**React Application Deployment**

**Web sites:**

* **GitHub Repo URL :** [**https://github.com/Rajalakshmi-144/React-web-application-docker.git**](https://github.com/Rajalakshmi-144/React-web-application-docker.git)
* **Deployed site URL :** [**http://3.248.219.82:80**](http://3.248.219.82)
* **Prometheus targets URL:** [**http://3.248.219.82:9090/targets**](http://3.248.219.82:9090/targets)
* **Docker image: dev repo – rajalakshmi1404/react-image**
* **Prod repo – rajalakshmi1404/react-image-prod**

# **Overview**

The React application is a mobile e-commerce website that has been **containerized using Docker**. The deployment process is **automated using Jenkins**, which manages the continuous integration and delivery (CI/CD) pipeline. Additionally, the **Docker image** is pushed to **Docker Hub** for easy access and deployment across different environments.

To monitor the application's health and performance, we use **Prometheus with node-exporter** for collecting metrics and **Grafana** for visualizing those metrics. If the application experiences downtime or performance degradation, an alert system is in place to **notify the team** of any issues.

**Goals**

1. We deploy our React application using **Docker** to ensure consistent operation across different environments, simplify dependency management, and facilitate easy deployment to the cloud.
2. The **production build** of our React app is served through the **Nginx** web server. **Nginx** is chosen for its high performance, scalability, and flexibility, making it the best-suited option for serving the static content produced by the React build process.
3. Push your code to **GitHub**, so you can access it from any machine with Git installed, it allows multiple developers to collaborate on the same codebase. Also you can trigger automated builds, tests, and deployments, allowing for continuous integration of your changes and smoother deployment processes
4. **Jenkins** is a widely used automation server for continuous integration and continuous delivery (CI/CD), known for its flexibility, scalability, and rich ecosystem of plugins.
5. We automate the deployment process through **Jenkins** for our CI/CD pipeline. By connecting **GitHub** to **Jenkins**, we configure auto-triggers for both the **main** and **dev** branches, ensuring continuous integration and deployment.
6. The Docker image is pushed to **Docker Hub**, a cloud-based registry where Docker images are stored, making it easier to share and access them. The image can be pulled from **Docker Hub** to different environments (such as staging, testing, or production) without needing to rebuild it each time, ensuring consistency across those environments.
7. To monitor the application's health and performance, we use **Prometheus** with the **node exporter** to collect time-series data (metrics) from application endpoints. **Grafana** is used to visualize these metrics in customizable dashboards, making it easier to monitor application performance and health.
8. **Alert manager**, a component of Prometheus, is used for managing alerts. It can send notifications to different channels (such as email or Slack) when the application's health condition triggers an alert, such as when the application goes down.

# **Prerequisite**:

1. AWS Cloud
2. AWS EC2 instance
3. Docker with Docker compose
4. Java
5. Jenkins
6. GitHub
7. Docker hub
8. Prometheus
9. Node-Exporter
10. Grafana
11. Alert manager

**Action to taken:**

## Log into the **aws console** and create an EC2 instance in aws with **t2.micro** as instance type and configure the **security group** to allow **HTTP/HTTPS** access to the react application and allow **SSH** port **22** only from my ip address to access the server.

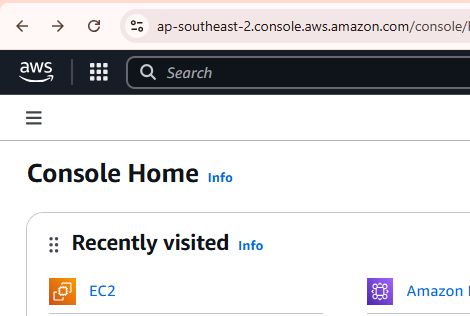
1. Install **Docker** and **Docker compose**. Create a **Docker file** to use nginx for serve the static content to be deployed to production. Create a **docker-compose.yml** file to build the Docker file and containerize the Docker application
2. Create 2 **bash scripts** to automate repetitive, complex tasks. **build.sh** for building the Docker image and **deploy.sh** for deploying the Docker image to server
3. Create **.gitignore** and **.Docker ignore** files to ensure that unnecessary or sensitive files are not included in your version control system and Docker image builds.
4. Create **GitHub repository with personal access token** for remote storage and version control
5. Create a devbranch in the local repository and **login** to the **GitHub** to push the files from the local repository to the GitHub(remote) repository.
6. Create 2 **Docker hub** repositories, one for **dev** repository which is **Public** for development, easier for collaborationand another for **prod** repository which is Privatefor production, ensuring only authorized users can access.
7. Install **java** and **Jenkins** .Create **jenkinsfile** to execute the build process such as build, push and deploy the application to the server
8. Connect **GitHub** to **Jenkins** and configure **auto-triggers** for both main and dev branches.
9. When code is pushed to the **dev** branch on GitHub, Jenkins will automatically build a Docker image and push it to the **dev** repository in Docker Hub.
10. When code is merged into the **main** branch on GitHub, Jenkins will build the Docker image again and push it to the **prod** repository in Docker Hub.
11. Install **Prometheus** and **node-exporter**. Update the **prometheus.yml** file to scrape the metrics exposed by the react application. Create **Prometheus. Service** file which manage the Prometheus systemd service, it can be configured to start Prometheus automatically when the system boots or on failure
12. Update the **nginx default conf** configuration file to serve the health check endpoint of react application and make sure Prometheus can check the status
13. Install **grafana** and configure **Prometheus data source** and **query** to fetch necessary metrics to be displayed in the dashboard of react application for real-time monitoring.
14. Create the **alert.rules** file in Prometheus to define **alerting rules** that allowing you to proactively detect and trigger notifications when certain conditions or thresholds are met in the monitored metrics
15. Install **alert manager**, update **alertmanager.yml** configuration file that defines how alerts are grouped, routed, and sent to different notification channels like email. Create **alertmanager.service** file which manages alert manager systemd service.
16. Check if the alert is triggered and the **notification** is sent to the recipient email, when the react application is **down**
17. Resolving the react application

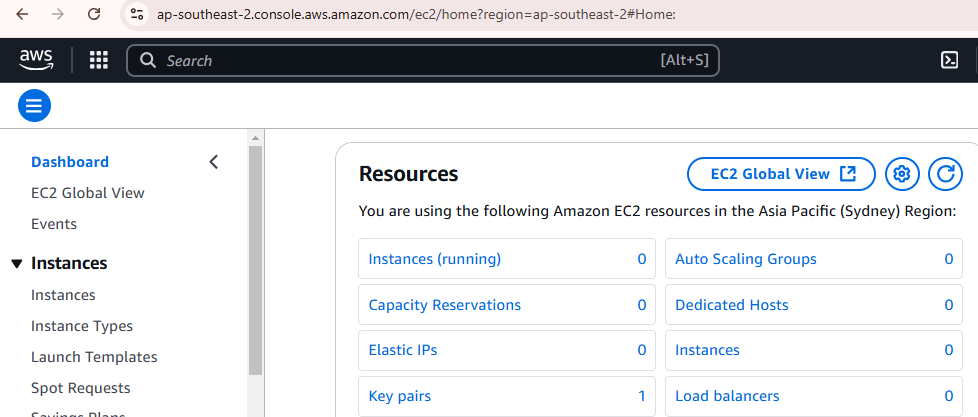
**Steps to be followed:**

**Step 1:**

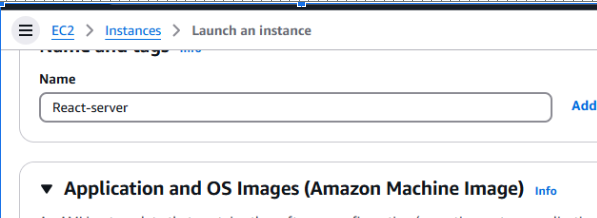
Log into the **aws console.** Create an **EC2** instance in aws with **t2.micro** as instance type and configure the **security group** to allow HTTP/HTTPS access to the react application and allow SSH port 22 only from my ip address to access the server.

* Login to the **aws console** and go to **EC2**

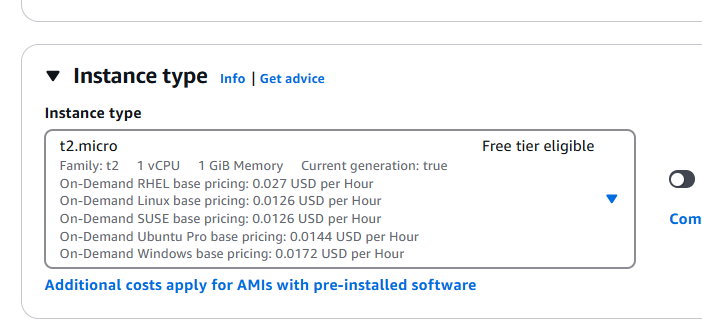
****

****

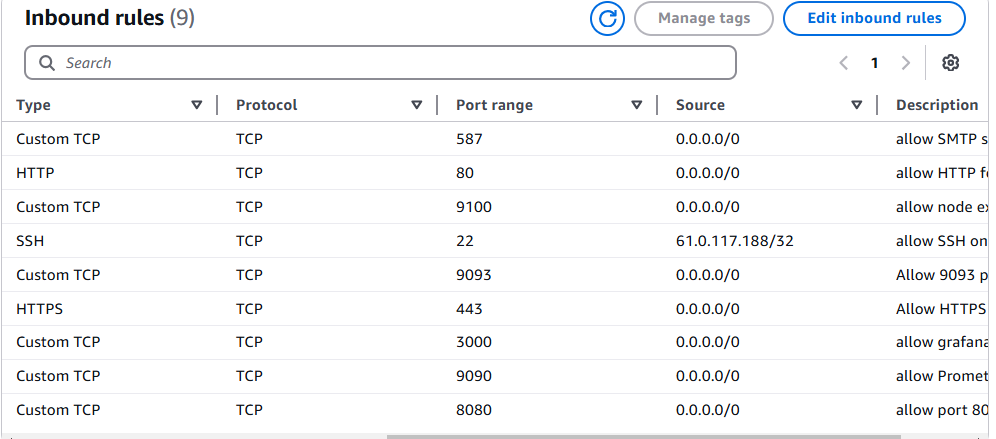
* Launch the **instance** and name it as **“React-server”**

****

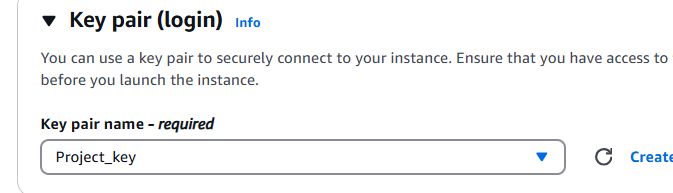
* Create a instance type as **t2.micro**

****

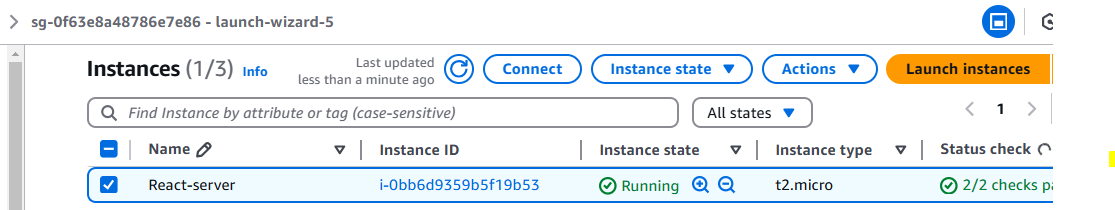
* Create a **security group**
* To allow HTTP port **80**  for access the react application
* To allow HTTPS port **443** for access the react application
* To allow SSH port **22** for access only **my ip address** to React-server
* Allow port **8080** for access **Jenkins**
* Allow port **9090** for access **Prometheus**
* Allow port **9100** for access **node exporter**
* Allow port **3000** for access **grafana**
* Allow port **9093** for access **alert manager**
* Allow port **587** for access **SMTP** server



* Create the **key pair** and name it as “**Project\_key**”



* Then keeping default options for the rest of the settings, click launch instance.
* Instance is created and **running**.

****

**Step 2:**

**Install Docker** and **Docker compose**. Create a **Docker file** to use nginx for serve the static content to be deployed to production. Create a **docker-compose.yml** file to build the Docker file and containerize the Docker application

* **Sudo su** – to switch to root user

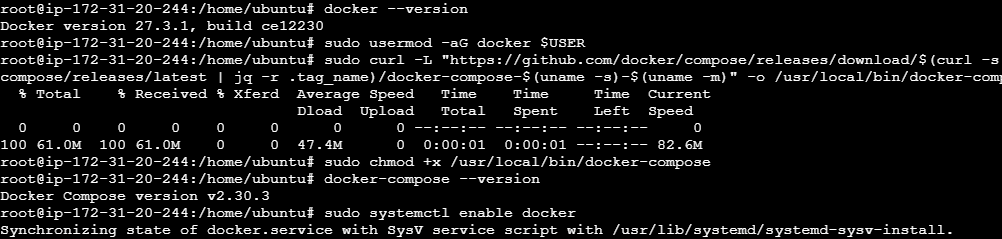
****

* **Sudo apt-get install docker-io –** install docker and its repository

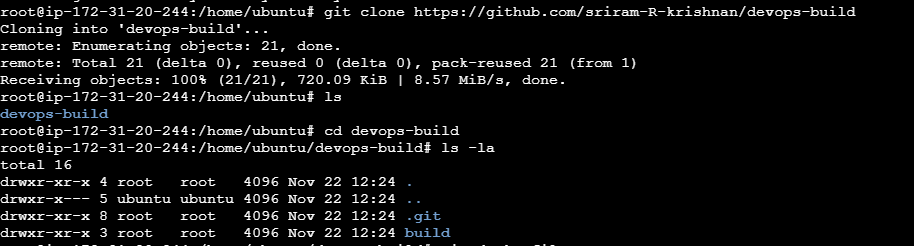
****

****

* **Docker - -version –** to check the docker version
* **sudo usermod -aG docker $USER -** to add a user to the Docker group
* **sudo curl -L "https://github.com/docker/compose/releases/download/$(curl -s https://api.github.com/repos/docker/compose/releases/latest | jq -r .tag\_name)/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose –**  install docker compose
* **sudo chmod +x /usr/local/bin/docker-compose -** change the file permission
* **docker-compose –version –** check the docker compose version
* **Sudo systemctl enable docker**  – start and enable the docker

****

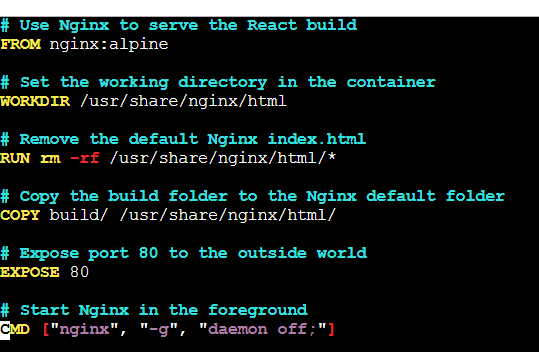
* **Git clone** [**https://github.com/sriram-R-krishnan/devops-build**](https://github.com/sriram-R-krishnan/devops-build) **–** Clone the git repo URL which has the react build process folder
* **Cd devops-build –** go into the folder
* **Ls -la –**  list all the files and folders

****

* **Vim dockerfile –** Create the docker file

****

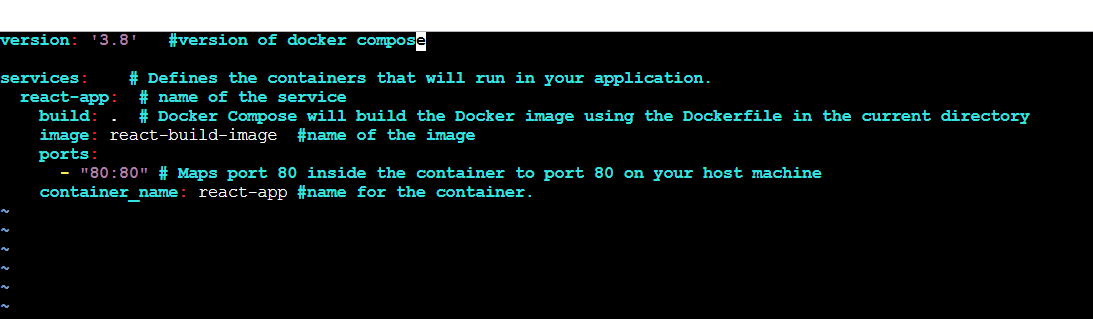
* **Docker file contains**  series of instructions on how to build a Docker image

****

* Vim **docker-compose.yml** –Create docker-compose.ymlfile

****

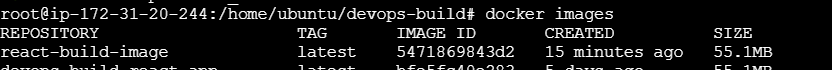
* **Docker-compose.yml file** includes instructions on how to build and run the Docker containers for each service. By using it in combination with Docker file and proper service configuration, you can easily deploy applications

****

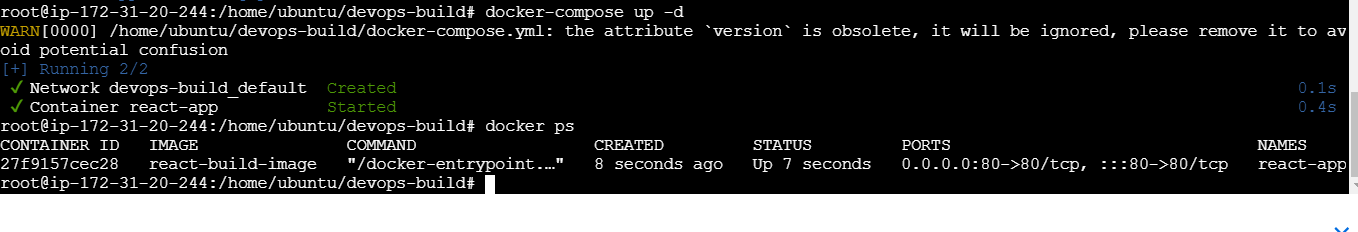
* Checking whether the docker image is building and the container is deploying the react application in our server
* **Docker-compose build** – to build the docker image based on the configuration in the docker-compose.yml file.

****

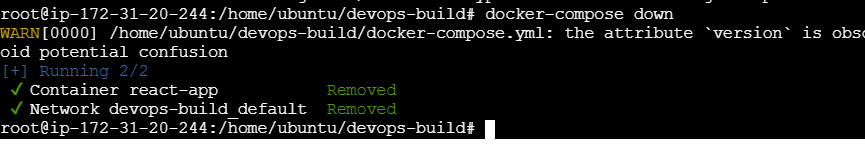
* **Docker images –** list the images

****

* **Docker-compose up -d –** build the image , create the containers and start the service defined in the docker-compose.yml in detached mode
* **Docker ps** - list the running container
* Now we containerized the react application

****

* **docker-compose down –** remove the container
* since we want to build through Jenkins for automation process

****

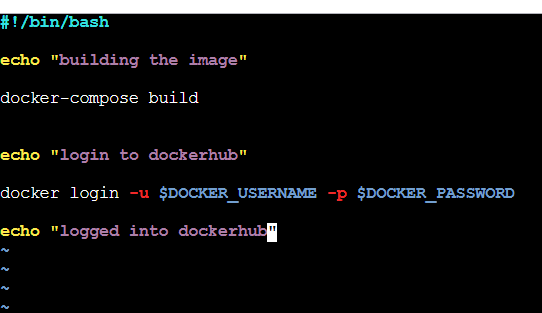
**Step 3:**

Create 2 **bash scripts** to automate repetitive, complex tasks. **build.sh** for building the docker image and **deploy.sh** for deploying the docker image to server

* **Vim build.sh** – create build.sh file

****

* **build.sh file** contains build command and docker login command

****

* **Vim deploy.sh –** create deploy.sh file

****

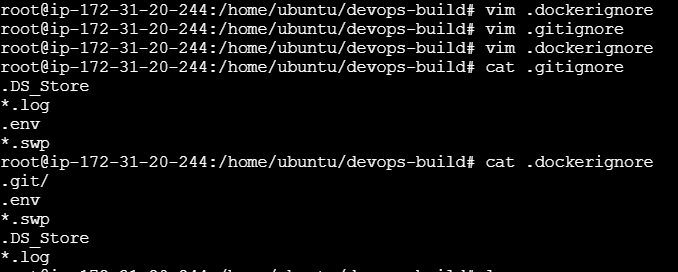
* **deploy.sh file** contains deploy command

****

**Step 4:**

Create **.gitignore** and **.dockerignore** files to ensure that unnecessary or sensitive files are not included in your version control system and Docker image builds.

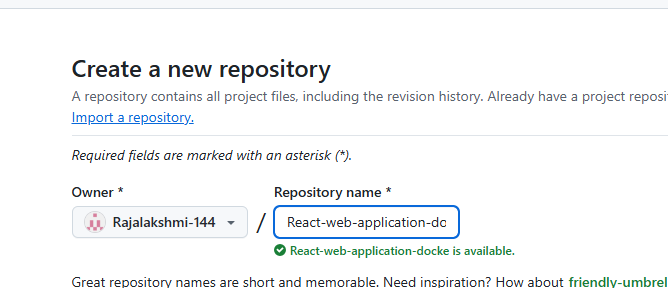
* **Vim .dockerignore**  – create .dockerignore file and add the file to ignore
* **Vim .gitignore** – create .gitignore file and add the file to ignore
* **Cat .dockerignore** – view the .dockerignore file
* **Cat .gitignore –**  view the .gitignore file

****

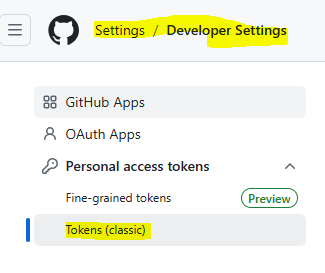
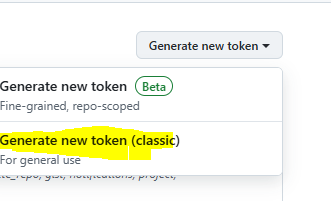
**Step 5:**

Create **GitHub repository with personal access token** for remote storage and version control

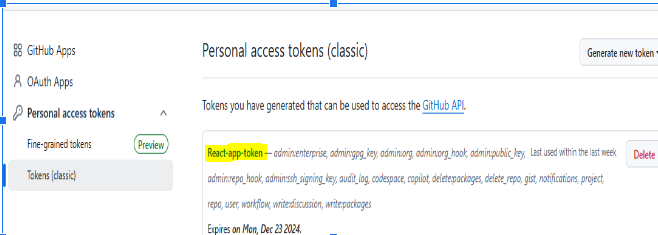
* **Login** to GitHub. Create new repository and name it as **“React-web-application-docker”**

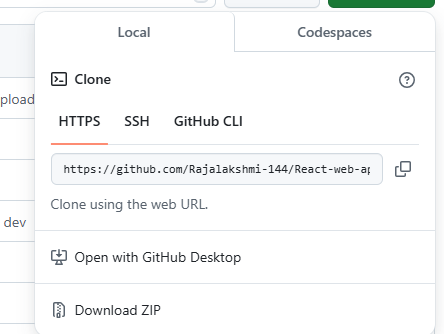
****

* Go to **setting → Developer setting → Token (classic) →generate token →generate new token (classic) →** name the token as **“React-app-token”** save it.

** **

* A new token of **React-app-token** is **created**. Copy and paste in notepad for later use

****

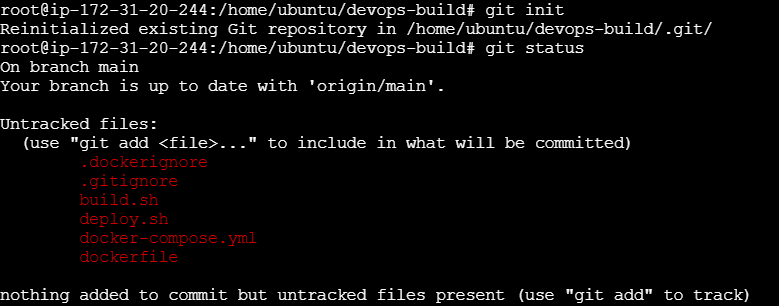
****

**GitHub Repo URL: https://github.com/Rajalakshmi-144/React-web-application-docker.git**

**Step 6:**

Create a **dev** branch in the local repository and **login** to the **GitHub** to push the files from the local repository to the GitHub(remote) repository.

* **Git init –** initialize the git repository
* **Git status –**  checking the current file to be add to git repo

****

* **Git branch dev –** create new branch as dev

****

* **Git branch –** check the current branch



* **Git add . –** add all the files in working directory to local repo

****

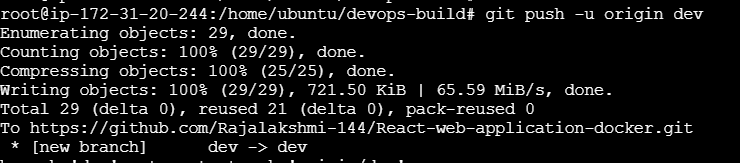
* **Git commit -m “Initial commit”**  – commit the file with description

****

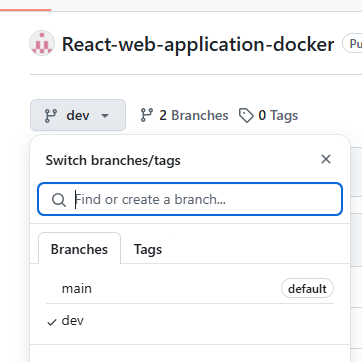
* **Git remote add origin** [**https://github.com/Rajalakshmi-144/React-web-application-docker.git**](https://github.com/Rajalakshmi-144/React-web-application-docker.git)–add remote repository
* **Git branch -M dev** – add the branch
* **Git push -u origin dev** — push the dev branch to GitHub repo
* Enter the username **Rajalakshmi-144**
* **git remote set-url origin** [**https://ghp\_6XstyER9oIPHcE2eTQ8R6vaOQmSdod1mTmzB@github.com/Rajalakshmi-144/React-web-application-docker.git**](https://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@github.com/Rajalakshmi-144/React-web-application-docker.git) **–** enter the remote token for access

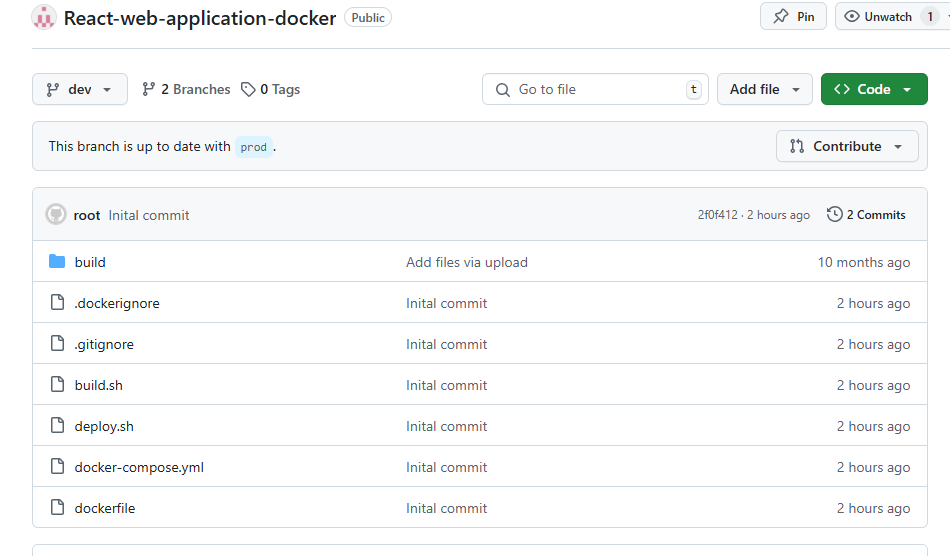
****

* **Git push -u origin dev –** push the dev branch to github repo

****

* In github repo we can see the **main** and **dev** branch is pushed and all the files from local repo is copied to github repo

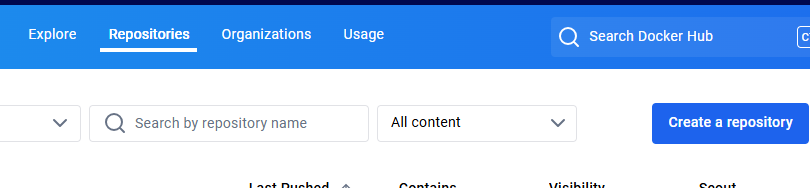
****

****

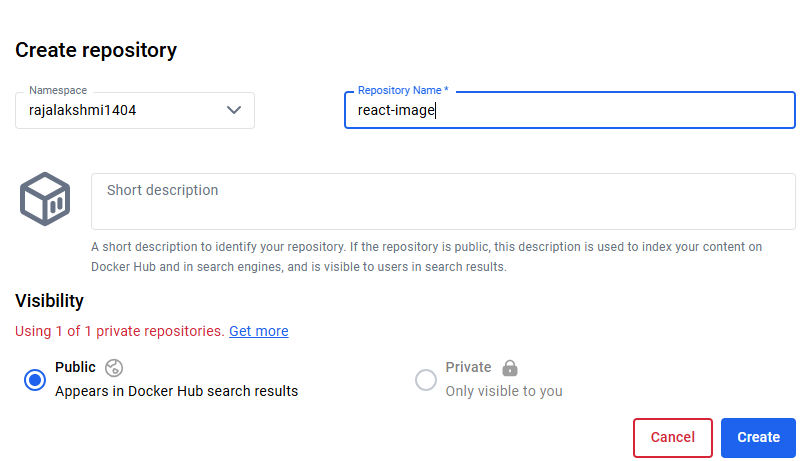
**Step 7:**

Create 2 **docker hub** repositories, one for **dev** repository which is **Public** for development, easier for collaborationand another for **prod** repository which is **Private** for production, ensuring only authorized users can access.

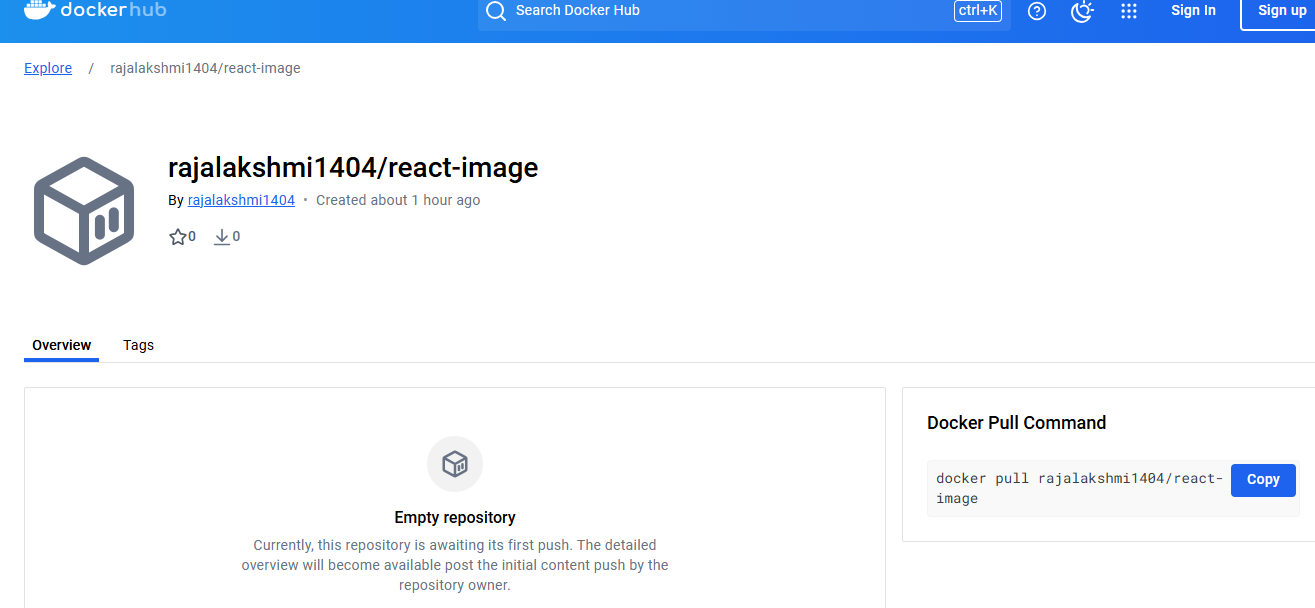
* **Login** to **docker hub** repository, create new repository

****

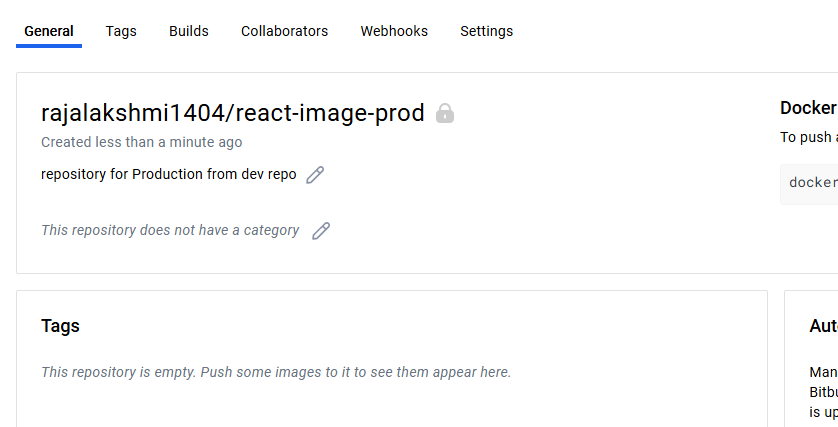
* Name the repository **“react-image”** for the **public** and create.
* Similarly create another repository name the repository as **“react-image-prod**” for the **private** and create

****

* Dev repo **– rajalakshmi1404/react-image**

****

* Prod repo – **rajalakshmi1404/react-image-prod**

****

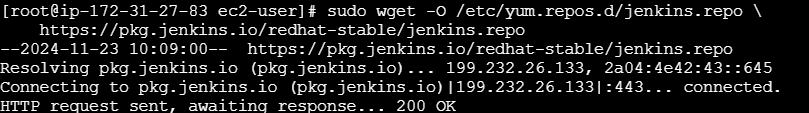
**Step 8:**

Install **java** and **Jenkins, create** **jenkinsfile** to execute the build process such as build, push and deploy the application to the server

* **Sudo yum update -y** – update the server

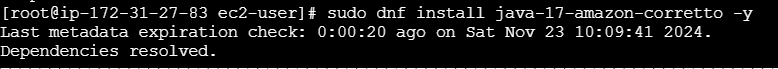
****

* Install the Jenkins repository

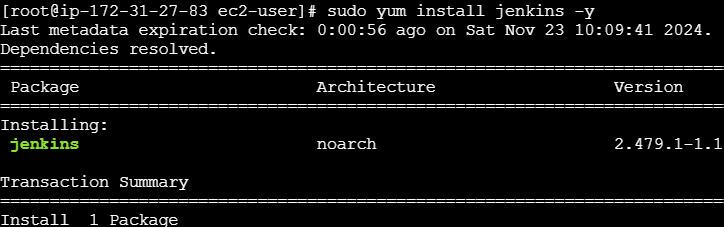
****

****

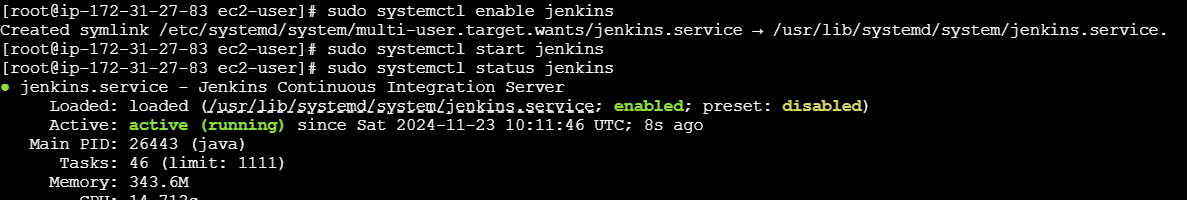
* **Sudo dnf install java-17-amazon-corretto -y —** install java

****

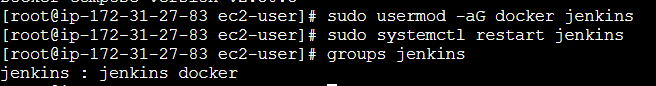
* **Sudo yum install Jenkins -y —** install Jenkins

****

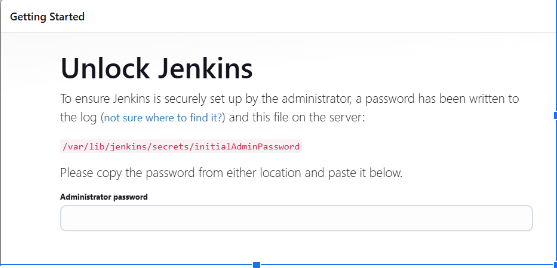
* **Sudo systemctl enable Jenkins –** start and enable the service
* **Sudo systemctl start Jenkins**
* **Sudo systemctl status Jenkins**

****

* **Sudo usermod -aG docker jenkins –** add jenkins user to docker group on linux
* **Sudo systemctl restart jenkins -** restart jenkins
* **Groups jenkins –** jenkins added

****

* Copy the ip address of server with port 8080 and browse **http://3.248.219.82:8080/**

****

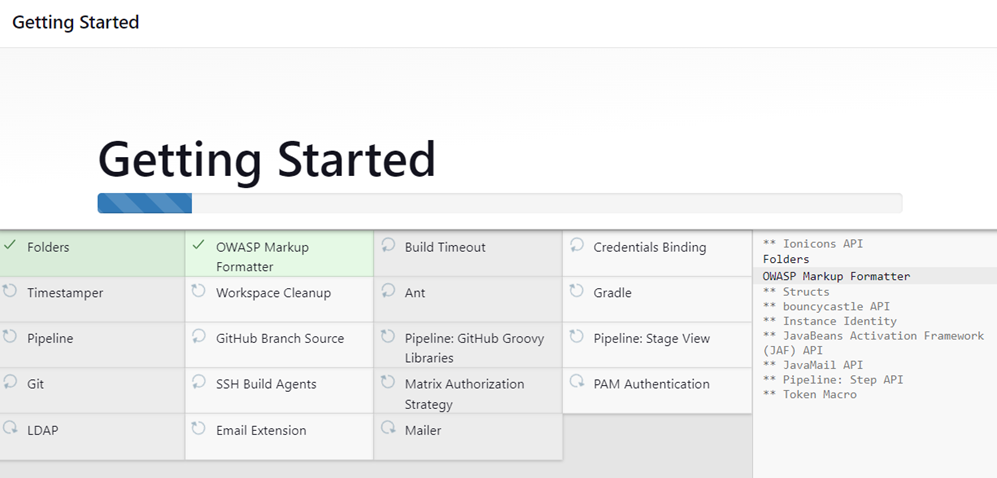
* We need to get the initial admin password by pasting the path on the command line: paste the password on the browser and proceed the next step

****

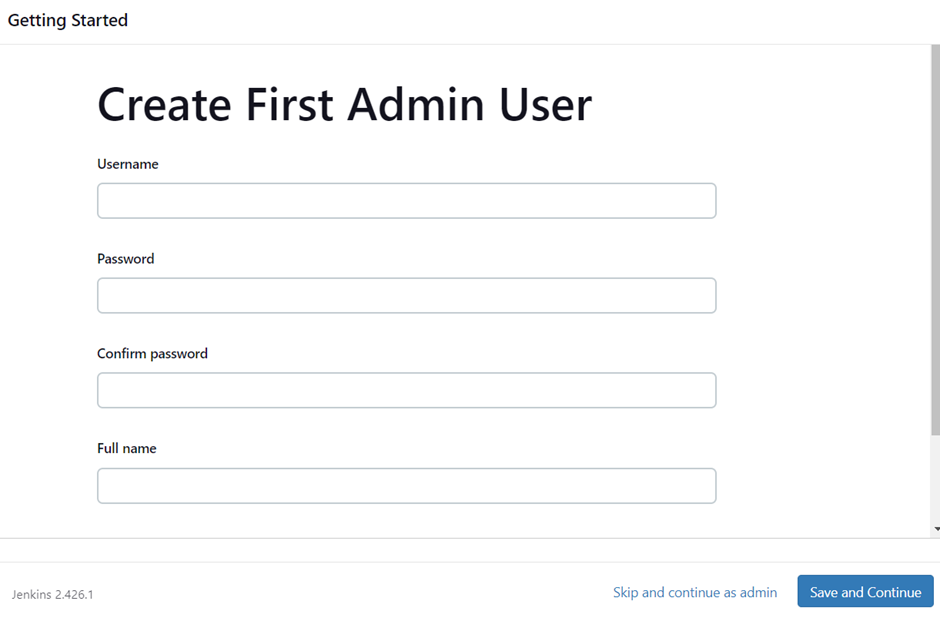
* Then click the install suggested plugins

****

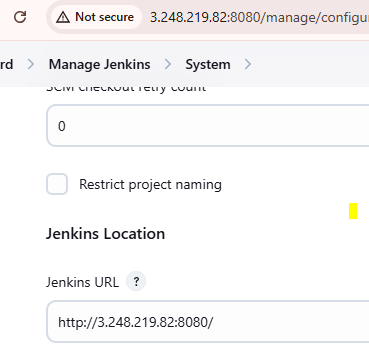
* Plugins will start to install

****

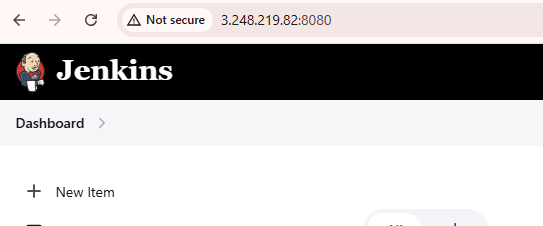
* Create username and password

****

* Then click save and continue
* Add the instance configuration as [**http://3.248.219.82:8080/**](http://3.248.219.82:8080/) and save and finish
* click start using Jenkins



* Jenkins Dashboard

****

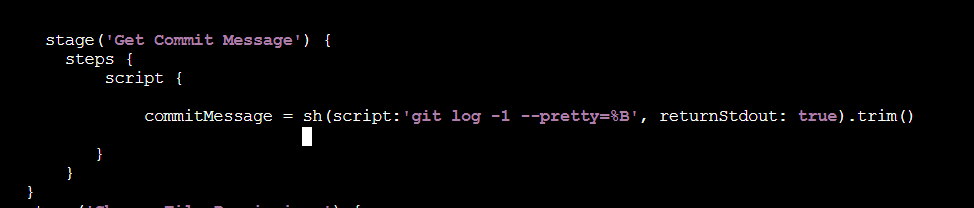
* In server, Create new file name it as **Jenkinsfile** which has the Jenkins build steps

****

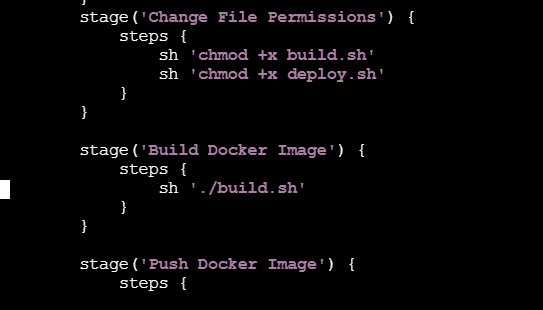
* Using the declarative pipeline , create the stages to execute while build , push, deploy the application
* In the **environment block**, create a variable for **Branch name** and command to get the **git branch name**, create variable for **commit message**.
* In first stages, check the branch name by printing the **current branch**

****

* In next stage, get the **latest commit message** in groovy script, jenkins execute the shell command to **fetch** the most recent commit and format the output to show the **commit message**, also return it as string by removing the whitespace

****

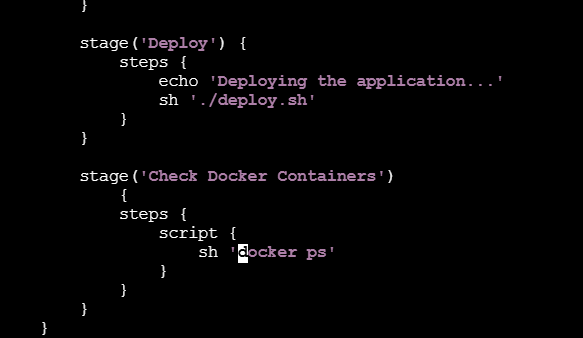
* In next stage, **change the file permission** of build.sh and deploy.sh file
* In next stage, **run** the bash file **build.sh** to build the docker image

****

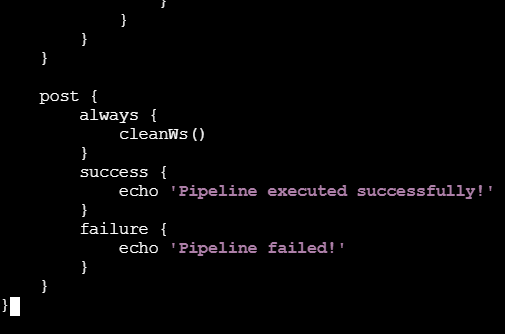
* In next stage, **push the image to docker hub**, by checking the branch name
* If the branch name is **‘origin/dev’**, then **tag** the image name as **“rajalakshmi1404/react-image:dev**” and **push** the docker image to dev (public) repository (i.e.) **“rajalakshmi1404/react-image:dev”**
* If the branch name is “**origin/main**” and the commit message contains the word “**merge**” both in uppercase or lowercase and contains the word “**dev**”, then **tag** the image name as **“rajalakshmi1404/react-image-prod:prod”**and **Push** the image to prod repository (i.e.) **“rajalakshmi1404/react-image-prod:prod”**
* Else **skip the push**

****

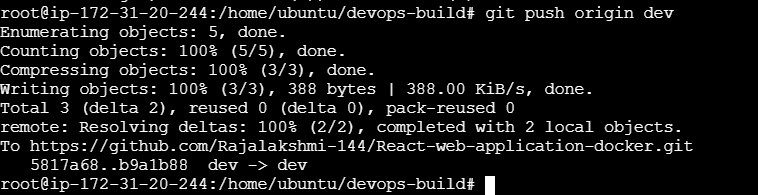
* In next stage, deploy the application by **run** the **deploy.sh** file
* In next stage, check the **docker running** container



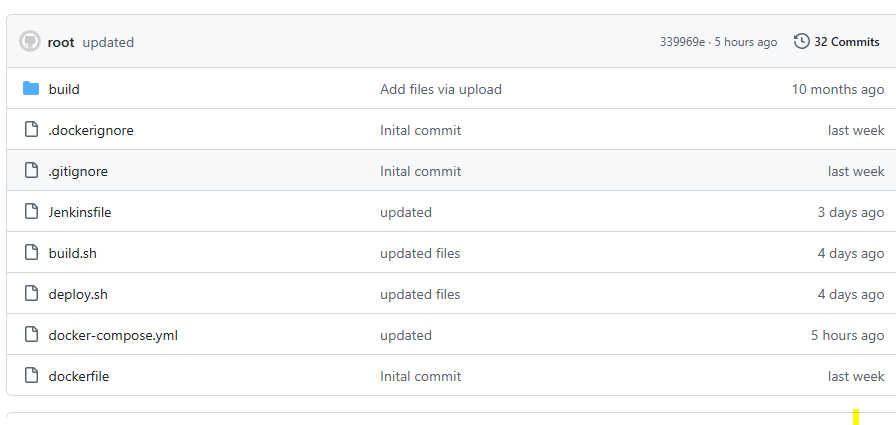
* In **post** block, ensure the **workspace** is **cleaned up** at the end of the pipeline execution
* And print the **success** message or **failure** message at the end of the pipeline execution

****

* **Git add .**
* **git commit -m “ add Jenkinsfile”**  — add the jenkinsfile
* **Git push origin dev** – push the jenkinsfile to github repo

****

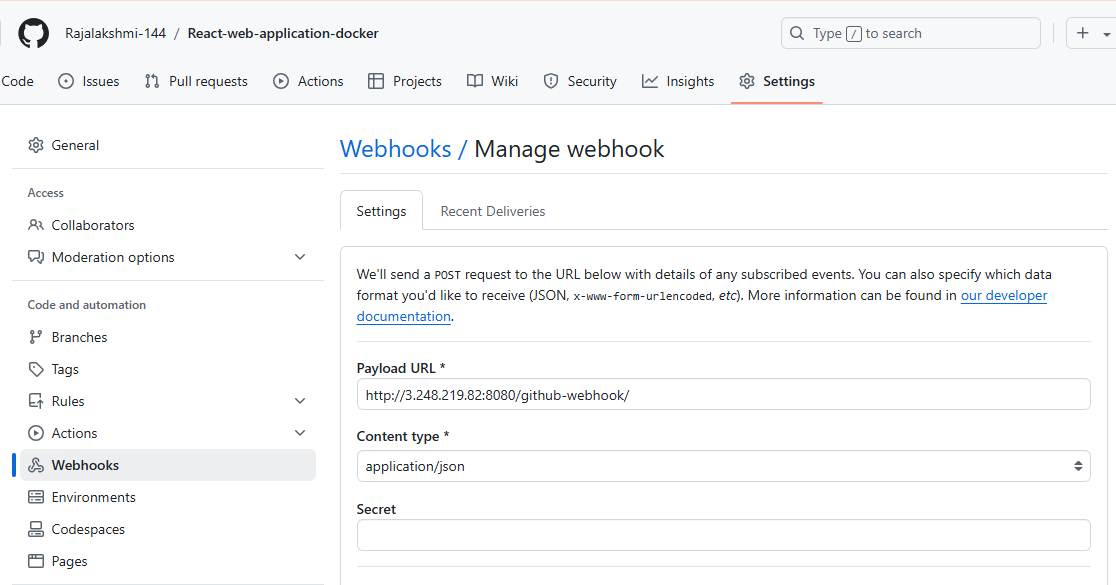
* Jenkinsfile added to github

****

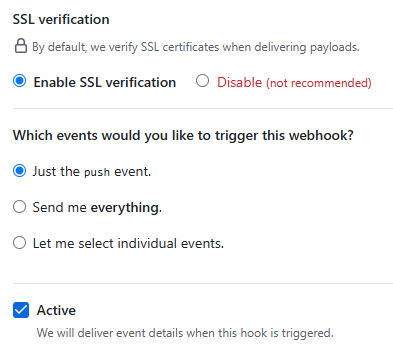
**Step 9:**

Connect **github** to **Jenkins** and configure **auto-triggers** for both main and dev branches.

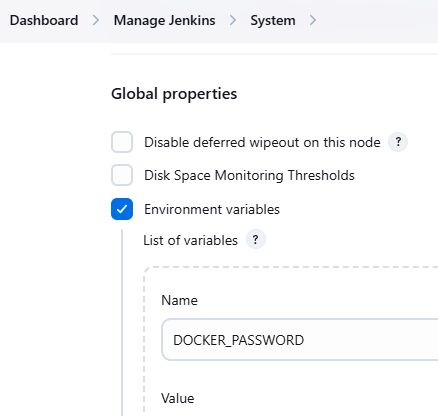
* In github,  **React-web-application-docker** repository go to **settings** select **web hook, Add Web hook**
* Add the payload as **“**[**http://3.248.219.82:8080/github-webhook/**](http://3.248.219.82:8080/github-webhook/)**”** and content type as “**application**/**json**”

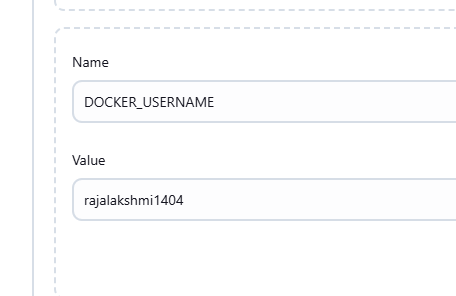
****

* **Save the web hook**

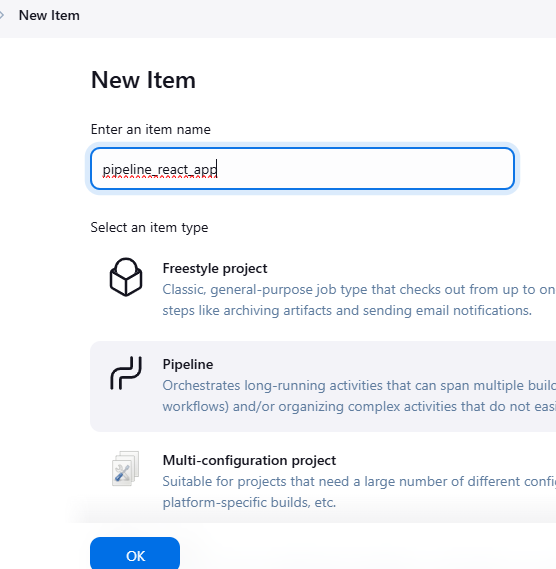
****

* In Jenkins , go to **manage Jenkins** → **system** , in global properties add Docker username and docker password

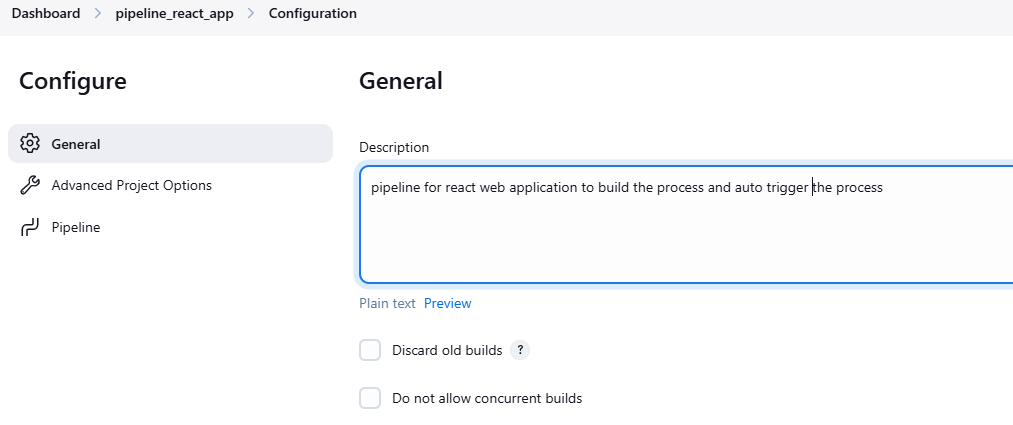
****

****

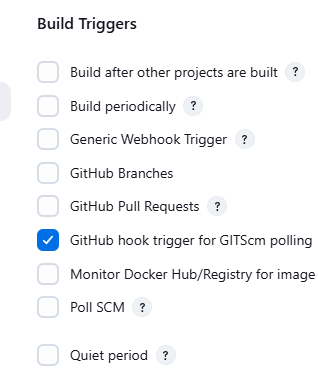
* Create a new job in the Jenkins pipeline name as “**pipeline\_react\_app”** and select **pipeline** create the job



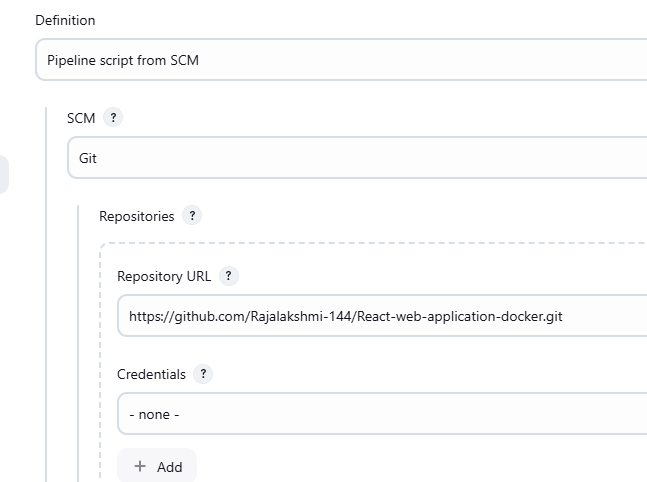
* Describe the pipeline

****

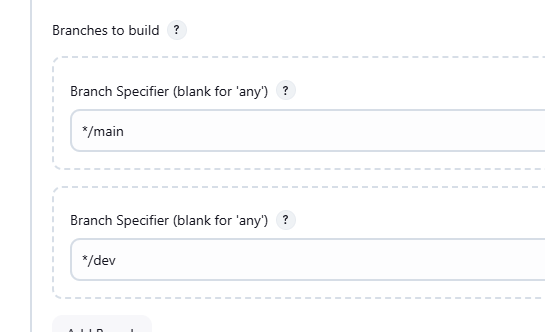
* select the **github hook trigger for GITScm polling**

****

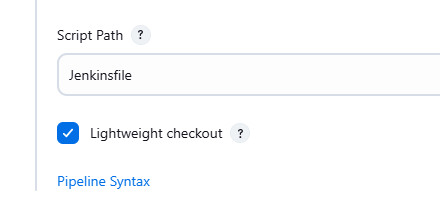
* Select **pipeline script from SCM**  and enter git repository URL

****

* Specify both **/\*main** and **\*dev branch**

****

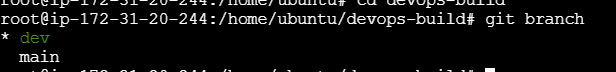
* Mention the script path as “**Jenkinsfile**”. Save and apply

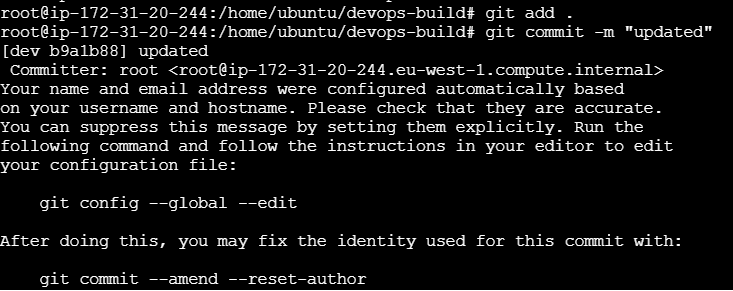
****

**Step 10:**

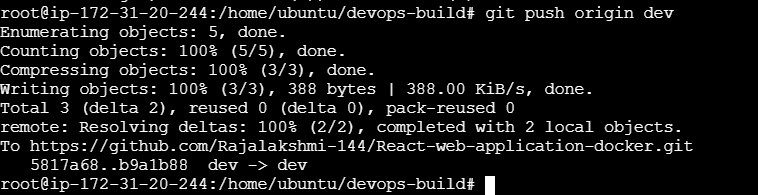
When code is pushed to the **dev** branch on GitHub, Jenkins will automatically build a Docker image and push it to the **dev** repository in Docker Hub.

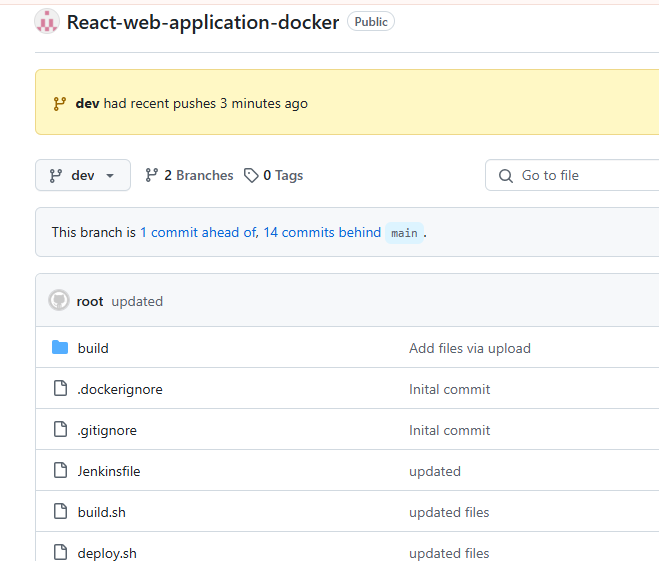
* If the **code** is **added** or **updated** in **dev** branch



****

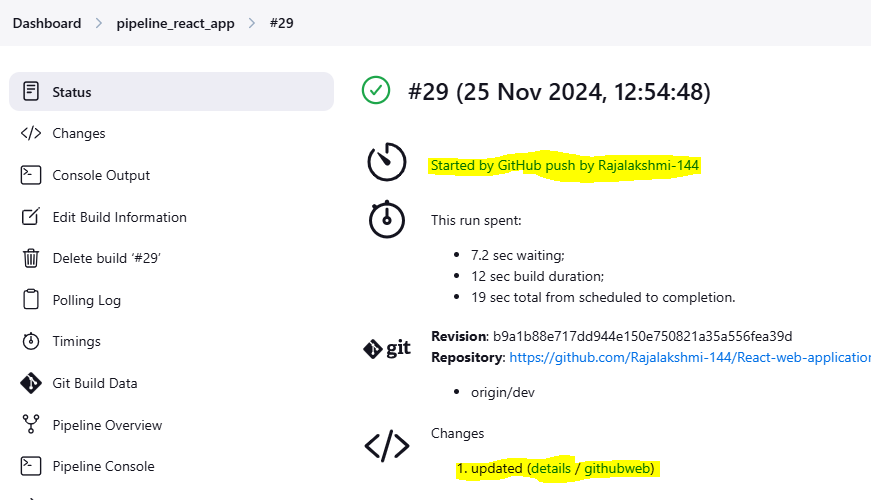
* Pushed dev branch to github

****

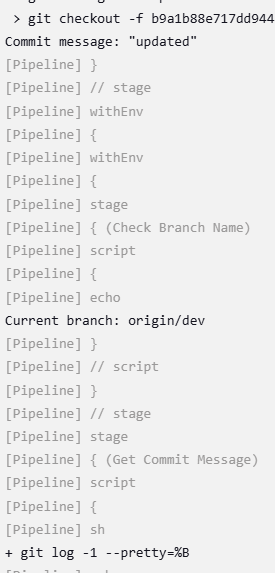
****

* It is **started** through github **user (Rajalakshmi 144)**

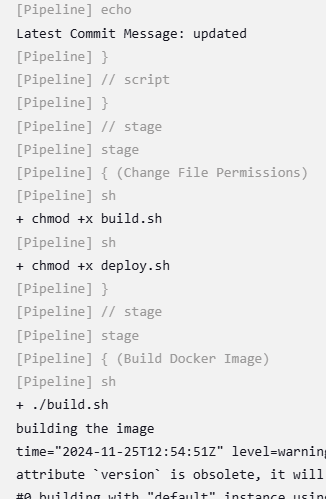
 – Github username

****

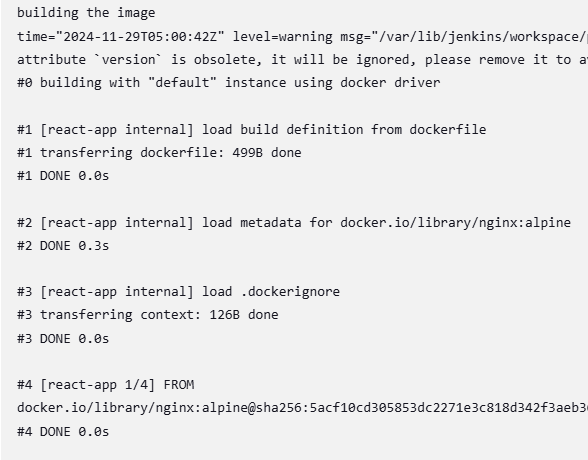
* In Jenkins, at **console outpu**t we can see the step by stage of pipeline execution
* Commit message is “**updated**”, current branch is “**origin**/**dev**”

****

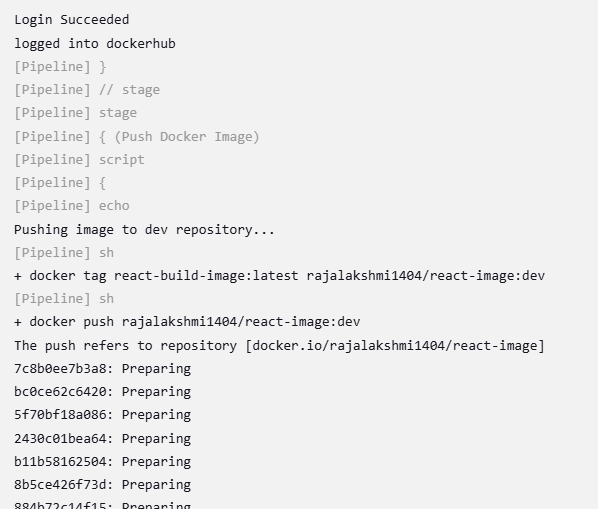
* Latest commit message is printed, change the file permission and run the build file

****

* Building the image



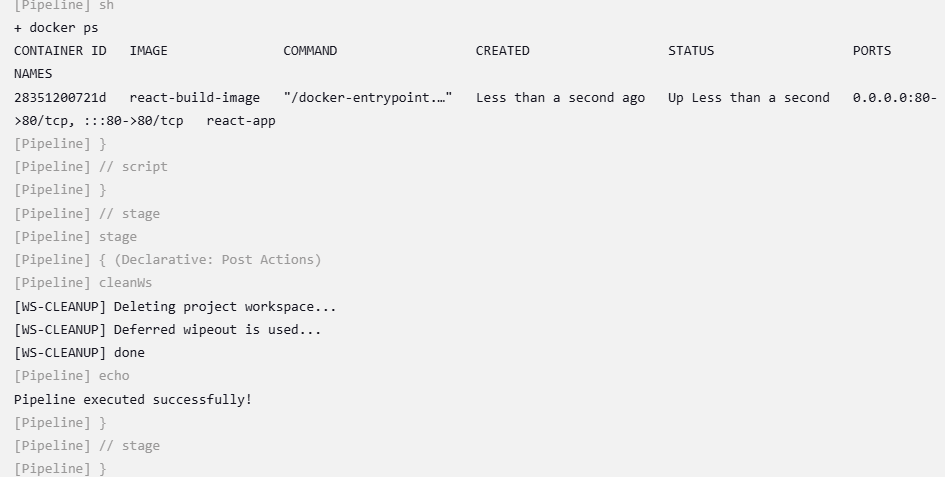
* logged into the docker hub, since it is dev branch the image is tagged as “**rajalakshmi1404/react-image:dev**” and pushed to the docker hub dev repo **“rajalakshmi1404/react-image:dev**”



* Deploying the application , created the container and started the container

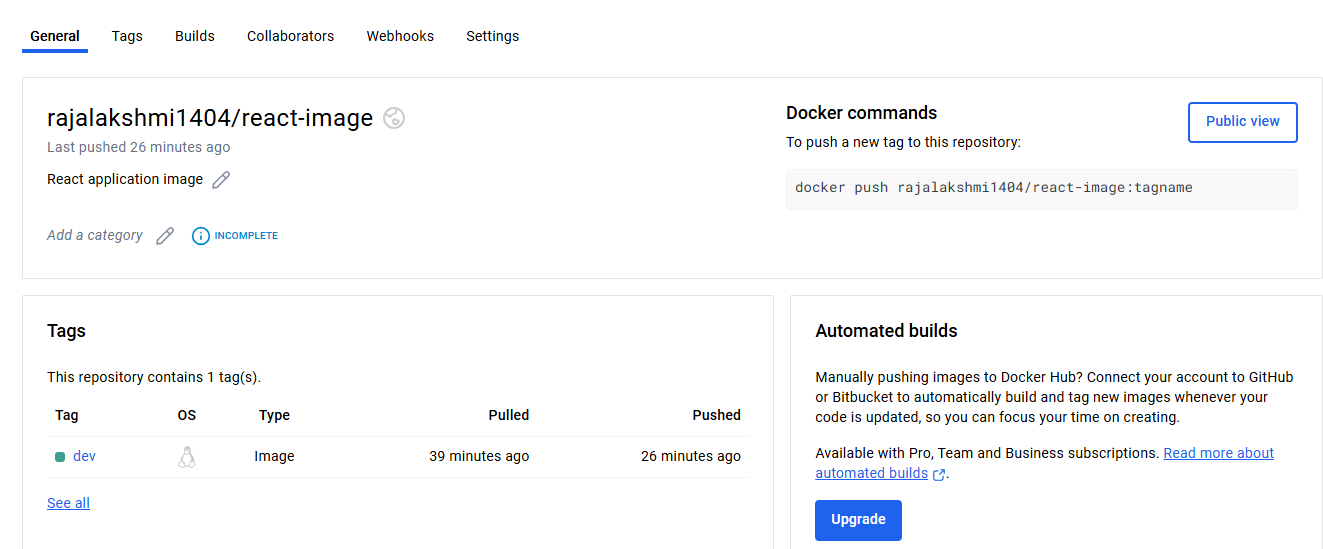


* Container is running at **80** ports, the name of the container is **“react-app**”.
* Cleaning the pipeline workspace and the pipeline is **success**

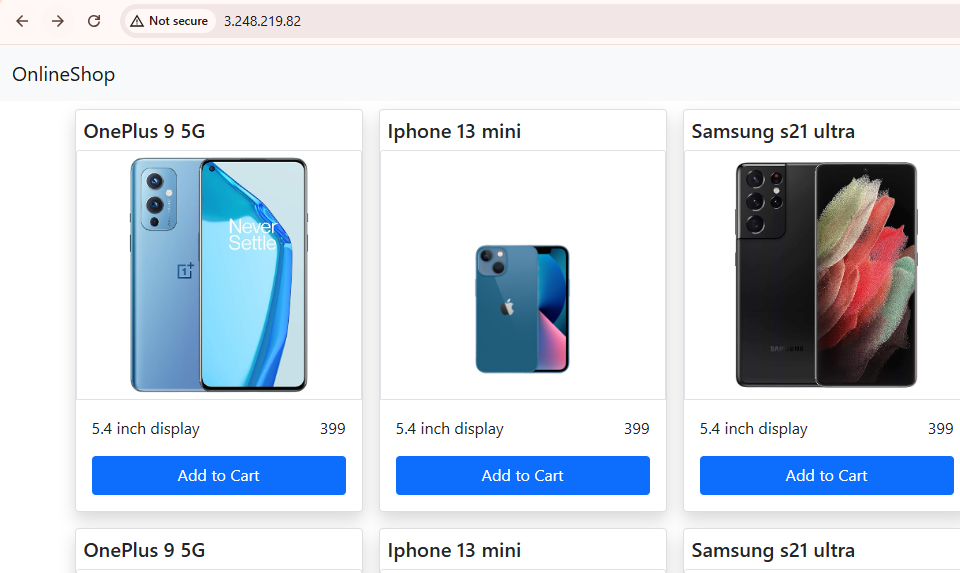




* **rajalakshmi1404/react-image** (public) repo image are stored here

****

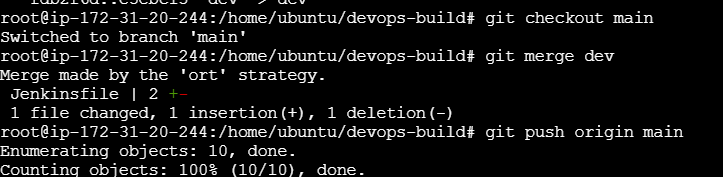
* Application is deployed and the **browser output** at **http://3.248.219.82:80/**

****

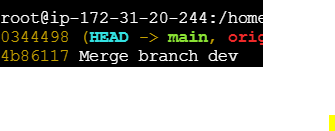
**Step 11:**

When code is merged into the **main** branch on github, Jenkins will build the Docker image again and push it to the **prod** repository in Docker Hub.

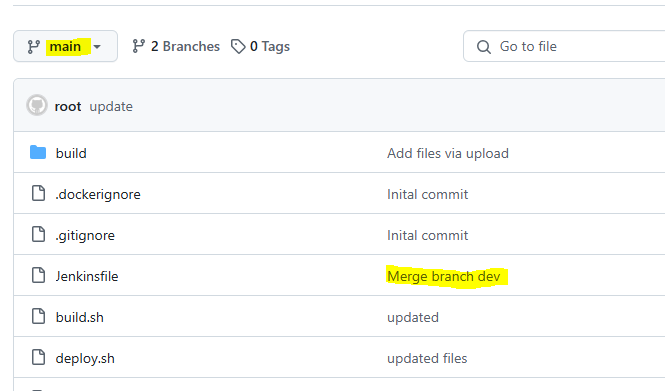
* In main branch, **git merge dev** – to merge dev into main

****

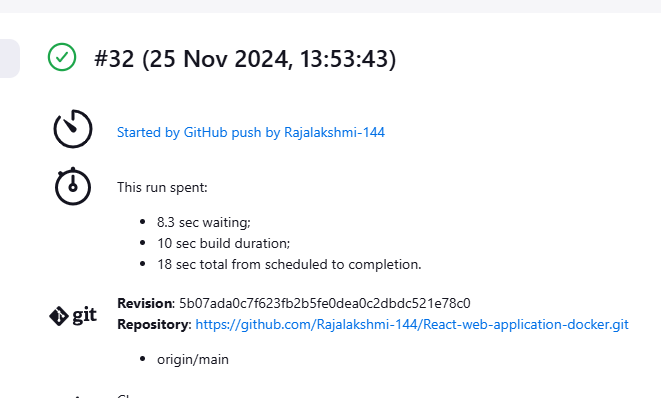
* And commit messageis **“Merge branch dev”**



* In github, Files are merged in main branch and pushed to github repo

****

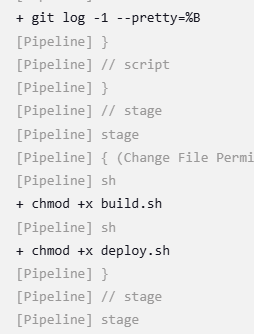
* In Jenkins, build **started** by github in **origin/main** branch

****

* Commit message is **“Merge branch dev”** and current branch as “**origin/main**”

****

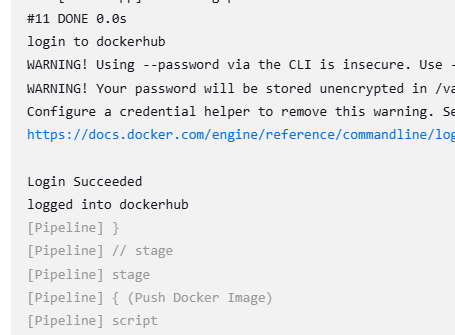
* Change the file permission of build.sh and deploy.sh

****

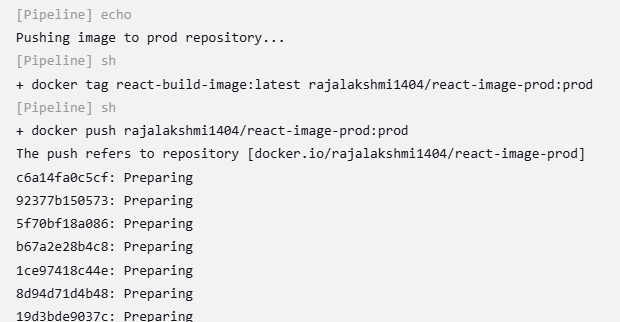
* Building the image

****

* Logged into docker hub

****

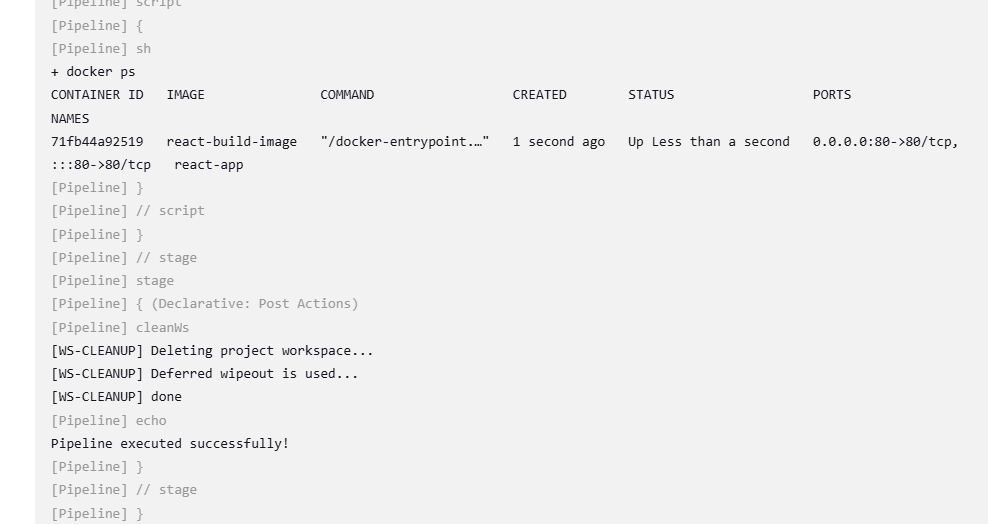
* since it is main branch, the image is tagged as **“rajalakshmi1404/react-image-prod:prod”** and pushed to the docker hub prod repo **“rajalakshmi1404/react-image-prod:prod”**



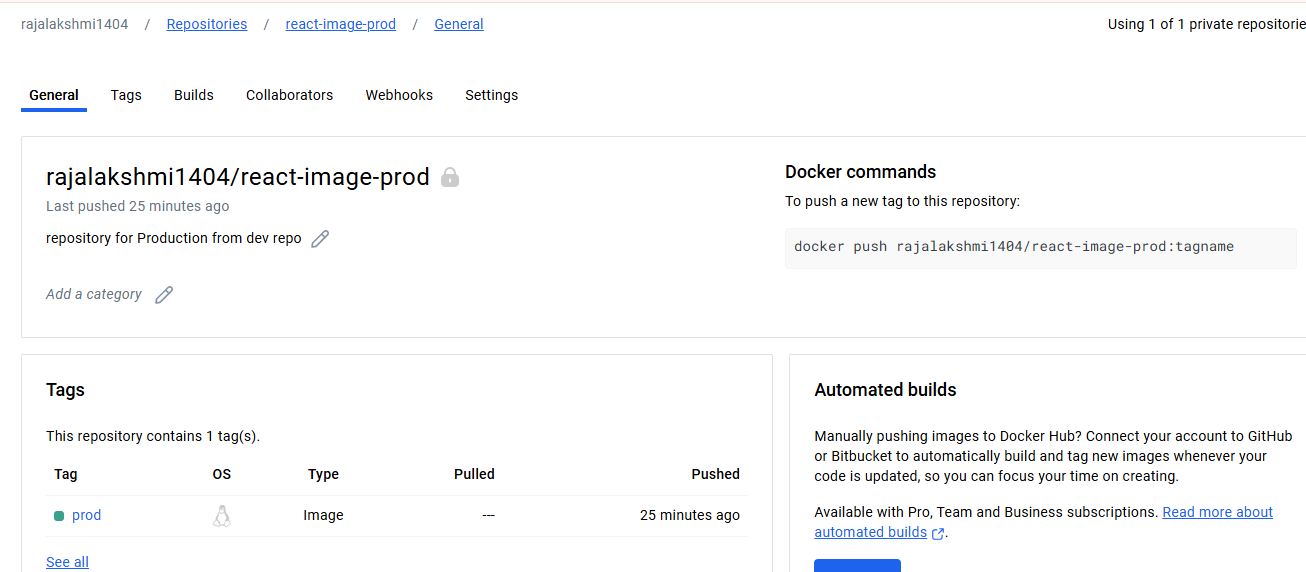
* Deployed the application

****

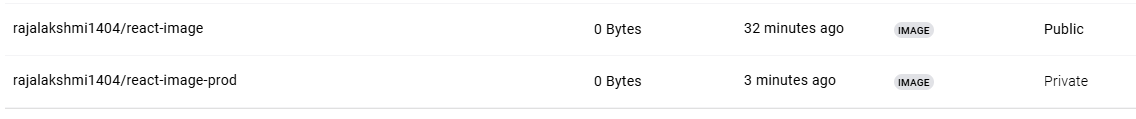
* Container is created and start the container, it is running in port **80**
* Clean the pipeline workspace and the pipeline is **success**

****

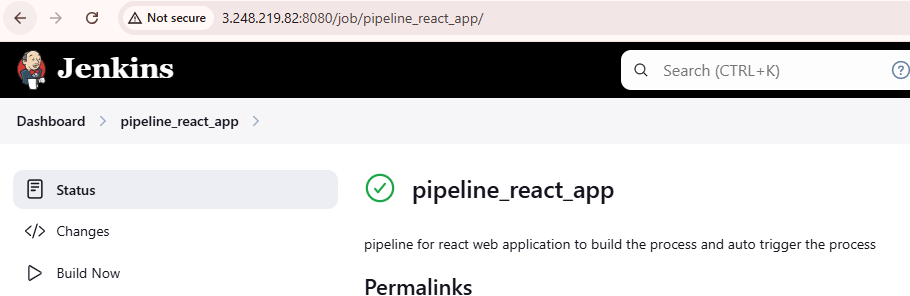
* **rajalakshmi1404/react-image-prod** repo (private) image are stored here

****

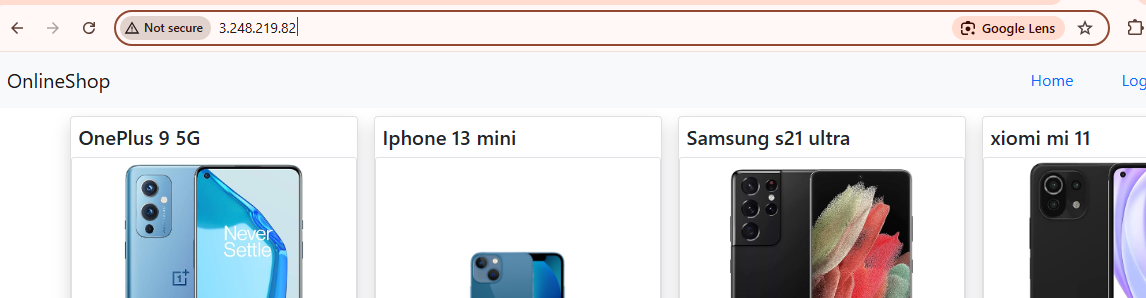
* Image is stored in docker hub, in rajalakshmi1404/react-image (public) repo and rajalakshmi1404/react-image-prod repo (private)

****

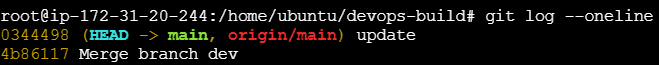
* Check the ip address of the server in browser to check whether the application is deployed

****

* **Browser output** [**http://3.248.219.82**](http://3.248.219.82/)**:80/**

****

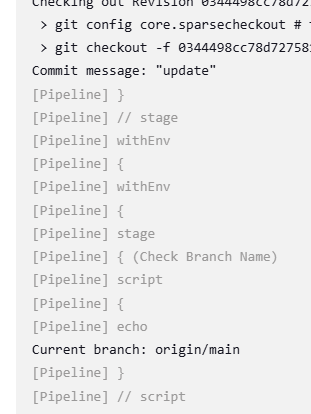
* If we update the main branch and push to github



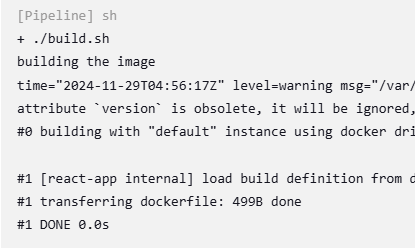
* It start to execute the pipeline in Jenkins



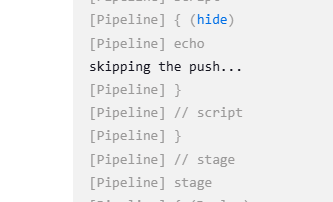
* Commit message is “update” and current branch is “origin/main”



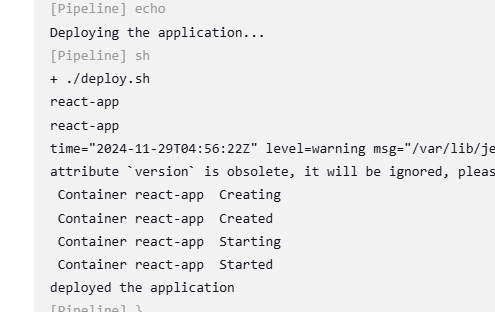
* Building the image

****

* **Skipped the pushing**, since it show the commit message as “**update**” in **origin**/**main** branch



* Deployed the application



* Clean the workspace and the pipeline is successful

****

**Step 12:**

Install **Prometheus** and **node-exporter**, update the **prometheus.yml** file to scrape the metrics exposed by the react application. Create **Prometheus.service** file which manage the Prometheus systemd service, it can be configured to start Prometheus automatically when the system boots or on failure

* download node exporter and export the files

****

****

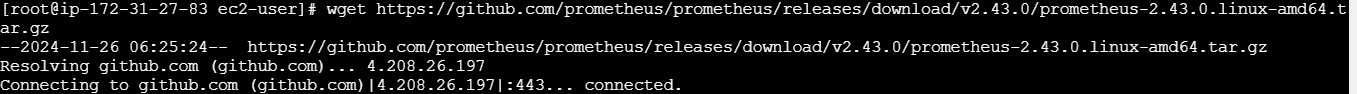
* Move the folder to local binary

****

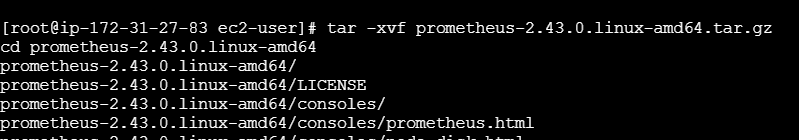
* Start and enable the node exporter

****

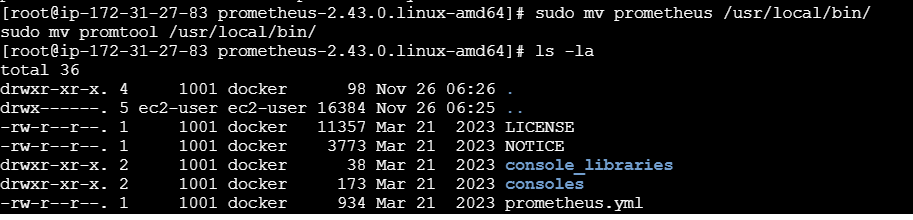
* Download Prometheus

****

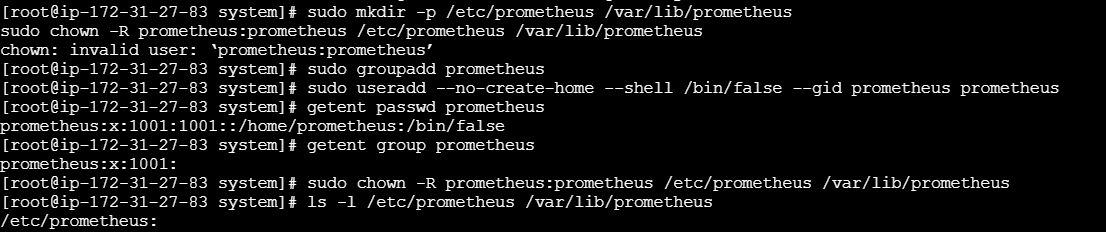
* Export the files and folders

****

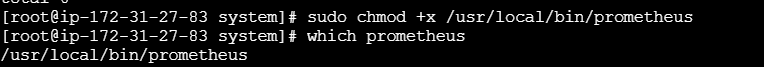
* Move the files and folders to local binary

****

* Make directory in /etc/prometheus and /var/lib/prometheus
* Add the group prometheus in Linux
* Give the file ownership permission to both /etc/prometheus and /var/lib/prometheus

****

* Also give file permission to prometheus in local binary

****

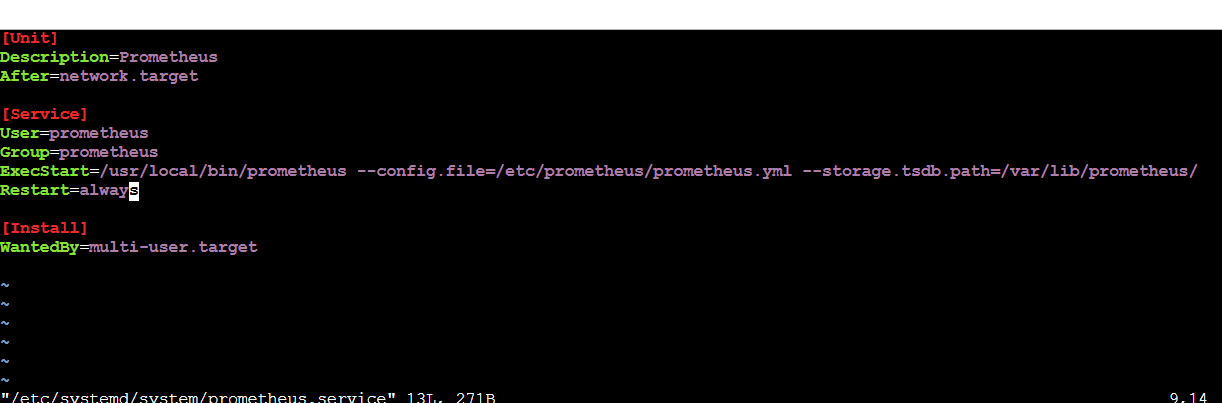
* Edit the prometheus.yml config file
* sudo mv prometheus.yml /etc/prometheus/ – move the file to /etc/prometheus

****

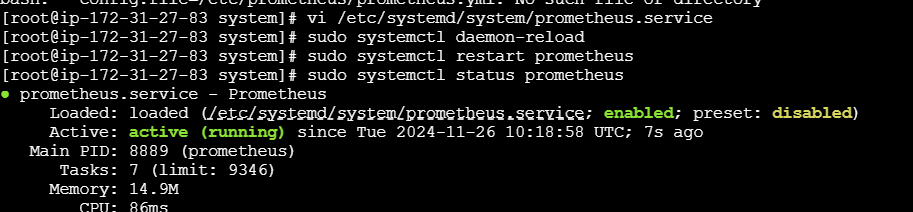
* **prometheus.yml** contains
* Add “node” for node exporter and “react-app-health” for react application as job name,
* Their target ip address and port, node exporter run at port **9100** and react application port is **80**
* mention the react application metrics path, which is **/health** to check the application status

****

* Create prometheus.service file in /etc/systemd/system/prometheus.service

****

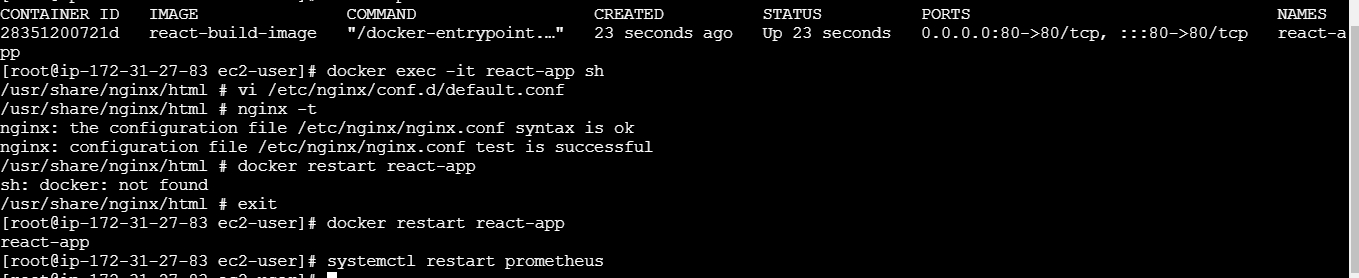
* reload the systemd manager configuration and start ,enable the prometheus

****

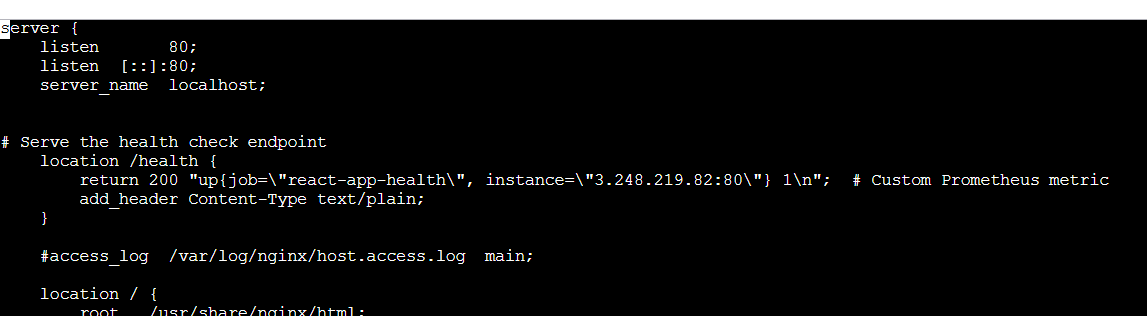
**Step 13:**

Update the **nginx default conf** configuration file to serve the health check endpoint of react application and make sure Prometheus can check the status

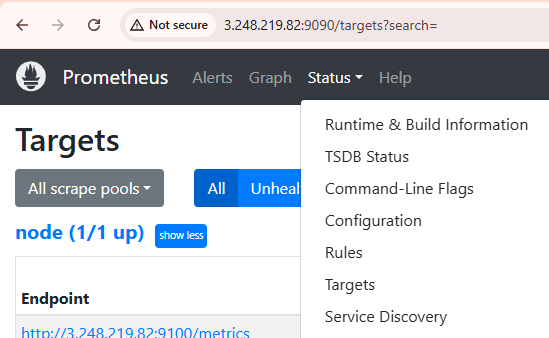
* **Docker exec -it react-app sh** – used to execute a shell inside a running Docker container named react-app.
* **Vi /etc/nginx/conf.d/default.conf**  — Edit the default.conf file to serve the health endpoint in react application
* **Nginx -t –** verify the syntax and test the config file
* **Exit -** from container
* **Docker restart react-app** - restart docker
* **Systemctl restart prometheus -** restart prometheus

****

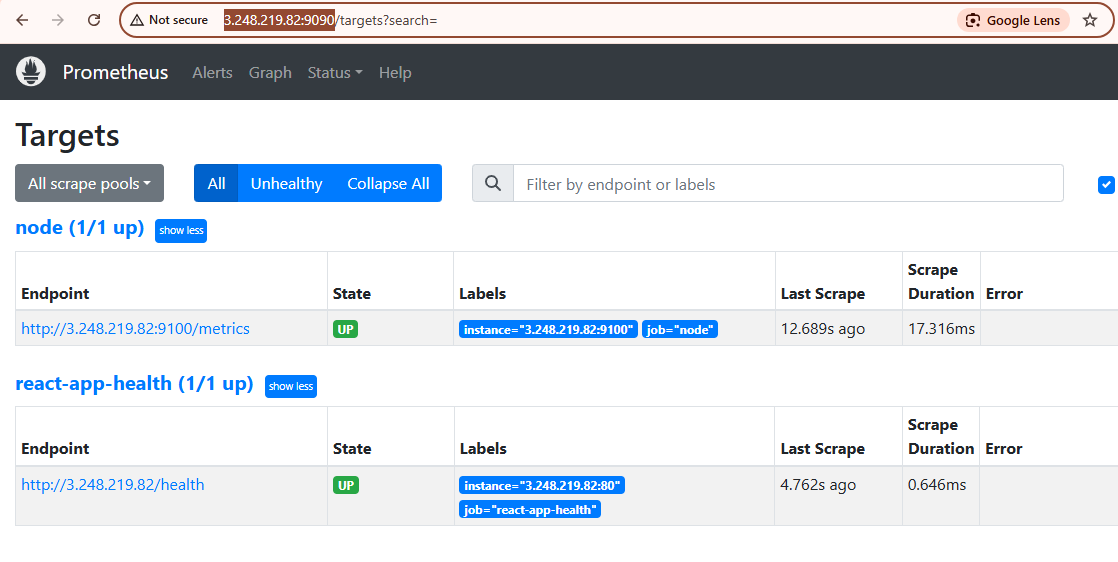
* **Default.conf** file contains,the **/health endpoint** in the React application which serves a health status in the Prometheus metric format.
* When Prometheus scrapes this endpoint, it queries for the up metric, which indicates whether the job is running correctly.
* **up {job=\”react-app-health\” , instance=\”**[**http://3.248.219.82:80\**](about:blank)**”} 1\n”** – indicating the name of the monitored job and the instance where the react application is hosted is up and healthy.
* **1** - indicate the application is up and healthy ; **0** - indicate the application is down
* Prometheus uses these values to monitor the health of your application and trigger alerts when necessary.

****

* Browse ip address at port 9090 for prometheus UI –[**http://3.248.219.82:9090/**](http://3.248.219.82:9090/)
* Go to **status** → **targets**

****

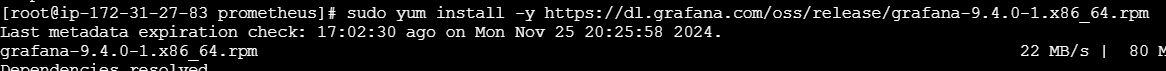
* **Node -**  endpoint indicates the node exporter metrics, node exporter scrape the metrics at 9100 port. Browse ip address at port 9100 with /metrics endpoint ([**http://3.248.219.82:9100/metrics**](http://3.248.219.82:9100/metrics)) to see the metrics scarped at the instance by node exporter. It state is **up**
* **React-app-health -** endpoint indicates the health endpoint of the react application, browse the ip address of the react application with /health endpoint ([**http://3.248.219.82/health**](http://3.248.219.82/health)) to see the health of the react application. It state is **up**

****

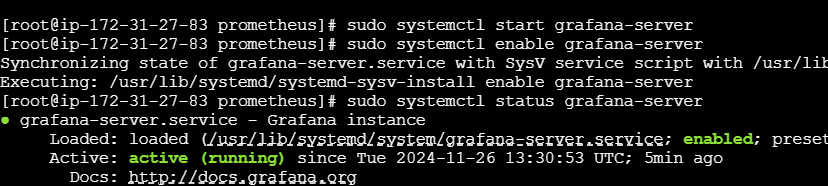
**Step 14:**

Install **grafana** and configure **Prometheus data source, query** to fetch necessary metrics to be displayed in the dashboard of react application for real-time monitoring.

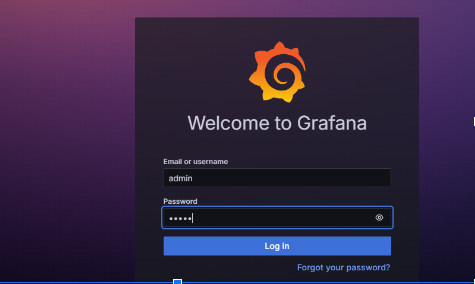
* Install **grafana** , to visualize the metrics scraped by prometheus in the grafana dashboard

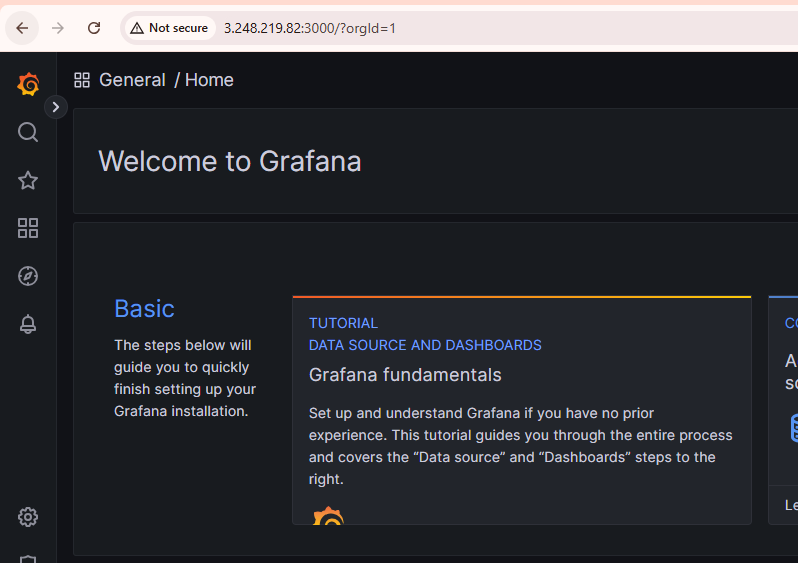
****

* Start and enable grafana-server

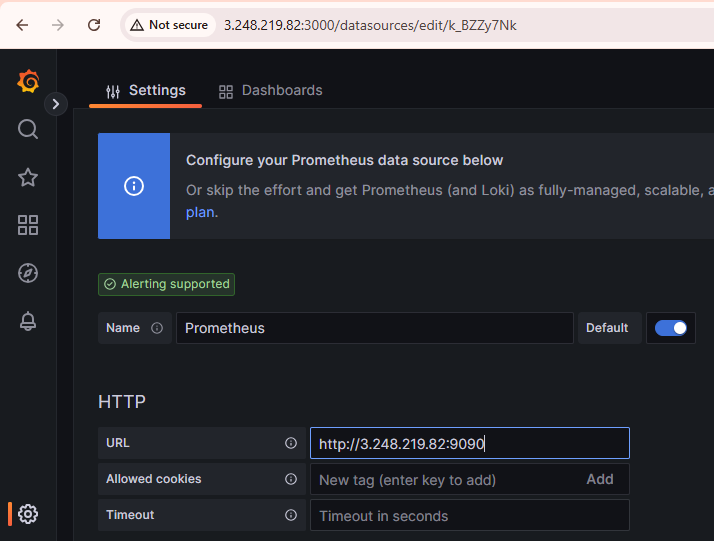
****

* Open your browser and go to [http://](http://3.25.86.167:3000)[3.248.219.82](http://3.248.219.82/health)[:3000](http://3.25.86.167:3000) for grafana UI
* Username and password as “**admin**” for grafana

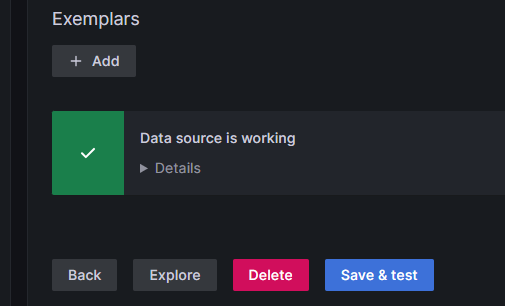




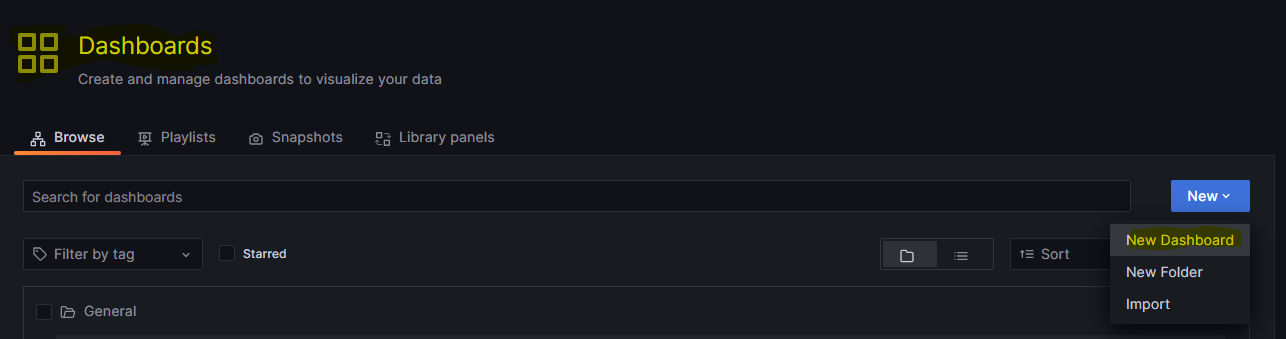
* In the Grafana dashboard, go to "**Connection**” **→ "Data Sources**"
* Click **"Add data source**"
* Select "**Prometheus**"
* Set the URL to [**http://**](http://3.25.86.167:3000)[**3.248.219.82**](http://3.248.219.82/health)[**:**](http://3.25.86.167:3000)**9090**

****

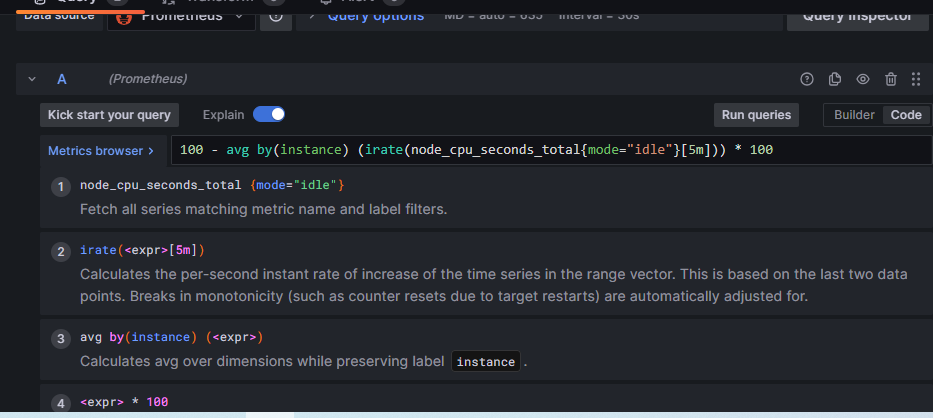
* Click "**Save & Test**" to verify the connection

****

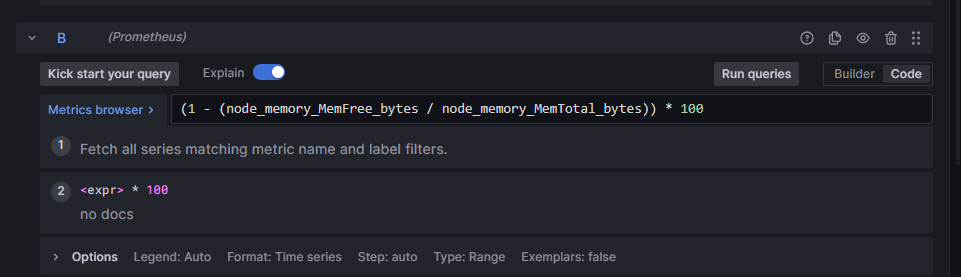
* Create new dashboard

****

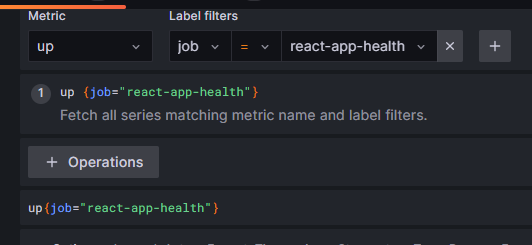
* Edit the panel, In the "**Query**" section, select  **Prometheus data source**
* Enter a query metrics as **node\_cpu\_seconds\_total** used for monitoring system-level CPU performance. here it tracks the total time in seconds at idle mode



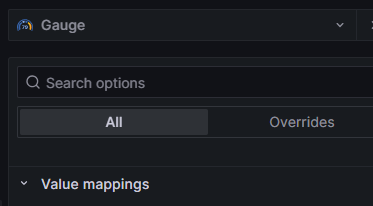
* Enter **node\_memory** query metrics to calculate the percentage of memory usage on a system

****

* Enter the **up** query metrics to check the status of the react-app-health job that is being monitored by Prometheus. The query will return “1” or “0”
* **1,**  if the React app health check endpoint (/health) is reachable and functioning correctly
* **0** if Prometheus is unable to scrape the health check endpoint or if the app is down

****

* Customize the visualization type as **gauge**

****

* Name the dashboard as **“instance\_react\_app\_status**” and save
* We can visualize the cpu performance, memory usage and react-app status in the dashboard. Now the health of the react application is up and healthy

****

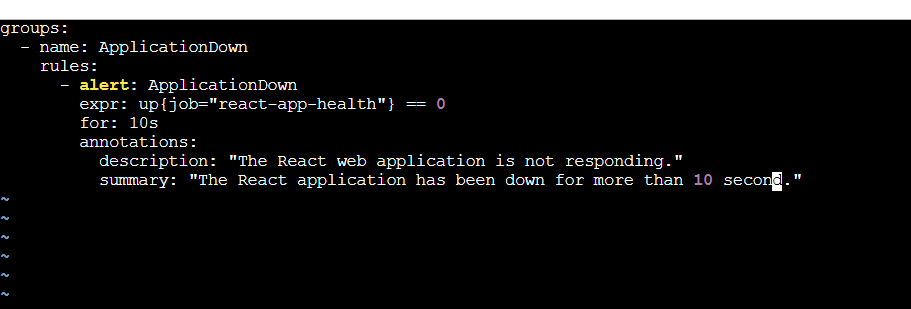
**Step 15:**

Create the **alert.rules** file in Prometheus to define **alerting rules** that allowing you to proactively detect and trigger notifications when certain conditions or thresholds are met in the monitored data

* Create **alert.rules** file in prometheus folder



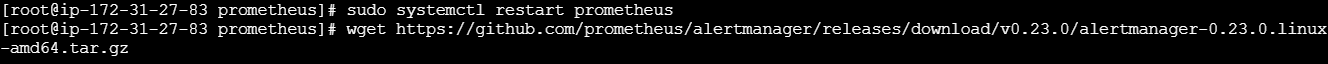
* Define the alert name as “**ApplicationDown**” and triggering condition like if the **up{job=”react-app-health”} ==0** (down condition) for 10 second, trigger the alert notification
* add the annotation to describe about the alert to be notified



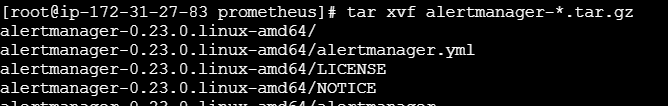
**Step 16:**

Install **alert manager**, update **alertmanager.yml** configuration file that defines how alerts are grouped, routed, and sent to different notification channels like email. Create **alertmanager.service** file which manages alert manager systemd service.

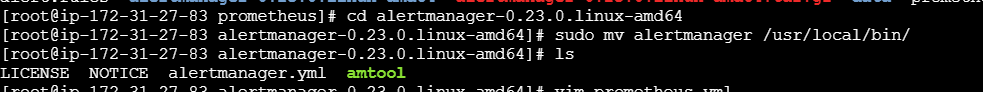
* Download alert manager

****

* Extract the files and folder



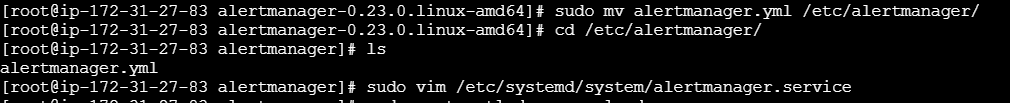
* Move the alter manager folder to local binary

****

* Create a directory prometheus in /etc/



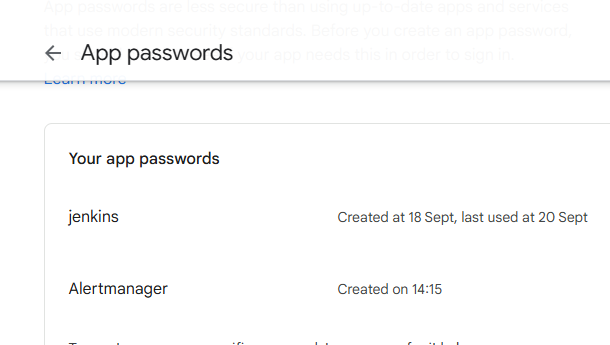
* Move the alertmanager.yml file in /etc/alertmanager
* Create alertmanager.service file in /etc/systemd/system

****

* **Alertmanager.system** file contains

****

* Create app password in Gmail for alert manager



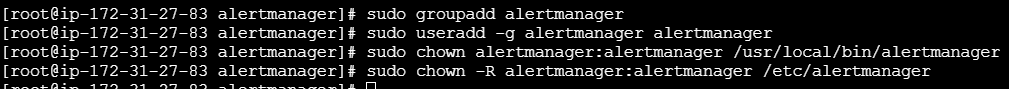
* Edit the alertmanager.yml file

****

* **alertmanager.yml** contains the routing logic for alerts that determines which alerts go to which receiver.
* Receiver defines the notification channels such as email that alerts are sent to the recipient email address which is specified in the receiver email configuration.
* Also Alert manager should notify when an alert is resolved. This ensures that the team is informed when the issue has been fixed

****

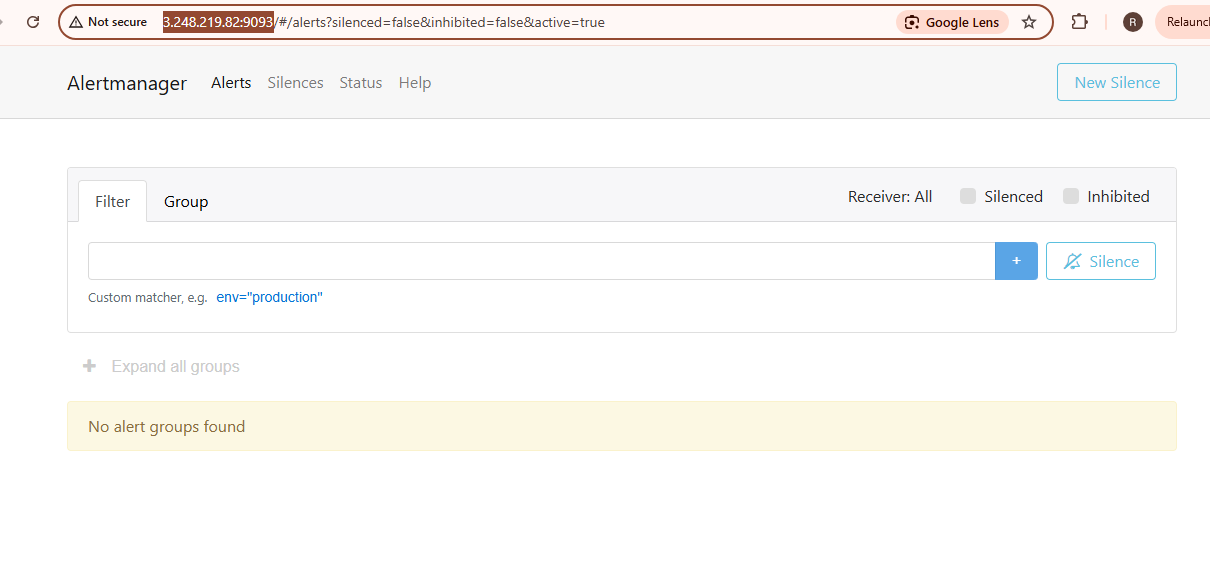
* Add alert manager user and group
* Give the file ownership permission to local binary alter manager and /etc/alertmanager

****

* reload the systemd manager configuration and start ,enable the alertmanager and check the status

****

* browse ip Address with port 3000 **(**[**http://3.248.219.82:9093/**](http://3.248.219.82:9093/)**)** for **alertmanager UI**

****

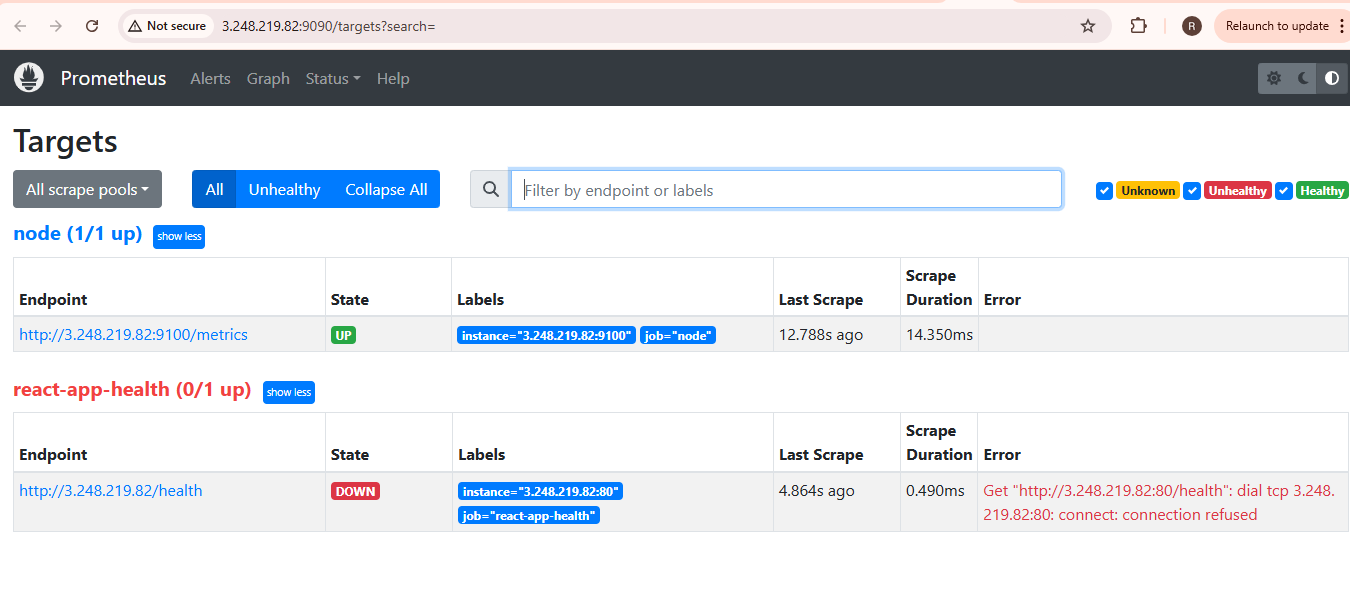
**Step 17:**

Check if the alert is triggered and the **notification** is sent to the recipient email, when the react application is **down**

* **Docker ps –**  currently the react-app is up and running
* **Docker stop <container\_id> -** stopping the running container
* **Docker ps –** now the application is down

****

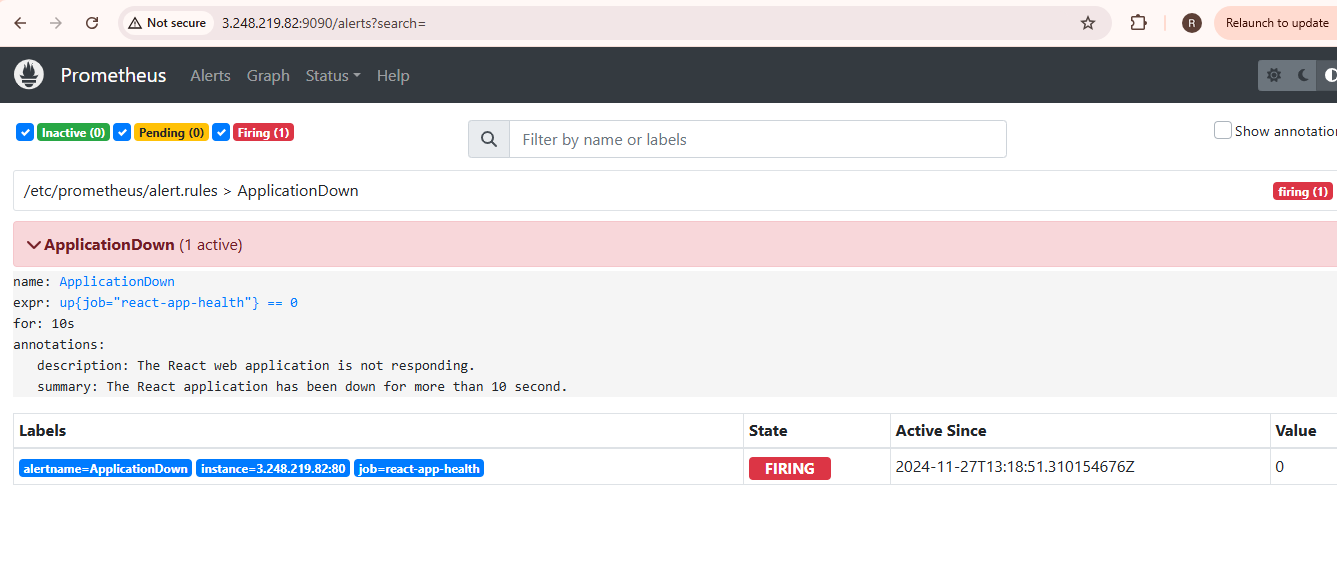
* In prometheus UI, the **react-app-health endpoin**t state is **down**

****

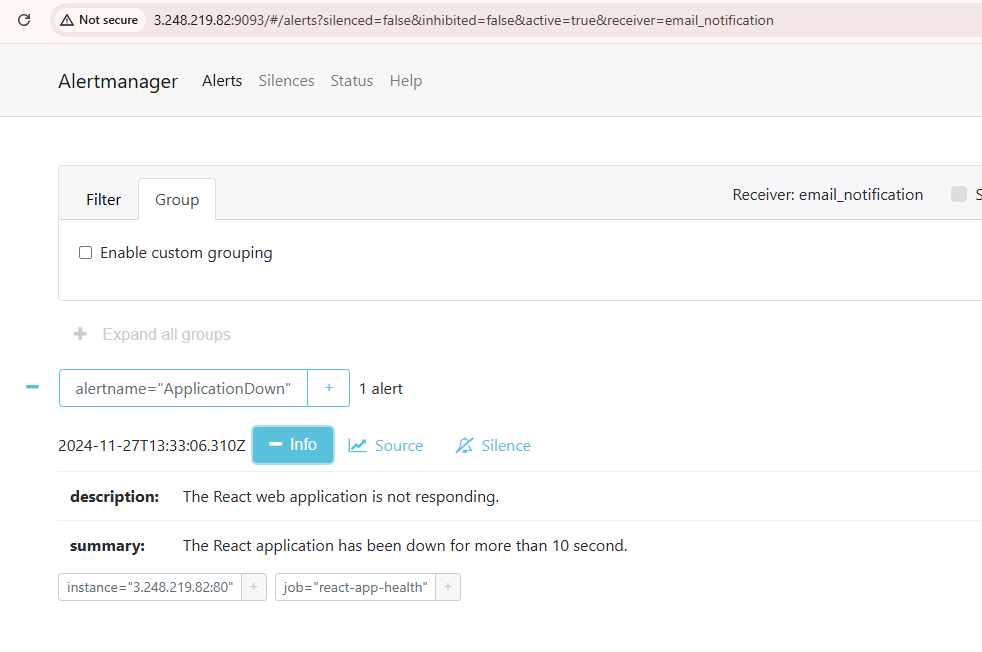
* In grafana dashboard , the job **react-app-health** metrics is **0** which means react application is **down**

****

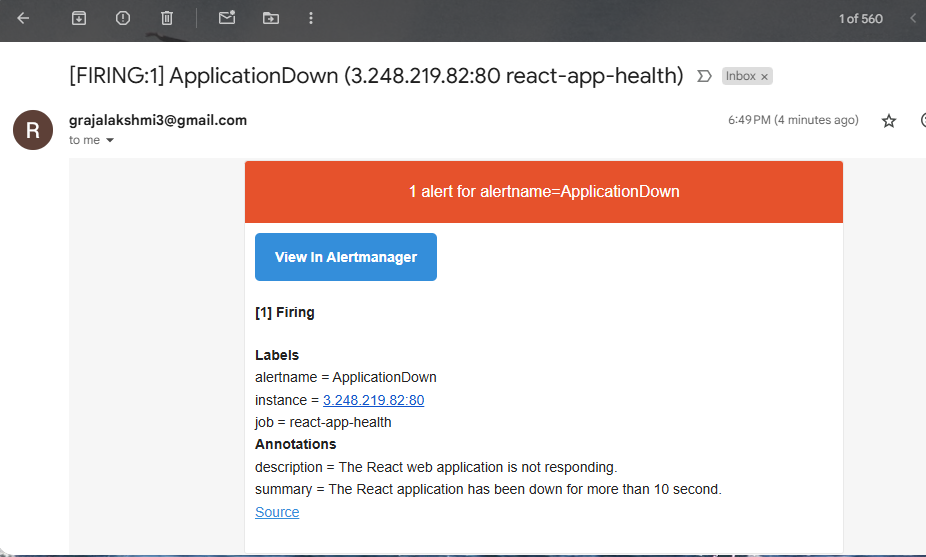
* In prometheus alert, “**ApplicationDown**” alert name is active which mean the react application is **down** and state is **firing**

****

* In alertmanager , the alert name **Application Down** is **triggered** and display the respective alert description

****

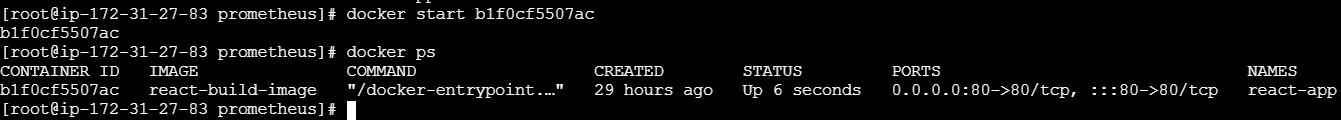
* Received 1 alert notification in my email
* The alert state is **firing** and the alert name is “**ApplicationDown**” along with labels and annotations

****

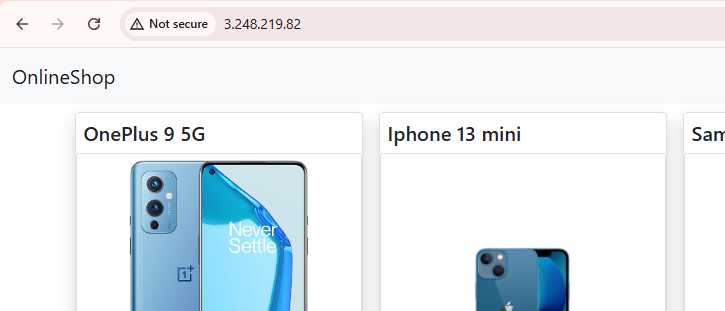
**Step 18:**

Resolving the react application

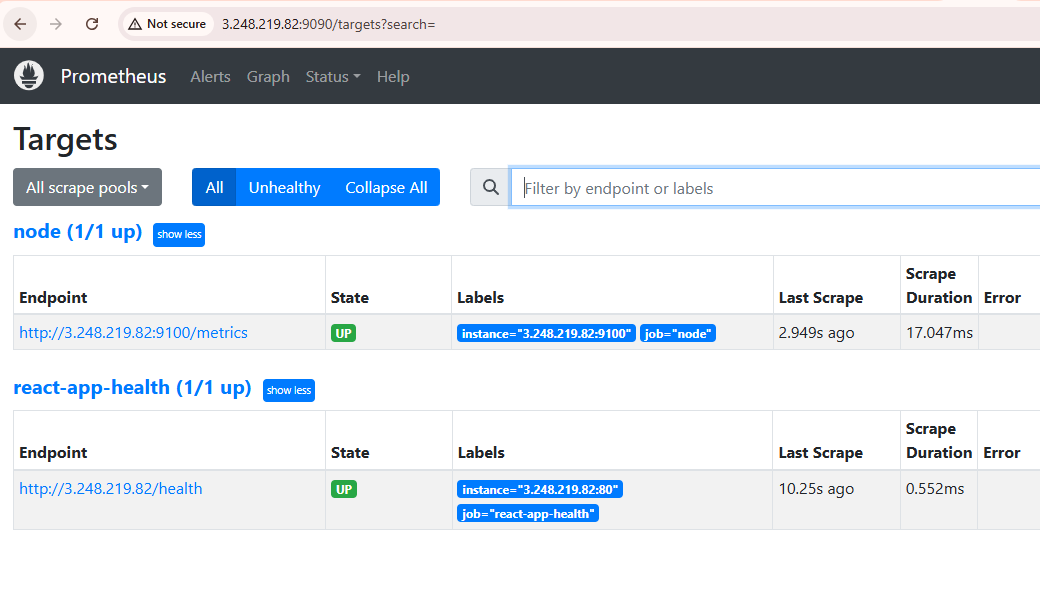
* **Docker start <container id> –** start the react application container
* **Docker ps**  –container is up and running

****

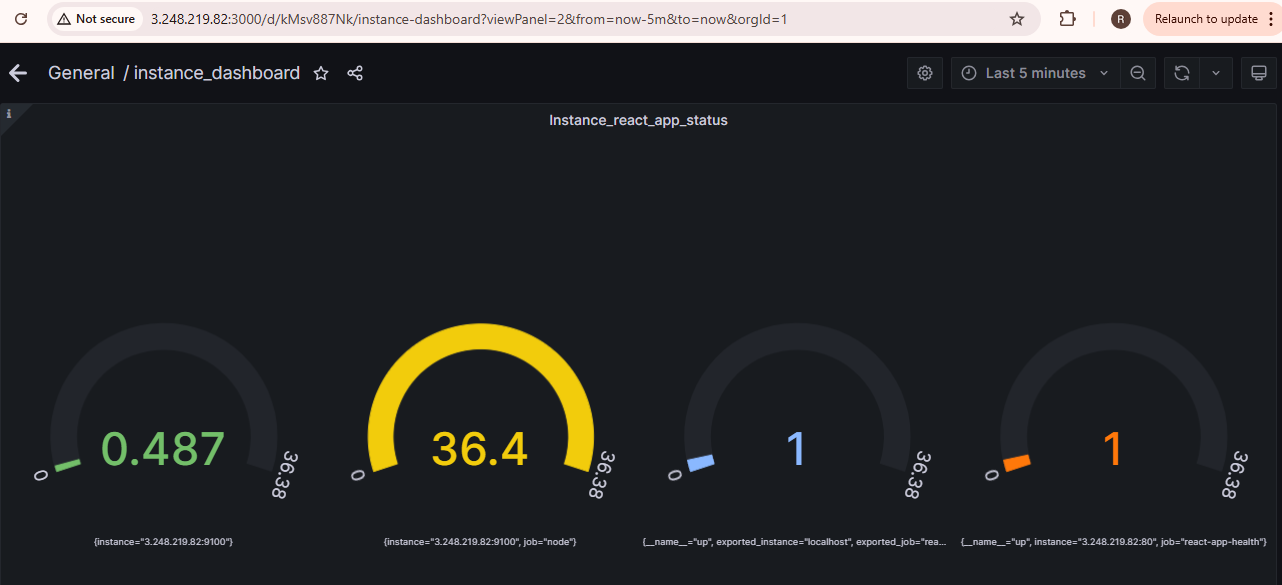
* Browser output at **http://3.248.219.82:80/**

****

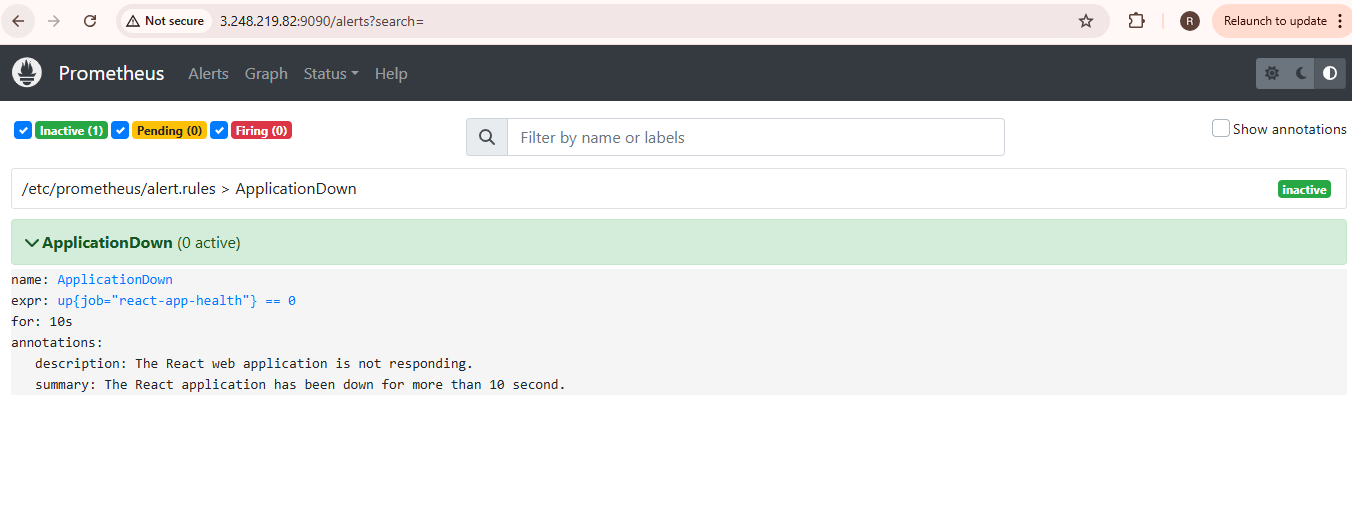
* In prometheus UI, the react-app-health endpoint state is **up** (application is healthy)

****

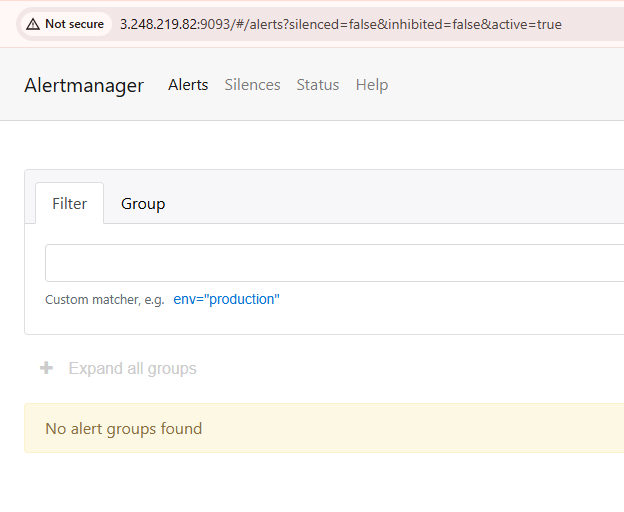
* In grafana dashboard , the job react-app-health metrics is **1** which means react application is **up** and **running**

****

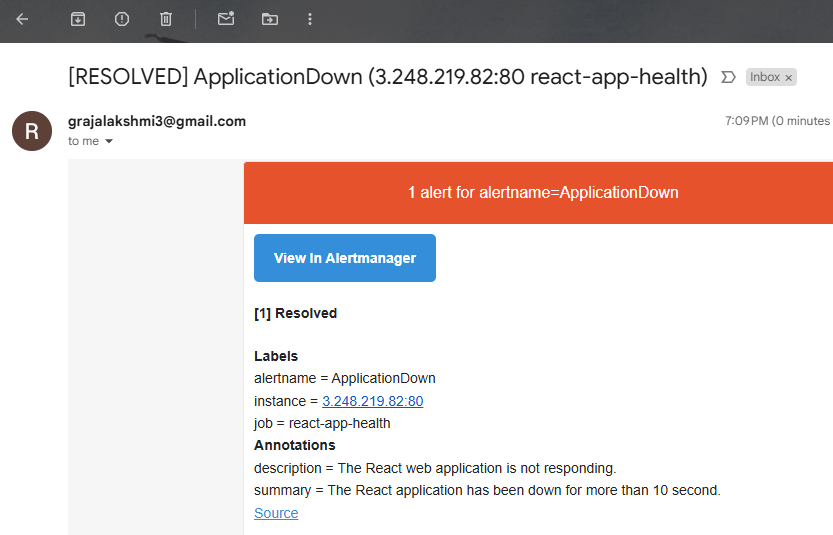
* In prometheus alert, **application down** is **inactive** which mean the react application is **up**

****

* In alertmanager UI , **no alert** is triggered

****

* Received the email as **resolved ApplicationDown** alert, means the react application which is down previously is resolved now, the application is **up** and running

****