REACTJS E-COMMERCE APPLICATION

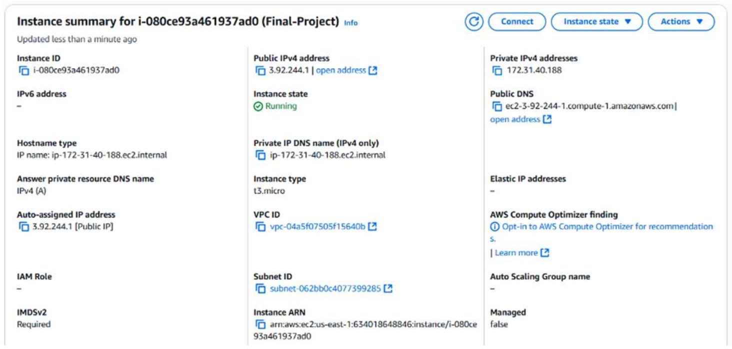
TASKS

Work Flow:

- Create an EC2 instance with the help of AWS Management Console with linux OS of required configuration.
- Now, Connect an EC2 instance with an help of Windows Terminal or Gitbash or Vbox.
- To connect an EC2 instance the command is:
  - ssh -i "**key_file**" ec2-user@"**Public_IP_address**"

Key_file --- Key file of the instance with the extension .pem

Public_IP_address --- Public IP address of the instance.



Instance summary for i-080ce93a461937ad0 (Final-Project)

Updated less than a minute ago

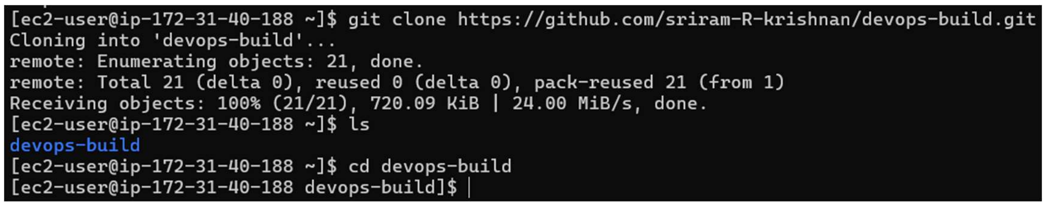
Instance ID i-080ce93a461937ad0	Public IPv4 address 3.92.244.1 | open address	Private IPv4 addresses 172.31.40.188
IPv6 address –	Instance state Running	Public DNS ec2-3-92-244-1.compute-1.amazonaws.com | open address
Hostname type IP name: ip-172-31-40-188.ec2.internal	Private IP DNS name (IPv4 only) ip-172-31-40-188.ec2.internal	
Answer private resource DNS name IPv4 (A)	Instance type t3.micro	Elastic IP addresses –
Auto-assigned IP address 3.92.244.1 [Public IP]	VPC ID vpc-04a5f07505f15640b	AWS Compute Optimizer finding Opt-in to AWS Compute Optimizer for recommendations. | Learn more
IAM Role –	Subnet ID subnet-062bb0c4077399285	Auto Scaling Group name –
IMDSv2 Required	Instance ARN arn:aws:ec2:us-east-1:634018648846:instance/i-080ce93a461937ad0	Managed false

1. **Deploy the given React application in a production-ready environment using Docker, Bash Scripts, GitHub, Jenkins, DockerHub, and AWS EC2 with monitoring.**

Application:

Step 1: Clone the Application Repository.

- ✓ To begin the deployment process, first, we need to clone the React application from the provided GitHub repository.
- ✓ This step ensures that the entire project codebase is available on your local machine or development server.
- ✓ Use the Git command-line interface (CLI) to perform this task. Open your terminal and execute the following command:
- ✓ After cloning, navigate into the project directory using:
- ✓ This step prepares the workspace for the next phases, including Dockerization and CI/CD integration.

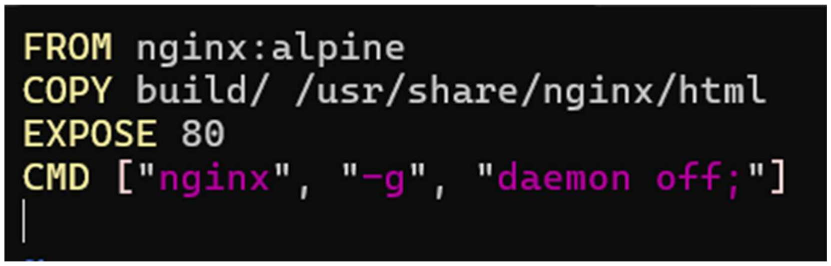


```
[ec2-user@ip-172-31-40-188 ~]$ git clone https://github.com/sriram-R-krishnan/devops-build.git
Cloning into 'devops-build'...
remote: Enumerating objects: 21, done.
remote: Total 21 (delta 0), reused 0 (delta 0), pack-reused 21 (from 1)
Receiving objects: 100% (21/21), 720.09 KiB | 24.00 MiB/s, done.
[ec2-user@ip-172-31-40-188 ~]$ ls
devops-build
[ec2-user@ip-172-31-40-188 ~]$ cd devops-build
[ec2-user@ip-172-31-40-188 devops-build]$ |
```

<u>**Docker:**</u>

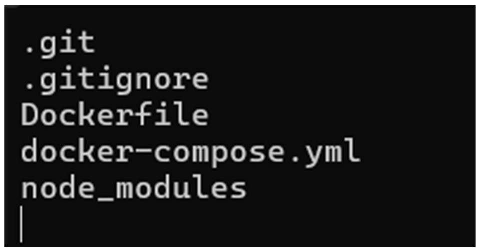
Step 1: Dockerize the Application

- ✓ In this step, we containerize the React application by creating a Dockerfile. This allows the application to be packaged with all its dependencies and ensures consistent behavior across different environments.
- ✓ We begin by creating a Dockerfile in the root directory of the project. This file uses the official Node.js base image to build the React application and then serves the production build using the serve package.
- ✓ DockerFile:



```
FROM nginx:alpine
COPY build/ /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

- ✓ The working directory inside the container is set to /app, and the application is built using npm install followed by npm run build.
- ✓ To serve the built application, the serve package is installed globally, and port 80 is exposed so that it can be accessed over HTTP. Finally, the container is configured to run the application on port 80 using the serve -s build -l 80 command.
- ✓ Additionally, we create a .dockerignore file to exclude unnecessary files and folders like node_modules, build, .git, and the Dockerfile itself from being copied into the Docker image. This reduces the image size and improves build performance.
- ✓ .dockerignore file:

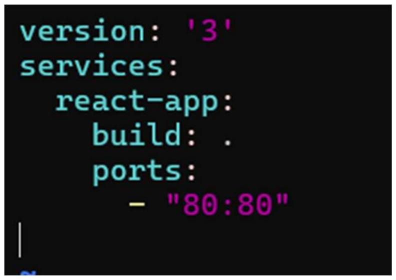


```
.git
.gitignore
Dockerfile
docker-compose.yml
node_modules
```

- ✓ This Dockerfile will be used later by Docker Compose and Jenkins to build and deploy the application in both development and production environments.

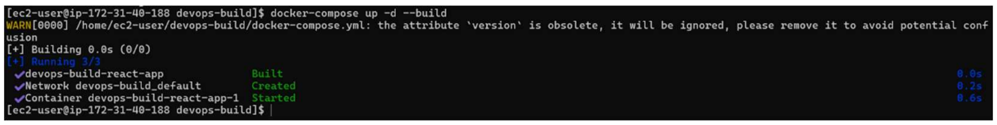
Step 2: Create a Docker Compose File

- ✓ In this step, we will define a docker-compose.yml file to simplify the process of running our Dockerized React application.
- ✓ Docker Compose allows us to manage multi-container applications using a single configuration file, though in this case, we are only running a single container.
- ✓ Inside the project root directory, create a new file named docker-compose.yml.
- ✓ In this file, define a service named react-app, which will:
  - Build the Docker image using the Dockerfile present in the same directory.
  - Map port 80 of the container to port 80 of the host machine, making the application accessible via HTTP on port 80.
- ✓ Below is the sample content for the docker-compose.yml file:

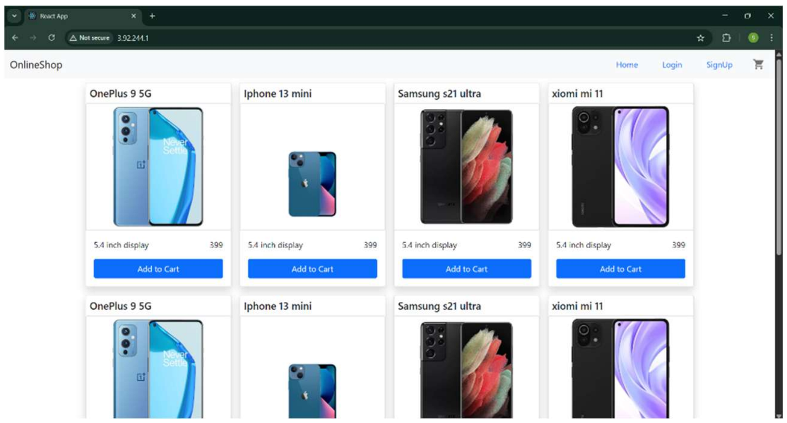


```
version: '3'
services:
  react-app:
    build: .
    ports:
      - "80:80"
```

- ✓ Save the file. You can now use the command docker-compose up to build and run the application container in one step.
- ✓ This file simplifies both local development and production deployment by bundling configuration into one manageable unit.



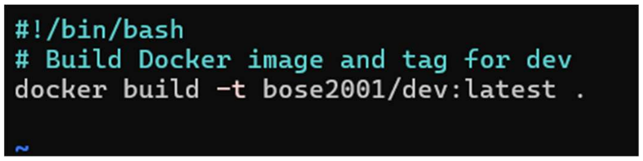
```
[ec2-user@ip-172-31-40-188 devops-build]$ docker-compose up -d --build
WARN[0000] /home/ec2-user/devops-build/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it to avoid potential conf
usion
[+] Building 0.0s (0/0)
[+] Running 3/3
 ✔devops-build-react-app           Built                                                                    0.0s
 ✔Network devops-build_default     Created                                                                  0.2s
 ✔Container devops-build-react-app-1  Started                                                               0.6s
[ec2-user@ip-172-31-40-188 devops-build]$ |
```



<u>Bash Scripting:</u>

- ✓ In this step, two bash scripts were created to automate the build and deployment process of the Dockerized application.
- ✓ **build.sh Script**:

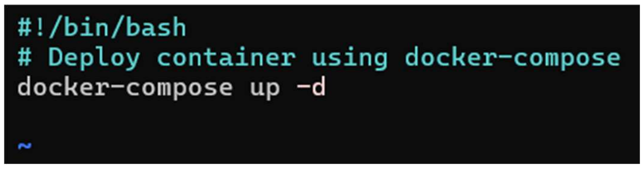
  This script is responsible for building the Docker image from the Dockerfile present in the project directory. It uses the docker build command with a custom tag and ensures that the application image is built consistently for further use.



```
#!/bin/bash
# Build Docker image and tag for dev
docker build -t bose2001/dev:latest .

~
```

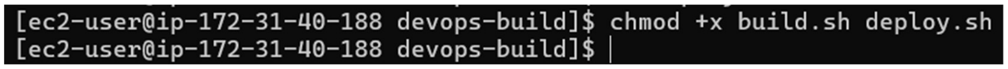
- ✓ **deploy.sh Script**:

  This script is used to deploy the application by running the Docker container using docker-compose. It ensures the container runs in detached mode, exposing port 80 for public access.

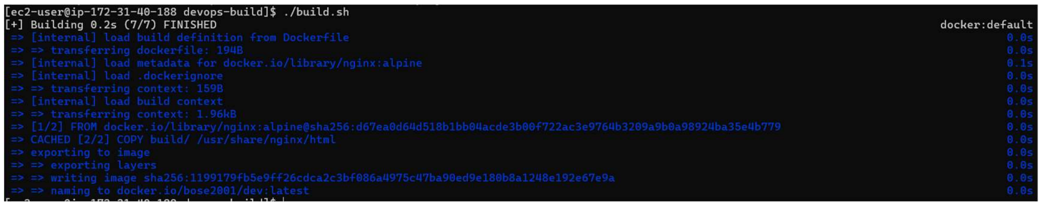


```
#!/bin/bash
# Deploy container using docker-compose
docker-compose up -d

~
```

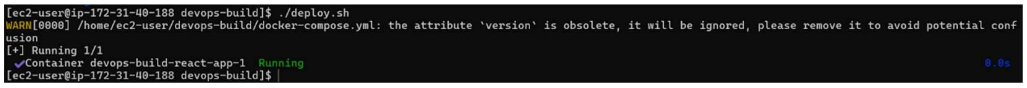
- ✓ These scripts help streamline the local development and deployment workflow by reducing the need to manually run repetitive Docker commands.



```
[ec2-user@ip-172-31-40-188 devops-build]$ chmod +x build.sh deploy.sh
[ec2-user@ip-172-31-40-188 devops-build]$ |
```



```
[ec2-user@ip-172-31-40-188 devops-build]$ ./build.sh
[+] Building 0.2s (7/7) FINISHED                                                                              docker:default
 => [internal] load build definition from Dockerfile                                                                     0.0s
 => => transferring dockerfile: 194B                                                                                     0.0s
 => [internal] load metadata for docker.io/library/nginx:alpine                                                          0.1s
 => [internal] load .dockerignore                                                                                        0.0s
 => => transferring context: 159B                                                                                        0.0s
 => [internal] load build context                                                                                        0.0s
 => => transferring context: 1.96kB                                                                                      0.0s
 => [1/2] FROM docker.io/library/nginx:alpine@sha256:d67ea0d64d518b1bb04acde3b00f722ac3e9764b3209a9b0a98924ba35e4b779     0.0s
 => CACHED [2/2] COPY build/ /usr/share/nginx/html                                                                       0.0s
 => exporting to image                                                                                                   0.0s
 => => exporting layers                                                                                                  0.0s
 => => writing image sha256:1199179fb5e9ff26cdca2c3bf086a4975c47ba90ed9e180b8a1248e192e67e9a                             0.0s
 => => naming to docker.io/bose2001/dev:latest                                                                           0.0s
```



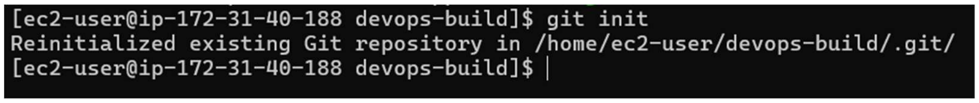
```
[ec2-user@ip-172-31-40-188 devops-build]$ ./deploy.sh
WARN[0000] /home/ec2-user/devops-build/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it to avoid potential conf
usion
[+] Running 1/1
 ✔Container devops-build-react-app-1  Running                                                                          0.0s
[ec2-user@ip-172-31-40-188 devops-build]$ |
```

<u>Version Control:</u>

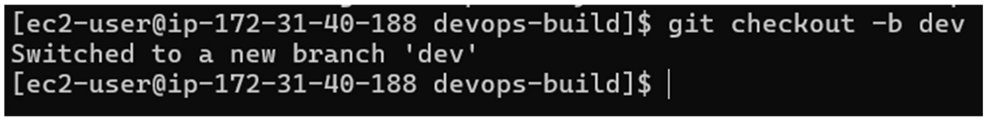
Step 1: Push the Code to GitHub Using CLI

- ✓ To maintain version control and enable collaboration, we pushed our application code to a remote GitHub repository using only Git CLI (Command Line Interface).

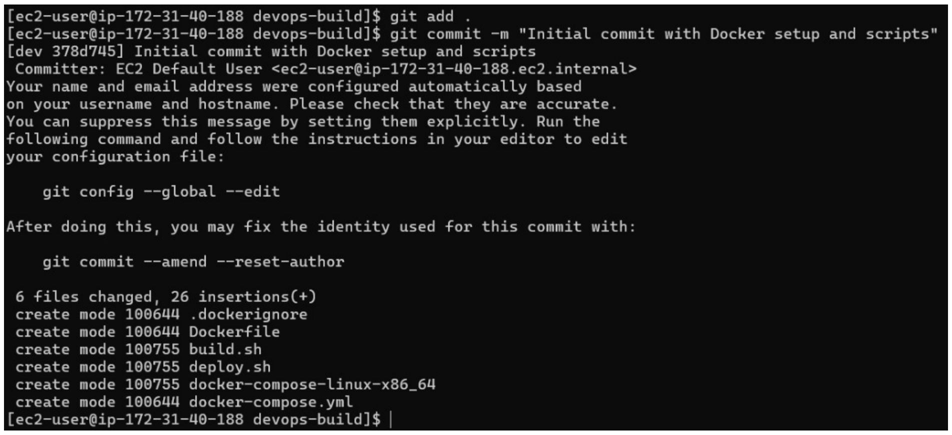
- ✓ First, we initialized a Git repository inside our project directory using the git init command. We then created a new branch named dev, as all development-related changes will first be committed to this branch before merging to production (master).
- ✓ Next, we added all the project files to Git using git add . and committed the changes with an appropriate message using git commit -m "Initial commit".
- ✓ After that, we connected our local Git repository to a remote GitHub repository using git remote add origin <repo-url>.



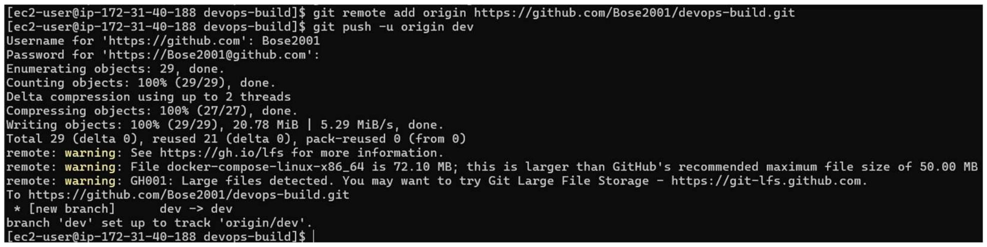
```
[ec2-user@ip-172-31-40-188 devops-build]$ git init
Reinitialized existing Git repository in /home/ec2-user/devops-build/.git/
[ec2-user@ip-172-31-40-188 devops-build]$
```



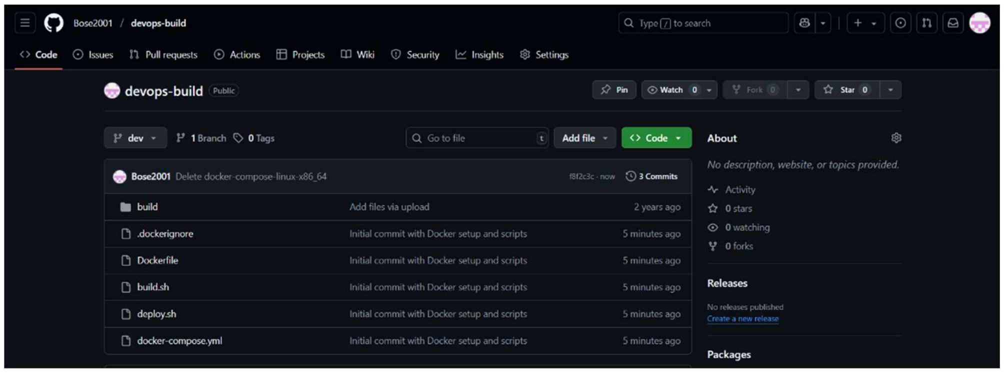
```
[ec2-user@ip-172-31-40-188 devops-build]$ git checkout -b dev
Switched to a new branch 'dev'
[ec2-user@ip-172-31-40-188 devops-build]$
```



```
[ec2-user@ip-172-31-40-188 devops-build]$ git add .
[ec2-user@ip-172-31-40-188 devops-build]$ git commit -m "Initial commit with Docker setup and scripts"
[dev 378d745] Initial commit with Docker setup and scripts
 Committer: EC2 Default User <ec2-user@ip-172-31-40-188.ec2.internal>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 6 files changed, 26 insertions(+)
 create mode 100644 .dockerignore
 create mode 100644 Dockerfile
 create mode 100755 build.sh
 create mode 100755 deploy.sh
 create mode 100755 docker-compose-linux-x86_64
 create mode 100644 docker-compose.yml
[ec2-user@ip-172-31-40-188 devops-build]$
```



```
[ec2-user@ip-172-31-40-188 devops-build]$ git remote add origin https://github.com/Bose2001/devops-build.git
[ec2-user@ip-172-31-40-188 devops-build]$ git push -u origin dev
Username for 'https://github.com': Bose2001
Password for 'https://Bose2001@github.com':
Enumerating objects: 29, done.
Counting objects: 100% (29/29), done.
Delta compression using up to 2 threads
Compressing objects: 100% (27/27), done.
Writing objects: 100% (29/29), 20.78 MiB | 5.29 MiB/s, done.
Total 29 (delta 0), reused 21 (delta 0), pack-reused 0 (from 0)
remote: warning: See https://gh.io/lfs for more information.
remote: warning: File docker-compose-linux-x86_64 is 72.10 MB; this is larger than GitHub's recommended maximum file size of 50.00 MB
remote: warning: GH001: Large files detected. You may want to try Git Large File Storage - https://git-lfs.github.com.
To https://github.com/Bose2001/devops-build.git
 * [new branch]      dev -> dev
branch 'dev' set up to track 'origin/dev'.
[ec2-user@ip-172-31-40-188 devops-build]$
```



devops-build Public

dev · 1 Branch · 0 Tags

Bose2001 Delete docker-compose-linux-x86_64 · f8f2c3c · now · 3 Commits

build	Add files via upload	2 years ago
.dockerignore	Initial commit with Docker setup and scripts	5 minutes ago
Dockerfile	Initial commit with Docker setup and scripts	5 minutes ago
build.sh	Initial commit with Docker setup and scripts	5 minutes ago
deploy.sh	Initial commit with Docker setup and scripts	5 minutes ago
docker-compose.yml	Initial commit with Docker setup and scripts	5 minutes ago

About

No description, website, or topics provided.

Activity
0 stars
0 watching
0 forks

Releases

No releases published
Create a new release

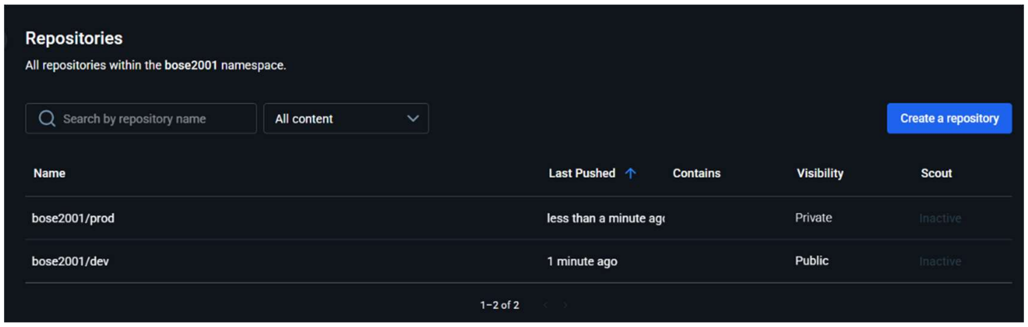
Packages

- ✓ Finally, we pushed our code to the dev branch on GitHub using the command git push -u origin dev.
- ✓ We also created two important configuration files, .gitignore and .dockerignore, to ensure that unnecessary or sensitive files (such as node_modules, .git, and Docker build artifacts) are excluded from Git and Docker image builds respectively. This helps in keeping the repository clean and optimized.

<u>DockerHub:</u>

Step 1: DockerHub Setup

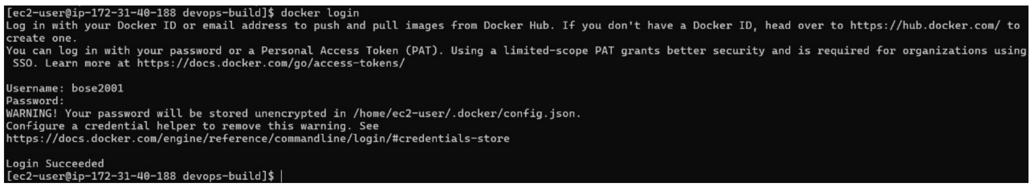
- ✓ To manage different deployment environments, we created two repositories on DockerHub: one named **"dev"** (public) and another named **"prod"** (private). These repositories are used to store Docker images built from our application.
- ✓ First, we logged into DockerHub using the CLI with the command docker login and provided our DockerHub username and password.
- ✓ After successfully logging in, we built the Docker image locally using the docker build command and tagged the image appropriately for each environment:
    - For the development environment, we tagged the image as your-dockerhub-username/dev:latest and pushed it to the public DockerHub repo using:
    - For the production environment, we tagged the image as your-dockerhub-username/prod:latest and pushed it to the private DockerHub repo using:
- ✓ This setup ensures separation of environments where the dev repository can be used for testing and staging, while the prod repository is reserved for stable, production-ready images.


Repositories

All repositories within the bose2001 namespace.

Name	Last Pushed	Contains	Visibility	Scout
bose2001/prod	less than a minute ag		Private	Inactive
bose2001/dev	1 minute ago		Public	Inactive

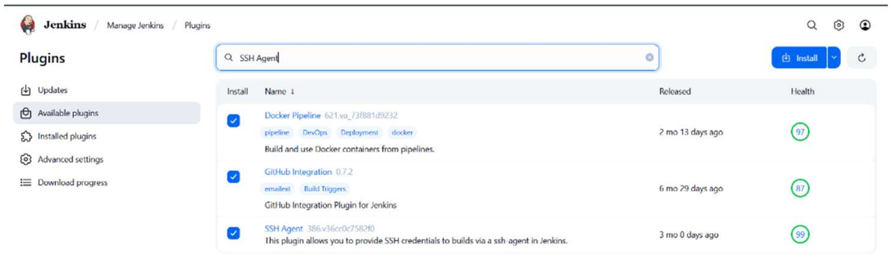
1–2 of 2


```
[ec2-user@ip-172-31-40-188 devops-build]$ docker login
Log in with your Docker ID or email address to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com/ to
create one.
You can log in with your password or a Personal Access Token (PAT). Using a limited-scope PAT grants better security and is required for organizations using
 SSO. Learn more at https://docs.docker.com/go/access-tokens/

Username: bose2001
Password:
WARNING! Your password will be stored unencrypted in /home/ec2-user/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
[ec2-user@ip-172-31-40-188 devops-build]$ |
```

Jenkins:

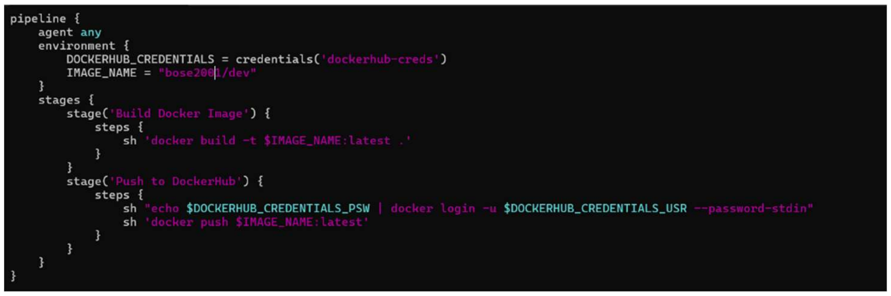
Step 1: Jenkins Setup and CI/CD Pipeline Configuration

- ✓ In this step, Jenkins is used to automate the build, test, and deployment process. First, Jenkins is installed on an Ubuntu-based EC2 instance.
- ✓ The installation involves updating the system packages, adding the Jenkins repository, and installing Jenkins using the package manager. Once Jenkins is installed, the service is started and enabled to launch on system boot
- ✓ The Jenkins dashboard is accessed via the EC2 instance's public IP and port 8080.
- ✓ After successful installation, GitHub is connected to Jenkins using webhooks to trigger automated builds on code push events.
- ✓ Two Jenkins freestyle projects (jobs) are created: one for the **dev** branch and another for the **master** branch.
    - The **dev job** is configured to pull code from the GitHub dev branch whenever there is a new commit. It builds a Docker image from the source code and pushes the image to the public "dev" repository on DockerHub.
    - The **master job** is triggered when changes are merged from dev to master. This job builds a production Docker image and pushes it to the private "prod" repository on DockerHub. After pushing, the job deploys the latest production image to the running AWS EC2 server using secure SSH-based deployment (can be automated using a shell script or Ansible).
- ✓ Credentials for GitHub and DockerHub are securely stored in Jenkins using the built-in credential manager to enable non-interactive logins during the CI/CD pipeline execution.
- ✓ This automated setup ensures that any code pushed to the **dev** branch triggers a build and stores the image in the dev repo, and a merge into **master** triggers a production-grade image deployment—making the entire development-to-deployment workflow seamless and production-ready.

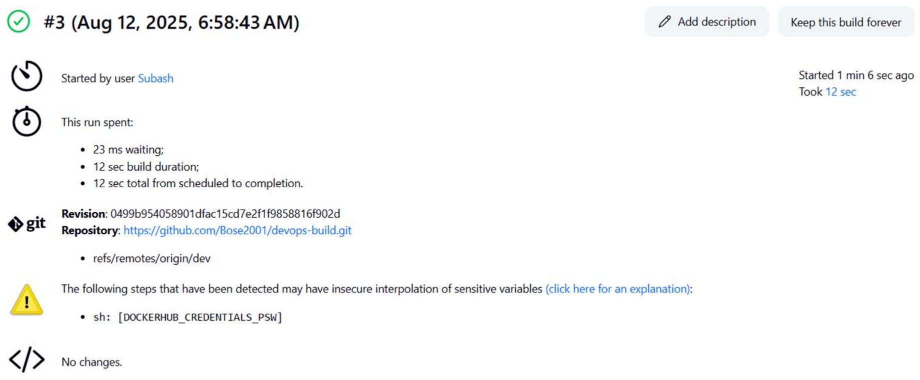


Install	Name ↓	Released	Health
✓	Docker Pipeline 621.va_73f881d9232 pipeline DevOps Deployment docker Build and use Docker containers from pipelines.	2 mo 13 days ago	97
✓	GitHub Integration 0.7.2 emailext Build Triggers GitHub Integration Plugin for Jenkins	6 mo 29 days ago	87
✓	SSH Agent 386.v36cc0c7582f0 This plugin allows you to provide SSH credentials to builds via a ssh-agent in Jenkins.	3 mo 0 days ago	99

✓ **Jenkinsfile (Development Build & Push)** – *Handles building the development version of the application and pushing it to the dev Docker Hub repository.*



```
pipeline {
    agent any
    environment {
        DOCKERHUB_CREDENTIALS = credentials('dockerhub-creds')
        IMAGE_NAME = "bose2001/dev"
    }
    stages {
        stage('Build Docker Image') {
            steps {
                sh 'docker build -t $IMAGE_NAME:latest .'
            }
        }
        stage('Push to DockerHub') {
            steps {
                sh "echo $DOCKERHUB_CREDENTIALS_PSW | docker login -u $DOCKERHUB_CREDENTIALS_USR --password-stdin"
                sh 'docker push $IMAGE_NAME:latest'
            }
        }
    }
}
```



#3 (Aug 12, 2025, 6:58:43 AM)

Started by user Subash

Started 1 min 6 sec ago
Took 12 sec

This run spent:

- 23 ms waiting;
- 12 sec build duration;
- 12 sec total from scheduled to completion.

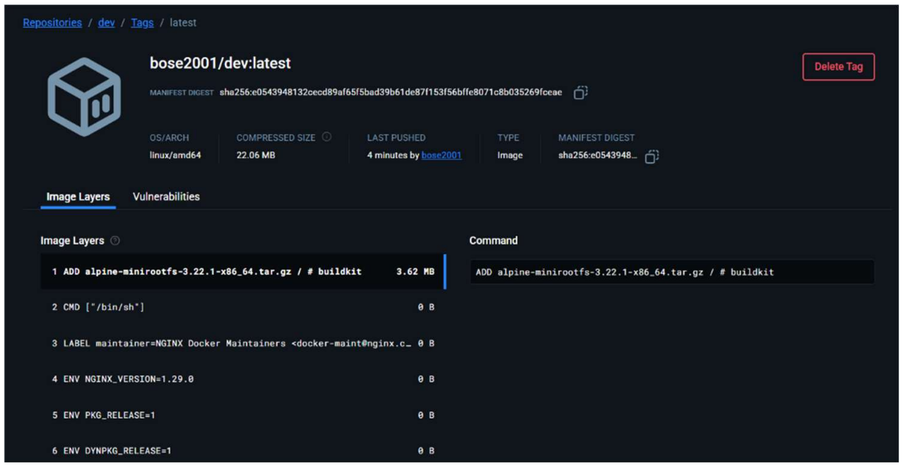
Revision: 0499b954058901dfac15cd7e2f1f9858816f902d
Repository: https://github.com/Bose2001/devops-build.git

- refs/remotes/origin/dev

The following steps that have been detected may have insecure interpolation of sensitive variables (click here for an explanation):

- `sh: [DOCKERHUB_CREDENTIALS_PSW]`

No changes.



Repositories / dev / Tags / latest

bose2001/dev:latest

MANIFEST DIGEST sha256:e0543948132cecd89af65f5bad39b61de87f153f56bffe8071c8b035269fceae

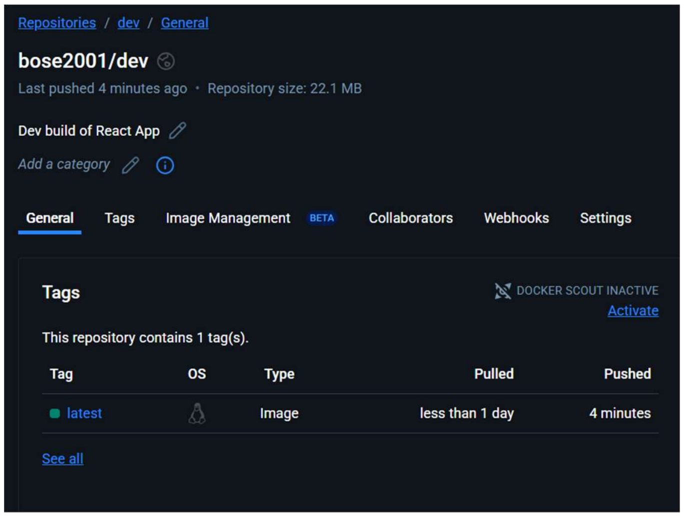
OS/ARCH	COMPRESSED SIZE	LAST PUSHED	TYPE	MANIFEST DIGEST
linux/amd64	22.06 MB	4 minutes by bose2001	Image	sha256:e0543948...

Image Layers

```
1 ADD alpine-minirootfs-3.22.1-x86_64.tar.gz / # buildkit       3.62 MB
2 CMD ["/bin/sh"]                                                   0 B
3 LABEL maintainer=NGINX Docker Maintainers <docker-maint@nginx.c… 0 B
4 ENV NGINX_VERSION=1.29.0                                          0 B
5 ENV PKG_RELEASE=1                                                 0 B
6 ENV DYNPKG_RELEASE=1                                              0 B
```

Command

```
ADD alpine-minirootfs-3.22.1-x86_64.tar.gz / # buildkit
```



Repositories / dev / General

bose2001/dev

Last pushed 4 minutes ago · Repository size: 22.1 MB

Dev build of React App

Add a category

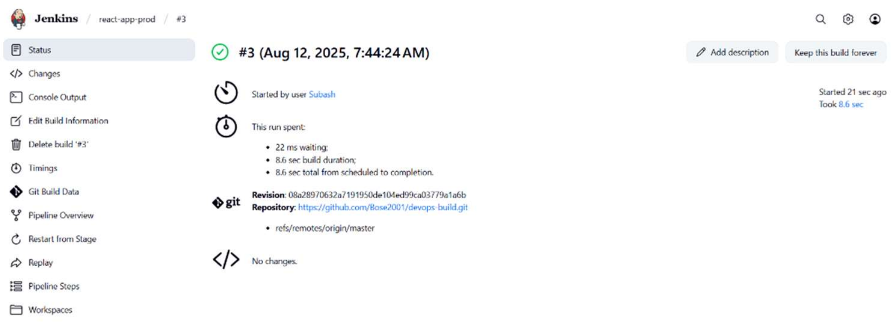
Tags

This repository contains 1 tag(s).

Tag	OS	Type	Pulled	Pushed
latest		Image	less than 1 day	4 minutes

See all

- ✓ **Jenkinsfile (Production Deploy)** – *Handles pulling the production image from Docker Hub and deploying it to the target EC2 instance.*



#3 (Aug 12, 2025, 7:44:24 AM)

Started by user Subash

Started 21 sec ago
Took 8.6 sec

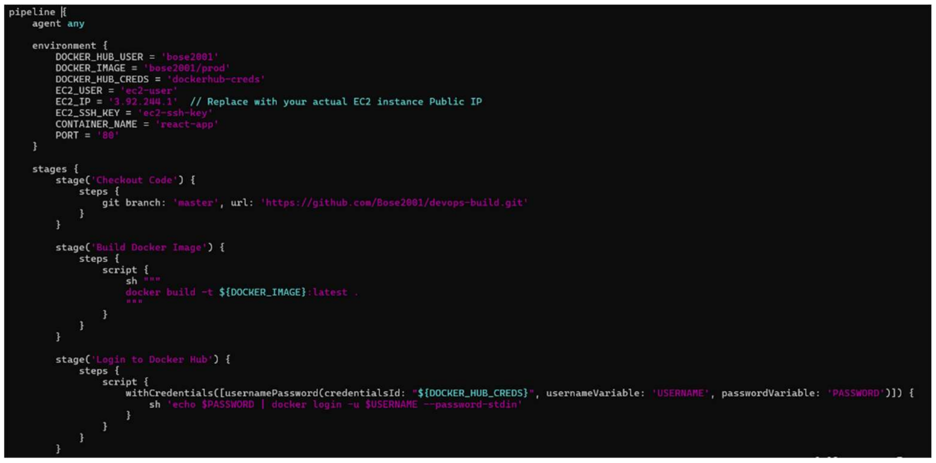
This run spent:

- 22 ms waiting;
- 8.6 sec build duration;
- 8.6 sec total from scheduled to completion.

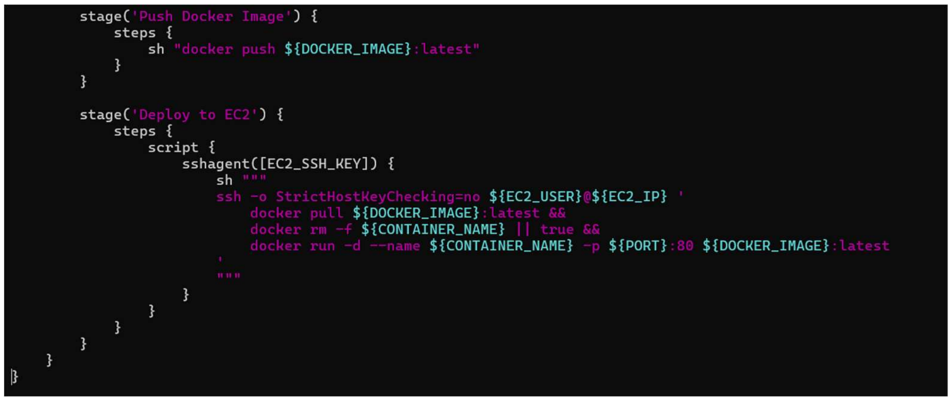
git **Revision**: 08a28970632a7191950de104ed99ca03779a1a6b
Repository: https://github.com/Bose2001/devops-build.git

- refs/remotes/origin/master

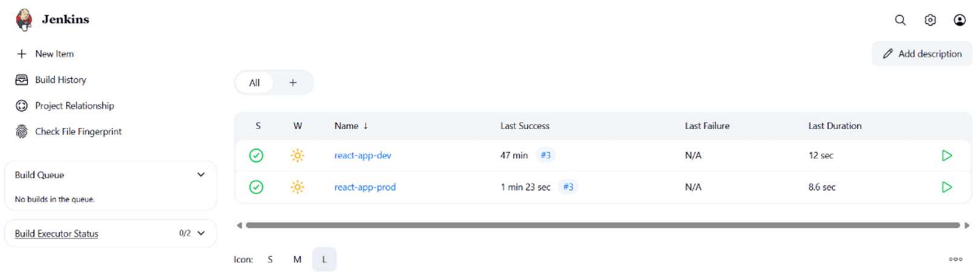
No changes.



```
pipeline {
    agent any

    environment {
        DOCKER_HUB_USER = 'bose2001'
        DOCKER_IMAGE = 'bose2001/prod'
        DOCKER_HUB_CREDS = 'dockerhub-creds'
        EC2_USER = 'ec2-user'
        EC2_IP = '3.92.244.1'  // Replace with your actual EC2 instance Public IP
        EC2_SSH_KEY = 'ec2-ssh-key'
        CONTAINER_NAME = 'react-app'
        PORT = '80'
    }

    stages {
        stage('Checkout Code') {
            steps {
                git branch: 'master', url: 'https://github.com/Bose2001/devops-build.git'
            }
        }

        stage('Build Docker Image') {
            steps {
                script {
                    sh """
                    docker build -t ${DOCKER_IMAGE}:latest .
                    """
                }
            }
        }

        stage('Login to Docker Hub') {
            steps {
                script {
                    withCredentials([usernamePassword(credentialsId: "${DOCKER_HUB_CREDS}", usernameVariable: 'USERNAME', passwordVariable: 'PASSWORD')]) {
                        sh 'echo $PASSWORD | docker login -u $USERNAME --password-stdin'
                    }
                }
            }
        }
```



```
        stage('Push Docker Image') {
            steps {
                sh "docker push ${DOCKER_IMAGE}:latest"
            }
        }

        stage('Deploy to EC2') {
            steps {
                script {
                    sshagent([EC2_SSH_KEY]) {
                        sh """
                        ssh -o StrictHostKeyChecking=no ${EC2_USER}@${EC2_IP} '
                            docker pull ${DOCKER_IMAGE}:latest &&
                            docker rm -f ${CONTAINER_NAME} || true &&
                            docker run -d --name ${CONTAINER_NAME} -p ${PORT}:80 ${DOCKER_IMAGE}:latest
                        '
                        """
                    }
                }
            }
        }
    }
}
```



No builds in the queue.

S	W	Name ↓	Last Success	Last Failure	Last Duration
✓	☀	react-app-dev	47 min #3	N/A	12 sec
✓	☀	react-app-prod	1 min 23 sec #3	N/A	8.6 sec

AWS:

- ✓ To host the application, we need to launch an EC2 instance on AWS and deploy the Dockerized React app on it.

Step 1: Launching the EC2 Instance

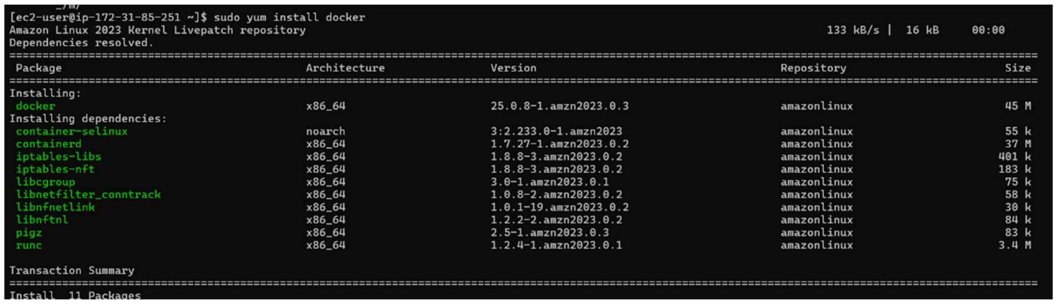
- ✓ We begin by logging into the AWS Management Console and navigating to the EC2 service. From there, we launch a new instance using the following configuration:
  - **Instance Type**: t2.micro (eligible under the free tier)
  - **AMI**: Ubuntu Server (latest stable version)
  - **Security Group Configuration**:
    - ➢ Allow **HTTP (port 80)** from anywhere so that the application can be accessed publicly.
    - ➢ Allow **SSH (port 22)** only from our own IP address to ensure secure login access.
- ✓ We also create or select an existing key pair (e.g., my-key.pem) for SSH access to the instance.

Step 2: Connecting to the EC2 Instance

- ✓ Once the instance is running, we connect to it securely using SSH. The following command is used in the terminal (from the directory where the .pem key is stored):
- ✓ This command gives us access to the server where we will install Docker and deploy the app.

Step 3: Installing Docker on EC2

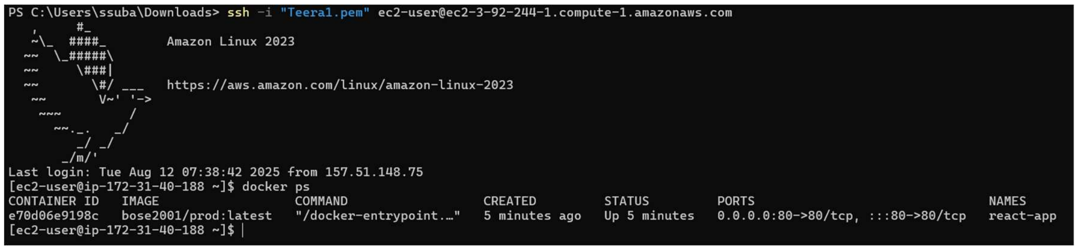
- ✓ After logging into the instance, we update the package lists and install Docker using the following commands:
- ✓ Once installed, we start the Docker service and add the ubuntu user to the Docker group to allow Docker commands without sudo:
- ✓ We then log out and log back in to apply the Docker group changes.



```
[ec2-user@ip-172-31-85-251 ~]$ sudo yum install docker
Amazon Linux 2023 Kernel Livepatch repository                                          133 kB/s |  16 kB     00:00
Dependencies resolved.
=========================================================================================================================
 Package                      Architecture          Version                          Repository                    Size
=========================================================================================================================
Installing:
 docker                       x86_64                25.0.8-1.amzn2023.0.3            amazonlinux                   45 M
Installing dependencies:
 container-selinux            noarch                3:2.233.0-1.amzn2023             amazonlinux                   55 k
 containerd                   x86_64                1.7.27-1.amzn2023.0.2            amazonlinux                   37 M
 iptables-libs                x86_64                1.8.8-3.amzn2023.0.2             amazonlinux                  401 k
 iptables-nft                 x86_64                1.8.8-3.amzn2023.0.2             amazonlinux                  183 k
 libcgroup                    x86_64                3.0-1.amzn2023.0.1               amazonlinux                   75 k
 libnetfilter_conntrack       x86_64                1.0.8-2.amzn2023.0.2             amazonlinux                   58 k
 libnfnetlink                 x86_64                1.0.1-19.amzn2023.0.2            amazonlinux                   30 k
 libnftnl                     x86_64                1.2.2-2.amzn2023.0.2             amazonlinux                   84 k
 pigz                         x86_64                2.5-1.amzn2023.0.3               amazonlinux                   83 k
 runc                         x86_64                1.2.4-1.amzn2023.0.1             amazonlinux                  3.4 M

Transaction Summary
=========================================================================================================================
Install  11 Packages
```

Step 4: Deploying the Application

- ✓ With Docker installed, we pull the Docker image from Docker Hub using the following command (make sure the correct image name is used):
- ✓ Finally, we run the container and map it to port 80 so it's accessible publicly via the instance's IP address:
- ✓ Now, when we open a browser and visit http://<EC2-Public-IP>, we should see the deployed React application running successfully.

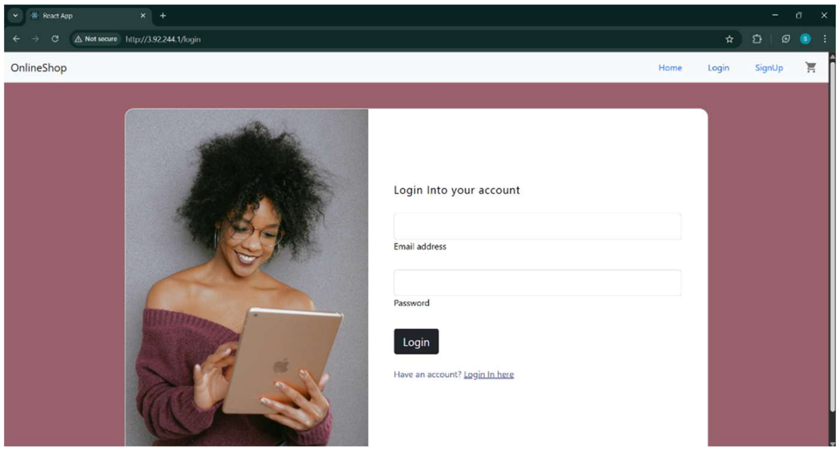


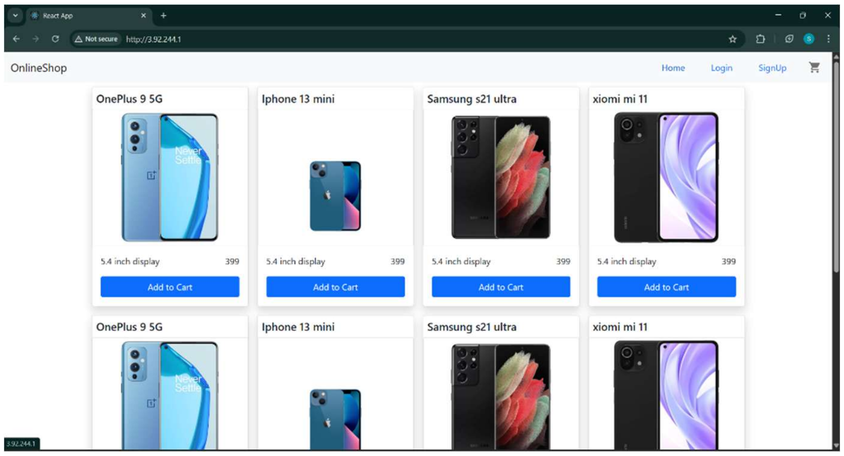
```
PS C:\Users\ssuba\Downloads> ssh -i "Teera1.pem" ec2-user@ec2-3-92-244-1.compute-1.amazonaws.com
   ,     #_
   ~\_  ####_        Amazon Linux 2023
  ~~  \_#####\
  ~~     \###|
  ~~       \#/ ___   https://aws.amazon.com/linux/amazon-linux-2023
   ~~       V~' '->
    ~~~         /
      ~~._.   _/
         _/ _/
       _/m/'
Last login: Tue Aug 12 07:38:42 2025 from 157.51.148.75
[ec2-user@ip-172-31-40-188 ~]$ docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED         STATUS         PORTS                               NAMES
e70d06e9198c   bose2001/prod:latest   "/docker-entrypoint.…"   5 minutes ago   Up 5 minutes   0.0.0.0:80->80/tcp, :::80->80/tcp   react-app
[ec2-user@ip-172-31-40-188 ~]$ |
```

<u>**Monitoring:**</u>

Step 1: Monitoring the Application

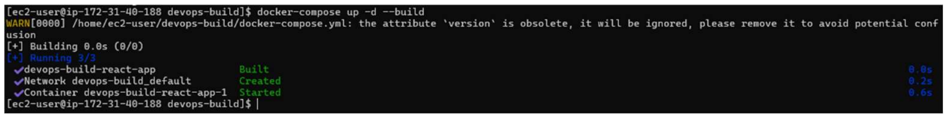
- ✓ To ensure the deployed application remains healthy and available, a monitoring system is set up using an open-source tool called Uptime Kuma.
- ✓ Uptime Kuma allows us to continuously check the health status of the application and notifies us only when the application goes down.
- ✓ First, we installed Docker on the EC2 instance (if not already installed), and then deployed Uptime Kuma using the following Docker command:
- ✓ This command runs the Uptime Kuma monitoring dashboard on port 3001. By accessing http://<EC2-PUBLIC-IP>:3001 in a browser, we can open the Uptime Kuma interface.
- ✓ From there, we create a new monitor to regularly check the availability of the deployed application running at http://<EC2-PUBLIC-IP>.
- ✓ Additionally, we configured **email notifications** in Uptime Kuma by setting up SMTP details in the notification settings.
- ✓ This ensures that alerts are sent via email **only when the application goes down**, helping maintain uptime and take immediate action if any issue arises.
- ✓ This setup provides basic but effective production-level monitoring without any third-party paid services, making it ideal for small to mid-scale deployments.





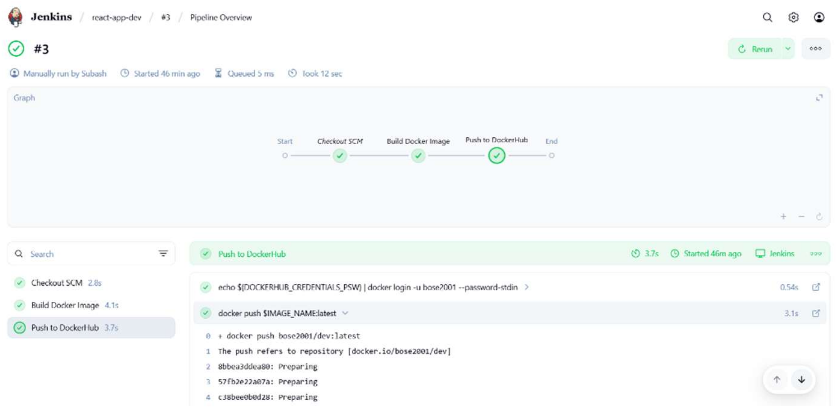
Final Test:

- ✓ After deploying the application, a series of tests were performed to ensure everything functions correctly across all components of the deployment pipeline:
- ✓ **Application Accessibility**:
  The deployed React application was accessed via a web browser using the EC2 instance's public IP address (http://<EC2-IP>). The application loaded successfully on port 80, confirming that the Docker container was running and serving the build correctly.



```
[ec2-user@ip-172-31-40-188 devops-build]$ docker-compose up -d --build
WARN[0000] /home/ec2-user/devops-build/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it to avoid potential conf
usion
[+] Building 0.0s (0/0)
[+] Running 3/3
 ✔devops-build-react-app          Built                                                                              0.0s
 ✔Network devops-build_default    Created                                                                            0.2s
 ✔Container devops-build-react-app-1  Started                                                                        0.6s
[ec2-user@ip-172-31-40-188 devops-build]$ |
```

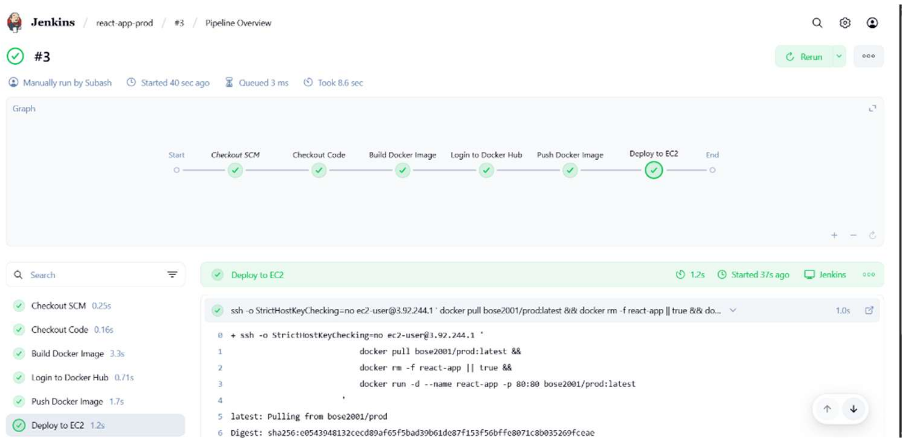
- ✓ **Dev Branch Pipeline Verification**:
  A new commit was pushed to the dev branch of the GitHub repository using Git CLI. This triggered the Jenkins job automatically. The Jenkins pipeline executed the build process, created a Docker image, and successfully pushed it to the public dev Docker Hub repository.



```
echo ${DOCKERHUB_CREDENTIALS_PSW} | docker login -u bose2001 --password-stdin   0.54s
docker push $IMAGE_NAME:latest   3.1s
+ docker push bose2001/dev:latest
The push refers to repository [docker.io/bose2001/dev]
8bbea3ddea80: Preparing
57fb2e22a07a: Preparing
c38bee0b0d28: Preparing
```

✓ **Master Branch Pipeline Verification**:

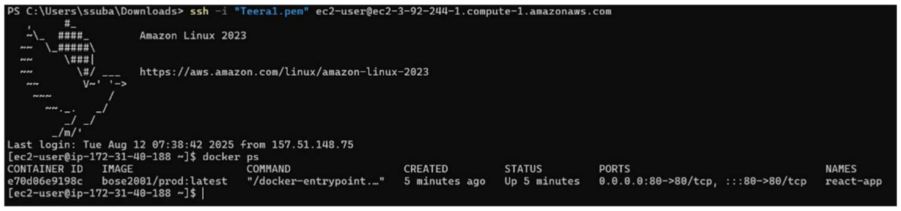
The dev branch was merged into the master branch. This action triggered another Jenkins job configured for the master branch. Jenkins built the production Docker image, pushed it to the private prod repository on Docker Hub, and deployed it on the AWS EC2 instance.



```
ssh -o StrictHostKeyChecking=no ec2-user@3.92.244.1 ' docker pull bose2001/prod:latest && docker rm -f react-app || true && do...   1.0s
+ ssh -o StrictHostKeyChecking=no ec2-user@3.92.244.1 '
                        docker pull bose2001/prod:latest &&
                        docker rm -f react-app || true &&
                        docker run -d --name react-app -p 80:80 bose2001/prod:latest
                      '
latest: Pulling from bose2001/prod
Digest: sha256:e0543948132cecd89af65f5bad39b61de87f153f56bffe8071c8b035269fceae
```

✓ **Automated Deployment Check**:

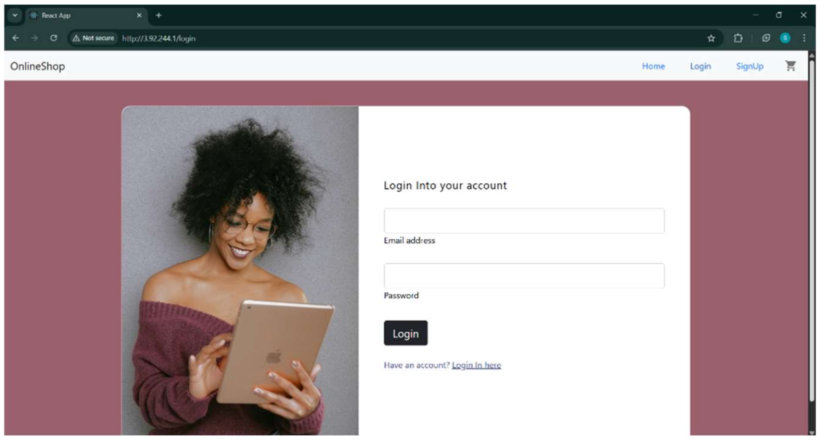
Post-deployment, the application was verified again via the browser to confirm that the newly deployed production version was accessible and working as expected on port 80.



```
PS C:\Users\ssuba\Downloads> ssh -i "Teera1.pem" ec2-user@ec2-3-92-244-1.compute-1.amazonaws.com
   ,     #_
   ~\_  ####_        Amazon Linux 2023
  ~~  \_#####\
  ~~     \###|
  ~~       \#/ ___   https://aws.amazon.com/linux/amazon-linux-2023
   ~~       V~' '->
    ~~~         /
      ~~._.   _/
         _/ _/
       _/m/'
Last login: Tue Aug 12 07:38:42 2025 from 157.51.148.75
[ec2-user@ip-172-31-40-188 ~]$ docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED         STATUS         PORTS                                 NAMES
e70d06e9198c   bose2001/prod:latest   "/docker-entrypoint.…"   5 minutes ago   Up 5 minutes   0.0.0.0:80->80/tcp, :::80->80/tcp     react-app
[ec2-user@ip-172-31-40-188 ~]$ |
```

✓ **Monitoring Validation**:

Uptime Kuma, an open-source monitoring tool, was configured on the EC2 instance and accessed via port 3001. The application URL (http://<EC2-IP>) was added as a monitor. The monitoring system was tested by temporarily stopping the container. A down alert was successfully triggered, and notification settings were validated to ensure email alerts are sent only when the application goes down.



- ✓ **Security Group Verification**:

  The EC2 security group was reviewed to confirm that:

  - Port 80 (HTTP) was open to all IPs for global access to the app.
  - Port 22 (SSH) was restricted to allow login only from the developer's IP address for security.

- ✓ **End-to-End Workflow Test**:

  A complete end-to-end test from pushing code → triggering CI/CD → updating Docker Hub → deploying to EC2 → confirming application availability and uptime monitoring was completed successfully without errors.

<u>Conclusion:</u>

- ✓ In this project, we successfully deployed a React application to a production-ready environment using modern DevOps practices. The application was containerized using Docker and managed with Docker Compose.
- ✓ Custom bash scripts were written to automate the build and deployment process. The entire source code was version-controlled using Git and pushed to GitHub via CLI, following a branch-based strategy.
- ✓ Docker images were built and pushed to both public and private repositories on Docker Hub, depending on the environment (dev or prod). Jenkins was configured to automate the CI/CD pipeline, enabling auto-builds on code pushes and merges.
- ✓ The application was hosted on an AWS EC2 instance with appropriate security configurations, and an open-source monitoring tool was set up to ensure uptime and send alerts in case of failures.
- ✓ This comprehensive deployment workflow demonstrates the implementation of key DevOps principles, ensuring scalability, automation, and reliability in delivering web applications.

************** **TASK COMPLETED** **************