Week 5 - Deep Dive Kubernetes

Introduction to Kubernetes:

- Defining Kubernetes and its Role in Container Orchestration
- Exploring the Features and History of Kubernetes
- Kubernetes Architecture:

In-Depth Analysis of Kubernetes Architecture

- Understanding Node Components, Manifest File Components, and Service Components
- Overview of Node and Pod Fundamentals
- Role of Master Node and Components of the Control Plane
- Installing and Configuring kubectl and minikube

Kubernetes Basics:

- Kubernetes Commands: Navigating and Interacting with Kubernetes
- Creation and Deletion of Pods
- Managing Kubernetes YAML Configurations
- Higher-Level Kubernetes Objects and Object Management
- Labels and Selectors in Kubernetes
- Kubernetes Networking, Services, and NodePort
- Understanding Namespaces in Kubernetes
- Multi Container Pod Setup
- Pods Design pattern - Sidecar, Ambassador

Working with Applications in Kubernetes:

- Installing Kubernetes on AWS
- Deploying Microservices Applications to Kubernetes Cluster
- ConfigMap and Secret Usage in Kubernetes
- Exploring Volumes in Kubernetes
- Persistent Volume and LivenessProbe in Kubernetes
- Replication, Auto-Healing, and Deployment in Kubernetes

Advanced Kubernetes Topics:

- Helm And Istio Service mesh
- Role-Based Access Control (RBAC) and Service Accounts
- Helm and Istio Integration in Kubernetes
- Kubernetes Interview Questions
- Differences Between Monolithic and Microservices Architecture
- HPA, Ingress, Taint and toleration,

Definition of container orchestration

Container orchestration is the process of automating the deployment, scaling, management, and networking of containers. Containers are a lightweight and portable way to package and run applications and their dependencies. Container orchestration tools provide a layer of abstraction that simplifies the deployment and management of containerized applications.

Container orchestration is necessary because containerized applications can be complex and difficult to manage at scale. Container orchestration tools provide a framework for managing the lifecycle of containers, from deployment to scaling to termination. By automating many of these tasks, container orchestration tools make it easier to manage containerized applications and ensure they are highly available, scalable, and reliable.

Some examples of container orchestration tools include:

1) Kubernetes: Kubernetes is an open-source container orchestration tool developed by Google. It has become the de facto standard for container orchestration due to its popularity, flexibility, and extensive feature set. Kubernetes automates many of the tasks associated with deploying, scaling, and managing containerized applications, including load balancing, network routing, and resource allocation.

2) Docker Swarm: Docker Swarm is a container orchestration tool built into the Docker platform. It provides a simple way to manage and scale Docker containers, and integrates with other Docker tools such as Docker Compose and Docker Registry.

3) Apache Mesos: Apache Mesos is an open-source cluster manager that can be used for container orchestration. It provides a distributed systems kernel that abstracts CPU, memory, storage, and other resources from physical or virtual machines, and makes them available to applications running in containers.

4) Amazon ECS: Amazon Elastic Container Service (ECS) is a fully-managed container orchestration service provided by Amazon Web Services (AWS). It supports Docker containers and integrates with other AWS services such as Amazon Elastic Load Balancing (ELB) and Amazon Elastic File System (EFS).

Overall, container orchestration is essential for managing the complexity of containerized applications at scale. By automating many of the tasks associated with deploying, scaling, and managing containers, container orchestration tools make it easier to develop and maintain modern applications.

Importance of container orchestration

here are some real-world examples that demonstrate the importance of container orchestration:

1) Scalability: Container orchestration makes it easy to scale applications horizontally and vertically. For example, a website that experiences a sudden increase in traffic can use

container orchestration to add more instances of its application to handle the load. This ensures that the website remains responsive and available to users.

2) High availability: Container orchestration helps ensure that applications are highly available by automatically managing their deployment and replication. For example, if a container crashes or fails, the container orchestration tool can automatically restart it or deploy a new instance to ensure that the application remains available.

3) Resource utilization: Container orchestration tools can automatically manage resource allocation and utilization for containerized applications. For example, if an application requires more resources to handle increased traffic, the container orchestration tool can allocate more resources to it. Conversely, if an application is not using its allocated resources, the container orchestration tool can free up those resources for other applications.

4) Easy deployment: Container orchestration tools make it easy to deploy containerized applications across different environments. For example, a developer can use a container orchestration tool to deploy their application to a local development environment, a staging environment, and a production environment with minimal effort.

5) Infrastructure management: Container orchestration tools can manage the underlying infrastructure required for running containerized applications. For example, a container orchestration tool can provision and manage virtual machines or cloud resources required for running containers, without the need for manual intervention.

Some real-world examples of container orchestration in action include:

1) Spotify: Spotify uses Kubernetes to manage the deployment and scaling of its microservices-based architecture. By using container orchestration, Spotify can easily manage its complex infrastructure and ensure that its applications are highly available and scalable.

2) Airbnb: Airbnb uses Docker and Kubernetes to manage its infrastructure and deploy its applications. By using container orchestration, Airbnb can quickly deploy updates and new features to its applications, while ensuring that they are highly available and scalable.

3) Pinterest: Pinterest uses Kubernetes to manage its infrastructure and deploy its applications. By using container orchestration, Pinterest can easily manage its complex infrastructure and ensure that its applications are highly available and scalable.

4) The New York Times: The New York Times uses Docker and Kubernetes to manage its infrastructure and deploy its applications. By using container orchestration, The New York Times can easily manage its complex infrastructure and ensure that its applications are highly available and scalable.

Overall, container orchestration is a critical component of modern application development and infrastructure management. It helps ensure that applications are highly available,

scalable, and efficient while minimizing the need for manual intervention and infrastructure management.

Kubernetes Architecture

Kubernetes is built around a distributed architecture that consists of several components working together to manage and orchestrate containers. Here is an overview of the Kubernetes architecture:

1) **Master Node**: The master node is the central control plane that manages the overall state of the Kubernetes cluster. It runs several components, including the API server, etcd, scheduler, and controller manager. These components work together to manage the configuration, scheduling, and deployment of containerized applications.

2) **API Server**: The API server is the primary interface for interacting with the Kubernetes cluster. It provides a RESTful API that can be used to manage containers, pods, services, and other resources.

3) **etcd**: etcd is a distributed key-value store that is used to store the configuration and state of the Kubernetes cluster. It is a critical component of the Kubernetes architecture, and is used by several other components to manage the state of the cluster.

4) **Scheduler**: The scheduler is responsible for assigning pods to nodes based on resource availability and other constraints. It takes into account factors such as CPU and memory usage, node capacity, and affinity and anti-affinity rules to ensure that pods are distributed optimally across the cluster.

5) **Controller Manager**: The controller manager is responsible for managing the various controllers that are used to manage the state of the Kubernetes cluster. These controllers include the ReplicaSet controller, which ensures that the desired number of pods are running at all times, and the Deployment controller, which manages the rolling update of containerized applications.

6) **Worker Node**: The worker node is where the containers are actually deployed and run. Each worker node runs a Kubernetes agent called the kubelet, which is responsible for managing the pods and containers on that node. The kubelet communicates with the API server to receive instructions and updates about the state of the cluster.

7) **Pod**: A pod is the smallest deployable unit in Kubernetes. It consists of one or more containers, along with shared resources such as network and storage volumes. Pods are scheduled and managed by Kubernetes, and can be used to run microservices or other types of containerized applications.

8) **Service**: A service is an abstraction that is used to expose pods to other pods or to external users. Services provide a stable IP address and DNS name that can be used to access a group of pods, even if the pods are moved or replaced.

Overall, the Kubernetes architecture is designed to be highly scalable, fault-tolerant, and flexible. The use of a distributed control plane and a shared data store (etcd) helps to ensure that the state of the cluster is consistent across all nodes, even in the event of node failures or other disruptions.

Kubernetes

- Kubernetes Overview - https://kubernetes.io/docs/concepts/overview/
- Kubernetes History - https://www.ibm.com/blog/kubernetes-history/
- Container Orchestration Overview - https://kubernetes.io/docs/concepts/overview/
- What is Microservices - https://aws.amazon.com/microservices/
- What is a Pod - https://kubernetes.io/docs/concepts/workloads/pods/
- Deployment Strategies - https://spacelift.io/blog/kubernetes-deployment-strategies

What is Monolithic Architecture?

A monolithic architecture is a singular, large computing network with one code base that couples all of the business concerns together.

What are Microservices

Microservices - also known as microservice architecture - is an architectural style that structures an application as a collection of services that are:
- Independently deployable
- Loosely coupled

Ref: https://microservices.io/

Need for Microservices:

- If we deploy a monolithic application, we need to deploy the entire application even if we make a small change in the code.
- If one part of the application is not working, the entire application will not work.
- So In this case, We can leverage the Microservices architecture to solve the above problems.

Differences Between Monolithic and Microservices Architecture

Monolithic Architecture:

- **Single Codebase**: All components are part of a single codebase and deployed together.
- **Tightly Coupled**: Components are tightly integrated, making changes and scaling difficult.
- **Centralized Data Storage**: Often uses a single database.

- **Deployment**: Changes require redeploying the entire application.
- **Scalability**: Hard to scale specific components independently.

Microservices Architecture:

- **Multiple Codebases**: Each service has its own codebase and can be deployed independently.
- **Loosely Coupled**: Services are loosely integrated via APIs, allowing for easier updates and maintenance.
- **Decentralized Data Storage**: Each service can have its own database.
- **Deployment**: Services can be deployed, scaled, and updated independently.
- **Scalability**: Easier to scale individual services based on demand.

Advantages of Microservices:

- **Agility**: Faster development and deployment cycles.
- **Scalability**: Independent scaling of services.
- **Resilience**: Failure in one service doesn't necessarily affect others.
- **Technology Diversity**: Different services can use different technologies.

Challenges of Microservices:

- **Complexity**: More complex to manage and deploy.
- **Inter-service Communication**: Requires robust communication mechanisms (e.g., service mesh).
- **Data Consistency**: Managing data consistency across services can be challenging.
-

Container Orchestration

Container orchestration automatically provisions, deploys, scales, and manages containerised applications without worrying about the underlying infrastructure

What is K8S?

- Kubernetes is an open-source container orchestration platform that automates many of the manual processes involved in deploying, managing, and scaling containerized applications.
- Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation.

Need for K8S

- If we have a large number of containers, it is difficult to manage them manually.
- Let's say we have 100 microservices and we need to deploy them in a server.
- We need to manage the 100 microservices in the server.
- We need to manage the scaling, load balancing, and other things.
- So In this case, We can leverage the K8S to solve the above problems.

K8S Setup

- First, we have to install docker to run the minikube.

```
*yum install docker -y*
*systemctl start docker*
```

- Then we have to install minikube. You can use this link to install minikube - https://minikube.sigs.k8s.io/docs/start/
- Once the minikube is installed, we can start the minikube server using the below command.

```
*minikube start –force*
```

Why did we install Minikube?

- Generally, K8S comes with a lot of services, So it's a lengthy process to install and manage them.
- So, we can use minikube to learn and develop for Kubernetes.

Kubectl

- Now, we have to install `*kubectl*` to communicate with the Kubernetes cluster.
- You can use this link to install kubectl - https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/ and remember to make it executable.

- `kubectl` is a client side program used to communicate with the Kubernetes cluster.
- We can use `kubectl` to deploy applications, inspect and manage cluster resources, and view logs.

K8s Common Commands

- *kubectl run pod_name --image=nginx* - This command runs a pod in the Kubernetes cluster.
- *kubectl get pods* - This command is used to get the pods in the Kubernetes cluster.
- *kubectl logs <pod name>* - Get the pod logs
- *docker exec -it minikube bash* - Go inside the minikube container with bash shell
- *kubectl exec -it mydep1-5d9699b6b7-mmjvv bas*h - Get the bash terminal of given pod
- *kubectl delete pod pod_name* - This command is used to delete the pod in the Kubernetes cluster.
- *kubectl api-resources* - List down the kubernetes api-resources and it's details
- *kubectl apply -f service.yml* - Create the k8s resources from the yml files.
- *kubectl expose pod pod_name --port=80 --name=nginx-service --type=NodePort* - This command is used to expose the pod in the Kubernetes cluster with NodePort Type.
- *kubectl get svc* - This command is used to get the services in the Kubernetes cluster.

- *kubectl create deployment mydep1 --image=nginx* - This command is used to create a deployment in the Kubernetes cluster.
- *kubectl get deployments* - This command is used to get the deployments in the Kubernetes cluster.
- *kubectl delete deployment mydep1* - This command is used to delete the deployment in the Kubernetes cluster.

Kubernetes file

- We can use commands to create the pods, deployments, services, etc. But, it's difficult to manage them.
- So, we can use the Kubernetes configuration files to manage them.
- It will be written in the YAML format.

- *Example of Kubernetes file:*

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
  labels:
    app: myapp
spec:

  containers:
  - name: mycontainer
    image: nginx
    ports:
    - containerPort: 80
```
- In this above example, we are creating a pod using the Kubernetes configuration file.
- Understand Persistent Volume - https://spacelift.io/blog/kubernetes-persistent-volumes

- Health Probes - https://semaphoreci.com/blog/kubernetes-probes#:~:text=Kubernetes%2C%20the%20industry%2Dleading%20container,pods%20and%20their%20hosted%20applications.
- Kubernetes Services - https://cloud.google.com/kubernetes-engine/docs/concepts/service#:~:text=applications%20using%20services.-,What%20is%20a%20Kubernetes%20Service%3F,contact%20Pods%20in%20the%20Service.
- Stateful vs Stateless App - https://www.redhat.com/en/topics/cloud-native-apps/stateful-vs-stateless
- Benefits of K8s - https://www.ibm.com/blog/top-7-benefits-of-kubernetes/

- history of K8s - https://www.ibm.com/blog/kubernetes-history/
- Deployment Strategies - https://spacelift.io/blog/kubernetes-deployment-strategies

Kubernetes Configuration for Deploying a MERN App

To deploy your MERN (MongoDB, Express.js, React.js, Node.js) application on Kubernetes, follow the steps below:

MERN Application on Kubernetes

Objective:

- To deploy a Mern application using K8S components like pods, deployments, services, etc.

What is MERN

- MERN is a full-stack JavaScript solution that enables you to build fast, robust, and maintainable production web applications using MongoDB, Express, React, and Node.js.

  Frontend/Backend Image: mongo-express:latest
  Database Image: mongo:5.0

Components Overview:

- mongo-express: A web-based MongoDB admin interface written with Node.js, Express, and Bootstrap3.
- MongoDB: A document database that stores data in JSON-like documents.

Deployment Strategy

- We will create two separate deployments: one for the application (frontend/backend) and one for the database.

Frontend/Backend Deployment:

- Image: mongo-express:latest
- Deployment: Ensures high availability by automatically creating new pods if any pod goes down.
- Service Type: NodePort to expose the frontend/backend to the outside world.
- NodePort: Exposes the deployment outside the cluster.

Database Deployment:

- Image: mongo:5.0
- Deployment: Similar to the frontend/backend, it ensures high availability by managing the lifecycle of the pods.
- Service Type: ClusterIP for internal communication within the cluster.
- ClusterIP: Exposes the deployment within the cluster.
- Secrets: Used to store sensitive information like username and password.
- ConfigMap: Stores non-sensitive data in key-value pairs, such as the database URL.
- PersistentVolume (PV) and PersistentVolumeClaim (PVC): Used to persist data so that it remains safe even if a pod goes down.

Steps to deploy Mern Application on Kubernetes

1. Create a Deployment for MongoDB

- Create a file named `mongo-app.yaml` and add the below content.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongo-deployment
  labels:
    dc: mumbai
    env: prod
    app: mongo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mongo
  template:
    metadata:
      name: mongodb
      labels:
        app: mongo
    spec:
      containers:
      - name: mongo-db
        image: mongo:5.0
        ports:
        - containerPort: 27017
        volumeMounts:
        - mountPath: /data/db
          name: mongo-volume
        env:
        - name: MONGO_INITDB_ROOT_USERNAME
          valueFrom:
            secretKeyRef:
              name: mongo-secret
              key: mongo-user
        - name: MONGO_INITDB_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mongo-secret
              key: mongo-password
      volumes:
```

```
    - name: mongo-volume
      persistentVolumeClaim:
        claimName: mongo-pvc

    #emptyDir Volume Type ->
    # volumes:
    # - name: mongo-volume
    #   emptyDir: {}

    #hostPath Volume Type ->
    #   volumes:
    # - name: mongo-volume
    #   hostPath:
    #     path: /data
```

Explanation:

- Deployment: Creates the MongoDB pods.
- Labels: Identifies the deployment.
- Replicas: Number of pods to be created.
- Selector: Matches the pods.
- Template: Defines the pod structure.
- Containers: Specifies container details.
- VolumeMounts: Defines the mount path for the volume.
- Volumes: Defines the volume.
- Environment Variables: Defines the environment variables.
- PersistentVolumeClaim: Claims the PersistentVolume.

2. Create a Deployment for the Web Application

- Create a file named `mongo-webapp.yaml` and add the below content.

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

```
    containers:
    - name: webapp
      image: mongo-express:latest
      ports:
      - containerPort: 8081
      env:
      - name: ME_CONFIG_MONGODB_ADMINUSERNAME
        valueFrom:
          secretKeyRef:
            name: mongo-secret
            key: mongo-user
      - name: ME_CONFIG_MONGODB_ADMINPASSWORD
        valueFrom:
          secretKeyRef:
            name: mongo-secret
            key: mongo-password
      - name: ME_CONFIG_MONGODB_SERVER
        valueFrom:
          configMapKeyRef:
            name: mongo-config
            key: mongo-url
```

Explanation:

- Deployment: Creates the web application pods.
- Labels: Identifies the deployment.
- Replicas: Number of pods to be created.
- Selector: Matches the pods.
- Template: Defines the pod structure.
- Containers: Specifies container details.
- Environment Variables: Defines the environment variables.
- ConfigMapKeyRef: Refers to the ConfigMap.

Now that we have created the application and database deployments, But How the application will communicate with the database?

- We need to create a service for the database to communicate with the frontend/backend.

3. Create a Service for MongoDB

- Create a file named `mongo-service.yaml` and add the below content.

```
apiVersion: v1
kind: Service
metadata:
  name: mongo-service
spec:
```

```
  selector:
    app: mongo
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 27017
      targetPort: 27017
```

Explanation:

- We are creating a service for the mongo database.
- We used `ClusterIP` type to communicate within the cluster Because we didn't want to expose the database to the outside world.
- selector: To select the pods.
- type: Type of the service.
- ports: To define the ports.
- protocol: Protocol to be used.
- port: The port number we want to access our service.
- targetPort: The port number, the container is listening on.

4. Create a Service for the Webapp

- Create a file named `webapp-service.yaml` and add the below content.

```
|
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
spec:
  type: NodePort
  selector:
    app: webapp
  ports:
    - protocol: TCP
      port: 8081
      targetPort: 8081
      nodePort: 30111
```

Explanation:

- We are creating a service for the webapp.
- We used `NodePort` type to communicate with the outside world.
- selector: To select the pods.
- type: Type of the service.
- ports: To define the ports.

- protocol: Protocol to be used.
- port: The port number we want to access our service.
- targetPort: The port number, the container is listening on.

5. Create a Secret for MongoDB

- Create a file named `mongo-secret.yaml` and add the below content. In Kubernetes, we can use secrets to store sensitive information like passwords, OAuth tokens, and SSH keys.

```
apiVersion: v1
kind: Secret
metadata:
  name: mongo-secret
type: Opaque
data:
  mongo-user: YWRtaW4=
  mongo-password: bXlnZmc=
```

Explanation:

- We are creating a secret for the mongo database.
- type: Type of the secret.
- data: To define the data.
- mongo-user: The username for the mongo database.
- mongo-password: The password for the mongo database.

6. Create a ConfigMap for MongoDB

- Create a file named `mongo-config.yaml` and add the below content. In Kubernetes, we can use ConfigMaps to store non-sensitive data in key-value pairs.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mongo-config
data:
  mongo-url: mongo-service
```

Explanation:

- We are creating a configMap for the mongo database.
- data: To define the data.
- mongo-url: We give mongo-service as the value, So that the webapp can use the mongo-service to communicate with the database.

7. Create a PersistentVolume and PersistentVolumeClaim for MongoDB

- Create a file named `mongo-pv.yaml` and add the below content.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongo-pv
spec:
  capacity:
    storage: 0.5Gi
  accessModes:
    - ReadWriteMany
  local:
    path: /storage/gfg
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
                - minikube
```

Explanation:

- We are creating a PersistentVolume for the mongo database.
- capacity: To define the capacity of the PV.
- accessModes: To define the accessModes.
- local: To define the path.
- nodeAffinity: To define the nodeAffinity.

8. Create a file named `mongo-pvc.yaml` and add the below content.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongo-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 0.3Gi
  storageClassName: ""
```

Explanation:

- We are creating a PersistentVolumeClaim for the mongo database.
- accessModes: To define the accessModes, here it is ReadWriteMany

- resources: To define the resources.
- storageClassName: To define the storageClassName.

Apply the configuration files with the following commands:

- #kubectl apply -f mongo-app.yaml
- #kubectl apply -f webapp-deployment.yaml
- #kubectl apply -f mongo-service.yaml
- #kubectl apply -f webapp-service.yaml
- #kubectl apply -f mongo-secret.yaml
- #kubectl apply -f mongo-config.yaml
- #kubectl apply -f mongo-pv.yaml
- #kubectl apply -f mongo-pvc.yaml

Checking the Status of Pods and Services

- kubectl get pods
- kubectl get svc

Accessing the Web Application

- NodePort Service: Exposes the web application outside the Kubernetes cluster.
- Local Machine Access: Use the Minikube IP and NodePort to access the web application.
- Cloud Deployment: Use socat to access the web application from the internet. Install socat with yum install socat -y and use the command socat TCP4-LISTEN:8080 TCP4:192.168.49.2:30111 &.

References:

- [Kubernetes](https://kubernetes.io/docs/home/)
- [MongoDB](https://www.mongodb.com/)
- [Mongo-Express](https://www.npmjs.com/package/mongo-express)
- [Kubernetes Secrets](https://kubernetes.io/docs/concepts/configuration/secret/)
- [Kubernetes ConfigMap](https://kubernetes.io/docs/concepts/configuration/configmap/)
- [Kubernetes PersistentVolume](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)
- [Kubernetes PersistentVolumeClaim](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistentvolumeclaims)
- [Kubernetes Services](https://kubernetes.io/docs/concepts/services-networking/service/)
- [Kubernetes Deployments](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)
- [Kubernetes Pods](https://kubernetes.io/docs/concepts/workloads/pods/pod/)
- [Kubernetes Volumes](https://kubernetes.io/docs/concepts/storage/volumes/)

- [Kubernetes NodePort](https://kubernetes.io/docs/concepts/services-networking/service/#nodeport)
- [Kubernetes ClusterIP](https://kubernetes.io/docs/concepts/services-networking/service/#clusterip)
- [Kubernetes Minikube](https://minikube.sigs.k8s.io/docs/start/)
- [Kubernetes Kubectl](https://kubernetes.io/docs/reference/kubectl/overview/)

Helm and Istio Service Mesh

Helm

Helm is a package manager for Kubernetes that simplifies the deployment and management of applications. Helm uses a packaging format called **Charts**, which are collections of files that describe a related set of Kubernetes resources.

Benefits of Helm:

- **Simplifies Kubernetes Deployments**: Helm Charts package Kubernetes resources into a single unit that can be deployed with a single command.
- **Version Control**: Helm allows you to version your deployments.
- **Configuration Management**: Helm Charts can be customized using values files.
- **Reusability**: Helm Charts can be reused across different environments and shared within the community.

Basic Helm Commands:

- helm install <release-name> <chart>: Install a Helm Chart.
- helm upgrade <release-name> <chart>: Upgrade an existing release.
- helm rollback <release-name> <revision>: Rollback a release to a previous revision.
- helm uninstall <release-name>: Uninstall a release.

Istio Service Mesh

Istio is an open-source service mesh that provides a uniform way to connect, manage, and secure microservices. It abstracts the network layer from application code and offers features like traffic management, security, and observability.

Key Features of Istio:

- **Traffic Management**: Fine-grained control of traffic behavior with rich routing rules, retries, failovers, and fault injection.
- **Security**: Istio provides strong identity, powerful policy, and transparent TLS encryption.
- **Observability**: Istio provides robust tracing, monitoring, and logging features.

Istio Components:

- **Envoy Proxy**: A sidecar proxy deployed with each service to handle network traffic.
- **Pilot**: Manages and configures proxies to route traffic.
- **Mixer**: Enforces access control and usage policies.
- **Citadel**: Provides strong service-to-service and end-user authentication.

Role-Based Access Control (RBAC) and Service Accounts

Role-Based Access Control (RBAC)

RBAC in Kubernetes is a method for regulating access to resources based on the roles of individual users or service accounts within your organization. It enables fine-grained control over what users and applications can do within a Kubernetes cluster.

Key Concepts:

- **Roles**: Define a set of permissions (verbs like get, list, create) over specific resources (pods, services).
- **ClusterRoles**: Similar to roles, but apply cluster-wide.
- **RoleBindings**: Bind a role to a user or service account within a namespace.
- **ClusterRoleBindings**: Bind a cluster role to a user or service account cluster-wide.

Example:

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

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: "jane"
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
```

```
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Service Accounts

Service Accounts in Kubernetes are used to provide an identity for processes that run in a Pod. This allows you to grant specific permissions to the pods and ensure secure access control.

Creating a Service Account:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-service-account
  namespace: default
```

Binding a Role to a Service Account:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: ServiceAccount
  name: my-service-account
  namespace: default
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

HPA, Ingress, Taint and Toleration

Horizontal Pod Autoscaler (HPA)

HPA automatically scales the number of pod replicas in a deployment or replica set based on observed CPU utilization or other select metrics.

Example:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: myapp-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: myapp
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

Ingress

Ingress manages external access to services in a cluster, typically HTTP. It provides load balancing, SSL termination, and name-based virtual hosting.

Ingress Controller in Kubernetes

- An Ingress Controller is a specialized load balancer for managing external access to Kubernetes services in a cluster, typically HTTP/HTTPS traffic. It routes traffic from outside the Kubernetes cluster to services within the cluster based on rules defined in an Ingress resource.

Why Use an Ingress Controller?

- Centralized Management: Provides a single entry point for all incoming traffic, simplifying routing and traffic management.
- Load Balancing: Distributes incoming traffic across multiple backend services to ensure high availability and reliability.
- SSL Termination: Manages SSL/TLS certificates, offloading SSL termination from the backend services.
- Path-Based Routing: Routes traffic to different services based on URL paths or hostnames.

How It Works

- Ingress Resource Definition: Define an Ingress resource with rules specifying how to route external traffic to internal services.
- Ingress Controller Deployment: Deploy an Ingress Controller (such as NGINX, HAProxy, Traefik, etc.) that watches for Ingress resources and configures the load balancer accordingly.

- Traffic Routing: The Ingress Controller receives incoming requests, evaluates the rules in the Ingress resource, and forwards the requests to the appropriate backend services.

Ingress Controller in Kubernetes: A Step-by-Step Guide

- *Find cert.yml, ingress.yml, self-signed-ClusterIssuer.yml Here-* ***https://github.com/sudhanshuvlog/GFG-Devops18/tree/main/K8s/Ingress%20Controller***

- Step 1: Ensure Minikube is Running Correctly
  - First, enable the ingress addon in Minikube:
  - *#minikube addons enable ingress*

- Step 2: Expose Minikube to the Host
  - To route traffic from your host to the Minikube cluster, set up IP tables:
  - *#sudo iptables -t nat -A DOCKER -p tcp --dport 80 -j DNAT --to-destination $(minikube ip):80*
  - *#sudo iptables -t nat -A DOCKER -p tcp --dport 443 -j DNAT --to-destination $(minikube ip):443*

- Step 3: Configure /etc/hosts on EC2 Instance
  - Add the Minikube IP to your /etc/hosts file to resolve your local domain:
  - #sudo vi /etc/hosts

    # Add the following line

    <MINIKUBE_IP> example.local

- Step 4: Install and Configure NGINX Ingress Controller
  - Install Helm: *#curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash*
  - Add Ingress NGINX Repo:

    #helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx

    #helm repo update

    - Remove Existing Ingress Class (if any):

      #kubectl delete ingressclass nginx

    - Install NGINX Ingress Controller:

#helm install nginx-ingress ingress-nginx/ingress-nginx --namespace ingress-nginx --create-namespace

- Step 5: Create Ingress Resource
  - Deploy Example Application:

    #kubectl create deployment example-app --image=gcr.io/google-samples/hello-app:1.0

    - Expose Deployment as a Service:
      - *#kubectl expose deployment example-app --port=8080 --target-port=8080 --name=example-service*

    - Apply Ingress Resource:
      - *#kubectl apply -f ingress.yaml*

- Step 6: Install and Configure cert-manager
  - Add Jetstack Helm Repository:
  - *#helm repo add jetstack https://charts.jetstack.io*
  - *#helm repo update*

    - Install cert-manager CRDs:
      - *#kubectl apply -f https://github.com/jetstack/cert-manager/releases/download/v1.10.1/cert-manager.crds.yaml*

    - Install cert-manager Using Helm:
      - *#kubectl create namespace cert-manager*
      - *#helm install cert-manager jetstack/cert-manager --namespace cert-manager --version v1.10.1*
      - Verify cert-manager Pods: *#kubectl get pods --namespace cert-manager*

    - Apply Self-Signed ClusterIssuer:
      - *#kubectl apply -f selfsigned-clusterissuer.yaml*
      - Apply Certificate Resource: *#kubectl apply -f cert.yml*

- Verify the Setup

  To verify that everything is working, use curl to access your application: *curl https://example.local -k*

Pod Design Patterns in Kubernetes

In Kubernetes, a Pod is the smallest deployable unit that can run a container. It represents a single instance of a running process in your cluster. Pod design patterns are reusable templates or best practices for organizing and managing containers within Pods to solve common application needs and challenges. These patterns help to streamline the deployment, scaling, and maintenance of applications. Here are some key Pod design patterns:

- Sidecar Pattern

  The Sidecar pattern involves running a helper container alongside the main container within the same Pod. The sidecar container enhances or extends the functionality of the main container without modifying it directly.

  Use Cases:

  - Logging and Monitoring: Use a sidecar container to collect and forward logs, metrics, or traces from the main container.
  - Security: Implement service mesh proxies like Envoy for traffic management and security features.
  - Configuration Management: Fetch and manage secrets and configuration data.

- Adapter Pattern

  The Adapter pattern, also known as the Ambassador pattern, involves using a container to act as an interface between the main container and external services. This pattern helps to abstract and manage the communication with external systems.

  Use Cases:

  - API Gateway: Use an adapter to manage external API calls and traffic routing.

- Service Proxy: Implement a proxy to handle requests to and from external services.

- Init Container Pattern

  Init containers run and complete before the main application containers start. They are used to perform initialization tasks such as setting up configurations, loading data, or waiting for dependencies to be ready.

  Use Cases:

  - Data Initialization: Pre-load data or configuration files.
  - Dependency Check: Ensure that dependencies or external services are available before starting the main application.

  Example:

  apiVersion: v1

  kind: Pod

  metadata:

  name: init-container-example

  spec:

  initContainers:

  - name: init-myservice

  image: busybox

  command: ['sh', '-c', 'echo Initializing...']

  containers:

  - name: main-app

  image: my-app:latest

Sidecar Containers in Kubernetes

A sidecar container is a secondary container that runs alongside the main application container within a single Kubernetes Pod. This design pattern enhances or extends the functionality of the main container without modifying it directly, allowing for additional capabilities such as logging, monitoring, proxying, security, or configuration management to be injected into your application in a modular and reusable manner.

Common Sidecar Strategies

- Logging and Monitoring:
    - Fluentd/Fluent Bit: A sidecar container can run Fluentd or Fluent Bit to collect, aggregate, and ship logs from the main container to a centralized logging system. This ensures consistent log handling and real-time monitoring.
    - Prometheus Exporter: A sidecar can run a Prometheus exporter to collect and expose metrics from the main container to Prometheus, enabling robust monitoring and alerting.

- **Security:**
    - Service Mesh Proxies (e.g., Istio, Linkerd): A sidecar proxy like Envoy (used by Istio) can manage inbound and outbound traffic for the main container, providing features like traffic encryption, mutual TLS, and traffic policies.
    - Authentication/Authorization: Sidecar containers can handle authentication and authorization tasks, simplifying the main application's code by offloading these responsibilities.