How to deploy webapp on kubernetes

Deploying a web application on Kubernetes(K8s) involves a series of steps that ensure your application is containerized, orchestrated, and managed efficiently. Below the key points for create and deploy a web application on k8s.

Create container image

K8s is a and orchestrator that works managing container image. So as first step a your application need to be containerized, Docker is the most common tool for this, but other tools can be used, like Podman or other open source alternatives. Below is a simple python application and an example Dockrfile that permit to create the container image:

Make sure you have `app.py` and `requirements.txt` in the same directory as your Dockerfile. Here's a simple `app.py` example for a Flask application:

Python Example

```python
import os
from flask import Flask
app = Flask(__name__)

# Use os.environ.get() to read the environment variable 'NAME'.
# Provide a default value in case 'NAME' is not set.
name = os.environ.get('NAME', 'World')

@app.route('/')
def hello():
    # Use the 'name' variable in your application's response
    return f"Hello {name}!"

if __name__ == '__main__':
    # Run the app on all available interfaces on port 80
    app.run(host='0.0.0.0', port=80)
```

And your `requirements.txt` should at least contain Flask, like so:

```
Flask
```

this below is the docker file used for create a container version of the test application

Dockerfile Example

```dockerfile
# Use an official Python runtime as a parent image
FROM python:3.9-slim

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

# Copy the current directory contents into the container at /usr/src/app
COPY . .

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

Here's a brief explanation of each command in the Dockerfile:

- `FROM python:3.9-slim`: This line sets the base image for the Docker image, using a slim version of the official Python 3.9 image.
- `WORKDIR /usr/src/app`: This line sets the working directory in the Docker image. Any relative file path will be set from this location.
- `COPY . .`: This command copies all files in the current directory on the host machine into the working directory in the Docker image.
- `RUN pip install --no-cache-dir -r requirements.txt`: This command installs the Python dependencies listed in the `requirements.txt` file.
- `EXPOSE 80`: This line informs Docker that the application listens on port 80. You might need to adjust this depending on the port your Flask app is set to listen on.
- `ENV NAME World`: This line sets an example environment variable that could be used by the application.
- `CMD ["python", "app.py"]`: This is the command that runs when the container starts up. It starts your Flask application.

You can then build and run your Docker image with:

```
docker build -t your-application-name:tag .
docker run -p 4000:80 your-application-name:tag
```

this is the link to the test application: app-test.zip

The application now need to pushed to a docker registry (Docker, Github, etc...)

```
docker tag your-application-name:tag registry-url/your-application-name:tag
docker push registry-url/your-application-name:tag
```

Set Up Kubernetes Environment

Usually one of the most used k8s test environment can be built using Minikube, or other open source alternatives.

Create k8s object

To deploy the created container on k8s we need to create the following k8s resources:

- Deployment: resource that Defines how your application runs and scales. It references the Docker image and defines the desired state.
- Service: exposes your application to the internet or internal users. Types include ClusterIP, NodePort, and LoadBalancer.

Deployment Resource (deploy.yaml)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: your-application-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: your-application
  template:
    metadata:
      labels:
        app: your-application
    spec:
      containers:
      - name: your-application
        image: registry-url/your-application-name:tag
        ports:
        - containerPort: 80
```

This Kubernetes resource is a Deployment configuration, which provides declarative updates for Pods and ReplicaSets. Let's break down its components:

- **`apiVersion: apps/v1`**: Specifies the API version for the Deployment resource. `apps/v1` indicates that it is a stable version of the deployment API, suitable for production use.
- **`kind: Deployment`**: Defines the kind of Kubernetes resource being configured. In this case, it's a Deployment, which manages the creation, scaling, and updating of Pods based on the specified template.
- **`metadata`**:
  - **`name: your-application-deployment`**: The name of the Deployment. This name is used to identify the Deployment within the Kubernetes cluster.
- **`spec`** (specification):
  - **`replicas: 3`**: Specifies the desired number of replica Pods the Deployment should manage. In this case, it's set to 3, meaning the Deployment will try to ensure that there are always three Pods running.

- **`selector`**:
    - **`matchLabels`**:
        - **`app: your-application`**: Specifies how the Deployment identifies which Pods to manage. The selector matches the labels assigned to Pods. In this case, the Deployment manages Pods that have the label `app` with the value `your-application`.
- **`template`**:
    - **`metadata`**:
        - **`labels`**:
            - **`app: your-application`**: Labels applied to Pods created from this template. These labels are used by the Deployment selector to identify the Pods it manages.
    - **`spec`**:
        - **`containers`**: Defines the container specifications for Pods created by this Deployment.
            - **`name: your-application`**: The name of the container within the Pod. This name is used to identify the container within the Pod and can be used for logging, debugging, and other operations.
            - **`image: registry-url/your-application-name:tag`**: Specifies the Docker image to use for the container. This should be the path to the image in a Docker registry, including the tag to specify the version of the image to use.
            - **`ports`**:
                - **`containerPort: 80`**: Specifies the port that the container exposes. In this case, it's set to port 80, indicating that the application within the container listens on port 80.

this Deployment configuration named **`your-application-deployment`** is designed to ensure that three replicas of a Pod are running at all times. Each Pod runs a container based on the specified Docker image **`registry-url/your-application-name:tag`** and exposes port 80. The Deployment manages Pods that are labeled with `app:` **`your-application`**, ensuring that the desired state of having three replicas is maintained, handling scaling and updates as defined by the Deployment configuration.

Service resource (service.yaml)

```
apiVersion: v1
kind: Service
metadata:
  name: your-application-service
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: your-application
```

This Kubernetes Service resource is a configuration that defines how to expose an application running on a set of Pods as a network service. Let's break down its components:

- **`apiVersion: v1`**: Specifies the API version for the Service resource. `v1` is the core group version and indicates a stable version of the resource definition.
- **`kind: Service`**: Defines the kind of Kubernetes resource being configured. In this case, it's a Service, which is used to expose applications running on Pods as network services.
- **`metadata`**:
    - **`name: your-application-service`**: The name of the Service resource. This name is how the Service will be identified within the Kubernetes cluster.
- **`spec`** (specification):
    - **`type: LoadBalancer`**: Specifies the type of Service. A LoadBalancer Service makes your application accessible from the internet by provisioning a cloud provider's load balancer to route external traffic to the Service. This type is commonly used when running Kubernetes in cloud environments that offer load balancer services.
    - **`ports`**: Defines the port settings for the Service.
        - **`port: 80`**: The port on which the Service is exposed. This is the port that will be used by external clients to access the Service.
        - **`targetPort: 80`**: Specifies the port on the Pods to which the traffic will be forwarded. In this case, traffic received on port 80 of the Service will be forwarded to port 80 on the Pods selected by the Service.
    - **`selector`**:
        - **`app: your-application`**: Specifies how the Service identifies which Pods to target for routing traffic. The selector matches the labels assigned to Pods. In this case, the Service routes traffic to Pods that have the label `app` with the value `your-application`.

this Service resource configuration creates a LoadBalancer type Service named **`your-application-service`**. It exposes the application on port 80 to the internet through a load balancer provided by the cloud platform. The Service forwards traffic arriving at this port to port 80 on Pods labeled with `app:` **`your-application`**. This setup is commonly used for applications that need to be accessible from outside the Kubernetes cluster, providing an easy way to expose services to the internet with minimal configuration.

Ingress resource (ingress.yaml)

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: your-application-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$2
    nginx.ingress.kubernetes.io/enable-cors: "true"
  labels:
    name: your-application-ingress
spec:
  rules:
  - host: "hostname.slac.stanford.edu"
    http:
      paths:
      - pathType: Prefix
        path: /uri/prefix(/|$)(.*)
        backend:
          service:
            name: elog-plus-backend-service
            port:
              number: 80
```

- **`apiVersion: networking.k8s.io/v1`**: Specifies the API version for the Ingress resource. `networking.k8s.io/v1` indicates that this is a stable version of the Ingress API.
- **`kind: Ingress`**: Defines the kind of Kubernetes resource being configured. In this case, it's an Ingress, which is used for managing access to services within the cluster from the outside.
- **`metadata`**:
  - **`name: your-application-ingress`**: The name of the Ingress resource.
  - **`annotations`**:
    - **`nginx.ingress.kubernetes.io/rewrite-target: /$2`**: This annotation for NGINX Ingress controller specifies the rewrite path target. When a request matches the given path pattern, the path is rewritten before forwarding the request to the backend service. `/$2` means that the captured group `(.*)` from the path is appended to the `/`, effectively transforming the original request to the captured suffix.
    - **`nginx.ingress.kubernetes.io/enable-cors: "true"`**: Enables Cross-Origin Resource Sharing (CORS) for requests coming to this Ingress, allowing resources to be requested from another domain.
  - **`labels`**:
    - **`name: your-application-ingress`**: A label to identify this Ingress resource. Labels can be used for organizing and selecting subsets of objects.
- **`spec`**:
  - **`rules`**: Defines the rules for routing traffic.
    - **`host: "hostname.slac.stanford.edu"`**: Specifies the host on which the rule applies. This means that the Ingress will apply to requests made to `hostname.slac.stanford.edu`.
    - **`http`**:
      - **`paths`**:
        - **`pathType: Prefix`**: Indicates that the path specified is to be interpreted as a prefix. This means any path that has this prefix will match.
        - **`path: /uri/prefix(/|$)(.*)`**: Specifies the URI path that, when matched, will route requests to the specified backend. The regex `(/|$)(.*)` captures everything after `/uri/prefix`, allowing for flexible path matching.
        - **`backend`**:
          - **`service`**:
            - **`name: elog-plus-backend-service`**: The name of the backend service to which traffic should be routed when the path matches.
            - **`port`**:
              - **`number: 80`**: The port number on the backend service to which the traffic should be sent.

This Ingress resource configuration effectively routes HTTP traffic coming to `hostname.slac.stanford.edu/uri/prefix` and any subpath (`/uri/prefix/anything-here`) to the `elog-plus-backend-service` on port 80. The path rewrite and CORS settings further customize how the incoming requests are handled and forwarded to the backend service.

Use other Kubernetes resources as needed (ConfigMaps, Secrets, Volumes) to manage configurations, sensitive information, and persistent data.

The relationship between Deployment, Service, and Ingress resources in Kubernetes is a fundamental aspect of how applications are deployed, exposed, and accessed within a Kubernetes cluster. Each of these resources serves a specific role in the application deployment and access workflow. Here’s how they interrelate:

Summary of the relationship between the above describe resources

Deployment

- **Role**: Manages the deployment and scaling of a set of Pods and ensures that the specified number of Pods are running and up-to-date. It abstracts the management of ReplicaSets and Pods, handling updates and rollbacks.
- **Relationship**: Deployments create and manage Pods based on a defined template. These Pods contain the actual running instances of your application containers.

Service

- **Role**: Provides a stable endpoint for accessing the Pods managed by a Deployment. Services abstract the Pod IP addresses, making the backend Pods accessible without needing to know the specific Pods. Services can be of different types, such as ClusterIP (default, internal communication), NodePort (exposes the Service on each Node’s IP at a static port), and LoadBalancer (exposes the Service externally using a cloud provider’s load balancer).
- **Relationship**: A Service selects Pods based on labels (specified in the Deployment's Pod template) and provides a single access point to access those Pods. This means that the Service acts as a load balancer, distributing network traffic to the Pods.

Ingress

- **Role**: Manages external access to the services within the cluster, typically HTTP/HTTPS traffic. Ingress can provide load balancing, SSL termination, and name-based virtual hosting. It acts as an entry point for your cluster's services, allowing you to define accessible URLs, load balancing policies, and more.
- **Relationship**: Ingress is used to route external requests to the Services within the cluster. The Ingress resource defines rules for routing traffic to different Services based on the request host or path. It allows more complex traffic routing than a Service of type LoadBalancer, enabling you to expose multiple services under a single IP address.

Workflow Summary

1. **Deployment**: You define a Deployment to manage your application’s Pods. The Deployment ensures the desired number of Pods are always running and manages updates to your application.
2. **Service**: To expose the Pods managed by the Deployment internally within the cluster or externally, you define a Service. The Service provides a stable endpoint (DNS name or IP address) that routes traffic to the Pods.
3. **Ingress**: For more advanced routing needs (e.g., path-based routing, SSL termination, domain name-based access), you define an Ingress. The Ingress controls access to the Services from outside the Kubernetes cluster, routing external requests to the appropriate Services based on the defined rules.

Deploy Your Application

When the resources are ready and the k8s is accessible with the kubectl CLI, the resources can be created in this way.

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

- How to deploy webapp on kubernetes