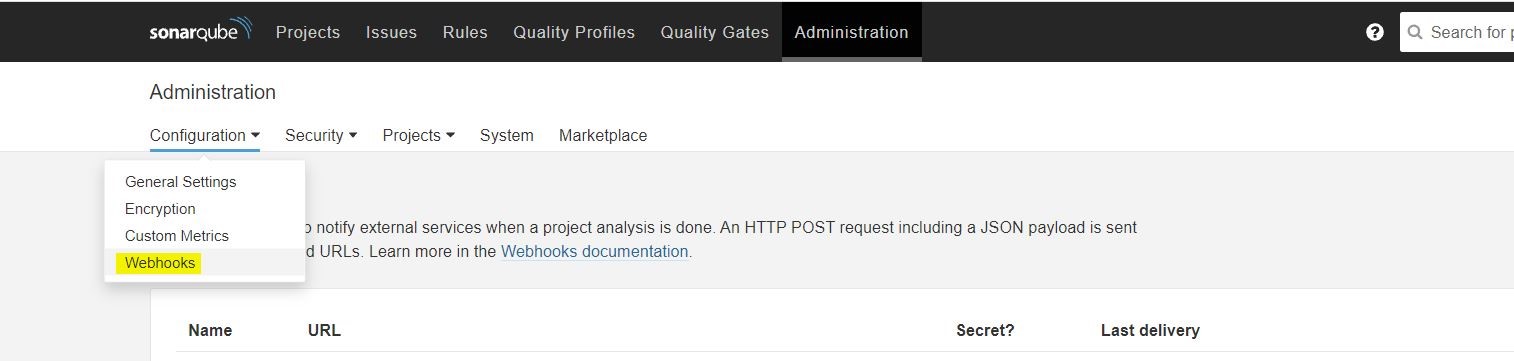
* + Back in Jenkins, navigate to "Manage Jenkins" > "Manage Credentials."



* + Add a new credential of kind "Secret Text" and paste the token in the "Secret" field.

1. Create a webhook to enable communication between Jenkins and SonarQube:

* In SonarQube, navigate to "Administration" > "Configuration" > "Webhooks."



* Provide a name for the webhook and the URL of your Jenkins server suffixed with "/sonarqube-webhook/".

timeout(time: 1, unit: 'HOURS') {

def qg = waitForQualityGate()

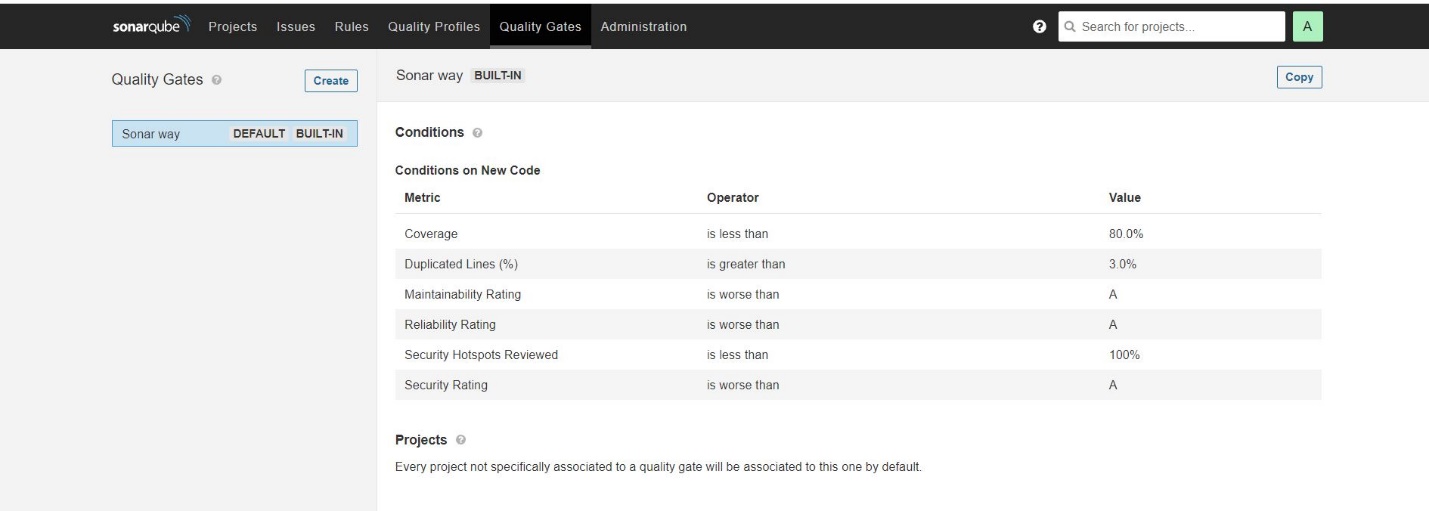
if (qg.status != 'OK') {

error "Pipeline aborted due to quality gate failure: ${qg.status}"

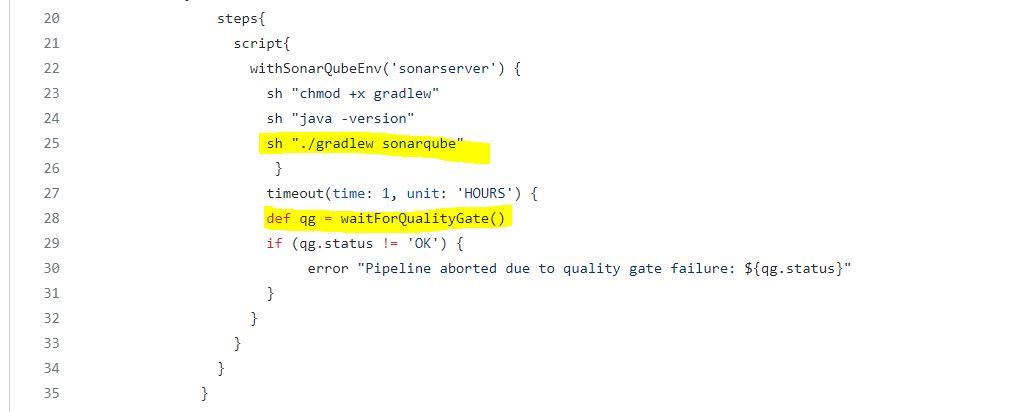
}

}

1. In your Jenkinsfile (pipeline script), include a stage to verify your code against SonarQube's quality gate. If your code meets the quality gate criteria, the build will proceed; otherwise, it will stop at that stage.



1. Configure the SonarQube quality gates as needed. You can customize these gates based on your project's requirements.



1. If the quality gate criteria are met, you will see a "Passed" status in the Jenkins build page.



In summary, these steps help you integrate SonarQube into your Jenkins pipeline to automatically analyze code quality and enforce quality gate checks. This ensures that only code meeting your quality standards proceeds through the pipeline.

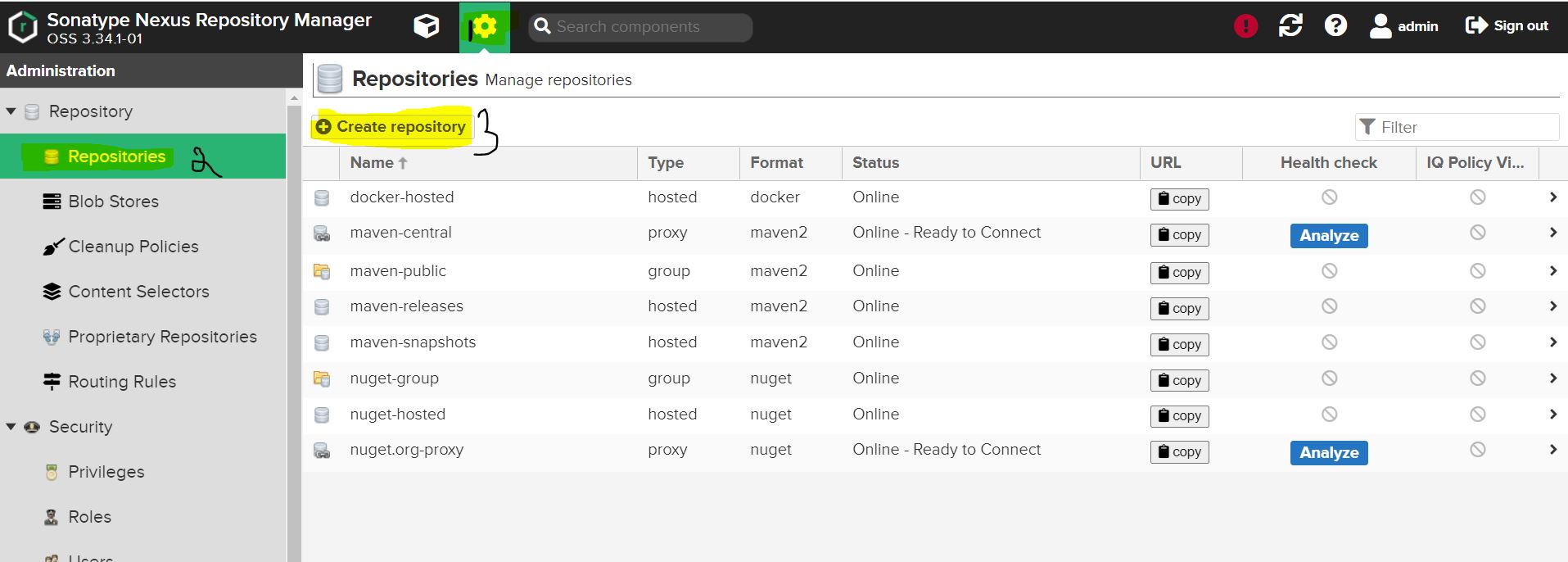
## Creating Docker hosted repository in Nexus and pushing the docker image through Jenkins

**Prerequisites:**

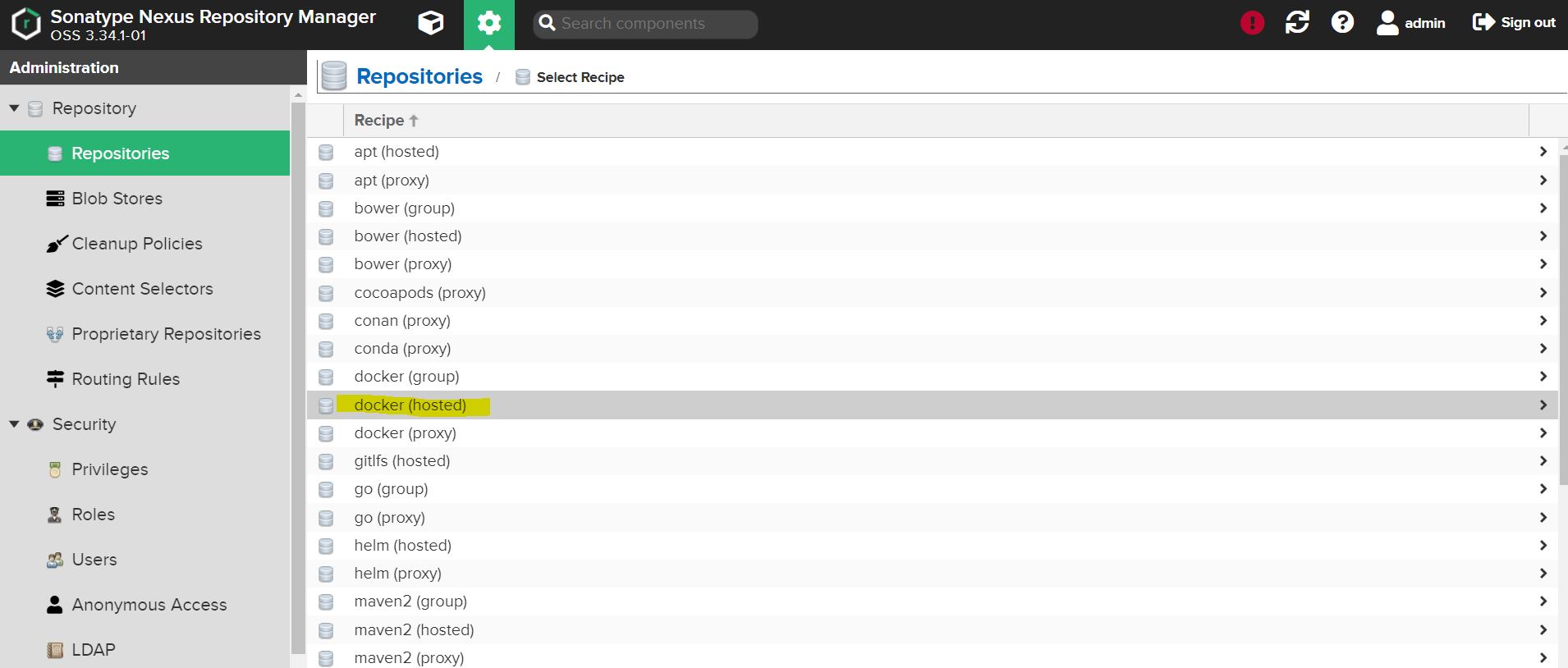
Ensure that you have Jenkins and Nexus servers up and running. By default, Jenkins runs on port 8080, and Nexus on port 8081. If you're using Ubuntu, you can refer to a guide for installation.

**Initial Setup:**

1. In Nexus, start by clicking on the gear icon, then navigate to "Repositories," and select "Create Repository" to set up a new repository.

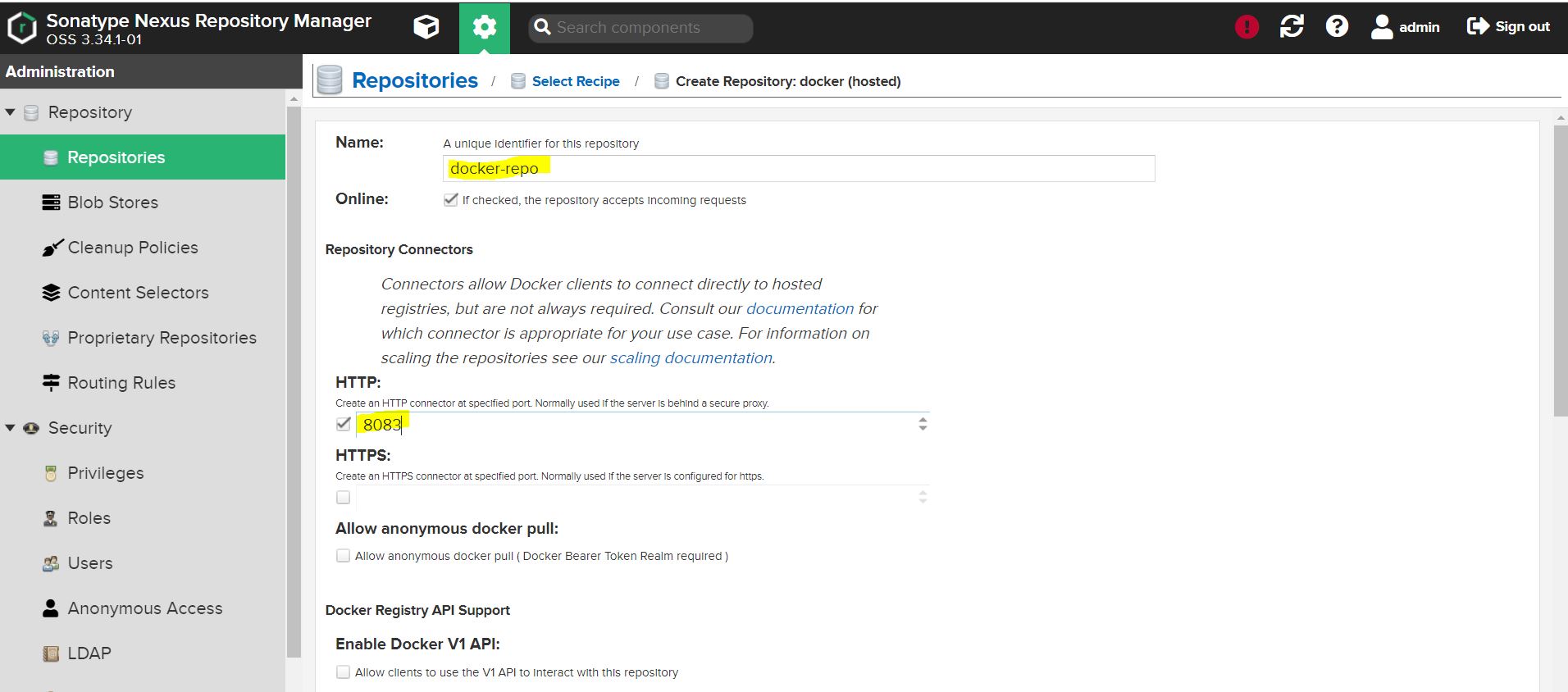


1. When prompted for the repository type, choose "Docker (hosted)" from the available options.

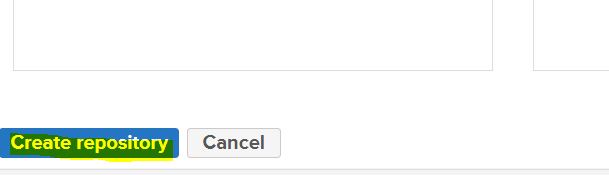


1. Fill in the necessary details for the repository:

* Name: Provide a unique name for your Docker repository.
* Enable the checkbox next to "HTTP" and specify a valid port, preferably 8083.



* After configuring these settings, click on "Create Repository" to complete the setup.



1. On your Jenkins host, you need to configure Insecure Registries to allow connections to your Nexus repository. To do this:

* Edit or create a file at /etc/docker/daemon.json on your Jenkins host.
* Add the following details, specifying the IP address of your Nexus machine and the chosen port (8083):

*{ "insecure-registries": ["nexus\_machine\_ip:8083"] }*

1. After making changes, apply them by restarting the Docker service:

*systemctl restart docker*

1. To verify that the registry has been successfully added, you can check the Docker configuration by running:

*docker info*

1. Finally, from your Jenkins host, you can test the Docker registry connection by executing:

*docker login -u nexus\_username -p nexus\_pass nexus\_ip:8083*

In summary, these steps guide you through creating a Docker hosted repository in Nexus, configuring Docker to recognize the Nexus registry, and testing the connection from your Jenkins host. This enables you to push Docker images to the Nexus repository as part of your CI/CD pipeline.

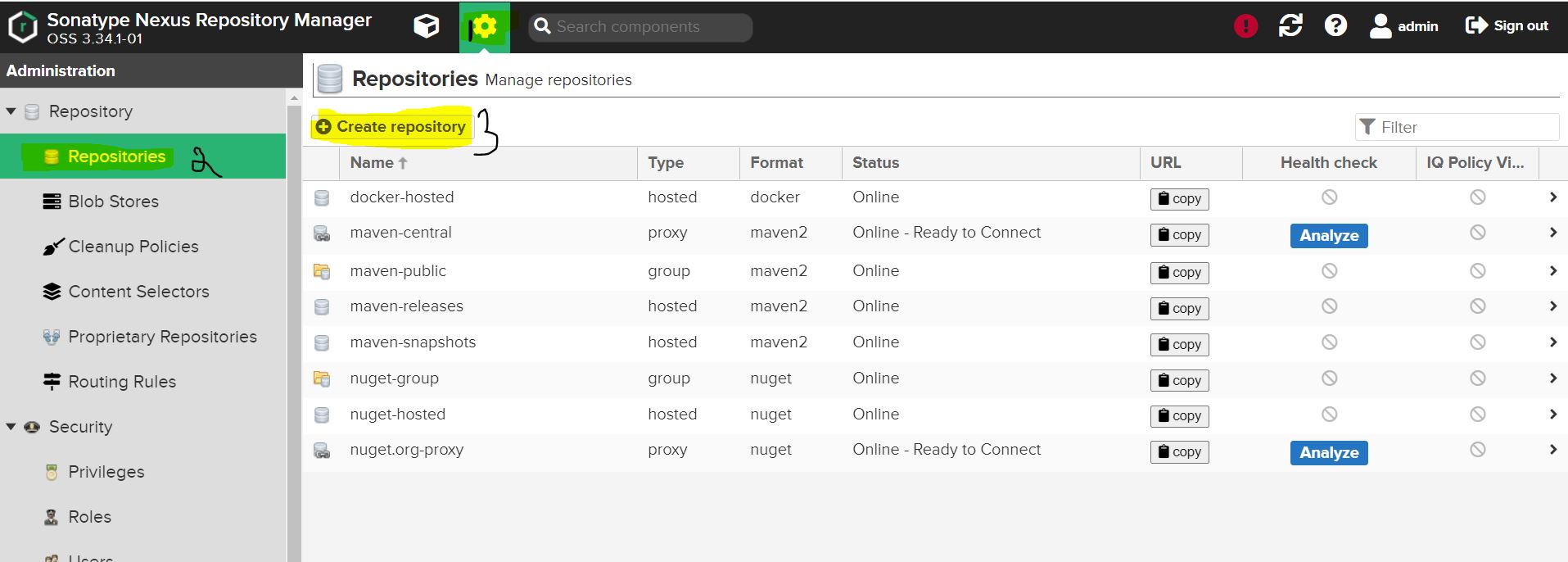
## Creating Helm hosted repository in Nexus and Pushing the helm charts

**Prerequisites:**

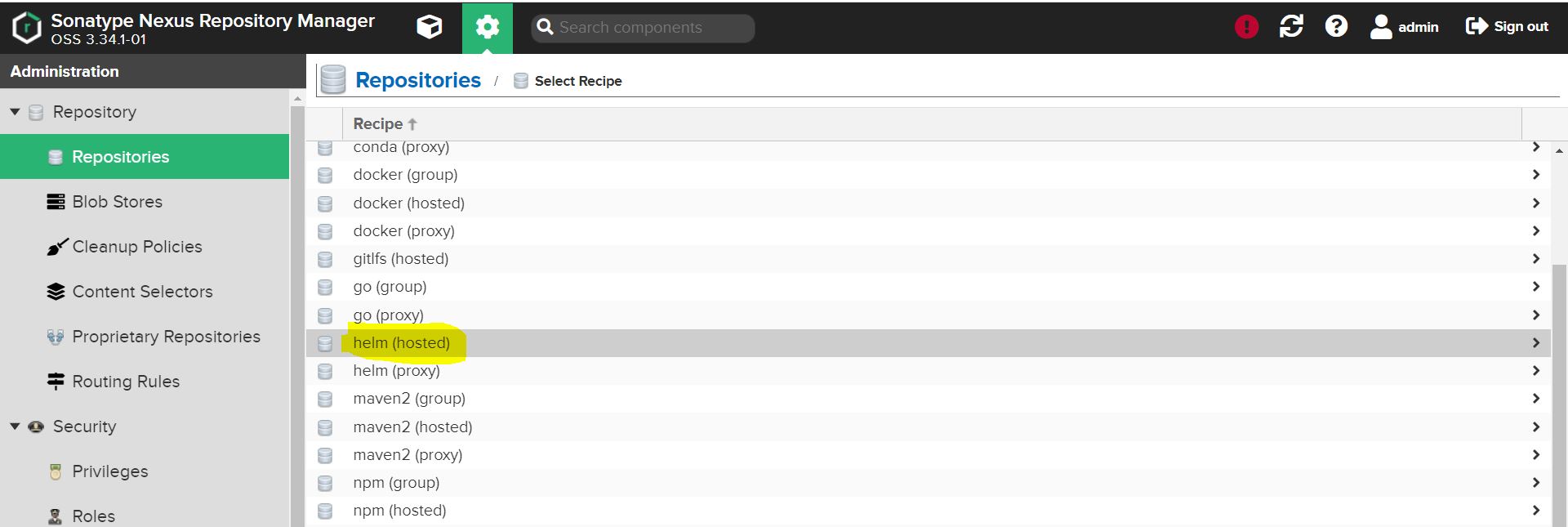
Ensure that you have Jenkins and Nexus servers up and running. By default, Jenkins runs on port 8080, and Nexus on port 8081. You can refer to an installation guide for Ubuntu.

**Initial Setup:**

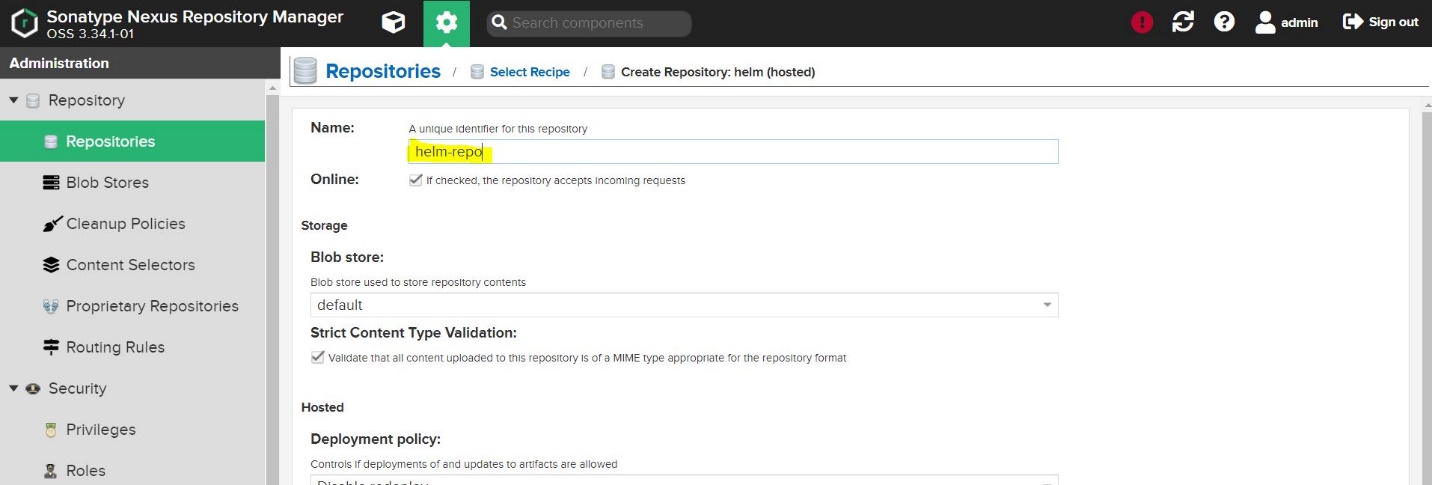
1. In Nexus, start by clicking on the gear icon, then navigate to "Repositories," and select "Create Repository" to set up a new repository.



1. When prompted for the repository type, choose "Helm (hosted)" from the available options.



1. Provide a meaningful name for the Helm repository and click on "Create Repository" to complete the setup.



1. No further configuration is needed on the Jenkins host because we'll interact with the Nexus repository via its API to publish Helm charts.
2. To push a Helm chart to the Nexus repository, you can use the following command as an example. Replace the placeholders with your actual values:

*curl -u admin:$nexus\_password http://nexus\_machine\_ip:8081/repository/helm-hosted/ --upload-file myapp-${helmversion}.tgz -v*

In summary, these steps guide you through creating a Helm hosted repository in Nexus and demonstrate how to push Helm charts to it using a simple curl command. This enables you to store and manage Helm charts in Nexus as part of your application deployment process.

## Configuring mail server in Jenkins (Gmail)

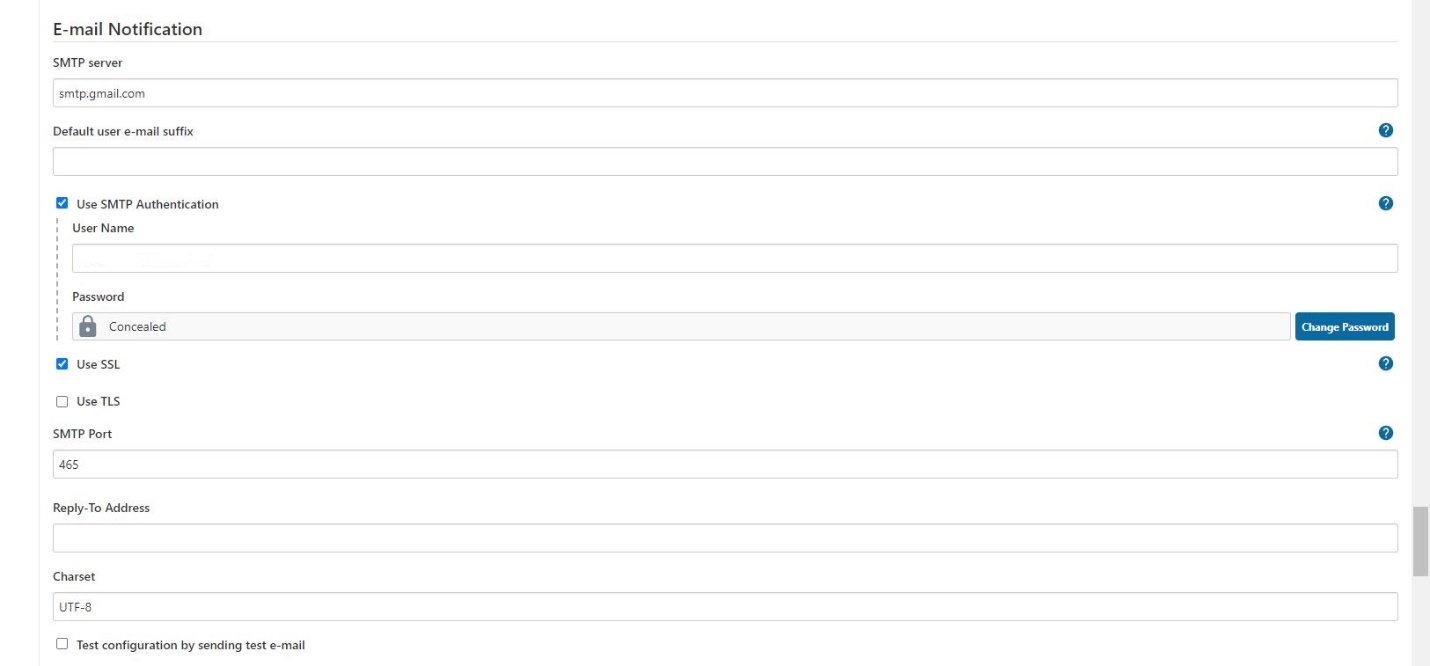
**Prerequisites:**

Ensure that you have a running Jenkins server and a Gmail account. Additionally, make sure that "insecure apps" access is enabled for your Gmail account.

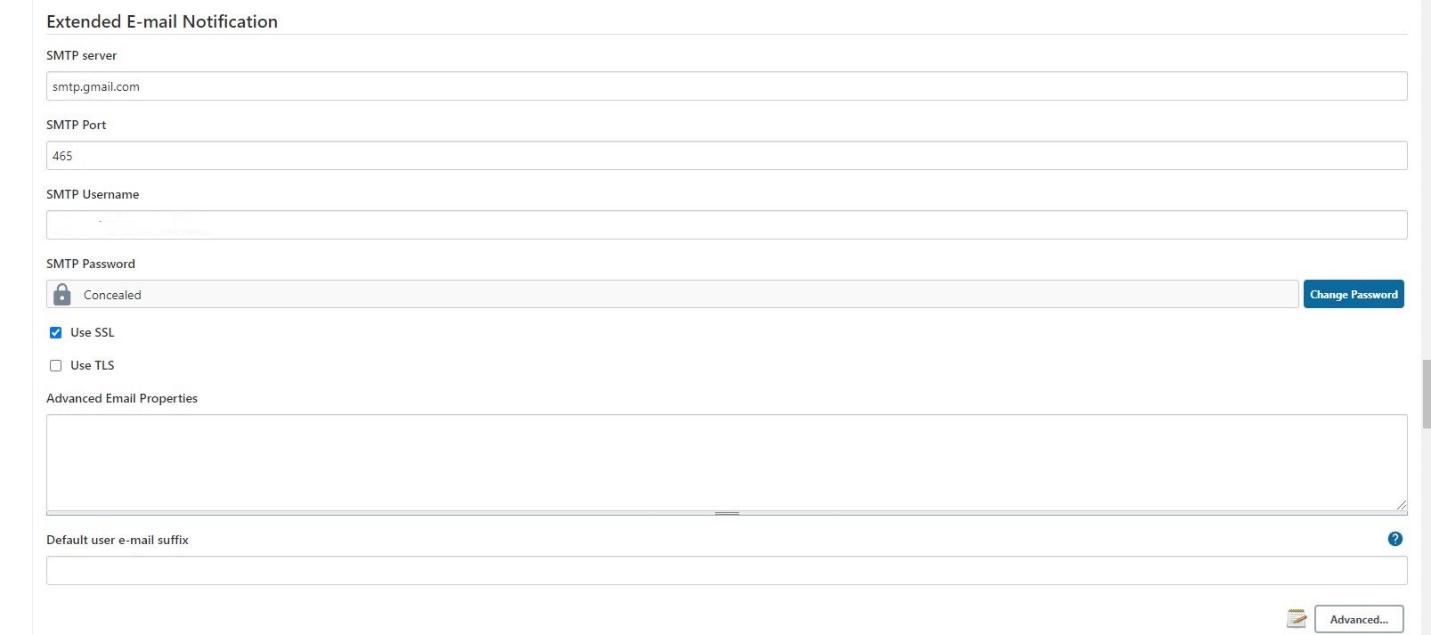
**Initial Setup:**

1. Install the "Email Extension Plugin" in Jenkins.
2. Once the plugin is installed in Jenkins, go to "Manage Jenkins" and click on "Configure System." In the "E-mail Notification" section, configure the following details as shown in the provided image.

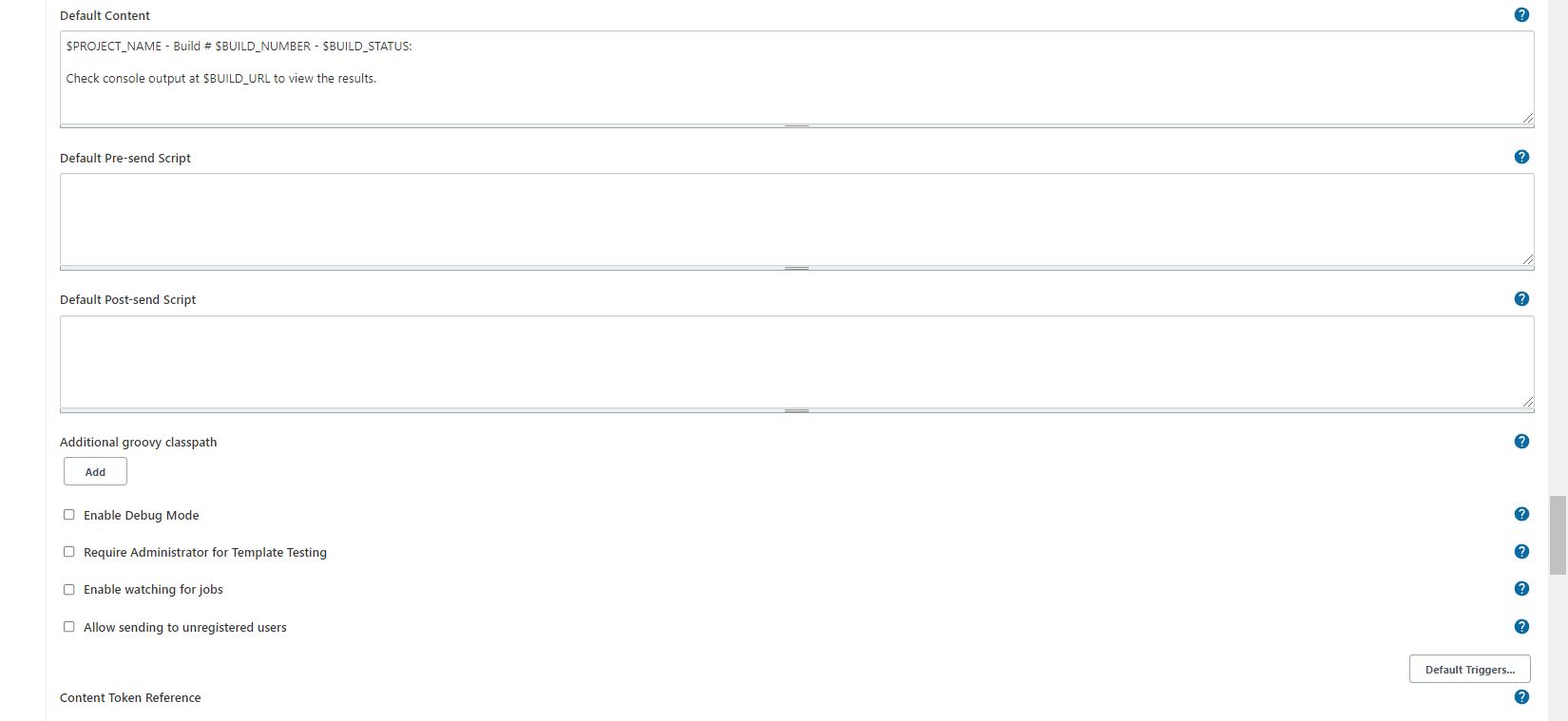
This is for verifying the mail configuration.



1. Under the "Extended E-mail Notification" section in the same configuration page, configure the details as shown in the provided images.







1. To send customized emails, you can use the following code as part of your Jenkins pipeline script:

post {

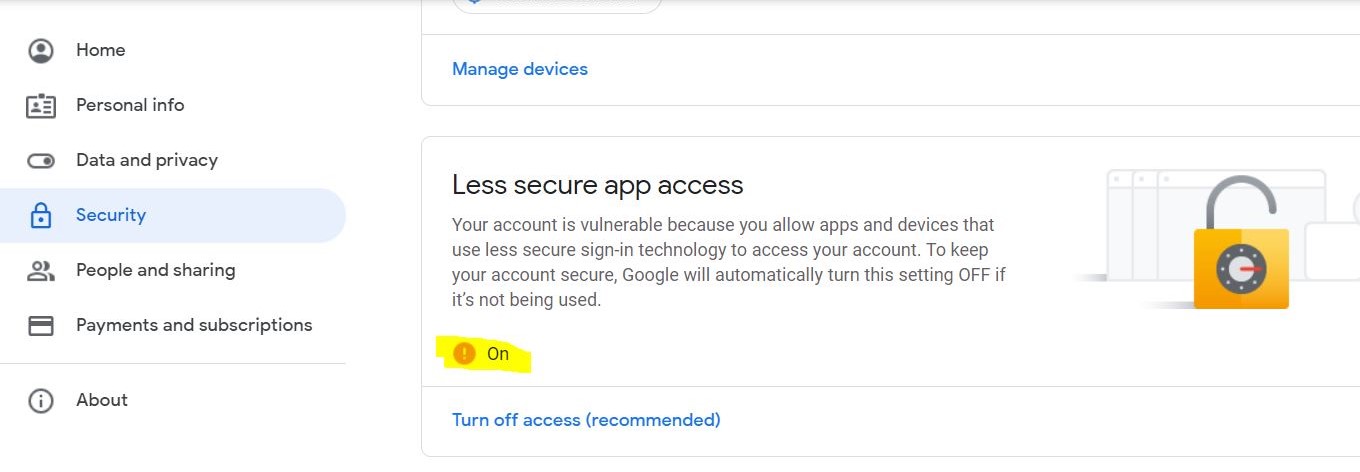
always {

mail bcc: '', body: "<br>Project: ${env.JOB\_NAME} <br>Build Number: ${env.BUILD\_NUMBER} <br> URL de build: ${env.BUILD\_URL}", cc: '', charset: 'UTF-8', from: '', mimeType: 'text/html', replyTo: '', subject: "${currentBuild.result} CI: Project name -> ${env.JOB\_NAME}", to: "anup.mhrzan7@gmail.com";

}

}

1. Make sure that the Gmail account used for authentication in Jenkins has "Less secure apps access" enabled. You can check and enable this setting in your Gmail account settings.



1. In some cases, you may need to perform additional settings for Gmail, such as allowing access from outside sources. This can be done by following the instructions at <https://g.co/allowaccess>. This step may be necessary to resolve certain issues.
2. You can customize the default email subject using variables such as **`$PROJECT\_NAME`**, **`$BUILD\_NUMBER`,** and **`$BUILD\_STATUS`.** For example:

* Default Subject: **`$PROJECT\_NAME - Build # $BUILD\_NUMBER - $BUILD\_STATUS!`**
* Default Subject (Alternative)**:`$PROJECT\_NAME - Build # $BUILD\_NUMBER - $BUILD\_STATUS:`**

1. To view the results of email notifications, check the console output at the provided **`$BUILD\_URL`.**

In summary, these steps guide you through configuring a Gmail-based mail server in Jenkins, allowing you to send customized email notifications as part of your Jenkins pipelines. Make sure to enable the necessary Gmail settings and perform any additional steps if required for external access.

## Configuring PR based trigger in Jenkins

**Prerequisites:**

Ensure you have a running Jenkins server and a GitHub repository.

**Setup:**

1. Install the required Jenkins plugins mentioned in the table below:

* GitHub Pull Request Builder (Version 1.42.1)
* Git (Version 4.2.2)
* GitHub Plugin (Version 1.30.0)
* GitHub API (Version 1.114.1)

**GitHub Pull Request Builder Plugin:**

* This plugin is essential for handling pull requests and triggering Jenkins builds based on them.
* It exposes a webhook URL (e.g., **<yourJenkins>/ghprbhook/**) that GitHub will use to send metadata.
* Configure this plugin to:
  + Add comments to your PRs requesting reviews.
  + Add a merge check to your branch protection settings.
  + Start Jenkins builds and report results to GitHub using its API.

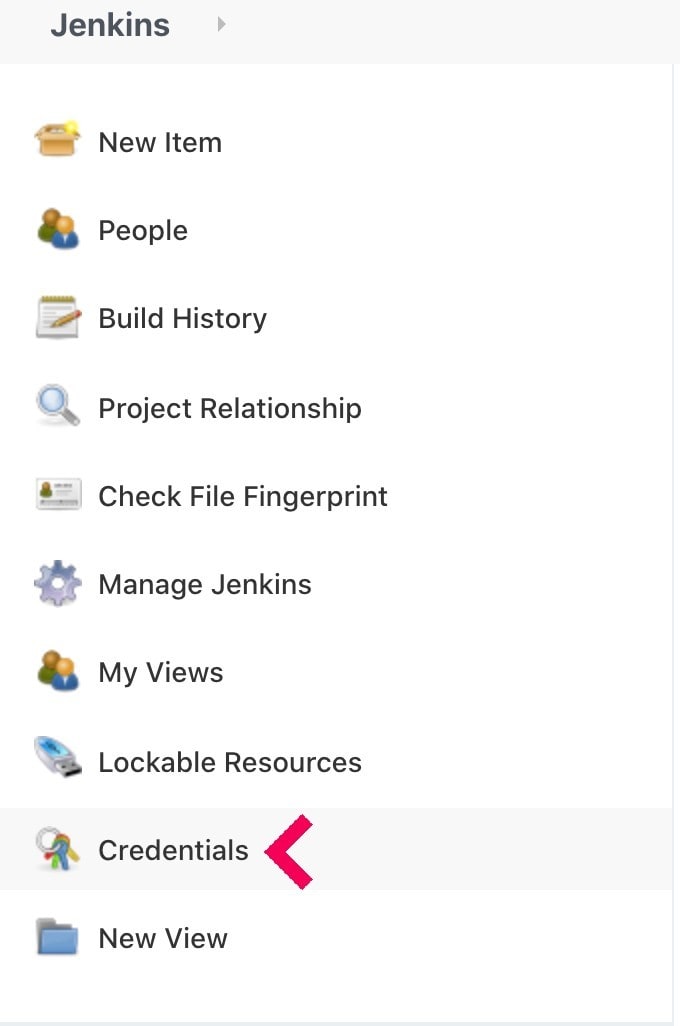
**Adding Credentials for GitHub Authentication:**

* You need credentials in Jenkins for two purposes:

1.1 Pulling your source code from GitHub.

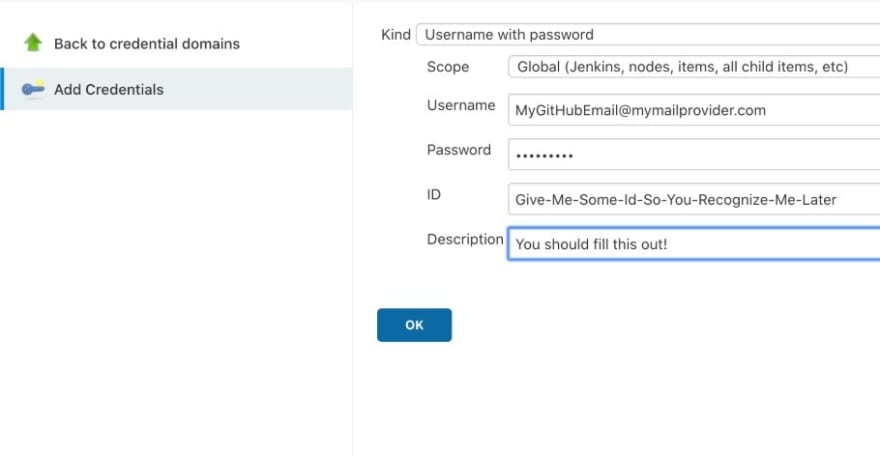
1.2 Using the GitHub API to comment on PRs and push merge check results.

* To add credentials, go to "Credentials" in the main Jenkins menu and choose a scope (e.g., global).
* Click "Add Credentials" and choose "Username with password" as the kind. Fill in the required information and save the credentials.
* You can also use an API token instead of a password for authentication if preferred.



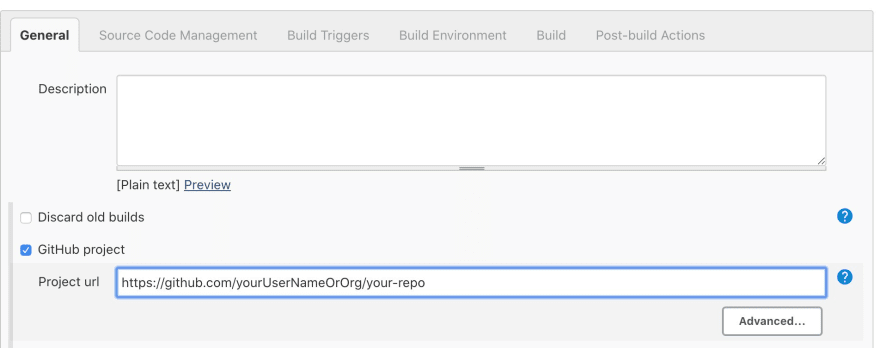






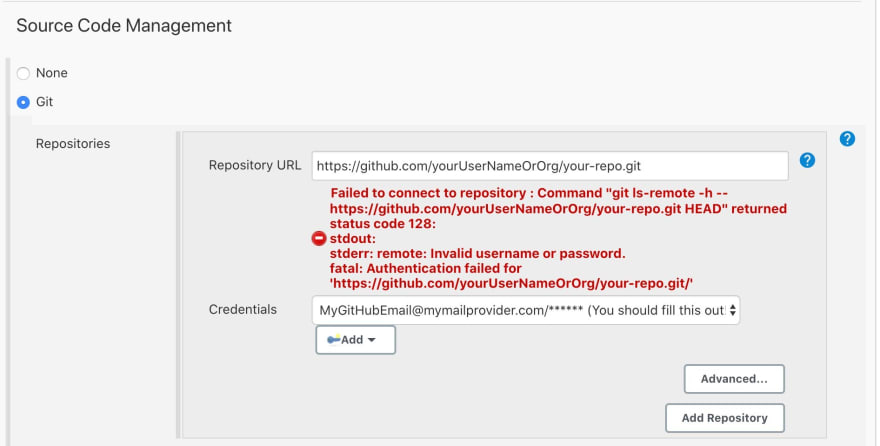
**Setting the GitHub Project URL:**

* Enter your GitHub project URL. This URL is crucial for routing and determining which project to build when the webhook receives a message. Leaving this field blank will prevent any builds from running.



**Adding the Repository:**

* To enable Jenkins to build your project, you must add the Git coordinates. Copy the HTTPS clone link from your GitHub project and select the credentials you created earlier.

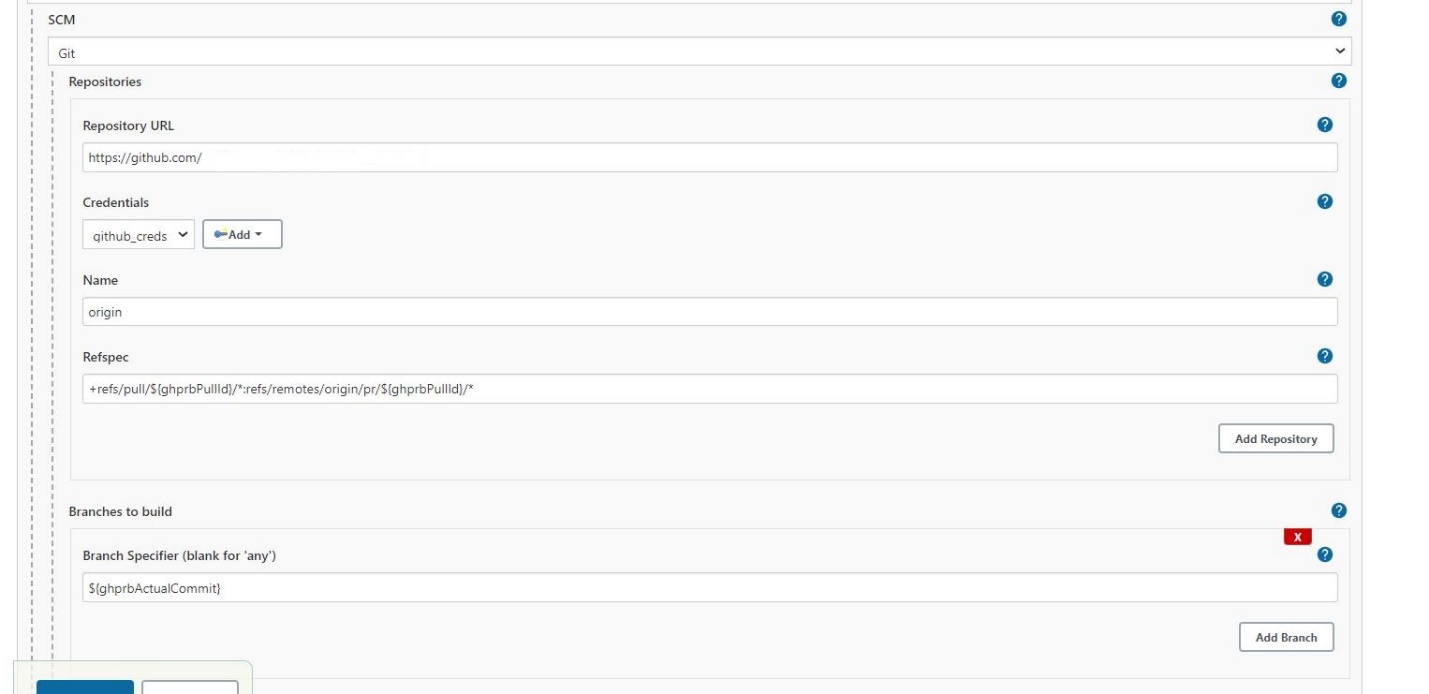


**Configuring the Branch Specifier and Refspec:**

* Configure the refspec as:

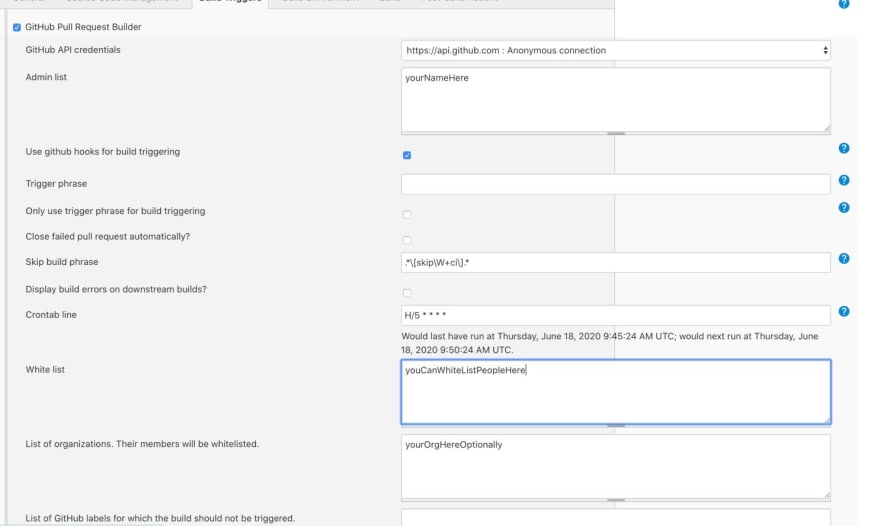
**+refs/pull/${ghprbPullId}/\*:refs/remotes/origin/pr/${ghprbPullId}/\***

* Set the Branch Specifier (blank for 'any') to: **${ghprbActualCommit}**



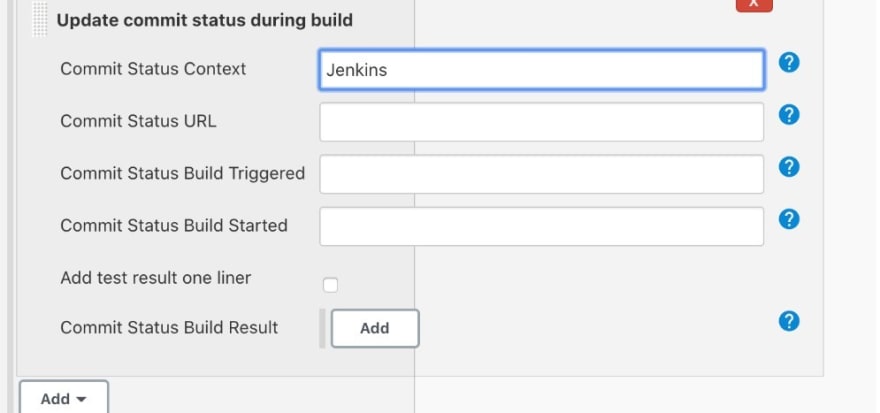
**Enabling GitHub Pull Request Builder:**

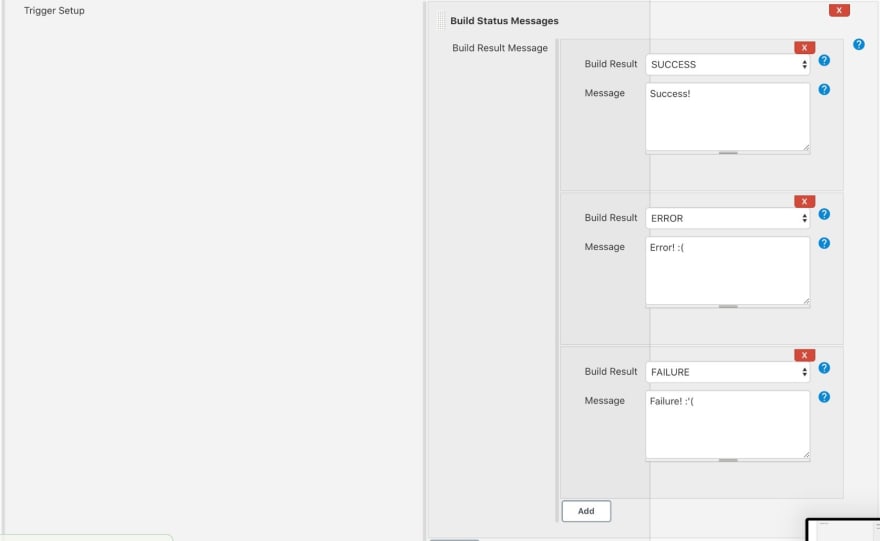
* Activate the "GitHub Pull Request Builder" within your Jenkins job.
* Enable the "Use github hooks for build triggering" option.
* For security reasons, you can restrict who can trigger builds. Avoid allowing everyone to prevent misuse or spam.



**Setting Up Trigger Context:**

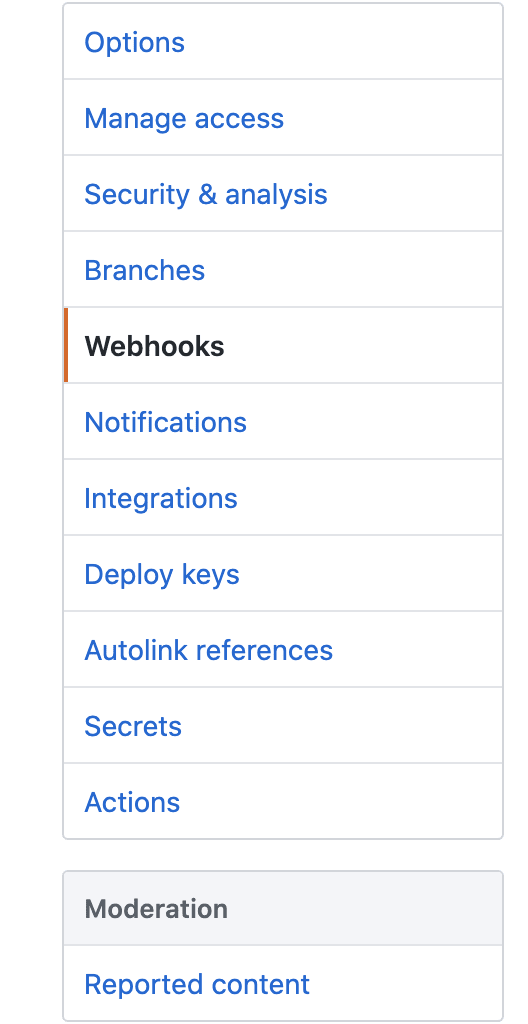
* Provide context for your builds by adding a string that will be displayed in merge check messages.
* Add triggers for "Update commit status during build" and "Build Status Messages" to keep your GitHub checks informative.



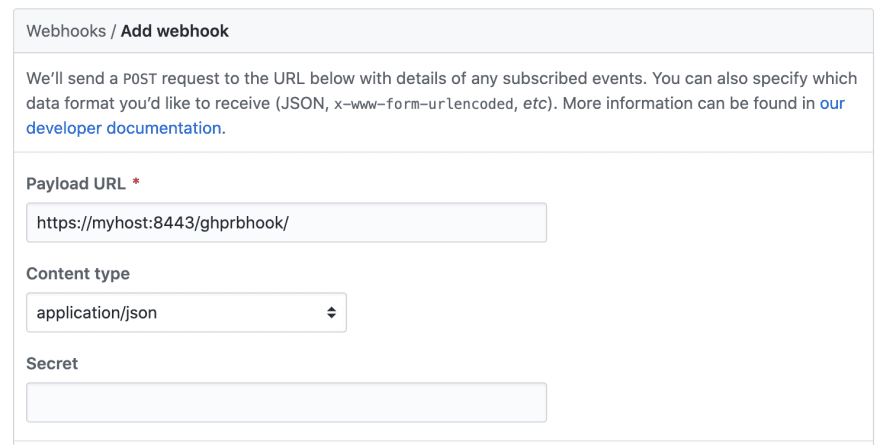


**Configuring GitHub Repository:**

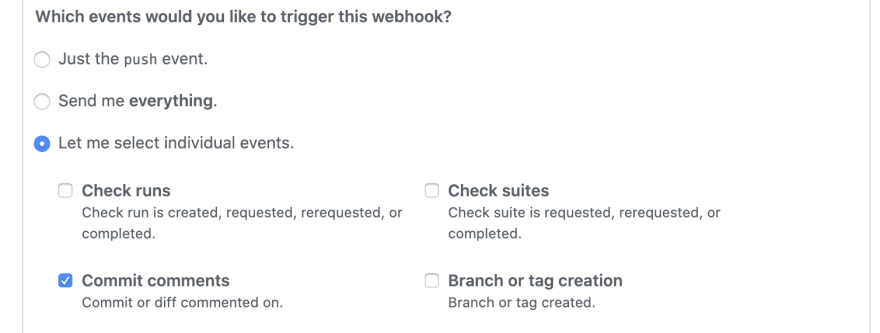
* In your GitHub repository, navigate to "Settings" and choose "Webhooks."



* Add a new webhook with your full Jenkins instance URL, including the webhook path **(http://Jenkins\_machine\_url:8080/ghprbhook/)** and select "application/json."



* Under "Which events would you like to trigger this webhook?", select "Commit comments" and "Pull requests."

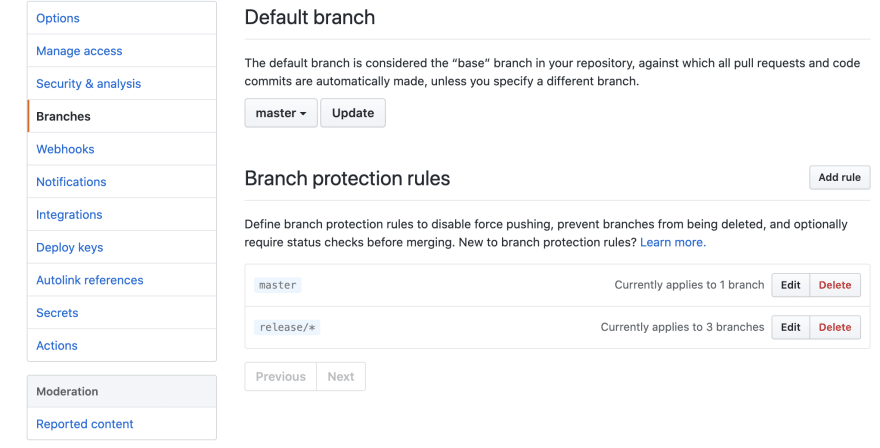


**Create a Test Pull Request:**

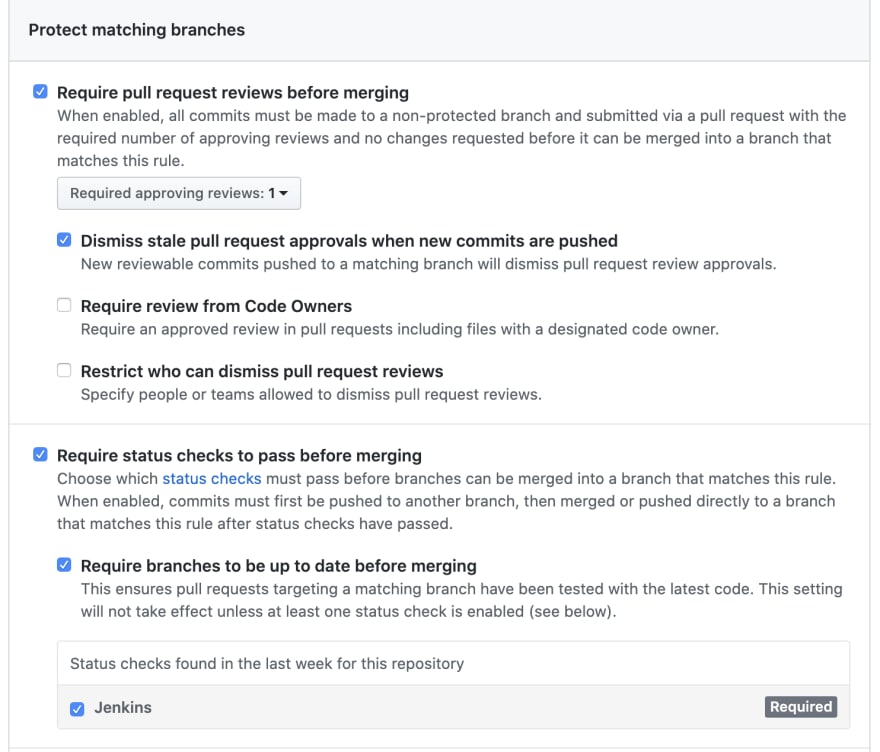
* Make a commit to your project on any feature branch and create a Pull Request for testing purposes. Jenkins should start the job you created. Verify that it runs successfully.

**Adding the PR Check:**

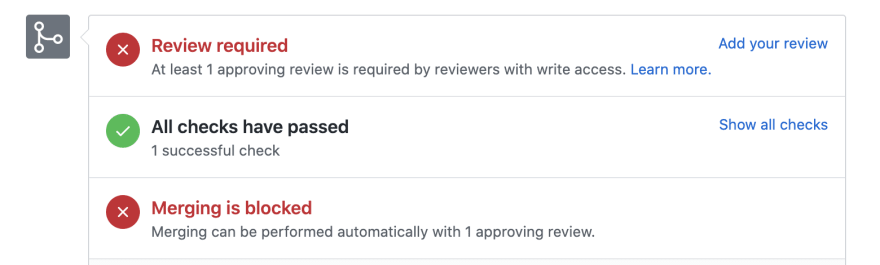
* To have checks on your pull requests, go to your repository's "Settings" and choose "Branches."



* Create a new protection rule targeting your base branch (e.g., main or master).
* Enable "Require status checks to pass before merging" and select your "Jenkins" context.



After completing these steps, your Pull Requests should have checks enabled, ensuring that builds and tests run before merging changes into your main branch.



## How to pull the images from private repository (nexus) to k8s kubeadm cluster

* create the secret in k8s cluster with proper details

*kubectl create secret docker-registry registry-secret --docker-server=nexus\_machine\_ip\_only:8083 --docker-username=admin --docker-password=admin --docker-email=not-needed@example.com*

* Once its done we can refer the same in k8s manifest files as below

apiVersion: v1

kind: Pod

metadata:

name: foo

spec:

containers:

- name: foo

image: nginx

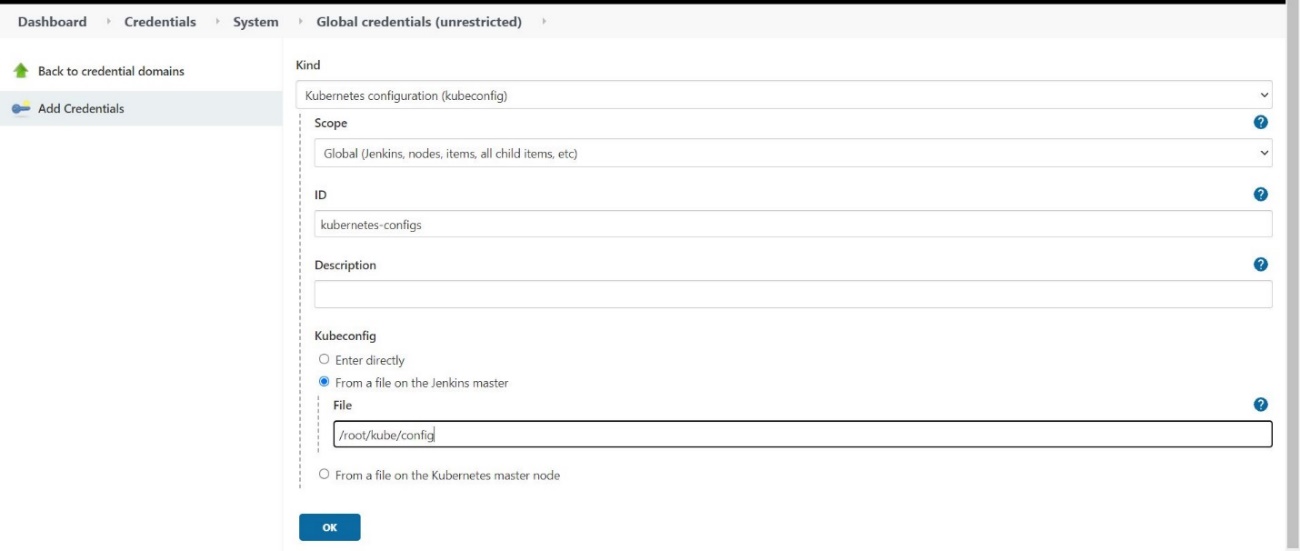
imagePullSecrets:

- name: registry-secret

## Connecting jenkins with kubernetes cluster

Install Kubernetes Continuous Deploy Plugin, once its installed successfully.

Govto manage jenkins --> manage credentials --> Click on jenkins global --> add credentials



select

**Kind** : Kubernetes Configuration

**Scope** : Global

**ID** : kubernetes-configs ( any meaningful name )

**Kubeconfig** : copy the content of kubernetes config file and place it at the file **/root/kube/config** in jenkins host

And click **OK**

and in Jenkinsfile you refer as mentioned below

stage('connecting to k8s cluster'){

steps{

script{

withCredentials([kubeconfigFile(credentialsId: 'kubernetes-config', variable: 'KUBECONFIG')]) {

dir ("kubernetes/"){

sh 'helm list'

sh 'helm upgrade --install --set image.repository="nexus\_ip:8083/springapp" --set image.tag="${VERSION}" myjavaapp myapp/ '

}

}

}

}

}

# Conclusion

In conclusion, the CI/CD workflow begins with code analysis using SonarQ, ensuring code quality. It then proceeds to Docker image creation, pushing the image to a private registry, and verifying Helm charts. A manual approval step guarantees deployment oversight. Subsequently, Helm charts are deployed to the Kubernetes cluster, and the application's correctness is verified. Depending on the verification outcome, the CI/CD job status is updated, and email notifications are sent accordingly. This workflow ensures a robust and controlled deployment process for your applications.

In closing, this documentation has provided with the essential knowledge and guidance needed to set up and harness the power of our CI/CD pipeline, which integrates Jenkins, Docker, Kubernetes, Helm, and GitHub. Let's recap the key takeaways from our journey:

**Summary of Key Points:**

We can learn how to configure a robust CI/CD pipeline that automates building, testing, and deploying applications, enabling faster and more reliable software delivery.

Prerequisites, installation, configuration and usage have been thoroughly covered, ensuring a comprehensive understanding of the pipeline.

Collaboration and continuous improvement are at the heart of DevOps, and our pipeline is designed to foster these principles.

This pipeline empowers to respond to user feedback and market demands with agility, maintain code quality, and reduce deployment risks.

**----THE END---**