Kubernetes

Kubernetes is an open-source orchestration tool.
It is designed by Google and written in Golang.
We can automate scaling, deploying and managing our application.

Why Use Kubernetes?

Kubernetes efficiently manages containerized applications by providing:

1. **Scalability**:
   Automatically scales applications up or down based on demand.
2. **High Availability**:
   Ensures applications remain available by restarting or rescheduling failed containers.
3. **Resource Optimization**:
   Efficiently distributes workloads across servers.
4. **Automation**:
   Automates deployment, scaling, and management.
5. **Load Balancing**:
   Distributes network traffic to avoid overload.
6. **Self-Healing**:
   Detects failures and replaces unhealthy containers.
7. **Multi-Cloud Support**:
   Works across cloud, on-premises, and hybrid environments.
8. **Rolling Updates**:
   Updates applications without downtime.
9. **Declarative Configuration**:
   Uses YAML files to define application states.
10. **Extensibility**:
    Supports plugins and tools for logging, monitoring, and security.

Kubernetes Architecture

Kubernetes follows a **client-server architecture** and consists of two main components:

1. **Master Node (Control Plane)**
2. **Worker Nodes**

---

1. Master Node (Control Plane)

The Master Node is the "brain" of Kubernetes. It controls and manages the entire cluster.

Key Components of the Master Node:

- **Kube-API Server**:
  - Entry point to the cluster.
  - Handles all API requests (e.g., to create Pods, Deployments, or Services).
  - Exposes Kubernetes API for communication between components.

    - Command-line tools like kubectl interact with the API Server.
- **Kube-Scheduler**:
    - Determines which Worker Node is best suited to run a Pod.
    - Considers resource availability (CPU, memory) and workload balance.
- **Controller Manager**:
    - The "controller" of Kubernetes that maintains the cluster's desired state.
    - Includes the following controllers:
        - **Node Controller**: Monitors node status (alive or failed).
        - **Replication Controller**: Ensures the desired number of Pods are running.
        - **Endpoint Controller**: Manages endpoints for services and Pods.
        - **Job Controller**: Ensures jobs (batch tasks) are completed successfully.
- **ETCD (Key-Value Store)**:
    - A database that stores all cluster data (e.g., configuration, Pod states, secrets, service names).
    - Acts as the "memory" of the cluster.
    - Highly available and consistent (critical for cluster health).

---

2. Worker Nodes

The Worker Nodes are where the actual work happens — they run your containerized applications.

Key Components of a Worker Node:

- **Kubelet**:
    - An agent that runs on each Worker Node.
    - Manages Pods on its node and ensures containers are healthy.
    - Communicates with the API Server on the Master Node.
- **Kube-Proxy**:
    - Handles network communication on the node.
    - Manages traffic between Pods and between Pods and the external world.
    - Ensures services are accessible.
- **Container Runtime**:
    - Software responsible for running containers (e.g., Docker, containerd, CRI-O).
    - Without a container runtime, Pods cannot run.

Pods

- A Pod is the smallest deployable unit in Kubernetes.
- Each Pod can contain one or more containers (e.g., Nginx, MySQL).
- **Containers in a Pod Share**:
    1. **Network**: Same IP address and port space.
    2. **Storage**: Shared volumes.

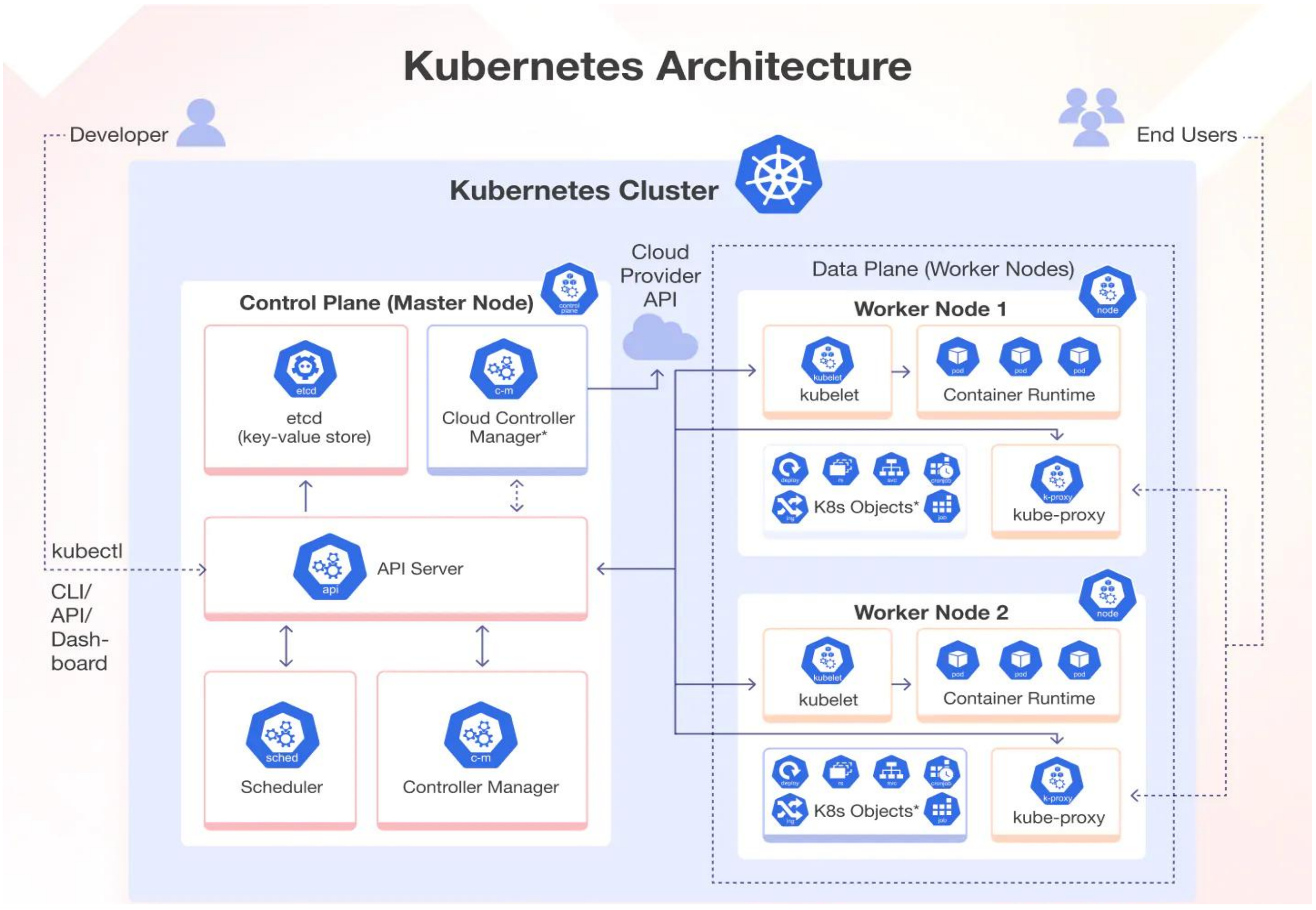
How It All Works Together

1. **User Interaction**:
    - A user (or CI/CD tool) interacts with the Kubernetes cluster using kubectl (CLI tool).
    - The request goes to the **Kube-API Server**.
2. **Kube-Scheduler**:

- Determines the best Worker Node to deploy the requested Pods.

3. **Kubelet**:
   - The Kubelet on the chosen Worker Node communicates with the API Server.
   - It creates and manages the Pods on that node.
4. **Container Runtime**:
   - Runs the actual container images (like Docker).
5. **Kube-Proxy**:
   - Manages networking and ensures that services can route traffic to the correct Pod.
6. **ETCD**:
   - Continuously stores the cluster state (e.g., Pod status, configurations).

Kubernetes Architecture Diagram



Cluster Creation

Minikube Setup

Minikube is a tool that allows you to run a Kubernetes cluster locally. It is ideal for learning Kubernetes, development, and testing environments without requiring a full-scale Kubernetes deployment.

- **Purpose**: Enables developers to spin up a single-node Kubernetes cluster on their local machine.
- **Key Features**:
    - Simulates a Kubernetes environment for testing configurations and applications.
    - Supports multiple container runtimes like Docker, CRI-O, and containerd.
    - Offers add-ons to test features like DNS, Ingress, and metrics.

Steps to Install Minikube:

1. Download the Minikube binary:

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
```

2. Install Minikube:

```
sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-linux-amd64
```

3. Start Minikube:

```
minikube start --force
```

4. Verify the pods in all namespaces:

```
kubectl get po -A
```

---

kubectl

kubectl is the Kubernetes command-line tool used to interact with Kubernetes clusters. It communicates with the Kubernetes API server to perform various operations.

- **Purpose**: Enables users to manage and automate Kubernetes cluster operations.
- **Key Features**:
    - Create, update, delete, and monitor Kubernetes resources.
    - Apply configuration files written in YAML or JSON formats.
    - Debug Kubernetes applications and control their state.

Steps to Install kubectl:

1. Download the kubectl binary:

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
```

2. Install kubectl:

- sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
- chmod +x kubectl

3. Move kubectl to the user's local bin directory:

- mkdir -p ~/.local/bin
- mv ./kubectl ~/.local/bin/kubectl

4. Verify the kubectl installation:

- kubectl version --client

Manifest YAML

YAML (YAML Ain't Markup Language) is a data serialization language widely used for defining configurations, especially in Kubernetes.

- **Purpose**: Provides a human-readable and structured format for defining Kubernetes resources.
- **Key Features**:
  - Extensively used for defining Pods, Deployments, Services, ConfigMaps, etc.
  - Flexible, easy-to-read syntax with support for hierarchical data.
  - Compatible with most programming languages.

Example YAML File for a Pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: nginx
```

YAML files typically have extensions .yaml or .yml.

EKS

Amazon Elastic Kubernetes Service (EKS) is a managed Kubernetes service by AWS. It simplifies deploying, managing, and scaling Kubernetes clusters on the AWS cloud.

- **Purpose**: Provides an enterprise-ready Kubernetes platform integrated with AWS infrastructure and services.
- **Key Features**:
  - Fully managed control plane and integration with AWS security and monitoring tools.
  - Multi-region and hybrid cloud support.
  - Supports both managed and self-managed worker nodes.

EKS Pre-requisites

1. **AWS CLI Installation**:
   AWS CLI is used to interact with AWS services.

- curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
- apt install unzip
- unzip awscliv2.zip
- sudo ./aws/install
- aws --version

2. **eksctl Installation**:
   eksctl is a CLI tool designed to simplify the creation and management of EKS clusters.

- curl --silent --location "https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$(uname -s)_amd64.tar.gz" | tar xz -C /tmp
- sudo mv /tmp/eksctl /usr/local/bin
- eksctl version

3. **kubectl Installation**:
   kubectl is required to manage Kubernetes resources in EKS.

- curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
- chmod +x kubectl
- sudo mv kubectl /usr/local/bin
- kubectl version --client

4. **IAM User/Role**:
   Ensure that your IAM role or user has the necessary permissions to manage EKS.

EKS Cluster Using Manifest File

Manifest Example:

```yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: UAT-Cluster1
  region: us-east-1
  version: "1.27"
nodeGroups:
  - name: UAT-nodegroup
    instanceType: t3.small
    minSize: 1
    desiredCapacity: 1
    maxSize: 3
```

Commands:

1. Create the cluster:

- eksctl create cluster -f cluster.yaml

2. Verify the cluster:

- eksctl get cluster

- kubectl get nodes -o wide

3. Delete the cluster:

- eksctl delete cluster --name UAT-Cluster1

EKS Cluster Using CLI

eksctl create cluster \
--name demo-cluster \
--region us-west-2 \
--nodegroup-name demo-nodegroup \
--node-type t3.medium \
--nodes 2 \
--nodes-min 1 \
--nodes-max 3 \
--managed

Explanation of Flags:

- --name: Specifies the name of the cluster.
- --region: Specifies the AWS region for the cluster.
- --nodegroup-name: Name for the worker node group.
- --node-type: Specifies the EC2 instance type for nodes.
- --nodes: Specifies the desired number of worker nodes.
- --nodes-min and --nodes-max: Define the minimum and maximum node scaling limits.
- --managed: Enables managed node groups.

Verification and Maintenance Commands:

1) List all clusters:

- eksctl get cluster --region us-west-2

2) Check cluster nodes:

- kubectl get nodes

3) Scale node group:

- eksctl scale nodegroup --cluster demo-cluster --name demo-nodegroup --nodes 4

4) Upgrade cluster:

- eksctl upgrade cluster --name demo-cluster --region us-west-2 –approve

5) Delete the cluster:

- eksctl delete cluster --name demo-cluster --region us-west-2

NAMESPACES IN KUBERNETES

A **Namespace** in Kubernetes is a way to organize and isolate resources within a cluster. Namespaces allow you to group related objects together, providing a mechanism for separating different environments or workloads within the same Kubernetes cluster. For example, you can have separate namespaces for **development**, **staging**, and **production** environments. Namespaces help prevent name collisions, provide access control, and enable resource quotas for each namespace.

Namespaces are ideal for:

- **Environment isolation**: Managing separate environments (e.g., dev, test, prod) in the same cluster.
- **Access control**: Different teams can manage different namespaces without affecting others.
- **Resource quotas**: You can define resource limits for a namespace, ensuring fair usage of resources.

Key Concepts

- **Default Namespace**: Every Kubernetes object (like Pods, Services, etc.) is created in the default namespace unless another namespace is specified.
- **System Namespaces**: Kubernetes has several built-in namespaces such as kube-system (for system components) and kube-public (for resources accessible by all users).
- **Resource Isolation**: Namespaces provide a scope for names, meaning two objects in different namespaces can have the same name.
- **Resource Quotas**: You can set limits on the number of resources like CPU, memory, and Pods per namespace, helping to avoid resource contention.

Example YAML to Create a Namespace:

```
apiVersion: v1
kind: Namespace
metadata:
  name: my-namespace
```

- **To create a namespace:**

  ```
  kubectl apply -f namespace.yml
  ```

- **You can set the namespace for your current kubectl context using the following command:**

  ```
  kubectl config set-context --current --namespace=my-namespace
  ```

- **To list all the namespaces in the cluster**:

  ```
  kubectl get namespaces
  ```

- **To get more details about a specific namespace:**

  ```
  kubectl describe namespace my-namespace
  ```

- **To delete a namespace and all the resources within it**:

  ```
  kubectl delete namespace my-namespace
  ```

Installing and Using kubens on Ubuntu

kubens is a helpful CLI utility for quickly switching between namespaces in Kubernetes. It simplifies the process of managing namespaces, as you can easily switch between them without manually editing kubeconfig files or running multiple kubectl commands.

Step-by-Step Installation of kubens on Ubuntu

Step 1: Install kubens Using apt-get (Ubuntu)

1. **Install Dependencies:** First, make sure your system is up-to-date and has the required dependencies:

   ```
   sudo apt update
   sudo apt install -y curl
   ```

2. **Download the Latest kubens Release:** Visit the GitHub releases page of kubectx to download the latest version. For example:

   ```
   curl -LO https://github.com/ahmetb/kubectx/releases/download/v0.9.0/kubens_0.9.0_linux_x86_64.tar.gz
   ```

3. **Extract the tarball:** After downloading, extract the archive:

   ```
   tar -xzvf kubens_0.9.0_linux_x86_64.tar.gz
   ```

4. **Move the kubens Binary to /usr/local/bin:** Move the kubens binary to a directory in your PATH, like /usr/local/bin, to make it executable globally:

   ```
   sudo mv kubens /usr/local/bin/
   ```

Step 2: Verify kubens Installation

After installation, you can verify that kubens is installed and working by running:

```
kubens –help
```

Using kubens to Switch Between Namespaces

- To list all available namespaces in your cluster, simply run:

  ```
  kubens
  ```

This will display a list of namespaces, and you can choose the one you want to switch to.

- To switch to a specific namespace, run:

  ```
  kubens my-namespace
  ```

This will set your current context to the specified namespace (my-namespace in this case).

Best Practices for Using Namespaces

- **Environment Segmentation**: Use namespaces to separate different environments, such as development, staging, and production. This helps in logically isolating different stages of the software lifecycle.
- **Role-Based Access Control (RBAC)**: Combine namespaces with RBAC to control which users or service accounts have access to specific resources within a namespace. For example, limit access to sensitive production environments for only a select set of users.
- **Resource Quotas**: Define resource quotas within namespaces to limit the amount of CPU, memory, and storage used, ensuring fair usage across different environments or teams.
- **Simplify Helm Deployments**: If you are using Helm to deploy applications, namespaces can help you separate releases of the same application across different environments.

Common Commands for Managing Namespaces

- **Create a Namespace:**

  ```
  kubectl create namespace <namespace-name>
  ```

- **Get a List of Namespaces:**

  ```
  kubectl get namespaces
  ```

- **Set the Default Namespace for Your kubectl Context:**

  ```
  kubectl config set-context --current --namespace=<namespace-name>
  ```

- **Delete a Namespace:**

  ```
  kubectl delete namespace <namespace-name>
  ```

- **List Resources in a Specific Namespace:**

  ```
  kubectl get pods --namespace=<namespace-name>
  ```

- **Apply Resources to a Specific Namespace:** You can specify the namespace when applying YAML files by using the `--namespace` flag:

  ```
  kubectl apply -f resource.yaml --namespace=<namespace-name>
  ```

- **Namespaces**: Provide logical separation and organization of resources within a Kubernetes cluster.
- **kubens**: A tool to quickly switch between namespaces, making managing contexts easier.
- **Namespaces Use Cases**:
  - Environment isolation (e.g., dev, prod).
  - Access control via Role-Based Access Control (RBAC).
  - Simplifying Helm deployments across different environments.
- **Default Namespace**: Kubernetes objects are placed in the default namespace unless specified otherwise..

---

1. Pods in Kubernetes

What is a Pod?

- A **Pod** is the smallest and simplest unit of deployment in Kubernetes.
- It represents a single instance of a running process in a cluster and encapsulates one or more containers, storage (volumes), and network settings.
- Pods are ephemeral in nature. When a Pod is terminated, the data inside it is lost unless persistent volumes are used.

Pod Lifecycle

- **Pending**: The Pod is being scheduled to a node.
- **Running**: The Pod is running, and its containers are executing.
- **Succeeded**: All containers in the Pod have terminated successfully.
- **Failed**: At least one container in the Pod has terminated with a failure.
- **Unknown**: The state of the Pod could not be determined.

Types of Containers in Pods

- **Main containers**: The primary containers that run the application logic.
- **Sidecar containers**: Auxiliary containers that are used to add additional functionality like monitoring, logging, or proxying.
- **Init containers**: Run before the main containers in the Pod and perform initialization tasks like setup, database migrations, or configuration fetching.

Pod Example:

YAML for a Simple Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Commands for Pods

- **Create a Pod**:
- `kubectl apply -f pod.yml`
- **Get Pods** (list Pods in the current namespace):
- `kubectl get pods`
- `kubectl get pods -A` to list Pods across all namespaces.
- **Get Pod Details**:
- `kubectl describe pod <pod-name>` Shows detailed information about the Pod, including events, status, and resource usage.
- **Run a Pod** (directly without a YAML file):
- `kubectl run web --image=nginx`
- **Delete a Pod**:
- `kubectl delete pod <pod-name>`

Use Cases for Pods

- **Single Container**: Most Pods run a single container, encapsulating an application or service.
- **Multiple Containers**: Used when you need multiple containers to cooperate, e.g., a web server and a logging agent running together.

---

2. ReplicaSet in Kubernetes

What is a ReplicaSet?

- A **ReplicaSet** is responsible for ensuring that a specified number of identical Pods are running at any time. It is commonly used to maintain high availability.
- **ReplicaSets** are often managed by **Deployments**, but can be used directly to control scaling and Pod availability.

ReplicaSet Lifecycle

- **Desired state**: The desired number of replicas that should be running.
- **Actual state**: The number of Pods currently running.
- **Self-healing**: If a Pod in the ReplicaSet fails or is deleted, the ReplicaSet will automatically create a new one to replace it.

Key Features of ReplicaSets

- Ensures **availability**: Ensures that the desired number of Pods are always available.
- **Self-healing**: Automatically replaces failed or terminated Pods.
- **Scaling**: You can scale the number of replicas up or down using the `kubectl scale` command.

YAML for a ReplicaSet

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Commands for ReplicaSet

- **Create a ReplicaSet**:
- `kubectl apply -f replicaset.yml`
- **List ReplicaSets**:
- `kubectl get replicasets`
- **Scale ReplicaSet** (to increase replicas):
- `kubectl scale replicasets/nginx-replicaset --replicas=5`
- **Delete ReplicaSet**:
- `kubectl delete replicaset nginx-replicaset`

Use Cases for ReplicaSets

- **High Availability**: Ensures that a fixed number of Pods are running, even if some Pods crash.
- **Stateless Applications**: Ideal for stateless applications where each Pod is identical and can be replicated.

3. Deployment in Kubernetes

What is a Deployment?

- A **Deployment** is a higher-level abstraction that manages **ReplicaSets** and ensures that applications are rolled out in a controlled and predictable manner.
- It allows for **rolling updates**, **rollbacks**, and **scaling** of applications.

Key Features of Deployments

- **Declarative updates**: Specify the desired state, and Kubernetes will ensure that it is reached.
- **Rolling updates**: Deploy changes to Pods gradually to ensure zero downtime.
- **Rollback**: Automatically revert to the previous stable version if something goes wrong.
- **Scaling**: Easily scale the number of replicas with `kubectl scale`.

YAML for a Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Commands for Deployment

- **Create a Deployment**:
- `kubectl apply -f deployment.yml`
- **Get Deployments**:
- `kubectl get deployments`
- **Update Deployment (Rolling Update)**:
- `kubectl set image deployment/nginx-deployment nginx=nginx:1.16.0`
- **Rollback Deployment**:
- `kubectl rollout undo deployment/nginx-deployment`
- **Scale Deployment**:
- `kubectl scale deployment/nginx-deployment --replicas=5`
- **Delete Deployment**:
- `kubectl delete deployment nginx-deployment`

Use Cases for Deployments

- **Continuous Deployment**: Ideal for rolling out new versions of applications with zero downtime.
- **Application Scaling**: Easily scale up or down as needed.

---

4. StatefulSet in Kubernetes

What is a StatefulSet?

- A **StatefulSet** is a Kubernetes controller used to manage stateful applications, such as databases or distributed systems, which require **stable, unique network identifiers** and **persistent storage**.
- It provides guarantees about the ordering and uniqueness of Pods.

Key Features of StatefulSets

- **Stable Network Identity**: Pods in a StatefulSet are given persistent names like `mysql-0`, `mysql-1`, etc., making them unique and addressable.
- **Ordered Deployment and Scaling**: StatefulSets deploy Pods in a sequential order and terminate them in reverse order.
- **Persistent Storage**: StatefulSets work with **Persistent Volumes (PVs)** to ensure that the data survives even if Pods are deleted.

YAML for a StatefulSet

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  serviceName: "mysql"
  replicas: 3
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:5.7
        volumeMounts:
        - name: mysql-data
          mountPath: /var/lib/mysql
  volumeClaimTemplates:
  - metadata:
      name: mysql-data
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
```

Commands for StatefulSet

- **Create a StatefulSet**:
- kubectl apply -f statefulset.yml
- **Get StatefulSets**:
- kubectl get statefulsets
- **Scale StatefulSet**:
- kubectl scale statefulset/mysql --replicas=5
- **Delete StatefulSet**:
- kubectl delete statefulset mysql

Use Cases for StatefulSets

- **Stateful Applications**: Databases, message queues, etc.
- **Persistent Storage**: Applications that require data storage persistence between Pod restarts.

---

5. DaemonSet in Kubernetes

What is a DaemonSet?

- A **DaemonSet** ensures that a Pod runs on every node in the cluster (or specific nodes).
- It is ideal for running background services, such as logging or monitoring agents, that need to run on every node.

Key Features of DaemonSets

- **Cluster-wide Services**: Ensure that certain services (e.g., logging, monitoring) are running on every node.
- **Node-Aware**: Automatically adds Pods when new nodes are added and removes Pods when nodes are removed.
- **Flexible Node Selection**: Can be limited to specific nodes using **node selectors** or **taints and tolerations**.

YAML for a DaemonSet

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
spec:
  selector:
    matchLabels:
      app: fluentd
  template:
    metadata:
      labels:
        app: fluentd
    spec:
      containers:
      - name: fluentd
        image: fluent/fluentd:v1.11-1
```

Commands for DaemonSet

- **Create a DaemonSet**:

  ```
  kubectl apply -f daemonset.yml
  ```

- **Get DaemonSets**:

  ```
  kubectl get daemonsets
  ```

- **Delete DaemonSet**:

  ```
  kubectl delete daemonset fluentd
  ```

Use Cases for DaemonSets

- **Logging/Monitoring**: Ensure that logging and monitoring agents are running on all nodes in the cluster.
- **Cluster-Wide Agents**: Ensure that agents running on every node are managed centrally.

KUBERNETES SERVICES

A **Service** in Kubernetes provides a stable endpoint for accessing a set of Pods. Services are critical for ensuring that Pods are discoverable and can communicate with each other, as well as enabling access to external traffic when needed. The Service layer abstracts the complex underlying details of Pods and their IP addresses.

TYPES OF SERVICES

1. CLUSTERIP (DEFAULT)

- Exposes the service on an internal IP address within the cluster. This is the default service type.
- This service type is only accessible within the cluster, meaning it's ideal for internal communication between Pods.

  Example YAML:

  ```
  apiVersion: v1
  kind: Service
  metadata:
    name: my-clusterip-service
  spec:
    selector:
      app: my-app
    ports:
      - protocol: TCP
        port: 80
        targetPort: 8080
    type: ClusterIP
  ```

  Commands:

  - **Create a ClusterIP service:**

```
kubectl apply -f clusterip-service.yml
```

- **Get services:**

```
kubectl get services
```

---

2. NodePort

- Exposes the service on each node's IP at a static port.
- A NodePort service is accessible externally through `<NodeIP>:<NodePort>`. This allows traffic to access the service from outside the Kubernetes cluster.

Example YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: my-nodeport-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
      nodePort: 30007
  type: NodePort
```

Commands:

- **Create a NodePort service:**

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

- **Get services:**

```
kubectl get services
```

---

3. LoadBalancer

- Exposes the service externally through a load balancer. This is often used in cloud environments (e.g., AWS, GCP, Azure).
- It provisions an external load balancer that routes traffic to the backend Pods. When you use this service type, a public IP is created for your service.

Example YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: my-loadbalancer-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: LoadBalancer
```

Commands:

- **Create a LoadBalancer service:**

  ```
  kubectl apply -f loadbalancer-service.yml
  ```

- **Get services:**

  ```
  kubectl get services
  ```

---

4. ExternalName

- Maps a service to an external DNS name. This is useful for integrating with external resources such as databases or other services that exist outside the Kubernetes cluster.
- The service will forward traffic to the specified external DNS name (e.g., an external API).

Example YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: my-externalname-service
spec:
  type: ExternalName
  externalName: example.com
```

Commands:

- **Create an ExternalName service:**

  ```
  kubectl apply -f externalname-service.yml
  ```

- **Get services:**

  ```
  kubectl get services
  ```

---

5. Headless Service

- A Headless Service does not have a `ClusterIP` and is used when you need to directly access individual Pods.
- This service type is useful in scenarios like StatefulSets, where each Pod requires its own DNS record.

Example YAML:

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

```
  name: my-headless-service
spec:
  selector:
    app: my-app
  clusterIP: None
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

Commands:

- **Create a Headless service:**

  ```
  kubectl apply -f headless-service.yml
  ```

- **Get services:**

  ```
  kubectl get services
  ```

Essential Commands for Working with Services

1. **Create a Service**
   To create a service from a YAML file:
2. kubectl apply -f <service-name>.yml
3. **Get Services**
   To list all services in the cluster:
4. kubectl get services
5. **Get Service Details**
   To get more detailed information about a specific service:
6. kubectl describe service <service-name>
7. **Delete a Service**
   To delete a service by its name:
8. kubectl delete service <service-name>
9. **Expose a Pod as a Service**
   If you want to expose a single Pod or a set of Pods as a service, use the following:
10. kubectl expose pod <pod-name> --type=<service-type> --port=<port> --target-port=<target-port>

Key Points :

- **ClusterIP**: Default service type; used for internal communication between Pods.
- **NodePort**: Exposes a port on each node's IP; accessible externally via <NodeIP>:<NodePort>.
- **LoadBalancer**: Exposes a service externally through a cloud-based load balancer.
- **ExternalName**: Maps to an external DNS name for accessing external resources.
- **Headless Service**: Does not assign a cluster IP; used to expose Pods directly (e.g., StatefulSets).

These types of services help manage communication and accessibility in your Kubernetes environment, enabling scalable, secure, and flexible interactions between applications.

Persistent Volumes (PV) and Persistent Volume Claims (PVC)

Persistent Volumes (PV)

A **Persistent Volume (PV)** is a piece of storage in the cluster that has been provisioned by an administrator. It is a resource in the Kubernetes API that represents storage resources in the cluster. PVs are managed by the Kubernetes control plane and exist independently of any Pod, providing a stable storage resource that survives Pod restarts. PVs are backed by various storage systems, such as NFS, AWS EBS, Google Cloud Storage, or even local storage on the nodes.

- **Access Modes**: Defines how the storage can be mounted by Pods:
  - **ReadWriteOnce (RWO)**: A volume can be mounted as read-write by a single node.
  - **ReadOnlyMany (ROX)**: A volume can be mounted as read-only by many nodes.
  - **ReadWriteMany (RWX)**: A volume can be mounted as read-write by many nodes.
- **ReclaimPolicy**: Defines what happens to a PV after its claim is deleted:
  - **Retain**: The PV is not deleted and remains available for manual reclamation.
  - **Recycle**: The PV is scrubbed and made available again for future claims.
  - **Delete**: The PV and its associated storage are deleted when the claim is deleted.

EXAMPLE PV YAML:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 5Gi                          # Define storage capacity
  volumeMode: Filesystem                  # Defines volume type (Filesystem or Block)
  accessModes:
    - ReadWriteOnce                       # Only one node can read/write at a time
  persistentVolumeReclaimPolicy: Retain   # Retain the volume after deletion of PVC
  storageClassName: manual                # Define the storage class for the PV
  hostPath:
    path: /mnt/data                       # Specifies where the volume is mounted on the node
```

PERSISTENT VOLUME CLAIMS (PVC)

A **Persistent Volume Claim (PVC)** is a request for storage by a user. It defines the amount of storage required and the access mode. PVCs allow users to request storage without needing to know where the storage is physically located. PVCs are bound to available PVs that meet the requested size and access mode. PVCs are used to decouple the storage request from the storage provisioning.

- **Requests**: PVCs request storage resources (size and access modes).
- **Storage Classes**: A PVC can specify a **storage class** to request a specific type of storage, which Kubernetes uses to match the PVC with an appropriate PV.

EXAMPLE PVC YAML:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce       # Request access mode for read/write by one node
  resources:
    requests:
      storage: 5Gi        # Request storage of 5Gi
```

Commands for PV/PVC:

- **Create a PVC:**

```
kubectl apply -f pvc.yml
```

- **Create a PV:**

```
kubectl apply -f pv.yml
```

- **Get PVs:**

```
kubectl get pv
```

- **Get PVCs:**

```
kubectl get pvc
```

- **Describe a PV:**

```
kubectl describe pv my-pv
```

- **Describe a PVC:**

```
kubectl describe pvc my-pvc
```

- **Delete a PVC:**

```
kubectl delete pvc my-pvc
```

- **Delete a PV:**

```
kubectl delete pv my-pv
```

SECRETS

Secrets are used to store sensitive data, such as passwords, OAuth tokens, and SSH keys. Kubernetes Secrets allow you to keep such information in a secure, encoded format (base64 encoded). Secrets are typically used in situations where sensitive data is needed by Pods, such as in application configuration or when connecting to databases.

- **Security**: Secrets are stored in the Kubernetes API server, and they are not easily readable (they are base64-encoded). However, they should still be treated with care and secured properly.
- **Types of Secrets**: Kubernetes supports various types of secrets, including:
  - **Opaque**: Default secret type, allows for arbitrary data.
  - **dockerconfigjson**: For storing Docker registry credentials.
  - **BasicAuth**, **SSHAuth**, **TLS**, etc.

Example Secrets YAML:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque  # Default type
data:
  username: YWRtaW4=  # base64 encoded value for "admin"
  password: cGFzc3dvcmQ=  # base64 encoded value for "password"
```

- **base64 Encoding**: Kubernetes stores secret data in base64-encoded format. For example, to encode the string admin, you can run:
- echo -n 'admin' | base64

Common Commands for Secrets:

- **Create a Secret from a file:**

```
kubectl create secret generic my-secret --from-file=<file-path>
```

- **Create a Secret from literals:**

```
kubectl create secret generic my-secret --from-literal=username=admin --from-literal=password=secretpassword
```

- **Get Secrets:**

```
kubectl get secrets
```

- **Describe a Secret:**

```
kubectl describe secret my-secret
```

- **Delete a Secret:**

```
kubectl delete secret my-secret
```

ConfigMaps

ConfigMaps are used to store non-sensitive configuration data in key-value pairs. ConfigMaps allow users to separate configuration from application code, making it easier to manage and update the configurations without having to modify or redeploy application code. ConfigMaps are typically used to store configuration data that needs to be shared across multiple Pods in the cluster.

- **Use Cases**: Common use cases for ConfigMaps include:
  - Database connection strings.
  - Application configuration settings.
  - Environment-specific configurations.
- **Volume Mounting**: ConfigMaps can be mounted as files or environment variables inside containers.

Example ConfigMap YAML:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-configmap
data:
  database_url: "mongodb://db.example.com"
  log_level: "debug"
```

- **Accessing ConfigMap Data in Pods**: You can access the data from a ConfigMap in Pods either as environment variables or by mounting them as files. For example, you can reference the database_url from the ConfigMap in your application container.

Example of using ConfigMap as environment variable in Pod YAML:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: my-image
      envFrom:
        - configMapRef:
            name: my-configmap
```

Common Commands for ConfigMaps:

- **Create a ConfigMap from a file:**

```
kubectl create configmap my-configmap --from-file=<file-path>
```

- **Create a ConfigMap from literals:**

```
kubectl create configmap my-configmap --from-literal=database_url="mongodb://db.example.com" --from-literal=log_level="debug"
```

- **Get ConfigMaps:**

```
kubectl get configmaps
```

- **Describe a ConfigMap:**

```
kubectl describe configmap my-configmap
```

- **Delete a ConfigMap:**

```
kubectl delete configmap my-configmap
```

- **Persistent Volumes (PV)**: Provide abstracted, persistent storage that can be used by Pods. PVs are independent of Pod lifecycles and allow stable, durable storage.
- **Persistent Volume Claims (PVC)**: Requests for storage resources by users that are bound to available PVs.
- **Secrets**: Store sensitive information securely in the Kubernetes cluster. Use base64 encoding to store the secrets.
- **ConfigMaps**: Store non-sensitive configuration data and provide a flexible way to decouple application configurations from code.

Ingress in Kubernetes

What is an Ingress?

In Kubernetes, **Ingress** is a collection of rules that allow external HTTP and HTTPS traffic to reach services within a cluster. It acts as an entry point to the cluster, managing access to services based on the host and path. Essentially, **Ingress** provides HTTP routing to services and can manage features like SSL termination, URL path-based routing, and host-based routing.

Ingress allows you to expose multiple services over a single external IP address, simplifying access and management. Ingress resources are typically defined as YAML configurations in Kubernetes and are managed by an **Ingress Controller**.

What is an Ingress Controller?

An **Ingress Controller** is a component that runs within a Kubernetes cluster and implements the rules defined in the **Ingress** resources. It is responsible for routing external traffic to the correct service based on the defined rules. Ingress Controllers are typically based on reverse proxy technologies like **NGINX**, **Traefik**, or **HAProxy**.

In other words, while an **Ingress** defines the rules, an **Ingress Controller** actually enforces those rules and manages the external traffic routing.

Key Features of Ingress:

- **Path-based routing**: Direct traffic based on URL paths (e.g., /api to one service, /web to another).
- **Host-based routing**: Route traffic to different services based on the domain name (e.g., api.example.com vs. www.example.com).
- **SSL/TLS termination**: Manage SSL certificates for HTTPS traffic.
- **Load balancing**: Distribute traffic to multiple instances of a service.
- **Authentication and Authorization**: Some Ingress Controllers offer advanced features for access control.

Creating an Ingress Resource in Kubernetes

Here's an example of an **Ingress** resource definition for routing traffic to a service named my-service based on different paths:

Example YAML for Ingress:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: myapp.example.com
    http:
      paths:
      - path: /api
        pathType: Prefix
        backend:
          service:
            name: my-service
            port:
              number: 80
      - path: /web
        pathType: Prefix
        backend:
          service:
            name: web-service
            port:
              number: 80
```

In this example:

- **Host**: Traffic directed to myapp.example.com will be routed.
- **Path**: Traffic with the /api path will be routed to the my-service service, and /web traffic will be routed to web-service.

- **Annotations**: In this case, the annotation rewrites the URL path to the root of the service (/).

COMMON COMMANDS FOR MANAGING INGRESS

- **Create an Ingress**:
- kubectl apply -f ingress.yml
- **Get List of Ingress Resources**:
- kubectl get ingress
- **Describe an Ingress Resource**:
- kubectl describe ingress my-ingress
- **Delete an Ingress**:
- kubectl delete ingress my-ingress
- **Check the Services Exposed by Ingress**:
- kubectl get svc -n ingress-nginx

- **Ingress** provides HTTP/S routing to services based on hostnames and paths. It simplifies exposing multiple services using a single external IP.
- **Ingress Controller** is responsible for processing the rules defined in the Ingress resource. It is often implemented using reverse proxy software like **NGINX**, **Traefik**, or **HAProxy**.
- **Annotations** in the Ingress resource allow customization of the behavior, such as URL rewriting or SSL settings.