Run a nodejs / python flask application on Kubernetes Cluster:

P.S: For testing purpose we can install Minikube or KinD Kubernetes Cluster on Linux (Ubuntu) Server

Here are the steps to create a real-time “nodejsapp” project and deploy it in Kubernetes:

1. Create a Node.js Application:

We will develop application using Node.js according to our preference and requirements. We will ensure it has the necessary functionalities you want to deploy. Here is simple nodejsaap as an example

Simple node.js App code

```
// index.js

const express = require('express');
const app = express();
const port = 3000;

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

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

Clarification of Steps:

1. **Develop Your Application**:
    - The provided Node.js code is a simple web application using Express.js framework.
    - It listens for incoming HTTP requests on port 3000 and responds with "Hello, World!" when the root URL is accessed.
2. **Organize Your Project Structure**:
    - In a typical Node.js project, we will organize files and directories as follows:

```
/your_project_name
├── node_modules/    (automatically created when installing dependencies)
├── index.js        (entry point of your application)
├── package.json     (defines project metadata and dependencies)
└── package-lock.json (locks the version of dependencies)
```

1. We can install dependencies using `npm install` command, and they will be listed in `package.json`.
    - Any additional assets such as HTML, CSS, or static files can be placed in appropriate directories within our project structure.

This Node.js application serves as a basic starting point. We can expand it with additional routes, middleware, or functionality as per our project requirements. Similarly, if we choose to develop a Python Flask application, we will follow a similar process of defining routes and organizing project files.

2. Build Docker Image and Push to AWS ECR/Docker Hub:

Write a Dockerfile:

- Create a file named `Dockerfile` in the root directory of your Node.js project.
- Define the Docker image for your application by specifying the base image, installing dependencies, copying source code, and exposing the necessary port.
- Here's an example of a `Dockerfile` for your Node.js application:

Dockerfile for node.js Application

```
# Use the official Node.js image as the base image

FROM node:14

# Set the working directory inside the container

WORKDIR /usr/src/app

# Copy package.json and package-lock.json to the working directory

COPY package*.json ./

# Install dependencies

RUN npm install

# Copy the rest of the application code

COPY . .

# Expose port 3000 to the outside world

EXPOSE 3000

# Command to run the application

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

Build the Docker image locally using the docker build command.

Open a terminal or command prompt.

- Navigate to the root directory of your Node.js project where the `Dockerfile` is located.
- Run the following command to build the Docker image:

```
docker build -t nodejsapp .
```

Tag the built image with the appropriate repository URL for AWS ECR or Docker Hub using docker tag.

For AWS ECR:

```
docker tag nodejsapp:<tag> <aws_account_id>.dkr.ecr.<region>.amazonaws.com/<repository_name>:<tag>
```

For Hub.docker:

```
docker tag nodejsapp:<tag> <docker_username>/<repository_name>:<tag>
```

Push the Docker image to your chosen registry using the docker push command.

docker push <registry_url>/<repository_name>:<tag>

Replace `<registry_url>` with the appropriate URL for AWS ECR (`<aws_account_id>.dkr.ecr.<region>.amazonaws.com`) or Docker Hub (`docker.io`).

Replace `<repository_name>` with the name of your repository on AWS ECR or Docker Hub.

Replace `<tag>` with the tag you assigned to your Docker image during tagging.

3. Deploy the Application to Kubernetes:

Below are the detailed steps for deploying your Node.js application to Kubernetes:

Write Kubernetes YAML Manifests:

Create YAML files for Deployment, Service, Secret, and ConfigMap resources. You can use separate files for each resource or combine them into a single file.
Define the specifications for each resource, including metadata, spec, and any other necessary fields.
Here's an example of a Deployment YAML file (`deployment.yaml`):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nodejsapp
  labels:
    app: nodejs
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nodejs
  template:
    metadata:
      labels:
        app: nodejs
    spec:
      containers:
      - name: nodejsapp
        image: <your_image_name>:<tag>
        ports:
        - containerPort: 3000
```

Replace `<your_image_name>:<tag>` with the name and tag of your Docker image.

1. **Define a Service Resource**:
    - Create a Service YAML file (`service.yaml`) to expose your application internally or externally.
    - Specify the type of service (ClusterIP, NodePort, LoadBalancer, or ExternalName) and the port mappings.
    - Here's an example of a Service YAML file:

```
apiVersion: v1
kind: Service
```

```
metadata:
  name: nodejs-service
spec:
  selector:
    app: nodejs
  ports:
  - protocol: TCP
    port: 80
    targetPort: 3000
  type: NodePort
```

2. **Manage Secrets and ConfigMaps**:
    - Create separate YAML files for Secrets and ConfigMaps if you need to manage sensitive data or configuration settings separately.
    - Define the data or configuration settings within the YAML files.
    - Here's an example of a Secret YAML file (`secret.yaml`):

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
data:
  username: <base64_encoded_username>
  password: <base64_encoded_password>
```

Replace `<base64_encoded_username>` and `<base64_encoded_password>` with the base64 encoded values of your sensitive data.

3. **Apply Kubernetes Manifests**:
    - Open a terminal or command prompt.
    - Navigate to the directory containing your Kubernetes YAML files.
    - Run the following command to apply the manifests to your Kubernetes cluster:

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

    - This command will create or update the specified resources in your Kubernetes cluster based on the YAML files provided.

By following these steps, you'll be able to deploy your Node.js application to Kubernetes and manage its resources using Deployment, Service, Secret, and ConfigMap resources.

4. In Kubernetes, Use of Ingress if required

- To expose your Node.js application externally using an Ingress resource in Kubernetes, follow these steps:

2. **Write Ingress Manifest**:
    - Create a YAML file named `ingress.yaml` to define the Ingress resource.
    - Specify the rules to route incoming requests to the appropriate Service based on the host or path.

- Here's an example of an Ingress YAML file:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nodejsapp-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: your.domain.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: nodejs-service
            port:
              number: 80
```

3. **Apply the Ingress Manifest**:
    - Open a terminal or command prompt.
    - Navigate to the directory containing your Ingress YAML file (`ingress.yaml`).
    - Run the following command to apply the Ingress manifest to your Kubernetes cluster:

```
kubectl apply -f ingress.yaml
```

By following these steps, you'll expose your Node.js application externally using an Ingress resource in Kubernetes. Make sure to replace `your.domain.com` with your actual domain name, and `nodejs-service` with the name of your Service resource defined earlier. Additionally, ensure that your Kubernetes cluster has an Ingress controller deployed and configured to handle incoming traffic.

5. Use Helm Chart to Manage Kubernetes YAML Files:

Below are the steps to use Helm to manage Kubernetes YAML files for your Node.js application:

1. **Organize Kubernetes YAML Files into a Helm Chart Structure**:
    - Create a new directory named after your Helm chart (e.g., `nodejs-app-chart`) in your project.
    - Organize your Kubernetes YAML files (Deployment, Service, Secret, ConfigMap, Ingress) within the appropriate directories (`templates`, `charts`, etc.) according to Helm's chart structure.
2. **Write a values.yaml File**:
    - Create a file named `values.yaml` in the root directory of your Helm chart.

- Define customizable parameters for your application, such as image repository, tag, service type, environment variables, etc.
- Here's an example of a `values.yaml` file:

```
# values.yaml
image:
  repository: your-docker-repo/nodejs-app
  tag: latest
service:
  type: NodePort
```

3. **Create Helm Templates**:
   - Write Helm templates for Deployment, Service, Secret, ConfigMap, and Ingress resources, utilizing the values defined in `values.yaml`.
   - Use Go template syntax to parameterize your YAML files and inject values from `values.yaml`.
   - Here's an example of a Helm template for Deployment (`templates/deployment.yaml`):

```
# deployment.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "nodejs-app-chart.fullname" . }}
  labels:
    app: {{ include "nodejs-app-chart.name" . }}
spec:
  replicas: 3
  selector:
    matchLabels:
      app: {{ include "nodejs-app-chart.name" . }}
  template:
    metadata:
      labels:
        app: {{ include "nodejs-app-chart.name" . }}
    spec:
      containers:
        - name: {{ .Chart.Name }}
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          ports:
            - containerPort: 3000
```

4. **Package Your Helm Chart**:
   - Open a terminal or command prompt.
   - Navigate to the root directory of your Helm chart.
   - Run the following command to package your Helm chart:

```
helm package .
```

5. **Deploy Your Helm Chart to Kubernetes**:
   - Use `helm install` to deploy your Helm chart to your Kubernetes cluster.

- You can override default values defined in `values.yaml` by providing a `--set` flag with the desired values.
- Here's an example deployment command:

  ```
  helm install my-nodejs-app ./nodejs-app-chart --set image.tag=v1
  ```

By following these steps, you'll be able to organize, customize, package, and deploy your Node.js application to Kubernetes using Helm, simplifying the management of Kubernetes resources and configuration.

Written By:
Engr. Muhammad Ramzan

==========================The end=================================