Automated CI/CD Pipeline with Jenkins and GitHub Integration

In this project, we set up an **automated CI/CD pipeline** using **Jenkins and GitHub integration** to streamline the software development lifecycle. The pipeline automates the **building, testing, and deployment** of applications whenever code changes are pushed to a GitHub repository.

This documentation provides a **step-by-step guide** to:

- Installing and configuring **Jenkins** on an AWS EC2 instance.
- Integrating Jenkins with **GitHub Webhooks** for automated builds.
- Creating a **Jenkins job** to fetch, build, and deploy code changes.
- Automating the entire **CI/CD process** for seamless deployments.

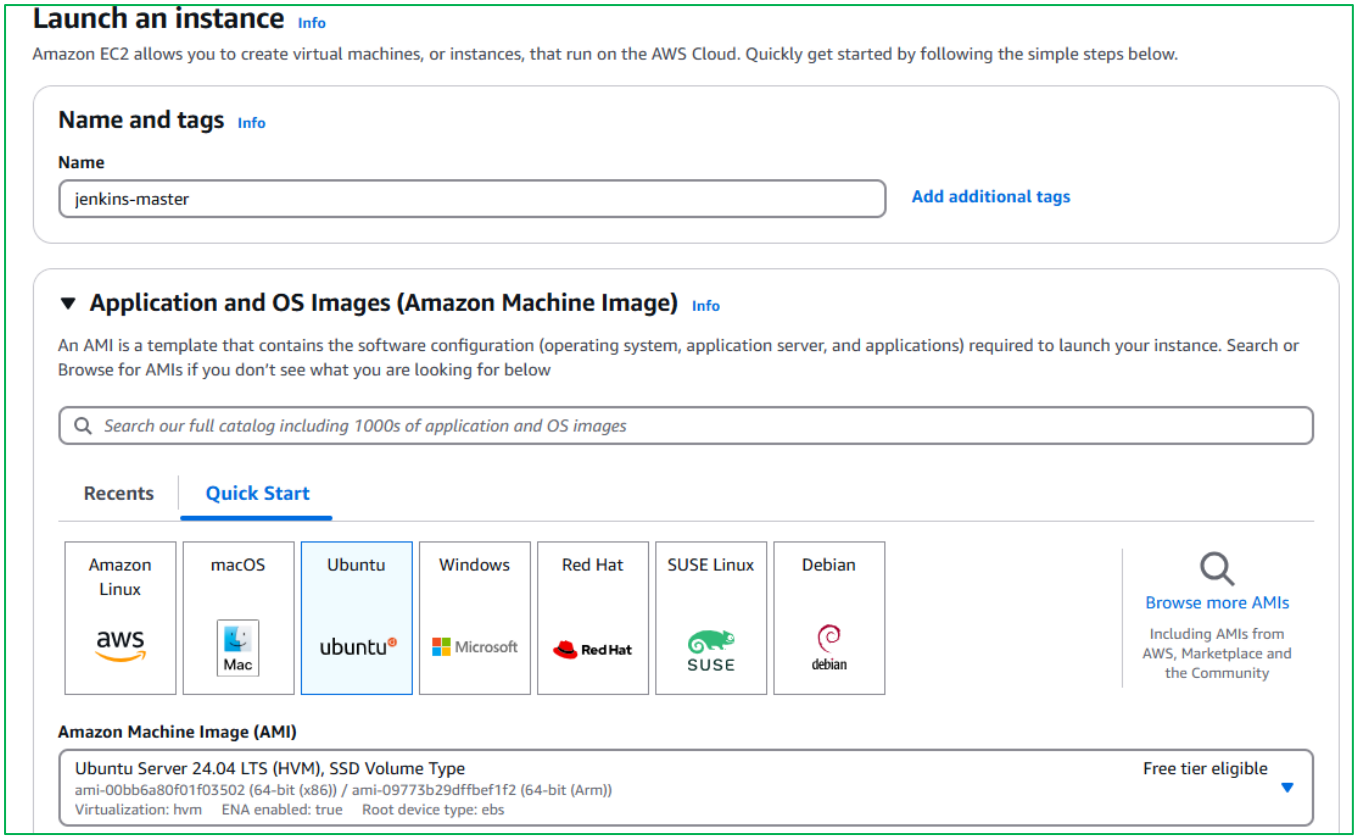
By the end of this project, you will have a fully functional **Jenkins CI/CD pipeline** that ensures continuous integration and deployment with minimal manual intervention.

Preliminary Steps: Preparing the Project from the External GitHub Repository

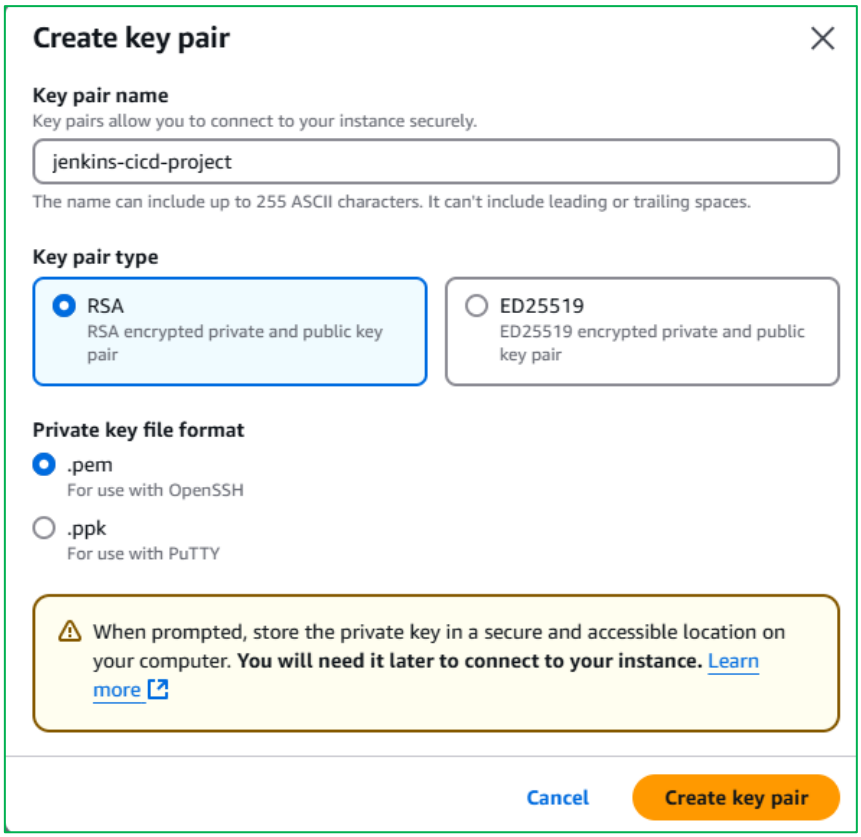
- Before starting with the deployment and other project-related tasks, you need to clone the project from the external GitHub repository available at **https://github.com/LondheShubham153/node-todo-cicd.git**. This repository contains the application that we will deploy. After cloning it to your local computer, you'll push the project to your own GitHub account. Once it's in your repository, we will proceed to execute the project from there.
- Once we have successfully cloned the repository to our local computer and pushed it to our own GitHub account, we can proceed with the next step: **Launching an AWS EC2 instance**. This instance will serve as the server where we will deploy and manage our application, setting up the necessary environment for Docker, Kubernetes, and Argo CD.

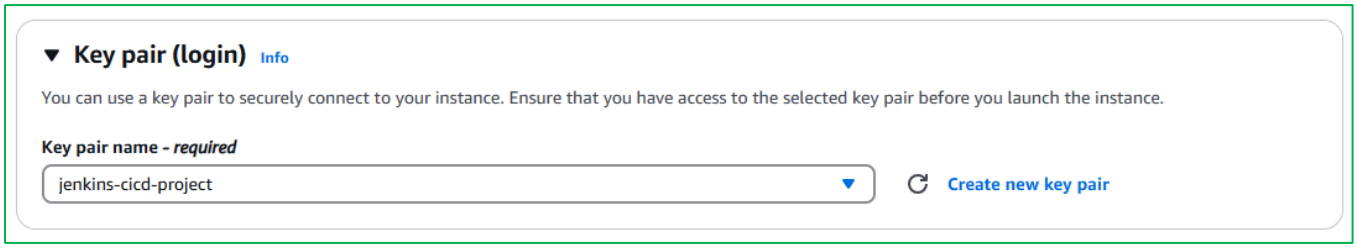
Launch an AWS EC2 instance

- **Enter the Name of the Instance**, eg: jenkins-master
- Choose **Ubuntu Server 24.04 LTS (HVM)** under **Amazon Machine Image(AMI)**

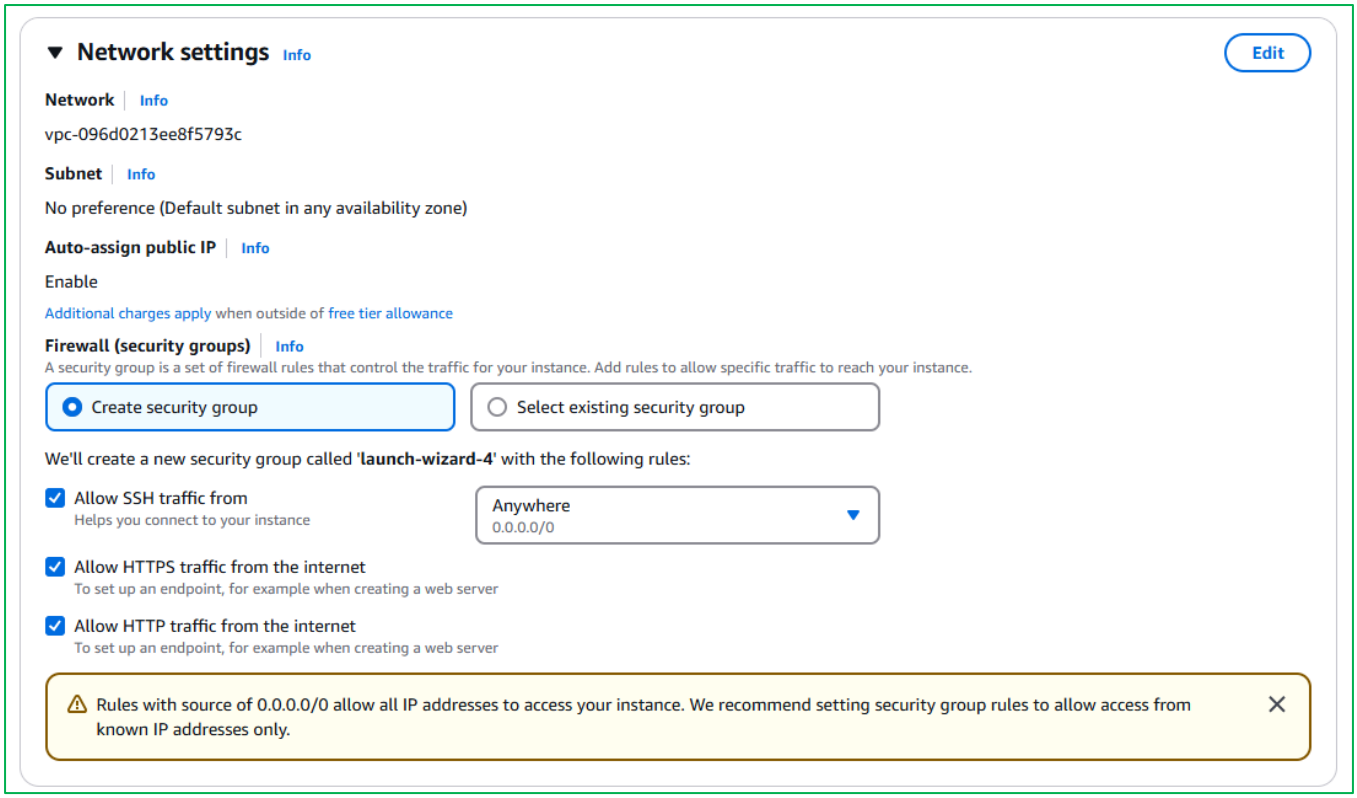


- Choose **t2.micro** under **Instance type**.
- Under **Key pair (login)**, give your key pair name or create a new key pair eg: jenkins-cicd-project is my keypair.

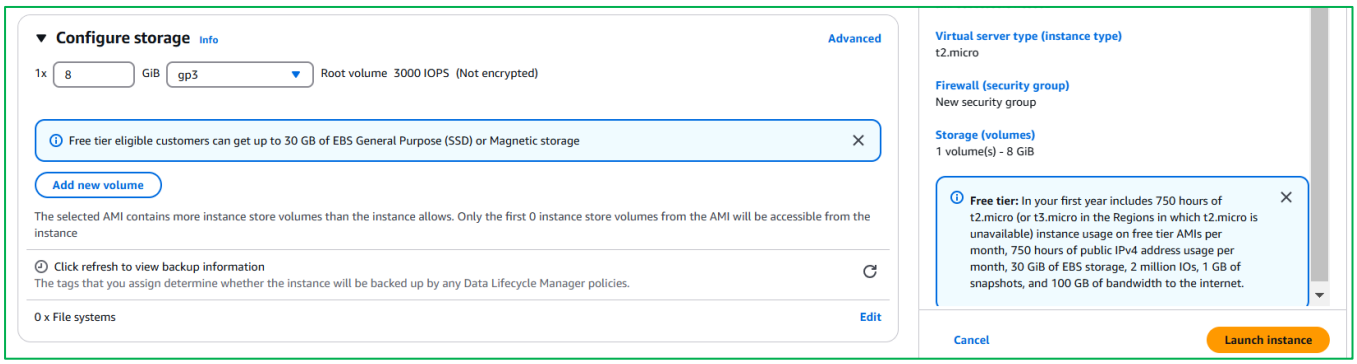




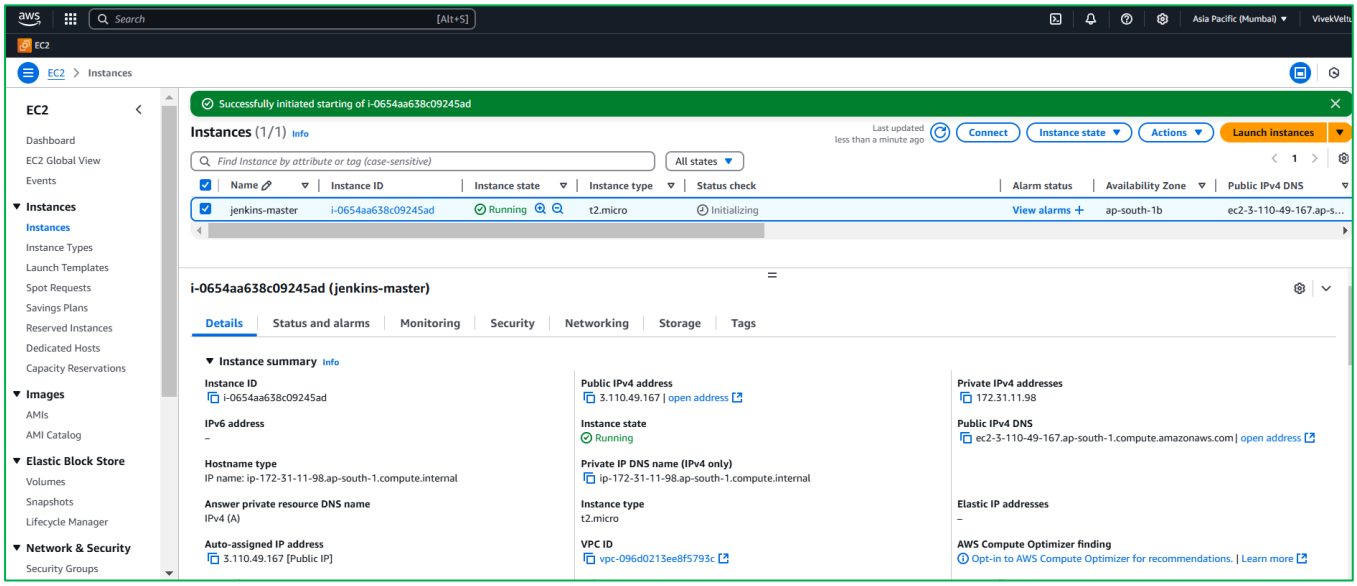
- In the network settings, choose the default VPC and subnet, and enable the option to auto-assign a public IP. For the firewall security groups, create a new security group and configure the following rules: allow **SSH** traffic from anywhere (port 22) to enable secure remote access, allow **HTTP** traffic from the internet (port 80) to support standard web traffic, and allow **HTTPS** traffic from the internet (port 443) for secure, encrypted web connections.

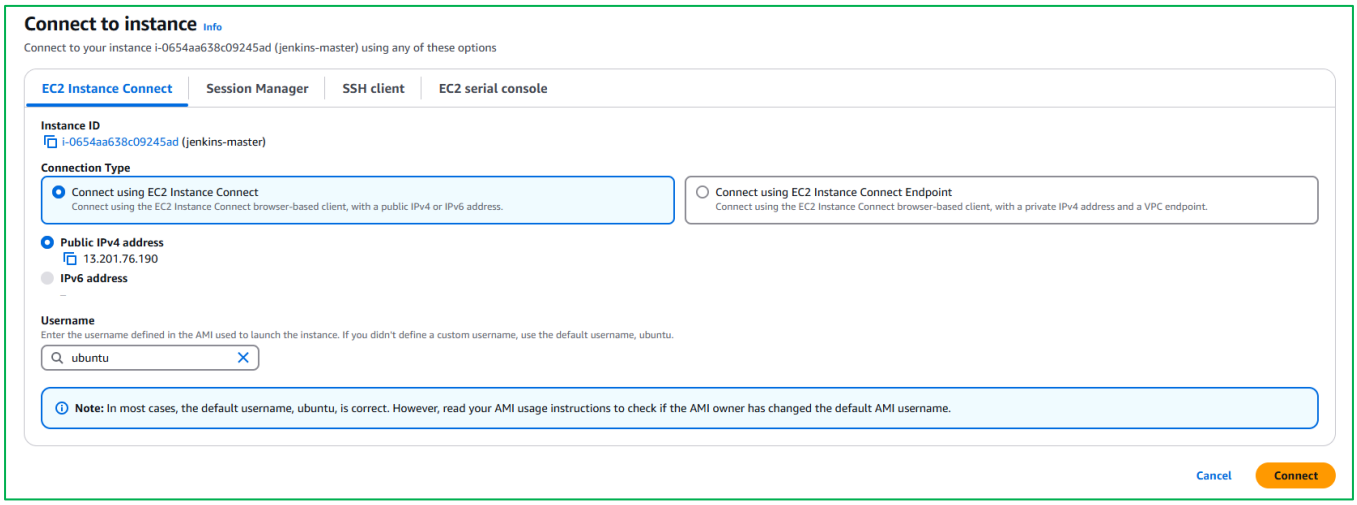


- Configure the storage settings by setting the root volume size to **8 GiB** to ensure sufficient space for the application and related dependencies.

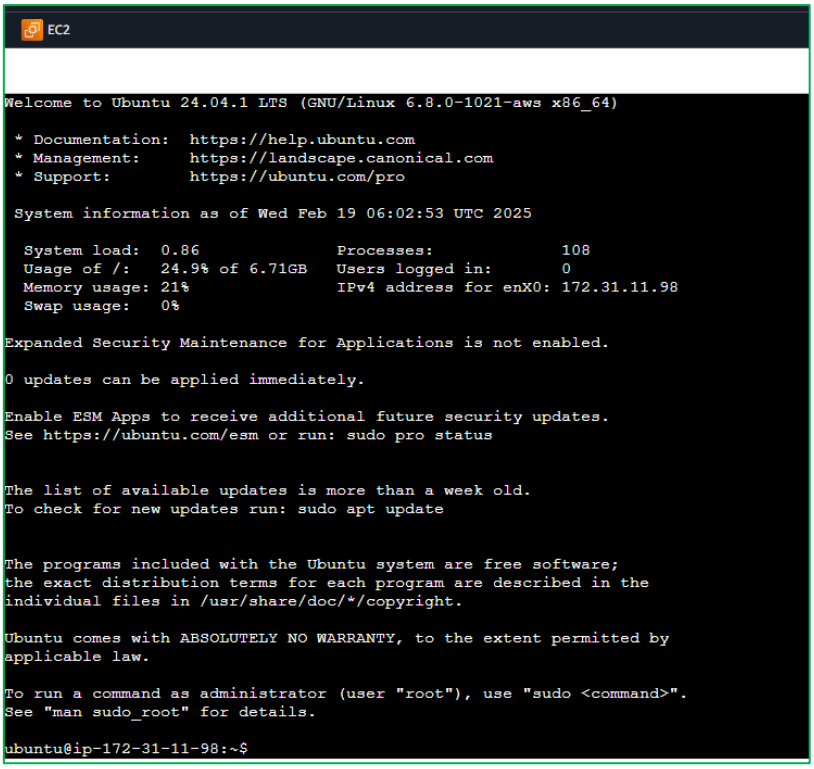


- After finalizing the storage configuration and reviewing all settings, proceed to **launch the EC2 instance**.



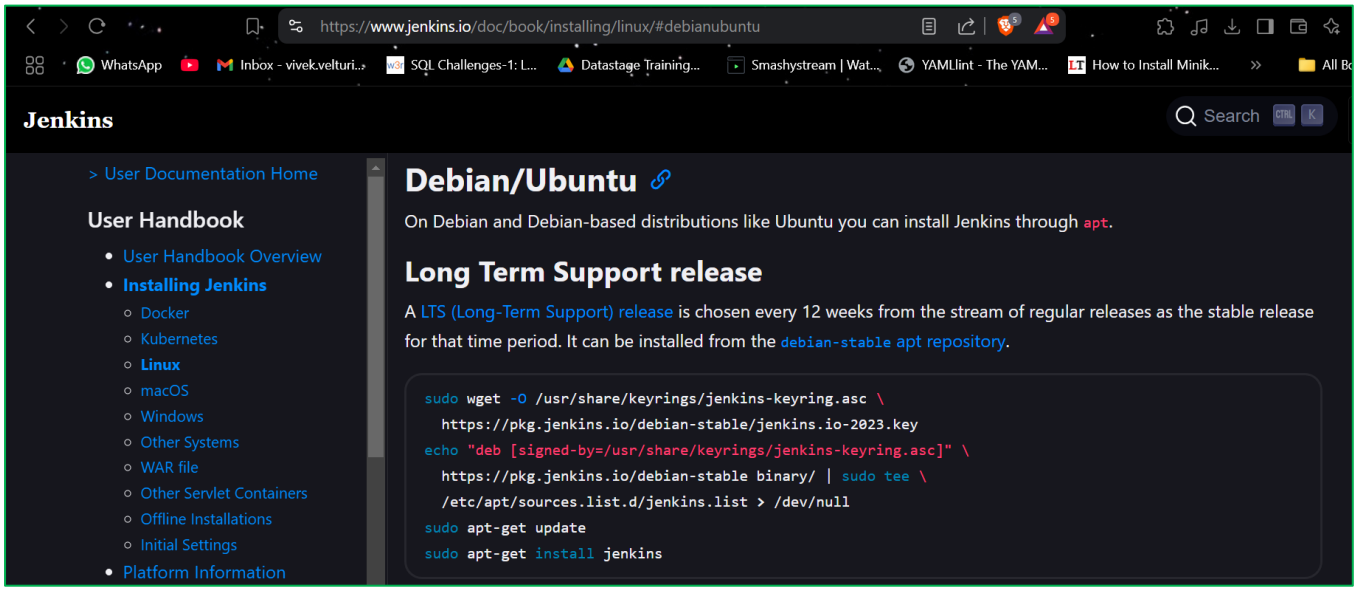


- To connect to the **Jenkins-Master EC2 instance**, select **EC2 Instance Connect** as the connection type and ensure that the **Public IPv4 Address** is correct. Use the default username **ubuntu**, unless your AMI specifies a different username. Once verified, click **Connect** to establish a secure browser-based SSH session, allowing you to configure and manage the jenkins-master instance efficiently.



Installing Jenkins on AWS EC2 instance

- For installing Jenkins on an **EC2 instance running Ubuntu**, we first navigate to the **official Jenkins documentation** to follow the recommended installation steps.



  - **Step 1: Choosing the Correct Installation Method**
    Since our EC2 instance is running Ubuntu, we select the Debian/Ubuntu installation method. Jenkins provides an LTS (Long-Term Support) release, which is updated every 12 weeks to ensure stability.
  - **Step 2: Adding the Jenkins Repository and Installing Jenkins**
    To install Jenkins, we follow these commands:

1. `sudo wget -O /usr/share/keyrings/jenkins-keyring.asc \ https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key`

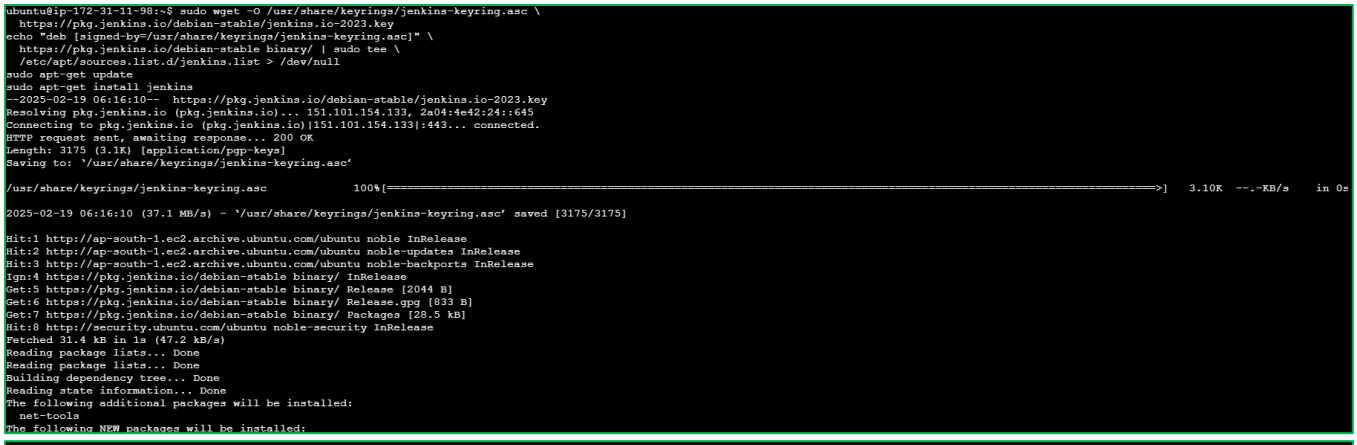
2. `echo "deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] \ https://pkg.jenkins.io/debian-stable binary/" | sudo tee \ /etc/apt/sources.list.d/jenkins.list > /dev/null`

3. `sudo apt-get update`

4. `sudo apt-get install Jenkins`

1. **Download the Jenkins key** – This ensures package authenticity.
2. **Add the Jenkins repository** – This allows the system to fetch Jenkins packages.
3. **Update the package list** – Ensures we get the latest version.
4. **Install Jenkins** – Downloads and installs the Jenkins service.

After running these commands, Jenkins will be installed on the EC2 instance. The next step is **starting Jenkins and accessing its web interface**.



- To start Jenkins, first, we need to run the command `sudo systemctl start jenkins`. This command initializes the Jenkins service, allowing it to run in the background. If Jenkins was not previously running, this will start the process and make it accessible.

```
ubuntu@ip-172-31-11-98:~$ sudo systemctl start jenkins
ubuntu@ip-172-31-11-98:~$
```

- Next, to ensure that Jenkins starts automatically every time the system reboots, we enable it using the command `sudo systemctl enable jenkins`. This command creates a symbolic link that registers Jenkins as a service that should start on boot, preventing the need to manually start it each time the machine restarts.

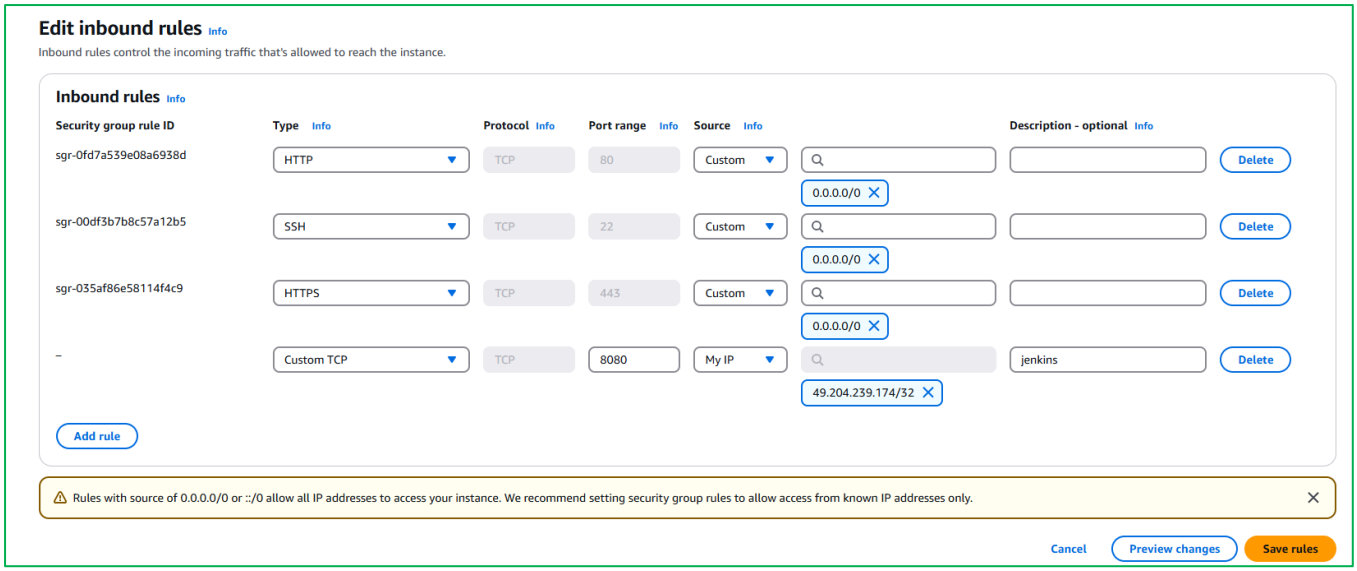
```
ubuntu@ip-172-31-11-98:~$ sudo systemctl enable jenkins
Synchronizing state of jenkins.service with SysV service script with /usr/lib/systemd/systemd-sysv-install.
Executing: /usr/lib/systemd/systemd-sysv-install enable jenkins
ubuntu@ip-172-31-11-98:~$
```

- After starting and enabling Jenkins, it is good practice to verify that the service is running properly. We can check the status by executing `sudo systemctl status jenkins`. If Jenkins is active, the output will indicate that the service is running, along with details such as the process ID and recent logs. If there are any issues, reviewing the status output can help diagnose errors or misconfigurations.

```
ubuntu@ip-172-31-11-98:~$ sudo systemctl status jenkins
● jenkins.service - Jenkins Continuous Integration Server
     Loaded: loaded (/usr/lib/systemd/system/jenkins.service; enabled; preset: enabled)
     Active: active (running) since Wed 2025-02-19 06:17:16 UTC; 4min 18s ago
   Main PID: 5814 (java)
      Tasks: 38 (limit: 1130)
     Memory: 317.3M (peak: 343.0M)
        CPU: 15.469s
     CGroup: /system.slice/jenkins.service
             └─5814 /usr/bin/java -Djava.awt.headless=true -jar /usr/share/java/jenkins.war --webroot=/var/cache/jenkins/war --httpPort=8080

Feb 19 06:17:08 ip-172-31-11-98 jenkins[5814]: 4cda11b51afa4baca2f9efe0a69c29de
Feb 19 06:17:08 ip-172-31-11-98 jenkins[5814]: This may also be found at: /var/lib/jenkins/secrets/initialAdminPassword
Feb 19 06:17:08 ip-172-31-11-98 jenkins[5814]: ************************************************************
Feb 19 06:17:08 ip-172-31-11-98 jenkins[5814]: ************************************************************
Feb 19 06:17:08 ip-172-31-11-98 jenkins[5814]: ************************************************************
Feb 19 06:17:16 ip-172-31-11-98 jenkins[5814]: 2025-02-19 06:17:16.567+0000 [id=30]       INFO        jenkins.InitReactorRunner$1#onAttained: Completed initialization
Feb 19 06:17:16 ip-172-31-11-98 jenkins[5814]: 2025-02-19 06:17:16.598+0000 [id=23]       INFO        hudson.lifecycle.Lifecycle#onReady: Jenkins is fully up and running
Feb 19 06:17:16 ip-172-31-11-98 systemd[1]: Started jenkins.service - Jenkins Continuous Integration Server.
Feb 19 06:17:18 ip-172-31-11-98 jenkins[5814]: 2025-02-19 06:17:18.194+0000 [id=46]       INFO        h.m.DownloadService$Downloadable#load: Obtained the updated data file for hudson.tasks.Maven.MavenInstal
Feb 19 06:17:18 ip-172-31-11-98 jenkins[5814]: 2025-02-19 06:17:18.197+0000 [id=46]       INFO        hudson.util.Retrier#start: Performed the action check updates server successfully at the attempt #1
lines 1-20/20 (END)
```

- Now, the Jenkins service is active and running, but we need to confirm the exact port it is using before updating the security group. By default, Jenkins runs on port 8080, but we can verify this by checking its configuration.
  To check the configured port, we can run `sudo cat /etc/default/jenkins | grep HTTP_PORT`. This command displays the port Jenkins is set to use. If it shows `HTTP_PORT=8080`, Jenkins is running on port 8080.

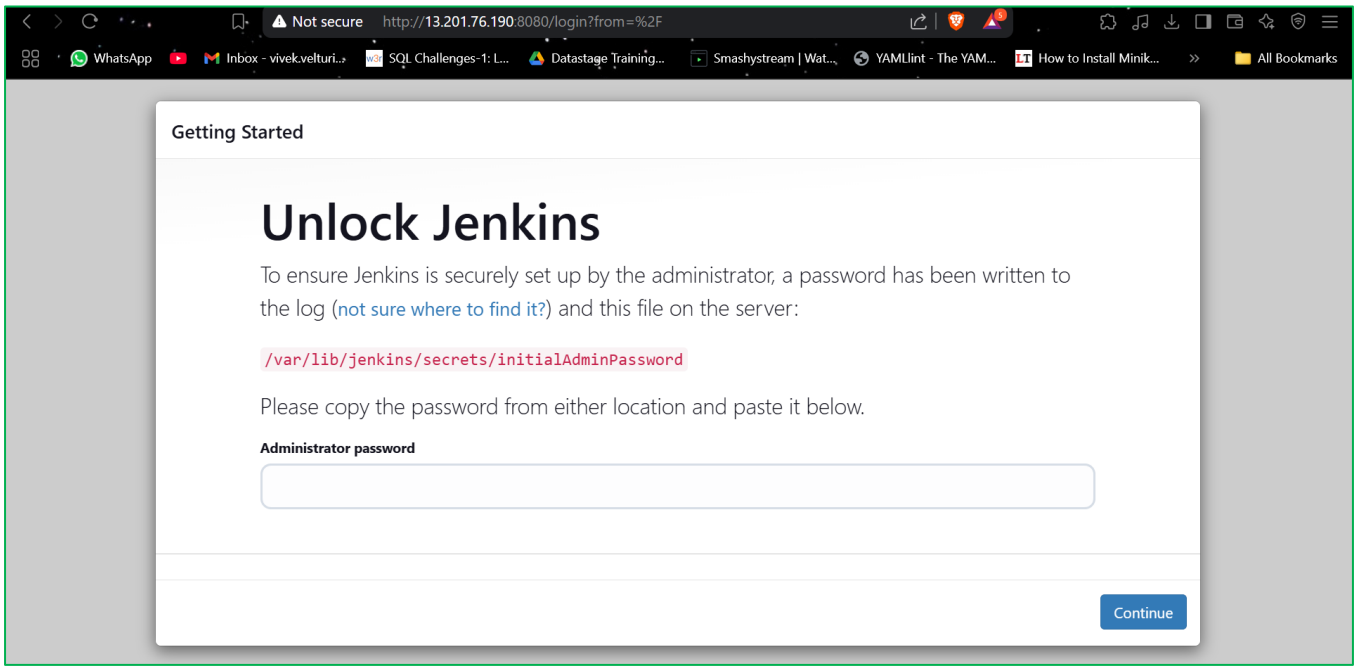
- Once we confirm the port, we need to update the security group of our EC2 instance (`jenkins-master`) to allow access only from our IP.
  To do this,
  we go to the **EC2 Dashboard** in the AWS Management Console,
  select the `jenkins-master` instance, and open its **Security Group** settings.
  Under **Inbound Rules**, we click **Edit inbound rules**,
  add a new rule,
  set the **port range** to **8080**,
  enter **Jenkins** as the **description**,
  select **My IP** as the **source**. This ensures that only our current public IP can access Jenkins.



- **After saving the changes**, Jenkins will be accessible at `http://<instance-public-ip>:8080`. If access issues persist, we should verify firewall settings and confirm that Jenkins is running.

Setting-Up Jenkins

- Now, the Jenkins service is active and accessible via the instance's public IP on port 8080. When we access `http://<instance-public-ip>:8080` in a web browser for the first time, Jenkins presents the **"Getting Started"** page, prompting us to unlock Jenkins by entering the administrator password.

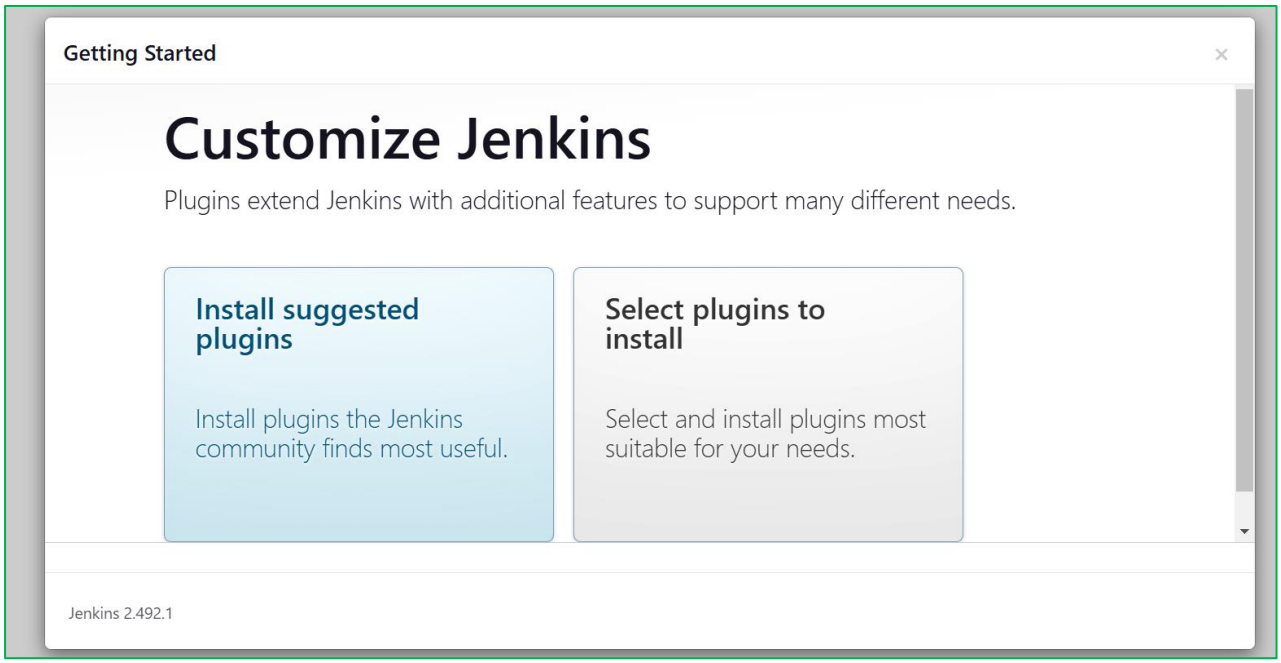


- Jenkins provides the path where the initial administrator password is stored on the instance. To retrieve this password, we need to access the instance via SSH and use the command `sudo cat /var/lib/jenkins/secrets/initialAdminPassword`. This

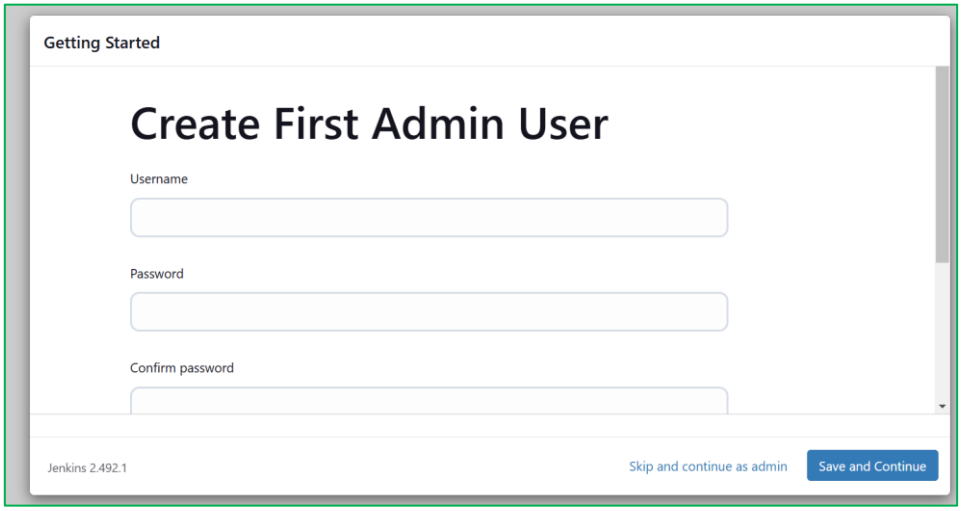
command reads the contents of the file where Jenkins has stored the password. Once executed, it displays the administrator password, which we then copy and paste into the Jenkins setup page to unlock Jenkins.

```
ubuntu@ip-172-31-11-98:~$ sudo cat /var/lib/jenkins/secrets/initialAdminPassword
4cda11b51afa4baca2f9efe0a69c29de
ubuntu@ip-172-31-11-98:~$
```

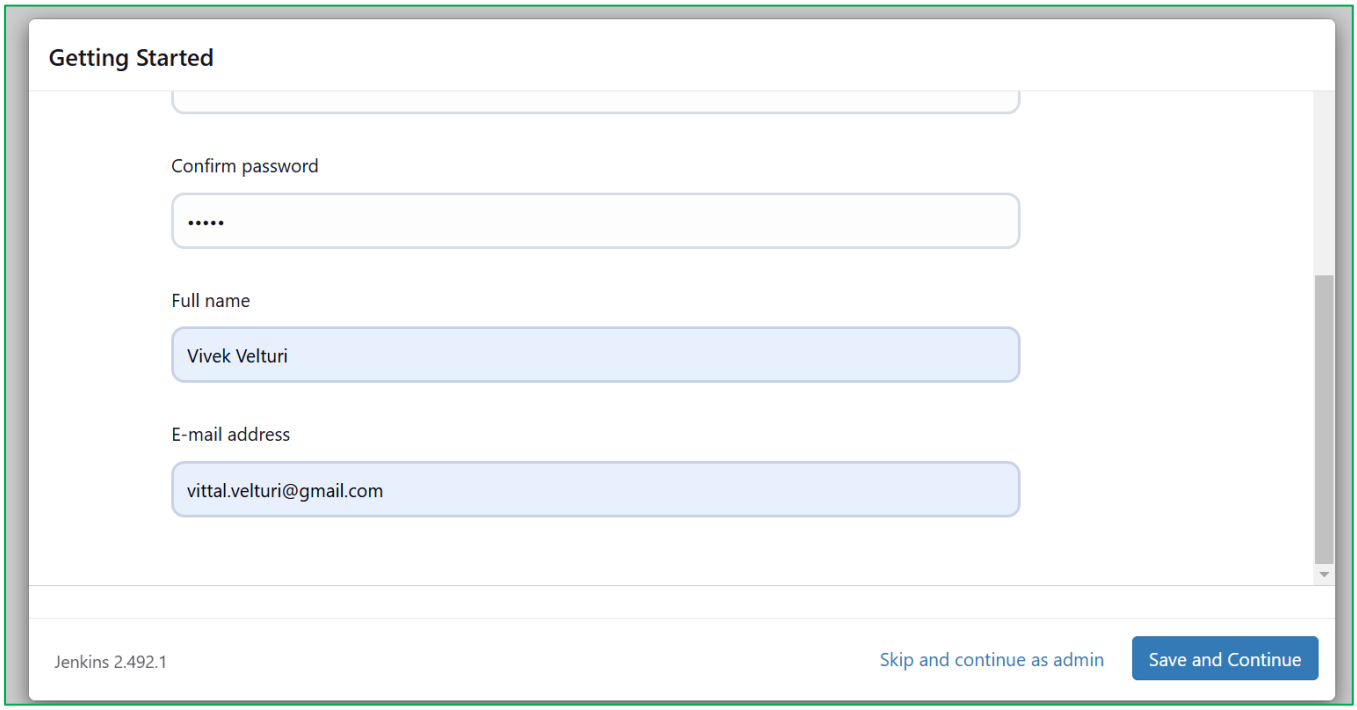
- Now, after unlocking Jenkins, we reach the **Customize Jenkins** page, where we choose **"Install Suggested Plugins"** to proceed with the default setup. Jenkins then begins installing essential plugins, including integrations like Git, Pipeline, and Credentials management. This process may take a few minutes, depending on system resources and network speed.



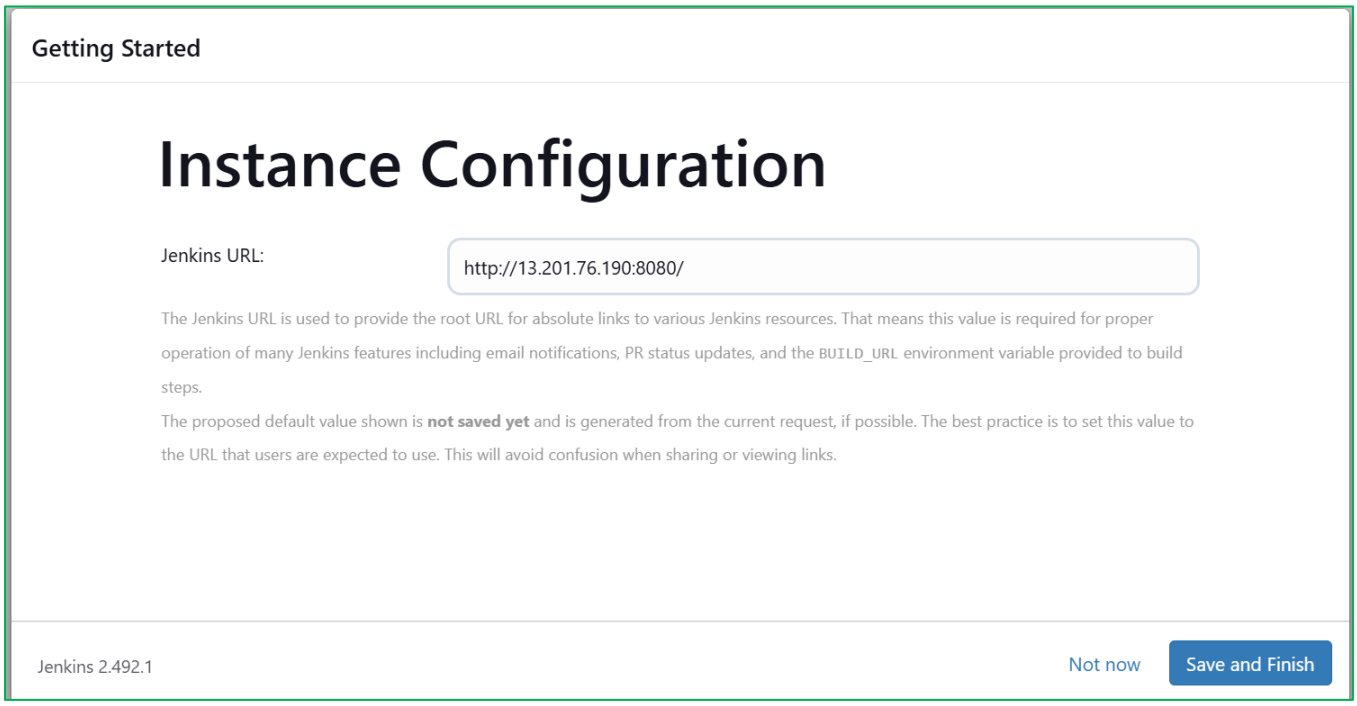
- Once the installation is complete, Jenkins moves to the next step, prompting us to create the first admin user. If any plugins fail to install, we can retry or install them later through **Manage Jenkins** in the dashboard.



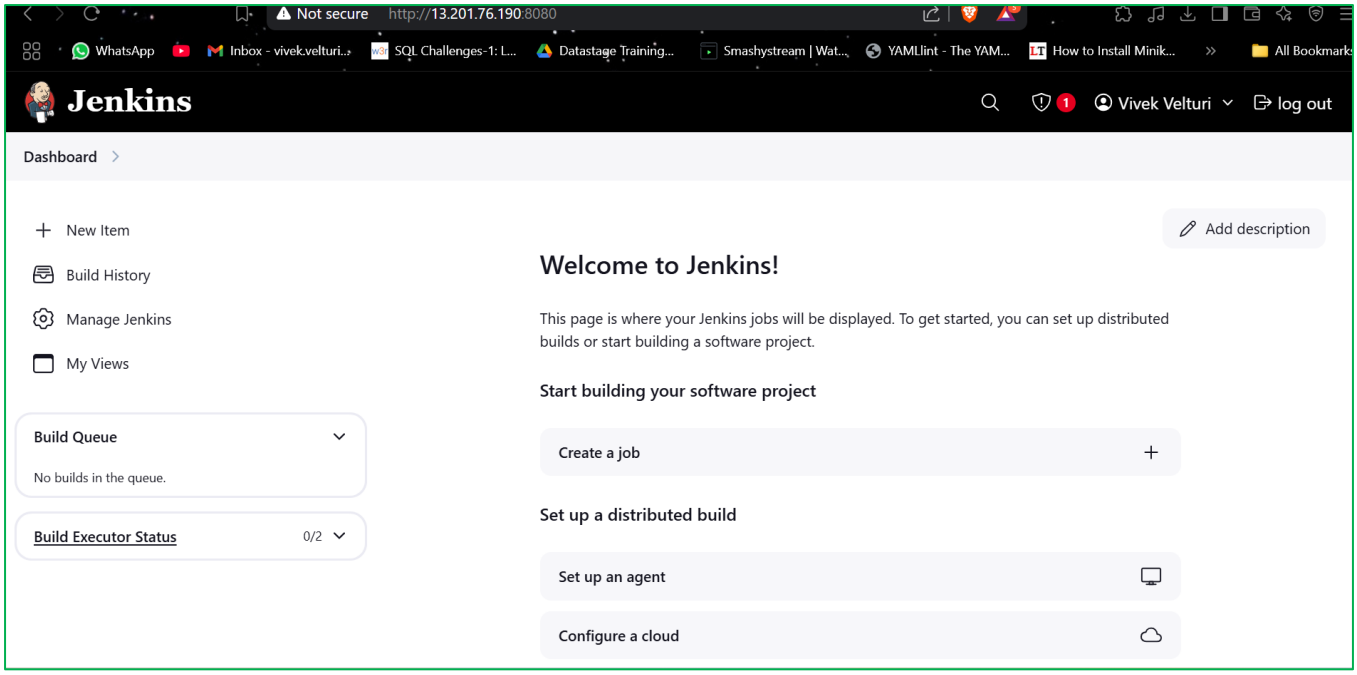
- Now, after the plugin installation is complete, Jenkins prompts us to create the first admin user. We enter our desired **username, password, full name, and email address**, then click **Save and Continue** to proceed.



- Next, the **Instance Configuration** page appears, displaying the Jenkins URL. We verify that the URL is correct, then click **Save and Finish** to complete the setup.

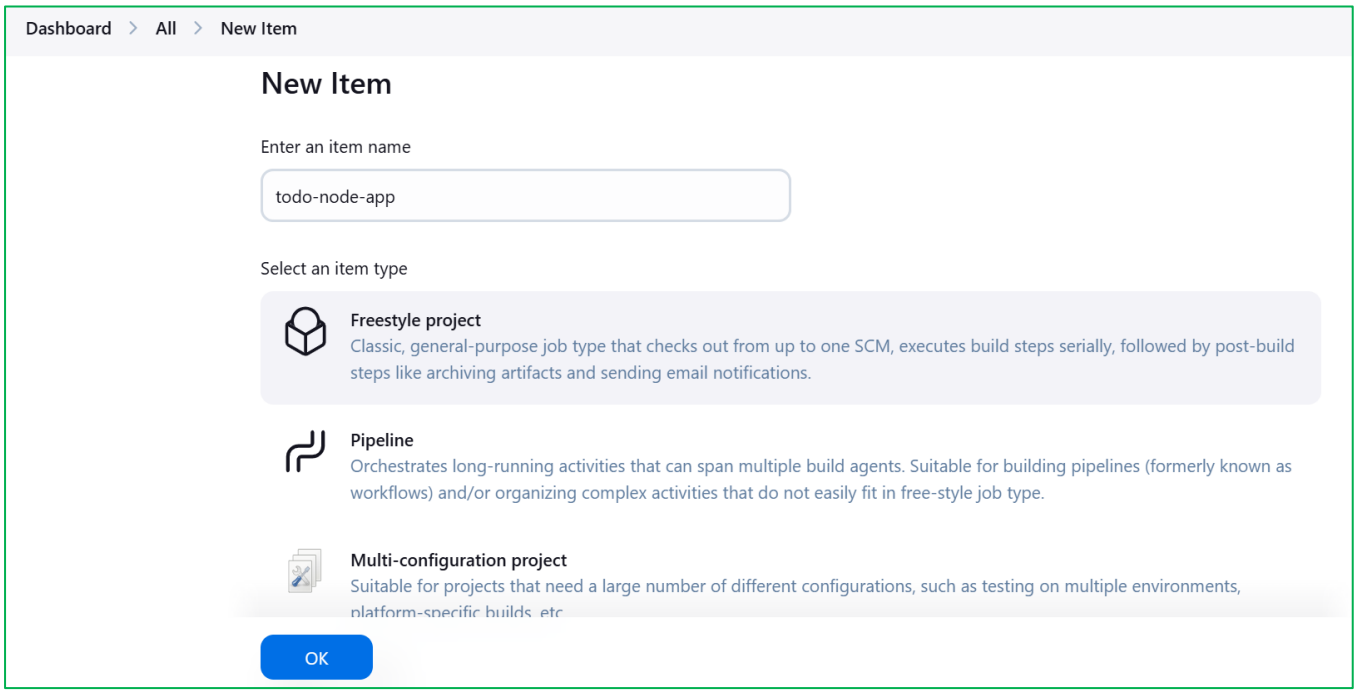


- With this, Jenkins is fully configured, and we can now access the dashboard using our newly created **username and password**.

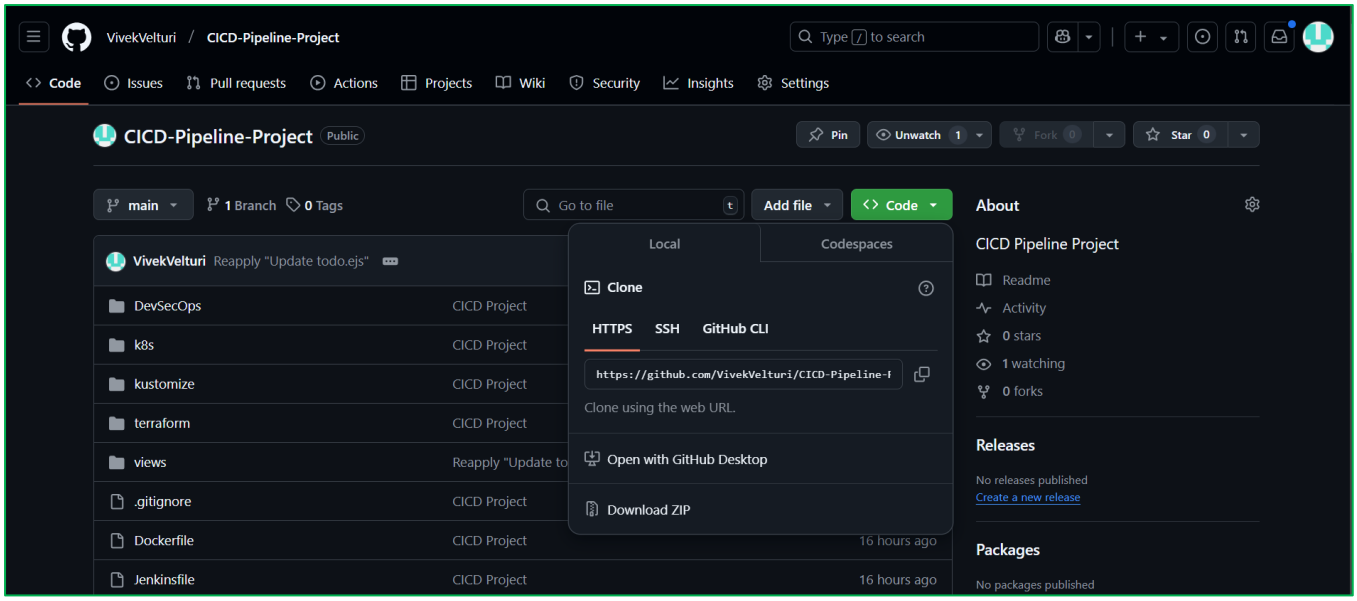


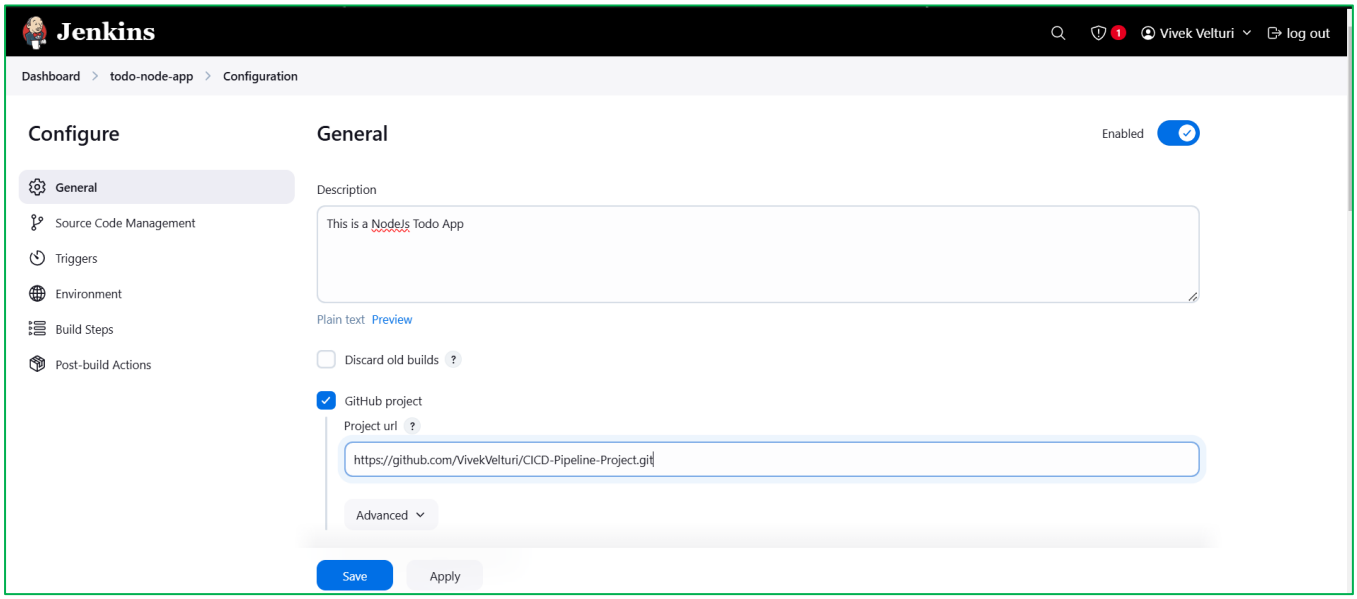
Configuring a Job in Jenkins

- To create a new job in Jenkins, go to the **Jenkins Dashboard** and click on **"New Item"** in the left-hand menu. On the next page, enter **"todo-node-app"** as the item name and select **"Freestyle project"** as the item type. Then, click **OK.**



- Once the item is created, you'll be taken to the configuration page. In the **Description** field, enter: **"This is a NODEJS Todo App"**.
  Next, under the **General section**, select **GitHub Project** and **enter the URL of the GitHub repository** that contains the code for todo-node-app (the repository you cloned earlier and pushed to your own GitHub account





- Once the repository URL is entered, you can proceed with other configurations. To enable secure communication between GitHub and Jenkins, **an SSH key** was generated and integrated into both GitHub and Jenkins. This ensures Jenkins can securely access the GitHub repository without exposing sensitive credentials, allowing Jenkins to pull the code over SSH when the job runs.

Setting Up SSH Key Configuration

- To generate an SSH key for secure GitHub integration with Jenkins, follow these steps:
    - first open the terminal on your Jenkins server (instance: jenkins-master) and run the command `ssh-keygen`. This command will generate a new SSH key pair, consisting of a public and a private key.(in our case by default ed25519 algorithm is used for key generation)

- You will then be prompted to choose a location to save the SSH key. By default, the key is saved in the **`~/.ssh/`** directory.
  To accept the default location, press **Enter**. This will store the key as **`/home/ubuntu/.ssh/id_ed25519`**.

- Next, you will be asked to enter a passphrase for the key. This is optional, and for extra security, you can set a passphrase. However, if you prefer not to use a passphrase, you can leave it empty and press **Enter**.
- After the key is generated, you will see a confirmation output. This includes the location where the keys are saved and a **key fingerprint**.
- The **private key** is stored in **`id_ed25519`**, and the **public key** is stored in **`id_ed25519.pub`**.
- The output will also display a **randomart image**, which is a visual representation of the key fingerprint for easy verification.

```
ubuntu@ip-172-31-11-98:~$ ssh-keygen
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/ubuntu/.ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/ubuntu/.ssh/id_ed25519
Your public key has been saved in /home/ubuntu/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:pKJRqIHd4njXuEOqwoL/na8f4QY6A5e9HoARV1dcgBs ubuntu@ip-172-31-11-98
The key's randomart image is:
+--[ED25519 256]--+
|  . ... .+oo.    |
|.. =   .E .      |
|o = o   .o       |
| = = = o.        |
|o * O = S        |
| . O = + .       |
|o o * o +        |
|+o   * + .       |
|+.... =+o        |
+----[SHA256]-----+
ubuntu@ip-172-31-11-98:~$
```

- To view the SSH keys that were generated, first, navigate to the SSH directory by running the command `cd ~/.ssh`.
  This will take you to the directory where the keys are stored by default.
- Once inside the directory, use the `ls` command to list all the files.
  You should see both the **public** and **private** keys listed,
  named **`id_ed25519`** (private key) and **`id_ed25519.pub`** (public key).

```
ubuntu@ip-172-31-11-98:~$ cd .ssh
ubuntu@ip-172-31-11-98:~/.ssh$ ls
authorized_keys  id_ed25519  id_ed25519.pub
ubuntu@ip-172-31-11-98:~/.ssh$
```

- To view the **public key**, use the `cat` command
  by running `cat id_ed25519.pub`.
  This will display the contents of the public key, which you will need to add to your GitHub account to establish secure authentication.
  If you need to view the **private key** (though this should remain secure and never be shared), you can use `cat id_ed25519`.

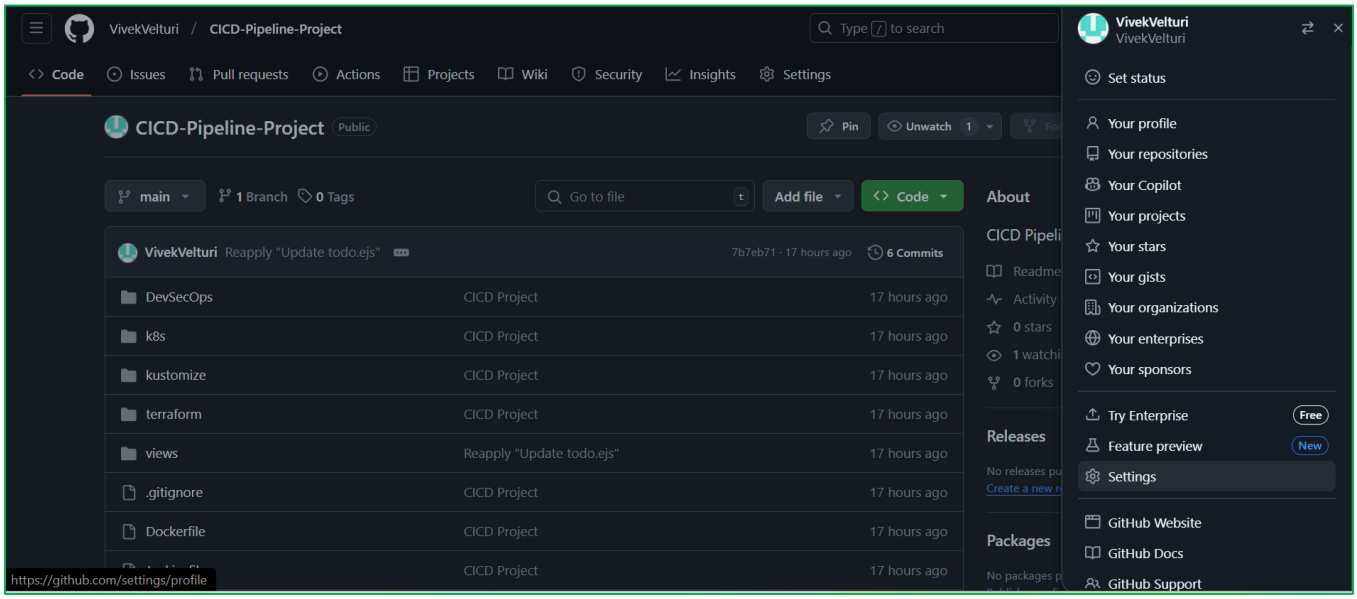
```
ubuntu@ip-172-31-11-98:~/.ssh$ cat id_ed25519
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAAAMwAAAAtzc2gtZW
QyNTUxOQAAACAQpKBQuCWKXAjdZs0CTuEdEDJDK3GdUwY4eyLWYmmtGAAAAKAq8rwhKvK8
IQAAAAtzc2gtZWQyNTUxOQAAACAQpKBQuCWKXAjdZs0CTuEdEDJDK3GdUwY4eyLWYmmtGA
AAAEC9GzBX+ES18kar73VdsRNwarNTKM2dN74HUv96ieTnHhCkoFC4JYpcCN1mzQJO4R0Q
MkMrcZ1TBjh7ItZiaa0YAAAAFnVidW50dUBpcC0xNzItMzEtMTEtOTgBAgMEBQYH
-----END OPENSSH PRIVATE KEY-----
```

```
ubuntu@ip-172-31-11-98:~/.ssh$ cat id_ed25519.pub
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIBCkoFC4JYpcCN1mzQJO4R0QMkMrcZ1TBjh7ItZiaa0Y ubuntu@ip-172-31-11-98
ubuntu@ip-172-31-11-98:~/.ssh$
```

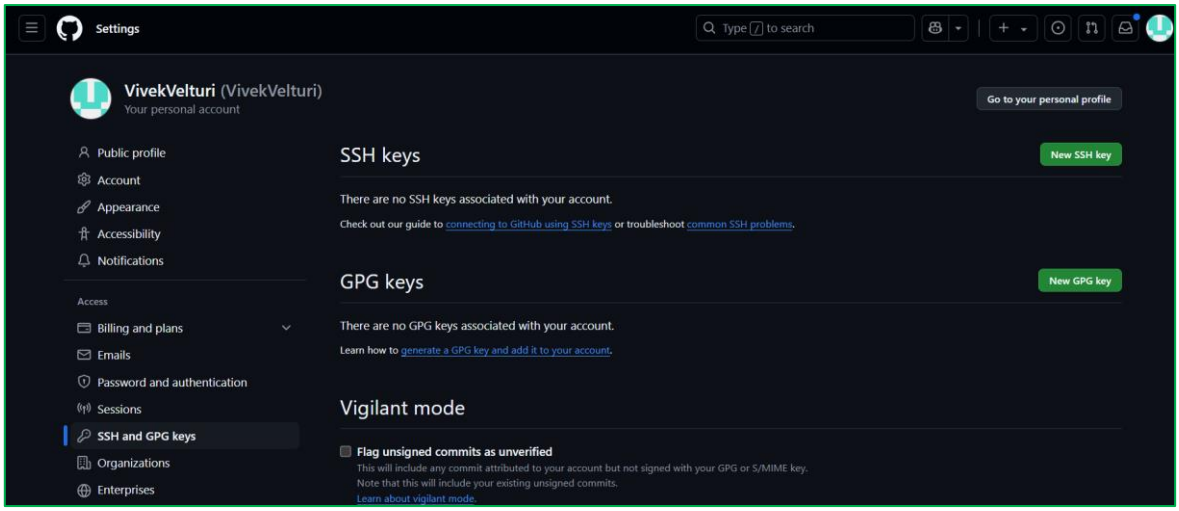
Once you've verified the keys, you can proceed to add the **public key** to GitHub and configure Jenkins to use the **private key** for authentication.

- To add the public key to GitHub, follow these steps:

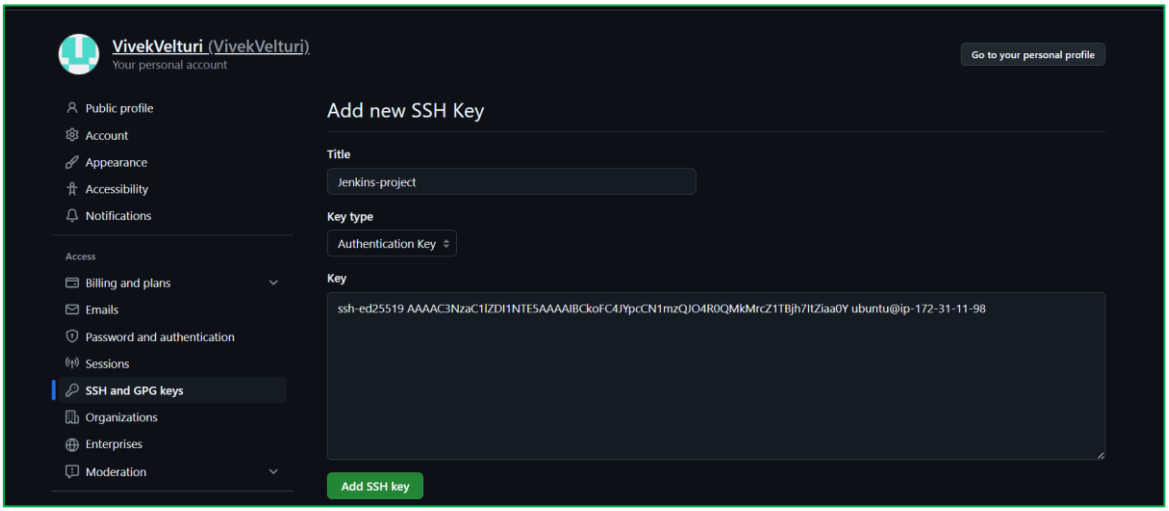
  - Go to your **GitHub account** and click on your profile picture at the top right corner. From the dropdown, select **Settings**.



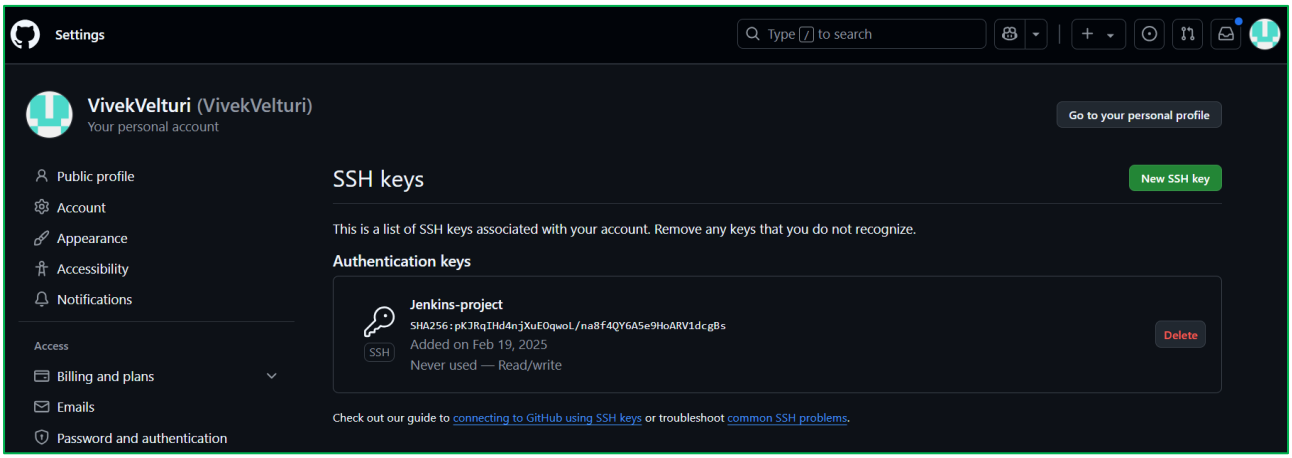
  - In the left sidebar, click on **SSH and GPG keys** under the **Access** section.
  - On the SSH and GPG keys page, click on **New SSH key**.



- In the **Title** field, enter a name for the key (in our case, enter **Jenkins-project)**.
- For **Key type**, choose **Authentication key**.
- Now, paste the public key you obtained earlier by running the `cat id_ed25519.pub` command on your Jenkins server.

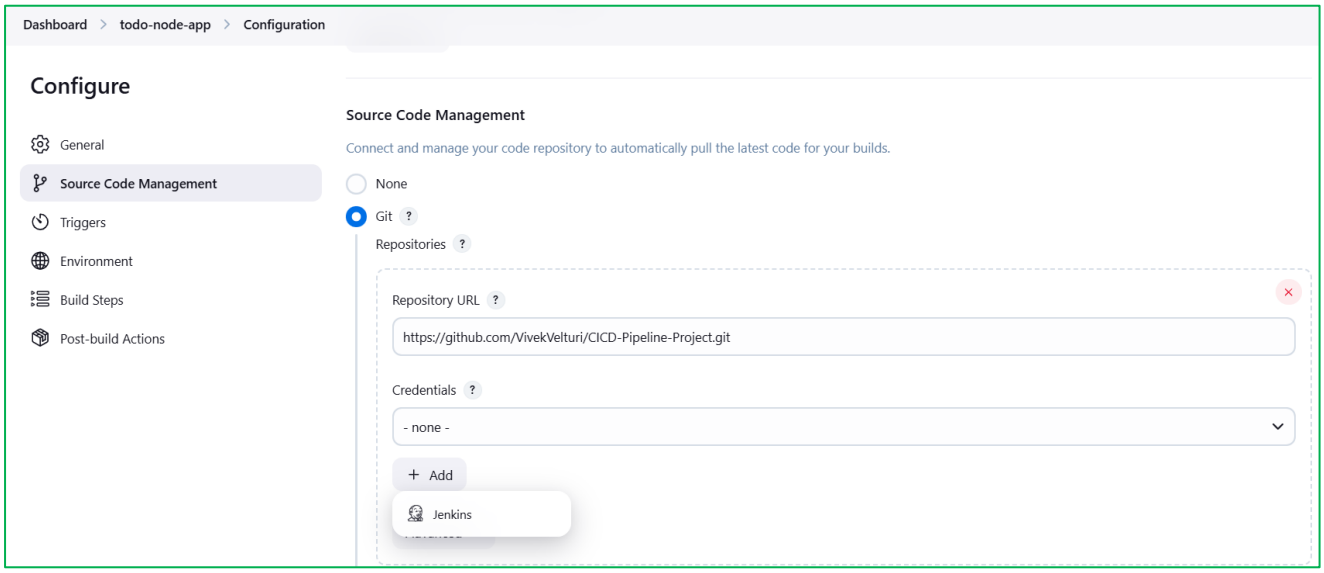


- Once the key is pasted, click on **Add SSH key** to save the key to your GitHub account.

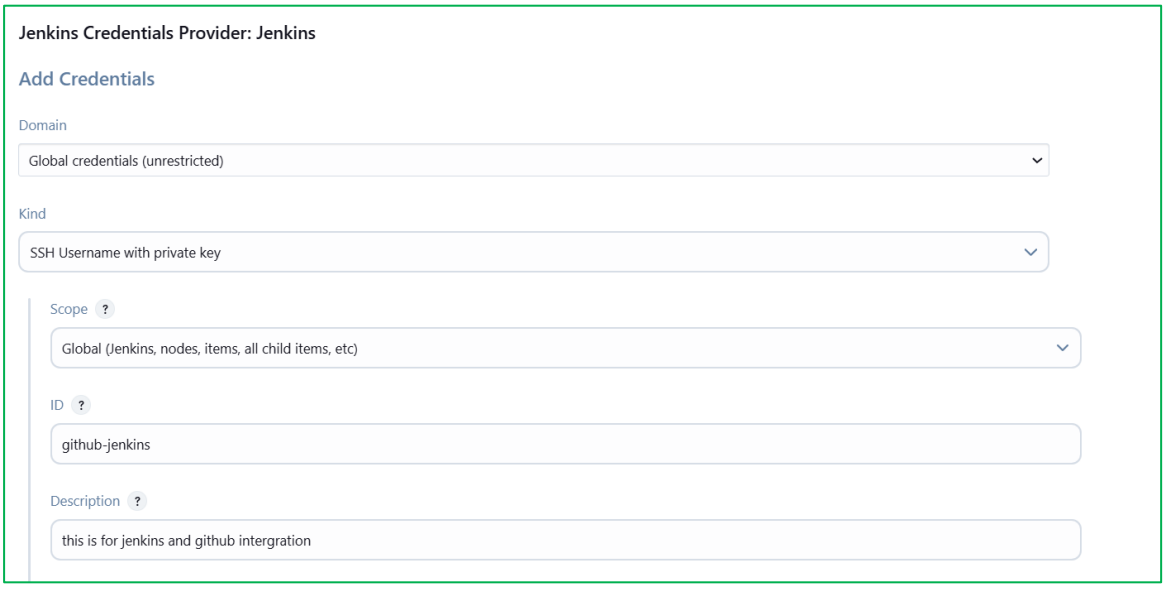


Now, the authentication key has been successfully added to your GitHub account, allowing Jenkins to authenticate and interact with your repository securely.

- To set up the private key in Jenkins for secure GitHub integration, follow these steps:
    - Go to your Jenkins dashboard and open the **todo-node-app** job. Click on **Configure** to open the job configuration page.
    - In the **Source Code Management** section, select **Git** as the version control system. Enter the **repository URL** for your GitHub repository.
    - Under the **Credentials** section, click on **Add** and select **Jenkins**. This will direct you to a new window titled **Jenkins Credential Provider**.

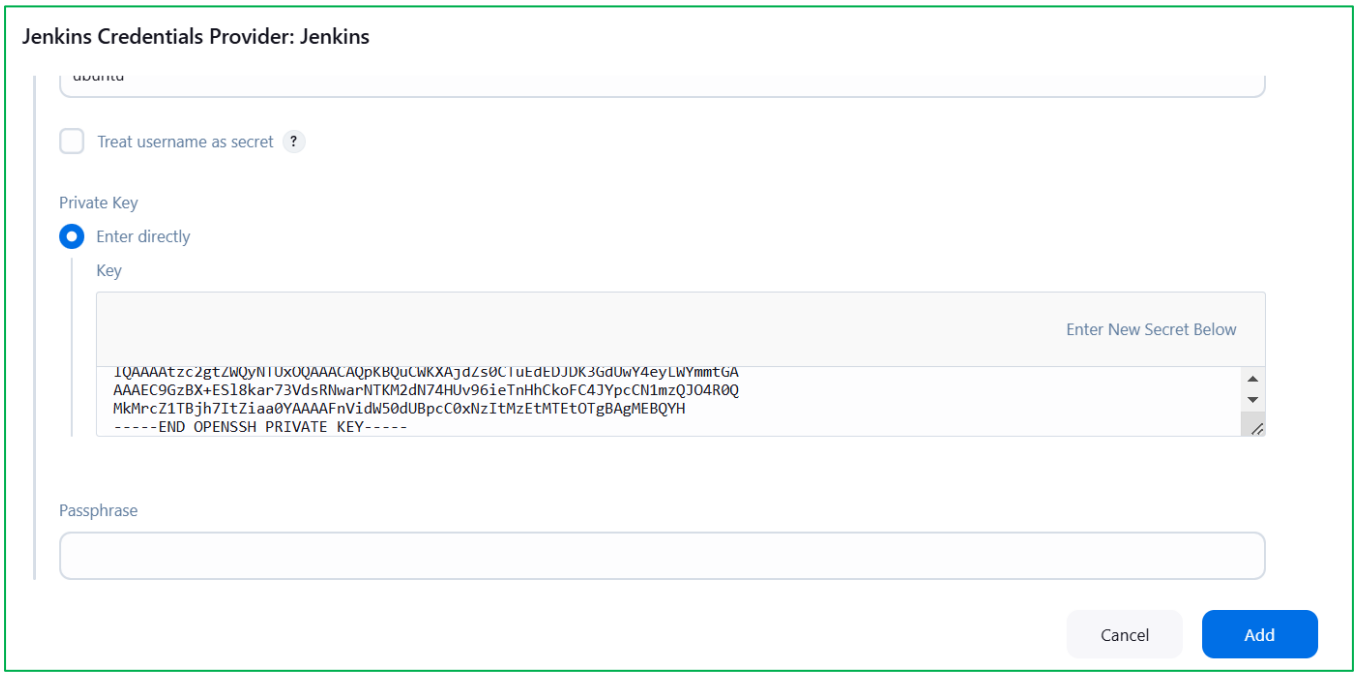


    - In the **Domain** field, select **Global credentials (unrestricted)**. Then, for the **Kind** field, choose **SSH Username with private key**.
    - Set the **Scope** to **Global** (Jenkins, nodes, items, all child items, etc.).
    - For the **ID**, enter **github-jenkins**. In the **Description**, enter **"this is for jenkins and github integration"**.

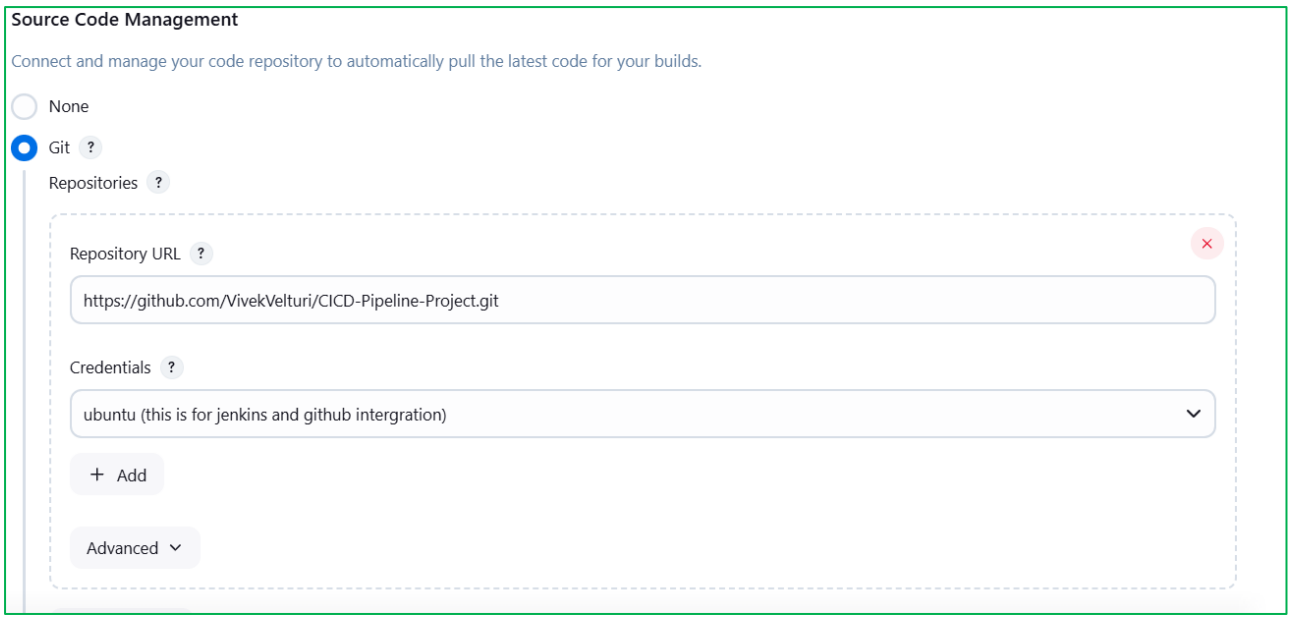


- In the **Username** field, enter **ubuntu** (as it is the username for your instance).
  If you prefer to keep the **username** secure, you can treat it as a **secret** in Jenkins. During the **SSH Username with private key** setup, add the username as a secret, and Jenkins will mask it to prevent it from being exposed in logs or configuration files. This adds an extra layer of security, especially in shared or public environments. However, if security is not a major concern, you can leave it as is.

- For the **Private Key**, select **Enter directly** and paste the private key you obtained earlier by running `cat id_ed25519` on your Jenkins server. Leave the **Passphrase** field empty (since you didn't set one during key generation).
- Click on **Add** at the bottom to save the credentials.

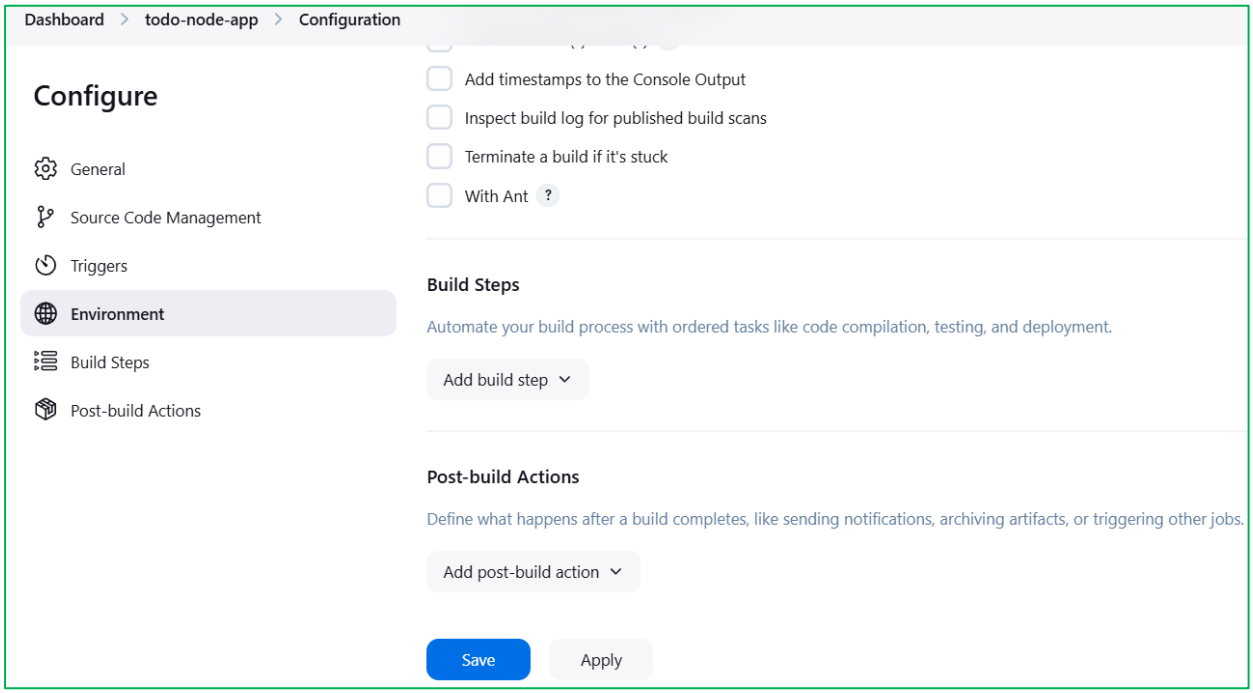


- Now that the credentials have been added, go back to the **Source Code Management** section in the job configuration. Click **Add** under **Credentials**, and you will see the credentials you just created (named **ubuntu** with the description **"This is for Jenkins and GitHub integration"**).



- Once the credentials are selected, go to your GitHub repository and check which branch your code is stored in. Under the **Branches to build** section, specify the branch you want Jenkins to build (in our case, it's likely `*/main`).

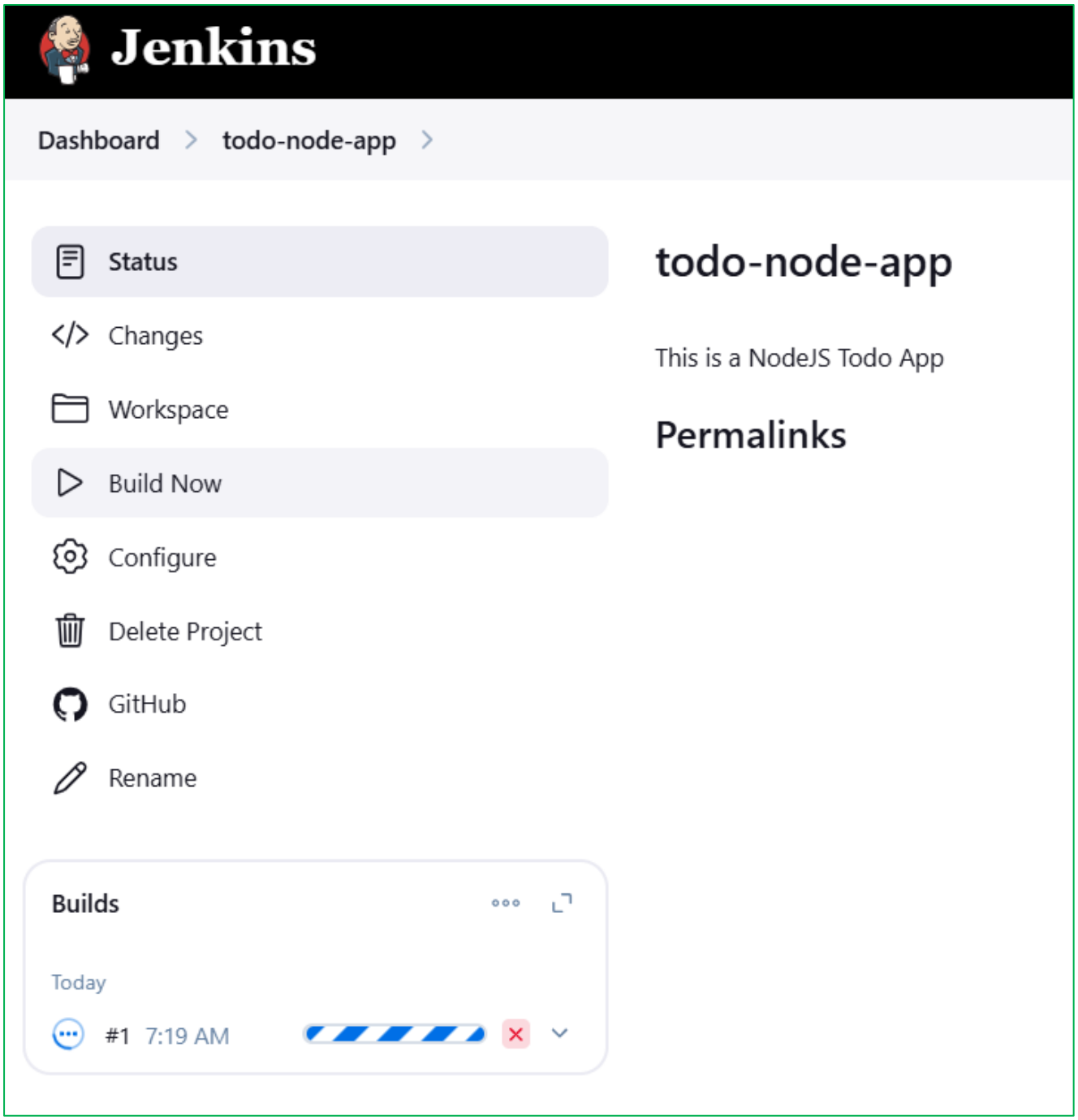
- Leave the other settings as default.
- After all configurations are done, click **Save** at the bottom to save the job configuration.



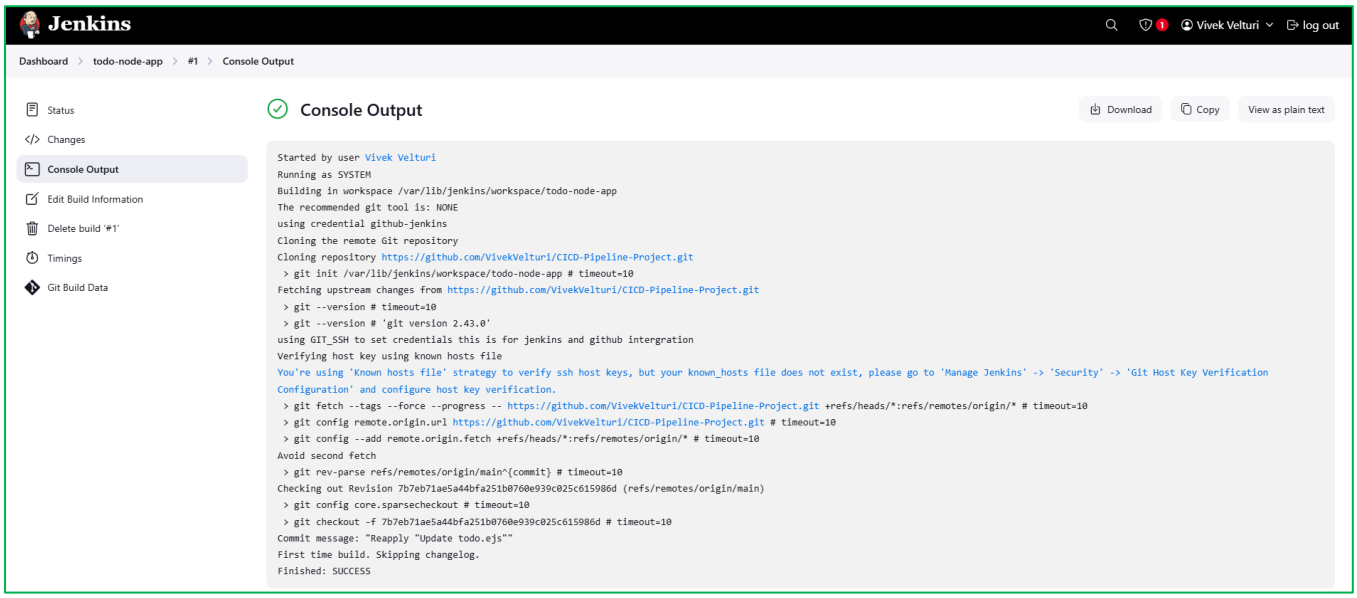
Now Jenkins is set up to securely pull the code from your GitHub repository using the private key you configured. The integration between Jenkins and GitHub is complete, allowing Jenkins to automatically access your repository and build your project.

Executing a Build for a Job in Jenkins

- Once the job configuration is complete, go to your **todo-node-app** job on the Jenkins dashboard. On the left-hand side, you will find the option **Build Now**. Click on **Build Now** to start the build process.
- Jenkins will begin pulling the code from the GitHub repository and trigger the build. You can monitor the build progress from the **Build History** section on the dashboard.



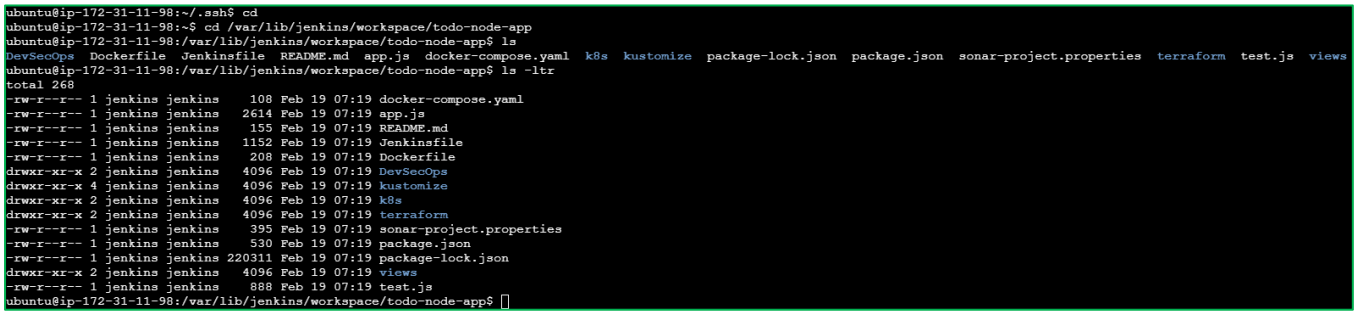
- To view the output of the build that was executed, go to the **todo-node-app** job on the Jenkins dashboard.
  In the **Build History** section on the left side, click on the **build number** (e.g., **#1**, **#2**, etc.) corresponding to the build you just triggered.
  This will take you to a page showing detailed information about the build.
  To view the **build output**, click on **Console Output**.
  This will display all the logs and steps executed during the build process, including repository cloning, build steps, and any success or failure messages.



- After clicking **Build Now** on the **todo-node-app** job, Jenkins started the build process.
  The output indicates that the system is running under the user **Vivek Velturi** and that the build is happening in the workspace directory: **/var/lib/jenkins/workspace/todo-node-app**.
  Jenkins used **Git** to clone the repository from **https://github.com/VivekVelturi/CICD-Pipeline-Project.git** and fetched the latest changes.
  It also confirmed the use of **SSH keys** for authentication to GitHub.
  During the process, there was a warning about the **known_hosts** file not existing, suggesting that the user go to **Manage Jenkins** > **Security** > **Git Host Key Verification Configuration** to resolve the issue.
  After verifying the repository and configuring the git settings, the build process successfully checked out the **main** branch from the GitHub repository.
  Finally, the build finished successfully, as indicated by the message **"First time build. Skipping changelog."** and **"Finished: SUCCESS"**.
  This output confirms that Jenkins successfully connected to the GitHub repository and executed the build process without issues.

- We can also verify the build output directly in our instance by checking the workspace directory mentioned in the Console Output.
  The output specifies that the build is executed in the directory **/var/lib/jenkins/workspace/todo-node-app** on the **jenkins-master** instance.

To inspect this directory, access the instance terminal and navigate to the workspace directory using `cd /var/lib/jenkins/workspace/todo-node-app`.
Once inside, listing the files using `ls` **or** `ls -ltr` will display all the files cloned from the GitHub repository into the workspace directory.



Observing these files confirms that Jenkins successfully pulled the repository contents and placed them in the appropriate build directory.

Deploying a Node.js Application Using the Instance

- In the GitHub repository that has been cloned into our Jenkins workspace, there is a **README.md** file. This file has been included by the developer to outline the necessary steps to run the **Node.js** application. To view its contents, navigate to the workspace directory in the **jenkins-master** instance and use the command **cat README.md**. This will display the instructions provided by the developer, helping us understand the steps required to set up and execute the application properly.

- The first command mentioned in the **README.md** file for setting up the **Node.js** application is `sudo apt install nodejs`. This command installs Node.js on the system, which is required to run the application. Since the application is built using

Node.js, having the correct runtime environment is essential for executing the project successfully.

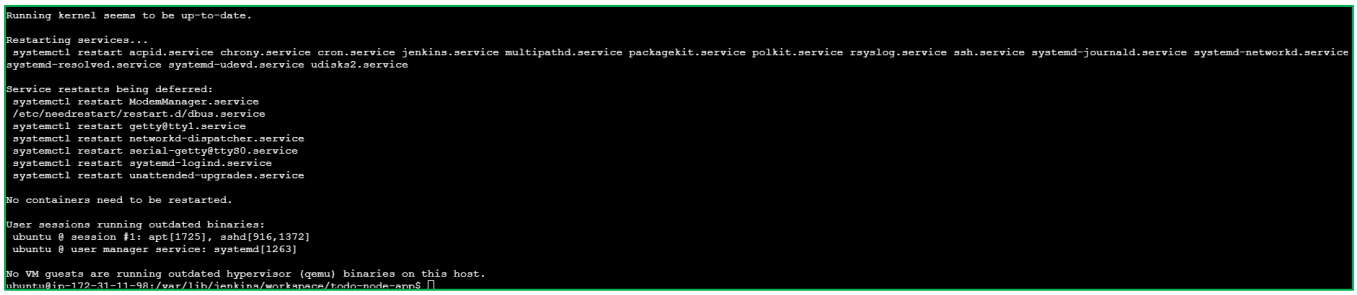
```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ sudo apt install nodejs
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  libcares2 libnode109 node-acorn node-busboy node-cjs-module-lexer node-undici node-xtend nodejs-doc
Suggested packages:
  npm
The following NEW packages will be installed:
  libcares2 libnode109 node-acorn node-busboy node-cjs-module-lexer node-undici node-xtend nodejs nodejs-doc
0 upgraded, 9 newly installed, 0 to remove and 96 not upgraded.
Need to get 16.1 MB of archives.
After this operation, 70.4 MB of additional disk space will be used.
Do you want to continue? [Y/n] 
```

Running `sudo apt install nodejs` prompts **"Do you want to continue? [Y/n]"**. Type **Y** and press **Enter** to proceed with the installation.

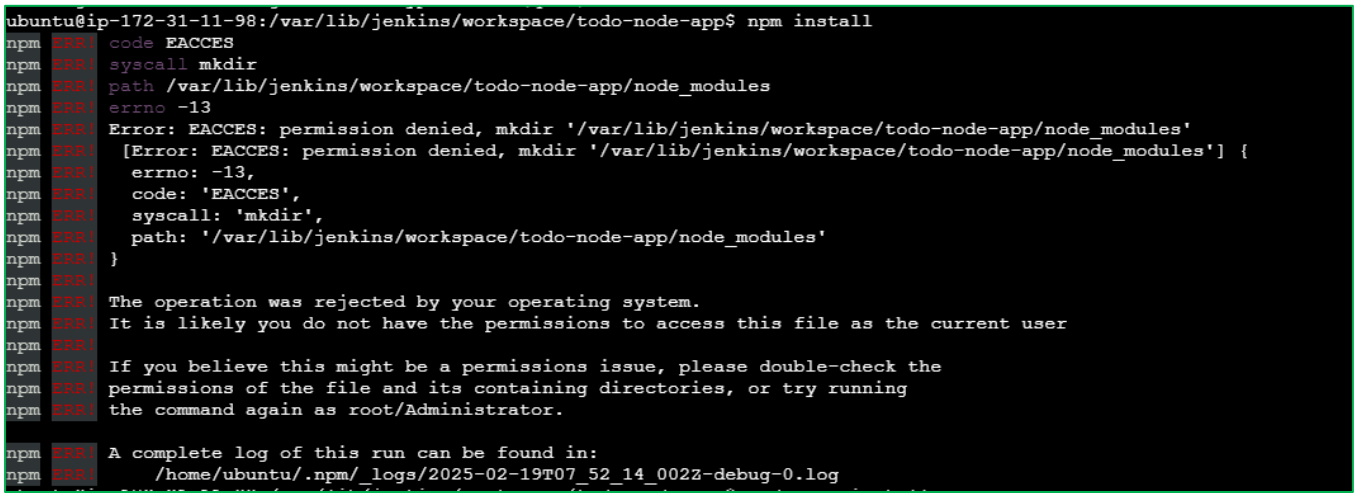
```
Running kernel seems to be up-to-date.

No services need to be restarted.

No containers need to be restarted.

No user sessions are running outdated binaries.

No VM guests are running outdated hypervisor (qemu) binaries on this host.
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ 
```

- The next command in the **README.md** file is `sudo apt install npm`, which installs **Node Package Manager (npm)**. This is required to manage dependencies and install the necessary packages for the **Node.js** application to run.

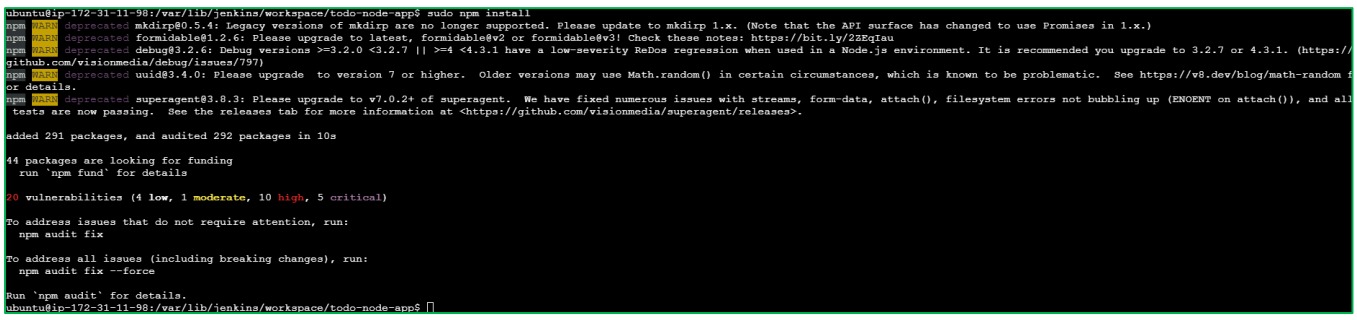
Running `sudo apt install npm` will prompt **"Do you want to continue? [Y/n]"**. Type **Y** and press **Enter** to proceed with the installation.



- The next command in the `README.md` file is `npm install`. This command is used to install all the dependencies listed in the `package.json` file. However, when executing `npm install`, we encounter a **permission denied** error while trying to create the `node_modules` directory inside the Jenkins workspace. This occurs because the current user does not have the necessary write permissions for this directory.



- To resolve this issue, we execute `sudo npm install`. Running the command with `sudo` grants administrative privileges, allowing the necessary directories and files to be created successfully. Once executed, this command installs all required dependencies, enabling the application to run properly.



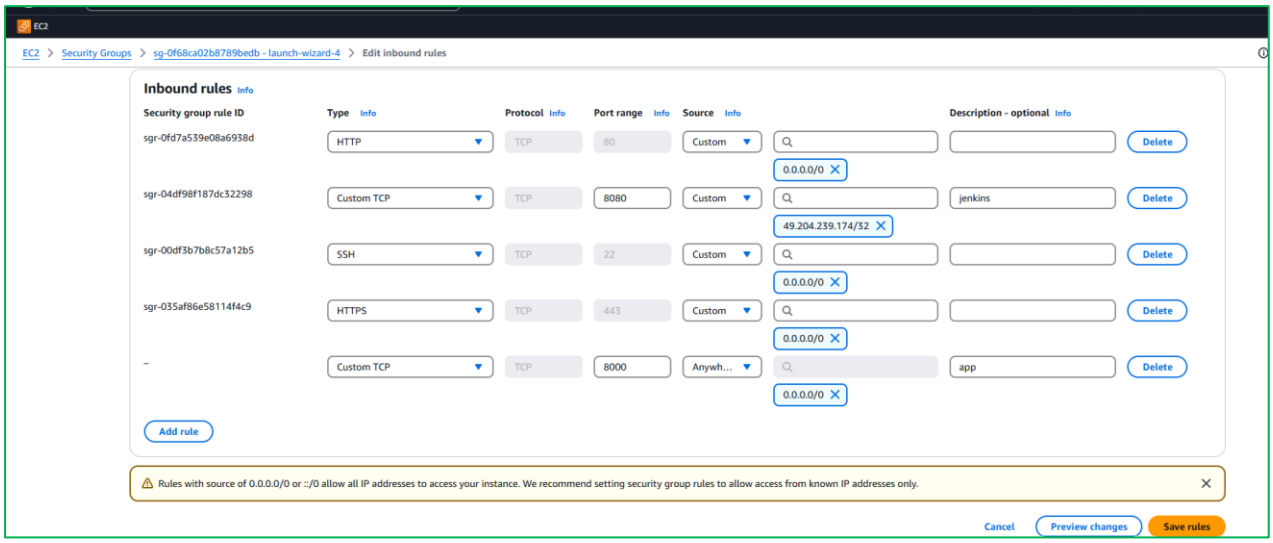
- The next command mentioned in the **README.md** file is `node app.js`. This command starts the **To-Do List application** by executing the `app.js` script present

in the cloned repository. When the command is executed, the output confirms that the application is running and accessible at `http://0.0.0.0:8000`. This indicates that the Node.js application has started successfully and can be accessed via a web browser using the server's public IP address followed by port `8000`.

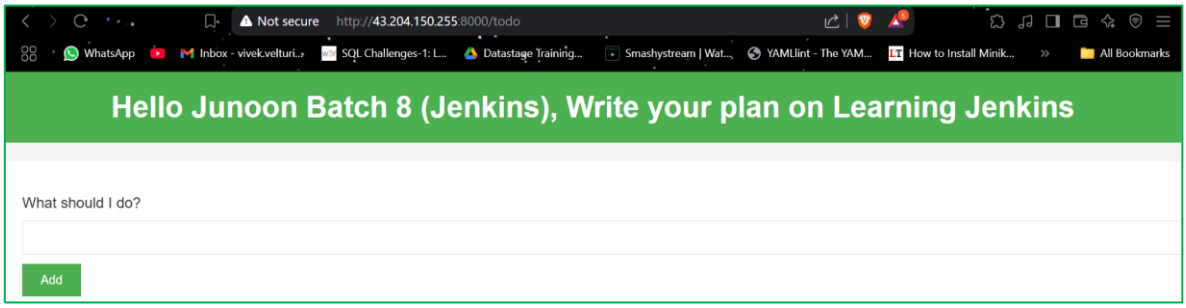
```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ node app.js
Todolist running on http://0.0.0.0:8000
```

- To access the application running on port **8000**, we first need to modify the **security group** settings of our EC2 instance. This requires editing the **inbound rules** to allow traffic through port **8000**.

  - Navigate to the **AWS EC2 Dashboard** and select the instance where Jenkins is running(jenkins-master).
  - Click on the **Security** tab and locate the **Security Groups** section.
  - Open the security group associated with the instance and go to the **Inbound rules** tab.
  - Click on **Edit inbound rules** and then **Add rule**.
  - Set the **Port Range** to **8000** and choose **Anywhere (0.0.0.0/0)** for **IPv4** access.
  - Add a **Description** such as **"app"** for reference.
  - Click **Save rules** to apply the changes.

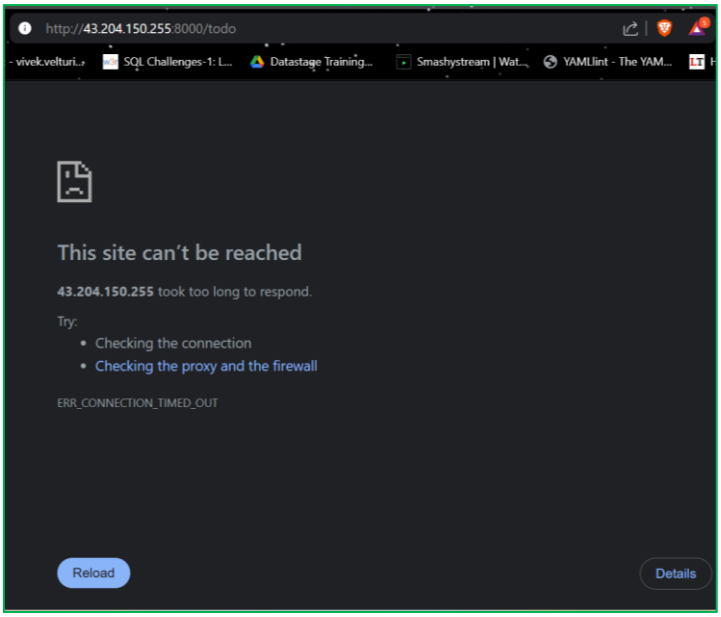


Once these steps are completed, the **To-Do List application** running on port **8000** will be accessible from any device using the **public IP** of the instance followed by `:8000` in the browser.



- Our application is now successfully accessible via the public IP on port **8000**. However, if we press **Ctrl + C** in the instance terminal, it will terminate the process, and the application will stop running. When we check the application again in the browser, it will no longer be accessible.

```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ node app.js
Todolist running on http://0.0.0.0:8000
^C
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$
```



- To prevent this from happening, we need to run the **Node.js** application in the background so that it remains active even after closing the terminal or logging out.

To run the **Node.js** application in the background, use the following command:
`node app.js &`
The `&` at the end ensures that the process runs in the background.
If you want the application to keep running even after you close the terminal, use **nohup command** : `nohup node app.js > output.log 2>&1 &`
The command **`nohup node app.js > output.log 2>&1 &`** is used to run the **Node.js application** in the background while ensuring that it continues running even after the terminal session is closed. The `nohup` (No Hang Up) command prevents the process from being terminated when the user logs out or closes the SSH session. The `node app.js` command starts the application, executing the `app.js` script. The `> output.log` part redirects the standard output (`stdout`) of the application to a file named **`output.log`**, while `2>&1` ensures that any error messages (`stderr`) are also redirected to the same log file, making it easier to debug issues later. Finally, the `&` at the end runs the command in the background, freeing up the terminal for other tasks.

With this setup, the application remains persistently active, and its logs can be reviewed using `cat output.log`. To verify that the process is still running, the `ps aux | grep node` command can be used. If needed, the application can be stopped by identifying its **process ID (PID)** and using the `kill <PID>` command. This method is particularly useful in **cloud environments or production servers** where applications need to remain active independently of the terminal session.

Containerizing the Node.js Application with Docker

- To ensure that the **Node.js application** runs consistently across different operating systems, we can use **Docker** to containerize the application. Docker allows the app to run in an isolated environment, making it OS-independent while ensuring the necessary dependencies are included.

  By creating a **Docker image**, the application can be deployed on **Linux, Windows, and macOS** without worrying about differences in system configurations. The Docker container includes **Node.js, npm, and all dependencies**, ensuring that the application runs the same way on any machine with Docker installed.

- To create a custom **Dockerfile** tailored to our requirements, we first need to remove the existing **Dockerfile** provided by the developer. Since the repository already contains a pre-existing **Dockerfile**,

```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ ls -ltr
total 280
-rw-r--r--   1 jenkins jenkins    108 Feb 19 07:19 docker-compose.yaml
-rw-r--r--   1 jenkins jenkins   2614 Feb 19 07:19 app.js
-rw-r--r--   1 jenkins jenkins    155 Feb 19 07:19 README.md
-rw-r--r--   1 jenkins jenkins   1152 Feb 19 07:19 Jenkinsfile
-rw-r--r--   1 jenkins jenkins    208 Feb 19 07:19 Dockerfile
drwxr-xr-x   2 jenkins jenkins   4096 Feb 19 07:19 DevSecOps
drwxr-xr-x   4 jenkins jenkins   4096 Feb 19 07:19 kustomize
drwxr-xr-x   2 jenkins jenkins   4096 Feb 19 07:19 k8s
drwxr-xr-x   2 jenkins jenkins   4096 Feb 19 07:19 terraform
-rw-r--r--   1 jenkins jenkins    395 Feb 19 07:19 sonar-project.properties
-rw-r--r--   1 jenkins jenkins    530 Feb 19 07:19 package.json
drwxr-xr-x   2 jenkins jenkins   4096 Feb 19 07:19 views
-rw-r--r--   1 jenkins jenkins    888 Feb 19 07:19 test.js
-rw-r--r--   1 jenkins jenkins 220311 Feb 19 07:52 package-lock.json
drwxr-xr-x 251 root    root     12288 Feb 19 07:52 node_modules
```

we delete it from our **workspace directory** using the command `sudo rm Dockerfile`.

```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ sudo rm Dockerfile
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ ls -ltr
total 276
-rw-r--r--   1 jenkins jenkins    108 Feb 19 07:19 docker-compose.yaml
-rw-r--r--   1 jenkins jenkins   2614 Feb 19 07:19 app.js
-rw-r--r--   1 jenkins jenkins    155 Feb 19 07:19 README.md
-rw-r--r--   1 jenkins jenkins   1152 Feb 19 07:19 Jenkinsfile
drwxr-xr-x   2 jenkins jenkins   4096 Feb 19 07:19 DevSecOps
drwxr-xr-x   4 jenkins jenkins   4096 Feb 19 07:19 kustomize
drwxr-xr-x   2 jenkins jenkins   4096 Feb 19 07:19 k8s
drwxr-xr-x   2 jenkins jenkins   4096 Feb 19 07:19 terraform
-rw-r--r--   1 jenkins jenkins    395 Feb 19 07:19 sonar-project.properties
-rw-r--r--   1 jenkins jenkins    530 Feb 19 07:19 package.json
drwxr-xr-x   2 jenkins jenkins   4096 Feb 19 07:19 views
-rw-r--r--   1 jenkins jenkins    888 Feb 19 07:19 test.js
-rw-r--r--   1 jenkins jenkins 220311 Feb 19 07:52 package-lock.json
drwxr-xr-x 251 root    root     12288 Feb 19 07:52 node_modules
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$
```

Once the existing **Dockerfile** is deleted, we can proceed with creating a new one from scratch, specifying our own configurations to containerize the **Node.js application** according to our needs.

- To use Docker for containerizing our **Node.js application**, we first need to install Docker on our instance.
  This is done by executing the command:
  `sudo apt install docker.io`
  which installs **Docker Engine** from the default Ubuntu package repository.

```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ sudo apt install docker.io
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  bridge-utils containerd dns-root-data dnsmasq-base pigz runc ubuntu-fan
Suggested packages:
  ifupdown aufs-tools cgroupfs-mount | cgroup-lite debootstrap docker-buildx docker-compose-v2 docker-doc rinse zfs-fuse | zfsutils
The following NEW packages will be installed:
  bridge-utils containerd dns-root-data dnsmasq-base docker.io pigz runc ubuntu-fan
0 upgraded, 8 newly installed, 0 to remove and 92 not upgraded.
Need to get 78.6 MB of archives.
After this operation, 302 MB of additional disk space will be used.
Do you want to continue? [Y/n]
```

During the installation, the system will prompt **"Do you want to continue? [Y/n]"**, where we need to type **Y** and press **Enter** to proceed. Once the installation is complete, Docker will be ready to use for building and running containerized applications.

```
Setting up containerd (1.7.24-0ubuntu1~24.04.1) ...
Created symlink /etc/systemd/system/multi-user.target.wants/containerd.service → /usr/lib/systemd/system/containerd.service.
Setting up ubuntu-fan (0.12.16) ...
Created symlink /etc/systemd/system/multi-user.target.wants/ubuntu-fan.service → /usr/lib/systemd/system/ubuntu-fan.service.
Setting up docker.io (26.1.3-0ubuntu1~24.04.1) ...
info: Selecting GID from range 100 to 999 ...
info: Adding group `docker' (GID 114) ...
Created symlink /etc/systemd/system/multi-user.target.wants/docker.service → /usr/lib/systemd/system/docker.service.
Created symlink /etc/systemd/system/sockets.target.wants/docker.socket → /usr/lib/systemd/system/docker.socket.
Processing triggers for dbus (1.14.10-4ubuntu4.1) ...
Processing triggers for man-db (2.12.0-4build2) ...
Scanning processes...
Scanning linux images...

Running kernel seems to be up-to-date.

No services need to be restarted.

No containers need to be restarted.

No user sessions are running outdated binaries.

No VM guests are running outdated hypervisor (qemu) binaries on this host.
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$
```

- Let's create a **Dockerfile** to containerize our Node.js application. The Dockerfile will define the necessary environment and instructions to build and run our application inside a container. To do this, execute the following command:
  `sudo vim Dockerfile`

```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ sudo vim Dockerfile
```

Once inside the editor, paste the following code:

```
FROM node:12.2.0-alpine
WORKDIR app
COPY ..
RUN npm install
EXPOSE 8000
CMD ["node", "app.js"]
```

Explanation of Each Line:

1. **FROM node:12.2.0-alpine**
   This line specifies the base image for our container. The **Node.js 12.2.0 Alpine** image is chosen because it is a lightweight and optimized version of Linux that includes Node.js. Using an Alpine-based image reduces the container size, making it more efficient and secure.
2. **WORKDIR app**
   The `WORKDIR` command sets the working directory inside the container to `app`. All subsequent commands in the Dockerfile will be executed within this directory. This helps maintain a clean and organized structure inside the container.
3. **COPY ..**
   This command copies all the files from the current directory on the host system to the **app** directory inside the container. This ensures that all necessary source code, configuration files, and dependencies are available inside the container.
4. **RUN npm install**
   The `RUN npm install` command installs all dependencies listed in the **package.json** file. This ensures that the Node.js application has all required libraries and modules installed before running.
5. **EXPOSE 8000**
   This line informs Docker that the containerized application will listen on port **8000**. Although this does not publish the port automatically, it serves as metadata, indicating that the application is designed to use this port.
6. **CMD ["node", "app.js"]**
   The `CMD` command specifies the default command that runs when the container starts. In this case, it executes `node app.js`, which launches the Node.js application. The **CMD** instruction ensures that the application starts automatically when the container runs.

This Dockerfile ensures that our Node.js application is properly containerized and ready for deployment across different environments. By using this setup, we ensure consistency and portability, making it easier to deploy the application on various systems.

After pasting the Dockerfile content in the **vim** editor, press `Esc` to switch to command mode, then type `:wq!` and press `Enter`. This command saves the file and exits the editor, ensuring that all changes are written and finalized before returning to the terminal.

```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ cat Dockerfile
FROM node:12.2.0-alpine
WORKDIR app
COPY ..
RUN npm install
EXPOSE 8000
CMD ["node","app.js"]
```

- Now, we will build the Docker image using the Dockerfile we have created. To do this, we executed the following command inside the **workspace directory**:
  `docker build -t todo-node-app`

```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ docker build . -t todo-node-app
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Post "http://%2Fvar%2Frun%2Fdocker.sock/v1.45/build?buildargs=%7B%7D&cachefrom=%5B%5D&cgroupparent=&cpuperiod=0&cpuquota=0&cpusetcpus=&cpusetmems=&cpushares=0&dockerfile=Dockerfile&labels=%7B%7D&memory=0&memswap=0&networkmode=default&rm=1&shmsize=0&t=todo-node-app&target=&ulimits=%5B%5D&version=1": dial unix /var/run/docker.sock: connect: permission denied
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$
```

However, we encountered a **permission denied** error while trying to connect to the Docker daemon socket. The error message indicated that the current user did not have the necessary permissions to interact with Docker.

- To resolve the **permission denied** error, we need to grant our user access to the Docker group by executing `sudo usermod -aG docker $USER`
  This command adds the current user to the **docker** group, allowing them to run Docker commands without requiring `sudo`.

```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ sudo usermod -aG docker $USER
```

However, changes to group memberships do not take effect immediately. To apply these changes, a system reboot is necessary because group memberships are assigned at login. Without a reboot (or logging out and back in), the user will still not have the required permissions. To ensure the new permissions take effect, we execute `sudo reboot`, which reboots the system, ensuring that the user is correctly recognized as a member of the **docker** group. Once the system restarts, we can proceed with building the Docker image without encountering permission issues.

```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ sudo reboot
```

- After the reboot, we first navigate back to our **workspace directory** using the command `cd /var/lib/jenkins/workspace/todo-node-app`.
  This ensures that we are in the correct location where our **Dockerfile** and project files are stored.
  Once inside the workspace directory, we execute `docker ps` to verify if the permission changes have taken effect.
  If the permissions are applied correctly, `docker ps` should run successfully and display the container list. Since we haven't created any containers yet, the output will be blank, indicating that no containers are currently running. However, the successful execution of the command confirms that our user now has the necessary permissions to use Docker.

```
ubuntu@ip-172-31-11-98:~$ cd /var/lib/jenkins/workspace/todo-node-app
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$
```

- Now, we execute the **Docker build** command inside our **workspace directory** using `docker build -t todo-node-app`.
  This command instructs Docker to create an image named **todo-node-app** using the **Dockerfile** present in the directory. Since we have already resolved the permission issue, the command should now execute successfully, pulling the necessary base image, installing dependencies, and setting up the container environment. Once the build process is complete, our Docker image will be ready for use.

```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ docker build . -t todo-node-app
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  25.32MB
Error response from daemon: dockerfile parse error on line 3: COPY requires at least two arguments, but only one was provided. Destination could not be determined
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ ^C
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ sudo vim Dockerfile
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ docker build . -t todo-node-app
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  25.32MB
Step 1/6 : FROM node:12.2.0-alpine
12.2.0-alpine: Pulling from library/node
e7c96db7181b: Pull complete
a9b145f64bbe: Pull complete
3bcb5e14be53: Pull complete
Digest: sha256:2ab3d9a1bac67c9b4202b774664adaa94d2f1e426d8d28e07bf8979df61c8694
Status: Downloaded newer image for node:12.2.0-alpine
 ---> f391dabf9dce
Step 2/6 : WORKDIR app
 ---> Running in db15a1df8040
 ---> Removed intermediate container db15a1df8040
 ---> d03b24db0dea
Step 3/6 : COPY . .
 ---> 49a0c385a903
Step 4/6 : RUN npm install
 ---> Running in 82e13725b3ad
npm WARN read-shrinkwrap This version of npm is compatible with lockfileVersion@1, but package-lock.json was generated for lockfileVersion@2. I'll try to do my best with it!

> ejs@2.7.4 postinstall /app/node_modules/ejs
> node ./postinstall.js
```

```
Thank you for installing EJS: built with the Jake JavaScript build tool (https://jakejs.com/)

npm WARN my-todolist@0.1.0 No repository field.
npm WARN my-todolist@0.1.0 No license field.

updated 291 packages and audited 291 packages in 10.179s
found 28 vulnerabilities (6 low, 3 moderate, 16 high, 3 critical)
  run `npm audit fix` to fix them, or `npm audit` for details
 ---> Removed intermediate container 82e13725b3ad
 ---> c4339968d4e3
Step 5/6 : EXPOSE 8000
 ---> Running in 35bda445f07e
 ---> Removed intermediate container 35bda445f07e
 ---> 4eafdc75228e
Step 6/6 : CMD ["node","app.js"]
 ---> Running in 25808fc2564d
 ---> Removed intermediate container 25808fc2564d
 ---> ebc9b2264759
Successfully built ebc9b2264759
Successfully tagged todo-node-app:latest
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$
```

- We can check if the Docker image has been successfully built by executing the command : `docker image ls`.
  This command lists all available Docker images on the system, including their **repository name, tag, image ID, creation time, and size**. If the build was successful, we should see **todo-node-app** listed among the images, confirming that our Docker image has been created and is ready to be used.

```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ docker image ls
REPOSITORY      TAG             IMAGE ID       CREATED          SIZE
todo-node-app   latest          ebc9b2264759   13 minutes ago   124MB
node            12.2.0-alpine   f391dabf9dce   5 years ago      77.7MB
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$
```

- Now, we build and run a Docker container using the following command:
  `docker run -d --name node-todo-app -p 8000:8000 todo-node-app:latest`

  Explanation of the Command:
  - **`docker run`**: This command is used to create and start a new container from an existing Docker image.
  - **`-d`**: Runs the container in detached mode, meaning it runs in the background without blocking the terminal.

- **`--name node-todo-app`**: Assigns the container a custom name (`node-todo-app`) for easy reference instead of using a randomly generated name.
- **`-p 8000:8000`**: Maps port `8000` of the container to port `8000` on the host machine, allowing access to the application from the outside.
- **`todo-node-app:latest`**: Specifies the Docker image (`todo-node-app`) and the tag (`latest`) to be used for creating the container.

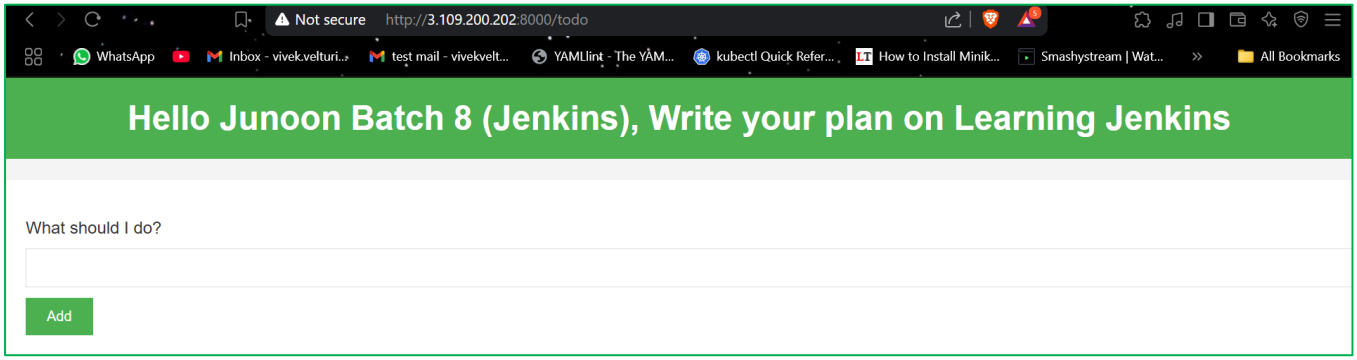
```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ docker run -d --name node-todo-app -p 8000:8000 todo-node-app:latest
c3e5ed9e0a55595bba5dcff572a80c3cc9d213e1d7dcf4614a9823551f6cb06a
```

After executing this command, the container starts running in the background, and our Node.js application inside the container will be accessible on port `8000` of the host machine.

- You can verify that the container was successfully created and is running by executing the command: `docker ps`
  This command lists all active containers, displaying important details such as **Container ID, Name, Image, Status, Ports**, and more. Since we just created and started our container, we should see `node-todo-app` in the list.

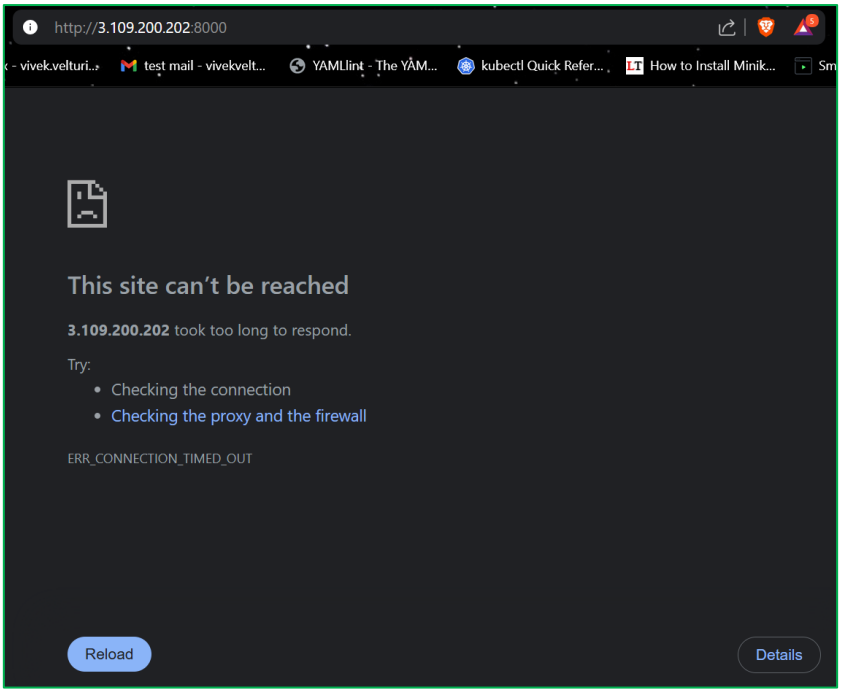
```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ docker ps
CONTAINER ID   IMAGE                 COMMAND           CREATED          STATUS          PORTS                                       NAMES
c3e5ed9e0a55   todo-node-app:latest   "node app.js"   5 seconds ago   Up 4 seconds   0.0.0.0:8000->8000/tcp, :::8000->8000/tcp   node-todo-app
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$
```

- Now, if we access port 8000 using the public IP of our instance in a web browser, we should be able to see our Node.js application running. The application should be accessible at: `http://<PUBLIC_IP of our instance>:8000`
  We have already given permission for port 8000 earlier, so the application should be accessible without additional configuration.



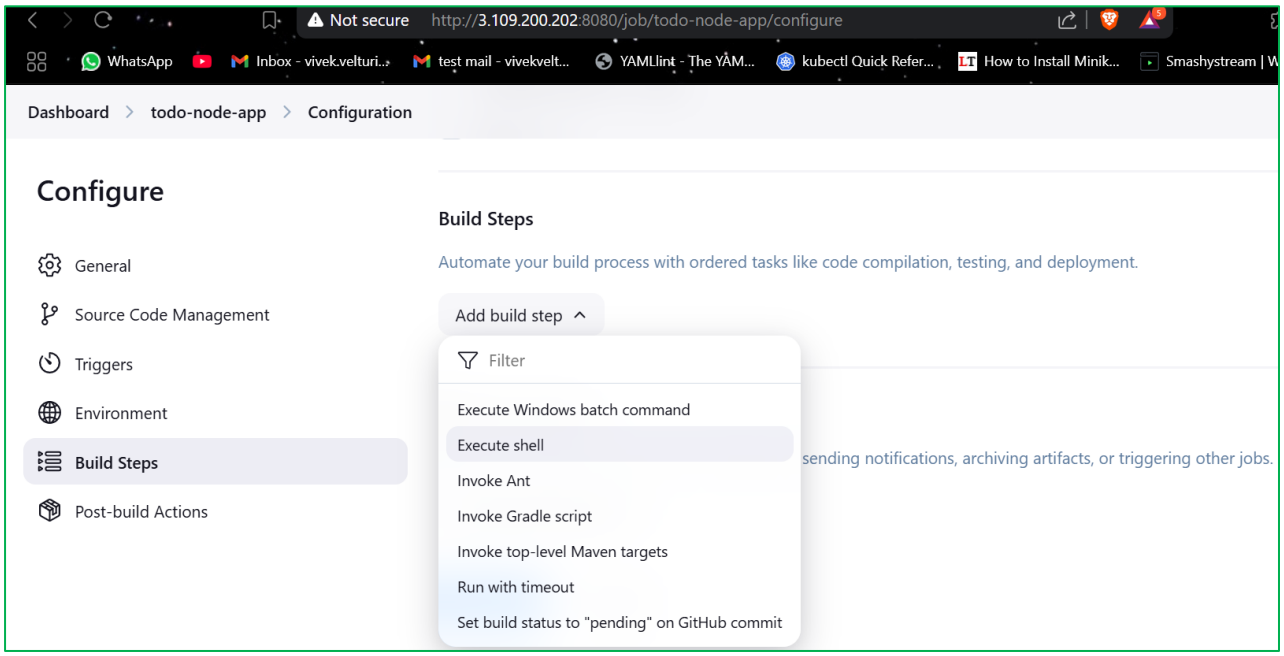
- We have the container ID **c3e5ed9e0a55** from the `docker ps` command. Now, if we kill the container using the command:
  `docker kill c3e5ed9e0a55`
  the container should be stopped, and the application webpage should no longer be accessible.

```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ docker kill c3e5ed9e0a55
c3e5ed9e0a55
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$
```



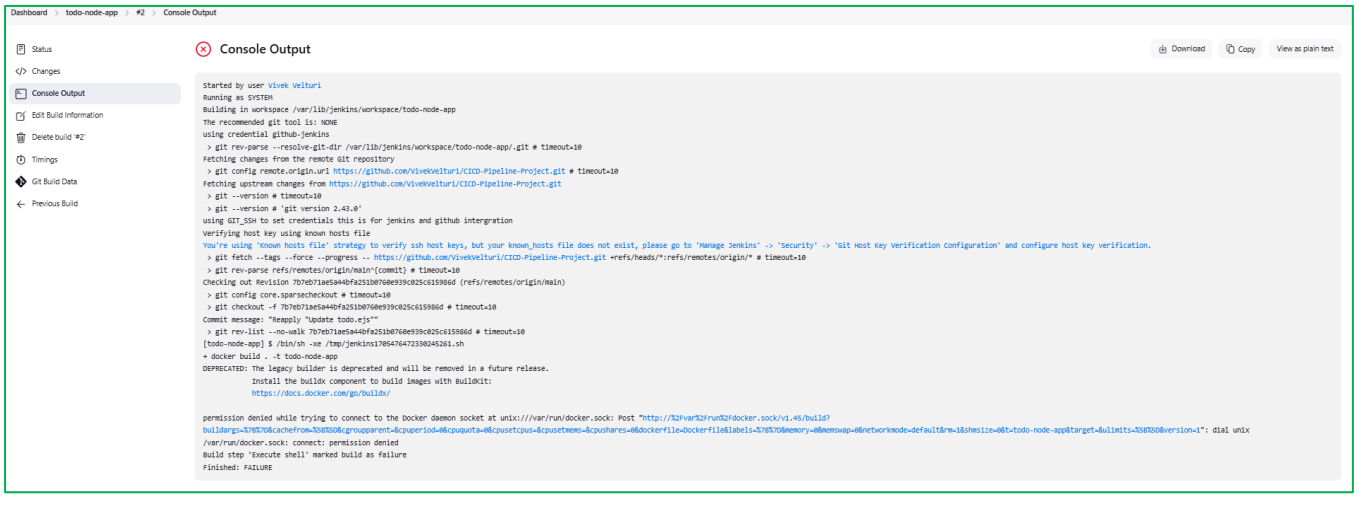
Automating Docker Image Build and Deployment Using Jenkins

- We have now successfully built a Docker image and deployed a container to run our application manually. Next, we will automate this process using Jenkins.
- We can achieve this by accessing Jenkins and navigating to the job **"todo-node-app"**, which we configured earlier. Now, we will modify the build steps to automate the process.
- In the **"todo-node-app"** job, we can automate the Docker image and container creation by adding the same commands we used earlier in the build steps. To do this, navigate to the job configuration, click on **"Add build step"**, select **"Execute shell"**, and enter the commands for building the Docker image and running the container.



- In the **Execute Shell** section of the build step, enter the commands to build the Docker image with the name **todo-node-app** and run a container named **node-todo-app**, mapping port 8000 of the container to port 8000 on the host.
  The commands to be entered are
  `docker build -t todo-node-app`
  followed by
  `docker run -d --name node-todo-app -p 8000:8000 todo-node-app:latest`
  After entering the commands, click **"Save"** to apply the changes.

- Now that we have configured the build steps and saved the changes, click on the **"Build Now"** option for the **"todo-node-app"** job to trigger the automated build process.
- You can check the **Console Output** of the build to monitor the execution process

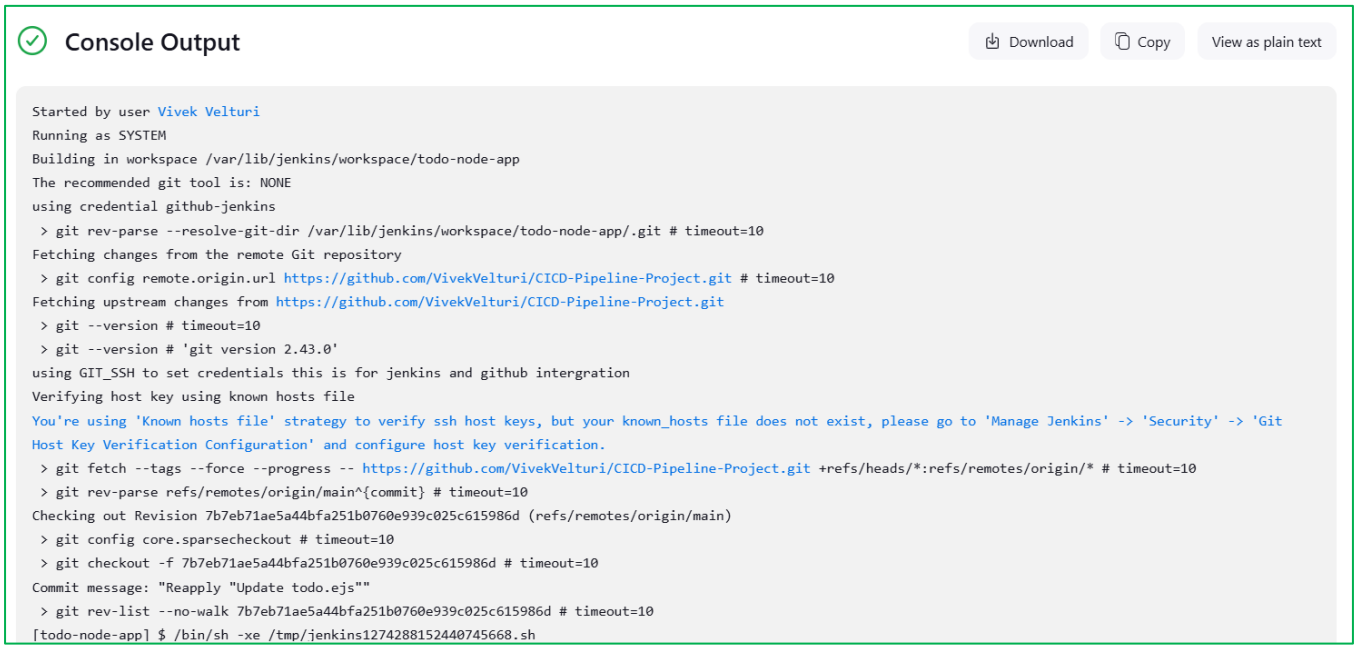


- The error message **"permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock"** indicates that the Jenkins user does not have permission to access Docker.

- To fix this, you need to ensure the Jenkins user is added to the **docker** group and then restart the Jenkins service. Run the following commands on your instance:
  `sudo usermod -aG docker jenkins`
  `sudo systemctl restart Jenkins`

```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ sudo usermod -aG docker jenkins
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ sudo systemctl restart jenkins
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$
```

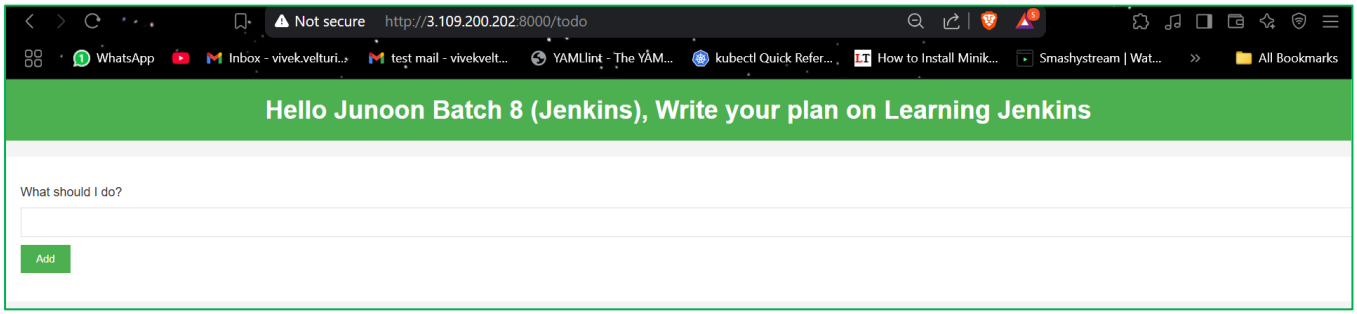
- Now, initiate the build again by going to **Jenkins Dashboard** > **todo-node-app** > Click on **Build Now**.
  Once the build starts, you can monitor its progress by clicking on the running build and selecting **Console Output** to verify if the issue is resolved.



Console Output

```
  run `npm audit fix` to fix them, or `npm audit` for details
 ---> Removed intermediate container 173a296d7e5e
 ---> ae38fb05e2c1
Step 5/7 : RUN npm run test
 ---> Running in 1a8890f04227

> my-todolist@0.1.0 test /node
> mocha --recursive --exit



  Simple Calculations
This part executes once before all tests
    Test1
executes before every test
      ✓ Is returning 5 when adding 2 + 3
executes before every test
      ✓ Is returning 6 when multiplying 2 * 3
    Test2
executes before every test
      ✓ Is returning 4 when adding 2 + 3
executes before every test
      ✓ Is returning 8 when multiplying 2 * 4
This part executes once after all tests


  4 passing (11ms)

 ---> Removed intermediate container 1a8890f04227
 ---> 2d54edf4c1a2
Step 6/7 : EXPOSE 8000
 ---> Running in 1f81fbf70068
 ---> Removed intermediate container 1f81fbf70068
 ---> d532751c143d
Step 7/7 : CMD ["node","app.js"]
 ---> Running in 71d2a94e8796
 ---> Removed intermediate container 71d2a94e8796
 ---> 0a9c701cd8c9
Successfully built 0a9c701cd8c9
Successfully tagged todo-node-app:latest
+ docker run -d --name node-app-container -p 8000:8000 todo-node-app
f2e561c7b67c6f07a4357f004cf0db07cfff765ebf401d584dd2d8d2f6c0a116
Finished: SUCCESS
```

- The Jenkins pipeline successfully built, tested, and deployed the `todo-node-app` using Docker. It first ran Mocha tests, ensuring all test cases passed. Then, it built a Docker image, exposing port **8000** and setting `app.js` as the entry point. Finally, the container was launched in detached mode with proper port mapping, confirming a **successful deployment**. The app can now be accessed via `http://<public IP of the instance>:8000`.



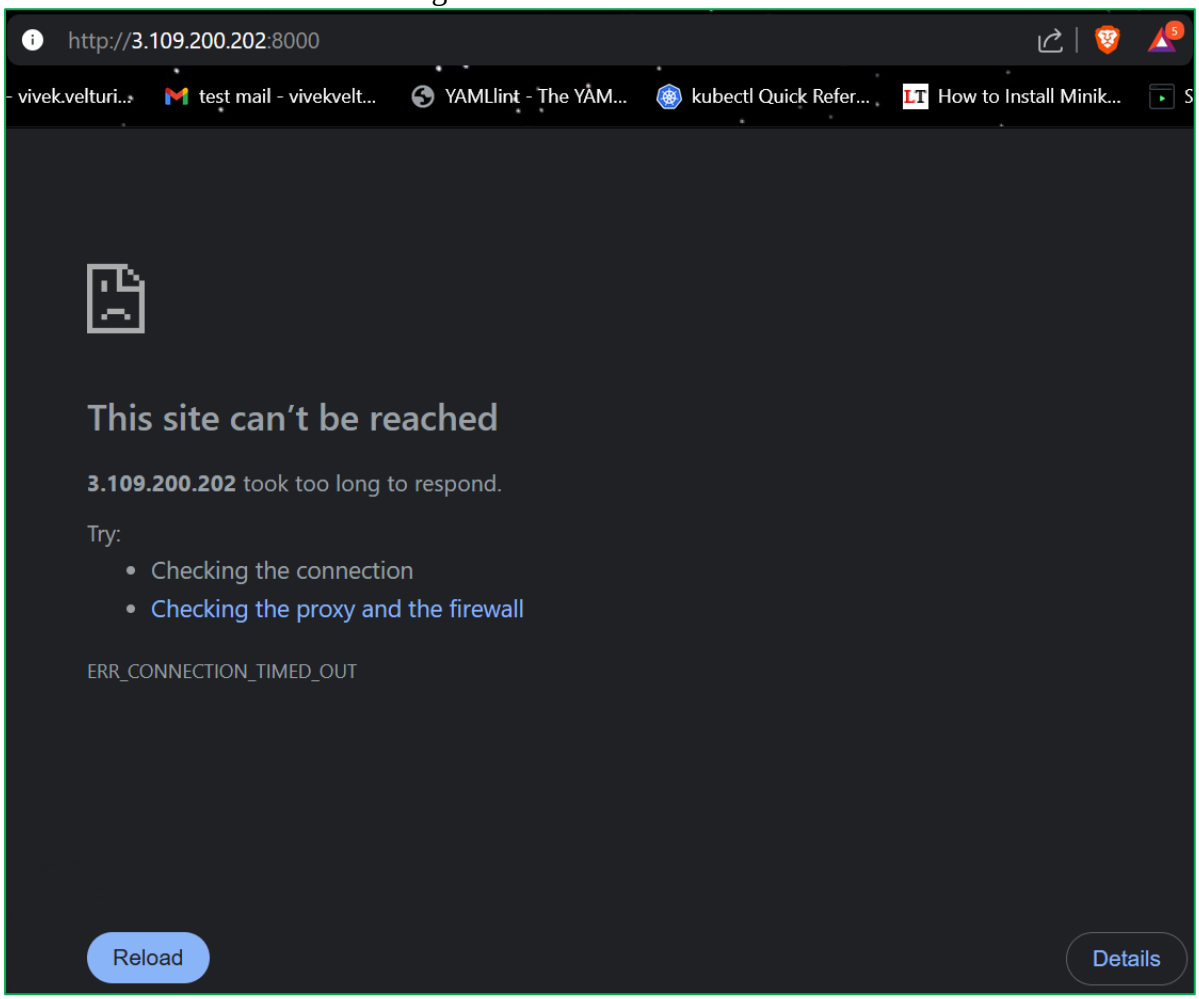
- After triggering the build in Jenkins, we can confirm that the Docker container has been successfully created by running the `docker ps` command. The container created by our Jenkins build will be listed, indicating that it is active and running as expected.

```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ docker ps
CONTAINER ID   IMAGE           COMMAND          CREATED          STATUS          PORTS                                       NAMES
f2e561c7b67c   todo-node-app   "node app.js"    12 minutes ago   Up 12 minutes   0.0.0.0:8000->8000/tcp, :::8000->8000/tcp   node-app-container
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$
```

- The **Container Name** and **Docker Image ID** match with the commands provided in the **Execute Shell** script and the details displayed in the **Console Output** of Jenkins. This confirms that the container was created as per the build trigger script and is running correctly on the instance.
- To stop the Docker container created during the Jenkins build process, we use the `docker kill` command.
  First, we identify the container ID by running `docker ps`, which lists active containers. In this case, the container ID is **f2e561c7b67c**.
  Using the command `docker kill f2e561c7b67c`, we terminate the running container. To confirm the successful termination, we run `docker ps` again, and the output should be empty, verifying that no active containers are running. This ensures a clean environment after the Jenkins build process.

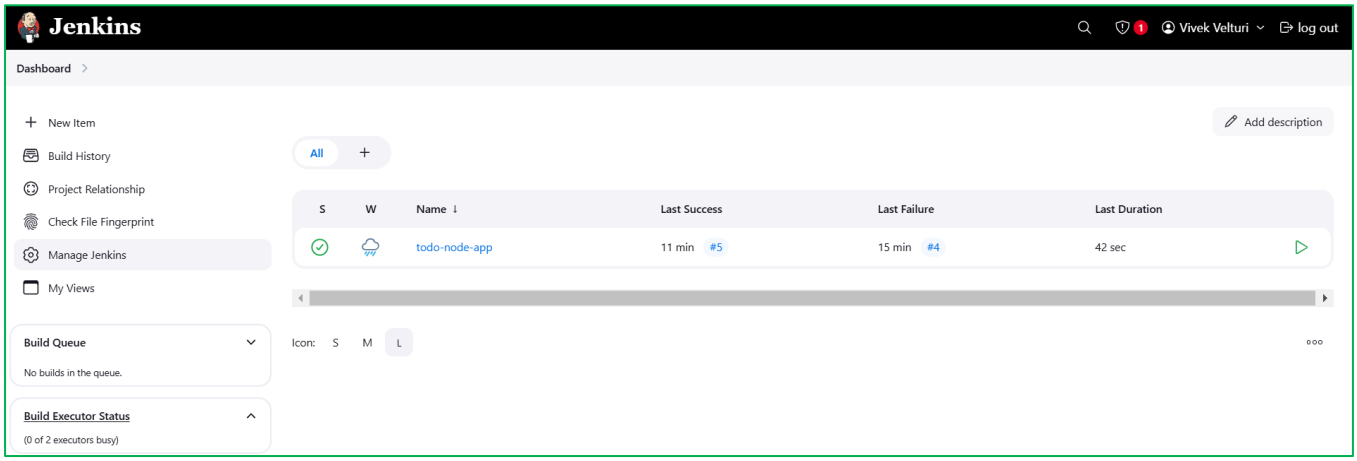
```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ docker ps
CONTAINER ID   IMAGE           COMMAND          CREATED          STATUS          PORTS                                       NAMES
f2e561c7b67c   todo-node-app   "node app.js"    12 minutes ago   Up 12 minutes   0.0.0.0:8000->8000/tcp, :::8000->8000/tcp   node-app-container
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ docker kill f2e561c7b67c
f2e561c7b67c
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$
```

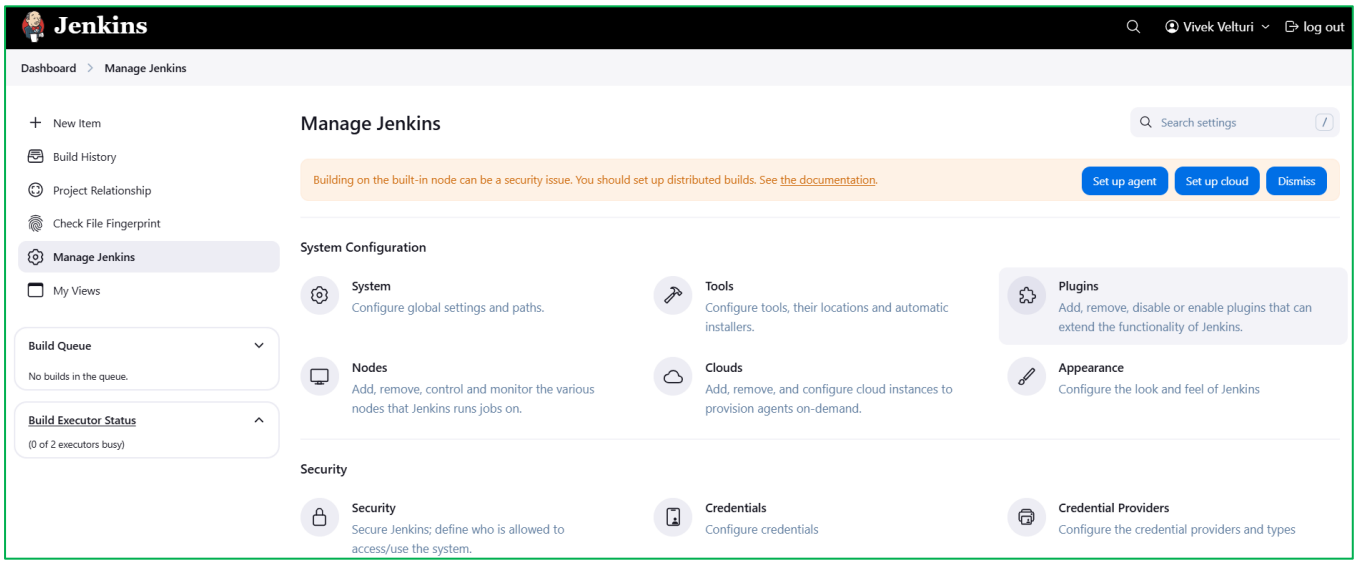
- After terminating the Docker container, if we try accessing the application URL again, we will notice that it is no longer functional.



Automating Build Triggers Using GitHub Webhooks and Jenkins Integration

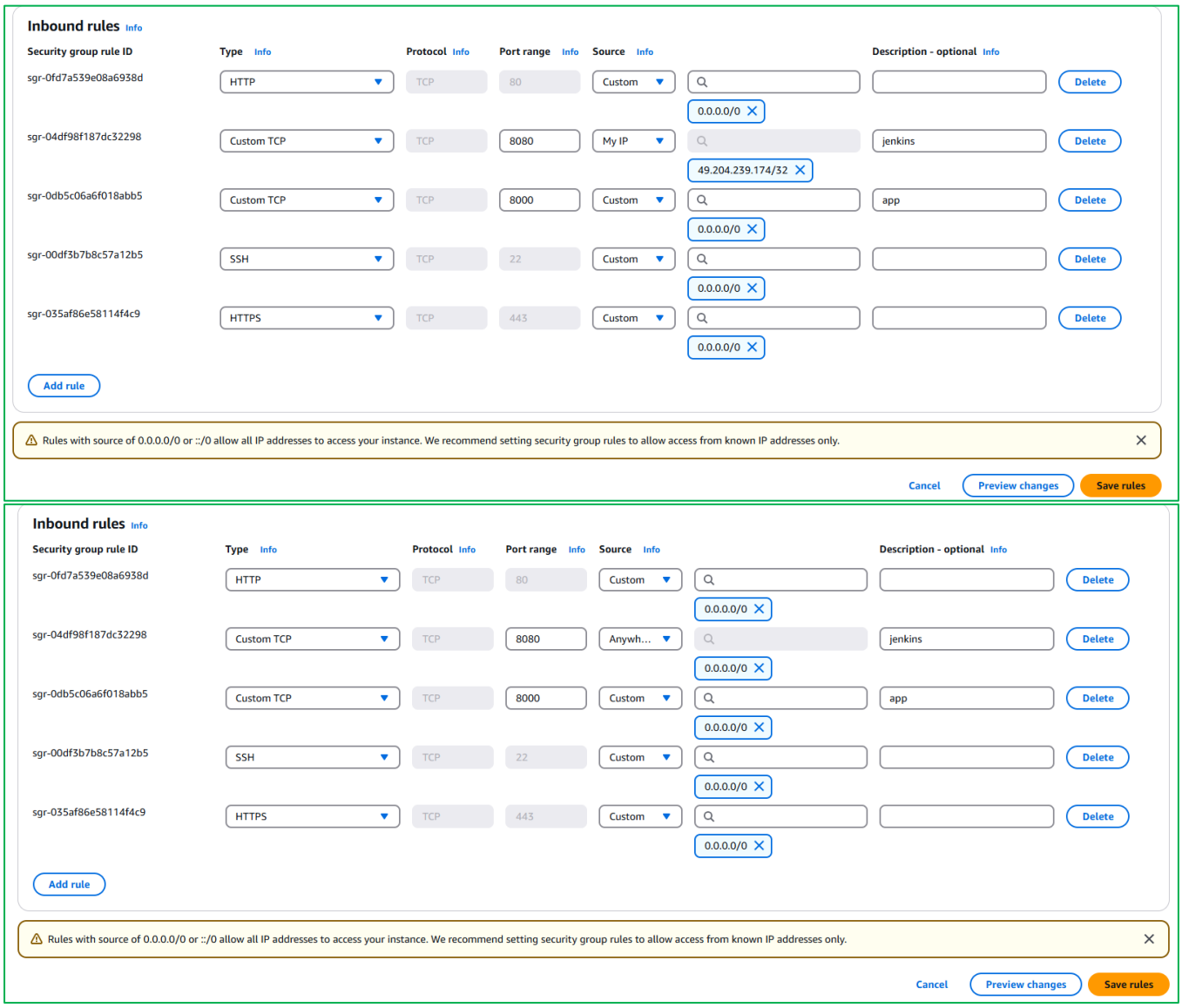
- Now, let's automate the build scheduling so that any new commit pushed to the GitHub repository automatically triggers a build in Jenkins.
- To automate build scheduling so that any new commit in the GitHub repository triggers a build in Jenkins, we first need to install the necessary plugin.
  Navigate to **Manage Jenkins** in the Jenkins dashboard, then go to **Plugins**.
  In the **Available Plugins** section, search for **GitHub Integration**, select it, and proceed with the installation.





- After installation, restart Jenkins to apply the changes.

- After restarting Jenkins, log in again to ensure that the changes take effect and the GitHub Integration plugin is properly installed.
- Now, we need to configure a webhook in our GitHub repository.
- Before proceeding, remember that we have currently restricted access to our port 8080 to only our IP address. To allow others to access it, we need to update the configuration to permit access from any IPv4 address.

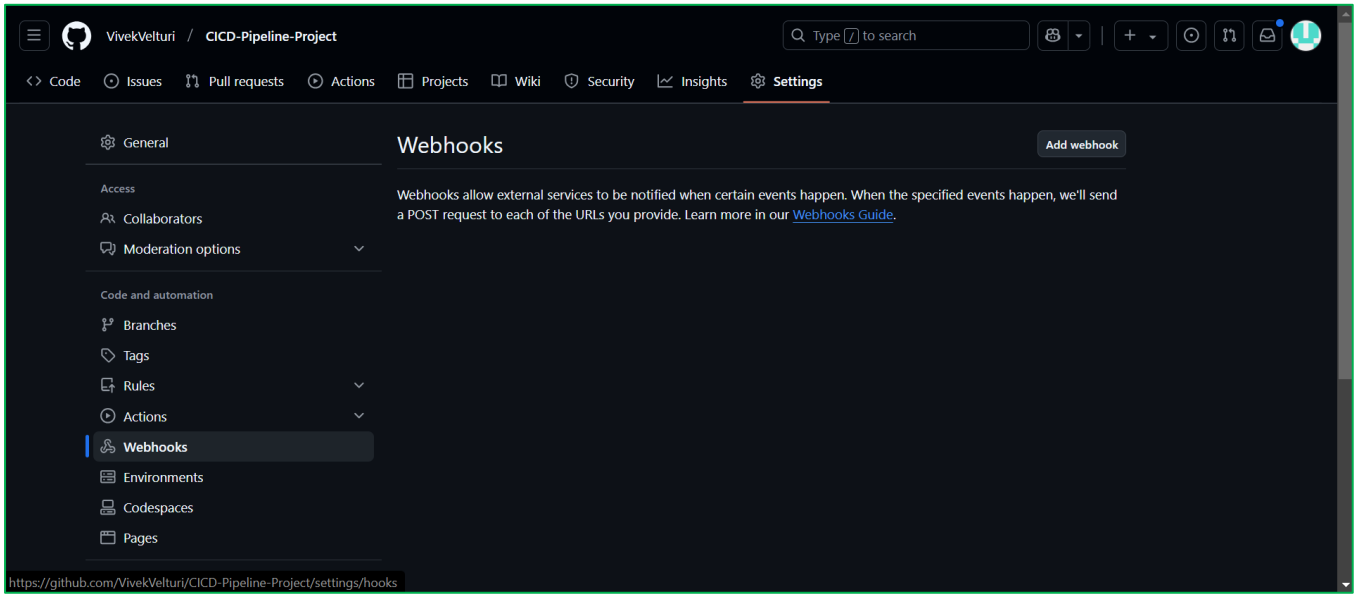


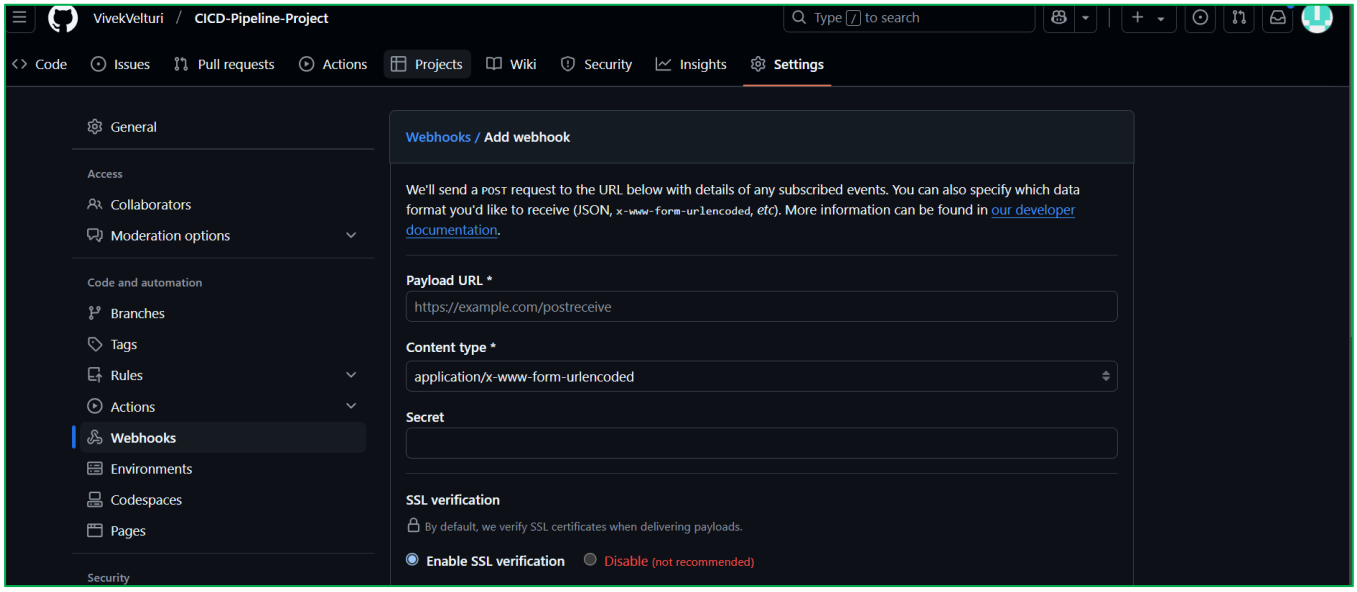
- Now to setup the webhook in our GitHub repository. Navigate to the **Settings** of the repository from which we are deploying our application.

What is a Webhook?

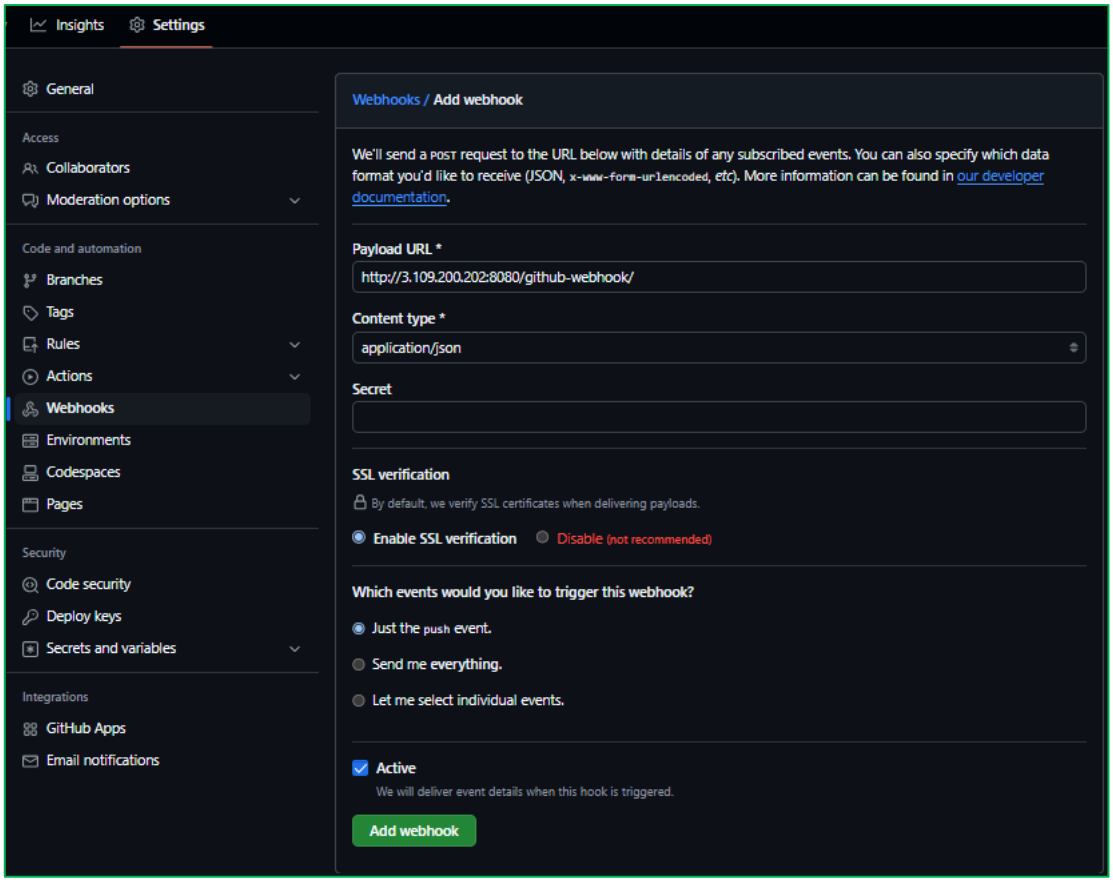
A webhook is a method for one application to instantly notify another application when a specific event occurs. It sends data automatically to a specific URL (webhook URL) as soon as the event happens, removing the need for the receiving application to repeatedly check for updates. This makes webhooks efficient for real-time updates, such as triggering builds in Jenkins when code is pushed to GitHub. Proper security, like HTTPS and authentication, is essential to ensure safe and reliable communication.

- Once you're in the settings, navigate to the "**Webhooks**" section on the left-hand side, and then click on "**Add Webhook**" to create a new one.

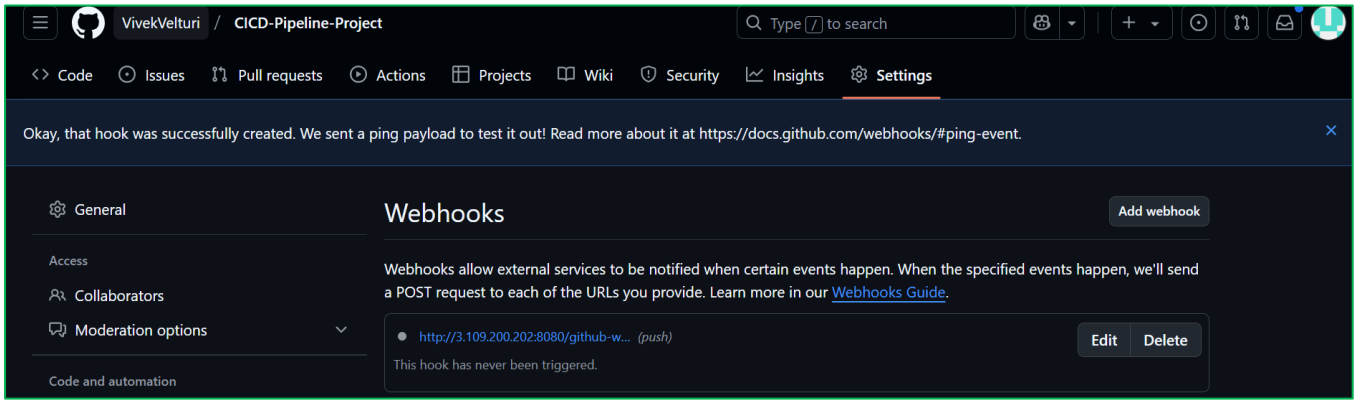




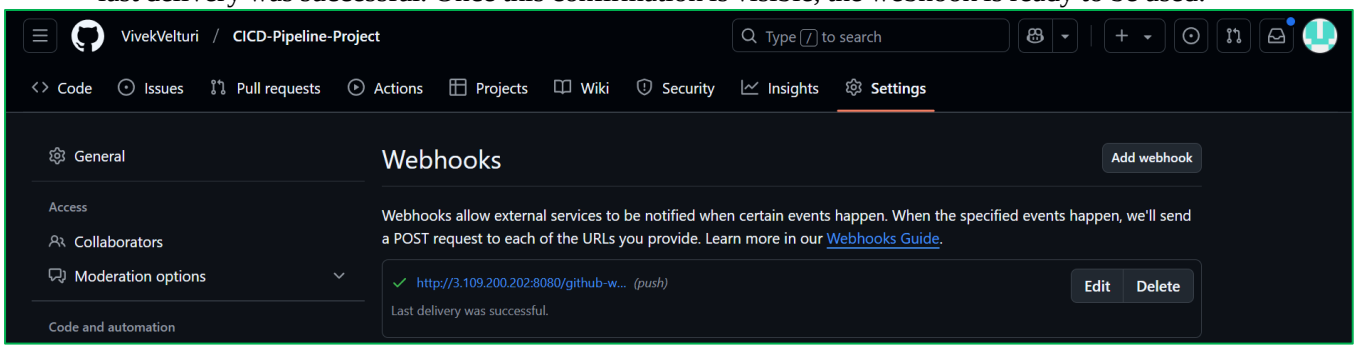
- In the "**Payload URL**" field, **enter the URL by combining your Jenkins dashboard URL with /github-webhook/**
  Set the **Content Type** to application/json,
  Enable **SSL verification**,
  Under **Which events would you like to trigger?**
  Select **Just the push event**.
  Finally, ensure the **Active** checkbox is selected,
  Click **Add Webhook** to complete the setup.



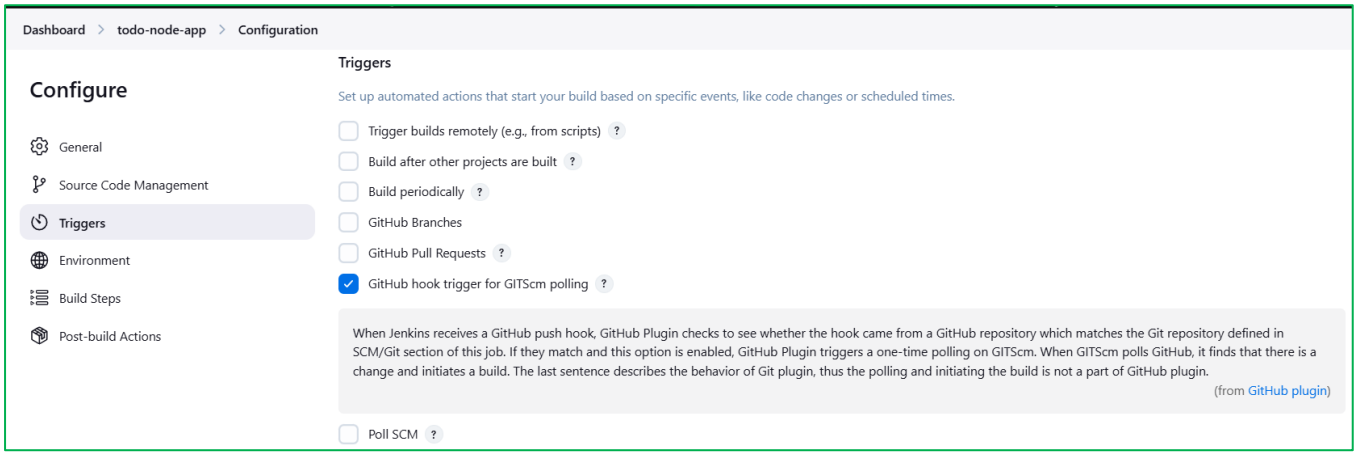
- We have successfully added the webhook to our repository. Currently, it shows that the hook has never been triggered.



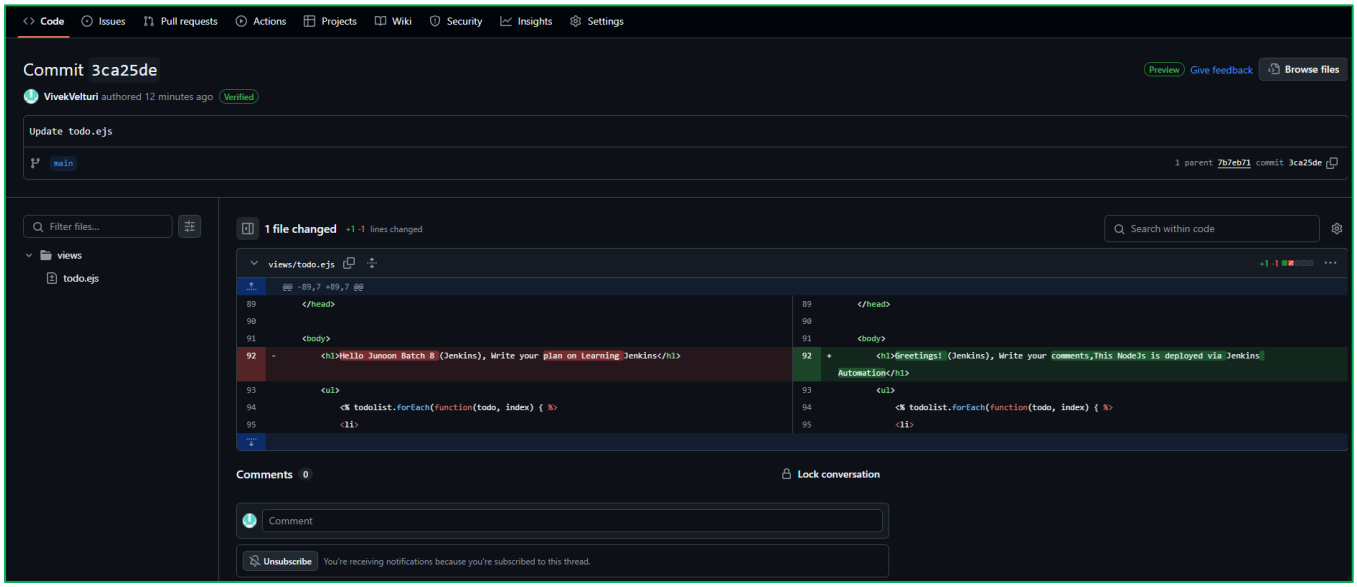
- After waiting for a while, a green tick mark will appear next to the webhook, indicating that the last delivery was successful. Once this confirmation is visible, the webhook is ready to be used.



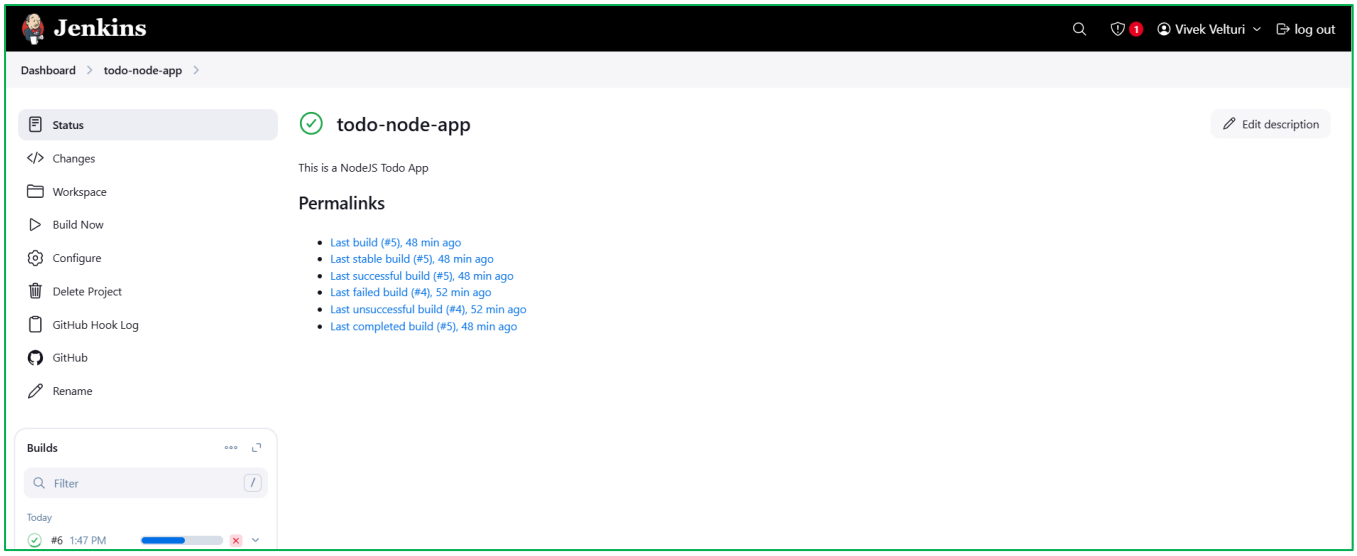
- You can now leverage it to automate workflows, such as triggering builds in Jenkins whenever a push event occurs in the repository.
- Now, go back to your Jenkins job named **todo-node-app**.
  Under the **Build Triggers** section,
  Select the option **GitHub hook trigger for GITScm polling**.
  Once selected, click **Save** to apply the changes.



- This will ensure that your Jenkins job is automatically triggered whenever a push event is detected in the connected GitHub repository via the webhook.
- Go to your GitHub repository, make edits to the code, and commit the changes.



- Once you return to Jenkins, you'll notice that the build for **todo-node-app** has been automatically triggered. This happens because the GitHub webhook is now configured to notify Jenkins of any push events, enabling seamless automation of the build process.



- After the build is automatically triggered, click on the **Build Number** in the Jenkins job dashboard. Then, click on **Console Output** to view the detailed logs of the build process. This will show you the real-time progress, including any steps, commands, or errors that occur during the build.

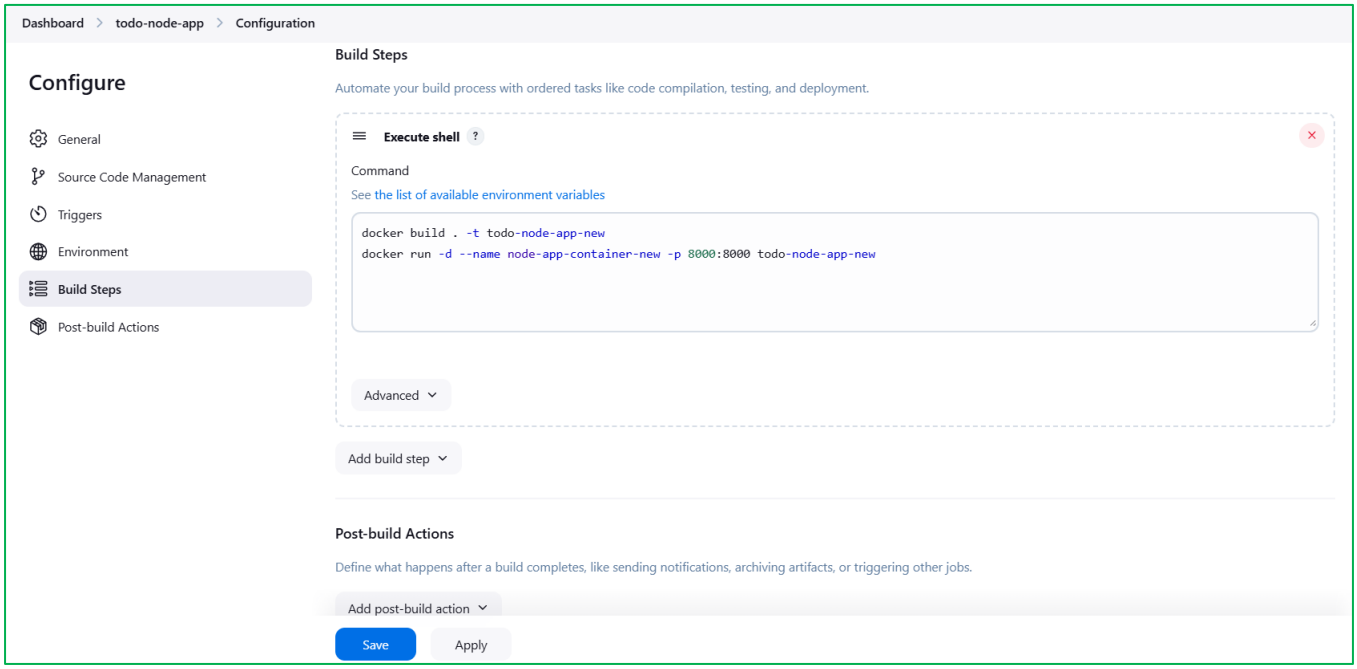
```
+ docker build . -t todo-node-app
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  25.33MB

Step 1/7 : FROM node:12.2.0-alpine
 ---> f391dabf9dce
Step 2/7 : WORKDIR /node
 ---> Using cache
 ---> 547322785f56
Step 3/7 : COPY . .
 ---> 5f6d5af27d26
Step 4/7 : RUN npm install
 ---> Running in 852cbf7f9829
[91mnpm WARN read-shrinkwrap This version of npm is compatible with lockfileVersion@1, but package-lock.json was generated for lockfileVersion@2. I'll try to do
my best with it!
[0m
> ejs@2.7.4 postinstall /node/node_modules/ejs
> node ./postinstall.js

Thank you for installing [35mEJS[0m: built with the [32mJake[0m JavaScript build tool ([32mhttps://jakejs.com/[0m)

[91mnpm[0m[91m [0m[91mWARN[0m[91m my-todolist@0.1.0 No repository field.
[0m[91mnpm[0m[91m WARN my-todolist@0.1.0 No license field.
[0m[91m
[0mupdated 291 packages and audited 291 packages in 11.326s
found 28 vulnerabilities (6 low, 3 moderate, 16 high, 3 critical)
  run `npm audit fix` to fix them, or `npm audit` for details
 ---> Removed intermediate container 852cbf7f9829
 ---> fee413b63fde
Step 5/7 : RUN npm run test
 ---> Running in d41315fc84b9

> my-todolist@0.1.0 test /node
> mocha --recursive --exit



  Simple Calculations
This part executes once before all tests
    Test1
executes before every test
      ✓ Is returning 5 when adding 2 + 3
executes before every test
      ✓ Is returning 6 when multiplying 2 * 3
    Test2
executes before every test
      ✓ Is returning 4 when adding 2 + 3
executes before every test
      ✓ Is returning 8 when multiplying 2 * 4
This part executes once after all tests


  4 passing (12ms)

 ---> Removed intermediate container d41315fc84b9
 ---> 3e9235e9d554
Step 6/7 : EXPOSE 8000
 ---> Running in a8ea8b9d3c9e
 ---> Removed intermediate container a8ea8b9d3c9e
 ---> 05bcd3ddf6d5
Step 7/7 : CMD ["node","app.js"]
 ---> Running in 2f89aca84577
 ---> Removed intermediate container 2f89aca84577
 ---> c64ececcb0f0
Successfully built c64ececcb0f0
Successfully tagged todo-node-app:latest
+ docker run -d --name node-app-container -p 8000:8000 todo-node-app
docker: Error response from daemon: Conflict. The container name "/node-app-container" is already in use by container
"f2e561c7b67c6f07a4357f004cf0db07cfff765ebf401d584dd2d8d2f6c0a116". You have to remove (or rename) that container to be able to reuse that name.
See 'docker run --help'.
Build step 'Execute shell' marked build as failure
Finished: FAILURE
```

- In the **Console Output**, you'll notice that the build failed because the container named **node-app-container** already exists from the previous build
- you can check the status of all containers, including the previous **node-app-container**, by running the command: `docker ps -a`

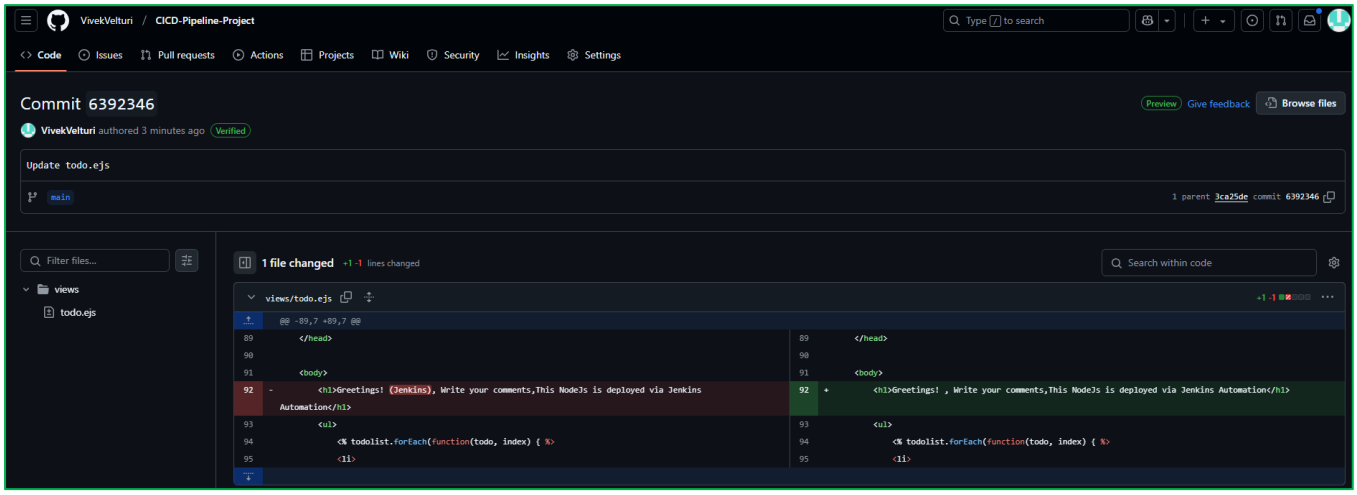
```
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$ docker ps -a
CONTAINER ID   IMAGE          COMMAND          CREATED          STATUS                          PORTS     NAMES
f2e561c7b67c   0a9c701cd8c9   "node app.js"    55 minutes ago   Exited (137) 41 minutes ago               node-app-container
c3e5ed9e0a55   ebc9b2264759   "node app.js"    2 hours ago      Exited (137) About an hour ago            node-todo-app
ubuntu@ip-172-31-11-98:/var/lib/jenkins/workspace/todo-node-app$
```

- You have two options to resolve the issue of the existing **node-app-container**:
  - **Manually Remove the Container**:
    Run the following command in your instance terminal to forcefully remove the container:
    `docker rm -f node-app-container`

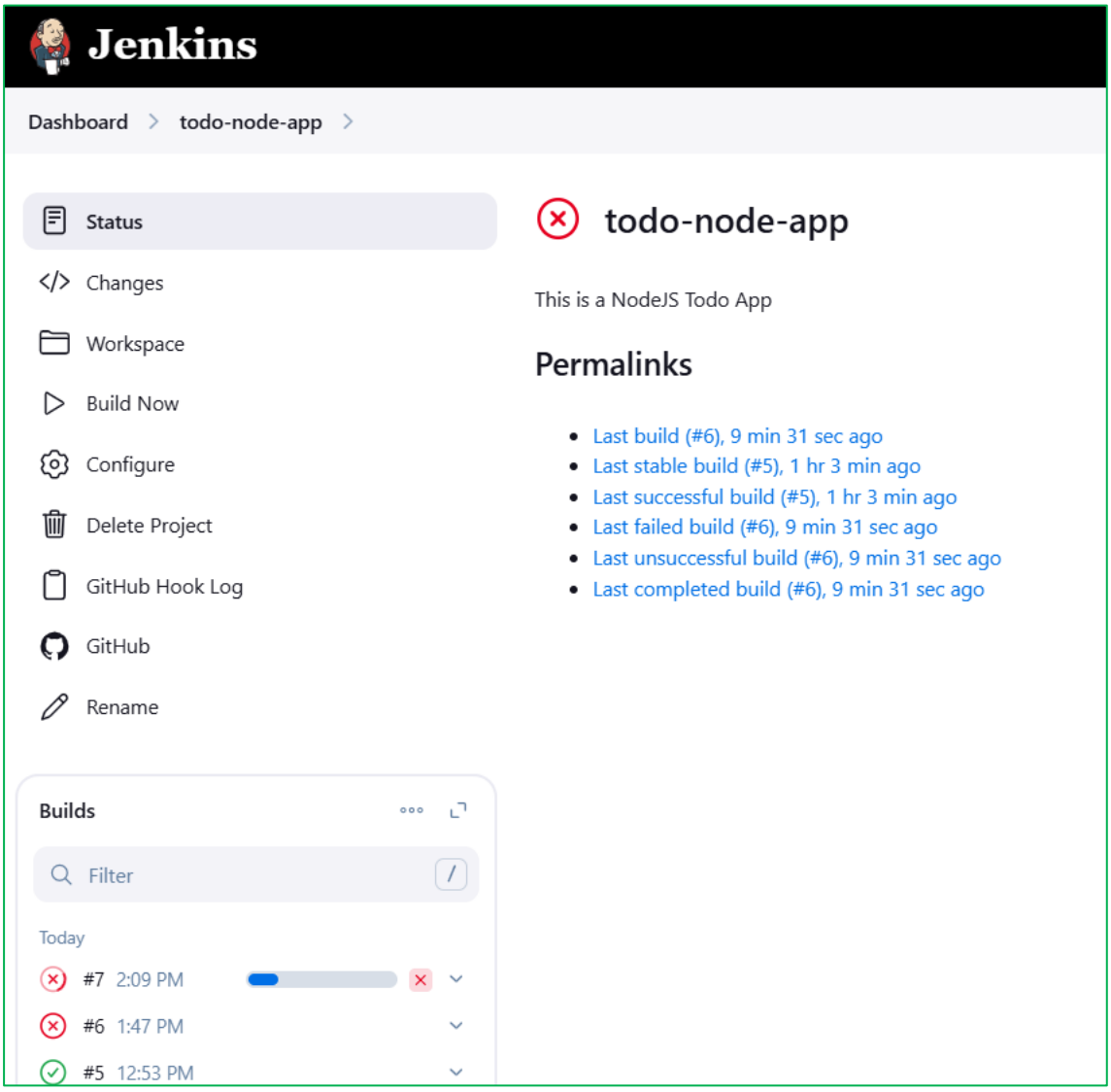
  - **Modify the Execute Shell in Jenkins Job Configuration**:
    Update the **Execute Shell** section under the **Build Steps** of your Jenkins job configuration.
    Update the docker run command to use a unique name for each build.
    This ensures that each build creates a container with a unique name, preventing conflicts with previous builds. Save the changes, and the next build will use the new naming convention.



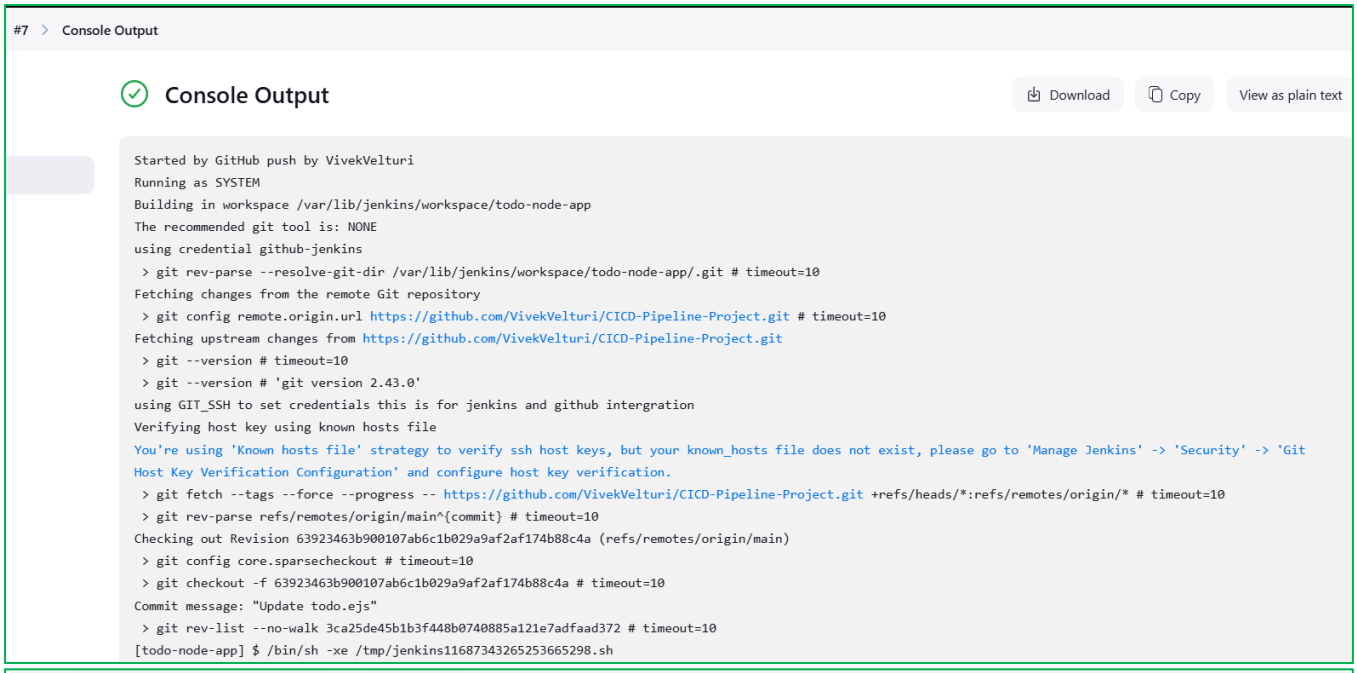
- Now that we've updated the commands in the **Execute Shell** section of the **Build Steps** in our Jenkins job, let's test the changes. Make a edit in your GitHub repository and commit the changes.



- This will trigger the GitHub webhook, which should automatically start a new build in Jenkins.



- Once the build is triggered:
  Go to your Jenkins job dashboard and check the **Build History**.
  Click on the latest build number and view the **Console Output** to verify that the build runs successfully.
  Confirm that the new container is created with a unique name (in our case: **node-app-container-new**).

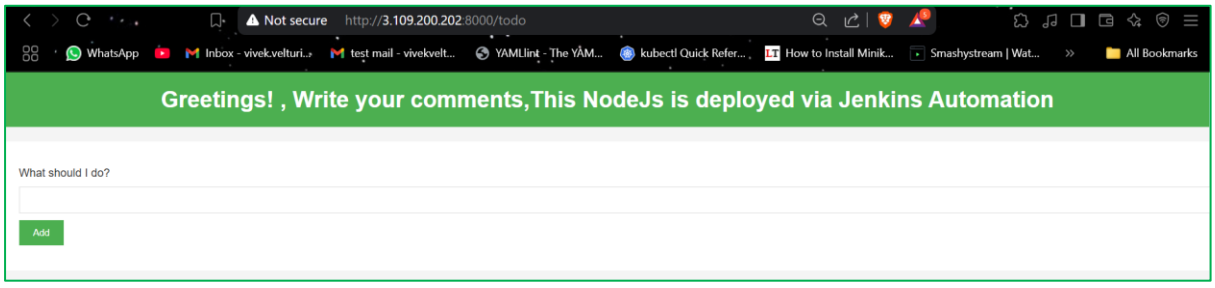



```
+ docker build . -t todo-node-app-new
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  25.34MB

Step 1/7 : FROM node:12.2.0-alpine
 ---> f391dabf9dce
Step 2/7 : WORKDIR /node
 ---> Using cache
 ---> 547322785f56
Step 3/7 : COPY . .
 ---> 442afe3b6510
Step 4/7 : RUN npm install
 ---> Running in 725c394e140a
[91mnpm WARN [0m[91mread-shrinkwrap This version of npm is compatible with lockfileVersion@1, but package-lock.json was generated for lockfileVersion@2. I'll try to do my best with it!
[0m
> ejs@2.7.4 postinstall /node/node_modules/ejs
> node ./postinstall.js

Thank you for installing [35mEJS[0m: built with the [32mJake[0m JavaScript build tool ([32mhttps://jakejs.com/[0m)

[91mnpm[0m[91m WARN my-todolist@0.1.0 No repository field.
npm WARN my-todolist@0.1.0 No license field.
[0m[91m
[0mupdated 291 packages and audited 291 packages in 10.887s
found 28 vulnerabilities (6 low, 3 moderate, 16 high, 3 critical)
```

```
  run `npm audit fix` to fix them, or `npm audit` for details
 ---> Removed intermediate container 725c394e140a
 ---> dde0c178d18c
Step 5/7 : RUN npm run test
 ---> Running in 48a95c32ff4f

> my-todolist@0.1.0 test /node
> mocha --recursive --exit



  Simple Calculations
This part executes once before all tests
    Test1
executes before every test
      ✓ Is returning 5 when adding 2 + 3
executes before every test
      ✓ Is returning 6 when multiplying 2 * 3
    Test2
executes before every test
      ✓ Is returning 4 when adding 2 + 3
executes before every test
      ✓ Is returning 8 when multiplying 2 * 4
This part executes once after all tests


  4 passing (10ms)

 ---> Removed intermediate container 48a95c32ff4f
```

```
 ---> 2eb8ad46852a
Step 6/7 : EXPOSE 8000
 ---> Running in 999d58c25212
 ---> Removed intermediate container 999d58c25212
 ---> 6ba7088e26ca
Step 7/7 : CMD ["node","app.js"]
 ---> Running in 8be0b4b1d69e
 ---> Removed intermediate container 8be0b4b1d69e
 ---> efcad34ab457
Successfully built efcad34ab457
Successfully tagged todo-node-app-new:latest
+ docker run -d --name node-app-container-new -p 8000:8000 todo-node-app-new
e54831f433a3be1cec6f415e0ebfa06fff7628261b3f8fe7359fed73ca3e7c91
Finished: SUCCESS
```

- From the Console Output, we can confirm that the build was successful. Now, open your application by navigating to your instance's public IP address on port 8000 (e.g., http://<your-instance-ip>:8000).
- You should see the updated application running, reflecting the changes you made in the code and committed to the GitHub repository.



- This will demonstrate that the changes are working as expected, and the build process is now conflict-free. It also demonstrates that the entire workflow—triggering a build via a GitHub webhook, building the Docker container, and deploying the updated application—is functioning as expected. The changes you made in the repository are now live and visible in the application.

The project is now complete. Here's a quick summary of what you've achieved:

1. **GitHub Webhook Integration**: Successfully set up a webhook to trigger Jenkins builds automatically on code changes.
2. **Jenkins Job Configuration**: Configured the Jenkins job to build and deploy the application using Docker, with unique container names to avoid conflicts.
3. **Automated Deployment**: Ensured that changes committed to the GitHub repository are automatically built and deployed.
4. **Application Testing**: Verified that the application runs correctly and reflects the latest changes by accessing it via the instance's public IP and port.