DAY-3 AND DAY-4

BHUVANKUMAR V

Kubernetes

Backend-Pandas and flask in python

Docker Build & Run Documentation

1. Verify File Structure

ls

Lists files in the current directory (should include Dockerfile, app.py, docker-compose.yml, requirements.txt, etc.).

---

2. Create or Edit CSV File

nano products.csv

Opens the products.csv file in the nano editor to add or modify product data.

---

3. Verify CSV File Content

cat products.csv

Displays the contents of products.csv to confirm the data.

---

4. Build Docker Image Without Cache

sudo docker build --no-cache -t backend:latest .

Builds a fresh Docker image, ensuring all changes are included. The --no-cache flag forces a rebuild of every layer.

---

5. Run the Docker Container

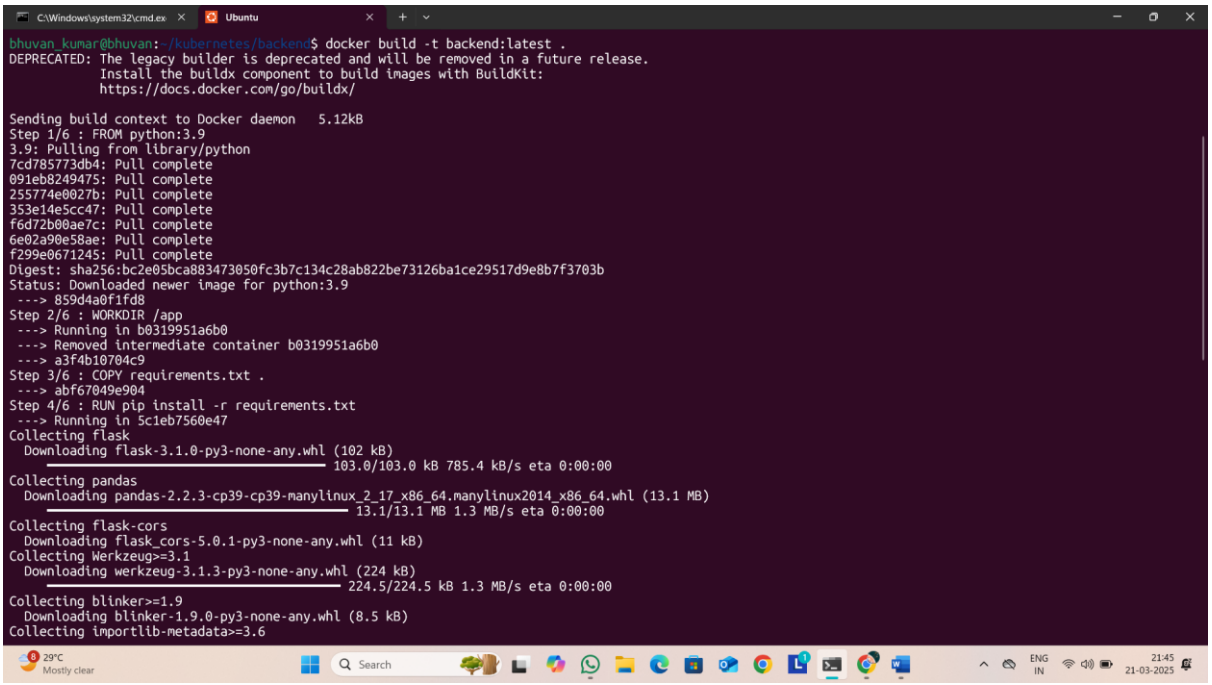
sudo docker run -d -p 7000:7000 backend:latest

Runs the container in detached mode and maps host port 7000 to container port 7000.

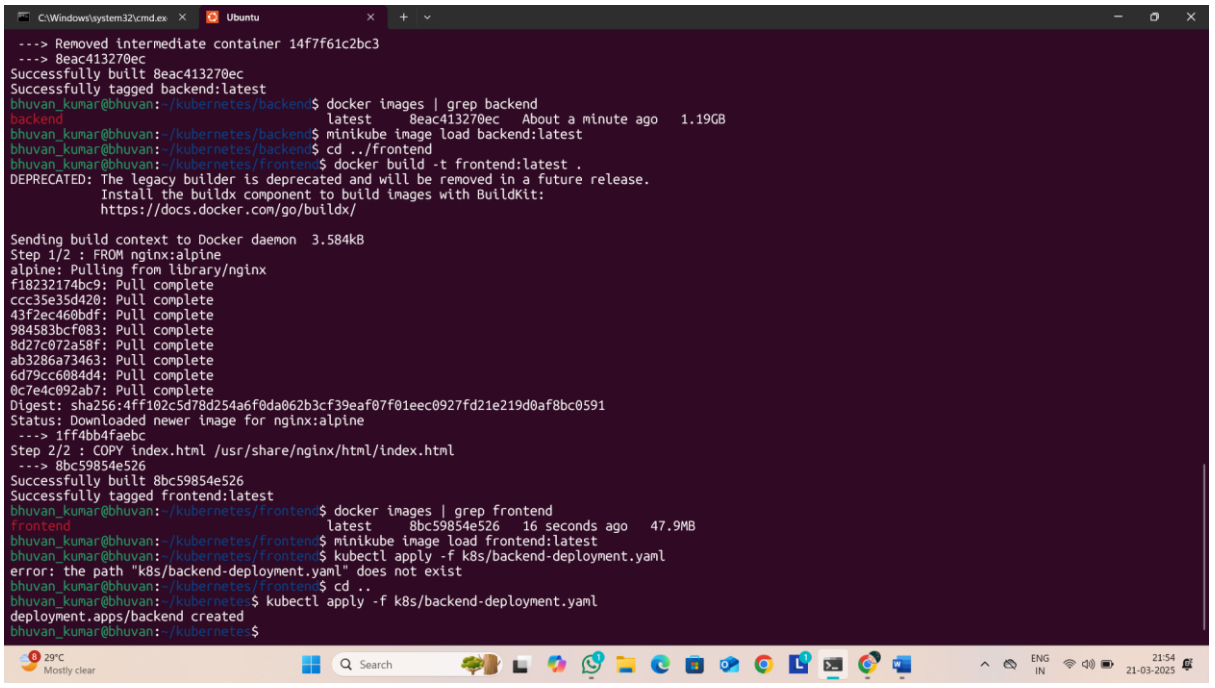
---

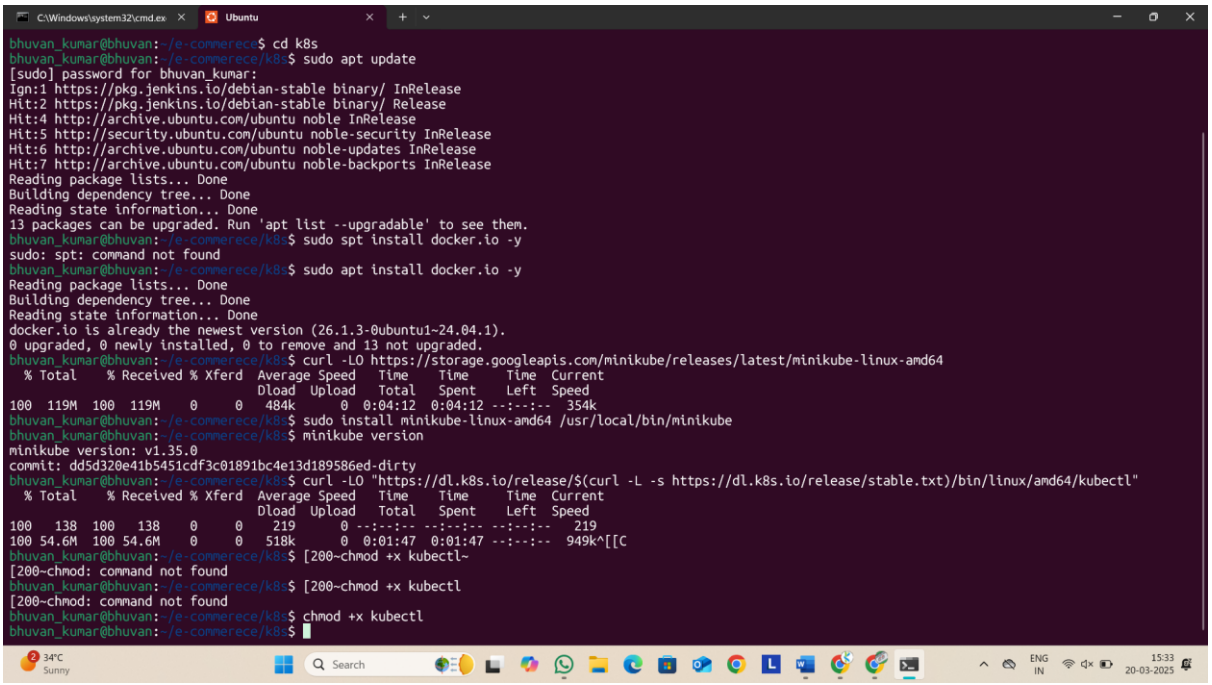
6. Check Container Logs

sudo docker logs <container_id>



Replace <container_id> with the actual container ID to view the running application's logs.





Creating a container for frontend

Below is the concise documentation content for your frontend Docker build:

Frontend Docker Build Documentation

1. Navigate to the Frontend Directory

cd frontend/

Changes directory to the frontend folder where your files are located.

---

2. Create or Edit the HTML File

nano index.html

Opens the index.html file in the nano editor for creating or modifying the webpage content.

3. Create or Edit the Dockerfile

nano Dockerfile

Opens the Dockerfile in the nano editor to set up instructions for building the Docker image.

---

4. Dockerfile Content

FROM nginx:alpine

COPY index.html /usr/share/nginx/html/index.html

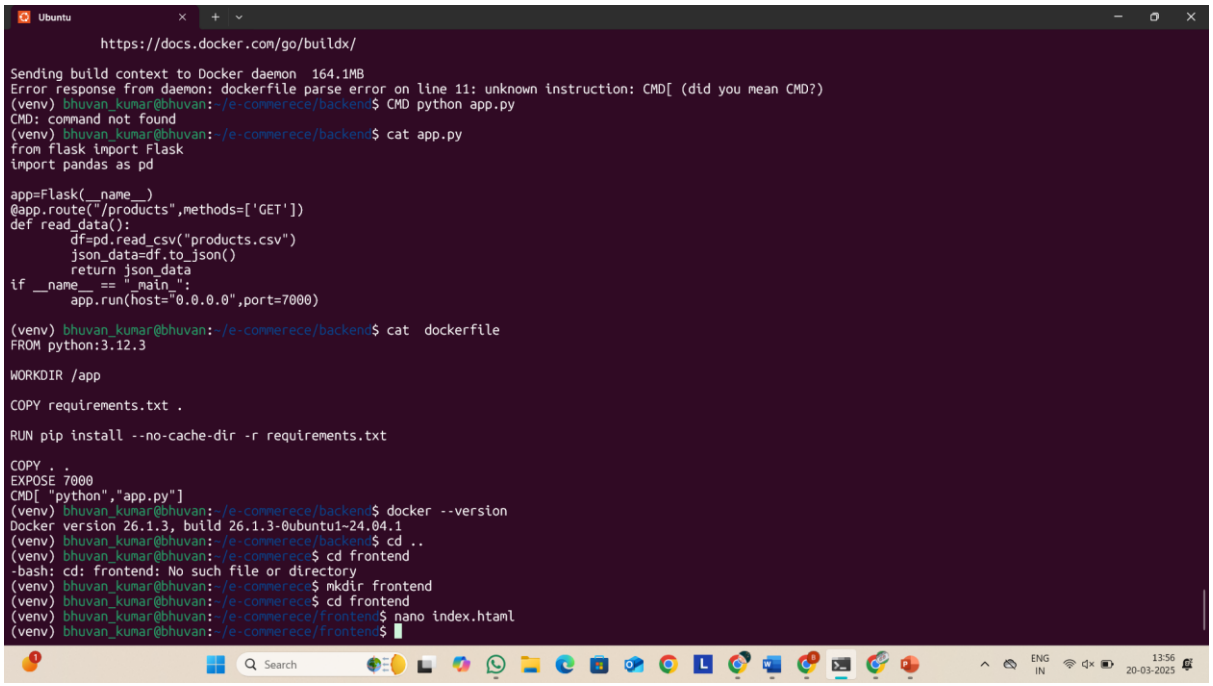
Specifies the base image as nginx:alpine and copies your index.html to the default Nginx HTML directory.

---

5. Build the Docker Image

sudo docker build -t frontend:latest .

Builds the Docker image for the frontend. The command pulls the necessary Nginx image, executes the copy command, and tags the image as frontend:latest.



Kubernets Deployment YAML files

Below is a brief, step-by-step description for setting up your Kubernetes deployments for both backend and frontend:

---

1. Organize Project Structure

- Create a Kubernetes Folder:
- mkdir k8s

This creates a separate folder (k8s) to store all your Kubernetes configuration files.

- Navigate to the Kubernetes Directory:
- cd k8s/

Move into the k8s directory to work on deployment files.

---

2. Create Backend Deployment Configuration

- Create/Edit the Backend Deployment File:
- nano backend-deployment.yaml

Opens the file in the nano editor to add deployment configuration for the backend.

- Backend Deployment File Content:
- apiVersion: apps/v1
- kind: Deployment
- metadata:
- name: backend
- spec:
- replicas: 1
- selector:
- matchLabels:
- app: backend
- template:
- metadata:
- labels:
- app: backend
- spec:
- containers:
- - name: backend

- image: backend:latest
- ports:
- - containerPort: 7000

This file defines a Kubernetes Deployment for your backend application:

  - apiVersion & kind: Specifies the resource type.
  - metadata: Names the deployment as backend.
  - spec.replicas: Sets the number of pod replicas.
  - selector & template.metadata.labels: Ensure that the Deployment manages pods with the label app: backend.
  - spec.template.spec.containers: Specifies the container details, including the Docker image (backend:latest) and the port (7000) that the container exposes.

---

3. Create Frontend Deployment Configuration

- Create/Edit the Frontend Deployment File:
- nano frontend-deployment.yaml

Opens the file in nano to add deployment configuration for the frontend.

- Frontend Deployment File Content:
- apiVersion: apps/v1
- kind: Deployment
- metadata:
- name: frontend
- spec:
- replicas: 1
- selector:
- matchLabels:
- app: frontend
- template:
- metadata:
- labels:
- app: frontend
- spec:

- containers:
- - name: frontend
- image: frontend:latest
- ports:
- - containerPort: 7500

This file defines a Kubernetes Deployment for your frontend application:

  - metadata: Names the deployment as frontend.
  - selector & template.metadata.labels: Ensure that the Deployment manages pods with the label app: frontend.
  - Container Specification: Sets the container to use the Docker image (frontend:latest) and exposes port 7500.

Summary

- Directory Setup:
  Organized your project by creating a k8s directory for Kubernetes configuration files.
- Backend Deployment:
  Created backend-deployment.yaml to deploy the backend container (using port 7000).
- Frontend Deployment:
  Created frontend-deployment.yaml to deploy the frontend container (using port 7500).

This documentation provides a brief and clear outline of your Kubernetes deployment process for both backend and frontend components.

Below is a step-by-step description for setting up Kubernetes Services for your backend and frontend applications using the provided YAML configuration:

---

1. Create a Service Configuration File

- **Command to Create/Edit the File:**
- nano service.yaml

This command opens a text editor (nano) to create or edit the service configuration file where you'll define both backend and frontend services.

---

2. Define the Backend Service

Paste the following YAML snippet for the backend service into your service.yaml file:

apiVersion: v1

```
kind: Service
metadata:
  name: backend-service
spec:
  selector:
    app: backend
  ports:
    - protocol: TCP
      port: 7000
      targetPort: 7000
type: ClusterIP
```

Explanation:

- **apiVersion: v1**
  Specifies the API version used to create the Service.
- **kind: Service**
  Indicates that the resource being created is a Service.
- **metadata.name: backend-service**
  Sets the name of the service to backend-service.
- **spec.selector:**
  Matches pods that have the label app: backend. This ensures the service routes traffic to the correct backend pods.
- **spec.ports:**
  Defines the port configuration:
  - **protocol:** TCP – The network protocol used.
  - **port:** 7000 – The port on which the service is exposed within the cluster.
  - **targetPort:** 7000 – The port on the pod to which traffic will be directed.
- **type: ClusterIP**
  Creates an internal service, exposing it only within the cluster.

---

3. Define the Frontend Service

Below the backend service configuration in the same file, add the following YAML snippet for the frontend service:

```
apiVersion: v1
```

```
kind: Service
metadata:
  name: frontend-service
spec:
  selector:
    app: frontend
  ports:
    - protocol: TCP
      port: 7500
      targetPort: 7500
type: NodePort
```

Explanation:

- **metadata.name: frontend-service**
  Names the service frontend-service.
- **spec.selector:**
  Matches pods with the label app: frontend so that this service routes traffic to your frontend pods.
- **spec.ports:**
  Defines the ports:
  - **protocol:** TCP – Uses the TCP protocol.
  - **port:** 7500 – The port exposed by the service inside the cluster.
  - **targetPort:** 7500 – The port on the frontend pod that will handle the incoming traffic.
- **type: NodePort**
  Exposes the service on a port on each node's IP, allowing external access to the frontend application.

---

4. Save and Exit

- **In nano:** Press Ctrl+O to write the changes, then Enter to confirm. Press Ctrl+X to exit the editor.

---

5. Apply the Service Configuration

- **Command to Apply the YAML File:**

- kubectl apply -f service.yaml

This command tells Kubernetes to create or update the services as defined in the service.yaml file.

---

6. Verify the Services

- **Command to Check Services:**
- kubectl get services

This command lists all services in the current namespace, confirming that both backend-service and frontend-service have been created and are running with the correct configuration.

Summary

- **Backend Service:**
  Uses ClusterIP to expose the backend on port 7000 internally. It routes traffic to pods labeled app: backend.
- **Frontend Service:**
  Uses NodePort to expose the frontend externally on port 7500, routing traffic to pods labeled app: frontend.

This detailed step-by-step guide covers creating a configuration file, defining services for both backend and frontend applications, applying the configuration with kubectl, and verifying that the services are correctly deployed.

Below is a step-by-step explanation for the provided ConfigMap YAML configuration:

---

1. Purpose of the ConfigMap

- **Objective:**
  The ConfigMap stores configuration data (in this case, a file path) that can be consumed by the backend application without hardcoding values into the container image.

---

2. YAML Breakdown

apiVersion: v1

kind: ConfigMap

metadata:

  name: backend-config

data:

DATABASE_FILE: "/backend/products.csv"

- **apiVersion: v1**
  Specifies the API version for the ConfigMap resource.
- **kind: ConfigMap**
  Indicates that the resource being created is a ConfigMap.
- **metadata:**
  - **name: backend-config**
    Sets the name of the ConfigMap to backend-config. This is how you will reference it in other configurations (like a Deployment).
- **data:**
  - **DATABASE_FILE:**
    Defines a key called DATABASE_FILE with a value of "/backend/products.csv". This key-value pair is the configuration data your backend application can use to locate the products CSV file.

---

3. Creating the ConfigMap File

- **Command to Create/Edit the ConfigMap File:**
- nano backend-config.yaml

Opens the nano text editor to create or modify the file containing the ConfigMap definition.

- **Paste the YAML Content:**
  Insert the above YAML content into the file and save it.

---

4. Applying the ConfigMap

- **Command to Create the ConfigMap in Kubernetes:**
- kubectl apply -f backend-config.yaml

This command tells Kubernetes to create or update the ConfigMap using the configuration specified in the YAML file.

---

5. Using the ConfigMap in Your Application

- **Reference in a Pod or Deployment:**
  You can reference the backend-config ConfigMap in your Deployment YAML to inject the DATABASE_FILE variable into your container. For example, under the container spec, you could add:
- env:
- - name: DATABASE_FILE

- valueFrom:
- configMapKeyRef:
- name: backend-config
- key: DATABASE_FILE

This makes the DATABASE_FILE environment variable available to your application at runtime, with the value /backend/products.csv.

Summary

- **What it Does:**
  The ConfigMap named backend-config stores a key-value pair where DATABASE_FILE points to the CSV file location.
- **Why It's Useful:**
  It decouples configuration from the container image, making it easier to update configuration without rebuilding the image.
- **How to Apply:**
  Create the YAML file, then run kubectl apply -f backend-config.yaml to deploy the configuration in your cluster.

This explanation covers the configuration's intent, its components, how to create and apply it, and how to integrate it into your application's deployment.

Below is a step-by-step description of the commands you executed and what each step accomplished:

---

1. Change Directory to the Kubernetes Folder

cd ~/e-commerce/k8s

Navigates to the k8s directory where you keep your Kubernetes configuration and installation files.

---

2. Download kubectl

curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"

This command downloads the latest stable version of the kubectl binary for Linux (amd64).

---

3. Make kubectl Executable

chmod +x kubectl

Gives the downloaded kubectl binary execute permissions so it can run.

---

4. Move kubectl to a Directory in Your PATH

sudo mv kubectl /usr/local/bin/

Moves the kubectl binary to /usr/local/bin, allowing you to run it from anywhere in your terminal.

---

5. Verify kubectl Installation

kubectl version --client

Checks the installed version of kubectl to confirm the installation was successful.

---

6. Download minikube

curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64

Downloads the latest minikube binary, which will be used to run a local Kubernetes cluster.

---

7. Make minikube Executable

chmod +x minikube-linux-amd64

Sets the executable permission on the minikube binary.

---

8. Move minikube to a Directory in Your PATH

sudo mv minikube-linux-amd64 /usr/local/bin/minikube

Moves the minikube binary to /usr/local/bin and renames it to minikube so it can be executed easily.

Note:
An error like mv: missing destination file operand occurs if there's no space between the source and destination. Ensure you separate the source file (minikube-linux-amd64) and the destination (/usr/local/bin/minikube) with a space.

---

9. Start Minikube

minikube start

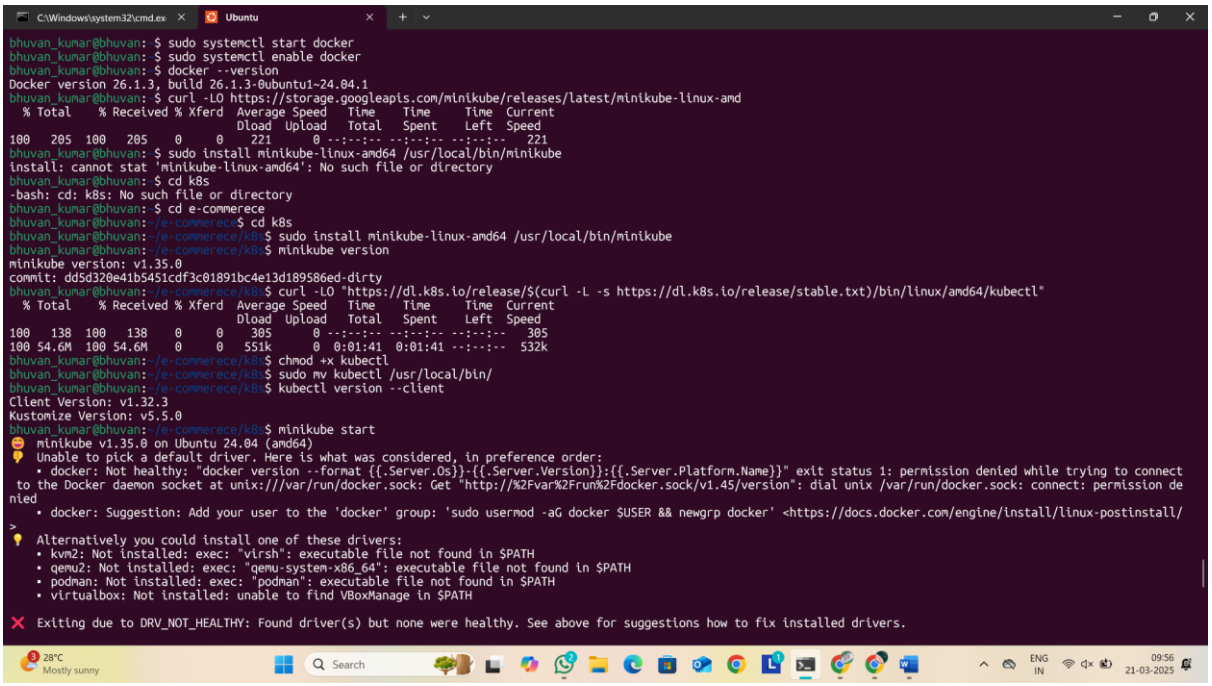
Initiates the minikube local Kubernetes cluster using Docker as the driver. During this process, minikube pulls necessary images and preloads Kubernetes components.

10. Verify Minikube Installation

minikube version

Displays the minikube version information to confirm that minikube is properly installed and running.

This complete sequence sets up both kubectl and minikube on your system, allowing you to manage and run a local Kubernetes cluster.



To stop all processes utilizing port **8080**, follow these detailed steps:

Step 1: Identify the Process Using Port 8080

Run the following command to check which process is using port **8080**:

sudo netstat -tulnp | grep ":8080"

Explanation:

- sudo → Runs the command with root privileges.
- netstat -tulnp → Displays active network connections.
    - -t → TCP connections.
    - -u → UDP connections.
    - -l → Listening sockets.
    - -n → Show numerical addresses instead of resolving hostnames.

- -p → Show the process ID (PID) and program name.

- | grep ":8080" → Filters the output to show only lines with port **8080**.

Example Output:

tcp 0 0 0.0.0.0:8080 0.0.0.0:* LISTEN 12345/nginx

Here, **12345** is the **PID** of the process using port **8080**.

Step 2: Kill the Process

Once you have the **PID**, replace <PID> with the actual process ID and run:

sudo kill -9 12345

Explanation:

- kill -9 → Forcefully terminates the process.
- 12345 → The process ID (PID) obtained from the previous step.

Step 3: Verify If Port 8080 Is Free

After killing the process, run:

sudo netstat -tulnp | grep ":8080"

If no output is shown, the port is free.

Alternative: Kill All Processes Using 8080 in One Command

If multiple processes are using port 8080, you can terminate them all at once:

sudo kill -9 $(sudo netstat -tulnp | grep ":8080" | awk '{print $7}' | cut -d'/' -f1)

Explanation:

- awk '{print $7}' → Extracts the PID/ProgramName column.
- cut -d'/' -f1 → Extracts only the PID.
- kill -9 $(...) → Kills all matching PIDs.

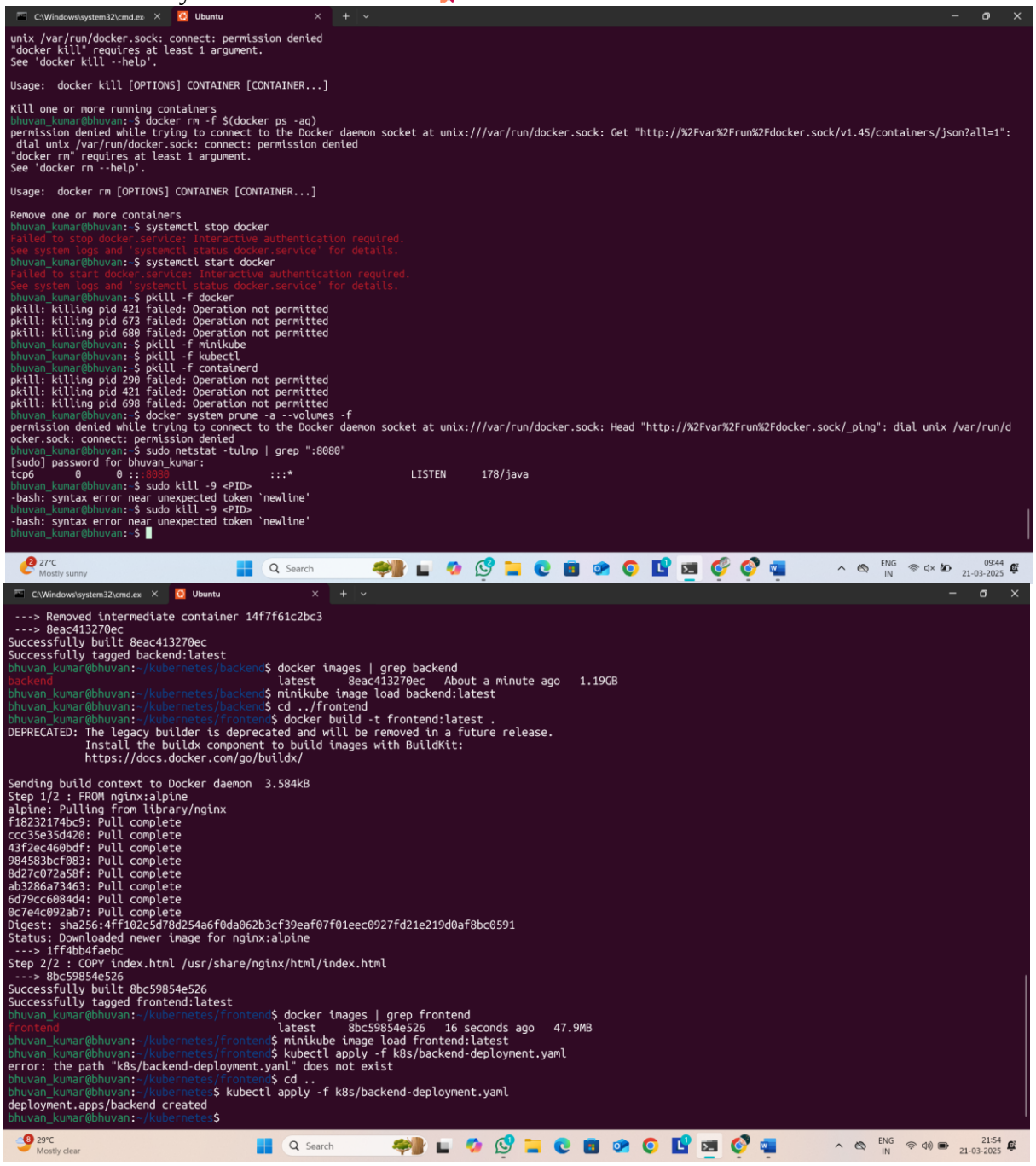
Step 4: Restart the Service (Optional)

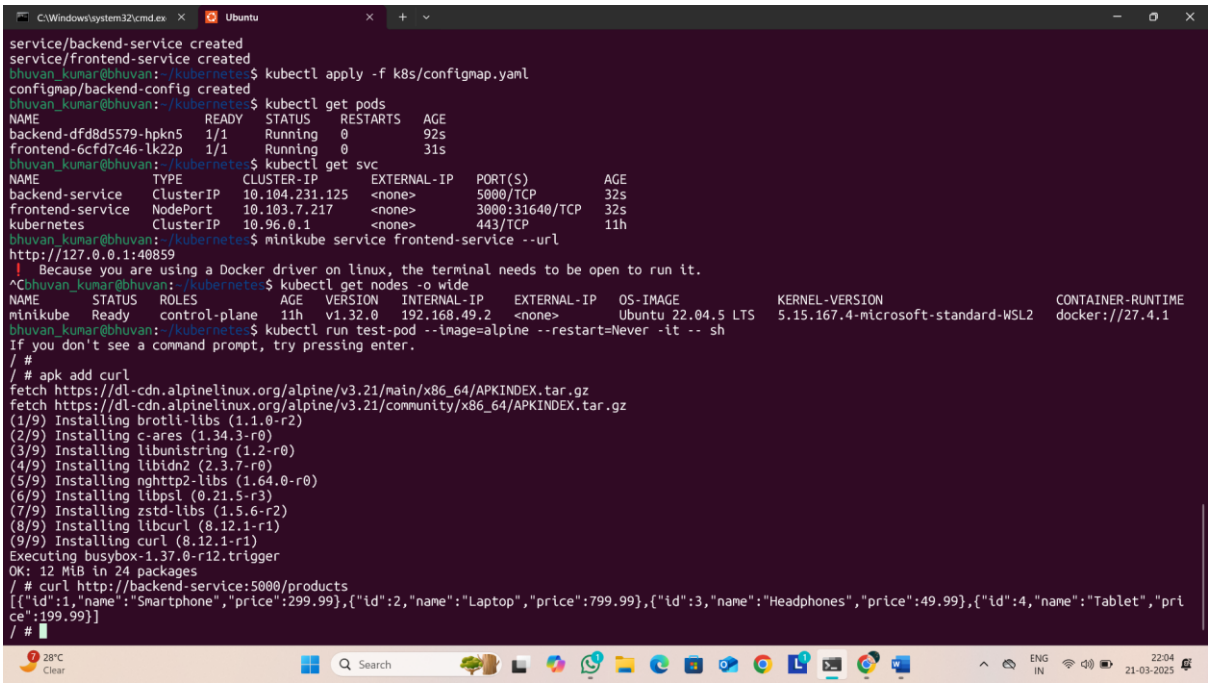
If you need to restart the application that was using port 8080, use:

sudo systemctl restart <service-name>

Replace <service-name> with the actual service (e.g., nginx, apache2, docker, etc.).

Let me know if you need more details! 🚀





Here’s a step-by-step breakdown of the commands you provided:

1. Set up Docker to use Minikube's Docker daemon:

eval $(minikube docker-env)

This command sets the environment variables so Docker can build images directly inside Minikube’s virtual machine, instead of your local Docker daemon.

2. Build the backend Docker image:

cd backend

docker build -t backend:latest .

This command builds the backend Docker image from the Dockerfile in the backend folder and tags it as backend:latest.

3. Verify backend image exists:

docker images | grep backend

This checks if the backend:latest image exists in your local Docker registry.

4. Load the backend image into Minikube:

minikube image load backend:latest

This command loads the backend:latest image into Minikube’s Docker daemon so it can be used by Kubernetes.

5. Build the frontend Docker image:

cd ../frontend

docker build -t frontend:latest .

This builds the frontend Docker image from the Dockerfile in the frontend folder and tags it as frontend:latest.

6. Verify frontend image exists:

docker images | grep frontend

This checks if the frontend:latest image exists in your local Docker registry.

7. Load the frontend image into Minikube:

minikube image load frontend:latest

This loads the frontend:latest image into Minikube's Docker daemon.

8. Apply Kubernetes configurations for the backend, frontend, and services:

kubectl apply -f k8s/backend-deployment.yaml

kubectl apply -f k8s/frontend-deployment.yaml

kubectl apply -f k8s/service.yaml

kubectl apply -f k8s/configmap.yaml

These commands apply the Kubernetes configuration files for deploying the backend, frontend, services, and config maps. The deployment.yaml files describe how to run the containers, and service.yaml defines how they interact.

9. Check the status of pods and services:

kubectl get pods

kubectl get svc

These commands list all the pods (containers) running and services exposed in your Kubernetes cluster.

10. Access the frontend service URL:

minikube service frontend-service --url

This provides the external URL of the frontend service running in Minikube.

11. Get node details to confirm the cluster's node setup:

kubectl get nodes -o wide

This command provides a detailed list of the nodes in your Kubernetes cluster.

12. Test the backend by creating a temporary pod:

kubectl run test-pod --image=alpine --restart=Never -it -- sh

apk add curl # Install curl if not available

curl http://backend-service:5000/products

- kubectl run creates a test pod using the alpine image (a lightweight Linux container).
- apk add curl installs curl inside the pod to make HTTP requests.

- curl http://backend-service:5000/products makes an HTTP request to the backend service at port 5000 to check if it returns the products data.

This is a quick overview of the deployment and testing process! Let me know if you need more details on any step.