APPLICATION DEPLOYMENT - TRENDSTORE

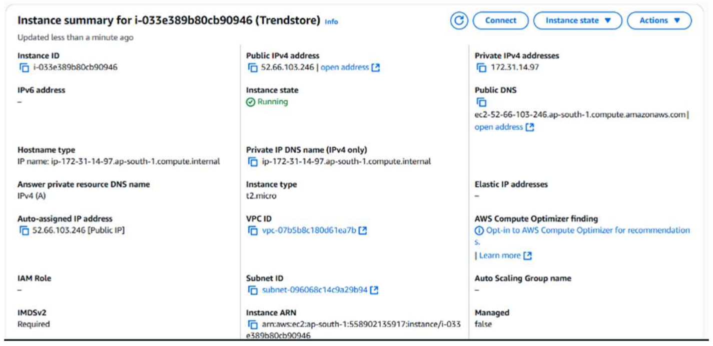
TASKS

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-033e389b80cb90946 (Trendstore)
Updated less than a minute ago

Instance ID i-033e389b80cb90946	Public IPv4 address 52.66.103.246 | open address	Private IPv4 addresses 172.31.14.97
IPv6 address –	Instance state Running	Public DNS ec2-52-66-103-246.ap-south-1.compute.amazonaws.com | open address
Hostname type IP name: ip-172-31-14-97.ap-south-1.compute.internal	Private IP DNS name (IPv4 only) ip-172-31-14-97.ap-south-1.compute.internal	
Answer private resource DNS name IPv4 (A)	Instance type t2.micro	Elastic IP addresses –
Auto-assigned IP address 52.66.103.246 [Public IP]	VPC ID vpc-07b5b8c180d61ea7b	AWS Compute Optimizer finding Opt-in to AWS Compute Optimizer for recommendations. | Learn more
IAM Role –	Subnet ID subnet-096068c14c9a29b94	Auto Scaling Group name –
IMDSv2 Required	Instance ARN arn:aws:ec2:ap-south-1:558902135917:instance/i-033e389b80cb90946	Managed false

1. Production-Grade Deployment of React Application Using Jenkins, DockerHub, and AWS Kubernetes

Application:

Step 1: Clone the React Application.

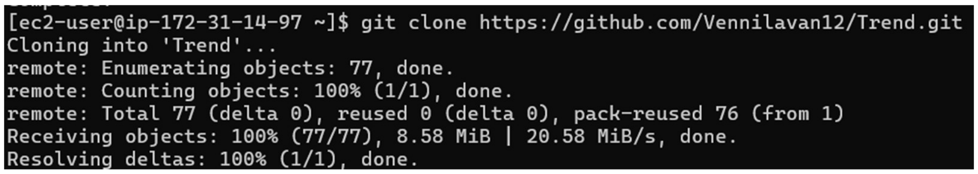
✓ To begin the deployment process, the first step is to obtain the source code of the React application.

✓ The code is publicly available in a GitHub repository. We use the git clone command to download the project to our local system.

✓ Command:

- **git clone https://github.com/Vennilavan12/Trend.git**
- **cd Trend**

✓ This command clones the repository named **Trend** from GitHub and navigates into the project directory.

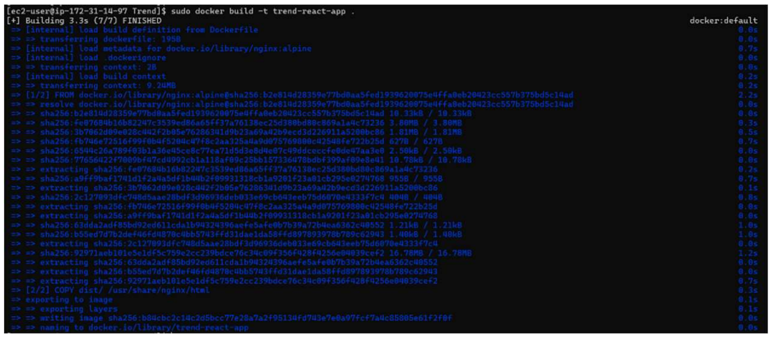


```
[ec2-user@ip-172-31-14-97 ~]$ git clone https://github.com/Vennilavan12/Trend.git
Cloning into 'Trend'...
remote: Enumerating objects: 77, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 77 (delta 0), reused 0 (delta 0), pack-reused 76 (from 1)
Receiving objects: 100% (77/77), 8.58 MiB | 20.58 MiB/s, done.
Resolving deltas: 100% (1/1), done.
```

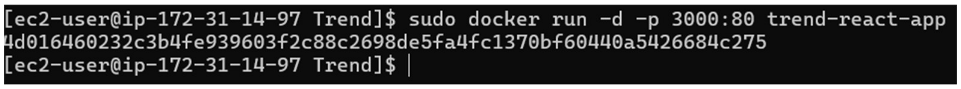
- ✓ The application files will be used in the following steps to create a Docker image, push it to DockerHub, and deploy it on AWS infrastructure.

Step 2: Run the Application Locally on Port 3000

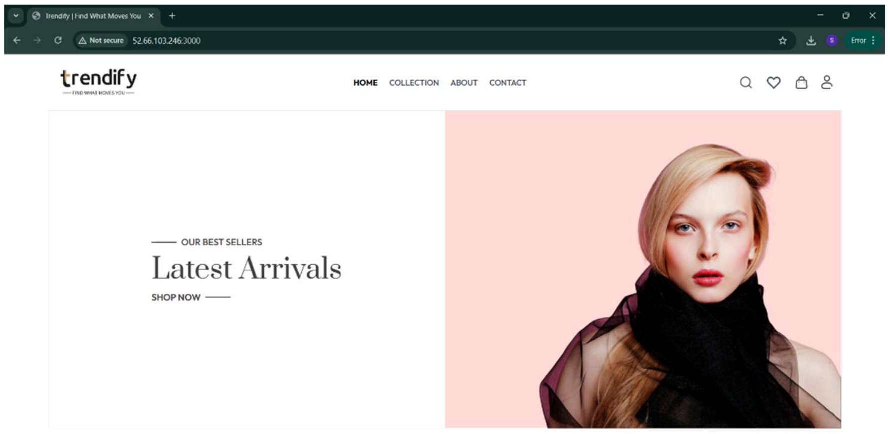
- ✓ Before proceeding with containerization and deployment, it is essential to verify that the React application runs successfully in the local environment.
- ✓ After cloning the repository, we navigate into the project directory and install the required dependencies using the **npm install** command.
- ✓ Once the installation is complete, the application is started using the **npm start** command, which by default runs the React development server on **port 3000**.
- ✓ Upon successful execution, the application becomes accessible via **http://localhost:3000** in a web browser.
- ✓ This step ensures that the source code is functional and ready for further steps like Dockerization and deployment.



```
[ec2-user@ip-172-31-14-97 Trend]$ sudo docker build -t trend-react-app .
[+] Building 3.3s (7/7) FINISHED                                                                                    docker:default
 => [internal] load build definition from Dockerfile                                                                          0.0s
 => => transferring dockerfile: 195B                                                                                          0.0s
 => [internal] load metadata for docker.io/library/nginx:alpine                                                               0.7s
 => [internal] load .dockerignore                                                                                             0.0s
 => => transferring context: 2B                                                                                               0.0s
 => [internal] load build context                                                                                             0.2s
 => => transferring context: 9.24MB                                                                                           0.2s
 => [1/2] FROM docker.io/library/nginx:alpine@sha256:b2e814d28359e77bd0aa5fed1939620075e4ffa0eb20423cc557b375bd5c14ad         2.2s
 => => resolve docker.io/library/nginx:alpine@sha256:b2e814d28359e77bd0aa5fed1939620075e4ffa0eb20423cc557b375bd5c14ad         0.0s
 => => sha256:b2e814d28359e77bd0aa5fed1939620075e4ffa0eb20423cc557b375bd5c14ad 10.33kB / 10.33kB                              0.0s
 => => sha256:fe07684b16b82247c3539ed86a65ff37a76138ec25d380bd80c869a1a4c73236 3.80MB / 3.80MB                                0.3s
 => => sha256:3b7062d09e028c442f2b05e76286341d9b23a69a42b9ecd3d226911a5200bc86 1.81MB / 1.81MB                                0.5s
 => => sha256:fb746e72516f99f0b4f5204c47f8c2aa325a4a9d075769800c42548fe722b25d 627B / 627B                                    0.7s
 => => sha256:6544c26a789f03b1a36e45ce8c77ea71d5d3e8d4e07c49ddceccfe0de47aa3e0 2.50kB / 2.50kB                                0.0s
 => => sha256:77656422f7009bf47cd4992cb1a118af09c25bb157336478bdbf399af09e8e41 10.78kB / 10.78kB                              0.0s
 => => extracting sha256:fe07684b16b82247c3539ed86a65ff37a76138ec25d380bd80c869a1a4c73236                                     0.2s
 => => sha256:a9ff9baf1741d1f2a4a5df1b44b2f09931318cb1a9201f23a01cb295e0274768 955B / 955B                                    0.7s
 => => extracting sha256:3b7062d09e028c442f2b05e76286341d9b23a69a42b9ecd3d226911a5200bc86                                     0.1s
 => => sha256:2c127093dfc748d5aae28bdf3d96936deb033e69cb643eeb75d6070e4333f7c4 404B / 404B                                    0.8s
 => => extracting sha256:fb746e72516f99f0b4f5204c47f8c2aa325a4a9d075769800c42548fe722b25d                                     0.0s
 => => extracting sha256:a9ff9baf1741d1f2a4a5df1b44b2f09931318cb1a9201f23a01cb295e0274768                                     0.0s
 => => sha256:63dda2adf85bd92ed611cda1b94324396aefe5afe0b7b39a72b4ea6362c40552 1.21kB / 1.21kB                                1.0s
 => => sha256:b55ed7d7b2def46fd4878c4bb5743ffd31dae1da58ffd897893978b789c62943 1.40kB / 1.40kB                                1.0s
 => => extracting sha256:2c127093dfc748d5aae28bdf3d96936deb033e69cb643eeb75d6070e4333f7c4                                     0.0s
 => => sha256:92971aeb101e5e1df5c759e2cc239bdce76c34c09f356f428f4256e04039cef2 16.78MB / 16.78MB                              1.2s
 => => extracting sha256:63dda2adf85bd92ed611cda1b94324396aefe5afe0b7b39a72b4ea6362c40552                                     0.0s
 => => extracting sha256:b55ed7d7b2def46fd4878c4bb5743ffd31dae1da58ffd897893978b789c62943                                     0.0s
 => => extracting sha256:92971aeb101e5e1df5c759e2cc239bdce76c34c09f356f428f4256e04039cef2                                     0.7s
 => [2/2] COPY dist/ /usr/share/nginx/html                                                                                    0.3s
 => exporting to image                                                                                                        0.1s
 => => exporting layers                                                                                                       0.1s
 => => writing image sha256:b84cbc2c14c2d5bcc77e28a7a2f95134fd743e7e0a97fcf7a4c85805e61f2f0f                                  0.0s
 => => naming to docker.io/library/trend-react-app                                                                            0.0s
```



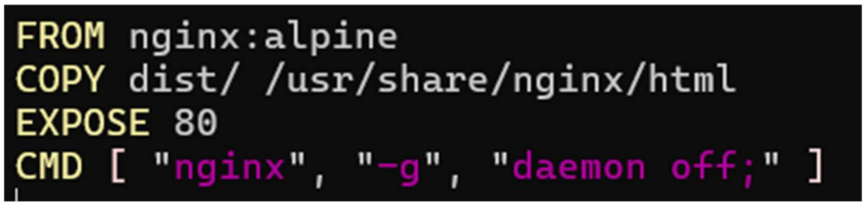
```
[ec2-user@ip-172-31-14-97 Trend]$ sudo docker run -d -p 3000:80 trend-react-app
4d016460232c3b4fe939603f2c88c2698de5fa4fc1370bf60440a5426684c275
[ec2-user@ip-172-31-14-97 Trend]$ |
```



Docker:

Step 1: Dockerize the Application

- ✓ In this step, we containerize the React application using Docker by creating a Dockerfile in the root directory of the project.
- ✓ The Dockerfile uses a multi-stage build where the first stage installs dependencies and builds the React application using a Node.js base image, and the second stage uses an Nginx base image to serve the static files generated during the build.
- ✓ This method ensures that the final Docker image is efficient, production-ready, and contains only the necessary files for serving the application.
- ✓ After creating the Dockerfile, we built the Docker image using the docker build command and tagged the image as trend-app.
- ✓ Once the image was built successfully, we ran the Docker container using the docker run command and mapped port 80 inside the container to port 3000 on the host machine.
- ✓ This confirmed that the Dockerized application runs correctly and is accessible through port 3000 in a production-like environment.



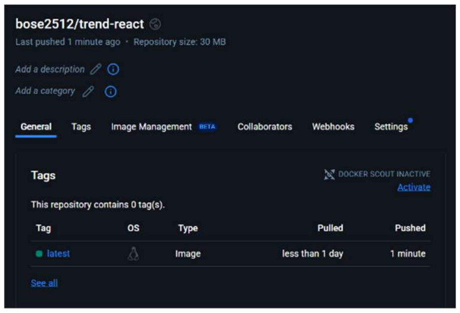
```
FROM nginx:alpine
COPY dist/ /usr/share/nginx/html
EXPOSE 80
CMD [ "nginx", "-g", "daemon off;" ]
```

DockerHub:

Step 1: Push Docker Image to DockerHub

- ✓ In this step, we pushed the Docker image of our React application to DockerHub to make it accessible for deployment in Kubernetes.
- ✓ **Created a DockerHub Repository**

  First, we logged into our DockerHub account and created a new public repository named trend-react (or any custom name).



bose2512/trend-react

Last pushed 1 minute ago · Repository size: 30 MB

Add a description

Add a category

General | Tags | Image Management BETA | Collaborators | Webhooks | Settings

Tags

DOCKER SCOUT INACTIVE
Activate

This repository contains 0 tag(s).

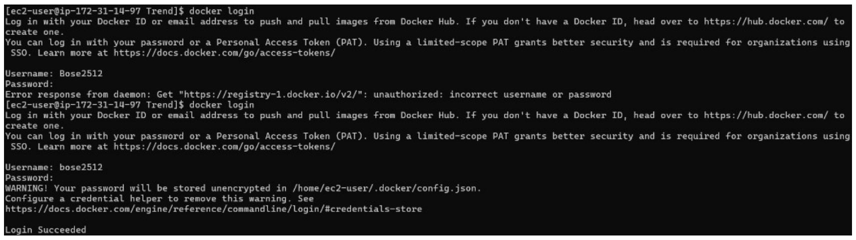
Tag	OS	Type	Pulled	Pushed
latest		Image	less than 1 day	1 minute

See all

- ✓ **Tagged the Docker Image**

  After successfully building the Docker image locally, we tagged the image using the following command to match the DockerHub repository format:
- ✓ **Logged in to DockerHub via CLI**

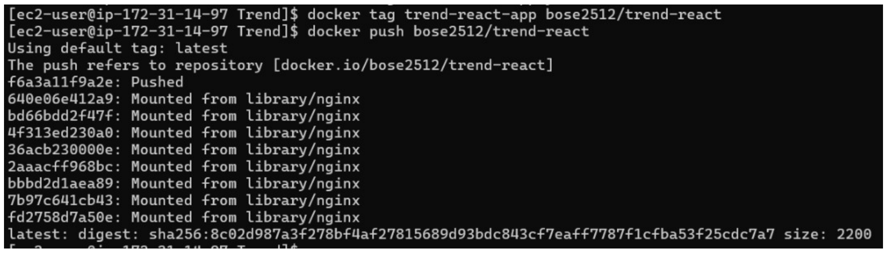
  We used the Docker CLI to authenticate with DockerHub:
- ✓ We entered the DockerHub username and password when prompted.



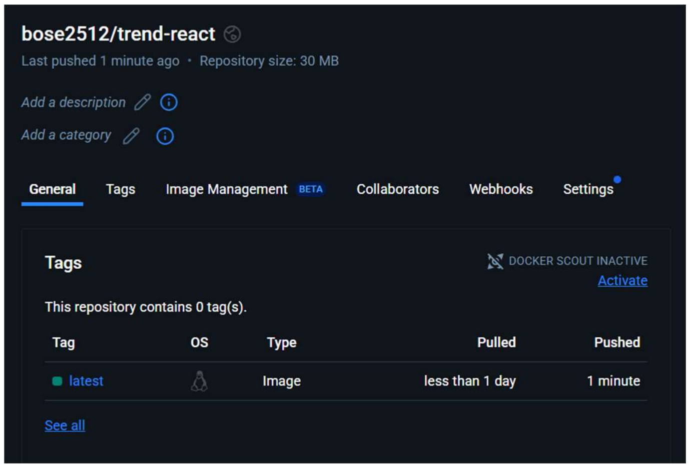
```
[ec2-user@ip-172-31-14-97 Trend]$ docker login
Log in with your Docker ID or email address to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com/ to
create one.
You can log in with your password or a Personal Access Token (PAT). Using a limited-scope PAT grants better security and is required for organizations using
 SSO. Learn more at https://docs.docker.com/go/access-tokens/

Username: Bose2512
Password:
Error response from daemon: Get "https://registry-1.docker.io/v2/": unauthorized: incorrect username or password
[ec2-user@ip-172-31-14-97 Trend]$ 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: bose2512
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
```

- ✓ **Pushed the Docker Image**

  Once authenticated, we pushed the tagged image to DockerHub:
- ✓ After completing this step, the image was available on DockerHub and ready to be pulled from any environment, such as Kubernetes during the deployment phase.



```
[ec2-user@ip-172-31-14-97 Trend]$ docker tag trend-react-app bose2512/trend-react
[ec2-user@ip-172-31-14-97 Trend]$ docker push bose2512/trend-react
Using default tag: latest
The push refers to repository [docker.io/bose2512/trend-react]
f6a3a11f9a2e: Pushed
640e06e412a9: Mounted from library/nginx
bd66bdd2f47f: Mounted from library/nginx
4f313ed230a0: Mounted from library/nginx
36acb230000e: Mounted from library/nginx
2aaacff968bc: Mounted from library/nginx
bbbd2d1aea89: Mounted from library/nginx
7b97c641cb43: Mounted from library/nginx
fd2758d7a50e: Mounted from library/nginx
latest: digest: sha256:8c02d987a3f278bf4af27815689d93bdc843cf7eaff7787f1cfba53f25cdc7a7 size: 2200
```



bose2512/trend-react

Last pushed 1 minute ago · Repository size: 30 MB

Tags

This repository contains 0 tag(s).

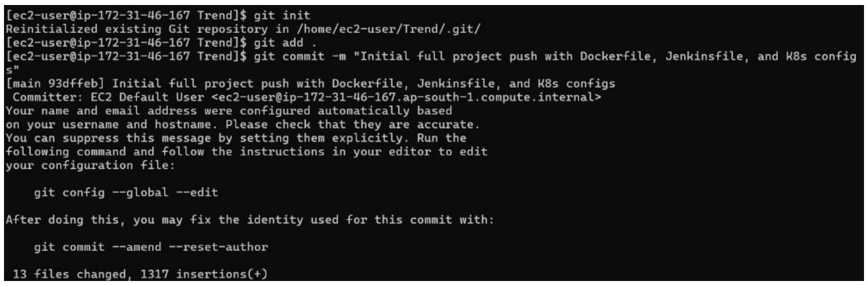
Tag	OS	Type	Pulled	Pushed
latest		Image	less than 1 day	1 minute

<u>Version Control:</u>

Step 1: Push the Project Code to GitHub

- ✓ To manage and track changes in the application source code, we used GitHub as the version control system.

- ✓ First, we initialized a local Git repository inside the project directory using the git init command.
- ✓ We then added all the files using git add . and committed them with an appropriate message using git commit -m "Initial commit".
- ✓ Next, we created a new repository on GitHub (for example, Trend) and linked our local repository to the remote repository using the command:
- ✓ We then pushed the code to the GitHub repository using:
- ✓ To ensure that unnecessary files are not tracked by Git, we created a .gitignore file and added entries such as node_modules, build, and .env which should be excluded from version control.
- ✓ Additionally, we created a .dockerignore file to improve the Docker build process by ignoring unnecessary files and directories like node_modules, .git, and Dockerfile itself. This helps in optimizing the Docker image size and build time.
- ✓ These ignore files play a crucial role in managing clean and efficient code and image builds.



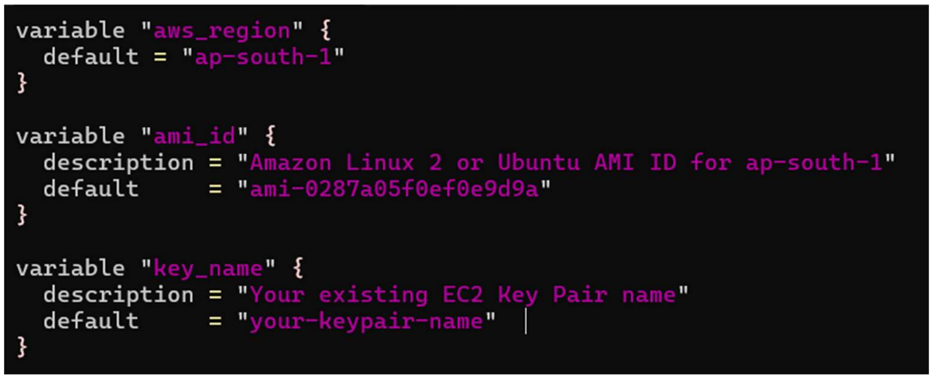
```
[ec2-user@ip-172-31-46-167 Trend]$ git init
Reinitialized existing Git repository in /home/ec2-user/Trend/.git/
[ec2-user@ip-172-31-46-167 Trend]$ git add .
[ec2-user@ip-172-31-46-167 Trend]$ git commit -m "Initial full project push with Dockerfile, Jenkinsfile, and K8s config
s"
[main 93dffeb] Initial full project push with Dockerfile, Jenkinsfile, and K8s configs
 Committer: EC2 Default User <ec2-user@ip-172-31-46-167.ap-south-1.compute.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

 13 files changed, 1317 insertions(+)
```

<u>Terraform:</u>

Step 1: Setup Infrastructure with Terraform

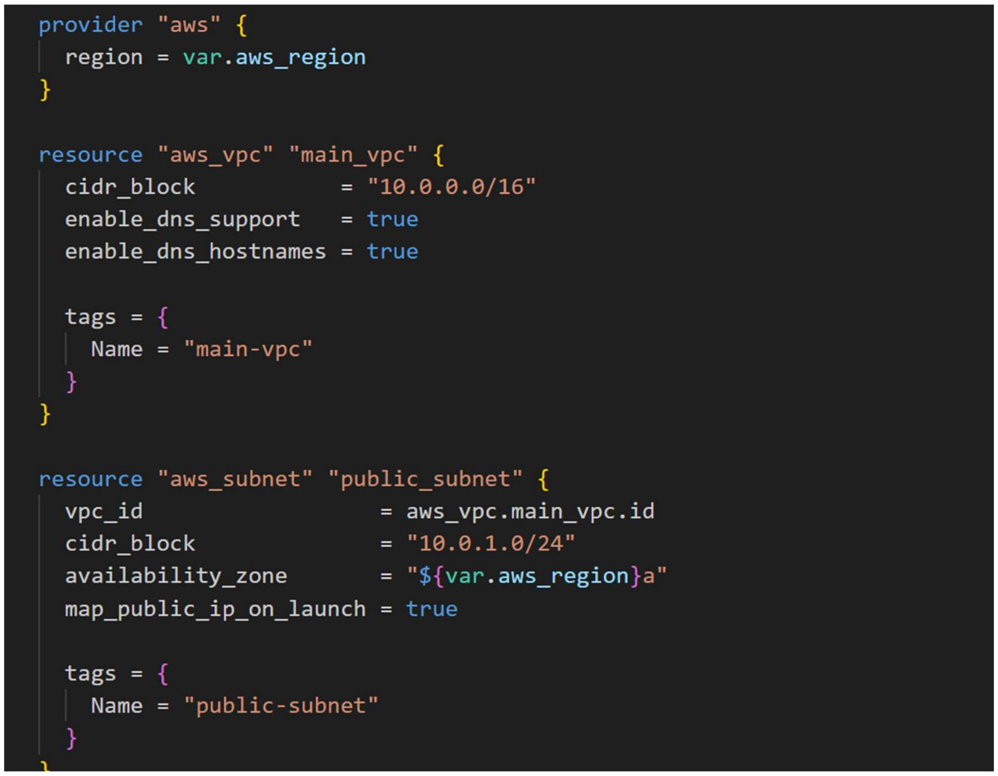
- ✓ In this step, we define and provision the cloud infrastructure required to deploy our application using Terraform. Terraform is an Infrastructure as Code (IaC) tool that allows us to create, manage, and update AWS resources programmatically.
- ✓ We start by creating a main.tf file that contains configurations to launch essential resources such as a Virtual Private Cloud (VPC), subnets, security groups, an IAM role, and an EC2 instance that will host Jenkins.
- ✓ This setup ensures a secure and isolated environment for our CI/CD pipeline and application deployment.
- ✓ After writing the Terraform configuration, we initialize Terraform using the terraform init command, which prepares the working directory and downloads required provider plugins.

- ✓ We then run terraform plan to review the resources that will be created, followed by terraform apply to provision the infrastructure.
- ✓ Once applied successfully, an EC2 instance is created with Jenkins pre-installed using the user data script.
- ✓ This instance acts as the automation server for building and deploying our Dockerized React application. The VPC and its networking components provide the necessary connectivity and isolation for the application stack.
- ✓ This automated approach using Terraform significantly reduces manual setup effort and ensures consistency and repeatability across environments.
- ✓ **Variables.tf:**



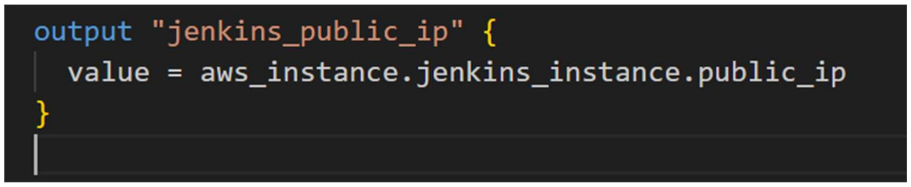
```
variable "aws_region" {
  default = "ap-south-1"
}

variable "ami_id" {
  description = "Amazon Linux 2 or Ubuntu AMI ID for ap-south-1"
  default     = "ami-0287a05f0ef0e9d9a"
}

variable "key_name" {
  description = "Your existing EC2 Key Pair name"
  default     = "your-keypair-name"
}
```

- ✓ **Main.tf:**

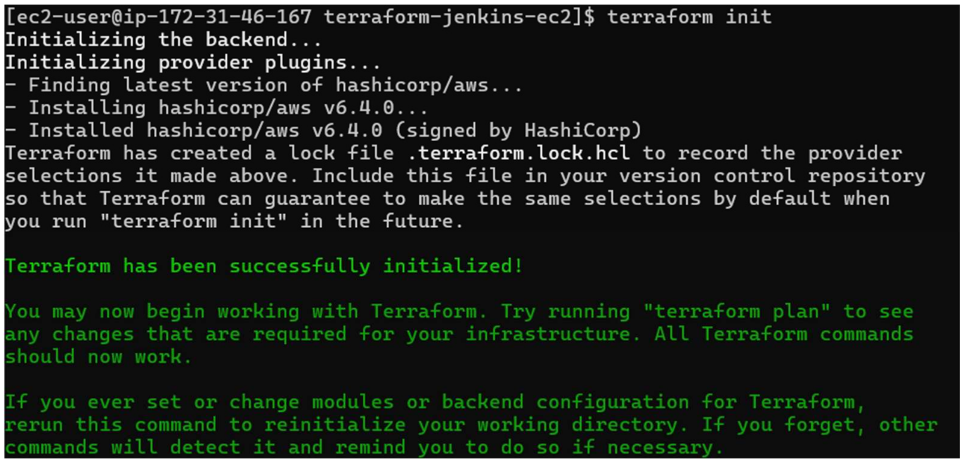


```
provider "aws" {
  region = var.aws_region
}

resource "aws_vpc" "main_vpc" {
  cidr_block           = "10.0.0.0/16"
  enable_dns_support   = true
  enable_dns_hostnames = true

  tags = {
    Name = "main-vpc"
  }
}

resource "aws_subnet" "public_subnet" {
  vpc_id                  = aws_vpc.main_vpc.id
  cidr_block              = "10.0.1.0/24"
  availability_zone       = "${var.aws_region}a"
  map_public_ip_on_launch = true

  tags = {
    Name = "public-subnet"
  }
```

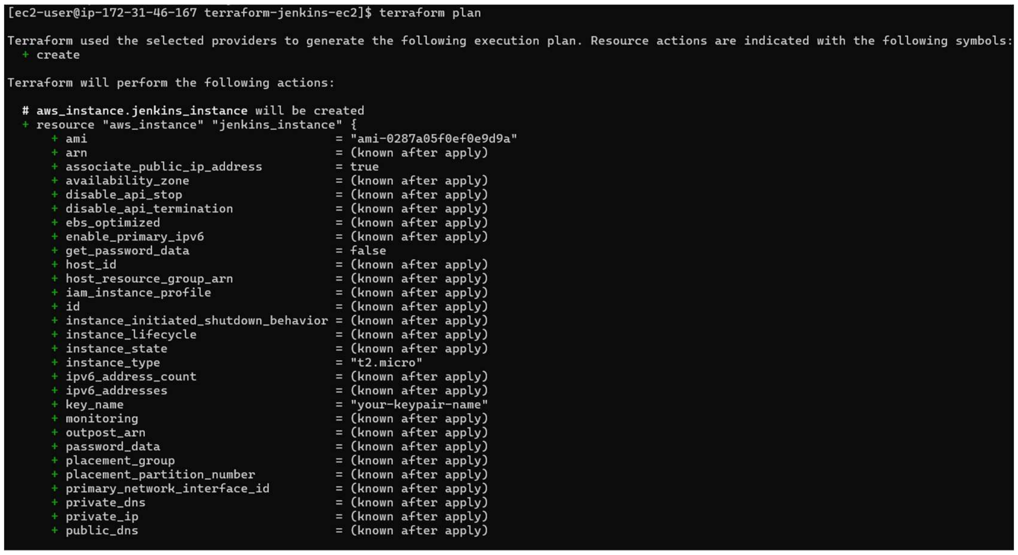
- ✓ Output.tf:



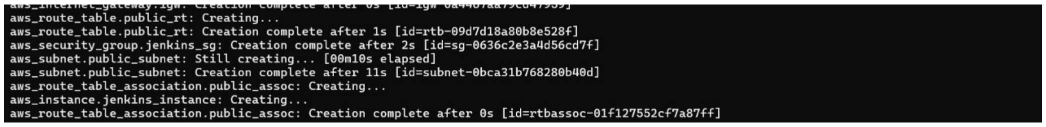
```
output "jenkins_public_ip" {
  value = aws_instance.jenkins_instance.public_ip
}
```



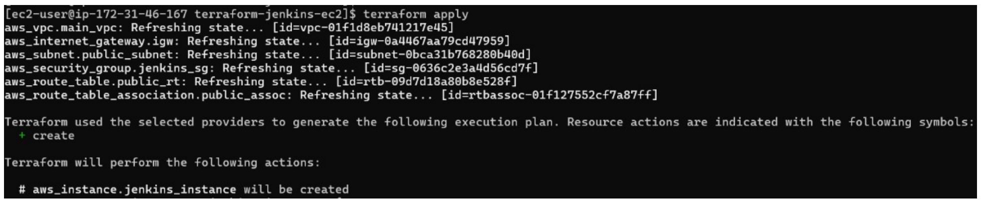
```
[ec2-user@ip-172-31-46-167 terraform-jenkins-ec2]$ terraform init
Initializing the backend...
Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v6.4.0...
- Installed hashicorp/aws v6.4.0 (signed by HashiCorp)
Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```



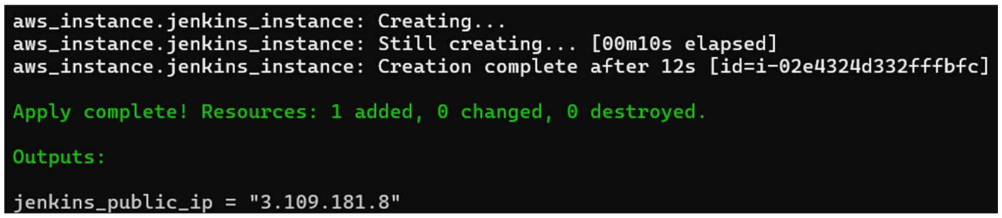
```
[ec2-user@ip-172-31-46-167 terraform-jenkins-ec2]$ terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.jenkins_instance will be created
  + resource "aws_instance" "jenkins_instance" {
      + ami                                  = "ami-0287a05f0ef0e9d9a"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = true
      + availability_zone                    = (known after apply)
      + disable_api_stop                     = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
      + enable_primary_ipv6                  = (known after apply)
      + get_password_data                    = false
      + host_id                              = (known after apply)
      + host_resource_group_arn              = (known after apply)
      + iam_instance_profile                 = (known after apply)
      + id                                   = (known after apply)
      + instance_initiated_shutdown_behavior = (known after apply)
      + instance_lifecycle                   = (known after apply)
      + instance_state                       = (known after apply)
      + instance_type                        = "t2.micro"
      + ipv6_address_count                   = (known after apply)
      + ipv6_addresses                       = (known after apply)
      + key_name                             = "your-keypair-name"
      + monitoring                           = (known after apply)
      + outpost_arn                          = (known after apply)
      + password_data                        = (known after apply)
      + placement_group                      = (known after apply)
      + placement_partition_number           = (known after apply)
      + primary_network_interface_id         = (known after apply)
      + private_dns                          = (known after apply)
      + private_ip                           = (known after apply)
      + public_dns                           = (known after apply)
```



```
aws_route_table.public_rt: Creating...
aws_route_table.public_rt: Creation complete after 1s [id=rtb-09d7d18a80b8e528f]
aws_security_group.jenkins_sg: Creation complete after 2s [id=sg-0636c2e3a4d56cd7f]
aws_subnet.public_subnet: Still creating... [00m10s elapsed]
aws_subnet.public_subnet: Creation complete after 11s [id=subnet-0bca31b768280b40d]
aws_route_table_association.public_assoc: Creating...
aws_instance.jenkins_instance: Creating...
aws_route_table_association.public_assoc: Creation complete after 0s [id=rtbassoc-01f127552cf7a87ff]
```



```
[ec2-user@ip-172-31-46-167 terraform-jenkins-ec2]$ terraform apply
aws_vpc.main_vpc: Refreshing state... [id=vpc-01f1d8eb741217e45]
aws_internet_gateway.igw: Refreshing state... [id=igw-0a4467aa79cd47959]
aws_subnet.public_subnet: Refreshing state... [id=subnet-0bca31b768280b40d]
aws_security_group.jenkins_sg: Refreshing state... [id=sg-0636c2e3a4d56cd7f]
aws_route_table.public_rt: Refreshing state... [id=rtb-09d7d18a80b8e528f]
aws_route_table_association.public_assoc: Refreshing state... [id=rtbassoc-01f127552cf7a87ff]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.jenkins_instance will be created
```

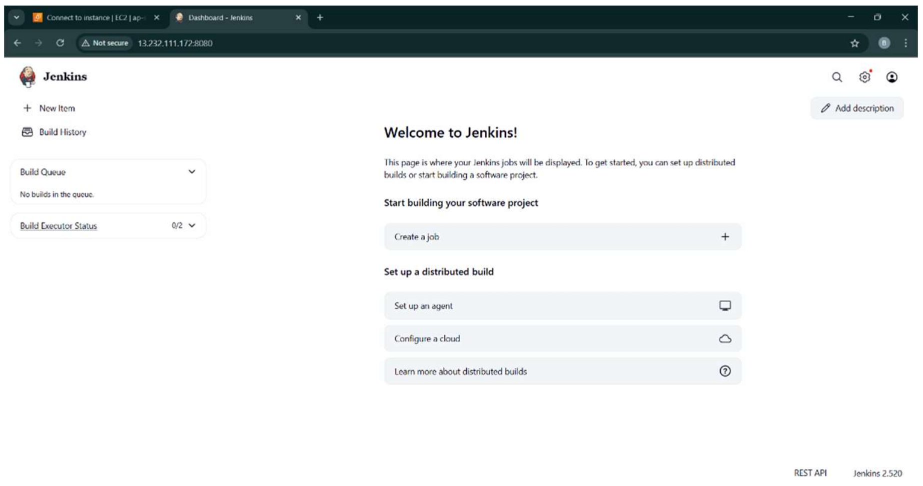


```
aws_instance.jenkins_instance: Creating...
aws_instance.jenkins_instance: Still creating... [00m10s elapsed]
aws_instance.jenkins_instance: Creation complete after 12s [id=i-02e4324d332fffbfc]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

jenkins_public_ip = "3.109.181.8"
```

<u>Jenkins:</u>

Step 1: Install Jenkins on EC2 Instance

- ✓ To configure our CI/CD pipeline, we need to install Jenkins on the EC2 instance provisioned using Terraform. After the EC2 instance is up and running, we connect to it using SSH.
- ✓ Once connected, we begin by updating the system packages using sudo yum update -y. Then, we install Java, which is a prerequisite for Jenkins, by executing sudo yum install java-11-amazon-corretto -y.
- ✓ Next, we add the Jenkins repository to the system using the official Jenkins repo URL and import the required GPG key.
- ✓ Once the repository is added, we install Jenkins by running sudo yum install jenkins -y. After the installation is complete, we start the Jenkins service using sudo systemctl start jenkins and enable it to run on system boot using sudo systemctl enable jenkins.
- ✓ We then adjust the firewall rules or security group settings to allow inbound traffic on port 8080, which is the default port Jenkins runs on.
- ✓ After that, we access Jenkins via http://<EC2-Public-IP>:8080 in a web browser. The initial admin password is retrieved using the command sudo cat /var/lib/jenkins/secrets/initialAdminPassword.


Connect to instance | EC2 | ap-
Dashboard - Jenkins
Not secure 13.232.111.172:8080
Jenkins
New Item
Build History
Build Queue
No builds in the queue.
Build Executor Status 0/2
Add description
Start building your software project
Create a job
Set up a distributed build
Set up an agent
Configure a cloud
Learn more about distributed builds
REST API
Jenkins 2.520

Welcome to Jenkins!

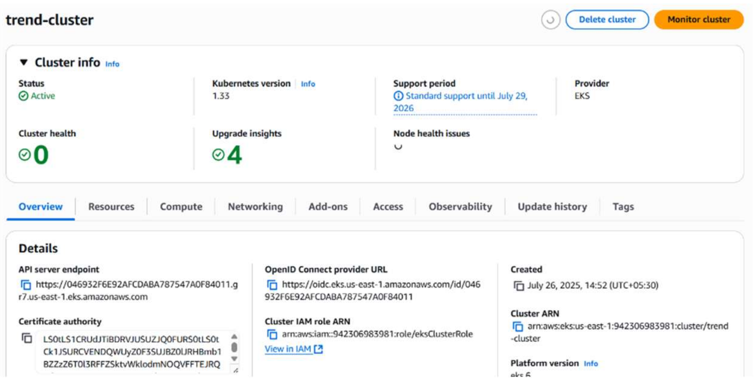
This page is where your Jenkins jobs will be displayed. To get started, you can set up distributed builds or start building a software project.

- ✓ Once logged in, we complete the setup by installing the suggested plugins.
- ✓ Finally, we install essential Jenkins plugins such as **Docker**, **Git**, **Kubernetes**, **GitHub**, and **Pipeline** which are required for building, pushing, and deploying our application as part of the CI/CD workflow.

<u>Kubernetes:</u>

Step 1: Setup Kubernetes Cluster Using AWS EKS

- ✓ To deploy the application on Kubernetes, we set up an Elastic Kubernetes Service (EKS) cluster on AWS. EKS provides a managed Kubernetes service that simplifies the process of running Kubernetes workloads on AWS infrastructure.
  - ✓ First, we used the AWS Management Console or Terraform scripts to create an EKS cluster. The cluster was configured with the necessary roles, networking (VPC and subnets), and worker nodes.



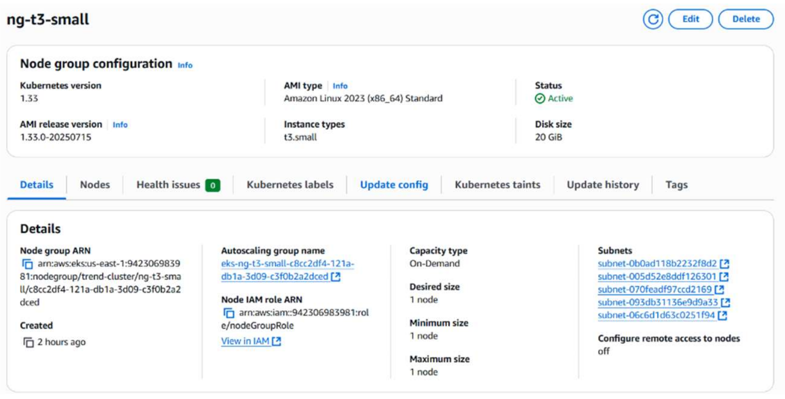
trend-cluster

Cluster info

Status: Active
Kubernetes version: 1.33
Support period: Standard support until July 29, 2026
Provider: EKS
Cluster health: 0
Upgrade insights: 4
Node health issues

Details

API server endpoint: https://046932F6E92AFCDABA787547A0F84011.gr7.us-east-1.eks.amazonaws.com
Certificate authority: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURCVENDQWUyZ0F3SUJBZ0lJRHBmb1BZZzZ6T0l3RFFZSktvWklodmNOQVFFTEJRQ
OpenID Connect provider URL: https://oidc.eks.us-east-1.amazonaws.com/id/046932F6E92AFCDABA787547A0F84011
Cluster IAM role ARN: arn:aws:iam::942306983981:role/eksClusterRole
Created: July 26, 2025, 14:52 (UTC+05:30)
Cluster ARN: arn:aws:eks:us-east-1:942306983981:cluster/trend-cluster
Platform version: eks.6



ng-t3-small

Node group configuration

Kubernetes version: 1.33
AMI type: Amazon Linux 2023 (x86_64) Standard
Status: Active
AMI release version: 1.33.0-20250715
Instance types: t3.small
Disk size: 20 GiB

Details

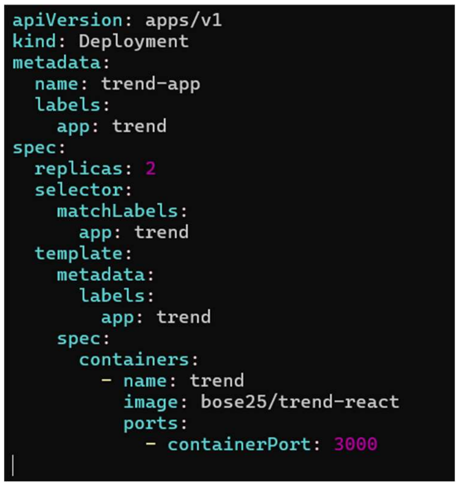
Node group ARN: arn:aws:eks:us-east-1:942306983981:nodegroup/trend-cluster/ng-t3-small/c8cc2df4-121a-db1a-3d09-c3f0b2a2dced
Created: 2 hours ago
Autoscaling group name: eks-ng-t3-small-c8cc2df4-121a-db1a-3d09-c3f0b2a2dced
Node IAM role ARN: arn:aws:iam::942306983981:role/nodeGroupRole
Capacity type: On-Demand
Desired size: 1 node
Minimum size: 1 node
Maximum size: 1 node
Subnets: subnet-0b0ad118b2232f8d2, subnet-005d52e8ddf126301, subnet-070feadf97ccd2169, subnet-093db31136e9d9a33, subnet-06c6d1d63c0251f94
Configure remote access to nodes: off

  - ✓ Once the EKS cluster was created, we configured kubectl (the Kubernetes CLI) on our local machine to communicate with the cluster. We did this by running the following AWS CLI command:
    - **aws ecr get-login-password --region <your-region> | docker login --username AWS --password-stdin <aws_account_id>.dkr.ecr.<your-region>.amazonaws.com**
  - ✓ After successful configuration, we verified the connection to the cluster using:
  - ✓ This command displayed the list of nodes in the EKS cluster, confirming that the cluster was active and accessible.
  - ✓ With the EKS cluster set up and running, we were ready to deploy our application using Kubernetes manifests.

- ✓ This step ensured a production-ready Kubernetes environment was available for deploying the Dockerized React application.

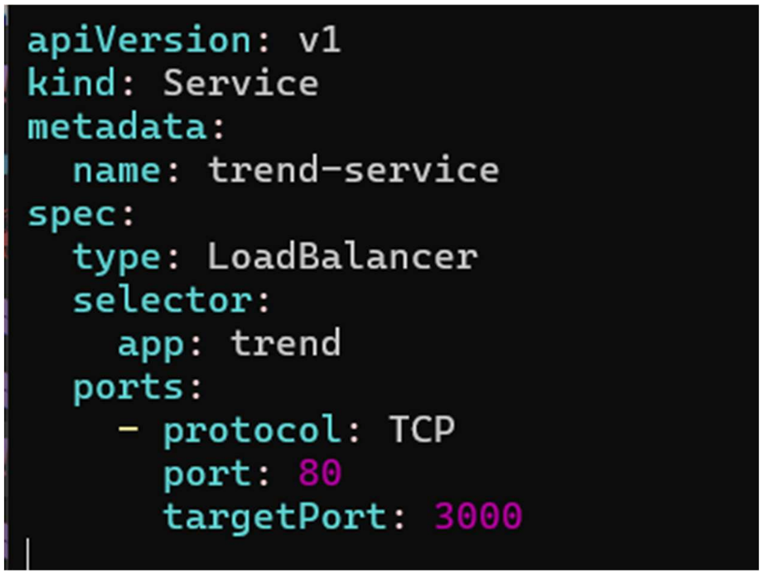
Step 2: Kubernetes Deployment and Service Setup

- ✓ In this step, we deployed the Dockerized React application to an AWS EKS cluster using Kubernetes manifests.
- ✓ The deployment is responsible for managing the application's pods, while the service exposes it to the internet via a LoadBalancer.
- ✓ We began by creating two YAML configuration files: deployment.yaml and service.yaml. The deployment file defines how many replicas (pods) of the application should run, which Docker image to use, and which port the container listens on (port 3000 in this case). We used the Docker image that was previously built and pushed to DockerHub.
- ✓ Below is the content of the deployment.yaml file:
- ✓ This configuration ensures that two replicas of the application pod are always running, providing redundancy and load balancing.
- ✓ Next, we created the service.yaml file to expose the application to the internet. We used the LoadBalancer type, which automatically provisions an AWS ELB and assigns a public DNS endpoint.
- ✓ Below is the content of the service.yaml file:
- ✓ After preparing both files, we deployed them to the Kubernetes cluster using the following kubectl commands:
- ✓ Once deployed, we verified the deployment status and ensured the service was assigned a public IP using:
- ✓ After a few minutes, the application became accessible via the external LoadBalancer URL provided by the service.
- ✓ This step successfully brought the Dockerized React app live on AWS EKS using Kubernetes best practices.
- ✓ **Deployment.yaml**



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: trend-app
  labels:
    app: trend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: trend
  template:
    metadata:
      labels:
        app: trend
    spec:
      containers:
        - name: trend
          image: bose25/trend-react
          ports:
            - containerPort: 3000
```

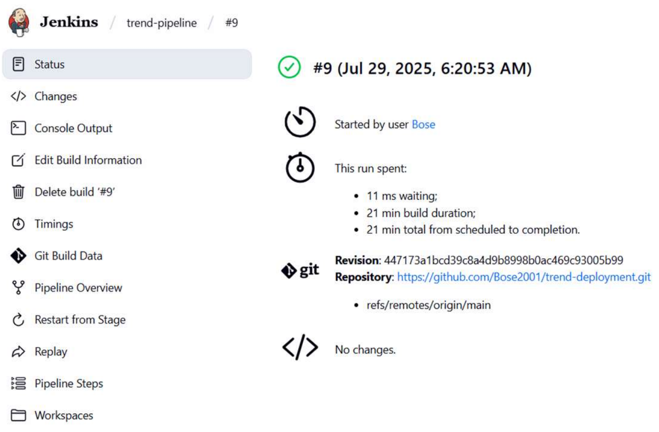
- ✓ **Service.yaml**



```yaml
apiVersion: v1
kind: Service
metadata:
  name: trend-service
spec:
  type: LoadBalancer
  selector:
    app: trend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
```

Step 3: Jenkins CI/CD Pipeline Setup

- ✓ In this step, we set up Jenkins to automate the build, push, and deployment process of the application using a CI/CD pipeline.
- ✓ First, we installed Jenkins on an EC2 instance and ensured that all the required plugins were available. These included the Docker plugin, Git plugin, Kubernetes plugin, Pipeline plugin, and the GitHub Integration plugin.
- ✓ Next, we configured the GitHub repository to trigger Jenkins builds automatically. This was done by creating a webhook in the GitHub repository that points to the Jenkins server's webhook URL (http://<jenkins-server>/github-webhook/). The webhook is triggered every time a new commit is pushed to the repository, initiating the CI/CD pipeline.
- ✓ We then created a new pipeline project in Jenkins. Inside this project, we used a declarative Jenkinsfile that defines the entire pipeline process, including cloning the repository, building the Docker image, pushing the image to DockerHub, and deploying the application to the EKS cluster using kubectl.



#9 (Jul 29, 2025, 6:20:53 AM)

Started by user Bose

This run spent:

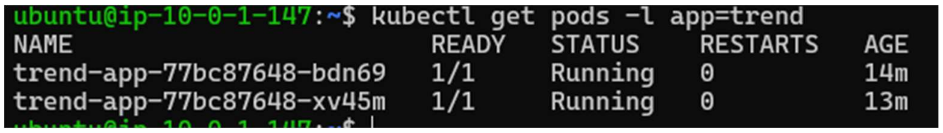
- 11 ms waiting;
- 21 min build duration;
- 21 min total from scheduled to completion.

Revision: 447173a1bcd39c8a4d9b8998b0ac469c93005b99
Repository: https://github.com/Bose2001/trend-deployment.git

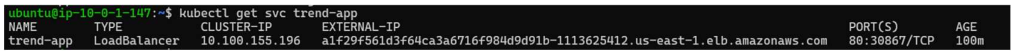
- refs/remotes/origin/main

No changes.

- ✓ The Jenkinsfile includes the following stages:
  - **Clone Repo**: This stage pulls the latest code from the GitHub repository.
  - **Build Docker Image**: This stage builds a new Docker image from the project using the Dockerfile.
  - **Push Docker Image**: In this stage, the image is pushed to DockerHub using credentials securely stored in Jenkins.
  - **Deploy to Kubernetes**: This final stage applies the Kubernetes deployment and service YAML files to the EKS cluster using kubectl.
- ✓ By integrating GitHub and Jenkins, we established an automated pipeline that triggers every time code is committed to the repository.
- ✓ This pipeline builds the application, pushes the updated image to DockerHub, and deploys the new version to Kubernetes, ensuring continuous integration and continuous deployment.



```
ubuntu@ip-10-0-1-147:~$ kubectl get pods -l app=trend
NAME                          READY   STATUS    RESTARTS   AGE
trend-app-77bc87648-bdn69     1/1     Running   0          14m
trend-app-77bc87648-xv45m     1/1     Running   0          13m
```

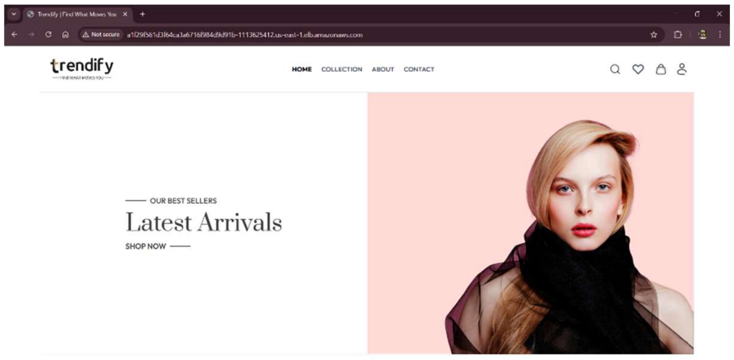


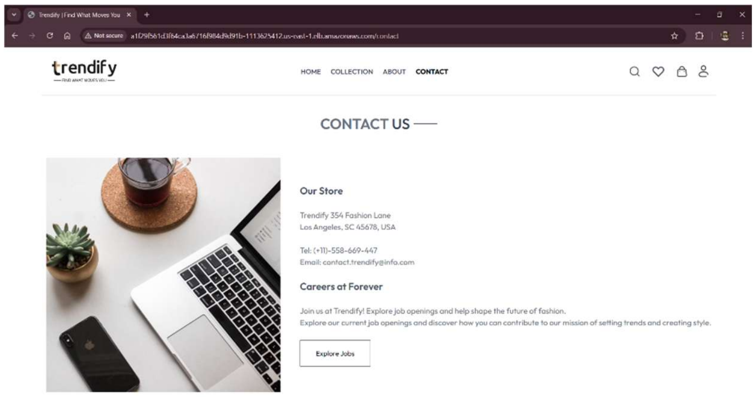
```
ubuntu@ip-10-0-1-147:~$ kubectl get svc trend-app
NAME        TYPE           CLUSTER-IP       EXTERNAL-IP                                                               PORT(S)        AGE
trend-app   LoadBalancer   10.100.155.196   a1f29f561d3f64ca3a6716f984d9d91b-1113625412.us-east-1.elb.amazonaws.com   80:30867/TCP   100m
```

<u>Monitoring:</u>

Step 1: Monitoring the Kubernetes Cluster and Application

- ✓ To ensure the health, performance, and reliability of the deployed React application and the underlying Kubernetes cluster, we implemented a basic monitoring setup using open-source tools.
- ✓ We used Prometheus and Grafana, two widely adopted tools in the DevOps ecosystem, for monitoring purposes. Prometheus is responsible for collecting metrics from Kubernetes components and application pods, while Grafana is used to visualize these metrics using dashboards.
- ✓ We installed Prometheus and Grafana using Helm charts, which simplified the deployment process in the EKS cluster. First, we added the Helm repository for Prometheus:
- ✓ Then, we installed the monitoring stack with the following command:
- ✓ After successful installation, Prometheus started collecting metrics from the cluster, such as CPU usage, memory usage, and pod health. Grafana provided a web-based dashboard interface to monitor these metrics in real time.
- ✓ We accessed the Grafana dashboard by forwarding the Grafana service port to our local machine:
- ✓ This monitoring setup helped us detect issues early, understand resource usage, and ensure smooth operation of the production environment.


Trendify | Find What Moves You
Not secure
a1f29f561d3f64ca3a6716f984d9d91b-1113625412.us-east-1.elb.amazonaws.com
trendify
HOME
COLLECTION
ABOUT
CONTACT
OUR BEST SELLERS
Latest Arrivals
SHOP NOW


Trendify | Find What Moves You
Not secure
a1f29f561d3f64ca3a6716f984d9d91b-1113625412.us-east-1.elb.amazonaws.com/contact
trendify
HOME
COLLECTION
ABOUT
CONTACT
Explore Jobs

CONTACT US

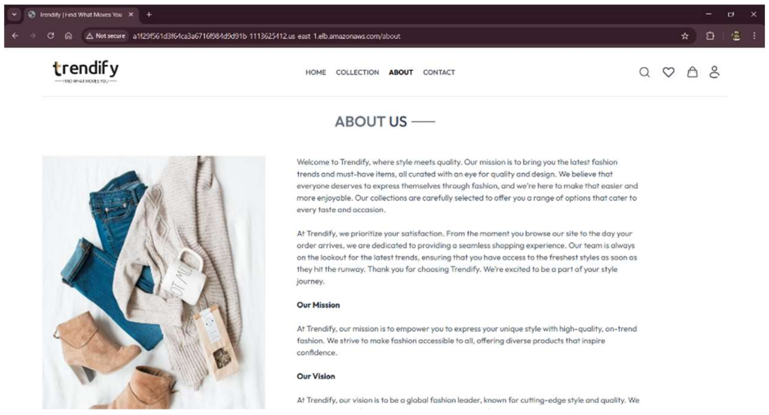
Our Store

Trendify 354 Fashion Lane
Los Angeles, SC 45678, USA

Tel: (+11)-558-669-447
Email: contact.trendify@info.com

Careers at Forever

Join us at Trendify! Explore job openings and help shape the future of fashion.
Explore our current job openings and discover how you can contribute to our mission of setting trends and creating style.


Trendify | Find What Moves You
Not secure
a1f29f561d3f64ca3a6716f984d9d91b-1113625412.us-east-1.elb.amazonaws.com/about
trendify
HOME
COLLECTION
ABOUT
CONTACT

ABOUT US

Welcome to Trendify, where style meets quality. Our mission is to bring you the latest fashion trends and must-have items, all curated with an eye for quality and design. We believe that everyone deserves to express themselves through fashion, and we're here to make that easier and more enjoyable. Our collections are carefully selected to offer you a range of options that cater to every taste and occasion.

At Trendify, we prioritize your satisfaction. From the moment you browse our site to the day your order arrives, we are dedicated to providing a seamless shopping experience. Our team is always on the lookout for the latest trends, ensuring that you have access to the freshest styles as soon as they hit the runway. Thank you for choosing Trendify. We're excited to be a part of your style journey.

Our Mission

At Trendify, our mission is to empower you to express your unique style with high-quality, on-trend fashion. We strive to make fashion accessible to all, offering diverse products that inspire confidence.

Our Vision

At Trendify, our vision is to be a global fashion leader, known for cutting-edge style and quality. We

Conclusion:

- ✓ In this project, we successfully deployed a production-ready React application by containerizing it with Docker, managing infrastructure using Terraform, setting up CI/CD with Jenkins, and deploying it to an EKS cluster on AWS. This end-to-end pipeline ensures automated builds, scalable deployments, and streamlined monitoring, laying the foundation for modern DevOps practices in real-world applications.
- ✓ **Application deployed kubernetes Loadbalancer ARN: http://a1f29f561d3f64ca3a6716f984d9d91b-1113625412.us-east-1.elb.amazonaws.com/**

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