Project details are available from page 13

1. ConfigMap

ConfigMap is a Kubernetes object that lets you store configuration data in key-value pairs. It is used to manage non-sensitive configuration information separately from the application code.

Creating a ConfigMap

You can create a ConfigMap from a literal value or from a file. Here's an example of creating a ConfigMap from literal values:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: web-config
data:
  DATABASE_URL: "jdbc:mysql://db-server:3306/mydatabase"
  APP_ENV: "production"
```

Using ConfigMap in a Pod

To use the ConfigMap in a Pod, you need to reference it in your Pod specification. Here's how you can inject ConfigMap values as environment variables:

```
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: web-container
    image: my-web-app:latest
    env:
    - name: DATABASE_URL
      valueFrom:
        configMapKeyRef:
          name: web-config
          key: DATABASE_URL
    - name: APP_ENV
      valueFrom:
        configMapKeyRef:
```

```
          name: web-config
          key: APP_ENV
```

Example Use Case: Mounting ConfigMap as a File

Sometimes, an application may require configuration files. You can mount a ConfigMap as a file inside a container.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-files
data:
  config.yaml: |
    database:
      url: "jdbc:mysql://db-server:3306/mydatabase"
    environment: "production"
```

Mount the ConfigMap as a volume in the Pod:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: web-container
    image: my-web-app:latest
    volumeMounts:
    - name: config-volume
      mountPath: /etc/config
  volumes:
  - name: config-volume
    configMap:
      name: config-files
```

The configuration file `config.yaml` will be available at `/etc/config/config.yaml` inside the container.

2. Secrets

Secrets is a Kubernetes object designed to hold sensitive data such as passwords, OAuth tokens, and SSH keys. Secrets ensure that sensitive information is stored securely.

Creating a Secret

You can create a Secret from literal values or from files. Here's an example of creating a Secret from literal values:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
type: Opaque
data:
  username: dXNlcm5hbWU=   # base64 encoded 'username'
  password: cGFzc3dvcmQ=   # base64 encoded 'password'
```

Using Secrets in a Pod

To use the Secret in a Pod, reference it in your Pod specification and inject it as environment variables:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: web-container
    image: my-web-app:latest
    env:
    - name: DB_USERNAME
      valueFrom:
        secretKeyRef:
          name: db-credentials
          key: username
    - name: DB_PASSWORD
      valueFrom:
        secretKeyRef:
          name: db-credentials
          key: password
```

Example Use Case: Mounting Secrets as Files

For applications that require secrets as files, you can mount the Secret as a volume inside a container.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: ssh-keys
type: Opaque
data:
  ssh-privatekey: <base64-encoded-private-key>
  ssh-publickey: <base64-encoded-public-key>
```

Mount the Secret as a volume in the Pod:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: web-container
    image: my-web-app:latest
    volumeMounts:
    - name: ssh-volume
      mountPath: /etc/ssh
      readOnly: true
  volumes:
  - name: ssh-volume
    secret:
      secretName: ssh-keys
```

The SSH keys will be available at `/etc/ssh` inside the container.

3. Environment Variables

Environment variables are a way to pass configuration settings to applications running inside containers. They can be defined directly in the Pod specification or sourced from ConfigMaps and Secrets.

Example Use Case: Passing Configuration to a Container

Environment variables can be used to pass various configurations like application mode, API endpoints, and feature flags to the container.

Defining Environment Variables in Pod Specification

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: web-container
    image: my-web-app:latest
    env:
    - name: APP_MODE
      value: "production"
    - name: API_ENDPOINT
      value: "https://api.example.com"
```

Example Use Case: Using Environment Variables from ConfigMaps and Secrets

Combining ConfigMaps and Secrets with environment variables provides a flexible and secure way to manage configurations.

Using ConfigMap and Secret Environment Variables Together

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  containers:
  - name: web-container
    image: my-web-app:latest
    env:
    - name: DATABASE_URL
      valueFrom:
        configMapKeyRef:
          name: web-config
          key: DATABASE_URL
    - name: APP_ENV
      valueFrom:
        configMapKeyRef:
          name: web-config
          key: APP_ENV
```

```yaml
    - name: DB_USERNAME
      valueFrom:
        secretKeyRef:
          name: db-credentials
          key: username
    - name: DB_PASSWORD
      valueFrom:
        secretKeyRef:
          name: db-credentials
          key: password
```

Autoscaling in Kubernetes

1. Horizontal Pod Autoscaler (HPA)

1.1. Define a Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web-app
  template:
    metadata:
      labels:
        app: web-app
    spec:
      containers:
      - name: web-container
        image: my-web-app:latest
        ports:
        - containerPort: 80
        resources:
          requests:
            cpu: "500m"
          limits:
            cpu: "1"
```

1.2. Apply the Deployment

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

1.3. Create a Service

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

1.4. Apply the Service

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

1.5. Create an HPA

Define an HPA to scale the number of pods based on CPU utilization:

```yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: web-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: web-app
  minReplicas: 2
```

```yaml
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

1.6. Apply the HPA

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

Vertical Pod Autoscaler (VPA)

2.1. Define a Deployment

Create a Deployment for the batch job:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: batch-job
spec:
  replicas: 1
  selector:
    matchLabels:
      app: batch-job
  template:
    metadata:
      labels:
        app: batch-job
    spec:
      containers:
      - name: batch-container
        image: my-batch-job:latest
        resources:
          requests:
            cpu: "500m"
            memory: "1Gi"
```

```yaml
          limits:
            cpu: "1"
            memory: "2Gi"
```

2.2. Apply the Deployment

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

2.3. Create a VPA

Define a VPA to manage the resource requests and limits for the Pod:

```yaml
apiVersion: verticalpodautoscaler.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: batch-job-vpa
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: batch-job
  updatePolicy:
    updateMode: Auto
```

2.4. Apply the VPA

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

Linux Scripts

Viewing Processes (ps, top)

Use Cases:

1. **System Monitoring**:

    - **Example**: An administrator needs to check the status of all running processes to ensure that critical applications are running smoothly.

Commands:

```
ps aux  # Displays detailed information about all running processes
top      # Interactive view of system processes, updates in real time
```

2. **Troubleshooting Performance Issues**:
    - **Example**: A developer notices that the server is slow and needs to find out which processes are consuming the most CPU or memory.

Commands:

```
top  # Look for processes consuming high CPU or memory
ps -eo pid,comm,%cpu,%mem --sort=-%cpu | head  # Display top 10 processes by CPU usage
```

3. **Identifying Zombie Processes**:
    - **Example**: The system administrator is dealing with processes that are stuck in the “zombie” state.

Commands:

```
ps aux | grep 'Z'  # Finds processes in a zombie state
```

Examples:
Example 1:

```
ps aux | grep nginx
```

- Finds processes related to the `nginx` web server.

Example 2:

```
top -u username
```

- Displays processes owned by a specific user.

Managing Processes (kill, nice)

Use Cases:

1. **Stopping Unresponsive Applications**:
   - **Example**: A user needs to stop a process that has become unresponsive or is consuming excessive resources.

Commands:

```
kill -9 12345  # Forcefully terminates the process with PID 12345
```

2. **Adjusting Process Priority**:
   - **Example**: A system administrator wants to lower the priority of a process to ensure it does not hog resources.

Commands:

```
nice -n 10 command  # Start a process with a lower priority
renice +10 -p 12345  # Change the priority of an existing process with PID 12345
```

3. **Gracefully Stopping Services**:
   - **Example**: An admin needs to restart a service to apply configuration changes.

Commands:

```
kill -HUP 12345  # Sends a SIGHUP signal to the process to reload configuration
```

Examples:
Example 1:

```
killall -9 firefox
```

- Kills all processes named `firefox`.

Example 2:

```
nice -n -10 ./heavy_script.sh
```

- Runs `heavy_script.sh` with a higher priority.

Configure SSH

Shell Scripts

Writing Basic Shell Scripts

Use Cases:

1. **Automating Routine Tasks**:
   - **Example**: A sysadmin wants to automate the backup of log files.

Commands:

```
#!/bin/bash
cp /var/log/syslog /backup/syslog-$(date +%F).log
```

   -
2. **System Maintenance**:
   - **Example**: A developer creates a script to clean up temporary files.

Commands:

```
#!/bin/bash
rm -rf /tmp/*
```

3. **Batch Processing**:
   - **Example**: A data analyst needs to process multiple files in a directory.

Commands:

```
#!/bin/bash

process_file() {
  local file="$1"
  echo "Processing $file"
  # Add more commands to process the file here
}

for file in /data/*.csv; do
  process_file "$file"
done
```

Project 01

In this project, you will develop a simple Node.js application, deploy it on a local Kubernetes cluster using Minikube, and configure various Kubernetes features. The project includes Git version control practices, creating and managing branches, and performing rebases. Additionally, you will work with ConfigMaps, Secrets, environment variables, and set up vertical and horizontal pod autoscaling.

Project 01

Project Steps

1. Setup Minikube and Git Repository

Start Minikube:

```
minikube start
```

1.2 Set Up Git Repository
Create a new directory for your project:

```
mkdir nodejs-k8s-project
cd nodejs-k8s-project
```

Initialize Git repository:

```
git init
```

Create a `.gitignore` file:

```
node_modules/
.env
```

Add and commit initial changes:

```
git add .
git commit -m "Initial commit"
```

2. Develop a Node.js Application

2.1 Create the Node.js App
Initialize the Node.js project:

```
npm init -y
```

Install necessary packages:

```
npm install express body-parser
```

Create `app.js`:

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();
const PORT = process.env.PORT || 3000;

app.use(bodyParser.json());

app.get('/', (req, res) => {
  res.send('Hello, World!');
});

app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

Update `package.json` to include a start script:

```
"scripts": {
  "start": "node app.js"
}
```

2.2 Commit the Node.js Application
Add and commit changes:

```
git add .
git commit -m "Add Node.js application code"
```

3. Create Dockerfile and Docker Compose

3.1 Create a `Dockerfile`
Add `Dockerfile`:

```
# Use official Node.js image
```

```
FROM node:18

# Set the working directory
WORKDIR /usr/src/app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application code
COPY . .

# Expose the port on which the app runs
EXPOSE 3000

# Command to run the application
CMD [ "npm", "start" ]
```

Create a `.dockerignore` file:

```
node_modules
.npm
```

3.2 Create `docker-compose.yml` (optional for local testing)
Add `docker-compose.yml`:

```
version: '3'
services:
  app:
    build: .
    ports:
      - "3000:3000"
```

Add and commit changes:

```
git add Dockerfile docker-compose.yml
git commit -m "Add Dockerfile and Docker Compose configuration"
```

4. Build and Push Docker Image

4.1 Build Docker Image

Build the Docker image:

```
docker build -t nodejs-app:latest .
```

4.2 Push Docker Image to Docker Hub

Tag and push the image:

```
docker tag nodejs-app:latest your-dockerhub-username/nodejs-
app:latest
docker push your-dockerhub-username/nodejs-app:latest
```

Add and commit changes:

```
git add .
git commit -m "Build and push Docker image"
```

5. Create Kubernetes Configurations

5.1 Create Kubernetes Deployment

Create `kubernetes/deployment.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nodejs-app-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nodejs-app
  template:
    metadata:
      labels:
        app: nodejs-app
    spec:
```

```yaml
      containers:
      - name: nodejs-app
        image: your-dockerhub-username/nodejs-app:latest
        ports:
        - containerPort: 3000
        env:
        - name: PORT
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: PORT
        - name: NODE_ENV
          valueFrom:
            secretKeyRef:
              name: app-secrets
              key: NODE_ENV
```

5.2 Create ConfigMap and Secret

Create `kubernetes/configmap.yaml`:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  PORT: "3000"
```

Create `kubernetes/secret.yaml`:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: app-secrets
type: Opaque
data:
  NODE_ENV: cHJvZHVjdGlvbmFs # Base64 encoded value for "production"
```

Add and commit Kubernetes configurations:

```
git add kubernetes/
```

```
git commit -m "Add Kubernetes deployment, configmap, and secret"
```

5.3 Apply Kubernetes Configurations
Apply the ConfigMap and Secret:

```
kubectl apply -f kubernetes/configmap.yaml
kubectl apply -f kubernetes/secret.yaml
```

Apply the Deployment:

```
kubectl apply -f kubernetes/deployment.yaml
```

6. Implement Autoscaling

6.1 Create Horizontal Pod Autoscaler
Create `kubernetes/hpa.yaml`:

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: nodejs-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nodejs-app-deployment
  minReplicas: 2
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

Apply the HPA:

```
kubectl apply -f kubernetes/hpa.yaml
```

6.2 Create Vertical Pod Autoscaler
Create `kubernetes/vpa.yaml`:

```
apiVersion: autoscaling.k8s.io/v1beta2
kind: VerticalPodAutoscaler
metadata:
  name: nodejs-app-vpa
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nodejs-app-deployment
  updatePolicy:
    updateMode: "Auto"
```

Apply the VPA:

```
kubectl apply -f kubernetes/vpa.yaml
```

7. Test the Deployment

7.1 Check the Status of Pods, Services, and HPA
Verify the Pods:

```
kubectl get pods
```

Verify the Services:

```
kubectl get svc
```

Verify the HPA:

```
kubectl get hpa
```

7.2 Access the Application

Expose the Service:

```
kubectl expose deployment nodejs-app-deployment --type=NodePort --
name=nodejs-app-service
```

-

Get the Minikube IP and Service Port:

```
minikube service nodejs-app-service --url
```

- **Access the Application** in your browser using the URL obtained from the previous command.

8. Git Version Control

8.1 Create a New Branch for New Features

Create and switch to a new branch:

```
git checkout -b feature/new-feature
```

Make changes and commit:

```
# Make some changes
git add .
git commit -m "Add new feature"
```

Push the branch to the remote repository:

```
git push origin feature/new-feature
```

8.2 Rebase Feature Branch on Main Branch

Switch to the main branch and pull the latest changes:

```
git checkout main
git pull origin main
```

Rebase the feature branch:

```
git checkout feature/new-feature
```

```
git rebase main
```

Resolve conflicts if any, and continue the rebase:

```
git add .
git rebase --continue
```

Push the rebased feature branch:

```
git push origin feature/new-feature --force
```

9. Final Commit and Cleanup

Merge feature branch to main:

```
git checkout main
git merge feature/new-feature
```

Push the changes to the main branch:

```
git push origin main
```

Clean up:

```
git branch -d feature/new-feature
git push origin --delete feature/new-feature
```

Project 02

Deploy a Node.js application to Kubernetes with advanced usage of ConfigMaps and Secrets. Implement Horizontal Pod Autoscaler (HPA) with both scale-up and scale-down policies. The project will include a multi-environment configuration strategy, integrating a Redis cache, and monitoring application metrics.

Project Setup

1.1 Initialize a Git Repository

Create a new directory for your project and initialize Git:

```
mkdir nodejs-advanced-k8s-project
cd nodejs-advanced-k8s-project
git init
```

1.2 Create Initial Files

Create the initial Node.js application and Docker-related files:

```
npm init -y
npm install express redis body-parser
```

app.js

```
const express = require('express');
const bodyParser = require('body-parser');
const redis = require('redis');
const app = express();
const PORT = process.env.PORT || 3000;

// Connect to Redis
const redisClient = redis.createClient({
  url: `redis://${process.env.REDIS_HOST}:${process.env.REDIS_PORT}`
});
redisClient.on('error', (err) => console.error('Redis Client Error',
err));

app.use(bodyParser.json());

app.get('/', async (req, res) => {
  const visits = await redisClient.get('visits');
  if (visits) {
    await redisClient.set('visits', parseInt(visits) + 1);
  } else {
    await redisClient.set('visits', 1);
  }
  res.send(`Hello, World! You are visitor number ${visits || 1}`);
});

app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
```

```
});
```

Dockerfile

```
FROM node:18

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 3000

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

.dockerignore

```
node_modules
.npm
```

1. Build and push Docker image:

```
   docker build -t your-dockerhub-username/nodejs-advanced-
app:latest .
   docker push your-dockerhub-username/nodejs-advanced-app:latest
```

Apply Kubernetes configurations:

```
kubectl apply -f kubernetes/
```

Access the application:

```
minikube service nodejs-advanced-app-service --url
```

2. Advanced Kubernetes Configuration

2.1 Deployment Configuration

Create `kubernetes/deployment.yaml` to deploy the Node.js application with Redis dependency:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nodejs-advanced-app-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nodejs-advanced-app
  template:
    metadata:
      labels:
        app: nodejs-advanced-app
    spec:
      containers:
      - name: nodejs-advanced-app
        image: your-dockerhub-username/nodejs-advanced-app:latest
        ports:
        - containerPort: 3000
        env:
        - name: PORT
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: PORT
        - name: REDIS_HOST
          valueFrom:
            configMapKeyRef:
              name: redis-config
              key: REDIS_HOST
        - name: REDIS_PORT
          valueFrom:
            configMapKeyRef:
              name: redis-config
              key: REDIS_PORT
```

```yaml
        - name: NODE_ENV
          valueFrom:
            secretKeyRef:
              name: app-secrets
              key: NODE_ENV
      - name: redis
        image: redis:latest
        ports:
        - containerPort: 6379
```

2.2 ConfigMap for Application and Redis

Create `kubernetes/configmap.yaml` to manage application and Redis configurations:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  PORT: "3000"

---
apiVersion: v1
kind: ConfigMap
metadata:
  name: redis-config
data:
  REDIS_HOST: "redis"
  REDIS_PORT: "6379"
```

2.3 Secret for Sensitive Data

Create `kubernetes/secret.yaml` to manage sensitive environment variables:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: app-secrets
type: Opaque
data:
```

```
  NODE_ENV: cHJvZHVjdGlvbg== # Base64 encoded value for "production"
```

2.4 Service Configuration

Create `kubernetes/service.yaml` to expose the Node.js application:

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

2.5 Horizontal Pod Autoscaler with Scale-Up and Scale-Down Policies

Create `kubernetes/hpa.yaml` to manage autoscaling:

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nodejs-advanced-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nodejs-advanced-app-deployment
  minReplicas: 2
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
```

```yaml
        averageUtilization: 50
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 70
  behavior:
    scaleUp:
      stabilizationWindowSeconds: 30
      selectPolicy: Max
      policies:
      - type: Pods
        value: 2
        periodSeconds: 30
      - type: Resource
        resource: cpu
        value: 2
        periodSeconds: 30
    scaleDown:
      stabilizationWindowSeconds: 30
      selectPolicy: Min
      policies:
      - type: Pods
        value: 1
        periodSeconds: 30
      - type: Resource
        resource: memory
        value: 1
        periodSeconds: 30
```

2.6 Vertical Pod Autoscaler Configuration

Create `kubernetes/vpa.yaml` to manage vertical scaling:

```yaml
apiVersion: autoscaling.k8s.io/v1beta2
kind: VerticalPodAutoscaler
metadata:
  name: nodejs-advanced-app-vpa
spec:
  targetRef:
```

```yaml
    apiVersion: apps/v1
    kind: Deployment
    name: nodejs-advanced-app-deployment
  updatePolicy:
    updateMode: "Auto"
```

2.7 Redis Deployment

Add a Redis deployment configuration to `kubernetes/redis-deployment.yaml`:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis:latest
        ports:
        - containerPort: 6379
```

Add Redis service configuration to `kubernetes/redis-service.yaml`:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: redis-service
spec:
  selector:
    app: redis
  ports:
```

```
  - protocol: TCP
    port: 6379
    targetPort: 6379
  type: ClusterIP
```

2.8 Apply Kubernetes Configurations

Apply all configurations to your Minikube cluster:

```
kubectl apply -f kubernetes/redis-deployment.yaml
kubectl apply -f kubernetes/redis-service.yaml
kubectl apply -f kubernetes/configmap.yaml
kubectl apply -f kubernetes/secret.yaml
kubectl apply -f kubernetes/deployment.yaml
kubectl apply -f kubernetes/service.yaml
kubectl apply -f kubernetes/hpa.yaml
kubectl apply -f kubernetes/vpa.yaml
```

2.9 Verify Deployments and Services

Check the status of your deployments and services:

```
kubectl get all
```

Access the application via Minikube:

```
minikube service nodejs-advanced-app-service --url
```

2.10 Testing Scaling

Simulate load on the application to test the HPA:

```
kubectl run -i --tty --rm load-generator --image=busybox --
restart=Never -- /bin/sh
# Inside the pod, run the following command to generate load
while true; do wget -q -O- http://nodejs-advanced-app-service; done
```

2.11 Validate Autoscaling Behavior

Observe the HPA behavior:

```
kubectl get hpa
```

Watch the scaling events and verify that the application scales up and down based on the policies you configured.

3. Project Wrap-Up

3.1 Review and Clean Up

After completing the project, review the configurations and clean up the Minikube environment if needed:

```
minikube delete
```