Name-Anjali Savalkar

Roll no- 140

Usn-01fe22bcs196

1 . E-Commerce Cloud Deployment -Service model PAAS (platform as a service)

---

1. Project Overview

Our project focuses on building a cloud-based **e-commerce application** that provides a seamless online shopping experience. The platform enables users to register, browse products, manage their shopping cart, and for admins to manage product listings.

---

2. Application Functionalities

The core features implemented in the application are:

- **User Authentication**
  Secure registration and login system for customers.
- **Product Listing and Browsing**
  Products displayed with details like name, price, image, and description.
- **Add/Remove to Cart**
  Users can manage cart contents by adding or removing products.
- **Cart Value Calculation**
  Automatically calculates the total price of all items in the cart.
- **Admin Panel**
  Admin users can add new products to the system via a secure interface.

---

3. Cloud Service Provider

We chose **Render** as our cloud provider for the following reasons:

- **Ease of Use**
  Simplified deployment for full-stack applications (frontend and backend).
- **Free Tier**
  Offers free hosting for student projects, including web services and databases.
- **Integrated Services Used**
  - **Web Services**: Hosting Node.js + Express backend
  - **Static Sites**: Hosting the frontend (React, HTML, CSS, JS)
  - **PostgreSQL Database**: Storing user data, product info, and cart contents

---

4. Cloud Architecture

i) Deployment Model – Public Cloud

Our application is hosted on Render's **public cloud**, making it accessible globally. All resources are managed and maintained by Render.

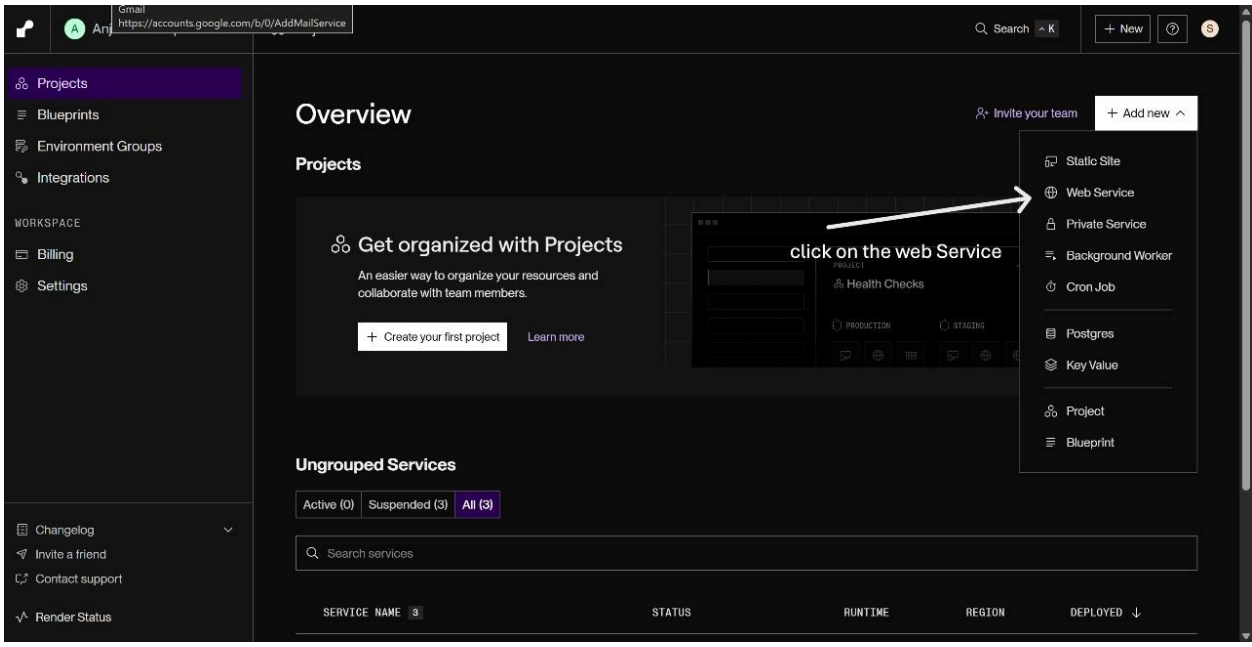
- **Advantages**:
  - Cost-effective for small projects
  - Scalable and flexible
  - No infrastructure management required

ii) Service Model – Platform as a Service (PaaS)

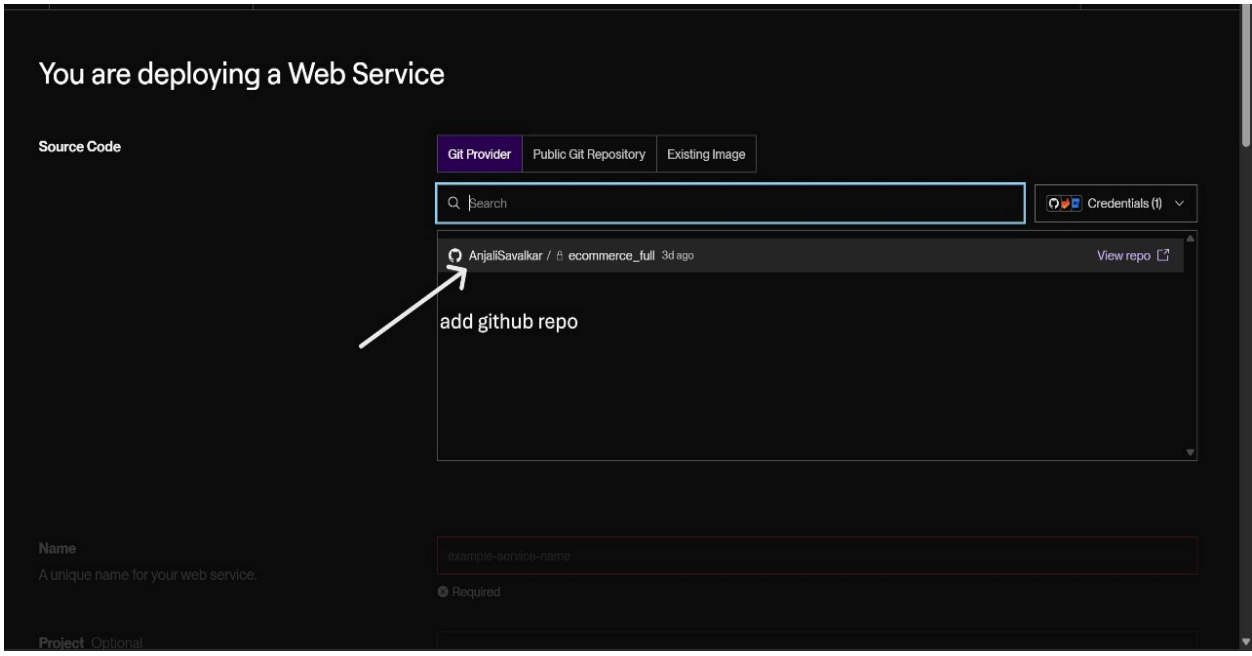
We used Render's **PaaS** capabilities, which allowed us to focus on development without managing the backend infrastructure.

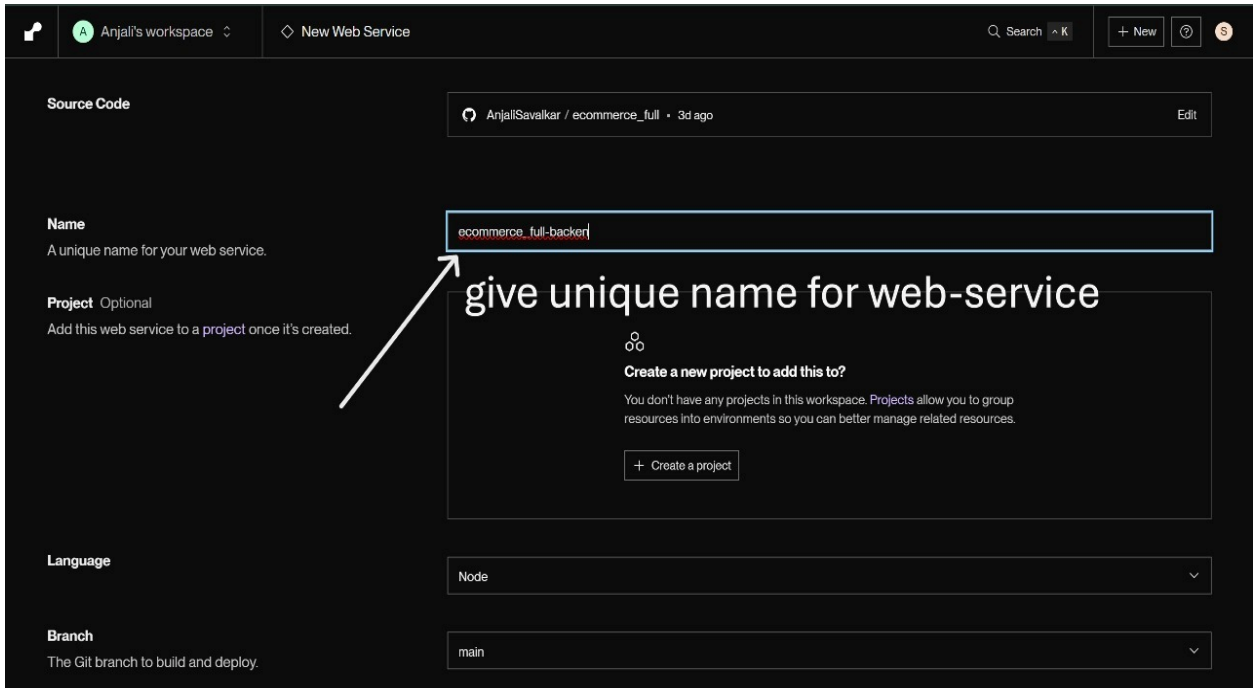
- **Benefits**:
  - Easy CI/CD integration
  - Built-in support for backend/frontend hosting
  - Automated scaling and updates

5. Deployment Steps

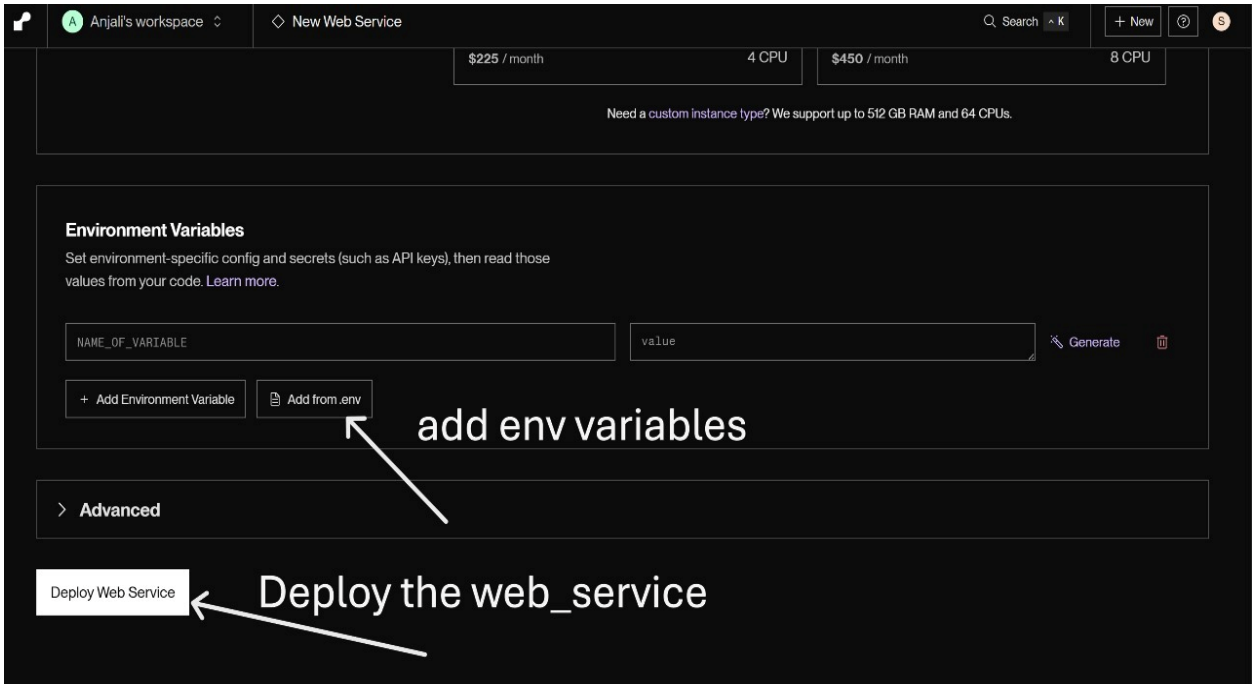
1. **Render Account Setup**
    - Registered at Render.com
    - Connected to our GitHub repository for automatic deployments
    - 

2. **Frontend Deployment**
    - Chose "Static Site" on Render
    - Linked GitHub repo (HTML/CSS/JS or React files)
    - Configured build commands and publish directory
    - Application hosted with a live URL

- 

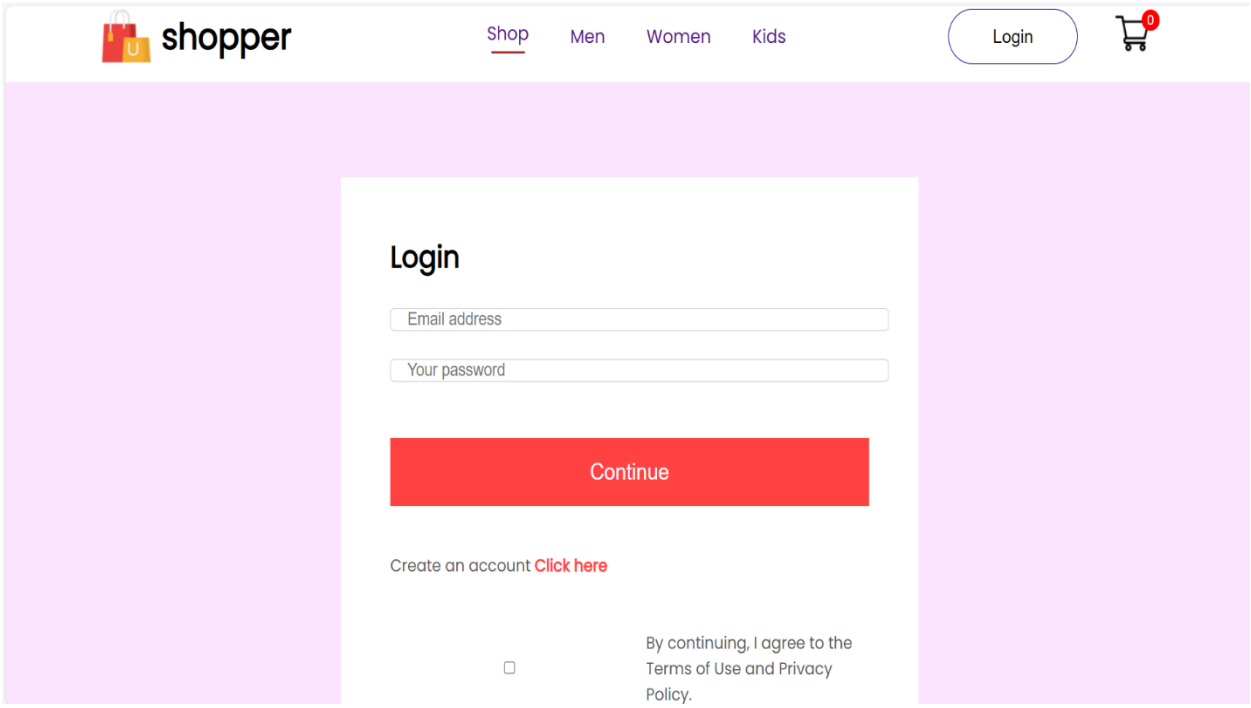
3. 

4. **Backend Deployment**
   - Set up a "Web Service" for Node.js + Express
   - Configured environment variables (e.g., database credentials)
   - Enabled auto-deployments from GitHub branch

- 

6. Hosted Website

Screenshots or links of the deployed site are provided to demonstrate successful deployment and working features.



SHOPPER
Admin Panel

ecommerce-full-admin1.onrender.com says
Product added successfully!
OK

Add Product
Product List

Product title
Shirt
Price
600
Offer Price
500
Product category
Men

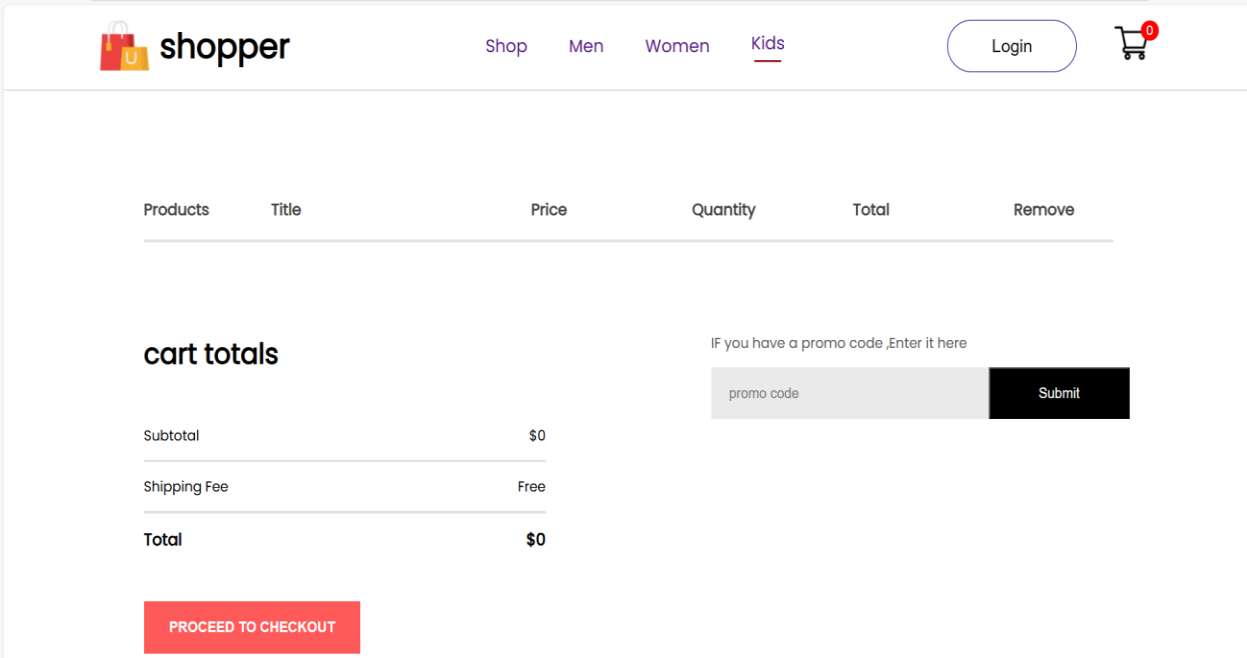
Add

SHOPPER
Admin Panel

Add Product
Product List

All Products List

Product	Title	Old_price	new_price	Category	Remove
	Shirt	$600	$500	men	
	Hooded Jacket	$799	$699	men	
	Denim Jacket	$800	$850	men	
	Jacket	$999	$850	men	



---

7. Conclusion

This project showcases the development and cloud deployment of a full-stack e-commerce application using Render's PaaS platform. With features like authentication, cart management, and admin control, the application ensures a complete shopping experience while leveraging the cloud for scalability and ease of maintenance.

2 . Shopper App - Containerization of the application

Overview

The **Shopper Project** consists of three main components:

- **Frontend (shopper-frontend)**: The user interface for customers.
- **Admin Frontend (shopper-admin)**: A separate interface for managing the application.
- **Backend (shopper-backend)**: The server that handles API requests.

All components are containerized using Docker and connected via a shared Docker network.

Prerequisites

Make sure the following tools are installed:

- **Docker**
- **Node.js**
- **Docker Compose**

Dockerfiles

backend/Dockerfile

```
FROM node:18

WORKDIR /app

COPY package*.json ./

RUN npm install

COPY . .

RUN npm run build

EXPOSE 5050

CMD ["npm", "start"]
```

frontend/Dockerfile

```
FROM node:18

WORKDIR /app

COPY package*.json ./
```

```
RUN npm install
COPY . .
RUN npm run build
RUN npm install -g serve
EXPOSE 5173
CMD ["serve", "-s", "dist"]
```

admin/vite-project/Dockerfile

```
FROM node:18
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
RUN npm install -g serve
EXPOSE 5174
CMD ["serve", "-s", "dist"]
```

Docker Compose Setup

docker-compose.yml

```
version: '3'
services:
  backend:
    build: ./backend
    image: shopper-backend:latest
    container_name: fervent_hoover
    ports:
```

```
    - "5050:5050"
    networks:
      - demo

  admin-frontend:
    build: ./admin/vite-project
    image: shopper-admin:latest
    container_name: reverent_raman
    ports:
      - "5174:5174"
    networks:
      - demo

  user-frontend:
    build: ./frontend
    image: shopper-frontend:latest
    container_name: condescending_menin
    ports:
      - "5173:5173"
    networks:
      - demo

networks:
  demo:
    driver: bridge
```

Setup Instructions

1. Clone the Repository

git clone <repository-url>

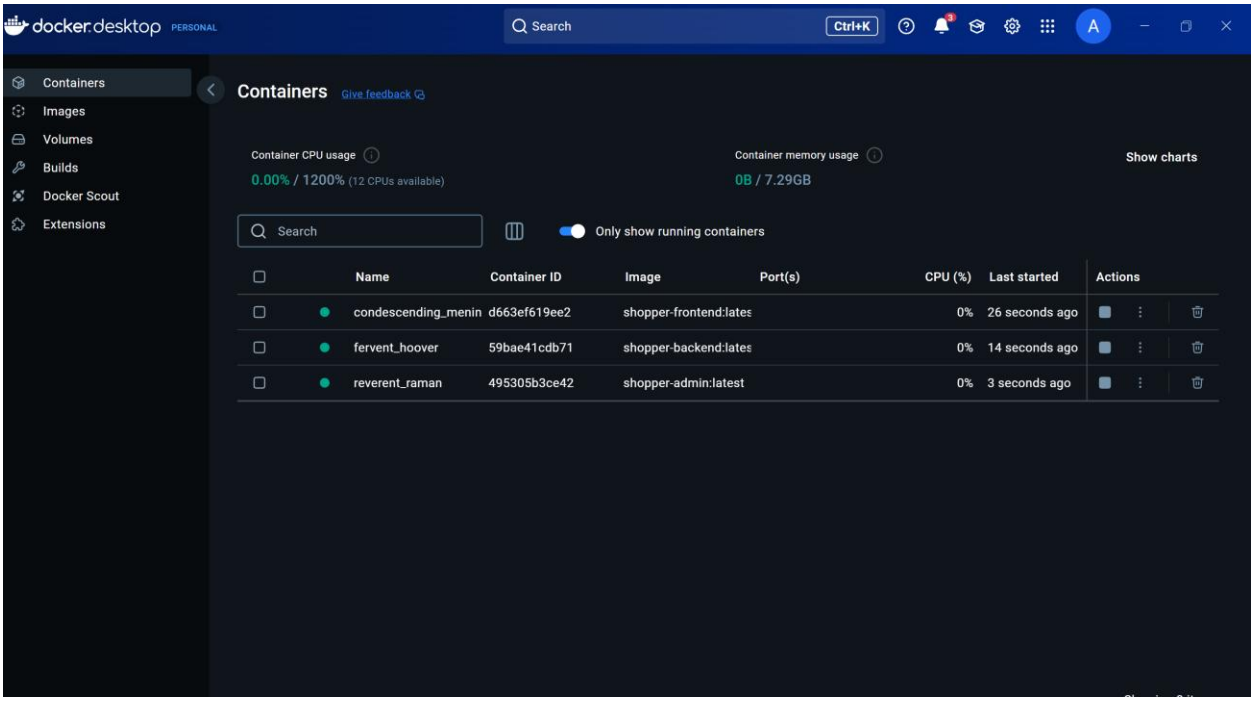
cd shopper

2. Run Docker Compose

docker-compose up --build

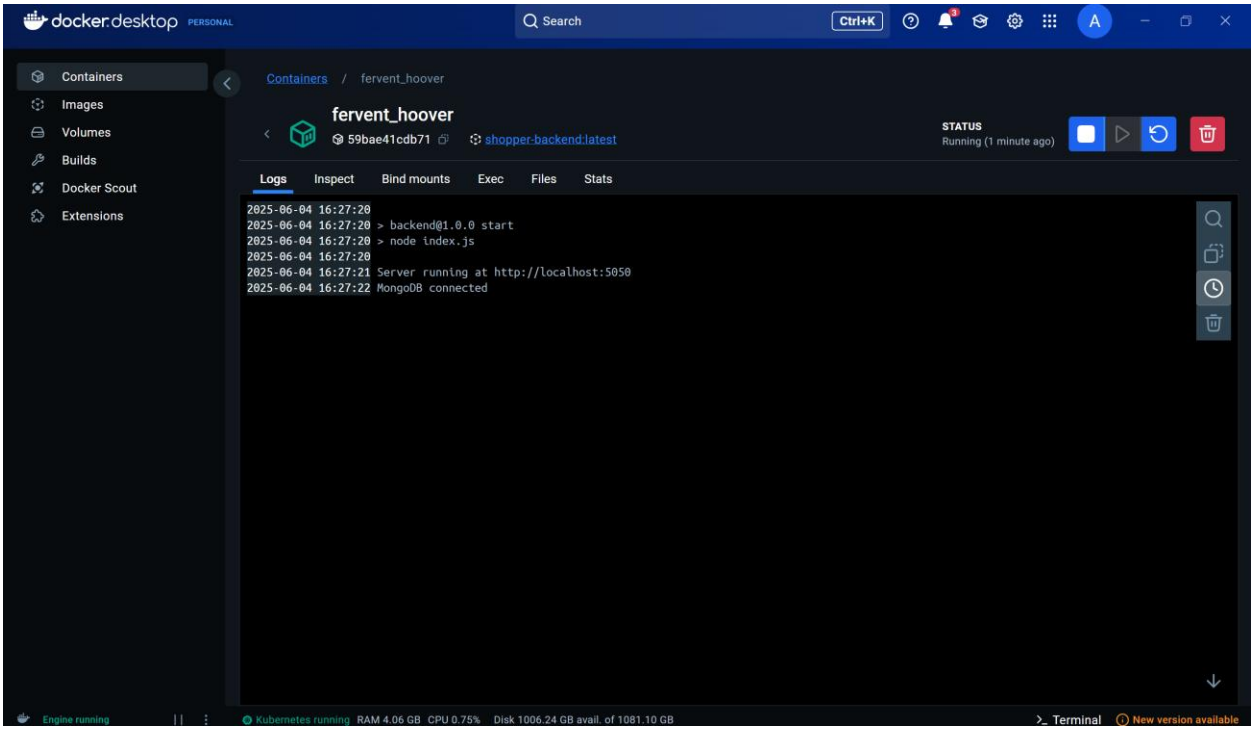
This command will build and run all three containers (backend, admin-frontend, user-frontend) and connect them to the demo network.

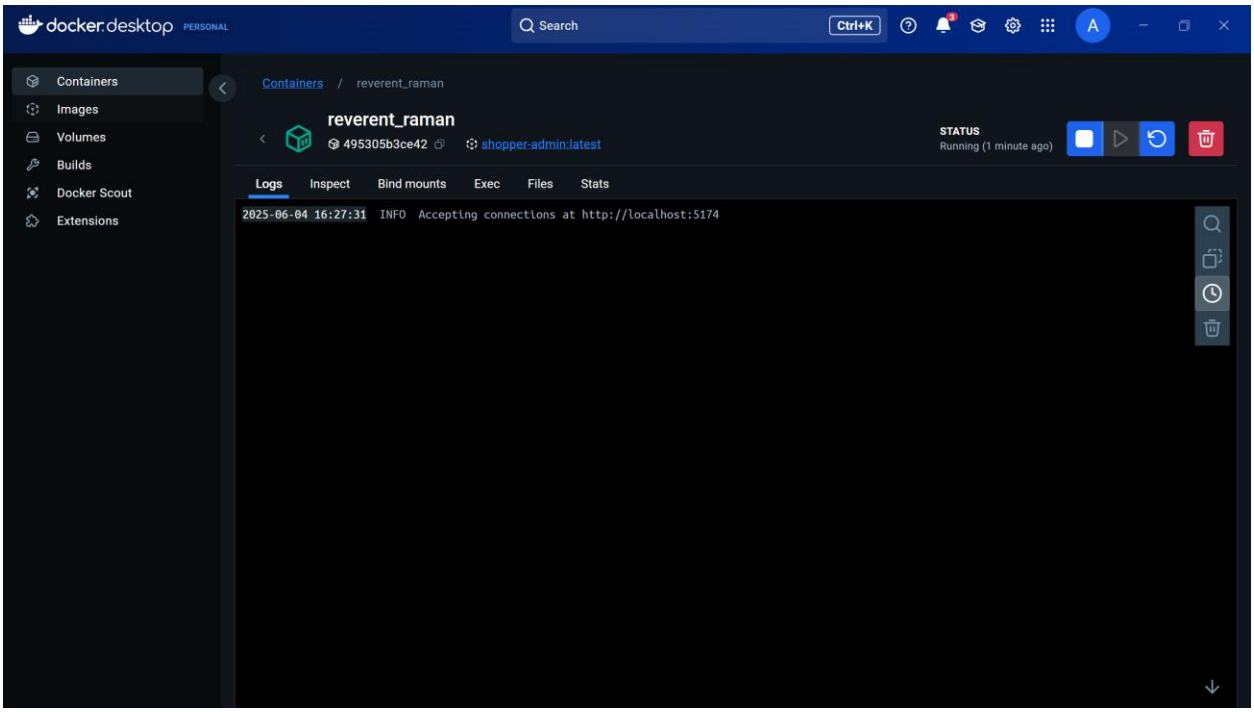
Verifying the Setup

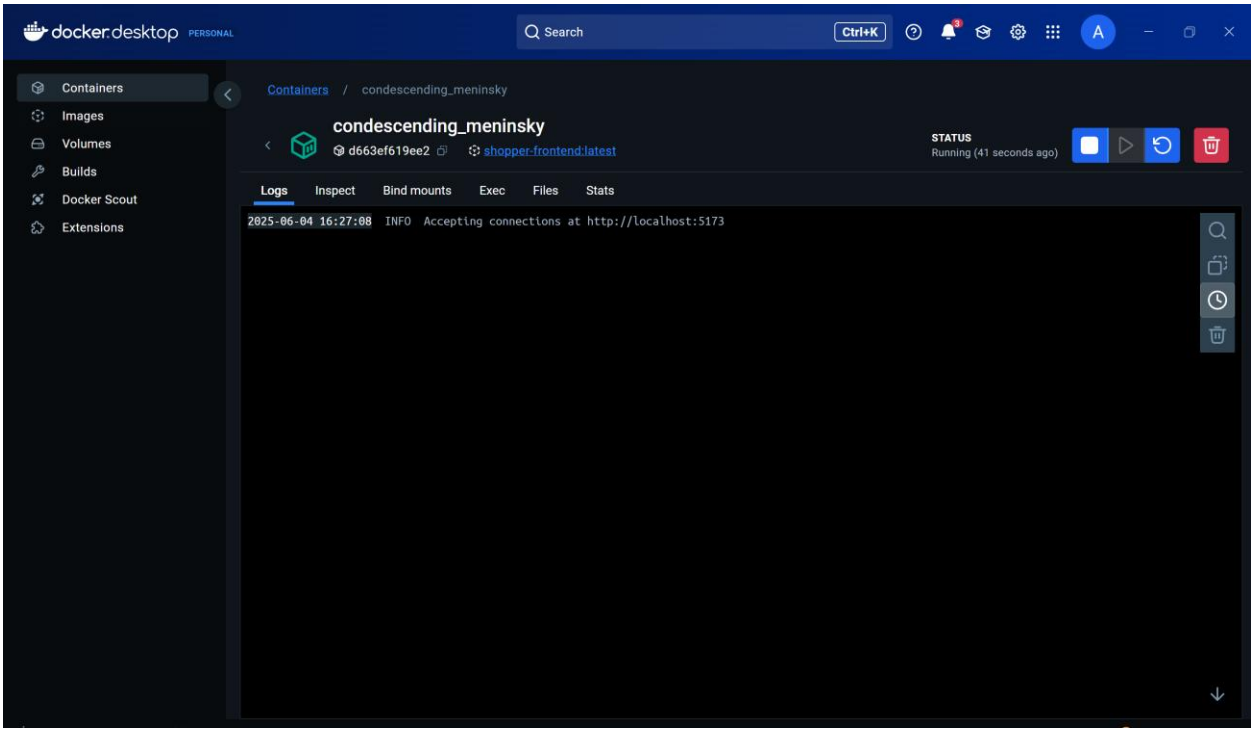


Check Running Containers

docker ps


docker desktop PERSONAL
Search
Ctrl+K
Containers
Images
Volumes
Builds
Docker Scout
Extensions
Containers / fervent_hoover
fervent_hoover
59bae41cdb71
shopper-backend:latest
STATUS
Running (1 minute ago)
Logs
Inspect
Bind mounts
Exec
Files
Stats
2025-06-04 16:27:20
2025-06-04 16:27:20 > backend@1.0.0 start
2025-06-04 16:27:20 > node index.js
2025-06-04 16:27:20
2025-06-04 16:27:21 Server running at http://localhost:5050
2025-06-04 16:27:22 MongoDB connected
Engine running
Kubernetes running RAM 4.06 GB CPU 0.75% Disk 1006.24 GB avail. of 1081.10 GB
Terminal
New version available


docker desktop PERSONAL
Search
Ctrl+K
Containers
Images
Volumes
Builds
Docker Scout
Extensions
Containers / reverent_raman
reverent_raman
495305b3ce42
shopper-admin:latest
STATUS
Running (1 minute ago)
Logs
Inspect
Bind mounts
Exec
Files
Stats
2025-06-04 16:27:31 INFO Accepting connections at http://localhost:5174



Access URLs

- User Frontend: http://localhost:5173
- Admin Frontend: http://localhost:5174
- Backend API: http://localhost:5050

---

3 . Docker Kubernetes – todo app

📝 Todo App Deployment Documentation

Overview

This documentation outlines the complete process of containerizing a static frontend Todo application using Docker and deploying it on a local Kubernetes cluster via Minikube.

📁 Tech Stack

- **Frontend**: HTML, CSS, JavaScript
- **Web Server**: Nginx (used for serving static content)
- **Containerization**: Docker
- **Orchestration**: Kubernetes (Minikube)

🚀 Project Structure

todo-list-main/

├── index.html

├── styles.css

├── app.js

├── Dockerfile

└── README.md

Todo App Deployment Documentation

Overview

This documentation outlines the complete process of creating, containerizing, and deploying a static frontend Todo application using Docker and Kubernetes via Minikube.

Project Structure

```
todo-list-main/
├── index.html        # Main HTML page for the app
├── styles.css        # Styling for the Todo UI
├── app.js            # JavaScript to manage Todo items
├── Dockerfile        # Containerizes the app using Nginx
└── README.md         # Project overview and deployment instructions
```

Code Files

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>Todo App</title>
  <link rel="stylesheet" href="styles.css" />
</head>
<body>
  <div class="container">
    <h1>Todo List</h1>
    <input type="text" id="todo-input" placeholder="Add a new task" />
```

```
    <button onclick="addTodo()">Add</button>
    <ul id="todo-list"></ul>
  </div>
  <script src="app.js"></script>
</body>
</html>
```

styles.css

```
body {
  font-family: Arial, sans-serif;
  background: #f9f9f9;
  padding: 50px;
}

.container {
  background: white;
  padding: 20px;
  max-width: 500px;
  margin: auto;
  border-radius: 5px;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}

input[type="text"] {
  width: 70%;
  padding: 10px;
  margin: 10px 0;
```

```
}

button {
  padding: 10px;
}

ul {
  list-style: none;
  padding: 0;
}

li {
  padding: 8px;
  border-bottom: 1px solid #ddd;
}
```

app.js

```
function addTodo() {
  const input = document.getElementById('todo-input');
  const text = input.value.trim();
  if (text === '') return;

  const list = document.getElementById('todo-list');
  const li = document.createElement('li');
  li.textContent = text;
  list.appendChild(li);
```

```
  input.value = '';
}
```

Deployment Steps

Step 1: Create the Todo App

Develop your basic HTML, CSS, and JS Todo list app.

Step 2: Dockerize the App

Create a Dockerfile:

```
FROM nginx:alpine
COPY . /usr/share/nginx/html
EXPOSE 80
```

Build the Docker image:

```
docker build -t todo-app .
```

Run locally (optional):

```
docker run -d -p 8080:80 todo-app
```

Test: http://localhost:8080

Step 3: Push Image to Docker Hub (Optional)

```
docker login
docker tag todo-app your-username/todo-app
docker push your-username/todo-app
```

Step 4: Start Minikube

```
minikube start --nodes 2
```

Step 5: Create Kubernetes Manifests

deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: todo-frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: todo
  template:
    metadata:
      labels:
        app: todo
    spec:
      containers:
      - name: todo-container
        image: todo-app
        ports:
        - containerPort: 80
```

service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: todo-service
spec:
```

```
  type: NodePort
  selector:
    app: todo
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 30036
```

Apply the files:

```
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
```

Step 6: Access the Application

Alternative (if NodePort is not accessible):

```
kubectl port-forward service/todo-service 8080:80
```

Then open: http://localhost:8080

Notes

- Avoid referencing non-existent folders like ./dist in your Dockerfile unless it exists.
- You can replace **Nginx** with any other static file server if preferred.
- To view deployed pods and services:

```
kubectl get pods
kubectl get services
```

Conclusion

This guide provides a step-by-step process to containerize and deploy a simple static frontend app on Kubernetes. It's ideal for learning containerization, Docker basics, and Kubernetes deployment via Minikube.

Hpa and vpa

Horizontal Pod Autoscaling (HPA) for Todo App in Kubernetes

This guide walks you through setting up Horizontal Pod Autoscaling (HPA) for a simple Todo application deployed in a Kubernetes environment using Minikube.

Prerequisites

- Minikube installed and configured
- kubectl command-line tool installed
- Basic Todo app files: index.html, styles.css, app.js

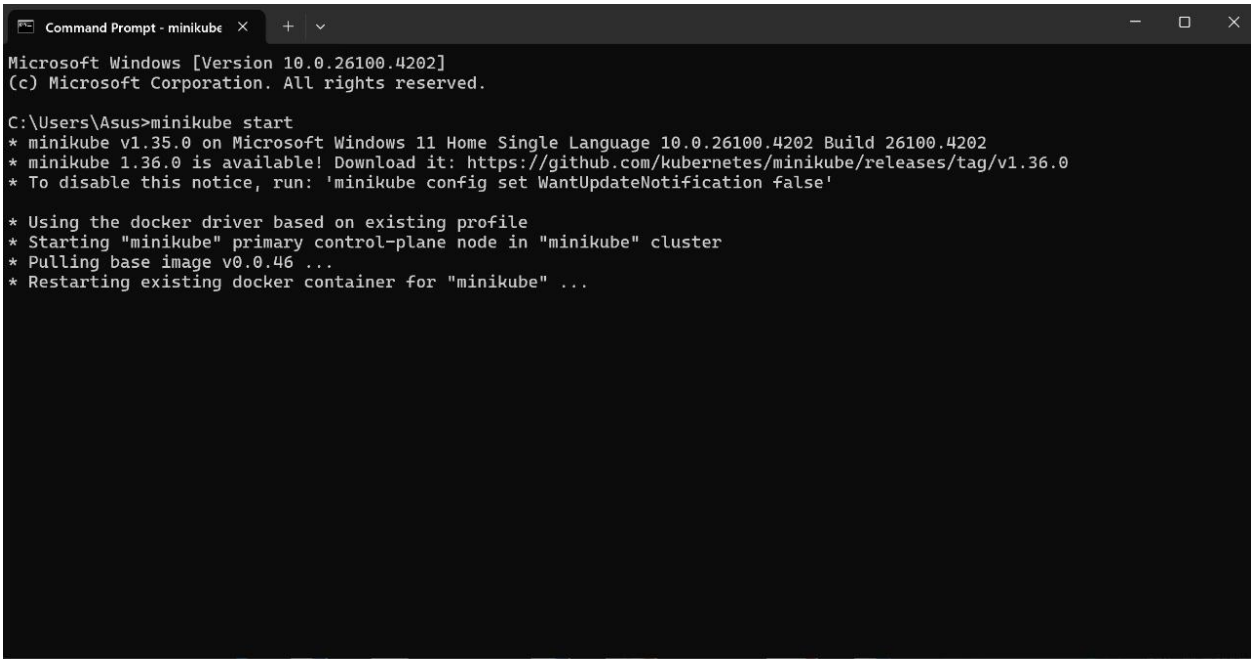
Step 1: Start Minikube with Metrics Server

HPA requires metrics to auto-scale pods. The Metrics Server provides this.

```bash
# Start Minikube
minikube start

# Enable metrics-server addon
minikube addons enable metrics-server

# Verify metrics-server is running
kubectl get pods -n kube-system | grep metrics-server
```




```

**Step 2: Create ConfigMap for Todo App Files**

Since we use a public Nginx image, create a ConfigMap to inject your app files:

```bash
kubectl create configmap todo-app-files --from-file=index.html --from-file=styles.css --from-file=app.js
```

Step 3: Create Deployment

Create a file named todo-deployment.yaml:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: todo-app
  labels:
    app: todo-app
spec:
  replicas: 2
  selector:
    matchLabels:
```
```

```
    app: todo-app
  template:
    metadata:
      labels:
        app: todo-app
    spec:
      containers:
      - name: todo-app
        image: nginx:alpine
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
          limits:
            cpu: 200m
            memory: 256Mi
        volumeMounts:
        - name: config-volume
          mountPath: /usr/share/nginx/html
      volumes:
      - name: config-volume
        configMap:
          name: todo-app-files
```

Apply the deployment:

```bash
kubectl apply -f todo-deployment.yaml
```

Step 4: Create Service

Create a file named todo-service.yaml to expose your app:

```yaml
apiVersion: v1
```

```
kind: Service
metadata:
  name: todo-app-service
spec:
  selector:
    app: todo-app
  ports:
  - port: 80
    targetPort: 80
  type: NodePort
```

Apply the service:

```bash
kubectl apply -f todo-service.yaml
```

Step 5: Create Horizontal Pod Autoscaler

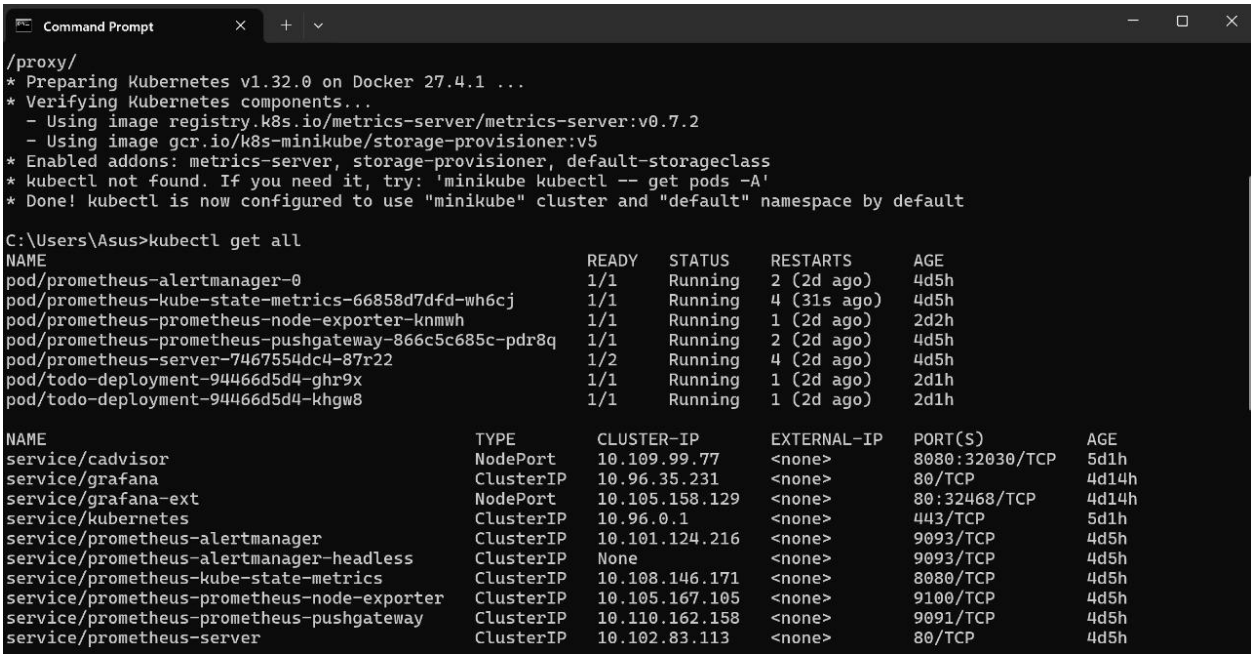
Create a file named todo-hpa.yaml to enable autoscaling based on CPU utilization:

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: todo-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: todo-app
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
```

averageUtilization: 70
```

Apply the HPA:

```bash
kubectl apply -f todo-hpa.yaml
```



Step 6: Verify and Monitor HPA

Check the status of the Horizontal Pod Autoscaler:

```bash
kubectl get hpa todo-app-hpa
```

Expected output:

```
NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE
todo-app-hpa Deployment/todo-app 0%/70% 2 10 2 X
```

Step 7: Access Your Todo App

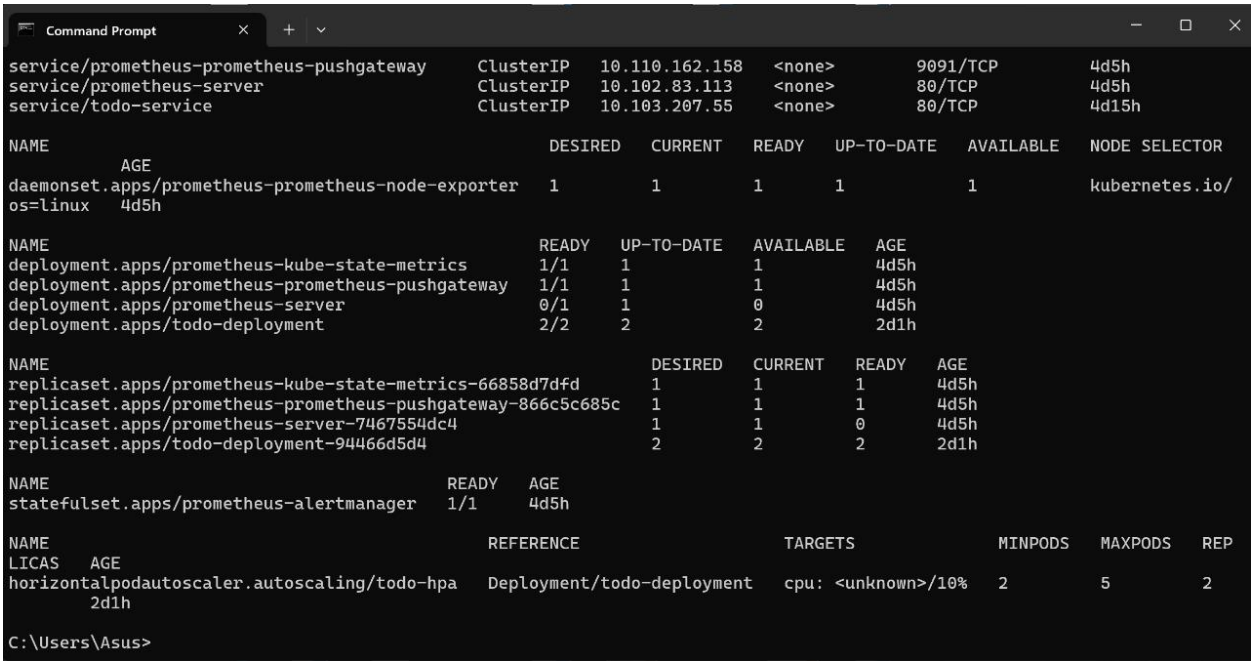
Get the URL to access the Todo app:
```

```bash
minikube service todo-app-service --url
```

Or use port forwarding:

```bash
kubectl port-forward service/todo-app-service 8080:80
```

Open the URL in your browser or access via http://localhost:8080.



Step 8: Generate Load to Test Autoscaling

Create load-generator.yaml to simulate traffic and increase CPU usage:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: load-generator
spec:
  replicas: 1
  selector:
    matchLabels:
      app: load-generator
```

```
  template:
    metadata:
      labels:
        app: load-generator
    spec:
      containers:
      - name: load-generator
        image: busybox
        command: ["/bin/sh", "-c"]
        args:
        - while true; do
            wget -q -O- http://todo-app-service;
            sleep 0.01;
          done
```

Apply the load generator:

```bash
kubectl apply -f load-generator.yaml
```

Step 9: Monitor Autoscaling in Action

Watch the HPA scaling your pods as CPU utilization rises:

```bash
kubectl get hpa todo-app-hpa --watch
```

In another terminal, watch pod creation:

```bash
kubectl get pods -w
```

You should see new pods created as load increases.

Step 10: Clean Up Resources

When finished, remove all Kubernetes objects created:

```bash
kubectl delete -f load-generator.yaml
```

kubectl delete -f todo-hpa.yaml
kubectl delete -f todo-service.yaml
kubectl delete -f todo-deployment.yaml
kubectl delete configmap todo-app-files
```

**Troubleshooting**

**Metrics Not Showing**

If HPA shows <unknown>/70%:

```bash
minikube addons disable metrics-server
minikube addons enable metrics-server
# Wait a few minutes for metrics collection to start
```

Pods Not Scaling

Check current pod CPU usage:

```bash
kubectl top pods
```

Ensure the load generator is creating enough traffic.

Other Checks

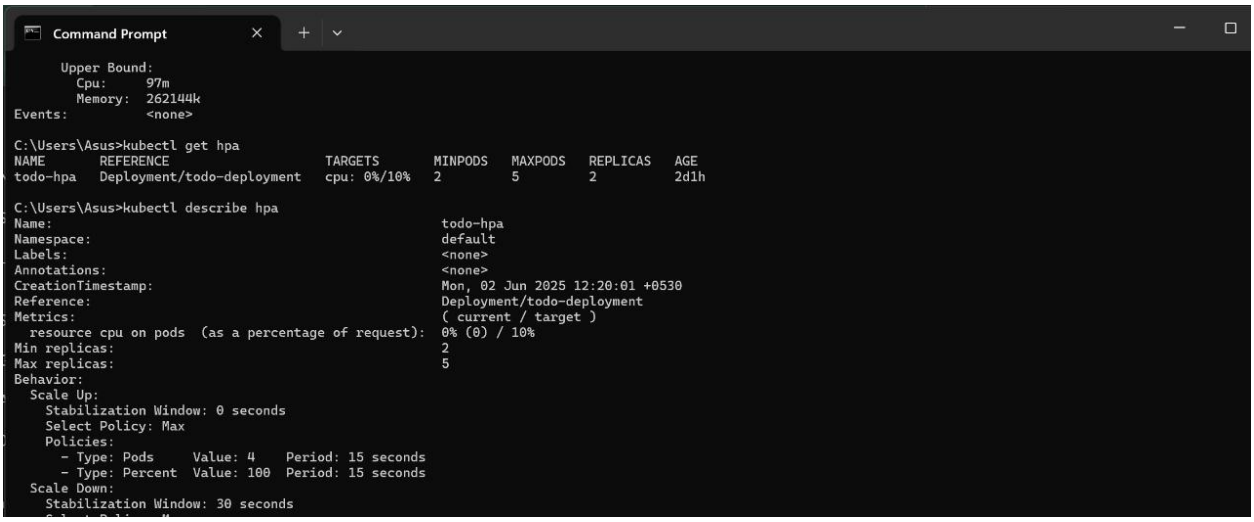
View recent cluster events:

```bash
kubectl get events --sort-by='.lastTimestamp'
```

Describe HPA for details:

```bash
kubectl describe hpa todo-app-hpa
```
```

```

```
C:\Users\Asus>kubectl get hpa
NAME   REFERENCE   TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
todo-hpa   Deployment/todo-deployment   cpu: 0%/10%   2   5   2   2d1h

C:\Users\Asus>
```




**Understanding HPA Behavior**

- HPA checks metrics every 15 seconds by default.
- It averages metrics over a window to prevent rapid scaling fluctuations.
- Scale-down cooldown is longer than scale-up cooldown to avoid flapping.
- Autoscaling reactions may take several minutes to reflect.

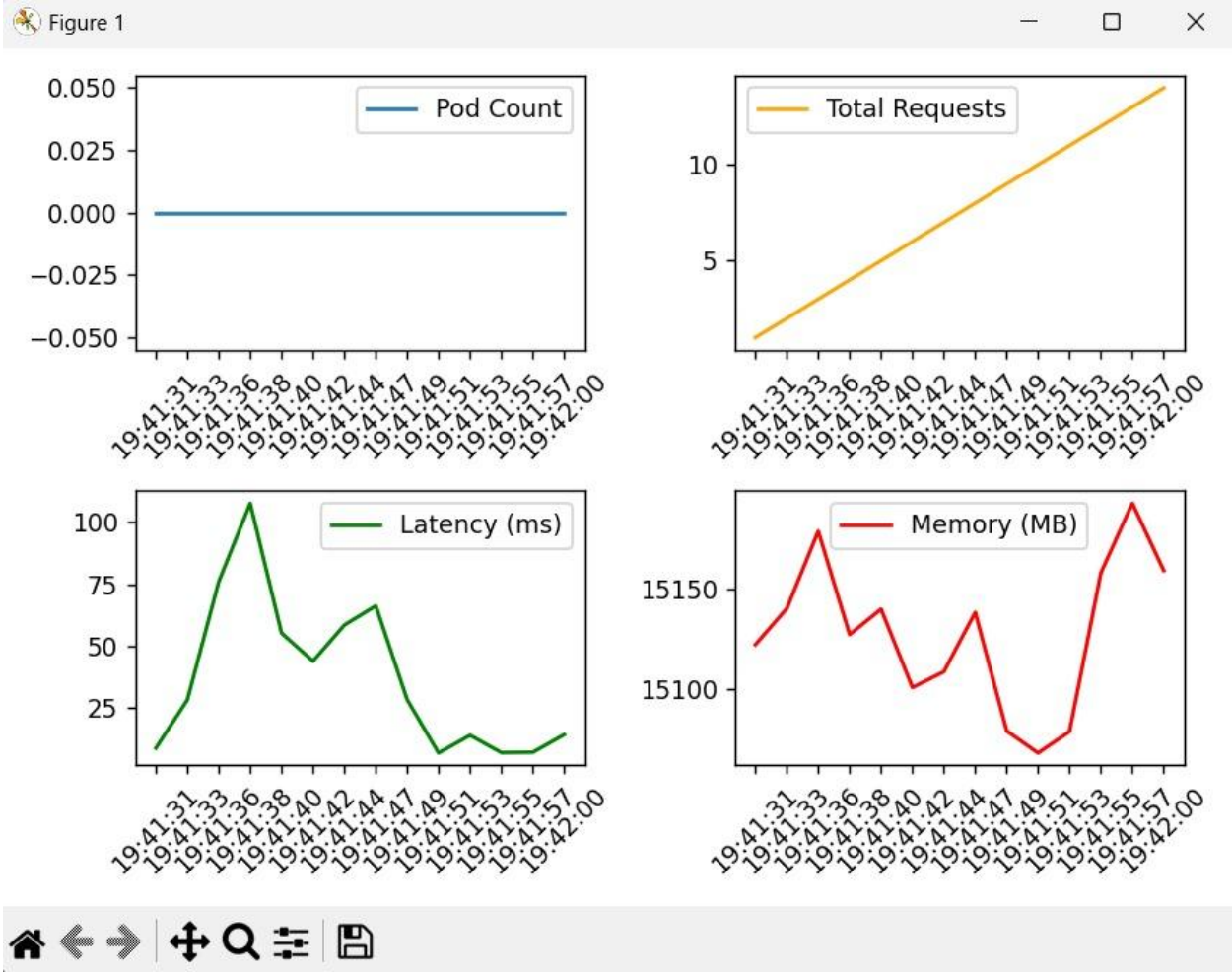
**Alternative: Using a Custom Docker Image**

For production use, consider building your own Docker image instead of using ConfigMaps:
1. Build and push your Docker image to a container registry (e.g., Docker Hub).
2. Update your deployment to use your image and remove ConfigMap volume mounts:

```yaml
spec:
  containers:
  - name: todo-app
    image: yourusername/your-todo-app:latest
    # Remove the configMap volumeMounts and volumes sections
```

This method simplifies deployment and improves performance in production environments.
```

Latency and Throughput Graph in Graphana



Vpa

Vertical Pod Autoscaler (VPA) Report for TODO Application in Kubernetes

This guide explains how to generate a Vertical Pod Autoscaler (VPA) report for a TODO application running on Kubernetes. The report provides recommended CPU and memory usage for better resource optimization.

Prerequisites

• You have a TODO application running as a Kubernetes Deployment

• VPA is installed and configured on your cluster (as shown in the guide you referenced)

• Metrics server is installed and running

Step 1: Create the VPA Resource

Create a file named `vpa.yaml` for your TODO app. Here's an example VPA resource:

```yaml
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: todo-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: todo-deployment
  updatePolicy:
    updateMode: "Auto"  # Or use "Off" to only observe recommendations
```

Apply it:

```bash
kubectl apply -f vpa.yaml
```

Step 2: Generate Load (Optional but recommended)

To see meaningful recommendations, generate some traffic/load to the TODO application. You can use `hey` or `ab`, or a simple busybox pod like:

```bash
kubectl run -i --tty load-tester --image=busybox --restart=Never -- /bin/sh
```

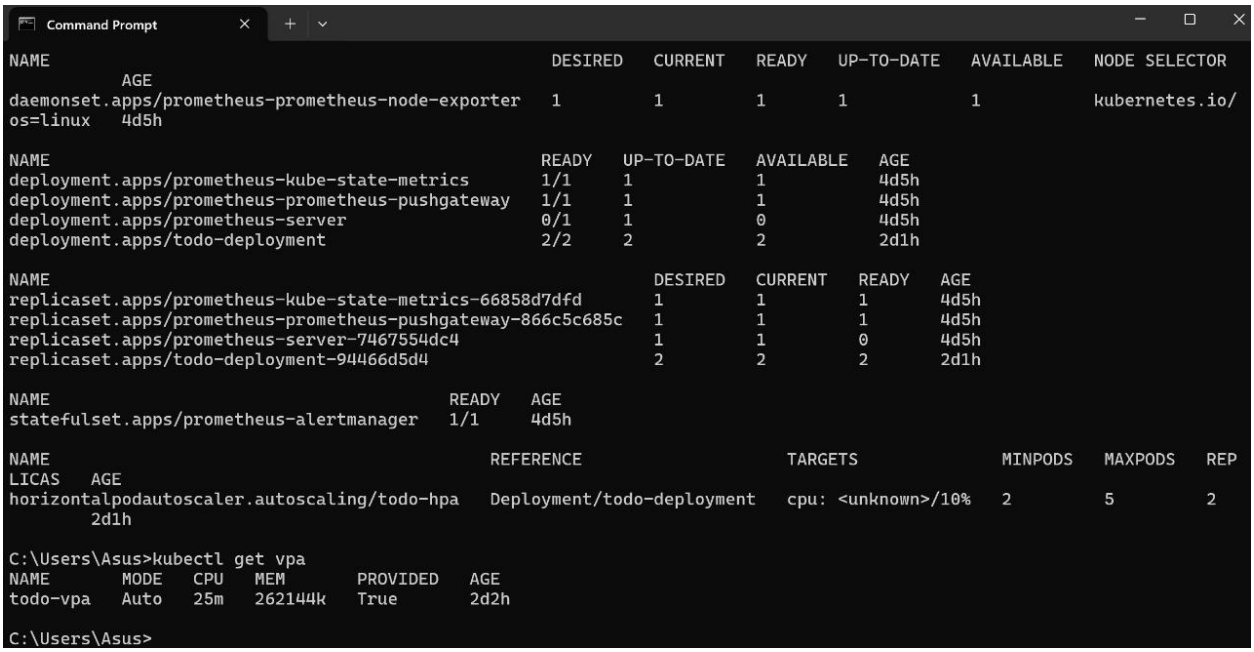
 # Inside the shell
while true; do wget -q -O- http://todo-service; done
```

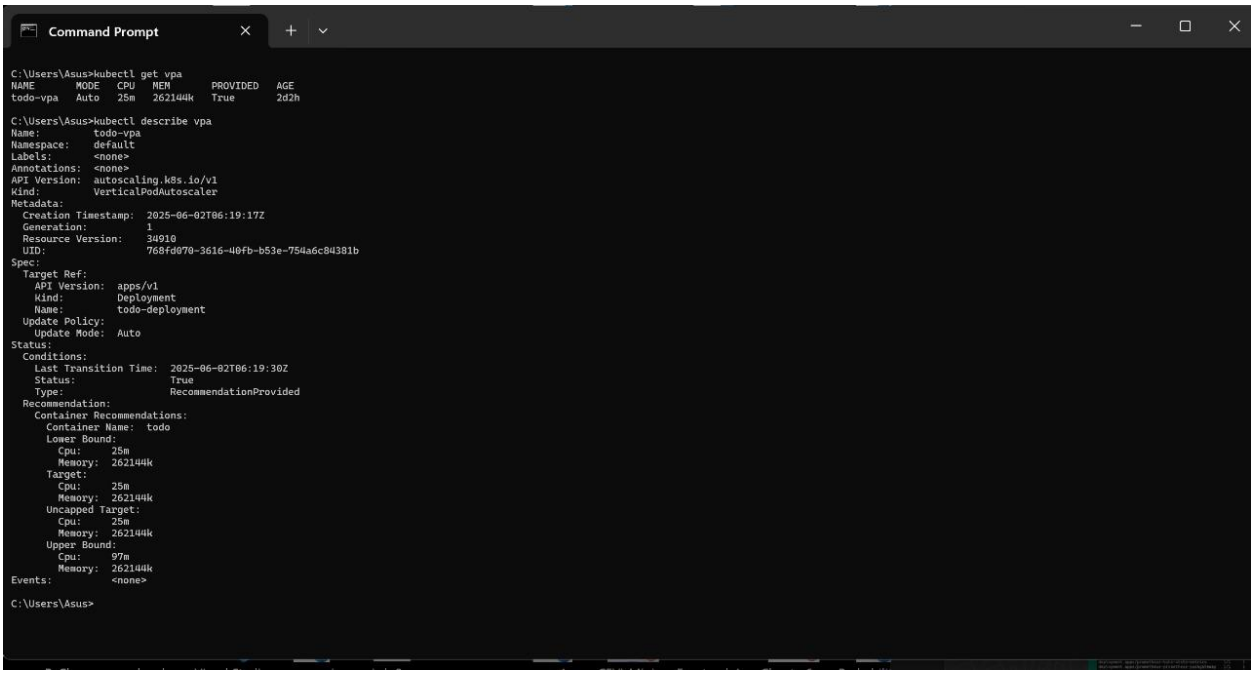
***Replace `todo-service` with the correct service name.***

**Step 3: View VPA Report**

After some time (a few minutes of running load), run the following command to get the VPA recommendations:

```***bash***
kubectl get vpa todo-vpa -o yaml
```


```

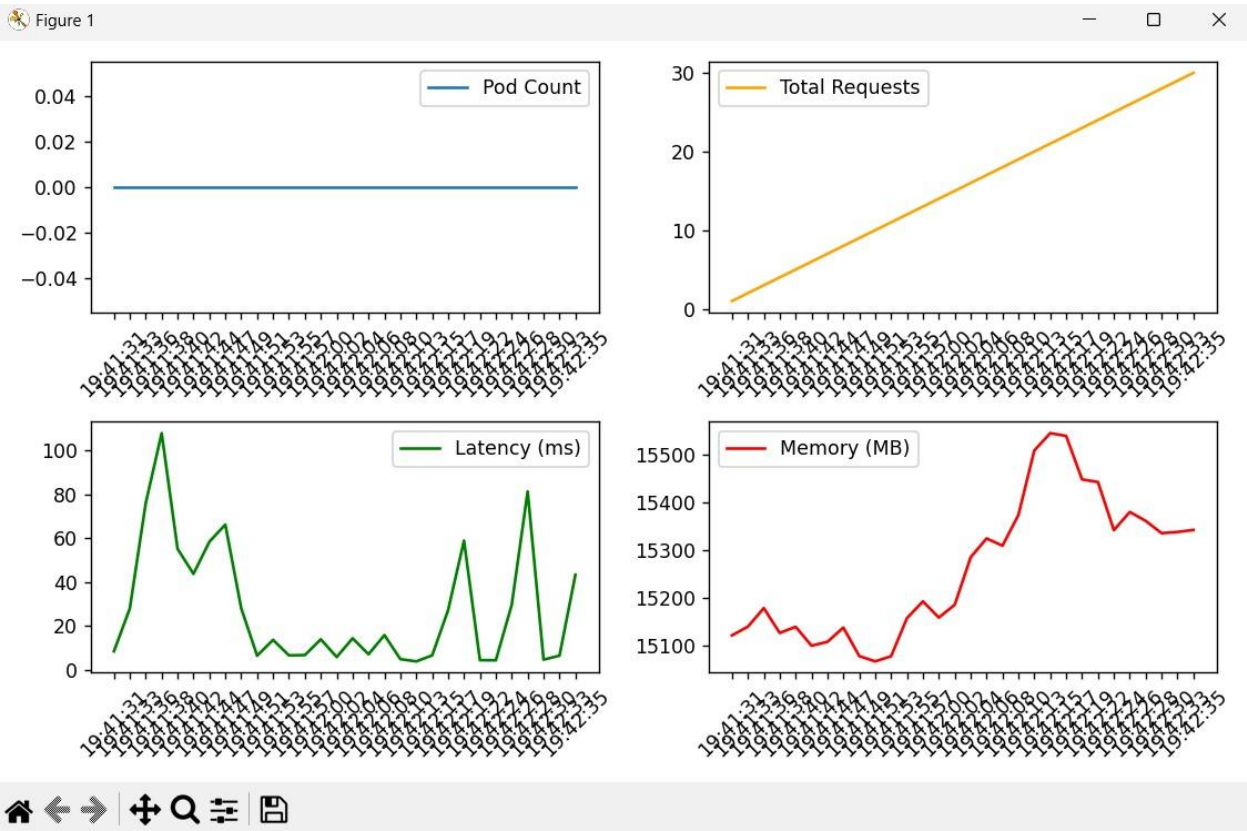


Sample Output

Here's an example snippet of what the VPA output may look like:

```yaml
status:
  recommendation:
    containerRecommendations:
    - containerName: todo-container
      lowerBound:
        cpu: 20m
        memory: 100Mi
      target:
        cpu: 50m
        memory: 200Mi
      upperBound:
        cpu: 100m
        memory: 400Mi
```

Latency and Throughput Graph in Graphana



4 Microservices Cab Booking App with 4 Containers

Step 1: Define Your 4 Microservices

For a cab booking app, 4 common services could be:

1. User Service - User login/profile UI (React or plain HTML + JS)
2. Cab Search - Search available cabs UI (React or Vue)
3. Booking Service - Booking confirmation UI (React or Angular)
4. Payment Service - Payment UI (mock payment) (React or plain HTML + JS)
Since no backend, these will be UI-only apps that simulate the flow and talk to each other via mock APIs or localStorage or simple JSON.

Step 2: Setup Your Environment

- You need Docker installed on your machine
- You need a code editor (VSCode preferred)
- Basic knowledge of React or HTML + JS (React for consistency)

Step 3: Create Each Service as a React App

We'll create 4 React apps, each a simple frontend showing their UI and having buttons or forms.

Step 4: Sample Code for Each Service

4.1 User Service

Purpose: Login screen (simulate login)

Create a React app user-service:

```
npx create-react-app user-service
cd user-service
```

Replace src/App.js with:

```
import React, { useState } from 'react';

function App() {
  const [username, setUsername] = useState('');
  const [loggedInUser, setLoggedInUser] = useState(localStorage.getItem('user') || '');

  const login = () => {
    localStorage.setItem('user', username);
    setLoggedInUser(username);
  };

  const logout = () => {
    localStorage.removeItem('user');
    setLoggedInUser('');
    setUsername('');
  };

  return (
```

```
    <div style={{ padding: 20 }}>
      <h2>User Service</h2>
      {loggedInUser ? (
        <div>
          <p>Welcome, {loggedInUser}</p>
          <button onClick={logout}>Logout</button>
        </div>
      ) : (
        <div>
          <input
            type="text"
            placeholder="Enter username"
            value={username}
            onChange={e => setUsername(e.target.value)}
          />
          <button onClick={login} disabled={!username}>Login</button>
        </div>
      )}
    </div>
  );
}

export default App;
```

4.2 Cab Search Service

Create cab-search-service:
npx create-react-app cab-search-service
cd cab-search-service

Replace src/App.js with:

```
import React, { useState } from 'react';

const cabs = [
```

```
  { id: 1, name: 'Cab A', location: 'Downtown' },
  { id: 2, name: 'Cab B', location: 'Airport' },
  { id: 3, name: 'Cab C', location: 'Suburbs' },
];

function App() {
  const [selectedCab, setSelectedCab] = useState(null);

  const bookCab = (cab) => {
    localStorage.setItem('selectedCab', JSON.stringify(cab));
    alert(`Cab ${cab.name} selected. Proceed to Booking Service.`);
  };

  return (
    <div style={{ padding: 20 }}>
      <h2>Cab Search Service</h2>
      <ul>
        {cabs.map(cab => (
          <li key={cab.id}>
            {cab.name} - {cab.location}{' '}
            <button onClick={() => bookCab(cab)}>Select</button>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default App;
```

4.3 Booking Service

Create booking-service:

```
npx create-react-app booking-service
cd booking-service
```

Replace src/App.js with:

```
import React, { useState, useEffect } from 'react';

function App() {
  const [cab, setCab] = useState(null);
  const [bookingConfirmed, setBookingConfirmed] = useState(false);

  useEffect(() => {
    const selectedCab = localStorage.getItem('selectedCab');
    if (selectedCab) {
      setCab(JSON.parse(selectedCab));
    }
  }, []);

  const confirmBooking = () => {
    localStorage.setItem('bookingConfirmed', 'true');
    setBookingConfirmed(true);
  };

  return (
    <div style={{ padding: 20 }}>
      <h2>Booking Service</h2>
      {cab ? (
        <>
          <p>Selected Cab: {cab.name}</p>
          {!bookingConfirmed ? (
            <button onClick={confirmBooking}>Confirm Booking</button>
          ) : (
            <p>Booking Confirmed!</p>
          )}
        </>
      ) : (
        <p>No cab selected. Please select a cab from the Cab Search Service.</p>
      )}
    </div>
```

```
  );
}

export default App;
```

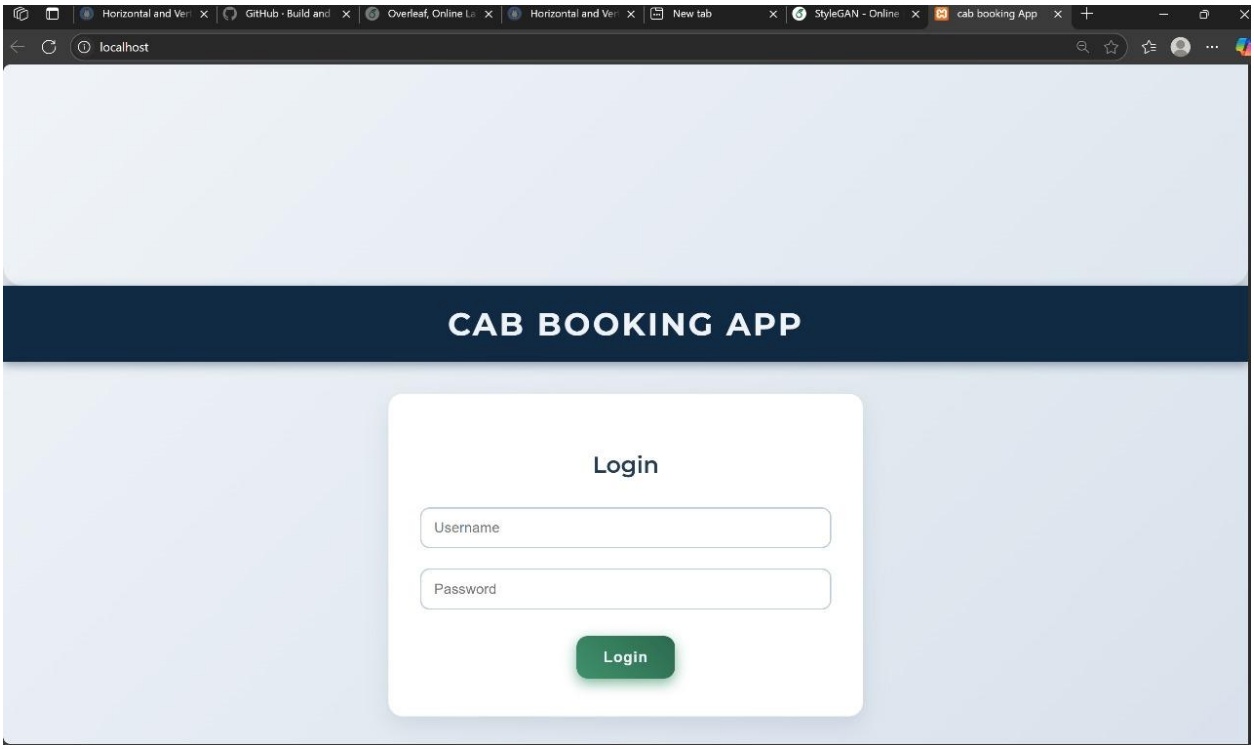
4.4 Payment Service

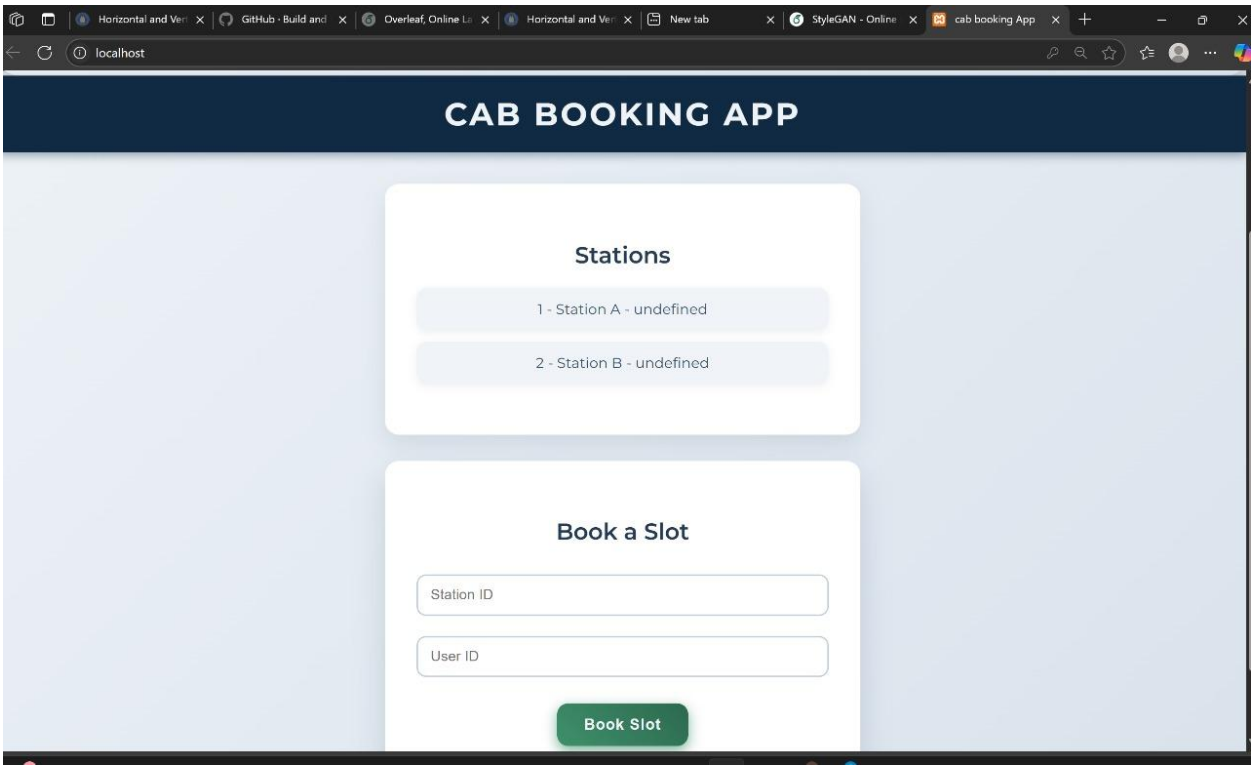
Create payment-service:
npx create-react-app payment-service
cd payment-service

Replace src/App.js with:

```
import React, { useState, useEffect } from 'react';

function App() {
  const [bookingConfirmed, setBookingConfirmed] = useState(false);
  const [paymentDone, setPaymentDone] = useState(false);

  useEffect(() => {
    const booking = localStorage.getItem('bookingConfirmed');
    if (booking === 'true') {
      setBookingConfirmed(true);
    }
  }, []);

  const makePayment = () => {
    setPaymentDone(true);
    alert('Payment successful! Thank you.');
  };

  return (
    <div style={{ padding: 20 }}>
      <h2>Payment Service</h2>
      {!bookingConfirmed ? (
```

```
      <p>Please confirm booking first in Booking Service.</p>
    ) : (
      <>
       {!paymentDone ? (
         <button onClick={makePayment}>Pay Now</button>
       ) : (
         <p>Payment completed. Have a safe journey!</p>
       )}
      </>
    )}
   </div>
  );
}

export default App;
```





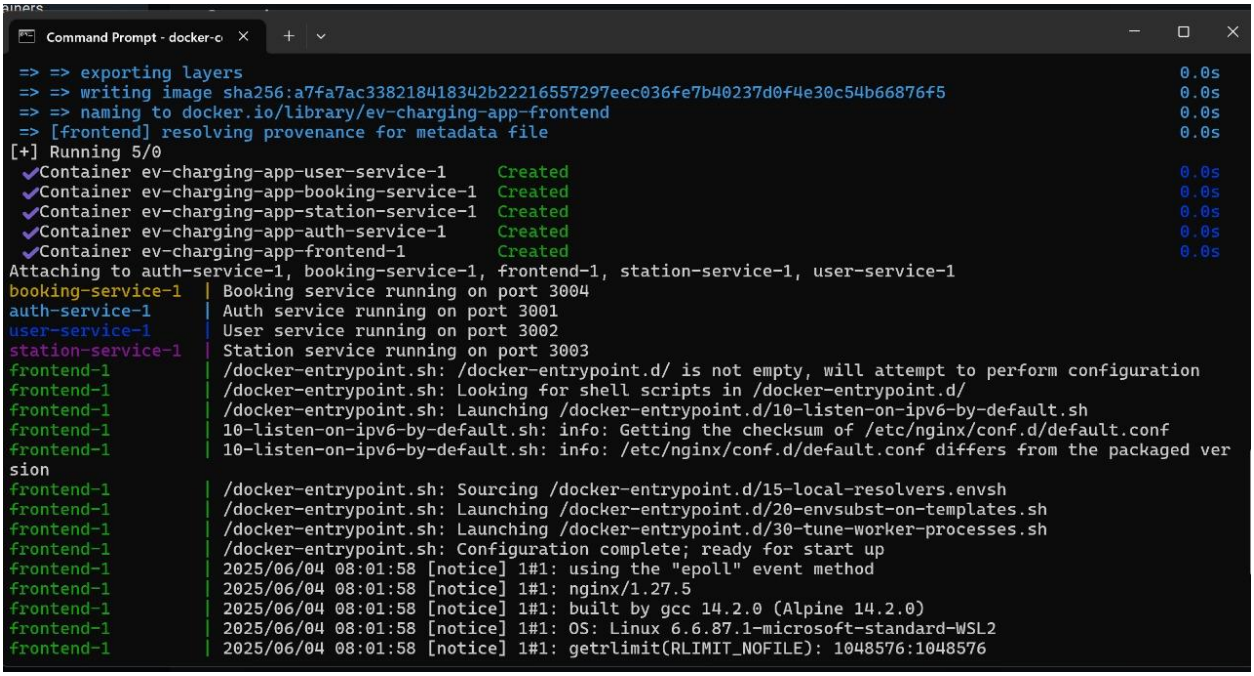
Step 5: Dockerize Each Service

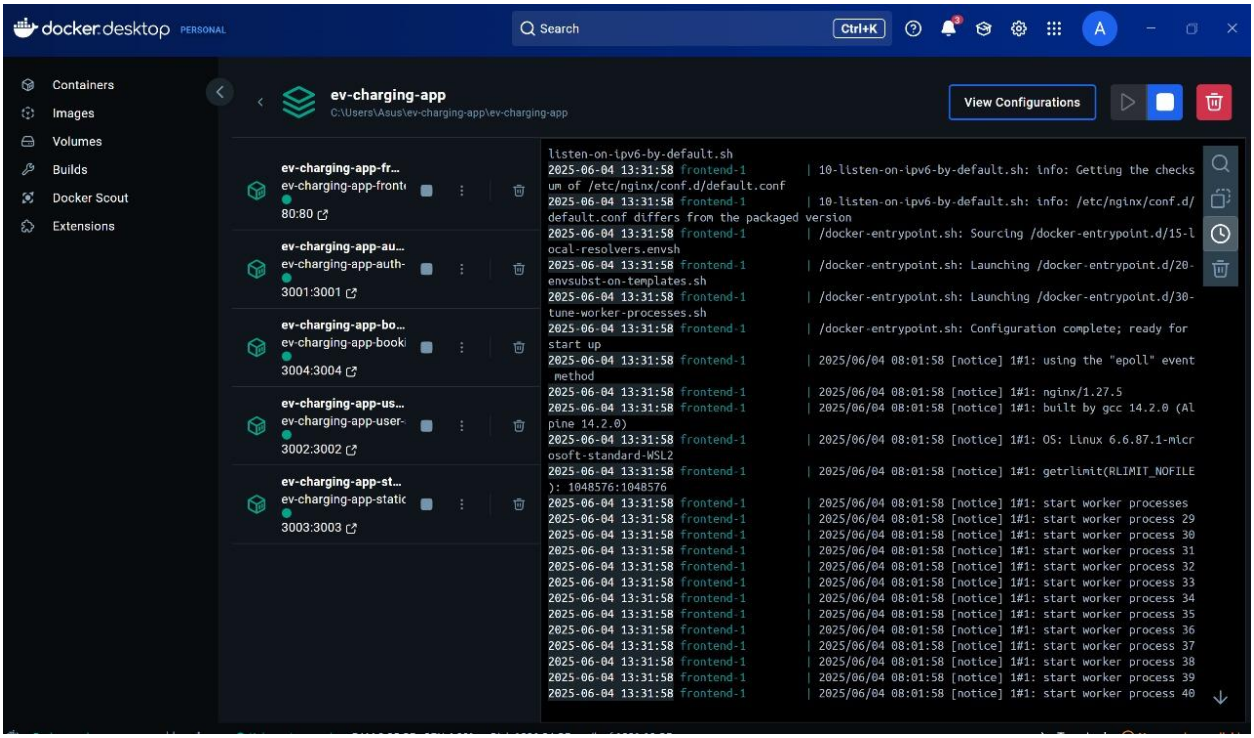
Create a Dockerfile inside each service folder (user-service/Dockerfile, etc.)

Dockerfile for React apps:

 # Dockerfile
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

FROM nginx:alpine
COPY --from=0 /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]





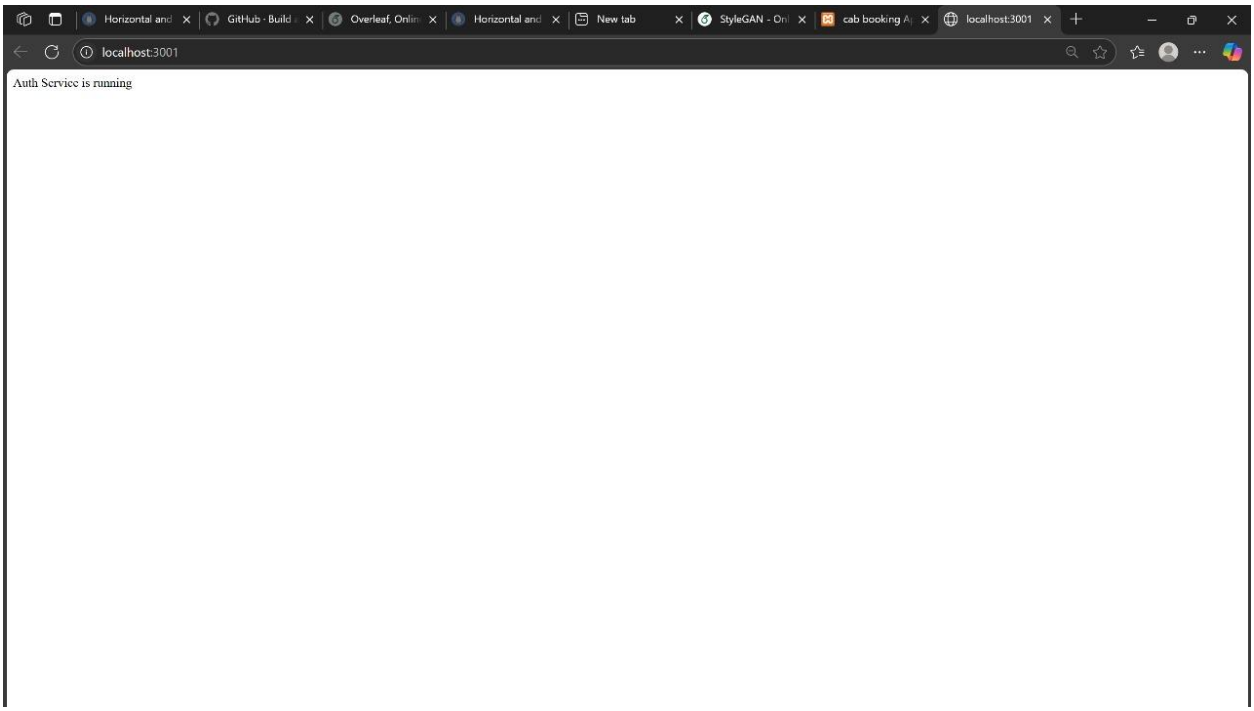
Step 6: Build and Run Docker Containers Locally

From inside each service folder:
docker build -t <service-name> .

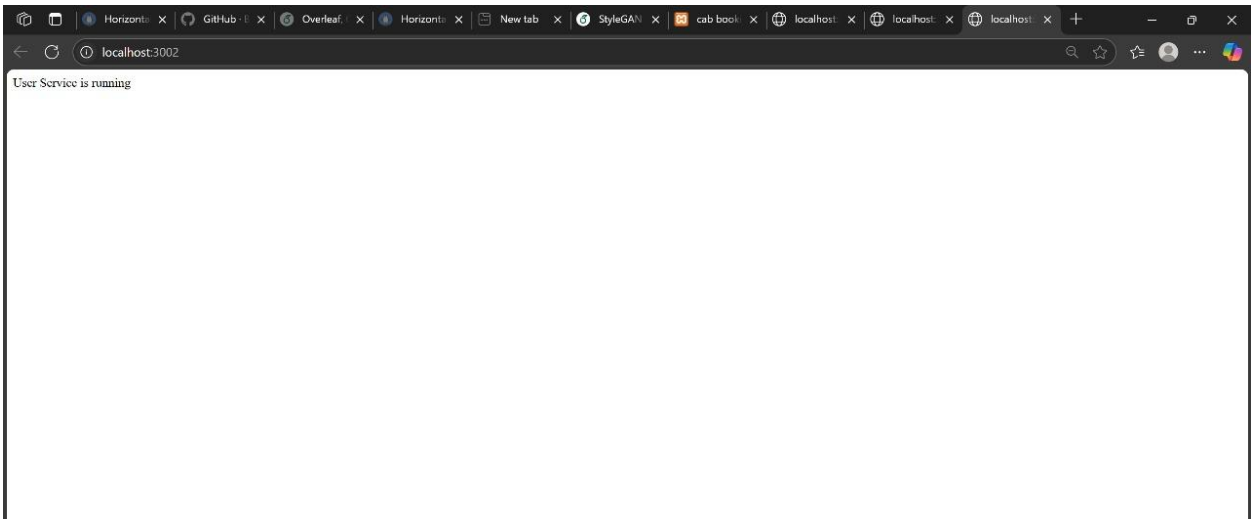
Example:
docker build -t user-service .

Run container:
docker run -d -p <port>:80 --name <service-name> user-service

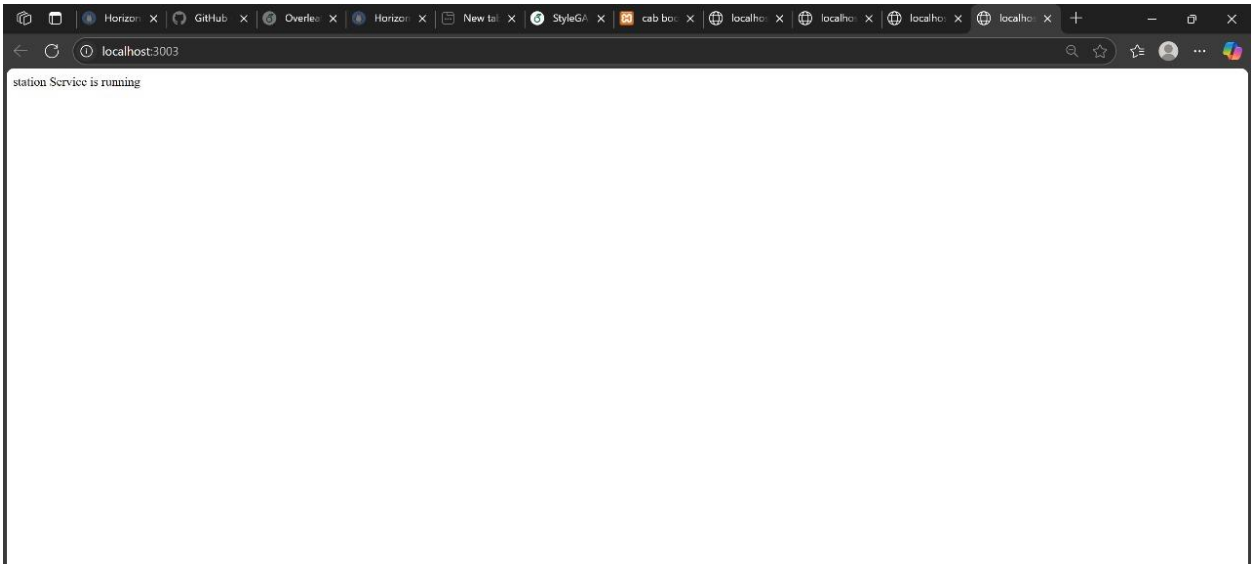
Example:
docker run -d -p 3001:80 --name user-service user-service



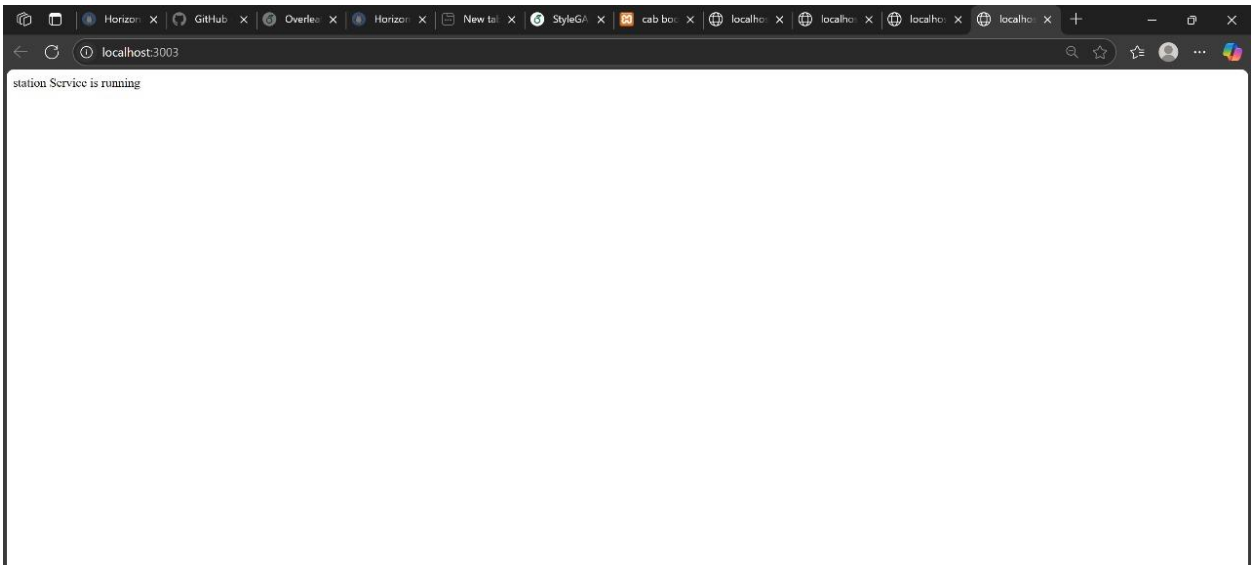
docker run -d -p 3002:80 --name cab-search-service cab-search-service



docker run -d -p 3003:80 --name booking-service booking-service



docker run -d -p 3004:80 --name payment-service payment-service



Step 7: Access Services

- User Service: http://localhost:3001
- Cab Search: http://localhost:3002
- Booking Service: http://localhost:3003
- Payment Service: http://localhost:3004

Step 8: Test Flow

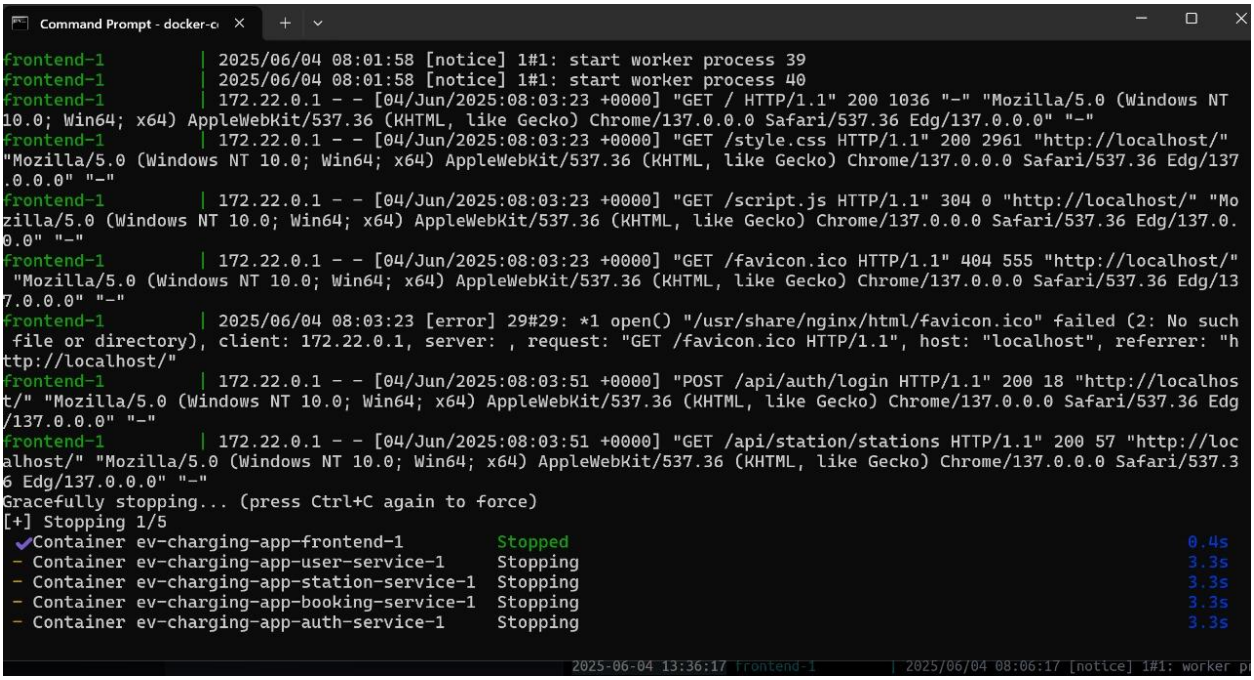
- Login at User Service (simulate login)
- Go to Cab Search (choose cab)
- Go to Booking (confirm booking)
- Go to Payment (pay)

Optional Step 9: Compose with Docker Compose

Create a docker-compose.yml in the root folder:

```
version: '3'
services:
  user-service:
    build: ./user-service
    ports:
      - "3001:80"
  cab-search-service:
    build: ./cab-search-service
    ports:
      - "3002:80"
  booking-service:
    build: ./booking-service
    ports:
      - "3003:80"
  payment-service:
    build: ./payment-service
    ports:
      - "3004:80"
```

Then run:
docker-compose up --build


Command Prompt - docker-c
frontend-1 | 2025/06/04 08:01:58 [notice] 1#1: start worker process 39
frontend-1 | 2025/06/04 08:01:58 [notice] 1#1: start worker process 40
frontend-1 | 172.22.0.1 - - [04/Jun/2025:08:03:23 +0000] "GET / HTTP/1.1" 200 1036 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/137.0.0.0 Safari/537.36 Edg/137.0.0.0" "-"
frontend-1 | 172.22.0.1 - - [04/Jun/2025:08:03:23 +0000] "GET /style.css HTTP/1.1" 200 2961 "http://localhost/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/137.0.0.0 Safari/537.36 Edg/137.0.0.0" "-"
frontend-1 | 172.22.0.1 - - [04/Jun/2025:08:03:23 +0000] "GET /script.js HTTP/1.1" 304 0 "http://localhost/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/137.0.0.0 Safari/537.36 Edg/137.0.0.0" "-"
frontend-1 | 172.22.0.1 - - [04/Jun/2025:08:03:23 +0000] "GET /favicon.ico HTTP/1.1" 404 555 "http://localhost/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/137.0.0.0 Safari/537.36 Edg/137.0.0.0" "-"
frontend-1 | 2025/06/04 08:03:23 [error] 29#29: *1 open() "/usr/share/nginx/html/favicon.ico" failed (2: No such file or directory), client: 172.22.0.1, server: , request: "GET /favicon.ico HTTP/1.1", host: "localhost", referrer: "http://localhost/"
frontend-1 | 172.22.0.1 - - [04/Jun/2025:08:03:51 +0000] "POST /api/auth/login HTTP/1.1" 200 18 "http://localhost/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/137.0.0.0 Safari/537.36 Edg/137.0.0.0" "-"
frontend-1 | 172.22.0.1 - - [04/Jun/2025:08:03:51 +0000] "GET /api/station/stations HTTP/1.1" 200 57 "http://localhost/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/137.0.0.0 Safari/537.36 Edg/137.0.0.0" "-"
Gracefully stopping... (press Ctrl+C again to force)
[+] Stopping 1/5
✔ Container ev-charging-app-frontend-1 Stopped 0.4s
- Container ev-charging-app-user-service-1 Stopping 3.3s
- Container ev-charging-app-station-service-1 Stopping 3.3s
- Container ev-charging-app-booking-service-1 Stopping 3.3s
- Container ev-charging-app-auth-service-1 Stopping 3.3s
2025-06-04 13:36:17 frontend-1 | 2025/06/04 08:06:17 [notice] 1#1: worker p

5 Apache Deployment using Ansible

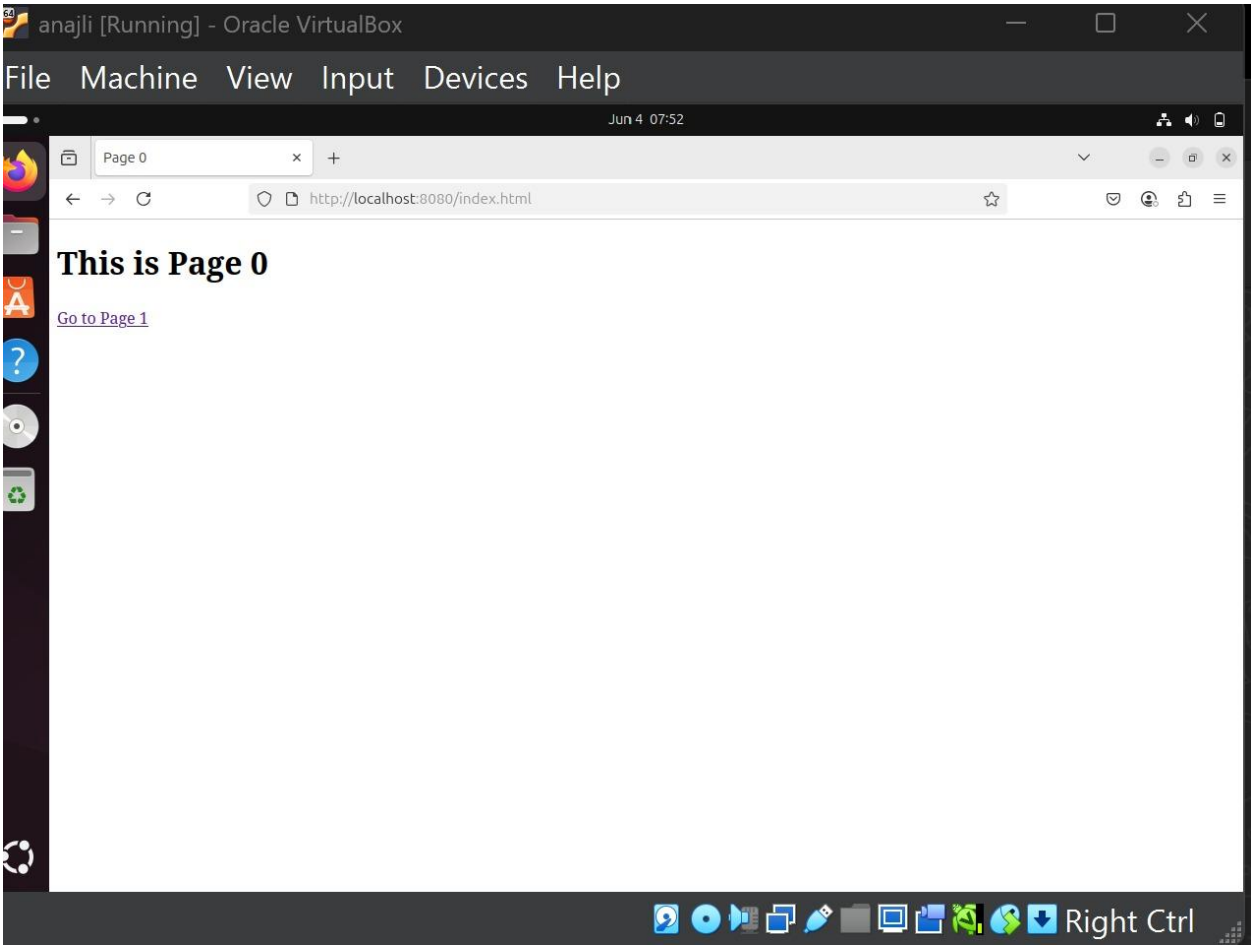
Steps

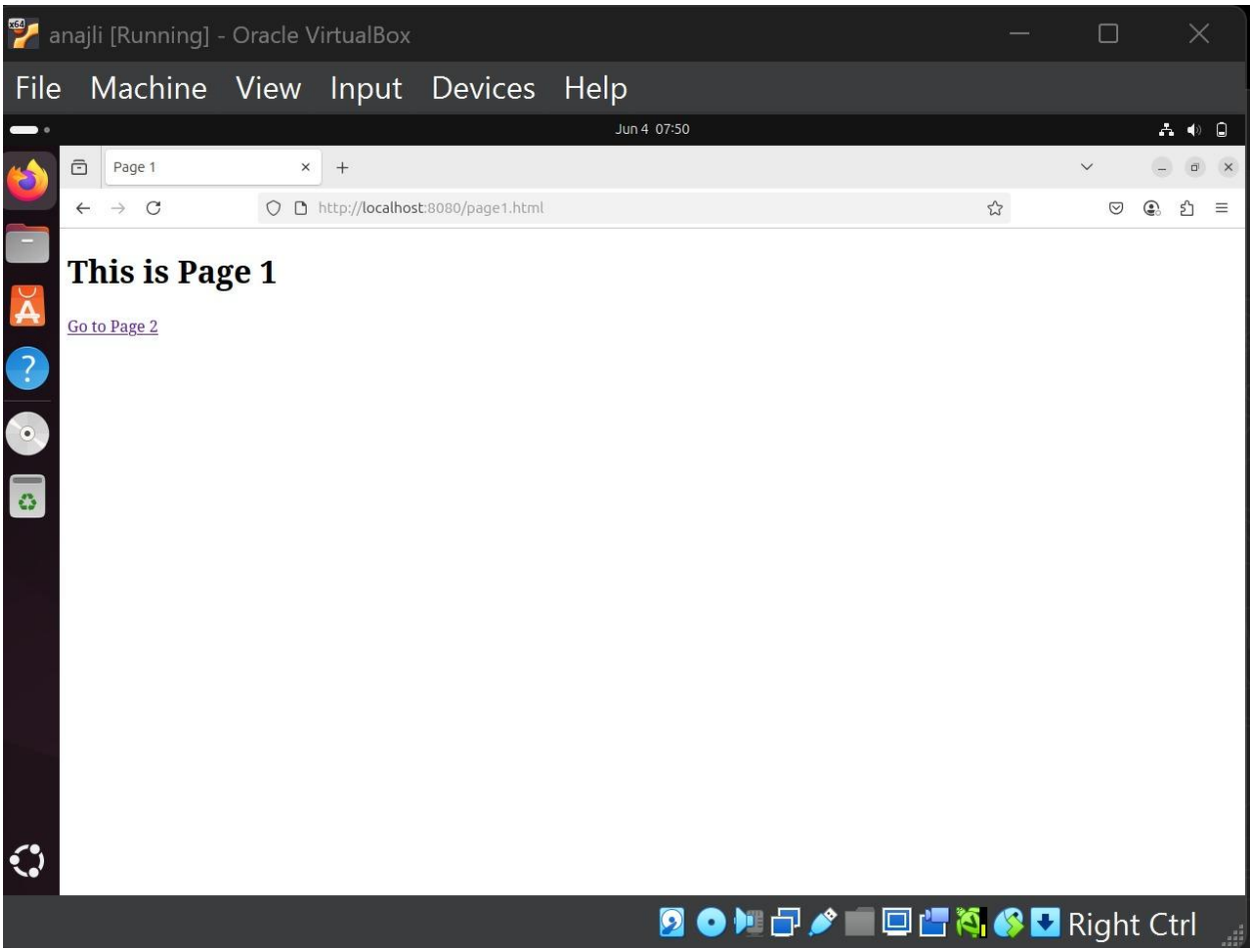
1. Setup SSH connectivity (passwordless SSH) between the control node and managed node.
2. Install Ansible on the control node.
3. Create an inventory file (/etc/ansible/hosts) with managed node IP.
4. Create a project directory: ~/ansible-apache-nginx
5. Create a playbook file deploy_apache.yml with the following content:

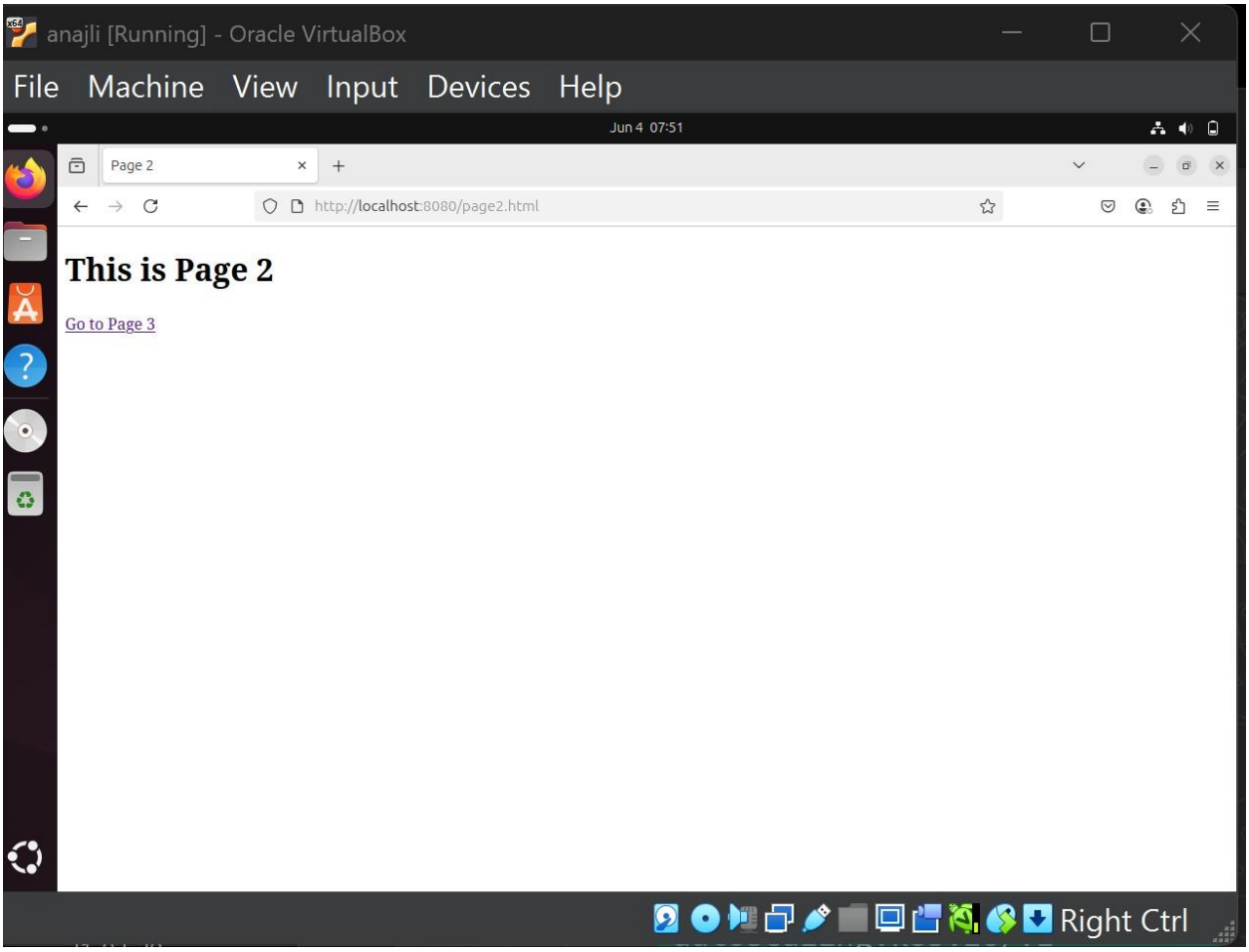
```
---
- name: Deploy Apache2 with 4 customized HTML pages
  hosts: webservers
  become: yes

  tasks:
    - name: Install Apache2
      apt:
        name: apache2
        state: present
        update_cache: yes
```

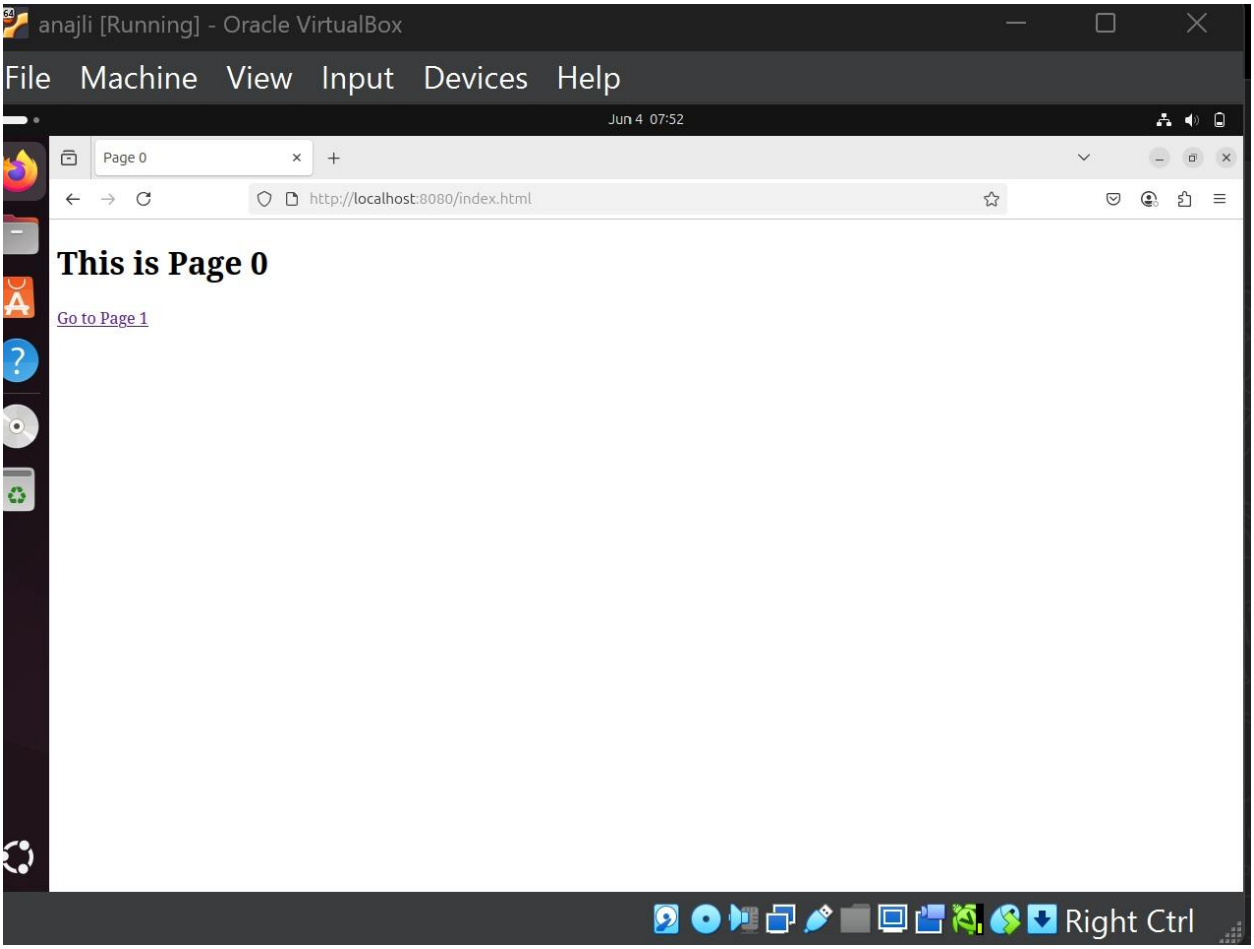
```
  - name: Ensure Apache listens on port 80
    lineinfile:
      path: /etc/apache2/ports.conf
      regexp: '^Listen '
      line: 'Listen 80'
    notify: Restart Apache

  - name: Deploy 4 customized HTML pages
    copy:
      dest: "/var/www/html/page{{ item }}.html"
      content: |
        <html>
        <head><title>Apache Page {{ item }}</title></head>
        <body><h1>Apache Page {{ item }}</h1><p>Deployed by Ansible</p></body>
        </html>
      mode: '0644'
    loop: [1, 2, 3, 4]

handlers:
  - name: Restart Apache
    service:
      name: apache2
      state: restarted
```

5. Run the playbook:
   ansible-playbook -i /etc/ansible/hosts deploy_apache.yml

6. 
anajli [Running] - Oracle VirtualBox
File Machine View Input Devices Help
Jun 4 07:52
Page 0
http://localhost:8080/index.html
This is Page 0
Go to Page 1
Right Ctrl


anajli [Running] - Oracle VirtualBox
File Machine View Input Devices Help
Jun 4 07:50
Page 1
http://localhost:8080/page1.html
This is Page 1
Go to Page 2
Right Ctrl


anajli [Running] - Oracle VirtualBox
File Machine View Input Devices Help
Jun 4 07:51
Page 2
http://localhost:8080/page2.html
This is Page 2
Go to Page 3
Right Ctrl



7. Access pages via browser: http://<managed-node-ip>/page1.html

Nginx Deployment using Ansible

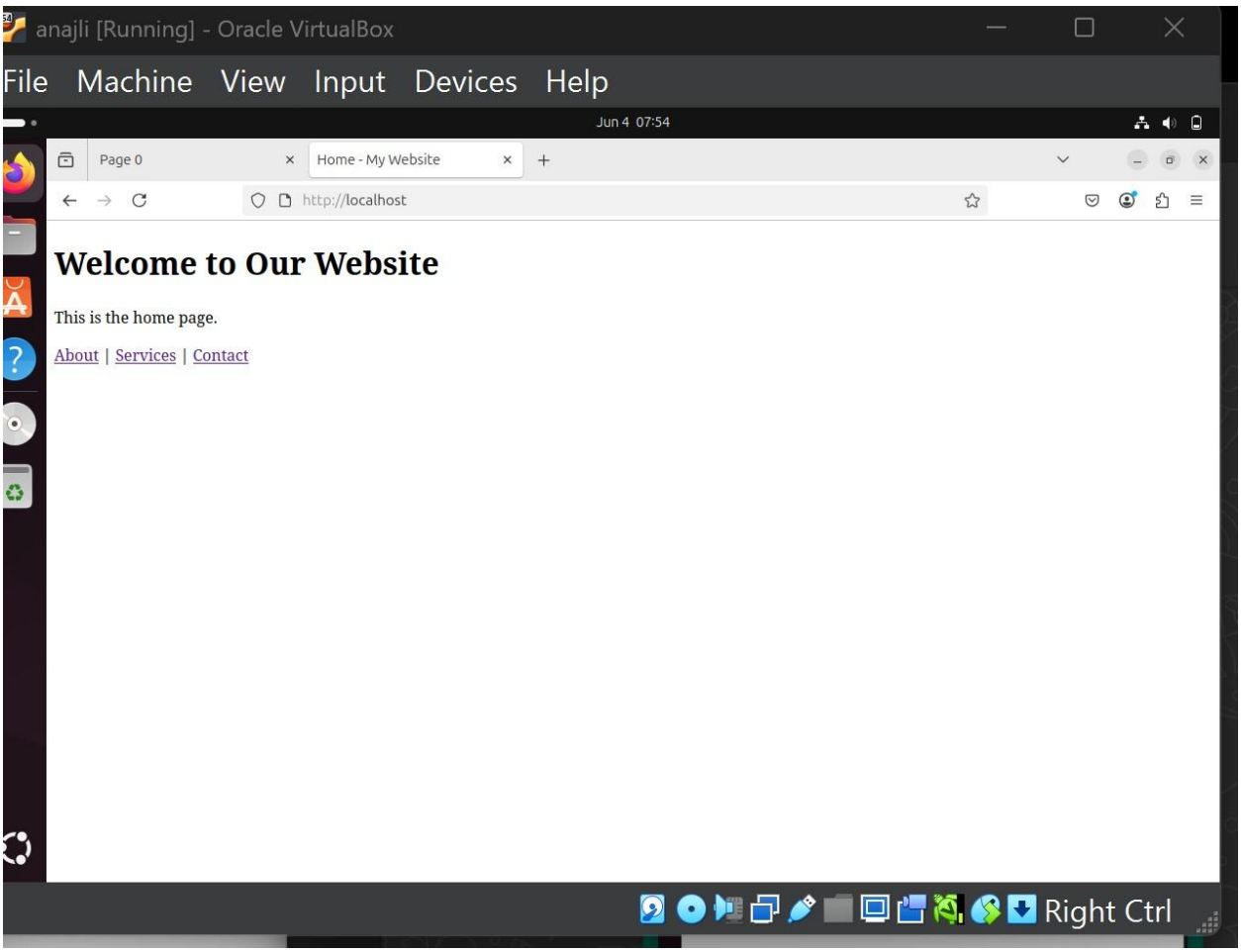
Steps

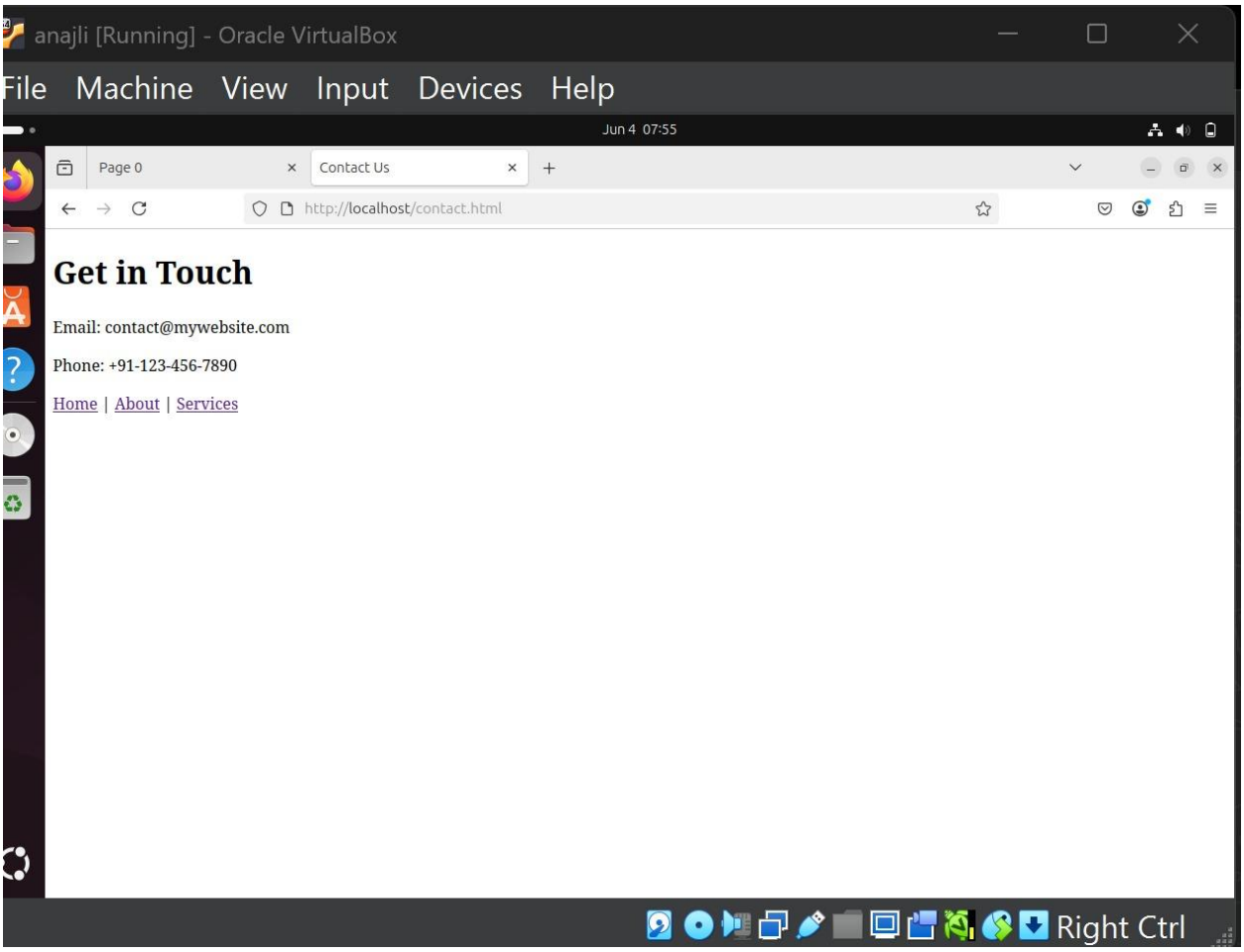
1. Setup SSH connectivity (passwordless SSH) between the control node and managed node.
2. Install Ansible on the control node.
3. Create an inventory file (/etc/ansible/hosts) with managed node IP.
4. Create a project directory: ~/ansible-apache-nginx
5. Create a playbook file deploy_nginx.yml with the following content:

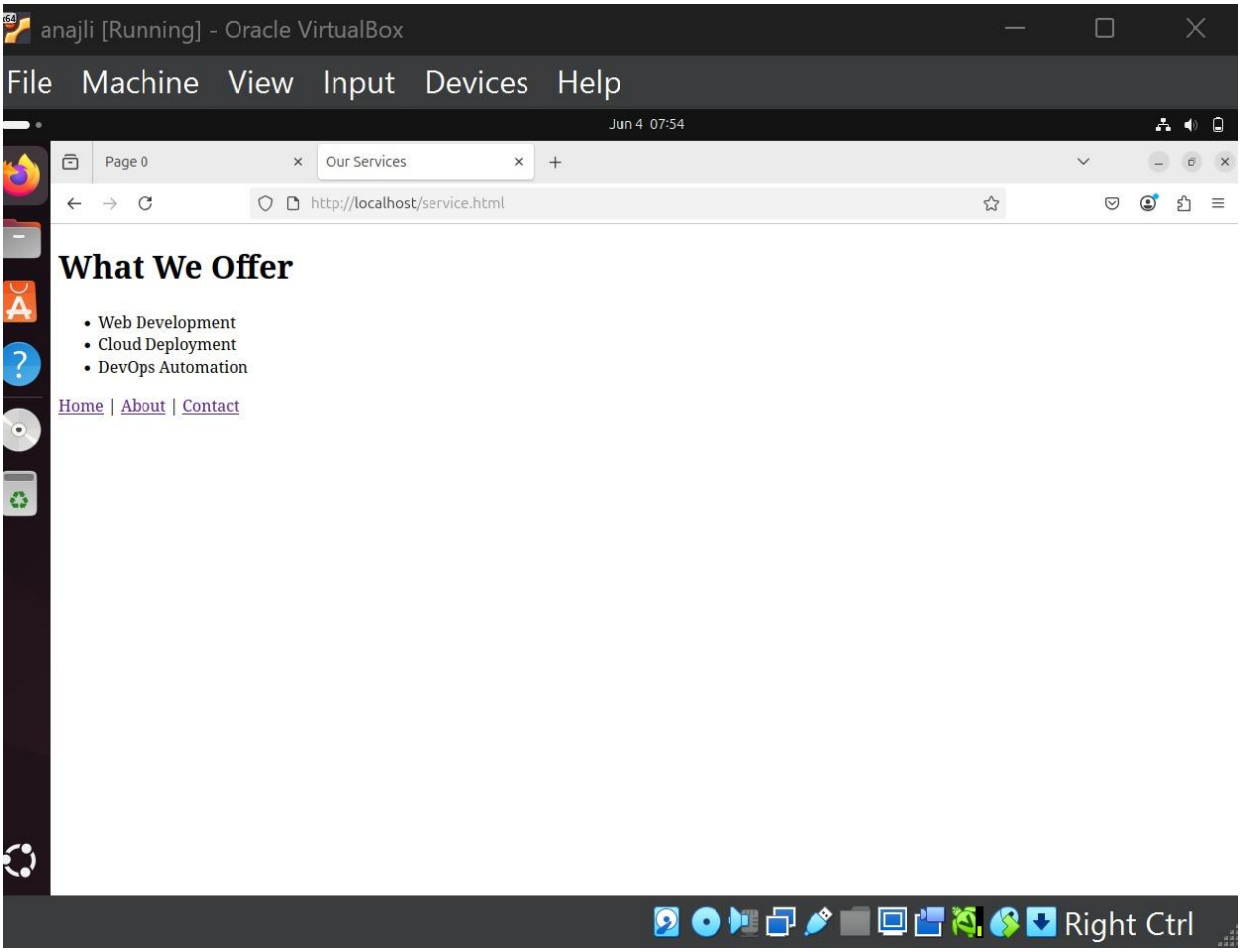
```
---
- name: Deploy Nginx with 4 customized HTML pages
  hosts: webservers
  become: yes

  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present

    - name: Configure Nginx to listen on port 100
      lineinfile:
        path: /etc/nginx/sites-available/default
        regexp: 'listen '
        line: '        listen 100 default_server;'
      notify: Restart Nginx

    - name: Deploy 4 customized HTML pages
      copy:
        dest: "/var/www/html/nginx_page{{ item }}.html"
        content: |
          <html>
          <head><title>Nginx Page {{ item }}</title></head>
          <body><h1>Nginx Page {{ item }}</h1><p>Deployed by Ansible</p></body>
          </html>
        mode: '0644'
      loop: [1, 2, 3, 4]
```

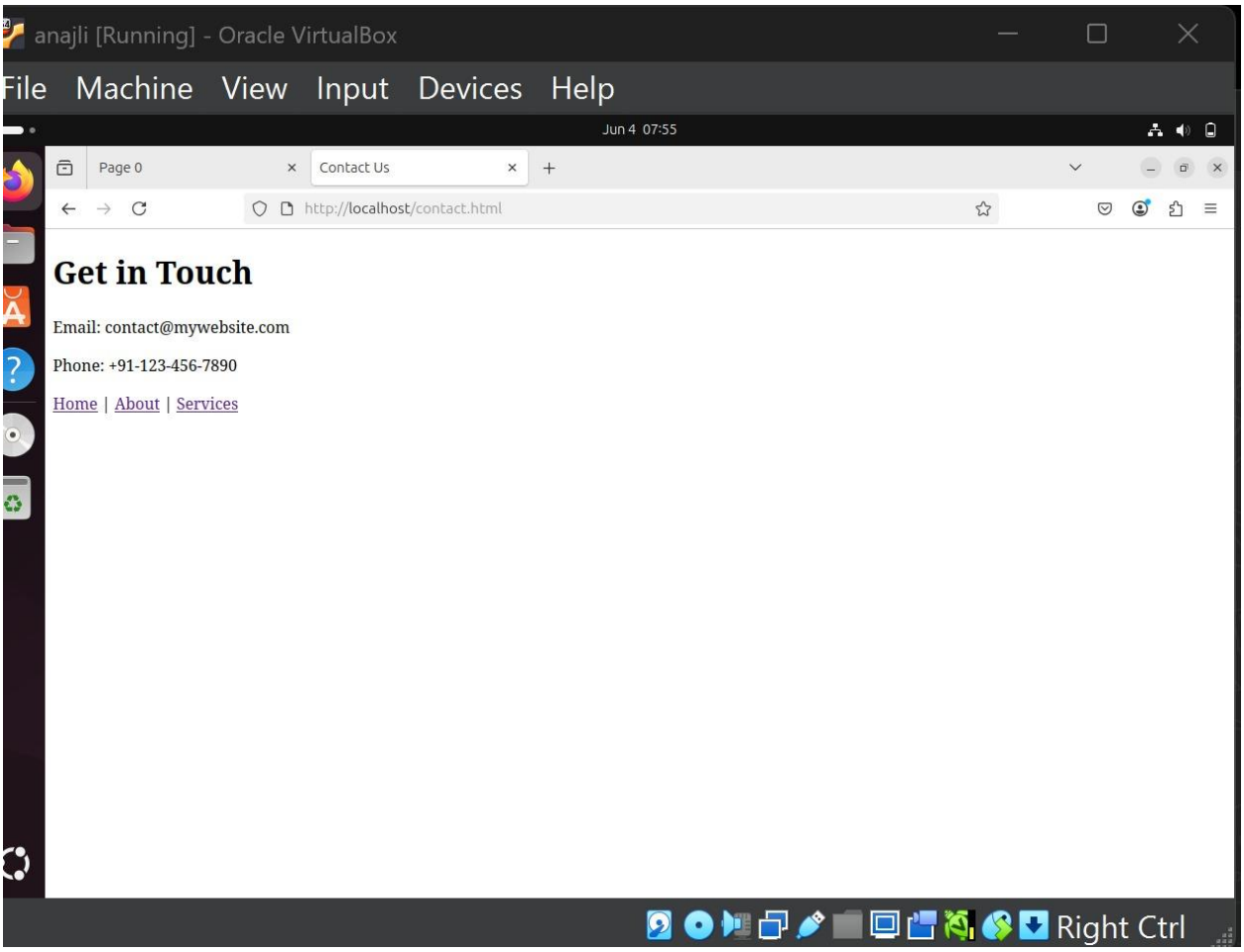
```
handlers:
  - name: Restart Nginx
    service:
      name: nginx
      state: restarted
```

7. Run the playbook:
   ansible-playbook -i /etc/ansible/hosts deploy_nginx.yml




anajli [Running] - Oracle VirtualBox
File Machine View Input Devices Help
Jun 4 07:55
Page 0
Contact Us
http://localhost/contact.html
Get in Touch
Email: contact@mywebsite.com
Phone: +91-123-456-7890
Home | About | Services
Right Ctrl


anajli [Running] - Oracle VirtualBox
File Machine View Input Devices Help
Jun 4 07:54
Page 0
Our Services
http://localhost/service.html
What We Offer
Web Development
Cloud Deployment
DevOps Automation
Home | About | Contact
Right Ctrl



7. Access pages via browser: http://<managed-node-ip>:100/nginx_page1.html