5-Week Training Plan: Service Mesh, Kubernetes, and Related Technologies

        - Core concepts of service mesh
        - Problems service meshes solve
        - Evolution of ingress
    - Service Mesh Architecture
        - Data Plane
        - Control Plane
    - Key Features and Use Cases
        - Service Discovery and Load Balancing
        - Traffic Management
        - Observability
        - Security
        - Challenges and Best Practices
        - Performance Considerations
        - Complexity Management
        - Monitoring and Troubleshooting
- Day 3-4: Python for Kubernetes
    - Python basics review (if needed)
    - Data Types
    - Python Package Management
        - Installing a package:
        - Upgrading a package:
    - Python Virtual Environments
        - Creating a virtual environment:
        - Activating a virtual environment:On Unix or MacOS:
        - Installing packages in a virtual environment:
        - Deactivating a virtual environment:
        - Creating a requirements file:
        - Installing from a requirements file:
    - Kubernetes Python client library
    - Simple Python scripts for Kubernetes interaction
- Day 5: Helm Basics
    - Helm's Purpose and Architecture
    - Creating and Structure of a Helm Chart
    - Deploying Applications with Helm
    - Advanced Helm Concepts
        - Hooks
    - Dependencies
    - Templating
    - Creating Helm Charts with Python Templates
        - Using Jinja2 for Templating
        - Generating Kubernetes Manifests Dynamically
        - Integrating with CI/CD Pipelines

- Week 3: Istio Deep Dive
    - Day 1: Istio Basics
        - Installing Istio on your Kubernetes cluster
            - Download Istio
            - Install Istio
            - Enable automatic sidecar injection
        - Istio's architecture and core components
            - Control Plane
            - Data Plane
            - Addons

- Day 2: Istio Traffic Management
  - Exploring Istio's traffic management features
  - Implementing canary deployments and A/B testing
  - Istio's load balancing and circuit breaking capabilities
- Day 3: Istio Security and Observability
  - Istio's security features
    - mTLS (Mutual TLS)
    - Authorization Policies
  - Exploring Istio's observability stack
    - Prometheus
    - Grafana
    - Kiali
    - Jaeger/Zipkin
- Day 4-5: Deploying a Sample Application with Istio
  - Objective
  - Prerequisites
  - Enable Istio Sidecar Injection
  - Deploy a Sample Application
  - Create a Virtual Service
  - Create a Destination Rule
  - Test the Routing
  - Implement Canary Deployment
  - observability

- Week 4: Linkerd and Practical Applications
  - Day 1: Linkerd Basics
    - Installing Linkerd on your Kubernetes cluster
      - Install CLI
      - Install Linkerd on Your Minikube Cluster
      - Validate cluster
      - Install Linkerd
      - Install viz
      - Linkerd's architecture and core components
      - Control Plane
      - Data Plane
      - Add-ons
    - Linkerd Features
    - Traffic management capabilities
    - Linkerd's observability and security features
  - Day 2-4: Hands-on Exercise
    - Deploying and Managing emojivoto with Linkerd
      - Deploy the emojivoto sample application
      - Inject Linkerd into the application
      - Observe traffic
        - Install smi
      - Visualize the service mesh
      - Implement a traffic split for canary deployment
      - Observe the traffic split
      - Gradually increase traffic to the new version
      - Monitor the canary deployment
  - Day 5: Service Mesh Comparison
    - Comparing Istio, Linkerd, and other service mesh solutions
      - Istio

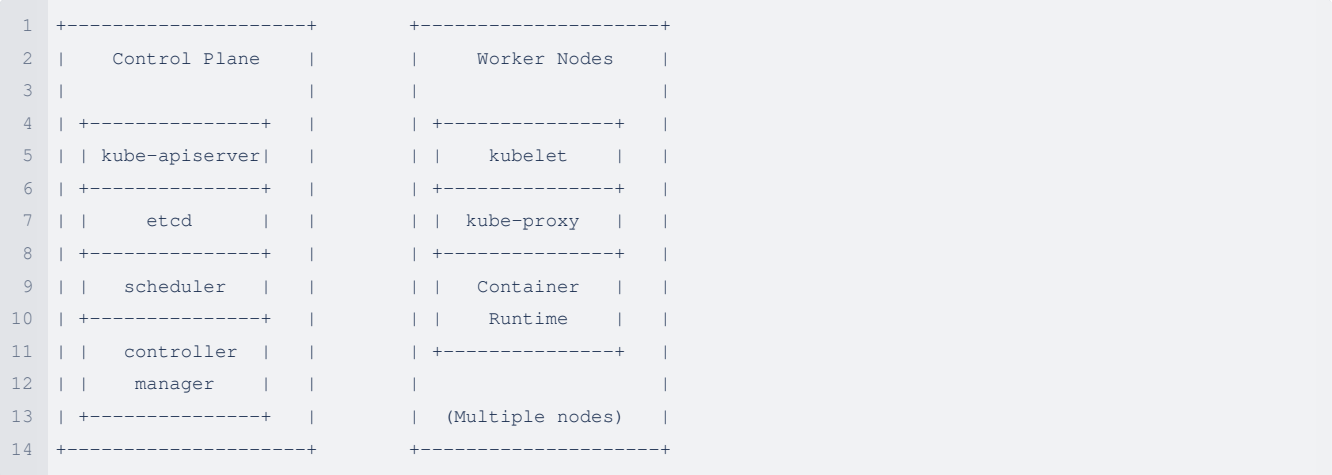
This document is meant to be a central spring point to allow you to understand points to cover yet expects the user to use external resources to dig deeper in the points and subjests

Week 1: Fundamentals and Kubernetes

Day 1-2: Kubernetes Basics and Local Development Environments

Kubernetes Architecture and Core Concepts

Kubernetes is a powerful container orchestration platform that manages containerized applications across multiple hosts. Its architecture consists of two main components: the control plane and worker nodes
source k8s.



Control Plane Components

1. kube-apiserver: The API server is the front-end for the Kubernetes control plane. It exposes the Kubernetes API and handles all administrative operations.
2. etcd: A consistent and highly-available key-value store used as Kubernetes' backing store for all cluster data.
3. kube-scheduler: Responsible for assigning newly created pods to nodes based on resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, and more.
4. kube-controller-manager: Runs controller processes that regulate the state of the system. These controllers include the node controller, replication controller, endpoints controller, and service account & token controllers.
5. cloud-controller-manager: (Optional) Integrates with underlying cloud providers.

```
|                  Control Plane                  |
|                                                 |
| +----------------+  +--------------------------+ |
| | kube-apiserver |  |        scheduler         | |
| | (API Gateway)  |  | (Assigns Pods to Nodes)  | |
| +----------------+  +--------------------------+ |
|                                                 |
| +----------------+  +--------------------------+ |
| |      etcd      |  |    controller manager    | |
| | (Cluster State |  | (Maintains Desired State) | |
| |   Database)    |  |                          | |
| +----------------+  +--------------------------+ |
+-------------------------------------------------+
```

Node Components

1. kubelet: An agent that runs on each node, ensuring containers are running in a Pod.
2. kube-proxy: Maintains network rules on nodes, implementing part of the Kubernetes Service concept.
3. Container runtime: Software responsible for running containers (e.g., Docker, containerd, CRI-O).
4. Pods: The smallest deployable units in Kubernetes, consisting of one or more containers

```
+-------------------------------------------------+
|                   Worker Node                   |
| +---------------------+  +--------------------+ |
| |       kubelet       |  |    kube-proxy      | |
| |    (Node Agent)     |  |  (Network Proxy)   | |
| +---------------------+  +--------------------+ |
|                                                 |
| +---------------------------------------------+ |
| |              Container Runtime              | |
| |          (e.g., Docker, containerd)         | |
| +---------------------------------------------+ |
|                                                 |
| +---------------------------------------------+ |
| |                    Pods                     | |
| | +------------+ +------------+ +------------+ | |
| | | Container  | | Container  | | Container  | | |
| | +------------+ +------------+ +------------+ | |
| +---------------------------------------------+ |
+-------------------------------------------------+
```

Core Concepts

1. Pods: The smallest deployable units in Kubernetes, consisting of one or more containers.
2. Services: An abstraction that defines a logical set of Pods and a policy by which to access them.
3. Deployments: Provide declarative updates for Pods and ReplicaSets.
4. Namespaces: Virtual clusters backed by the same physical cluster, providing a way to divide cluster resources between multiple users.

Additional Components

These components include the Dashboard (a web-based UI), cluster-level logging, container resource monitoring, and network plugins.

```
+-------------------------------------------------+
|               Additional Components             |
|                                                 |
| +---------------------+  +--------------------+ |
| |      Dashboard      |  |  Cluster-level Logging | |
```

```
| |       (Web UI)       |  |(Centralized Log Storage)| |
| +----------------------+  +-------------------------+ |
|                                                       |
| +----------------------+  +-------------------------+ |
| |      Monitoring      |  |     Network Plugins     | |
| |(Resource Monitoring)|  |     (Implement CNI)     | |
| +----------------------+  +-------------------------+ |
+-------------------------------------------------------+
```

Local Kubernetes Development Options

kind (Kubernetes in Docker)

kind is a tool for running local Kubernetes clusters using Docker container "nodes". It's designed for testing Kubernetes itself, but can be used for local development or CI.

Installation

```
go install sigs.k8s.io/kind@v0.24.0
```

Or for macOS users

```
brew install kind
```

Creating a cluster

```
kind create cluster
```

Advantages of kind:

1. Lightweight and fast to start up, making it ideal for rapid development cycles.
2. Supports multi-node clusters, allowing you to simulate more complex environments.
3. Runs Kubernetes inside Docker containers, which is efficient and consistent across different host systems.
4. Ideal for testing and CI/CD pipelines due to its speed and reproducibility

Minikube

Minikube is a tool that makes it easy to run Kubernetes locally. It runs a single-node Kubernetes cluster inside a VM on your laptop.

Installation

For macOS

```
brew install minikube
```

For other systems, refer to the official documentation

Starting a cluster

```
minikube start
```

Advantages of Minikube:

1. More established and feature-rich, with a large community and extensive documentation.
2. Supports multiple hypervisors (VirtualBox, HyperKit, etc.), allowing flexibility in your local setup.
3. Provides built-in addons for common services, making it easy to enable additional functionality.
4. Offers a dashboard for visual management of your cluster.

Practice with Basic Kubernetes Resources

To solidify your understanding, practice creating and managing these basic Kubernetes resources in both kind and Minikube environments:

1. Pods: The smallest deployable units in Kubernetes.
2. Deployments: Manage the deployment and scaling of a set of Pods.
3. Services: Expose your application to network traffic.

Example commands:

Create a deployment

`kubectl create deployment nginx --image=nginx`

Expose the deployment as a service

`kubectl expose deployment nginx --port=80 --type=LoadBalancer`

List pods

`kubectl get pods`

List services

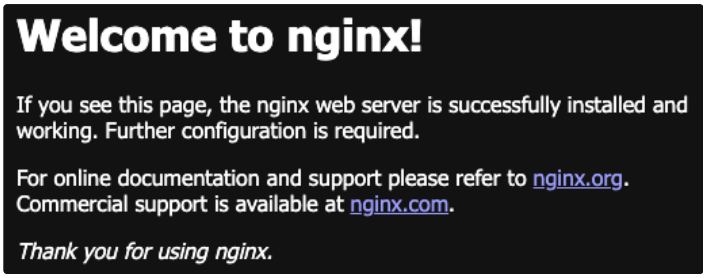
`kubectl get services`

By thoroughly understanding these concepts and practicing with both kind and Minikube, you'll build a solid foundation for working with Kubernetes in various environments.

You will need to search so that you can view the nginx on your localhost

eg: `minikube external ip expose command`

You will ultimately see the nginx default banner



Day 3-4: Advanced Kubernetes

ConfigMaps, Secrets, and Volumes

```
+-----------------------------------------------------+
|                         Pod                         |
|                                                     |
| +-----------------+  +----------------------------+ |
| |   Container     |  |      Volume Mounts         | |
| | (Application)   |  | /etc/config -> ConfigMap   | |
| |                 |  | /etc/secrets -> Secret     | |
| +-----------------+  +----------------------------+ |
|                                                     |
| +-----------------+  +----------------------------+ |
| |  Environment    |  |         ConfigMap          | |
| |   Variables     |  |                            | |
| | (from ConfigMap|  |    key1: value1            | |
| |  and Secret)    |  |    key2: value2            | |
| +-----------------+  +----------------------------+ |
|                                                     |
|                      +----------------------------+ |
|                      |          Secret            | |
|                      |  username: base64(user)    | |
|                      |  password: base64(pass)    | |
|                      +----------------------------+ |
+-----------------------------------------------------+
```

ConfigMaps

ConfigMaps

- Used to store non-confidential data in key-value pairs.
- Can be consumed as environment variables, command-line arguments, or configuration files in a volume.
- Example creation:
  `kubectl create configmap name --from-literal=name='{"first": "John", "second": "Doe"}'`
- Example extract
  `kubectl get configmap name -o jsonpath='{.dataname}'` or `kubectl get configmap name3 -o json | jq -r '.data.name'| jq -r .first`

Secrets

Managing Secrets using kubectl

- Similar to ConfigMaps but intended for confidential data.
- Base64 encoded by default (not encrypted).
- Can be mounted as files or exposed as environment variables.
- Example creation:
  `kubectl create secret generic user-pass --from-literal=username=john --from-literal=password=s3cr3t`
- Example extract:
  `kubectl get secrets user-pass -o json | jq -r .data.password | base64 -D`

Volumes

Volumes

- Provide persistent storage for pods.
- Types include emptyDir, hostPath, nfs, and cloud provider-specific options.
- PersistentVolumes (PV) and PersistentVolumeClaims (PVC) provide a way to use storage resources in a pod-independent manner.
- Example
  Create a configmap to hold your var
  `kubectl create configmap config-vol --from-literal=log_level=debug`
  Now create a pod with a running container that mounts the configmap as a var

```
cat <<EOF | k apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod
spec:
  containers:
    - name: test
      image: busybox:1.28
      command: ['sh', '-c', 'echo "The app is running!" && tail -f /dev/null']
      volumeMounts:
        - name: config-vol
          mountPath: /etc/config
  volumes:
    - name: config-vol
      configMap:
        name: config-vol # Corrected to match the ConfigMap name
        items:
          - key: log_level
            path: log_level
EOF
```

  Run a command to extract the var held at this point

`kubectl exec -it configmap-pod -- cat /etc/config/log_level`

OR

Exec into the container

`kubectl exec -it configmap-pod -- sh`

Here you can navigate to the location

`cd etc/config`

`ls` < here you should see log_level

`cat log_level`

`debug/etc/config`

To give a cleave output

`cat log_level ; echo`

This could easily be a static volume location as opposed to a configmap

```
+-------------------------------------------------+
|                     Node                        |
| +--------------------+  +----------------------+ |
| |        Pod         |  |   Persistent Volume  | |
| |  +--------------+  |  |                      | |
| |  |  Container   |  |  | (Network File Ststem,| |
| |  | /Volume Mount|  |  |  Cloud Storage, etc. | |
| |  +--------------+  |  |                      | |
| +--------------------+  +----------------------+ |
|                                                 |
| +--------------------+  +----------------------+ |
| |  Empty Dir Volume  |  |  Host Path Volume    | |
| |(Temporary Storage) |  | (Node's file system) | |
| +--------------------+  +----------------------+ |
|                                                 |
+-------------------------------------------------+
```

Kubernetes Networking and Ingress

Networking is a large area of K8s and is the largest challenge or concept to learn.

```
                    External Traffic
                           |
                           v
                +-----------------------+
                |     Load Balancer     |
                +-----------------------+
                           |
                           v
               +---------------------------+
               |    Ingress Controller     |
               |  (e.g., NGINX, Traefik)   |
               +---------------------------+
                 |                       |
                 v                       v
       +-----------------+      +-----------------+
       | Ingress Rule 1  |      | Ingress Rule 2  |
       |  host: foo.com  |      |  host: bar.com  |
       |  path: /app1    |      |  path: /app2    |
       +-----------------+      +-----------------+
                |                        |
                v                        v
   +----------------------+ +----------------------+
```

```
   |      Service 1       | |      Service 2       |
   | (ClusterIP/NodePort) | | (ClusterIP/NodePort) |
   +----------------------+ +----------------------+
          |          |            |          |
          v          v            v          v
     +--------+ +--------+    +--------+ +--------+
     | Pod 1A | | Pod 1B |    | Pod 2A | | Pod 2B |
     +--------+ +--------+    +--------+ +--------+
         |          |             |          |
         v          v             v          v
     +-------------------------------------------+
     |             Container Network             |
     |   (e.g., Flannel, Calico, Weave, Cilium)  |
     +-------------------------------------------+
                          |
                          v
                 +----------------+
                 |  Node Network  |
                 +----------------+
```

This diagram illustrates:

1. External traffic enters through a Load Balancer.
2. The Ingress Controller (e.g., NGINX or Traefik) receives the traffic and processes it based on Ingress Rules.
3. Ingress Rules define how traffic should be routed based on hostnames and paths.
4. Services (ClusterIP or NodePort) receive traffic from the Ingress Controller and distribute it to Pods.
5. Pods contain the application containers and are distributed across nodes.
6. The Container Network (implemented by CNI plugins like Flannel, Calico, Weave, or Cilium) enables communication between Pods across nodes.
7. The Node Network connects all nodes in the cluster.

Networking Model

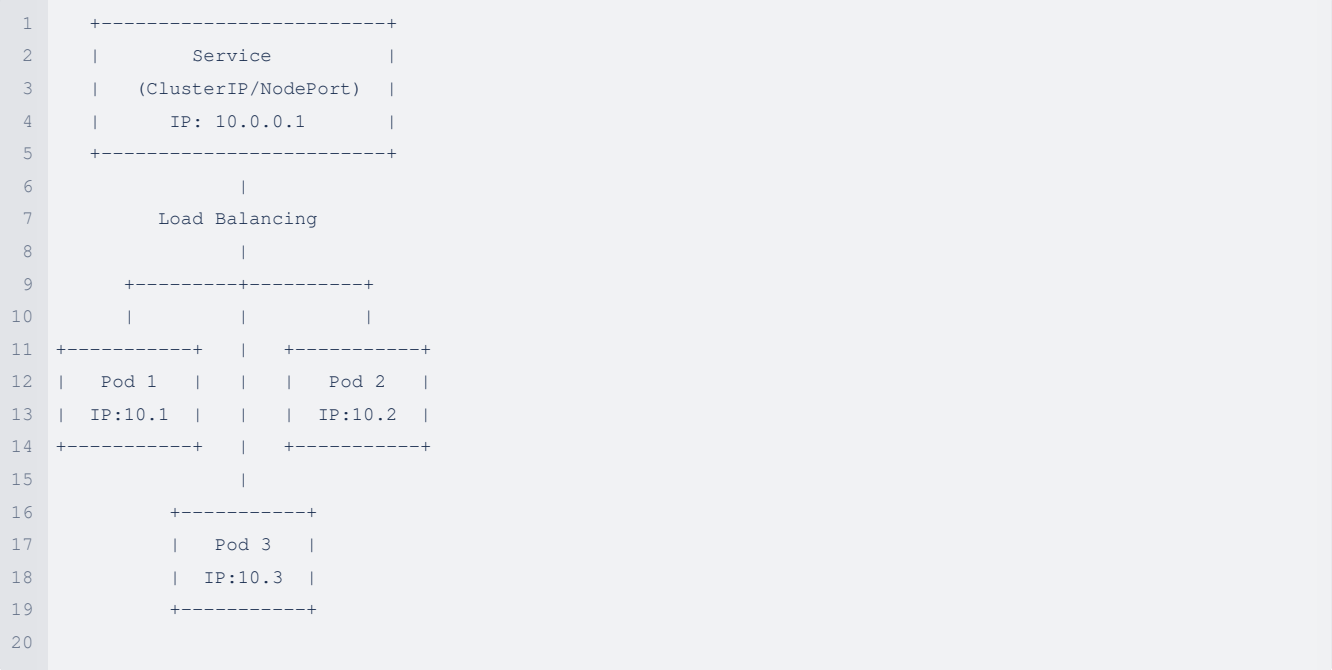
```
+-------------------------------------+    +-------------------------------------+
|                Node 1               |    |                Node 2               |
| +--------------+   +--------------+ |    | +--------------+   +--------------+ |
| |     Pod1     |   |     Pod2     | |    | |     Pod3     |   |    Pod4      | |
| |  IP: 10.1.1  |   |  IP: 10.1.2  | |    | |  IP: 10.2.1  |   |  IP: 10.2.2  | |
| +--------------+   +--------------+ |    | +--------------+   +--------------+ |
|                  |                  |    |                  |                  |
|      Virtual Ethernet Bridge        |    |        Virtual Ethernet Bridge      |
|                  |                  |    |                  |                  |
+------------------|------------------+    +------------------|------------------+
                   |                                          |
                   |          Cluster Network Fabric          |
                   +------------------------------------------+

```

- Pod IP Addressing: Each pod is assigned a unique IP address from the cluster-wide CIDR range. This ensures that every pod has a distinct identity within the cluster.
- Direct Communication: Pods can communicate directly with each other using their assigned IP addresses, without the need for Network Address Translation (NAT) or port mapping.
- Intra-Node Communication: For pods on the same node, communication occurs through a virtual ethernet bridge. This allows for efficient local traffic routing.
- Inter-Node Communication: When pods on different nodes need to communicate, the cluster-level network layer handles routing based on the pod IP ranges assigned to each node.

- CNI Plugins: Container Network Interface (CNI) plugins implement the actual networking, ensuring proper routing and connectivity across the cluster. Popular CNI plugins include Calico, Flannel, and Weave.

This architecture simplifies application design and deployment, as pods can be treated similarly to VMs or physical hosts from a networking perspective.

Services

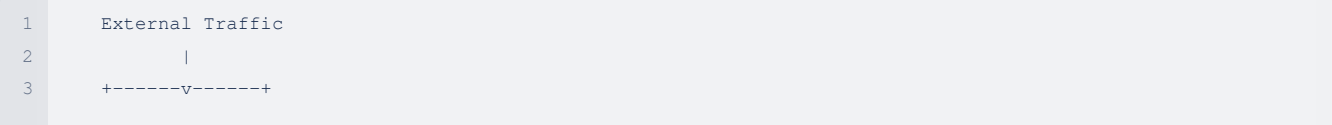


Kubernetes Services provide a stable network endpoint for a set of Pods, enabling reliable communication within the cluster. Services abstract the underlying Pod network, offering a consistent way to access applications regardless of Pod lifecycle changes. Key aspects of Kubernetes Services include:

- Service Types:
  - ClusterIP (default): Exposes the service on an internal IP in the cluster
  - NodePort: Exposes the service on each node's IP at a static port
  - LoadBalancer: Exposes the service externally using a cloud provider's load balancer
  - ExternalName: Maps the service to the contents of the externalName field
  - Headless: Allows direct access to individual pod IPs
- Service Discovery: Services can be discovered through DNS or environment variables, making it easy for applications to find and communicate with each other.
- Load Balancing: Services automatically distribute incoming traffic across all backend pods, ensuring even load distribution.
- Stable Endpoints: Services provide stable IP addresses and DNS names for groups of pods, abstracting away the dynamic nature of pod lifecycles.
- Cloud Integration: Services can integrate with cloud provider load balancers for external access, simplifying the process of exposing applications to the internet.

Services play a crucial role in microservices architectures, facilitating seamless communication between application components and enabling scalability and resilience in Kubernetes environments

Ingress



```
    |   Ingress   |
    | Controller  |
    +------+------+
           |
    +------v------+
    |   Ingress   |
    |    Rules    |
    +------+------+
           |
    +------v------+
    |  Services   |
    +------+------+
           |
      +----v----+
      |  Pods   |
      +---------+

```

Kubernetes Ingress is an API object that manages external access to services within a cluster, providing HTTP and HTTPS routing rules. It acts as a single entry point for incoming traffic, simplifying the exposure of multiple services through a unified interface. Key features of Ingress include:

- Traffic Routing: Ingress can route traffic based on URL paths, hostnames, or other criteria, allowing for complex routing scenarios.
- SSL/TLS Termination: Ingress can handle SSL/TLS termination, offloading this responsibility from individual services.
- Load Balancing: Ingress can distribute traffic across multiple backend services, acting as a load balancer.
- Name-based Virtual Hosting: Ingress supports routing to different services based on the hostname, enabling multiple applications to share a single IP address.
- Ingress Controller: Ingress requires an Ingress Controller to function, which implements the actual routing and load balancing logic. Popular Ingress Controllers include NGINX, Traefik, and Istio.

By consolidating routing rules into a single resource, Ingress simplifies network management and reduces the need for multiple load balancers, making it an essential component for production-ready Kubernetes deployments.

Examples:
Create a simple web application

```
cat <<EOF | k apply -f -
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-app
        image: nginx:latest
        ports:
        - containerPort: 80
---
apiVersion: v1
```

```
kind: Service
metadata:
  name: web-app-service
spec:
  selector:
    app: web-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80

EOF
```

This will create an app named web-app with a port 80 exposure to the pod.
It will also create a service directing calls to the deployment named web-app on port 80 to port 80 of one of the containers.

`kubectl get deployments`

`kubectl get pods`

`kubectl get services`

giving something like

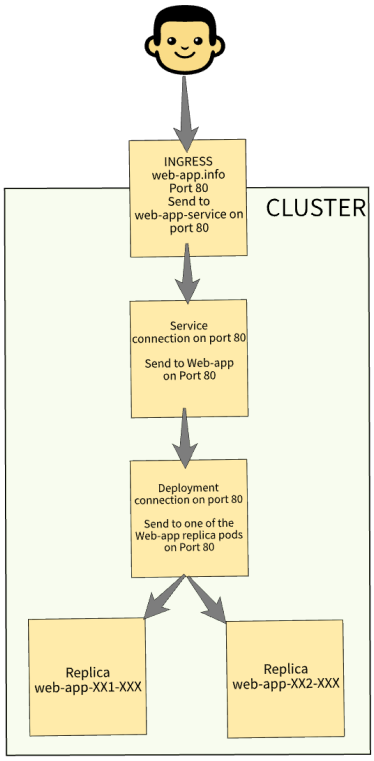
```
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
web-app   2/2     2            2           46s

NAME                       READY   STATUS    RESTARTS   AGE
web-app-6fdf6bcdd6-cfkjk   1/1     Running   0          42s
web-app-6fdf6bcdd6-nxv7f   1/1     Running   0          42s

NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
kubernetes        ClusterIP   10.96.0.1       <none>        443/TCP   57s
web-app-service   ClusterIP   10.110.70.144   <none>        80/TCP    46s
```

Now create an ingress to create access

```
cat <<EOF | k apply -f -
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: web-app-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$1
spec:
  rules:
  - host: web-app.info
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: web-app-service
            port:
              number: 80
EOF
```

This will create an ingress that will create a connection outside of the cluster with web-app.info as the host name that will direct all connections to port 80 of web-app-service service that will then forward this to port 80 of the deployment for forwarding to one of the replicas

for connection.



`kubectl get ingress`

```
NAME             CLASS   HOSTS   ADDRESS   PORTS   AGE
web-app-ingress  <none>  http://web-app.info  80  2m28s
```

Ensure that the Ingress addon is enabled in Minikube.

`minikube addons enable ingress`

This command enables the NGINX Ingress Controller in your Minikube cluster.

Obtain the IP address of your Minikube cluster.

`minikube ip`

This will return the IP address of your Minikube cluster.

Add an entry to your hosts file for `web-app.info` to the Minikube IP.

```
echo "$(minikube ip) web-app.info" | sudo tee -a /etc/hosts
```

This step is necessary because you've specified `web-app.info` as the host in your Ingress resource.

Now you should be able to access your application by opening a web browser and navigating to: `http://web-app.info`

If everything is set up correctly, you should see the NGINX welcome page.

If you're unable to access the application, try the following:
Check Ingress status `kubectl get ingress`, ensure that the ADDRESS field is populated with an IP address.

Verify ingress `kubectl get pods -n ingress-nginx` , make sure the Ingress Controller pod is running.

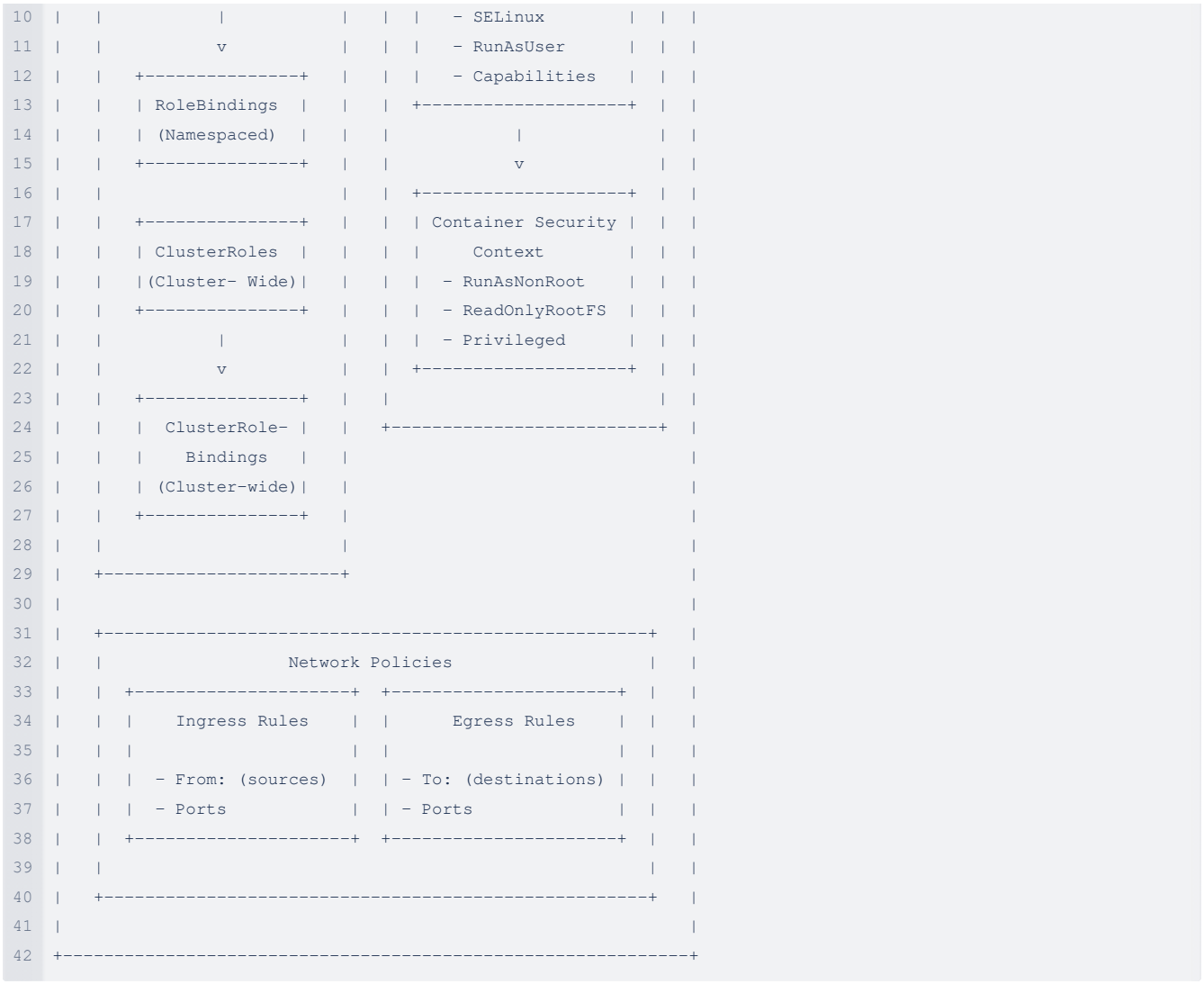
Check ingress logs looking for ERRORS `kubectl logs -n ingress-nginx $(kubectl get pods -n ingress-nginx -o name)` (this can be run in seperate parts `kubectl get pods -n ingress-nginx -o name` then run `kubectl logs -n ingress-nginx` with the ingress)

Last resort you can try port forwarding. `kubectl port-forward svc/web-app-service 8080:80` , now access the application at `http://localhost:8080`.

Remember that Minikube is running inside a VM, so network access can sometimes be tricky depending on your setup. The methods described above should work in most cases, but you might need to adjust based on your specific environment

Kubernetes RBAC and Security Concepts

```
+-------------------------------------------------------------+
|                      Kubernetes Cluster                     |
|                                                             |
|   +-----------------------+   +---------------------------+ |
|   |      RBAC Objects     |   |    Security Contexts      | |
|   |   +---------------+   |   |  +---------------------+  | |
|   |   |     Roles     |   |   |  |    Pod Security     |  | |
|   |   | (Namespaced)  |   |   |  |      Context        |  | |
|   |   +---------------+   |   |  |   - User/Group      |  | |
```



- RBAC Objects:
  - Roles and RoleBindings (namespaced)
  - ClusterRoles and ClusterRoleBindings (cluster-wide)
    These objects define who can access what resources and perform what actions.
- Security Contexts:
  - Pod Security Context: Applies to all containers in a pod
  - Container Security Context: Specific to individual containers
    These define privilege and access control settings for pods and containers.
- Network Policies:
  - Ingress Rules: Control incoming traffic to pods
  - Egress Rules: Control outgoing traffic from pods
    These act as a virtual firewall for your Kubernetes cluster.

The diagram shows how these components interact within the Kubernetes cluster to provide a comprehensive security model. RBAC controls access to Kubernetes API resources, Security Contexts manage the runtime security settings for pods and containers, and Network Policies control the network traffic between pods and external sources

Role-Based Access Control (RBAC)

- Regulates access to resources based on the roles of individual users.
- Key objects: Role, ClusterRole, RoleBinding, ClusterRoleBinding.
  Example: Creating a role that allows reading pods:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
    namespace: default
    name: pod-reader
Rules:
- apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "watch", "list"]
```

Security Contexts

- Define privilege and access control settings for Pods or Containers.
- Can set UID, GID, capabilities, and other security parameters.

Network Policies

- Specify how groups of pods are allowed to communicate with each other and other network endpoints.
- Act as a virtual firewall for your Kubernetes cluster.

Exercise:
Deploying a Configurable Web ApplicationIn this exercise, we'll create a simple web application that reads its configuration from a ConfigMap. We'll then deploy it to Kubernetes and expose it using a Service and Ingress.
This exercise demonstrates:

1. Creating and using ConfigMaps
2. Deploying a web application with Kubernetes
3. Exposing the application using a Service and Ingress
4. Injecting configuration into a container using environment variables
5. Mounting ConfigMap data as a volume
6. Updating configuration and seeing the changes reflected in the application

- **Step 1:** Create a ConfigMap
  First, let's create a ConfigMap with some configuration data:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: ConfigMap
metadata:
  name: webapp-config
data:
  BACKGROUND_COLOR: "#f0f0f0"
  MESSAGE: "Welcome to our configurable web app!"
EOF
```

```
+-----------------------------------------------------+
|                  Kubernetes Cluster                 |
|                                                     |
|   +---------------------------------------------+   |
|   |               ConfigMap                     |   |
|   |  Name: webapp-config                        |   |
|   |  Data:                                      |   |
|   |  +-----------------------------------+      |   |
|   |  |        Key        |       Value   |      |   |
|   |  +-------------------+---------------+      |   |
```

```
|   |  | BACKGROUND_COLOR  |      "#f0f0f0"    |  |  |
|   |  +-------------------+-------------------+  |  |
|   |  | MESSAGE           | "Welcome to our   |  |  |
|   |  |                   | configurable      |  |  |
|   |  |                   | web app!"         |  |  |
|   |  +-------------------+-------------------+  |  |
|   |                                              |  |
|   +----------------------------------------------+  |
|                                                     |
+-----------------------------------------------------+
```

This diagram shows:

1. The overall Kubernetes cluster environment.
2. Within the cluster, a ConfigMap named "webapp-config" is created.
3. The ConfigMap contains two key-value pairs:
   - BACKGROUND_COLOR: "#f0f0f0"
   - MESSAGE: "Welcome to our configurable web app!"

The diagram illustrates how the ConfigMap stores configuration data as key-value pairs, which can be used by applications running in the cluster. This ConfigMap could be mounted as a volume or used as environment variables in a Pod, allowing the application to access these configuration values at runtime.

- **Step 2:** Create a Deployment
  Now, let's create a Deployment for our web application. We'll use a simple Nginx image and inject our configuration as environment variables:

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
spec:
  replicas: 2
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: webapp
        image: nginxtest
        ports:
        - containerPort: 80
        envFrom:
        - configMapRef:
            name: webapp-config
        volumeMounts:
        - name: config
          mountPath: /usr/share/nginx/html
      volumes:
      - name: config
        configMap:
          name: webapp-content
```

```
          items:
          - key: index.html
            path: index.html
EOF
```

This diagram illustrates:

1. The overall Kubernetes Deployment named "webapp".
2. The ReplicaSet managing 2 replicas (Pods).
3. The structure of each Pod, including:
   - The container named "webapp" using the nginx:alpine image.
   - The container port 80 exposed.

    - Environment variables loaded from the ConfigMap "webapp-config".
    - A volume mount for the "/usr/share/nginx/html" path.
4. The volume configuration, which mounts the "index.html" key from the "webapp-config" ConfigMap.

The diagram shows how the Deployment manages multiple identical Pods, each containing a container with the specified configuration. It also illustrates the use of ConfigMaps for both environment variables and file mounting, demonstrating how Kubernetes can inject configuration data into containers.

- **Step 3:** Create a ConfigMap for the HTML content
  Let's create another ConfigMap to hold our HTML content:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: ConfigMap
metadata:
  name: webapp-content
data:
  index.html: |
    <!DOCTYPE html>
    <html>
    <head>
      <title>Configurable Web App</title>
      <style>
        body { background-color: \${BACKGROUND_COLOR}; font-family: Arial, sans-serif; }
      </style>
    </head>
    <body>
      <h1>\${MESSAGE}</h1>
      <p>This page is served by Nginx and configured using Kubernetes ConfigMaps.</p>
    </body>
    </html>
EOF
```

```
|   |  +---------------------------------------+  |   |
|   |  |                Pod                    |  |   |
|   |  |   +-------------------------------+    |  |   |
|   |  |   |        Container: webapp      |    |  |   |
|   |  |   |                               |    |  |   |
|   |  |   |  - Image: nginx:alpine        |    |  |   |
|   |  |   |  - Port: 80                   |    |  |   |
|   |  |   |                               |    |  |   |
|   |  |   |  EnvFrom:                     |    |  |   |
|   |  |   |    ConfigMap: webapp-config   |    |  |   |
|   |  |   |                               |    |  |   |
|   |  |   |  VolumeMount:                 |    |  |   |
|   |  |   |    Name: config               |    |  |   |
|   |  |   |    MountPath: /usr/share/...  |    |  |   |
|   |  |   +-------------------------------+    |  |   |
|   |  |                                        |  |   |
|   |  |   +-------------------------------+    |  |   |
|   |  |   |         Volume: config        |    |  |   |
|   |  |   |  ConfigMap: webapp-content    |    |  |   |
|   |  |   |    Key: index.html            |    |  |   |
|   |  |   |    Path: index.html           |    |  |   |
|   |  |   +-------------------------------+    |  |   |
|   |  +----------------------------------------+  |   |
|   +-----------------------------------------------+   |
+-------------------------------------------------------+
```

This updated diagram now includes:

1. The original `webapp-config` ConfigMap with `BACKGROUND_COLOR` and `MESSAGE`.
2. The new `webapp-content` ConfigMap containing the `index.html` template.
3. The Deployment and Pod structure, showing how these ConfigMaps are used:
   - `webapp-config` is used as environment variables (EnvFrom).
   - `webapp-content` is mounted as a volume, providing the `index.html` file.

The new `webapp-content` ConfigMap contains an HTML template that uses the `${BACKGROUND_COLOR}` and `${MESSAGE}` variables. These variables will be replaced with the actual values from the `webapp-config` ConfigMap when the page is served.This setup allows for a dynamic, configurable web application where:

- The content of the page (HTML structure) is defined in one ConfigMap (`webapp-content`).
- The configuration values (background color and message) are defined in another ConfigMap (`webapp-config`).
- The Nginx container serves the HTML content, with the variables replaced by the actual configuration values.

This separation of concerns makes it easy to update either the content template or the configuration values independently, providing flexibility in managing your web application's appearance and content.

- **Step 4:** Create a Service
  Now, let's create a Service to expose our Deployment:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
spec:
  selector:
    app: webapp
  ports:
```

```
    - protocol: TCP
      port: 80
      targetPort: 80
EOF
```

```
|   +---------------------------------------------+  |
+-----------------------------------------------------+
```

This updated diagram now includes:

1. The original `webapp-config` ConfigMap with `BACKGROUND_COLOR` and `MESSAGE`.
2. The `webapp-content` ConfigMap containing the `index.html` template.
3. The Deployment and Pod structure, showing how these ConfigMaps are used.
4. The new `webapp-service` Service, which:
   - Selects Pods with the label `app: webapp`
   - Exposes port 80 and forwards traffic to the Pods' port 80

The Service acts as a stable network endpoint for the Pods created by the Deployment. It provides:

- Load balancing: Distributes incoming traffic across all Pods matching the selector.
- Service discovery: Provides a stable IP address and DNS name for the set of Pods.
- Port mapping: Maps the Service port (80) to the target port on the Pods (also 80 in this case).

This Service allows other components within the cluster (or external to the cluster, depending on the Service type) to access the webapp Pods without needing to know the individual Pod IP addresses. It adds a layer of abstraction that enhances the scalability and flexibility of your application.The flow of traffic would typically be:External Request -> Service (webapp-service) -> Pod (webapp) -> Container (nginx:alpine)This setup allows you to scale your Deployment (adding or removing Pods) without changing how other components interact with your webapp, as they will always communicate through the Service.

- **Step 5:** Create an Ingress
  If your cluster has an Ingress controller, you can create an Ingress resource:

```
cat <<EOF | kubectl apply -f -
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: webapp-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: webapp.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: webapp-service
            port:
              number: 80
EOF
```

```
+-----------------------------------------------------+
|                   Kubernetes Cluster                |
|                                                     |
|   +---------------------------------------------+   |
|   |               ConfigMap: webapp-config      |   |
|   |                                             |   |
|   |  Data:                                      |   |
|   |  BACKGROUND_COLOR: "#f0f0f0"                |   |
|   |  MESSAGE: "Welcome to our configurable..."  |   |
```

This updated diagram now includes:

1. The original `webapp-config` ConfigMap with `BACKGROUND_COLOR` and `MESSAGE`.
2. The `webapp-content` ConfigMap containing the `index.html` template.

3. The Deployment and Pod structure, showing how these ConfigMaps are used.
4. The `webapp-service` Service that exposes the Pods.
5. The new `webapp-ingress` Ingress resource, which:
    - Routes traffic for the host `webapp.example.com`
    - Directs all paths (`/`) to the `webapp-service` on port 80

The Ingress resource acts as an entry point for external traffic into the cluster. It provides:

- Host-based routing: It routes traffic based on the `webapp.example.com` hostname.
- Path-based routing: In this case, all paths (`/`) are routed to the backend service.
- Integration with the Ingress Controller: The `nginx.ingress.kubernetes.io/rewrite-target: /` annotation is specific to the NGINX Ingress Controller, indicating that the path should be rewritten to `/` when forwarding to the backend.

The flow of traffic would now be:External Request -> Ingress Controller -> Ingress (webapp-ingress) -> Service (webapp-service) -> Pod (webapp) -> Container (nginx:alpine)This setup allows you to:

1. Access your application from outside the cluster using a domain name (`webapp.example.com`).
2. Potentially host multiple applications on the same IP address using different hostnames.
3. Implement more complex routing rules if needed (e.g., routing different paths to different services).

Remember to ensure that:

- The Ingress Controller is installed in your cluster.
- The DNS for `webapp.example.com` is configured to point to your cluster's external IP.
- Any necessary TLS certificates are configured if you want to enable HTTPS.

This Ingress resource completes the basic setup of a web application in Kubernetes, providing a full path for external traffic to reach your containerized application.

- Step 6: Verify the deployment
  Check if all resources are created and running:
  `kubectl get configmaps`
  `kubectl get deployments`
  `kubectl get pods`
  `kubectl get services`
  `kubectl get ingress`

```
kubectl get configmaps
NAME               DATA   AGE
kube-root-ca.crt   1      21m
webapp-config      2      18m
webapp-content     1      13m

kubectl get deployments
NAME     READY   UP-TO-DATE   AVAILABLE   AGE
webapp   2/2     2            2           11m

kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
webapp-756448c658-8h5lz   1/1     Running   0          7m26s
webapp-756448c658-b6gnr   1/1     Running   0          7m33s

kubectl get services
NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
kubernetes       ClusterIP   10.96.0.1       <none>        443/TCP   22m
```

```
webapp-service   ClusterIP   10.107.192.80   <none>        80/TCP    5m20s

kubectl get ingress
NAME             CLASS    HOSTS                ADDRESS   PORTS   AGE
webapp-ingress   <none>   webapp.example.com             80      3m52s
```

- **Step 7:** Access the application

  If you're using Minikube, you can use port-forwarding to access the application:

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

  Then open a web browser and go to `http://localhost:8080`.If you're using an Ingress, add the following to your `/etc/hosts` file:

  `echo "127.0.0.1 web-app.info" | sudo tee -a /etc/hosts`

  Then access the application at `http://webapp.example.com`.

- **Step 8:** Modify the configuration

  Let's change the background color and message:

  `kubectl edit configmap webapp-config`

  Change the `BACKGROUND_COLOR` to `"#e0e0e0"` and the `MESSAGE` to `"Updated configuration!"`.

- **Step 9:** Restart the Deployment to pick up the new configuration

  `kubectl rollout restart deployment webapp`

- **Step 10:** Access the application again to see the changes

Day 5: Working with local K8s options

Docker Images in kind

Building a custom Docker image

- Create a Dockerfile for your application.
- Build the image: `docker build -t your-image:tag .`

Loading the image into kind cluster

- Use the command: `kind load docker-image your-image:tag`
- This copies the image from your local Docker daemon into the kind cluster.

Limitations and workarounds for Docker-in-Docker scenarios

- kind runs Kubernetes inside Docker, which can complicate building images inside the cluster.
- Workaround: Use kaniko or buildkit for in-cluster builds.

Creating deployments with custom images

- Create a deployment YAML file (e.g., `deployment.yaml`) referencing your custom image:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: your-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: your-app
  template:
```

```
    metadata:
      labels:
        app: your-app
    spec:
      containers:
      - name: your-app
        image: your-image:tag
        imagePullPolicy: Never
```

- Apply the deployment: `kubectl apply -f deployment.yaml`

Working with Images in Minikube

Minikube provides several options for working with Docker images:

Using the Host Docker Daemon

- Configure your terminal to use Minikube's Docker daemon:
  `eval $(minikube docker-env)`
- Build your image. It will now be available to Minikube without additional steps.

Loading Images into Minikube

- If you've built the image using your host's Docker daemon:
  `minikube image load your-image:tag`
- This copies the image from your local Docker daemon into Minikube.

Creating Deployments with Custom Images

- Create a deployment YAML file (e.g., `deployment.yaml`) referencing your custom image:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: your-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: your-app
  template:
    metadata:
      labels:
        app: your-app
    spec:
      containers:
      - name: your-app
        image: your-image:tag
        imagePullPolicy: IfNotPresent
```

- Apply the deployment: `kubectl apply -f deployment.yaml`

Minikube-Specific Features

Built-in Docker Registry

Minikube includes a built-in Docker registry. To use it:

- Enable the registry addon:
  `minikube addons enable registry`

- Push your image to the Minikube registry:
  `docker push $(minikube ip):5000/your-image:tag`
- Update your deployment to use the registry image:
  `image: localhost:5000/your-image:tag`

Direct Image Building

- Minikube can build images directly using its Docker daemon:
  `minikube image build -t your-image:tag .`
- This builds the image inside Minikube, making it immediately available for use.

Monitoring and Troubleshooting

- Check if your pods are running:
  `kubectl get pods`
- If pods are not in the "Running" state, check the logs:
  `kubectl logs <pod-name>`
- For more detailed troubleshooting, use:
  `kubectl describe pod <pod-name>`
- To access the Minikube Docker daemon logs:
  `minikube logs`

Cleaning Up

To remove unused images and free up space:
`minikube image rm your-image:tag`

By following these steps, you can effectively work with custom Docker images in your Minikube cluster, allowing you to develop and test your Kubernetes deployments locally. Minikube offers more flexibility in terms of image handling compared to kind, making it a popular choice for local Kubernetes development.

Best Practices

1. Use meaningful tags for your images, preferably based on git commit hashes or semantic versioning.
2. When updating your application, build a new image with a new tag, then update your deployment to use the new image tag.
3. For production-like setups, consider using a private Docker registry. Minikube can be configured to pull from private registries.

Week 2: Service Mesh Concepts and Python

Day 1-2: Service Mesh

Fundamentals

Core concepts of service mesh

- A dedicated infrastructure layer for handling service-to-service communication.
- Provides features like service discovery, load balancing, encryption, observability, traceability, authentication, and authorization.

Problems service meshes solve

- Complexity in microservices communication
- Lack of observability in distributed systems
- Inconsistent security policies across services
- Difficulty in implementing resilience patterns (circuit breaking, retries)

Evolution of ingress

- From simple L7 load balancers to advanced API gateways
- Integration with service mesh for consistent policy enforcement

Service Mesh Architecture

A service mesh consists of two primary components: the data plane and the control plane.

Data Plane

The data plane is composed of a network of lightweight proxies, typically deployed as sidecars alongside each service instance. These proxies intercept and manage all network traffic to and from the service.

Example:
Let's consider a simple e-commerce application with three microservices: Product, Order, and Payment. In a service mesh, each instance of these services would have a sidecar proxy deployed alongside it:

```
Product Service + Sidecar Proxy
Order Service + Sidecar Proxy
Payment Service + Sidecar Proxy
```

When the Order service needs to communicate with the Payment service, the request goes through the following path:

1. Order service -> Order's sidecar proxy
2. Order's sidecar proxy -> Payment's sidecar proxy
3. Payment's sidecar proxy -> Payment service

This allows the mesh to control and observe all inter-service communication.

Control Plane

The control plane manages and configures the proxies to enforce policies, collect telemetry, and handle service discovery.

Example:
Using Istio as an example, the control plane consists of several components:

- Pilot: Handles service discovery and traffic management
- Citadel: Manages security and access policies
- Galley: Validates configuration and distributes it to other components

The control plane would configure the sidecar proxies to implement specific routing rules, such as:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: payment-route
spec:
  hosts:
  - payment
  http:
  - route:
    - destination:
        host: payment
        subset: v1
      weight: 90
    - destination:
        host: payment
        subset: v2
      weight: 10
```

This configuration would route 90% of traffic to version 1 of the Payment service and 10% to version 2, enabling canary deployments or A/B testing.

Here is an example using Linkerd's control plane. This is simpler and consists of fewer components compared to Istio. The main components are:

1. Destination: Handles service discovery and provides configuration to proxies
2. Identity: Manages security and certificate issuance for mTLS
3. Proxy Injector: Injects the Linkerd proxy as a sidecar

For traffic splitting in Linkerd, you would use either a TrafficSplit resource (if using the SMI extension) or an HTTPRoute resource (which is the preferred method going forward).
Here's an example using HTTPRoute:

```
apiVersion: policy.linkerd.io/v1beta2
kind: HTTPRoute
metadata:
  name: payment-route
  namespace: your-namespace
spec:
  parentRefs:
  - name: payment
    kind: Service
    group: core
    port: 8080
  rules:
  - backendRefs:
    - name: payment-v1
      port: 8080
      weight: 90
    - name: payment-v2
      port: 8080
      weight: 10
```

This configuration would achieve the same result as the Istio example, routing 90% of traffic to version 1 of the Payment service and 10% to version 2.

Key Features and Use Cases

Service Discovery and Load Balancing

Service meshes provide dynamic service discovery and intelligent load balancing.

Example:
In our e-commerce application, if we scale the Payment service to three instances, the service mesh would automatically discover these instances and distribute traffic among them. It could use advanced load balancing algorithms like least connections or weighted round-robin.

Traffic Management

Service meshes offer fine-grained control over traffic routing.

Example:
Implementing a canary release for the Product service:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: product-canary
spec:
  hosts:
```

```
  - product
  http:
  - match:
    - headers:
        user-agent:
          regex: ".*Chrome.*"
    route:
    - destination:
        host: product
        subset: v2
  - route:
    - destination:
        host: product
        subset: v1
```

This configuration routes all traffic from Chrome browsers to version 2 of the Product service, while all other traffic goes to version 1.

With Linkerd use HTTPRoute resource to define the traffic splitting:

```
apiVersion: policy.linkerd.io/v1beta2
kind: HTTPRoute
metadata:
  name: product-canary
  namespace: your-namespace
spec:
  parentRefs:
  - name: product
    kind: Service
    group: core
    port: 8080
  rules:
  - matches:
    - headers:
      - name: user-agent
        regex: ".*Chrome.*"
    backendRefs:
    - name: product-v2
      port: 8080
  - backendRefs:
    - name: product-v1
      port: 8080
```

This configuration routes all traffic from Chrome browsers to version 2 of the Product service, while all other traffic goes to version 1.
For more advanced canary deployments, you can use tools like Flagger with Linkerd. Flagger automates the process of creating new Kubernetes resources, watching metrics, and incrementally sending users to the new version.
Here's an example of how you might set up a Flagger canary for the Product service:

```
apiVersion: flagger.app/v1beta1
kind: Canary
metadata:
  name: product
  namespace: test
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: product
  service:
    port: 8080
```

```
  analysis:
    interval: 30s
    threshold: 5
    maxWeight: 50
    stepWeight: 5
    metrics:
    - name: success-rate
      threshold: 99
      interval: 1m
    - name: latency
      threshold: 500
      interval: 1m
```

This configuration sets up a canary deployment that gradually increases traffic to the new version while monitoring success rate and latency.

Observability

Service meshes provide detailed insights into service-to-service communication.

Example:
Using Istio with Prometheus and Grafana, you can visualize request volume, latency, and error rates for each service. You might see a dashboard showing:

- Request rate for Product service: 100 requests/second
- 95th percentile latency for Order service: 250ms
- Error rate for Payment service: 0.1%

This level of observability helps quickly identify and troubleshoot issues in the distributed system.

Linkerd provides similar observability capabilities to Istio, there are some differences in how it implements and presents these features.

1. Using the Linkerd CLI:
   `linkerd viz stat deploy -n your-namespace`

This command would show you a table with metrics for each deployment, including:

- Success rate
- Request per second (RPS)
- Latency (P50, P95, P99)

2. Using the Linkerd dashboard:

You can access it by running:
`linkerd viz dashboard`

In the dashboard, you would see:

- Request rate for Product service: 100 req/sec
- 95th percentile latency for Order service: 250ms
- Success rate for Payment service: 99.9% (which is equivalent to a 0.1% error rate)

Security

Service meshes can enforce mutual TLS (mTLS) encryption and fine-grained access policies.

Example:
Enforcing mTLS between all services:

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
```

```
  name: default
  namespace: istio-system
spec:
  mtls:
    mode: STRICT
```

This configuration ensures all inter-service communication is encrypted and authenticated.

Linkerd automatically enables mTLS for all meshed services by default, so you don't need to explicitly configure it. However, if you want to ensure that only mTLS traffic is allowed, you can use Linkerd's authorization policies.

Challenges and Best Practices

While service meshes offer numerous benefits, they also introduce complexity and potential performance overhead.

Performance Considerations

The additional network hops introduced by sidecar proxies can increase latency. It's crucial to benchmark your application with and without the service mesh to understand the performance impact.

Best Practice: Start with a subset of your services in the mesh and gradually expand as you become more comfortable with the technology and its impact on your system.

Complexity Management

Service meshes add another layer to your infrastructure, which can increase operational complexity.

Best Practice: Invest time in your training.

Monitoring and Troubleshooting

While service meshes provide extensive observability, the volume of data can be overwhelming.

Best Practice: Define clear Service Level Objectives (SLOs) and set up alerts based on these. Use distributed tracing to debug complex issues across services.

In conclusion, service meshes offer powerful capabilities for managing microservices architectures, but they require careful planning and implementation. By understanding the core concepts and following best practices, organizations can leverage service meshes to build more resilient, observable, and secure distributed systems.

Day 3-4: Python for Kubernetes

Python basics review (if needed)

Data Types

Python has several built-in data types:

- **Numeric**: int, float, complex
- **Sequence**: list, tuple, range
- **Text**: str
- **Mapping**: dict
- **Set**: set, frozenset
- **Boolean**: bool

Example:

Numeric Types

int (Integer)

```
age = 30
```

```
year = 2024
temperature = -5
x = 5
```

float (Floating-point)

```
pi = 3.14159
weight = 68.5
temperature = -2.8
y = 3.14
```

complex

```
z = 3 + 4j
w = complex(2, -3)
```

Sequence Types

list

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
mixed = [1, "two", 3.0, [4, 5]]
```

tuple

```
coordinates = (10, 20)
rgb = (255, 0, 128)
person = ("John", 30, "London")
```

range

```
numbers = range(5)  # 0, 1, 2, 3, 4
even_numbers = range(0, 10, 2)  # 0, 2, 4, 6, 8
```

Text Type

str (String)

```
name = "Alice"
message = 'Hello, World!'
multiline = """This is a
multiline string."""
```

Mapping Type

dict (Dictionary)

```
person = {"name": "Bob", "age": 25, "city": "Manchester"}
scores = {
    "Alice": 95,
    "Bob": 87,
    "Charlie": 92
}
```

Set Types

set

```
unique_numbers = {1, 2, 3, 4, 5}
fruits = {"apple", "banana", "cherry"}
```

frozenset

```
immutable_set = frozenset([1, 2, 3, 4, 5])
```

Boolean Type

bool

```
is_raining = True
has_licence = False
is_adult = age >= 18
```

Here are some examples of how these data types can be used in practice:

```
# Calculating area of a circle
radius = 5.0
area = pi * radius**2
print(f"The area of the circle is {area:.2f} square units")

# Working with lists
fruits.append("orange")
print(f"The second fruit is {fruits[1]}")

# Using a dictionary
print(f"{person['name']} is {person['age']} years old and lives in {person['city']}")

# Set operations
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}
print(f"Union: {a | b}")
print(f"Intersection: {a & b}")

# Boolean logic
if is_adult and not is_raining:
    print("Let's go for a walk!")
```

These examples demonstrate the basic usage of each data type. Remember that Python is dynamically typed, meaning you don't need to declare the type of a variable explicitly. The interpreter infers the type based on the value assigned to it.

Control Structures

If-else statements:

```
if x > 0:
    print("Positive")
elif x < 0:
    print("Negative")
else:
    print("Zero")
```

For loops:

```
for i in range(5):
    print(i)
```

While loops:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

Functions

```
def greet(name):
    return f"Hello, {name}!"

message = greet("Alice")
print(message)
```

Classes

```
class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        return f"{self.name} says Woof!"

my_dog = Dog("Buddy")
print(my_dog.bark())
```

Python Package Management

pip

pip is the standard package manager for Python. It allows you to install and manage additional packages that are not part of the Python standard library.

Installing a package:

`python3 -m pip install requests`

Upgrading a package:

`python3 -m pip install --upgrade requests`

Python Virtual Environments

Virtual environments are isolated Python environments that allow you to install packages for specific projects without affecting your system-wide Python installation.

Creating a virtual environment:

`python3 -m venv .venv`

Here's a breakdown of what each part of the command does:

- `python3`: This specifies that you are using Python 3 to execute the command. It ensures that the virtual environment is created using Python 3.
- `-m venv`: The `-m` flag tells Python to run a module as a script. In this case, it runs the `venv` module, which is included in the standard library from Python 3.3 onwards, for creating virtual environments.

- `.venv`: This is the name of the directory where the virtual environment will be created. The dot (`.`) at the beginning makes it a hidden directory on Unix-like systems, which is a common convention to keep your project directory tidy.

Activating a virtual environment:On Unix or MacOS:

`source .venv/bin/activate`

On Windows:

`.venv\Scripts\activate`

Installing packages in a virtual environment:

Once activated, you can use pip to install packages, and they will be isolated to this environment.
`pip install requests`

Deactivating a virtual environment:

`deactivate`

Creating a requirements file:

To share your project's dependencies, you can create a requirements.txt file:
`pip freeze > requirements.txt`

Installing from a requirements file:

`pip install -r requirements.txt`

Remember, it's a good practice to use virtual environments for each of your Python projects to avoid conflicts between package versions.
Explore pyenv

Kubernetes Python client library

- Installation: `pip install kubernetes`
  This will allow Authentication and configuration, Creating, reading, updating, and deleting Kubernetes resources

Simple Python scripts for Kubernetes interaction

Here is an example to;

- Listing pods in a namespace
- Creating and managing deployments
- Watching for changes in resources

Example script to list pods:

Create a virtual env

- `python3 -m venv .venv`
- `source .venv/bin/activate`
- `pip install kubernetes`
- Create `testscript.py`

```python
from kubernetes import client, config

config.load_kube_config()
v1 = client.CoreV1Api()

pods = v1.list_pod_for_all_namespaces(watch=False)
for pod in pods.items:
    print(f"{pod.metadata.namespace}\t{pod.metadata.name}")
```

- `python3 testscript.py`

  If running minikube the output may look like this

```
default debug-env
default webapp-6988595754-qnkqp
default webapp-6d989cd746-8wgzs
default webapp-cf544bc7c-24zpb
kube-system coredns-7db6d8ff4d-t46mv
kube-system etcd-minikube
kube-system kube-apiserver-minikube
kube-system kube-controller-manager-minikube
kube-system kube-proxy-jkgd5
kube-system kube-scheduler-minikube
kube-system storage-provisioner
```

  You now have the basics to interact with a kubernetes cluster via python.

  Link:

  GitHub - kubernetes-client/python: Official Python client library for kubernetes

Day 5: Helm Basics

Helm's Purpose and Architecture

Helm is a package manager for Kubernetes that simplifies the deployment and management of applications. It allows you to define, install, and upgrade even the most complex Kubernetes applications.

Helm Essentials in Under an Hour < a good tutorial

Key Components:

1. **Helm Client**: The command-line tool used to create, package, and manage charts.
2. **Charts**: Packages of pre-configured Kubernetes resources.
3. **Releases**: Instances of a chart running in a Kubernetes cluster.

Creating and Structure of a Helm Chart

Let's create a chart and examine its structure:

You will have needed to install helm

`helm create mychart`

`cd mychart`

The chart structure:

```
mychart/
  Chart.yaml          # Metadata about the chart
  values.yaml         # Default configuration values
  charts/             # Directory for chart dependencies
  templates/          # Directory for template files
    deployment.yaml
    service.yaml
    ingress.yaml
    _helpers.tpl      # Template helpers
  .helmignore         # Patterns to ignore when packaging
```

Chart.yaml Example:

```
apiVersion: v2
```

```
name: mychart
description: A Helm chart for Kubernetes
type: application
version: 0.1.0
appVersion: "1.16.0"
```

values.yaml Example:

```
replicaCount: 1

image:
  repository: nginx
  pullPolicy: IfNotPresent
  tag: ""

service:
  type: ClusterIP
  port: 80

ingress:
  enabled: false
```

Deploying Applications with Helm

To install a chart:

`helm install myrelease ./mychart`

To customize values during installation:

`helm install myrelease ./mychart --set service.type=LoadBalancer`

Or using a custom values file:

`helm install myrelease ./mychart -f custom-values.yaml`

Advanced Helm Concepts

Hooks

Hooks allow you to intervene at certain points in a release's lifecycle. Here's an example of a pre-install hook:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: {{ .Release.Name }}-pre-install-job
  annotations:
    "helm.sh/hook": pre-install
spec:
  template:
    spec:
      containers:
      - name: pre-install-job
        image: busybox
        command: ['sh', '-c', 'echo Pre-install job running']
      restartPolicy: Never
```

Dependencies

You can define dependencies in the `Chart.yaml` file:

```
dependencies:
  - name: apache
    version: 1.2.3
```

```
    repository: https://charts.bitnami.com/bitnami
```

Then, update dependencies:

```
helm dependency update
```

Templating

Helm uses Go templates. Here's an example of a template using conditionals and loops:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-deployment
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: {{ .Chart.Name }}
  template:
    metadata:
      labels:
        app: {{ .Chart.Name }}
    spec:
      containers:
        - name: {{ .Chart.Name }}
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          ports:
            - containerPort: 80
          {{- if .Values.env }}
          env:
            {{- range $key, $value := .Values.env }}
            - name: {{ $key }}
              value: {{ $value | quote }}
            {{- end }}
          {{- end }}
```

Creating Helm Charts with Python Templates

While Helm natively uses Go templates, you can use Python to generate Helm charts dynamically.

Using Jinja2 for Templating

Here's an example of using Jinja2 to generate a Kubernetes manifest:

```
from jinja2 import Template

template = Template("""
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ name }}-deployment
spec:
  replicas: {{ replicas }}
  selector:
    matchLabels:
      app: {{ name }}
  template:
    metadata:
      labels:
```

```
        app: {{ name }}
    spec:
      containers:
        - name: {{ name }}
          image: {{ image }}
          ports:
            - containerPort: {{ port }}
""")

rendered = template.render(
    name="myapp",
    replicas=3,
    image="nginx:latest",
    port=80
)

print(rendered)
```

Generating Kubernetes Manifests Dynamically

You can use Python to read configuration from various sources and generate Helm charts:

```
import yaml
from jinja2 import Template

def generate_chart(config):
    # Load templates
    deployment_template = Template(open('templates/deployment.yaml').read())
    service_template = Template(open('templates/service.yaml').read())

    # Render templates
    deployment = deployment_template.render(config)
    service = service_template.render(config)

    # Combine rendered templates
    chart = f"{deployment}\n---\n{service}"

    return chart

# Read configuration
with open('app_config.yaml', 'r') as f:
    config = yaml.safe_load(f)

# Generate chart
chart = generate_chart(config)

# Write chart to file
with open('generated_chart.yaml', 'w') as f:
    f.write(chart)
```

Integrating with CI/CD Pipelines

You can incorporate this Python-based chart generation into your CI/CD pipeline:

```
# Example GitLab CI job
generate_helm_chart:
  stage: build
  script:
    - pip install pyyaml jinja2
    - python generate_chart.py
```

```
  artifacts:
    paths:
      - generated_chart.yaml
```

This job would generate the Helm chart as part of your CI/CD process, allowing for dynamic chart creation based on your application's needs.These examples demonstrate how to create more complex Helm charts, use advanced features, and even integrate Python for dynamic chart generation.

Week 3: Istio Deep Dive

Day 1: Istio Basics

Installing Istio on your Kubernetes cluster

Download Istio

Getting Started

Mac can use brew `brew install istionctl`

Install Istio

istio provides a demo for testing and learning:

- It installs more components than the default profile, including:
  - Istiod (the Istio control plane)
  - Ingress gateway
  - Egress gateway
- It enables a set of features that are suitable for demonstrating Istio's capabilities.
- It has higher resource requirements than the minimal or default profiles.
- It's not recommended for production use due to its expanded feature set and resource usage.

```
istioctl install --set profile=demo -y
```

Enable automatic sidecar injection

```
kubectl label namespace default istio-injection=enabled
```

Istio's architecture and core components

Control Plane

istiod: Combines Pilot, Citadel, and Galley into a single binary

Pilot

Pilot is a crucial module within Istiod that focuses on service discovery and traffic management. It is responsible for:

- **Service Discovery**: Registers services and manages their information, such as versions, IP addresses, and ports.
- **Traffic Management**: Directs traffic to different service versions or instances based on defined rules.
- **Routing and Load Balancing**: Routes traffic according to rules and balances load across services.

Pilot interacts with the data plane by configuring service proxies (like Envoy) to manage ingress and egress traffic effectively.

Citadel

Citadel is another component integrated into Istiod, primarily handling security aspects. It manages:

- **Certificate Management**: Provides certificate-based authentication and authorization.
- **Security Policies**: Enforces security policies based on service identity.

Galley

Galley was responsible for configuration management in Istio. It handled:

- **Configuration Verification and Distribution**: Ensured the validity of configuration rules and distributed them to other Istio components.
- **Configuration Storage**: Maintained properties and configuration information for Istio components.

Data Plane

Envoy proxy: Sidecar container deployed alongside each service

Addons

- **Prometheus**: An open-source system for metrics collection and monitoring, storing data as time series with flexible querying capabilities.
- **Grafana**: A platform for metrics visualization, providing a variety of visual representations to analyse time-series data from sources like Prometheus.
- **Jaeger or Zipkin**: Tools for distributed tracing that help monitor and troubleshoot microservices by collecting and analysing trace data.
- **Kiali**: A service mesh observability tool that visualizes the structure and health of an Istio service mesh, aiding in monitoring and troubleshooting.

Day 2: Istio Traffic Management

Exploring Istio's traffic management features

Virtual Services: Define routing rules for traffic

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
    name: reviews-route
spec:
    hosts:
    - reviews
       http:
    - route:
    - destination:
        host: reviews
        subset: v1
        weight: 75
    - destination:
        host: reviews
        subset: v2
    weight: 25
```

This configuration defines a `VirtualService` for managing HTTP traffic routing to different versions (subsets) of the `reviews` service. It splits traffic between two subsets, `v1` and `v2`, with 75% going to `v1` and 25% going to `v2`.

Destination Rules: Define policies that apply after routing

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
    name: reviews-destination
spec:
    host: reviews
    subsets:
    - name: v1
        labels:
        version: v1
    - name: v2
        labels:
```

```
        version: v2
```

This configuration defines a `DestinationRule` for the `reviews` service, specifying two subsets, `v1` and `v2`. Each subset is identified by labels that correspond to versions of the service. These subsets are referenced in the Istio configuration of the `VirtualService`, to route traffic to specific versions of a service. This is useful for scenarios like canary deployments or A/B testing.

Gateways: Manage inbound and outbound traffic for the mesh

Implementing canary deployments and A/B testing

1. Use VirtualService (as above) to split traffic between versions
2. Gradually adjust weights to increase traffic to new version
3. Monitor metrics to ensure new version performs as expected

Istio's load balancing and circuit breaking capabilities

Load Balancing: Configure in DestinationRule

```
spec:
  trafficPolicy:
    loadBalancer:
        simple: ROUND_ROBIN
```

Circuit Breaking: Define in DestinationRule

```
spec:
  trafficPolicy:
    outlierDetection:
    consecutiveErrors: 5
    interval: 5s
    baseEjectionTime: 30s
```

Day 3: Istio Security and Observability

Istio's security features

mTLS (Mutual TLS)

- Enable cluster-wide: `kubectl apply -f istio-1.x.x/samples/security/strict-mtls.yaml`
- Verify: `istioctl x authz check <pod-name>`

Authorization Policies

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-read
spec:
  action: ALLOW
  rules:
  - to:
    - operation:
        methods: ["GET"]
```

Exploring Istio's observability stack

Prometheus

- Access dashboard: `istioctl dashboard prometheus`
- Query metrics using PromQL

Grafana

- Access dashboard: `istioctl dashboard grafana`
- Explore pre-configured Istio dashboards

Kiali

- Access dashboard: `istioctl dashboard kiali`
- Visualize service mesh topology and health

Jaeger/Zipkin

- Access Jaeger UI: `istioctl dashboard jaeger`
- Analyze distributed traces

Day 4-5: Deploying a Sample Application with Istio

Objective

Deploy a simple web application with Istio sidecar injection and implement basic traffic routing.

Prerequisites

- Kubernetes cluster set up
- Istio installed with demo profile
- `kubectl` and `istioctl` configured

Enable Istio Sidecar Injection

First, let's enable Istio sidecar injection for the default namespace:
`kubectl label namespace default istio-injection=enabled`

(This can be verified with `kubectl get namespace default --show-labels`)

The command is used to enable automatic Istio sidecar injection for the default namespace in a Kubernetes cluster.

Key points about this command:

1. Namespace-level control: By labeling a namespace, you're enabling Istio sidecar injection for all pods created in that namespace, unless overridden at the pod level.
2. Automatic injection: When a namespace has this label, the Istio sidecar (Envoy proxy) will be automatically injected into all new pods deployed in that namespace.
3. Existing workloads: This label only affects new pods. Existing workloads will need to be redeployed to get the sidecar injected.
4. Override option: Even with this namespace-level setting, individual pods can opt out of injection using the `sidecar.istio.io/inject: "false"` annotation.
5. Verification: After applying this label, you can verify it worked by deploying a new pod in the namespace and checking for the presence of the `istio-proxy` container.
6. Reversibility: You can disable injection for the namespace by changing the label value to `disabled` or removing the label entirely.

Deploy a Sample Application

Create a file named `sample-app.yaml` with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
        version: v1
    spec:
      containers:
      - name: myapp
        image: nginx:1.14.2
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: myapp
spec:
  selector:
    app: myapp
  ports:
  - port: 80
    targetPort: 80
```

or to apply in one

```
cat <<EOF | kubectl -f -
  yaml
EOF
```

Deploy the application:

```
kubectl apply -f sample-app.yaml
```

Verify the deployment:

```
kubectl get pods
```

You should see two containers per pod (app + istio-proxy), indicating successful sidecar injection.

eg `k describe pod/<pod name>`

You will see something like

```
Events:
  Type    Reason     Age    From               Message
  ----    ------     ----   ----               -------
```

```
  Normal  Scheduled  5m     default-scheduler  Successfully assigned default/myapp-7d4cbc4c78-mhdmd to minikube
  Normal  Pulled     5m     kubelet            Container image "docker.io/istio/proxyv2:1.23.2" already present on machine
  Normal  Created    5m     kubelet            Created container istio-init
  Normal  Started    5m     kubelet            Started container istio-init
  Normal  Pulling    5m     kubelet            Pulling image "nginx:1.14.2"
  Normal  Pulled     4m54s  kubelet            Successfully pulled image "nginx:1.14.2" in 885ms (5.074s including waiting). Image size: 102757429 bytes.
  Normal  Created    4m54s  kubelet            Created container myapp
  Normal  Started    4m54s  kubelet            Started container myapp
  Normal  Pulled     4m54s  kubelet            Container image "docker.io/istio/proxyv2:1.23.2" already present on machine
  Normal  Created    4m54s  kubelet            Created container istio-proxy
  Normal  Started    4m54s  kubelet            Started container istio-proxy
```

Create a Virtual Service

Create a file named `virtual-service.yaml`:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: myapp-route
spec:
  hosts:
  - myapp
  http:
  - route:
    - destination:
        host: myapp
        subset: v1
```

Apply the Virtual Service:

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

View with
`kubectl get svc`

A VirtualService in Istio is a custom resource definition (CRD) that allows you to configure how requests are routed to services within the Istio service mesh. It acts as a flexible and powerful tool for traffic management, enabling you to define routing rules that dictate how traffic should be directed to different service versions or destinations based on specified criteria.

Key Features of VirtualService

- Traffic Routing.
- Decoupling Requests and Destinations.
- Advanced Traffic Management.
- Integration with Other Istio Resources.
- Internal and External Traffic Control.

Create a Destination Rule

Create a file named `destination-rule.yaml`:

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: myapp-destination
spec:
  host: myapp
  subsets:
  - name: v1
    labels:
      version: v1
```

Apply the Destination Rule:

```
kubectl apply -f destination-rule.yaml
```

verify with

`k get destinationrules`

Test the Routing

To test the routing, we'll need to access the application. For simplicity, let's use port-forwarding:

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

Now, in another terminal, you can access the application:

```
curl http://localhost:8080
```

You should see the nginx welcome page.

Implement Canary Deployment

Let's update our application to version 2. Create a file named `sample-app-v2.yaml`:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myapp
      version: v2
  template:
    metadata:
      labels:
        app: myapp
        version: v2
    spec:
      containers:
      - name: myapp
        image: nginx:1.16.0
        ports:
        - containerPort: 80
```

Deploy version 2:

```
kubectl apply -f sample-app-v2.yaml
```

Update the `virtual-service.yaml` to split traffic:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: myapp-route
spec:
  hosts:
  - myapp
  http:
  - route:
    - destination:
        host: myapp
        subset: v1
      weight: 75
    - destination:
        host: myapp
        subset: v2
      weight: 25
```

Update the `destination-rule.yaml`:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: myapp-destination
spec:
  host: myapp
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

Apply the updated configurations:

```
kubectl apply -f virtual-service.yaml
kubectl apply -f destination-rule.yaml
```

Now, when you access the application, 75% of the traffic will go to v1 and 25% to v2.

Testing can be run as `for i in {1..200}; do echo $(curl -s http://localhost:8080 | grep "version"); sleep .5; done`

observability

Apply Prometheus `kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.23/samples/addons/prometheus.yaml`

Apply kiali `kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.23/samples/addons/kiali.yaml`

Access dashboard `istioctl dashboard kiali`

Conclusion

In this lesson, we've deployed a sample application with Istio, implemented basic traffic routing, and set up a canary deployment. This demonstrates some of Istio's core traffic management capabilities. In a real-world scenario, you would monitor the performance of both versions and gradually adjust the traffic split until you're confident in the new version's performance.Remember to clean up your resources after the lesson:

```
kubectl delete -f sample-app.yaml
kubectl delete -f sample-app-v2.yaml
kubectl delete -f virtual-service.yaml
kubectl delete -f destination-rule.yaml
```

This lesson provides a practical introduction to Istio's traffic management features. For more advanced scenarios, you could explore features like fault injection, circuit breaking, and more complex routing rules.

If using minikube a simple `minikube delete` will remove all existance of the cluster

Week 4: Linkerd and Practical Applications

Day 1: Linkerd Basics

Installing Linkerd on your Kubernetes cluster

Install CLI

Installing Linkerd

Again Mac can use brew

`curl --proto '=https' --tlsv1.2 -sSfL https://run.linkerd.io/install | sh`

`export PATH=$PATH:$HOME/.linkerd2/bin`

`linkerd version`

Alternatively, you can download the binary directly from the Linkerd releases page.

Install Linkerd on Your Minikube Cluster

`linkerd install --crds | kubectl apply -f -`

`linkerd install --set proxyInit.runAsRoot=true | kubectl apply -f -`

Validate cluster

`linkerd check --pre`

Install Linkerd

`linkerd install | kubectl apply -f -`

Install viz

`linkerd viz install | kubectl apply -f -`

`linkerd viz check`

`linkerd viz dashboard`

Linkerd's architecture and core components

Control Plane

- controller: Manages and configures proxy instances
- destination: Service discovery and load balancing
- identity: Certificate management for mTLS

Data Plane

linkerd-proxy: Ultra-lightweight proxy (written in Rust)

Add-ons

- Grafana: Metrics visualization
- Prometheus: Metrics collection

Linkerd Features

Traffic management capabilities

Traffic Split:

```
apiVersion: split.smi-spec.io/v1alpha1
kind: TrafficSplit
metadata:
  name: web-split
spec:
  service: web-svc
  backends:
  - service: web-v1
      weight: 500m
  - service: web-v2
      weight: 500m
```

Retries and Timeouts: Configured via annotations

Linkerd's observability and security features

- Automatic mTLS:
  Enabled by default for all meshed servicesb.
- Metrics:
  Access via CLI or Grafana dashboards

```
linkerd viz stat deployment
```

- Live Traffic View:

```
linkerd viz top
```

- Traffic Inspection:

```
linkerd tap deployment/your-deployment
```

Day 2-4: Hands-on Exercise

Deploying and Managing emojivoto with Linkerd

Sheet here Linkerd in a Minikube environment

Deploy the emojivoto sample application

```
curl -sL https://run.linkerd.io/emojivoto.yml | kubectl apply -f -
```

This command downloads the emojivoto application manifest and applies it to your Kubernetes cluster. Verify the deployment:

```
kubectl get pods -n emojivoto
```

Inject Linkerd into the application

```
kubectl get -n emojivoto deploy -o yaml | linkerd inject - | kubectl apply -f -
```

This command retrieves all deployments in the emojivoto namespace, injects the Linkerd sidecar, and reapplies the configuration. Verify the injection:

```
kubectl get pods -n emojivoto
```

You should now see two containers per pod (the application container and the Linkerd proxy).

Observe traffic

Install smi

`helm repo add linkerd-smi https://linkerd.github.io/linkerd-smi`

`helm install smi linkerd-smi/linkerd-smi`

The Service Mesh Interface (SMI) is a standard specification for service meshes on Kubernetes, providing a set of common APIs to enable interoperability between different service mesh implementations, allowing users to manage microservices communication without being tied to a specific provider.

`linkerd viz stat -n emojivoto deploy`

This command shows real-time metrics for your deployments, including success rate, requests per second, and latency.

Visualize the service mesh

```
linkerd viz dashboard
```

This opens the Linkerd dashboard in your default browser. Explore the various sections to see detailed metrics, topology, and live calls.

In a terminal

create port fowarding

`kubectl -n emojivoto port-forward svc/web-svc 8080:80`

Create traffic

`for i in {1..20000}; do curl -s http://localhost:8080 ; done`

Implement a traffic split for canary deployment

First, let's create a new version of the voting service:

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: voting-v2
  namespace: emojivoto
spec:
  replicas: 1
  selector:
    matchLabels:
      app: voting-svc
      version: v2
  template:
    metadata:
      labels:
        app: voting-svc
        version: v2
    spec:
```

```
      containers:
      - name: voting-svc
        image: buoyantio/emojivoto-voting-svc:v11
        env:
        - name: GRPC_PORT
          value: "8080"
        ports:
        - containerPort: 8080
EOF
```

or (`kubectl get deployments web -n emojivoto -o yaml > web-deployment.yaml ; sed -i 's/name: web/name: web-v2/' web-deployment.yaml sed -i 's/image: emojivoto-web:v1/image: emojivoto-web:v2/' web-deployment.yaml ; kubectl apply -f web-deployment.yaml ;rm web-deployment.yaml)`

Now, create a TrafficSplit to gradually shift traffic:

```
cat <<EOF | kubectl apply -f -
apiVersion: split.smi-spec.io/v1alpha2
kind: TrafficSplit
metadata:
  name: voting-split
  namespace: emojivoto
spec:
  service: voting-svc
  backends:
  - service: voting
    weight: 900
  - service: voting-v2
    weight: 100
EOF
```

This configuration sends 90% of traffic to the original version and 10% to the new version.

run `kubectl get -n emojivoto deploy -o yaml | linkerd inject - | kubectl apply -f -`

Observe the traffic split

```
linkerd viz stat -n emojivoto deploy voting voting-v2
```

You should see traffic being split between the two versions according to the weights specified in the TrafficSplit resource.

Gradually increase traffic to the new version

As you gain confidence in the new version, you can update the TrafficSplit to increase traffic to v2:

```
cat <<EOF | kubectl apply -f -
apiVersion: split.smi-spec.io/v1alpha2
kind: TrafficSplit
metadata:
  name: voting-split
  namespace: emojivoto
spec:
  service: voting-svc
  backends:
  - service: voting
    weight: 500m
  - service: voting-v2
    weight: 500m
EOF
```

This updates the split to 50/50 between the two versions.

Monitor the canary deployment

Use the Linkerd dashboard or CLI to monitor the performance of both versions:

```
linkerd -n emojivoto stat deploy voting voting-v2
```

Keep an eye on success rates, latency, and request volumes to ensure the new version is performing as expected.

(In dashboard services → voting-svc will show the split and successes)

Conclusion

In this hands-on exercise, you've:

1. Deployed the emojivoto sample application
2. Injected Linkerd into the application
3. Observed traffic using Linkerd's CLI and dashboard
4. Implemented a canary deployment using TrafficSplit
5. Monitored the performance of both versions during the canary rollout

This exercise demonstrates Linkerd's key features for traffic management and observability, providing a practical introduction to service mesh concepts and canary deployments.

Day 5: Service Mesh Comparison

Comparing Istio, Linkerd, and other service mesh solutions

Istio

- Pros: Feature-rich, powerful traffic management
- Cons: Complex, resource-intensive

Linkerd

- Pros: Lightweight, simple, fast
- Cons: Fewer advanced features

Consul Connect

- Pros: Integrates well with HashiCorp ecosystem
- Cons: Less mature as a full service mesh

NGINX Service Mesh

- Pros: Builds on familiar NGINX technology
- Cons: Relatively new, smaller community

When to choose one service mesh over another

- Choose Istio for complex, feature-rich requirements
- Choose Linkerd for simplicity and performance
- Consider Consul Connect if already using HashiCorp tools
- NGINX Service Mesh if familiar with NGINX and need basic mesh features

Week 5: Practical Project

Designing and implementing a microservices application

1. Create 3-4 simple microservices (e.g., frontend, backend, database)
2. Containerize each service with Docker
3. Create Kubernetes manifests for each service

Deploying the application using Helm

1. Create a Helm chart for the entire application
2. Use subchart for each microservice
3. Define configurable values in values.yaml

Implementing service mesh features

1. Choose either Istio or Linkerd based on your preference
2. Implement traffic routing between service versions
3. Set up mTLS between services
4. Configure observability (metrics, tracing)

Creating Python scripts for automation

1. Script to deploy/update the Helm release
2. Script to check service health and metrics
3. Script to perform canary deployments

This comprehensive deep dive covers the entire 4-week training plan, providing a solid foundation in Kubernetes, service mesh technologies, and related tools. Remember to practice hands-on with each concept and refer to official documentation for the most up-to-date information.

Additional Resources and Best Practices

- Throughout the training, refer to official documentation for each technology
- Join community forums or discussion groups for each technology
- Consider working on a personal project that incorporates all these technologies
- Explore real-world use cases and examples
- Practice hands-on exercises daily

Tips for Successful Service Mesh Adoption

1. Start your service mesh journey early to allow your knowledge to grow organically as your microservices landscape evolves.
2. Avoid common design and implementation pitfalls by thoroughly understanding each technology.
3. Leverage your service mesh as the mission control of your multi-cloud microservices landscape.
4. Consider starting with a sample project to evaluate which service mesh solution you prefer before standardizing across all services.
5. Use service mesh as a 'bridge' while decomposing monolithic applications into microservices.
6. Implement service mesh incrementally, starting with the components you need most.

By following this training plan, you'll gain a solid foundation in service mesh concepts, Kubernetes, Helm, and Python, with practical experience in both Istio and Linkerd. Remember to adapt the pace and depth of each topic based on your prior knowledge and learning speed.

Tools

k9s : K9s: a Kubernetes Cluster Management Tool

jq : jq

kubectl : Install Tools

docker: Install