

Cloud Computing Project 2024

Online Shopping Platform

Team Members:

20201700432	SC	عبدالرحمن امام فضل امام
20201701161	SC	محمد محمد خايفه حسن
20191700557	SC	محمد عبدالرؤف احمد عبدالشافى
20191700053	SC	احمد عوض سيد حسن
20191700386	SC	عز خليل عبد الحليم
20191700442	Csys	فوزى احمد فوزى

1. *Introduction:*

This documentation provides a comprehensive overview of a Dockerized E-Commerce Web Application project. The project aims to containerize a full-stack web application using Docker containers, making it portable and easy to deploy across different environments. It utilizes various Dockerfiles, Docker Compose files, Kubernetes deployment configurations, Jenkinsfile for CI/CD, along with ConfigMaps and Secrets for managing configurations and sensitive data.

Purpose:

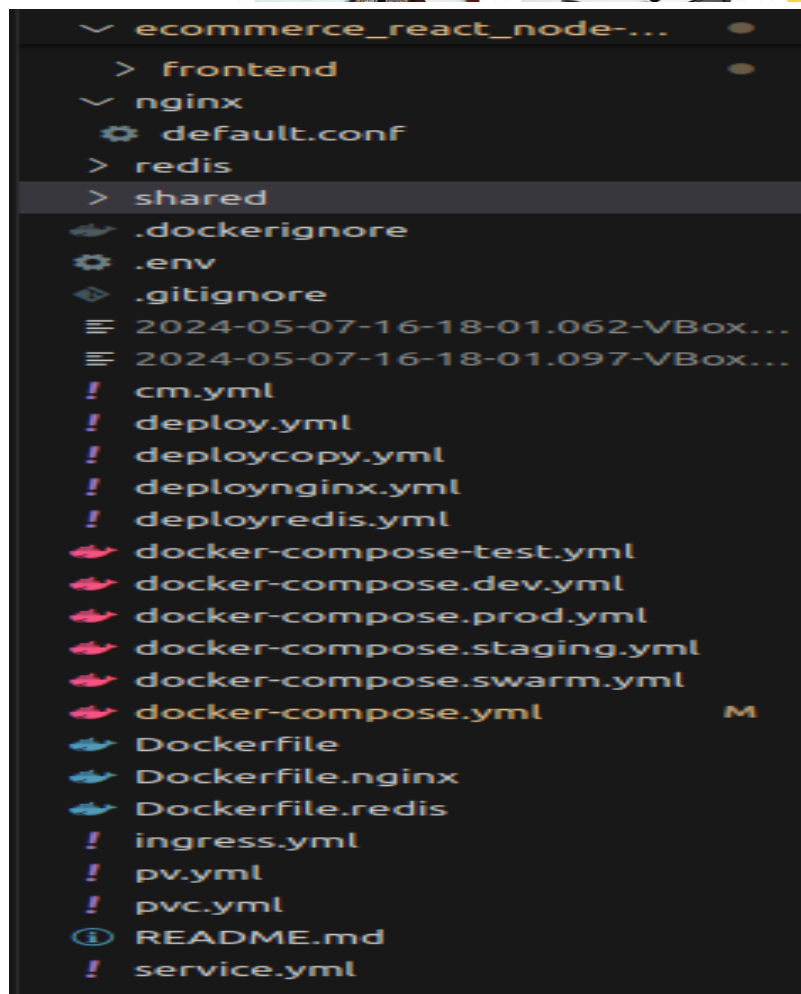
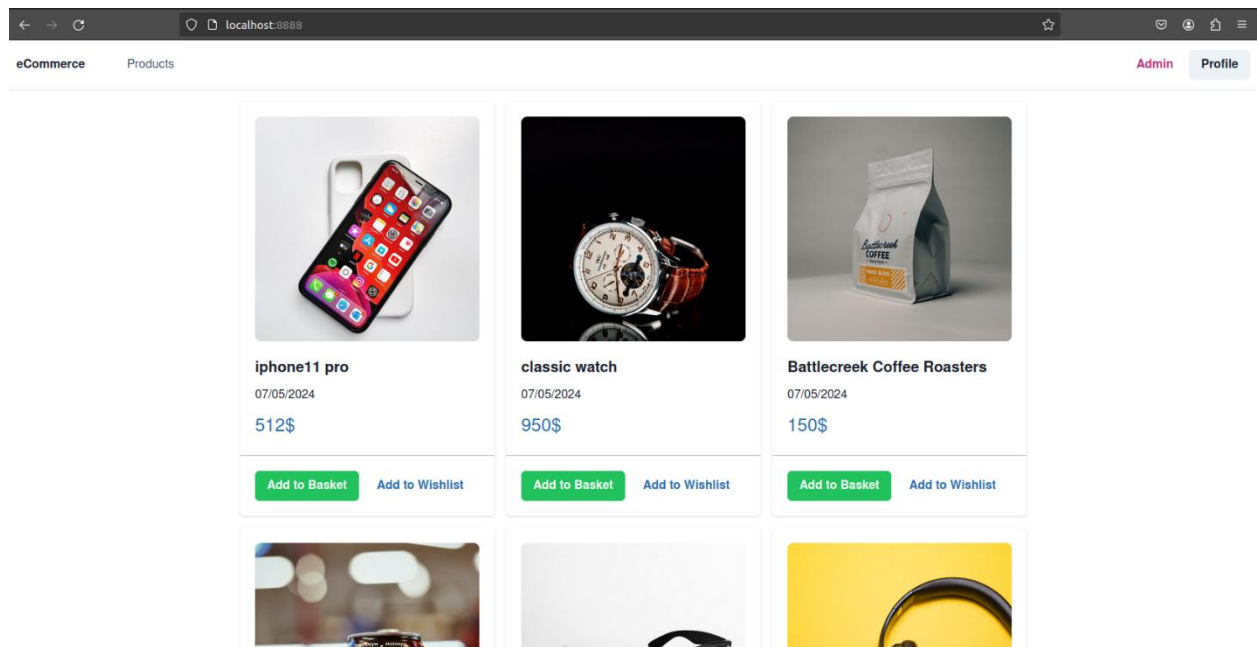
The purpose of this project is to demonstrate how to containerize and deploy an application using Docker and Kubernetes. By containerizing the application, it becomes easier to manage dependencies, scale components, and deploy consistently across different environments.

2. *Project Structure:*

The project follows a modular structure, Here's an overview of the main components:

- **Backend:** Node.js application handling server-side logic. With its dependencies in package.json file and with important env variables to connect to MongoDB and other purposes, you can reach it from port number 8000
- **Frontend:** React.js application responsible for the client-side interface with its dependencies in package.json, you can reach it from port number 3000
- **NGINX:** Acts as a reverse proxy and handles routing to the backend and frontend services.
- **Redis:** This is an optional in-memory data store used for caching or other application-specific purposes. but in our project we just use it in simple function (store hostname that come from container that hold backend).

- Configuration Files: Dockerfiles for building images, Docker Compose files for orchestrating containers, Kubernetes deployment files, ConfigMaps, and Secrets.



So our Technology Stack:

- Backend: Node.js, Express.js, MongoDB
- Frontend: React.js, Axios
- Containerization: Docker
- Orchestration: Kubernetes
- CI/CD: Jenkins

3. Development Environment Setup:

-Prerequisites

Before setting up the development environment, ensure that the following prerequisites are met:

Node.js installed

Docker installed

Kubernetes cluster configured (e.g., Minikube)

-Local Development Setup

To set up the development environment locally:

Clone the project repository (our SCM repo:

<https://github.com/Abdomuller11/Dockerize-ECommerce-web-app.git>).

Navigate to the project directory.

Install dependencies for both frontend and backend using npm install.

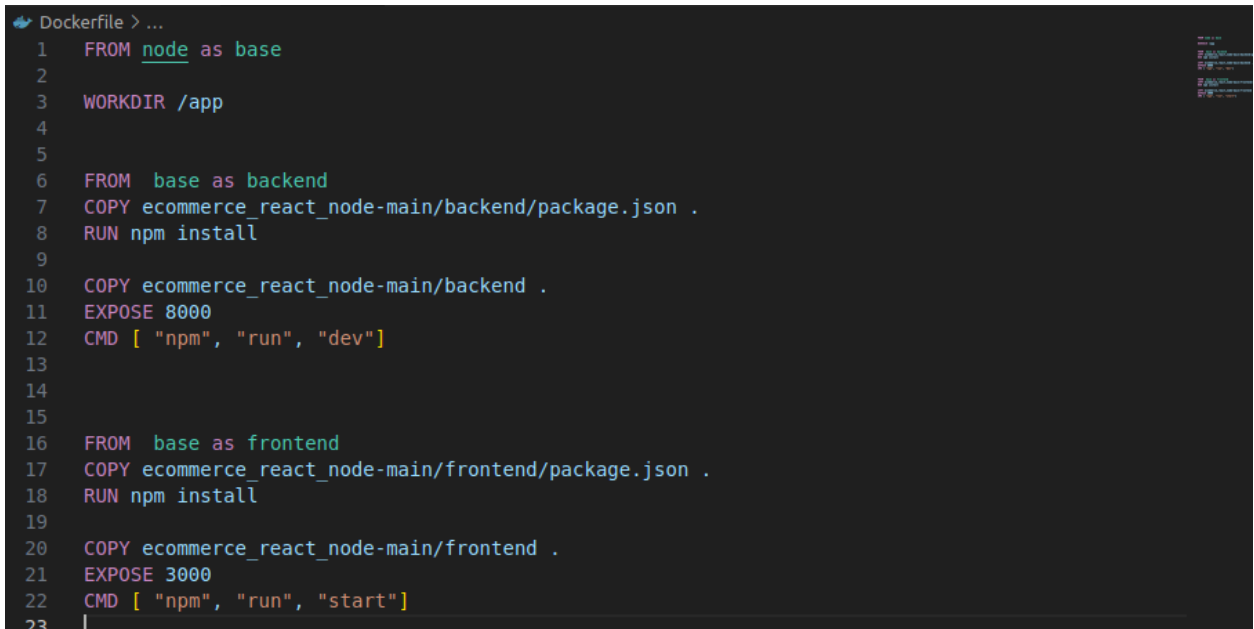
Run the backend server using npm run dev.

Run the frontend server using npm start.

4. Docker Configuration:

***First requirement** in FCIS cloud project is containerization with Docker, The Dockerfile defines the configuration for building Docker images for both frontend and backend components. It includes the necessary steps to install dependencies, copy source code, and expose ports.

We have 2 necessary Dockerfiles , 'Dockerfile' file is multistaging for backend and frontend



```
Dockerfile > ...
1  FROM node as base
2
3  WORKDIR /app
4
5
6  FROM base as backend
7  COPY ecommerce_react_node-main/backend/package.json .
8  RUN npm install
9
10 COPY ecommerce_react_node-main/backend .
11 EXPOSE 8000
12 CMD [ "npm", "run", "dev" ]
13
14
15
16 FROM base as frontend
17 COPY ecommerce_react_node-main/frontend/package.json .
18 RUN npm install
19
20 COPY ecommerce_react_node-main/frontend .
21 EXPOSE 3000
22 CMD [ "npm", "run", "start" ]
23
```

I will explain backend , same thing for frontend

1. FROM node as base: This line sets the base image for the Dockerfile as the official Node.js image. It also assigns an alias "base" to this stage.
2. WORKDIR /app: This line sets the working directory inside the Docker container to /app. All subsequent commands will be executed relative to this directory.
3. FROM base as backend: This line creates a new stage in the Dockerfile based on the "base" stage defined earlier. It creates a separate image for the backend component.
4. COPY ecommerce_react_node-main/backend/package.json .: This line copies the package.json file from the local directory ecommerce_react_node-main/backend/ into the Docker container's current working directory (/app). This step allows Docker to cache the npm dependencies installation for faster builds, I do this step separated from '6' to separate changes that happen to src code and packages to utilize caching benefits when build .

5. RUN npm install: This line installs the dependencies specified in the package.json file using the npm package manager.
6. COPY ecommerce_react_node-main/backend .: This line copies the entire backend application code from the local directory ecommerce_react_node-main/backend/ into the Docker container's current working directory (/app). This step includes all the source code files needed for the backend application to run.
7. EXPOSE 8000: This line exposes port 8000 on the Docker container, allowing external connections to the backend application running inside the container.
8. CMD ["npm", "run", "dev"]: This line specifies the command to run when the Docker container starts. In this case, it runs the backend application in development mode using the npm run dev command.

And we have Dockerfile.nginx for nginx to copy configuration file into nginx container

```
nginx > default.conf
1  upstream front0 {
2      server front0:3000;
3  }
4
5  upstream back0 {
6      server back0:8000;
7  }
8  server {
9      listen 80;
10     location / {
11         proxy_pass http://front0;
12     }
13     location /sockjs-node {
14         proxy_pass http://front0;
15         proxy_http_version 1.1;
16         proxy_set_header Upgrade $http_upgrade;
17         proxy_set_header Connection "Upgrade";
18     }
19     location /api {
20         rewrite /back0/(.*) /$1 break;
21         proxy_pass http://back0;
22     }
23 }
```

used for routing requests to different upstream servers based on the URL path. Here's a summary of each section:

1. `upstream front0`: Defines an upstream block named `front0`, which represents the backend server for serving frontend assets. The `server front0:3000;` line specifies the address and port of the frontend server.
2. `upstream back0`: Defines an upstream block named `back0`, which represents the backend server for handling API requests. The `server back0:8000;` line specifies the address and port of the backend server.
3. `server`: Begins the server block, which defines the settings for the NGINX server.
 - `listen 80;`: Configures NGINX to listen for incoming HTTP connections on port 80.
 - `location /`: Defines a location block for requests to the root path (`/`). Requests to the root path are proxied to the frontend server specified by the `proxy_pass http://front0;` directive.
 - `location /sockjs-node`: Defines a location block for requests to `/sockjs-node`. These requests are also proxied to the frontend server, but with additional proxy settings to handle WebSocket connections.
 - `location /api`: Defines a location block for requests to the `/api` path. Requests to this path are rewritten using `rewrite /back0/(.*) /$1 break;` to remove the `/back0` prefix and then

proxied to the backend server specified by proxy_pass
http://back0;.

***second requirement** in our project is: docker-compose

The docker-compose.yml file orchestrates the Docker containers for local development. It defines services for backend, frontend, Redis, and Nginx, along with their configurations (ease dockerization process for small numbers of containers).

We have docker-compose.yml (main or common for all environment) and for each env , we have docker-compose.xxx.yml where xxx is for prod or production , dev or development , test or testing , staging

```
docker-compose.yml > {} services
docker-compose.yml - The Compose specification establishes a standard for the definition of multi-container platform-agnostic applications (compose-spec.json)
1  version: '3'
2  services:
3
4  back0:
5    build:
6      context: .
7      dockerfile: Dockerfile
8      target: backend
9    ports:
10     #- "8000-8003:8000"
11     - "8000:8000"
12    environment:
13     - whois=backend
14    env_file:
15     - ecommerce_react_node-main/backend/.env
16    volumes:
17     - ./ecommerce_react_node-main/backend/src:/app/src:ro
18
19
20
21  front0:
22    build:
23      context: .
24      dockerfile: Dockerfile
25      target: frontend
26    ports:
27     #- "3000-3003:3000"
28     - "3000:3000"
29    environment:
30     - whois=frontend
31    env_file:
```

This Docker Compose service configuration defines multiple services for containerizing different components of an e-commerce web application. Let's focus on one service, back0, and summarize its key components:

1. Service Name: back0
2. Build Configuration:
 - Specifies the Dockerfile to use (Dockerfile).
 - Targets the backend stage in the multi-stage build process.
3. Ports:
 - Maps port 8000 of the container to port 8000 on the host.
4. Environment Variables:
 - Sets the environment variable whois to backend.
 - Loads additional environment variables from the .env file located in the ecommerce_react_node-main/backend/ directory.
5. Volumes:
 - Mounts the local directory ./ecommerce_react_node-main/backend/src to /app/src inside the container in read-only mode (ro).
6. Dependencies:
 - Declares dependencies on other services (back0 and redis) using depends_on (start after).

And we have a swarm.yml file for docker swarm (a container orchestration tool for clustering and scheduling Docker containers.) , management like specify replicas, resources, restart policy and so on

***Third requirement:** Orchestration with Kubernetes

YAML files describe various resources in a Kubernetes environment for deploying an e-commerce web application. Let's focus on the relationships between the resources and explain one deployment (frontend-deploy) in detail:

```

! deploy.yml > {} spec > {} template
48   apiVersion: apps/v1
49   kind: Deployment
50   metadata:
51     name: frontend-deploy
52   spec:
53     replicas: 1
54     selector:
55       matchLabels:
56         app: frontend
57     strategy:
58       type: RollingUpdate
59     template:
60       metadata:
61       labels:
62         app: frontend
63       spec:
64         containers:
65           - name: front0
66             image: abdoemam/ecommerce app:frontend
67             ports:
68               - containerPort: 3000
69             command: ["npm", "start"]
70             volumeMounts:
71               - name: front0-volume
72                 mountPath: /app/src
73               - name: secret-volume
74                 mountPath: /secrets
75               - name: shared
76                 mountPath: /shared
77               - name: test
78                 mountPath: /cloud0/ecommerce react node-main/frontend/
79
80
81
82
83

```

1.Deployment (frontend-deploy):

- This deployment manages the frontend component of the e-commerce web application.
- It ensures that a specified number of pod replicas are running to maintain availability.
- The pods are created based on the container image abdoemam/ecommerce app:frontend as Docker hub repo.
- The deployment performs rolling updates when changes are made.
- The frontend container listens on port 3000.
- It mounts volumes for storing application source code (front0-volume), secrets (secret-volume), and shared data (shared).
- It depends on PersistentVolumeClaims (frontendpvc, sharedpvc) for persistent storage

```

33   claimRef:
34     name: backendpvc
35     namespace: default
36 ---
37 apiVersion: v1
38 kind: PersistentVolume
39 metadata:
40   name: frontendpv
41 spec:
42   capacity:
43     storage: 10Gi
44   volumeMode: Filesystem
45   accessModes:
46     - ReadWriteOnce
47   persistentVolumeReclaimPolicy: Retain
48   hostPath:
49     path: "/cloud0/ecommerce_react_node-main/frontend/src"
50
51   claimRef:
52     name: frontendpvc
53     namespace: default
54
55 ---
56
57

```

2.Persistent Volumes (nginxpv, backendpv, frontendpv, sharedpv):

- These volumes provide persistent storage for different components of the application.
- They are bound to PersistentVolumeClaims to ensure that each component has the required storage capacity.
- Each volume has a specified storage capacity and access mode.

If I talk about frontend-deploy, I create 3 persistent volumes , first(front0-volume) for mapping between frontend/src into /app/src in my container, second(secret-volume) for mount secrets as a volumes to use it for credentials if forgotten, third(shared) is shared path between frontend and backend , don't forget to run minikube mount <path on physical machine>:<on minikube> to use mounted volumes in minikube as a hostpath in pv.yml

```

25 apiVersion: v1
26 kind: PersistentVolumeClaim
27 metadata:
28   name: frontendpvc
29 spec:
30   resources:
31     requests:
32       storage: 10Gi
33   volumeMode: Filesystem
34   accessModes:
35     - ReadWriteOnce
36 ---
37
38 apiVersion: v1
39 kind: PersistentVolumeClaim
40 metadata:
41   name: sharedpvc
42 spec:
43   resources:
44     requests:
45       storage: 5Gi
46   volumeMode: Filesystem
47   accessModes:
48     - ReadWriteOnce

```

3. Persistent Volume Claims (nginxpvc, backendpvc, frontendpvc, sharedpvc):

- These claims request storage resources from Persistent Volumes.
- They define the required storage capacity and access mode.
- Each claim is associated with a specific component of the application to ensure that it gets the required storage.

```

35   type: NodePort
36 ---
37 apiVersion: v1
38 kind: Service
39 metadata:
40   name: front0
41 spec:
42   selector:
43     app: frontend
44   ports:
45     - port: 3000
46       targetPort: 3000
47     type: NodePort
48

```

4. Services (nginx, redis, back0, front0) we can use it in container for network communication such as ping nslookup, curl ,api calls and so on, so don't worry we have a name resolution DNS :

- These services expose the frontend, backend, and other components to the external network.
- They define ports and target ports for communication.

- Each service selects pods based on specified labels (app: nginx, app: redis, app: backend, app: frontend).

```

1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: redis-cm
5  data:
6    redis-config: ""
7  ---
8  apiVersion: v1
9  kind: Secret
10 metadata:
11   name: mysecret
12   type: Opaque
13 data:
14   admin: ZW1hbUBnbWFpbC5jb20K
15   pwd: ZW1hbTEyMzQ1Ngo=
16 stringData:
17   username: oufal23@gmail.com
18   password: oufal23456
19

```

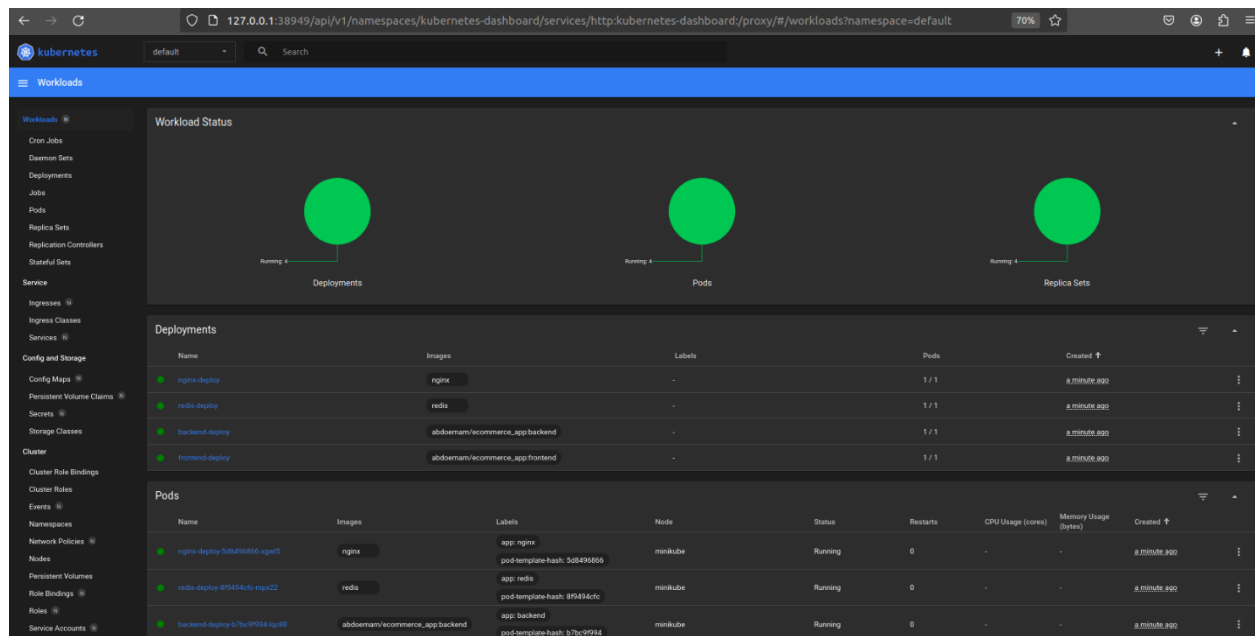
4. A-ConfigMap (redis-cm) we use for redis only:

- This ConfigMap stores configuration data for the Redis component.

4. B-Secret (mysecret):

- This Secret stores sensitive data such as credentials.
- It contains encoded admin credentials (admin, pwd) and plain text credentials (username, password).

Our Cluster up and running



```
NAME                                READY   STATUS    RESTARTS   AGE
pod/backend-deploy-b7bc9f994-lqc88  1/1     Running   0           6m15s
pod/frontend-deploy-7dc7dbc7cd-fqfk9 1/1     Running   0           6m15s
pod/nginx-deploy-5d8496866-xgw15     1/1     Running   0           6m6s
pod/redis-deploy-8f9494cfc-mpx22     1/1     Running   0           6m10s

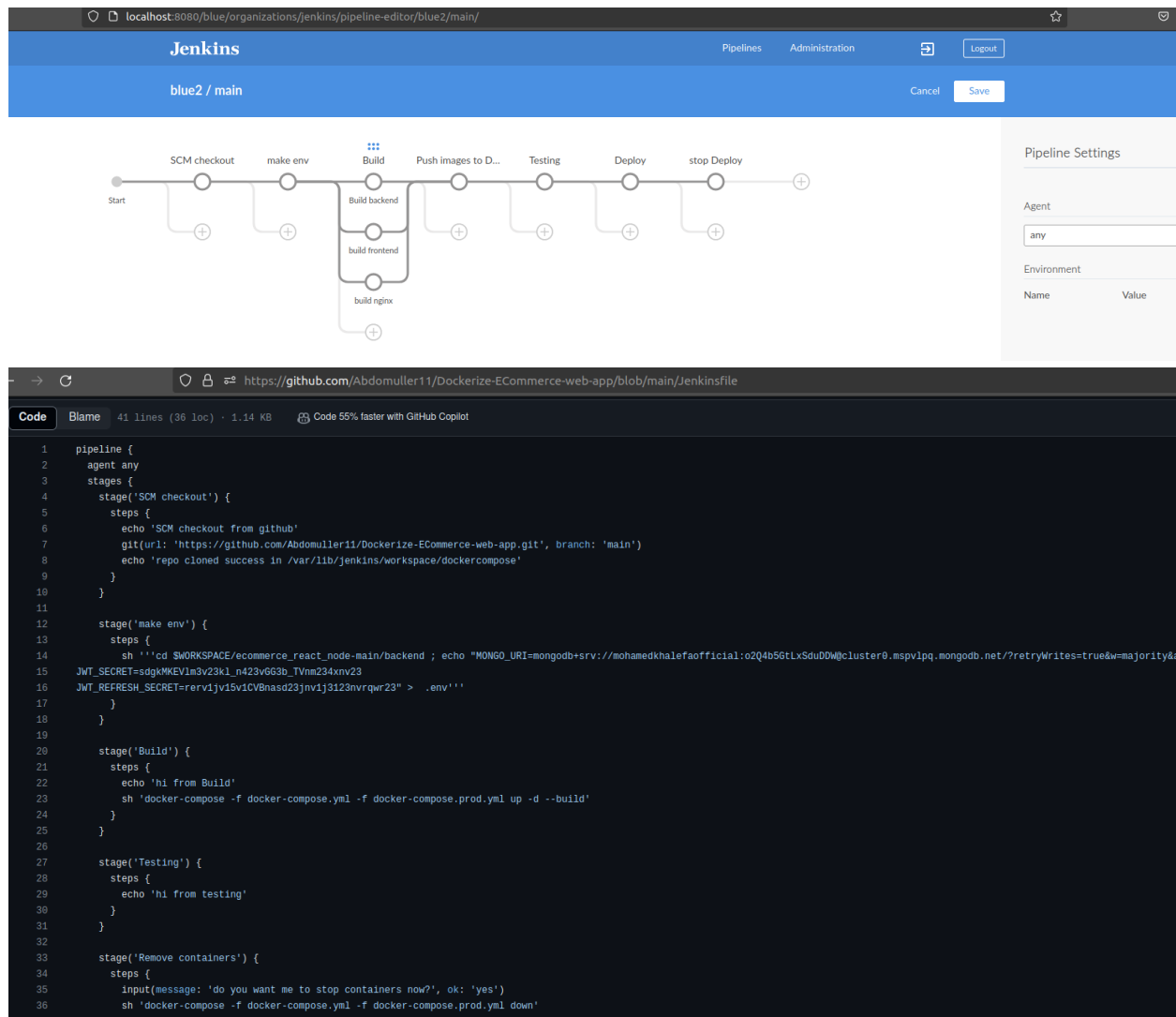
NAME                                TYPE          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
service/back0                        NodePort      10.107.159.1    <none>            8000:32152/TCP   6m19s
service/front0                       NodePort      10.98.182.69    <none>            3000:30945/TCP   6m19s
service/kubernetes                   ClusterIP     10.96.0.1       <none>            443/TCP          9m18s
service/nginx                        NodePort      10.99.137.245   <none>            8888:31867/TCP   6m19s
service/redis                        NodePort      10.97.202.76    <none>            6379:30439/TCP   6m19s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/backend-deploy      1/1     1             1           6m15s
deployment.apps/frontend-deploy     1/1     1             1           6m15s
deployment.apps/nginx-deploy        1/1     1             1           6m6s
deployment.apps/redis-deploy        1/1     1             1           6m10s

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/backend-deploy-b7bc9f994 1         1         1       6m15s
replicaset.apps/frontend-deploy-7dc7dbc7cd 1         1         1       6m15s
replicaset.apps/nginx-deploy-5d8496866     1         1         1       6m6s
replicaset.apps/redis-deploy-8f9494cfc     1         1         1       6m10s
u0@sec0:/cloud0$
```

***we also do a bonus task: Jenkins to automate integration and deployment or delivery with freestyle job and pipeline job**

Don't forget to install required plugin for pipeline



The image displays two screenshots related to a Jenkins pipeline. The top screenshot shows the Jenkins Pipeline Editor interface for a pipeline named 'blue2 / main'. The pipeline graph consists of several stages: 'SCM checkout', 'make env', 'Build' (which is expanded to show 'build backend', 'build frontend', and 'build nginx' steps), 'Push images to D...', 'Testing', 'Deploy', and 'stop Deploy'. The bottom screenshot shows the Jenkinsfile code for this pipeline, which is a YAML file defining the pipeline structure and steps.

```
1 pipeline {
2   agent any
3   stages {
4     stage('SCM checkout') {
5       steps {
6         echo 'SCM checkout from github'
7         git(url: 'https://github.com/Abdomuller11/Dockerize-ECommerce-web-app.git', branch: 'main')
8         echo 'repo cloned success in /var/lib/jenkins/workspace/dockercompose'
9       }
10    }
11
12    stage('make env') {
13      steps {
14        sh '''cd $WORKSPACE/eCommerce_react_node-main/backend ; echo "MONGO_URI=mongodb+srv://mohamedkhalefaofficial:o2Q4b5GtLxSduDDW@cluster0.mspvlpq.mongodb.net/?retryWrites=true&majority=&w=2"'''
15        sh 'JWT_SECRET=sdgkMKEVIm3v23k1_n423vGG3b_TVnm234xnv23'
16        sh 'JWT_REFRESH_SECRET=rerv1jv15v1CVBnasd23jnv1j3123nvrqwr23' > .env'''
17      }
18    }
19
20    stage('Build') {
21      steps {
22        echo 'hi from Build'
23        sh 'docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d --build'
24      }
25    }
26
27    stage('Testing') {
28      steps {
29        echo 'hi from testing'
30      }
31    }
32
33    stage('Remove containers') {
34      steps {
35        input(message: 'do you want me to stop containers now?', ok: 'yes')
36        sh 'docker-compose -f docker-compose.yml -f docker-compose.prod.yml down'
37      }
38    }
39  }
40 }
```

This Jenkins pipeline automates the process of building and deploying the Dockerized e-commerce web application. Here's a summary of each stage:

A.SCM Checkout:

- This stage checks out the source code from the GitHub repository.
- It clones the repository using the specified URL and branch (main).
- After successful cloning in `/var/lib/Jenkins/workspace/$WORKSPACE`, it echoes a message indicating the successful clone.

B.Make Environment:

- This stage sets up the environment variables required for the backend component of the application.
- It uses a shell script (sh) to navigate to the backend directory and create a .env file with MongoDB URI and JWT secrets.

C.Build:

- This stage builds and deploys the application using Docker Compose.
- It echoes a message indicating the start of the build process.
- It executes a shell command (sh) to run Docker Compose with the specified YAML files (docker-compose.yml and docker-compose.prod.yml).
- The containers are built (--build flag) and started in detached mode (-d flag).

D.Testing:

- This stage is intended for testing purposes.
- It echoes a message indicating that the testing stage is initiated.

E.Remove Containers:

- This stage prompts for user input to stop the containers.
- It displays a message asking if the user wants to stop the containers.
- If the user confirms (by clicking 'yes' in the Jenkins UI), it executes a shell command to stop the containers using Docker Compose.

Diagram:

