DAILY PROJECT ACTIVITY IS FROM PAGE 21 IN THIS DOC

1. Kubernetes Basics and Core Concepts

- **What is Kubernetes?**
  - Overview of Kubernetes
  - Use Cases and Benefits
- **Kubernetes Architecture Recap**
  - Master Node Components
  - Worker Node Components
- **Kubernetes API Server**
  - Function and Role
  - Interaction with Kubectl
- **Kubernetes Objects and Resources**
  - Pods
  - ReplicaSets
  - Deployments
  - Namespaces
  - Services

2. Kubernetes CLI (kubectl) Commands

- **Basic kubectl Commands**
  - `kubectl get`, `kubectl describe`, `kubectl logs`
- **Managing Resources**
  - Create, Update, Delete Resources
- **Namespace Management**
  - Creating and Switching Namespaces

3. Configuring and Managing Pods

- **Pod Lifecycle**
  - Creation, Running, Termination
- **Pod Configurations**
  - YAML Files, Labels, Annotations
- **Pod Networking**
  - Communication Between Pods
  - Network Policies

4. Deployments and ReplicaSets

- **Creating Deployments**
  - Basic Deployment Configuration
- **Scaling Deployments**
  - Manual and Automatic Scaling
- **Managing ReplicaSets**
  - Rolling Updates and Rollbacks

5. Services and Networking

- **Service Types**
  - ClusterIP, NodePort, LoadBalancer, ExternalName

    - GitOps with ArgoCD, Flux

13. Kubernetes Networking

- **Network Models and Concepts**
    - Overlay Networks, CNI Plugins
- **Ingress Controllers**
    - Setting Up and Managing Ingress Rules

14. Advanced Kubernetes Topics

- **Kubernetes Federation**
    - Overview and Use Cases
- **Custom Controllers and Operators**
    - Creating and Managing Operators
- **Kubernetes API Extensions**
    - CRDs and Aggregated APIs

15. Kubernetes Troubleshooting and Maintenance

- **Common Issues and Troubleshooting**
    - Logs Analysis, Debugging Pods
- **Cluster Maintenance**
    - Upgrades, Backup and Restore Strategies

16. Scaling Kubernetes and Cloud Providers

- **Horizontal and Vertical Scaling**
    - Auto-scaling Mechanisms
- **Integration with Cloud Providers**
    - AWS, GCP, Azure Specific Integrations

17. Kubernetes in Production

- **Best Practices for Production Deployments**
    - High Availability, Disaster Recovery
- **Cost Management and Optimization**

Basic Structure of a Kubernetes YAML Configuration File

```
apiVersion: <api-version>
kind: <resource-kind>
metadata:
  name: <resource-name>
  namespace: <namespace>  # Optional
  labels:                  # Optional
    <label-key>: <label-value>
spec:
  <spec-field>:            # Specification fields specific to the
resource type
```

```
    <subfield>: <value>
    ...
```

1. Pods

What is a Pod?

- **Definition**: A Pod is the smallest and simplest Kubernetes object. It represents a single instance of a running process in the cluster.
- **Components**:
  - **Container**: Runs the application or service.
  - **Storage Volumes**: Shared storage accessible by containers in the Pod.
  - **Networking**: A unique IP address for communication between Pods.

Example

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app-pod
spec:
  containers:
  - name: my-app-container
    image: nginx:latest
    ports:
    - containerPort: 80
```

Use Cases

- **Basic Application Deployment**:
  - **Example**: Running a simple web server or API service.
  - **Use Case**: Deploying a static website using Nginx.
- **Debugging and Development**:
  - **Example**: Running a single container for debugging or development.
  - **Use Case**: Creating a Pod to test a container image before deploying it with more complex configurations.

2. ReplicaSets

What is a ReplicaSet?

- **Definition**: A ReplicaSet ensures that a specified number of Pod replicas are running at any given time.
- **Purpose**: Maintains a stable set of replica Pods running at any given time.

Example

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: my-app-replicaset
spec:
  selector:
    matchLabels:
      app: my-app
  replicas: 3
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app-container
        image: nginx:latest
        ports:
        - containerPort: 80
```

Use Cases

- **Application Availability**:
  - **Example**: Ensuring multiple instances of a web application are always running.
  - **Use Case**: Scaling a web server to handle increased traffic.
- **Self-Healing Mechanism**:
  - **Example**: Automatically replacing failed Pods.
  - **Use Case**: Ensuring that if a Pod crashes, another one is created to replace it.

3. Deployments

What is a Deployment?

- **Definition**: A Deployment manages ReplicaSets and provides declarative updates to applications.
- **Purpose**: Automates the process of scaling applications, rolling updates, and rollbacks.

Example

```
apiVersion: apps/v1
```

```yaml
kind: Deployment
metadata:
  name: my-app-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app-container
        image: nginx:latest
        ports:
        - containerPort: 80
```

Use Cases

- **Application Upgrades**:
  - **Example**: Updating a web server’s image from version 1.0 to 2.0.
  - **Use Case**: Rolling out new features or bug fixes without downtime.
- **Rolling Back Changes**:
  - **Example**: Reverting to a previous version of an application.
  - **Use Case**: Rolling back if a new deployment introduces issues.

4. Namespaces

What is a Namespace?

- **Definition**: A Namespace is a way to divide cluster resources between multiple users or projects.
- **Purpose**: Helps in resource isolation and organization.

Example

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: development
```

Use Cases

- **Multi-Tenancy**:
  - **Example**: Creating separate environments for development, testing, and production.
  - **Use Case**: Different teams can work on the same cluster without interfering with each other's resources.
- **Resource Management**:
  - **Example**: Quotas and limits on resources for different teams.
  - **Use Case**: Setting resource quotas for each team or environment to manage usage.

5. Services

What is a Service?

- **Definition**: A Service is an abstraction that defines a logical set of Pods and a policy by which to access them.
- **Purpose**: Ensures reliable networking and load balancing between Pods.

Example

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

Use Cases

- **Internal Communication**:
  - **Example**: Exposing a web application to other Pods within the same cluster.
  - **Use Case**: Services allow Pods to communicate with each other and with external systems.
- **Load Balancing**:
  - **Example**: Distributing traffic among multiple Pods.
  - **Use Case**: Creating a Service to balance the load for a web application among several instances.

Service Type

- **ClusterIP**: Default type, only accessible within the cluster.
- **NodePort**: Exposes the Service on each Node's IP at a static port.

- **LoadBalancer (Cloud Specific)**: Provisioned by cloud providers to expose the Service externally.
- **ExternalName**: Maps the Service to an external DNS name.

ClusterIP: Default type, only accessible within the cluster

Use Case: **Internal Microservice Communication**

- **Scenario**: You have a set of microservices deployed within your Kubernetes cluster that need to communicate with each other but do not need to be accessible from outside the cluster.
- **Example**: You have a backend service called `backend-service` that needs to communicate with a database service called `db-service`.

```
apiVersion: v1
kind: Service
metadata:
  name: db-service
spec:
  type: ClusterIP
  selector:
    app: database
  ports:
  - protocol: TCP
    port: 5432
    targetPort: 5432
```

NodePort: Exposes the Service on each Node’s IP at a static port

Use Case: **Direct Access for Testing or Development**

- **Scenario**: You are in a development or testing environment and need to access a service directly on a specific port on any node in the cluster.
- **Example**: You want to expose a simple web application running in your cluster on port `30007` so it can be accessed from outside the cluster.

```
apiVersion: v1
kind: Service
metadata:
  name: web-service
spec:
  type: NodePort
  selector:
    app: web
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
    nodePort: 30007
```

LoadBalancer: Provisioned by cloud providers to expose the Service externally

Use Case: **Public Access to a Production Service**

- **Scenario**: You have a production application that needs to be accessible from the internet. You are using a cloud provider that supports LoadBalancer services.
- **Example**: You want to expose your production web application to the internet using a cloud provider's load balancer.

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

ExternalName: Maps the Service to an external DNS name

Use Case: **External Dependency Integration**

- **Scenario**: You have an application running in your cluster that needs to connect to an external service (e.g., a third-party API or an external database) via a DNS name.
- **Example**: You want your application to connect to an external MySQL database hosted outside the cluster at `mysql.external.com`.

```
apiVersion: v1
kind: Service
metadata:
  name: external-mysql
spec:
  type: ExternalName
  externalName: mysql.external.com
```

Deployment Strategies Types

1. Rolling Update Deployment

Definition:

Rolling updates allow updating the application to a new version gradually, replacing Pods one at a time with minimal downtime.

Characteristics:

- **Incremental:** Updates Pods incrementally to the new version.
- **Minimal Downtime:** Ensures some Pods are always available during the update.
- **Rollback Capability:** Can roll back to the previous version if something goes wrong.

Example:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rolling-update-deployment
  namespace: my-namespace
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-container
        image: nginx:1.14.2
        ports:
        - containerPort: 80
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
```

Use Case:

When you want to update an application with minimal disruption.

2. Recreate Deployment

Definition:

Recreate deployments delete all existing Pods before creating new ones. This strategy ensures that no old Pods are running alongside the new Pods.

Characteristics:

- **Downtime:** Causes a temporary downtime because old Pods are terminated before new ones are created.
- **Simplified Process:** Useful for applications where downtime is acceptable.

Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: recreate-deployment
  namespace: my-namespace
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-container
        image: nginx:1.14.2
        ports:
        - containerPort: 80
  strategy:
    type: Recreate
```

Use Case:

Suitable for applications that cannot handle running multiple versions simultaneously or require a complete restart.

3. Blue-Green Deployment

Definition:

Blue-Green deployment is a strategy where two environments, blue and green, are maintained. One environment serves production traffic while the other is used to deploy the new version.

Characteristics:

- **Zero Downtime:** Traffic is switched from the old version to the new version without downtime.
- **Rollback Capability:** Easy rollback to the previous version by switching traffic back.

Example:

This can be achieved using Kubernetes services and manual or automated scripts to switch traffic between environments.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: blue-green-deployment
  namespace: my-namespace
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
      version: blue  # Initial version
  template:
    metadata:
      labels:
        app: my-app
        version: blue
    spec:
      containers:
      - name: my-container
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Use Case:

Ideal for critical applications where zero downtime is essential and rollback needs to be quick and reliable.

4. Canary Deployment

Definition:

Canary deployment gradually rolls out the new version to a small subset of users before rolling it out to the entire infrastructure.

Characteristics:

- **Incremental Testing:** Allows testing the new version with a small user base.
- **Controlled Rollout:** Gradually increases the traffic to the new version.
- **Quick Rollback:** Easier to roll back if issues are found.

Example:

Create a second deployment for the canary release with fewer replicas:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: canary-deployment
  namespace: my-namespace
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
      version: canary
  template:
    metadata:
      labels:
        app: my-app
        version: canary
    spec:
      containers:
      - name: my-container
        image: nginx:1.15.0
        ports:
        - containerPort: 80
```

Adjust the Service to split traffic between stable and canary versions.

Hardlink-Softlink

Hardlink:

- **Explanation:** A hardlink is a direct pointer to the data on the disk. Multiple hardlinks can point to the same data, sharing the same inode. If the original file is deleted, the data remains accessible via the hardlink(s).
- **Example:**
  - **Backup Systems:** In a backup system, hardlinks can be used to create incremental backups without duplicating data. Each backup can have hardlinks to unchanged files, saving space.

Command:

```
ln original_file hardlink_to_file
```

  ○

Softlink (Symlink):

- **Explanation:** A softlink is a shortcut or a reference to another file or directory. It has its own inode and points to the original file by its path. If the original file is deleted, the softlink becomes a broken link.
- **Example:**
  - **Shared Configurations:** Symlinks can be used to share configuration files among different applications or directories. If the config file is updated, all symlinked references will see the updated version.

Command:

```
ln -s original_file symlink_to_file
```

Hardlink Example

Scenario: Efficient Disk Space Usage for Backup Files

Step-by-Step:

1. **Create an Original File:**

Create a text file with some content.

```
echo "This is the original file." > original_file.txt
```

2. **Create a Hardlink:**

Create a hardlink to the original file.

```
ln original_file.txt hardlink_to_file.txt
```

3. **Verify Inodes:**

Check the inodes of both files to confirm they are the same.

```
ls -i original_file.txt hardlink_to_file.txt
```

The output should show the same inode number for both files, indicating they point to the same data on the disk.

4. **Modify Content:**

Modify the content of the original file and observe the change in the hardlink.

```
echo "Adding more content to the original file." >>
original_file.txt
```

```
cat hardlink_to_file.txt
```

The content change will be reflected in the hardlink file as well.

5. **Delete Original File:**

Delete the original file and check the hardlink.

```
rm original_file.txt
```

```
cat hardlink_to_file.txt
```

The hardlink will still contain the original content since it points to the same data.

Softlink (Symlink) Example

1. **Create an Original File:**

Create a configuration file with some content.
bash
Copy code
```
echo "config_value=1" > config_file.txt
```

2. **Create a Symlink:**

Create a symlink to the original configuration file in another directory.

```
ln -s /path/to/config_file.txt
/path/to/other_directory/symlink_to_config.txt
```

3. **Verify Symlink:**

List the files to verify the symlink creation.

```
ls -l /path/to/other_directory/symlink_to_config.txt
```

The output should indicate that `symlink_to_config.txt` points to `/path/to/config_file.txt`.

4. **Modify Content:**

Modify the content of the original file and observe the change in the symlink.

```
echo "config_value=2" >> /path/to/config_file.txt
```

```
cat /path/to/other_directory/symlink_to_config.txt
```

The content change will be reflected in the symlinked file as well.

5. **Delete Original File:**

Delete the original file and check the symlink.

```
rm /path/to/config_file.txt
```

```
cat /path/to/other_directory/symlink_to_config.txt
```

- The symlink will be broken, and attempting to read the symlink will result in an error since the original file no longer exists.

Understanding of States like Running/Sleep/Stopped/Zombie

Use Case: System Monitoring and Process Management

- **Running:**
  - **Explanation:** The process is actively executing instructions on the CPU.
  - **Example:** A web server handling requests is in a running state.

Command:

```
top
```

- **Sleeping:**
  - **Explanation:** The process is not currently executing but is waiting for an event or resource.
  - **Example:** A database server waiting for new queries enters a sleeping state.

Command:

```
ps aux | grep 'S'
```

- **Stopped:**
  - **Explanation:** The process has been halted, typically by receiving a signal (SIGSTOP) from a user or another process.
  - **Example:** A user temporarily stops a long-running script to troubleshoot an issue.

Command:

```
kill -STOP <pid>
```

- **Zombie:**
  - **Explanation:** The process has completed execution but still has an entry in the process table, typically because the parent process has not yet read its exit status.

  - **Example:** A child process ends, but the parent process hasn't acknowledged the termination.

Command:

```
ps aux | grep 'Z'
```

  -

systemctl States

Use Case: Service Management in Linux Systems

- **Active (Running):**
  - **Explanation:** The service is running properly.
  - **Example:** The web server service (e.g., Apache) is serving web pages.

Command:

```
systemctl status apache2
```

  -
- **Inactive (Stopped):**
  - **Explanation:** The service is not running.
  - **Example:** A service like `ssh` is stopped for security reasons.

Command:

```
systemctl stop sshd
```

  -
- **Enabled:**
  - **Explanation:** The service is configured to start at boot.
  - **Example:** Ensuring the database server starts automatically after a reboot.

Command:

```
systemctl enable mysqld
```

  -
- **Disabled:**
  - **Explanation:** The service is not configured to start at boot.
  - **Example:** Disabling a test service from starting at boot.

Command:
```
systemctl disable nginx
```

Tar and Compression

Use Case: Data Backup and Archival

- **Tar:**
  - **Explanation:** `tar` is used to create archive files from multiple files and directories.

    - **Example:** Archiving a project directory before deploying changes.

Command:

```
tar -cvf project.tar /path/to/project
```

- **Compression (gzip, bzip2):**
    - **Explanation:** Compresses the tar archive to save space.
    - **Example:** Compressing the project archive to transfer it over the network.

Command:

```
tar -czvf project.tar.gz /path/to/project

tar -cjvf project.tar.bz2 /path/to/project
```

FTP, SFTP, Rsync

Use Case: File Transfer and Synchronization

- **FTP:**
    - **Explanation:** FTP is a protocol for transferring files between systems.
    - **Example:** Uploading website files to a remote server.

Command:
```
ftp <hostname>
```

- **SFTP:**
    - **Explanation:** SFTP is a secure version of FTP, using SSH for encryption.
    - **Example:** Securely transferring sensitive data to a remote server.

Command:

```
sftp <user>@<hostname>
```

- **Rsync:**
    - **Explanation:** Rsync efficiently syncs files and directories between two locations.
    - **Example:** Backing up a local directory to a remote server.

Command:

```
rsync -avz /local/dir user@remote:/remote/dir
```

    ○

Basic Networking & Communication

Use Case: Network Configuration and Troubleshooting

- **IP Address Configuration:**
    - **Explanation:** Assigning an IP address to a network interface.
    - **Example:** Setting a static IP for a server.

Command:

```
ip addr add 192.168.1.10/24 dev eth0
```

- **Checking Network Connectivity:**
  - **Explanation:** Verifying if a host is reachable over the network.
  - **Example:** Troubleshooting connectivity issues to a remote server.

Command:

```
ping google.com
```

- **Displaying Network Configuration:**
  - **Explanation:** Viewing the current network configuration and interfaces.
  - **Example:** Checking the IP address and network interface status.

Command:

```
ifconfig

ip addr show
```

- **Network Route Management:**
  - **Explanation:** Displaying and modifying the IP routing table.
  - **Example:** Adding a static route to a specific network.

Command:
```
route -n
```

  - `ip route add 192.168.2.0/24 via 192.168.1.1`

Project 01

Deploying a Node.js App Using Minikube Kubernetes

Overview

This project guides you through deploying a Node.js application using Minikube Kubernetes. You'll use Git for version control, explore branching and fast-forward merges, and set up Kubernetes services and deployment pods, including ClusterIP and NodePort service types.

Prerequisites

- Minikube installed
- kubectl installed
- Git installed
- Node.js installed (https://nodejs.org/en/download/package-manager/all#debian-and-ubuntu-based-linux-distributions)

Project Steps

1. Set Up Git Version Control

1.1. Initialize a Git Repository

Create a new directory for your project:

```
mkdir nodejs-k8s-project

cd nodejs-k8s-project
```

Initialize a Git repository:

```
git init
```

1.2. Create a Node.js Application

Initialize a Node.js project:

```
npm init -y
```

Install Express.js:

```
npm install express
```

Create an `index.js` file with the following content:

```
const express = require('express');

const app = express();

const port = 3000;


app.get('/', (req, res) => {

    res.send('Hello, Kubernetes!');

});


app.listen(port, () => {

    console.log(`App running at http://localhost:${port}`);
});
```

1.

Create a `.gitignore` file to ignore `node_modules`:

```
node_modules
```

1.3. Commit the Initial Code

Add files to Git:

```
git add .
```

Commit the changes:

```
git commit -m "Initial commit with Node.js app"
```

2. Branching and Fast-Forward Merge

2.1. Create a New Branch

Create and switch to a new branch `feature/add-route`:

```
git checkout -b feature/add-route
```

2.2. Implement a New Route

Modify `index.js` to add a new route:

```
app.get('/newroute', (req, res) => {
    res.send('This is a new route!');
});
```

Commit the changes:

```
git add .
git commit -m "Add new route"
```

2.3. Merge the Branch Using Fast-Forward

Switch back to the `main` branch:

```
git checkout main
```

Merge the `feature/add-route` branch using fast-forward:

```
git merge --ff-only feature/add-route
```

Delete the feature branch:

```
git branch -d feature/add-route
```

3. Containerize the Node.js Application

3.1. Create a Dockerfile

Create a `Dockerfile` with the following content:

```
FROM node:14

WORKDIR /app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD ["node", "index.js"]
```

3.2. Build and Test the Docker Image

Build the Docker image:

```
docker build -t nodejs-k8s-app .
```

Run the Docker container to test:

```
docker run -p 3000:3000 nodejs-k8s-app
```

1. Access `http://localhost:3000` to see the app running.

4. Deploying to Minikube Kubernetes

4.1. Start Minikube

Start Minikube:

```
minikube start
```

4.2. Create Kubernetes Deployment and Service Manifests

Create a `deployment.yaml` file:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nodejs-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nodejs-app
  template:
    metadata:
      labels:
        app: nodejs-app
    spec:
      containers:
      - name: nodejs-app
        image: nodejs-k8s-app:latest
        ports:
        - containerPort: 3000
```

Create a `service.yaml` file for ClusterIP:

```
apiVersion: v1
```

```yaml
kind: Service

metadata:

  name: nodejs-service

spec:

  selector:

    app: nodejs-app

  ports:

  - protocol: TCP

    port: 80

    targetPort: 3000

  type: ClusterIP
```

Create a `service-nodeport.yaml` file for NodePort:

```yaml
apiVersion: v1

kind: Service

metadata:

  name: nodejs-service-nodeport

spec:

  selector:

    app: nodejs-app

  ports:

  - protocol: TCP

    port: 80

    targetPort: 3000

    nodePort: 30001

  type: NodePort
```

4.3. Apply Manifests to Minikube

Apply the deployment:

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

Apply the ClusterIP service:

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

Apply the NodePort service:

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

4.4. Access the Application

Get the Minikube IP:

```
minikube ip
```

1. Access the application using the NodePort:

   ```
   curl http://<minikube-ip>:30001
   ```

Making Changes to the App and Redeploying Using Kubernetes

6. Making Changes to the Node.js Application

6.1. Create a New Branch for Changes

Create and switch to a new branch `feature/update-message`:

```
git checkout -b feature/update-message
```

6.2. Update the Application

Modify `index.js` to change the message:

```
const express = require('express');

const app = express();

const port = 3000;


app.get('/', (req, res) => {

    res.send('Hello, Kubernetes! Updated version.');

});
```

```
app.get('/newroute', (req, res) => {
    res.send('This is a new route!');
});

app.listen(port, () => {
    console.log(`App running at http://localhost:${port}`);
});
```

6.3. Commit the Changes

Add and commit the changes:

```
git add .
git commit -m "Update main route message"
```

7. Merge the Changes and Rebuild the Docker Image

7.1. Merge the Feature Branch

Switch back to the `main` branch:

```
git checkout main
```

Merge the `feature/update-message` branch:

```
git merge --ff-only feature/update-message
```

Delete the feature branch:

```
git branch -d feature/update-message
```

7.2. Rebuild the Docker Image

Rebuild the Docker image with a new tag:

```
docker build -t nodejs-k8s-app:v2 .
```

8. Update Kubernetes Deployment

8.1. Update the Deployment Manifest

Modify `deployment.yaml` to use the new image version:

```
apiVersion: apps/v1

kind: Deployment

metadata:

  name: nodejs-app

spec:

  replicas: 2

  selector:

    matchLabels:

      app: nodejs-app

  template:

    metadata:

      labels:

        app: nodejs-app

    spec:

      containers:

      - name: nodejs-app

        image: nodejs-k8s-app:v2

        ports:

        - containerPort: 3000
```

8.2. Apply the Updated Manifest

Apply the updated deployment:

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

8.3. Verify the Update

Check the status of the deployment:

```
kubectl rollout status deployment/nodejs-app
```

9. Access the Updated Application

9.1. Access Through ClusterIP Service

Forward the port to access the ClusterIP service:

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

1. Open your browser and navigate to `http://localhost:8080` to see the updated message.

9.2. Access Through NodePort Service

1. Access the application using the NodePort:

   ```
   curl http://<minikube-ip>:30001
   ```

Project 02

Deploying a Python Flask App Using Minikube Kubernetes

Overview

This project guides you through deploying a Python Flask application using Minikube Kubernetes. You'll use Git for version control, explore branching and fast-forward merges, and set up Kubernetes services and deployment pods, including ClusterIP and NodePort service types.

Prerequisites

- Minikube installed
- kubectl installed
- Git installed
- Python installed

Project Steps

1. Set Up Git Version Control

1.1. Initialize a Git Repository

Create a new directory for your project:

```
mkdir flask-k8s-project

cd flask-k8s-project
```

Initialize a Git repository:
sh
Copy code
```
git init
```

1.2. Create a Python Flask Application

Create a virtual environment:

```
python -m venv venv

source venv/bin/activate
```

Install Flask:
sh
Copy code
```
pip install Flask
```

Create an `app.py` file with the following content:
python
Copy code
```
from flask import Flask



app = Flask(__name__)



@app.route('/')

def hello_world():

    return 'Hello, Kubernetes!'



if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Create a `requirements.txt` file to list the dependencies:
Copy code

```
Flask
```

Create a `.gitignore` file to ignore `venv`:
Copy code

```
venv
```

1.3. Commit the Initial Code

Add files to Git:

```
git add .
```

Commit the changes:

```
git commit -m "Initial commit with Flask app"
```

2. Branching and Fast-Forward Merge

2.1. Create a New Branch

Create and switch to a new branch `feature/add-route`:

```
git checkout -b feature/add-route
```

2.2. Implement a New Route

Modify `app.py` to add a new route:

```
@app.route('/newroute')

def new_route():

    return 'This is a new route!'
```

Commit the changes:

```
git add .

git commit -m "Add new route"
```

2.3. Merge the Branch Using Fast-Forward

Switch back to the `main` branch:

```
git checkout main
```

Merge the `feature/add-route` branch using fast-forward:

```
git merge --ff-only feature/add-route
```

Delete the feature branch:

```
git branch -d feature/add-route
```

3. Containerize the Flask Application

3.1. Create a Dockerfile

Create a `Dockerfile` with the following content:

```
FROM python:3.8-slim


WORKDIR /app


COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt


COPY . .


EXPOSE 5000


CMD ["python", "app.py"]
```

3.2. Build and Test the Docker Image

Build the Docker image:

```
docker build -t flask-k8s-app .
```

Run the Docker container to test:

```
docker run -p 5000:5000 flask-k8s-app
```

1.
2. Access `http://localhost:5000` to see the app running.

4. Deploying to Minikube Kubernetes

4.1. Start Minikube

Start Minikube:

```
minikube start
```

4.2. Create Kubernetes Deployment and Service Manifests

Create a `deployment.yaml` file:

```
apiVersion: apps/v1

kind: Deployment

metadata:

  name: flask-app

spec:

  replicas: 2

  selector:

    matchLabels:

      app: flask-app

  template:

    metadata:

      labels:

        app: flask-app
    spec:
      containers:
      - name: flask-app
        image: flask-k8s-app:latest
        ports:
        - containerPort: 5000
```

Create a `service.yaml` file for ClusterIP:

```
apiVersion: v1
kind: Service
metadata:
  name: flask-service
spec:
  selector:
    app: flask-app
  ports:
  - protocol: TCP
    port: 80
    targetPort: 5000
  type: ClusterIP
```

Create a `service-nodeport.yaml` file for NodePort:

```
apiVersion: v1
kind: Service
metadata:
  name: flask-service-nodeport
```

```
spec:
  selector:
    app: flask-app
  ports:
  - protocol: TCP
    port: 80
    targetPort: 5000
    nodePort: 30001
  type: NodePort
```

4.3. Apply Manifests to Minikube

Apply the deployment:

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

Apply the ClusterIP service:

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

Apply the NodePort service:

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

4.4. Access the Application

Get the Minikube IP:

```
minikube ip
```

Access the application using the NodePort:

```
curl http://<minikube-ip>:30001
```

5. Clean Up

Stop Minikube:

```
minikube stop
```

Delete Minikube cluster:

```
minikube delete
```

6. Making Changes to the Flask Application

6.1. Create a New Branch for Changes

Create and switch to a new branch `feature/update-message`:

```
git checkout -b feature/update-message
```

6.2. Update the Application

Modify `app.py` to change the message:

```
@app.route('/')
def hello_world():
    return 'Hello, Kubernetes! Updated version.'


@app.route('/newroute')
def new_route():
    return 'This is a new route!'
```

6.3. Commit the Changes

Add and commit the changes:

```
git add .
git commit -m "Update main route message"
```

7. Merge the Changes and Rebuild the Docker Image

7.1. Merge the Feature Branch

Switch back to the `main` branch:

```
git checkout main
```

1.

Merge the `feature/update-message` branch:

```
git merge --ff-only feature/update-message
```

Delete the feature branch:

```
git branch -d feature/update-message
```

7.2. Rebuild the Docker Image

Rebuild the Docker image with a new tag:

```
docker build -t flask-k8s-app:v2 .
```

8. Update Kubernetes Deployment

8.1. Update the Deployment Manifest

Modify `deployment.yaml` to use the new image version:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: flask-app
  template:
    metadata:
      labels:
        app: flask-app
    spec:
      containers:
      - name: flask-app
        image: flask-k8s-app:v2
        ports:
        - containerPort: 5000
```

8.2. Apply the Updated Manifest

Apply the updated deployment:
sh
Copy code
```
kubectl apply -f deployment.yaml
```

8.3. Verify the Update

Check the status of the deployment:
sh
Copy code
```
kubectl rollout status deployment/flask-app
```

9. Access the Updated Application

9.1. Access Through ClusterIP Service

Forward the port to access the ClusterIP service:

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

1. Open your browser and navigate to `http://localhost:8080` to see the updated message.

9.2. Access Through NodePort Service

1. Access the application using the NodePort:

   ```
   curl http://<minikube-ip>:30001
   ```