DevOps Shack

Kubernetes Comprehensive Hands-On Guide

Click Here To Enrol To Batch-6 | DevOps & Cloud DevOps

Introduction to Kubernetes

Kubernetes is an open-source platform designed to automate deploying, scaling, and operating application containers. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF).

Key Concepts

1. **Cluster**: A set of nodes (machines) running Kubernetes, managed by a control plane.
2. **Node**: A single machine in the cluster, which runs containerized applications.
3. **Pod**: The smallest deployable unit in Kubernetes, which can contain one or more containers.
4. **Deployment**: A controller that manages the deployment and scaling of Pods.
5. **Service**: An abstraction that defines a logical set of Pods and a policy by which to access them.
6. **ConfigMap**: Used to manage configuration data separately from application code.
7. **Secret**: Used to manage sensitive data, such as passwords, OAuth tokens, etc.

Kubernetes Architecture

Introduction

Kubernetes, often abbreviated as K8s, is an open-source container orchestration platform designed to automate the deployment, scaling, and operation of containerized applications. It is built to handle a production environment, with multiple machines and services running simultaneously. Understanding the architecture of Kubernetes is crucial for effectively deploying and managing applications.

Overview of Kubernetes Architecture

Kubernetes follows a client-server architecture and comprises the following key components:

- **Control Plane**: Manages the Kubernetes cluster.
- **Node Components**: Run on every node in the cluster.
- **Objects**: Persistent entities in the Kubernetes system.

Control Plane Components

The control plane components make global decisions about the cluster (e.g., scheduling) and detect and respond to cluster events.

1. **etcd**:
    - **Role**: A consistent and highly-available key-value store used as Kubernetes' backing store for all cluster data.
    - **Importance**: Stores configuration data, representing the state of the cluster at any given point in time.
2. **kube-apiserver**:
    - **Role**: Exposes the Kubernetes API. It is the front-end for the Kubernetes control plane.
    - **Importance**: Validates and configures the data for the API objects, which include pods, services, replication controllers, and others.
3. **kube-scheduler**:
    - **Role**: Watches for newly created Pods with no assigned node and selects a node for them to run on.
    - **Importance**: Ensures Pods are placed on nodes that have sufficient resources to run them.
4. **kube-controller-manager**:
    - **Role**: Runs controller processes.
    - **Importance**: Manages different types of controllers, including the replication controller, which ensures the specified number of pod replicas are running at any one time.
5. **cloud-controller-manager**:
    - **Role**: Runs controllers specific to the cloud provider.
    - **Importance**: Manages cloud-specific control logic, allowing for Kubernetes to be cloud-agnostic.

Node Components

These run on every node in the cluster, ensuring containers are running in a Pod.

1. **kubelet**:
    - **Role**: An agent that runs on each node in the cluster. It ensures containers are running in a Pod.
    - **Importance**: Communicates with the control plane to receive instructions and report back the status.
2. **kube-proxy**:
    - **Role**: Maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.
    - **Importance**: Facilitates the Kubernetes networking model.
3. **Container Runtime**:
    - **Role**: Software responsible for running containers.
    - **Importance**: Examples include Docker, containerd, and CRI-O.

Persistent Storage Components

Kubernetes supports several storage mechanisms, which allow applications to store and retrieve data.

1. **PersistentVolumes (PV)**:
    - **Role**: A piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes.
    - **Importance**: Provides a way for users to store data persistently.
2. **PersistentVolumeClaims (PVC)**:
    - **Role**: A request for storage by a user.
    - **Importance**: Users create PVCs to request storage resources and bind to PVs.
3. **Storage Classes**:
    - **Role**: Define the types of storage classes available in a cluster.
    - **Importance**: Allow dynamic provisioning of PVs.

Kubernetes Objects

These persistent entities represent the state of the cluster. Kubernetes uses these entities to represent the desired state of the cluster and change the actual state to the desired state.

1. **Pods**:
    - **Role**: The smallest and simplest Kubernetes object. A Pod represents a set of running containers on your cluster.
    - **Importance**: The fundamental unit of deployment.
2. **Services**:
    - **Role**: An abstract way to expose an application running on a set of Pods as a network service.

    - **Importance**: Allows for Pods to communicate with each other or external services.
3. **ReplicaSets**:
    - **Role**: Ensures a specified number of pod replicas are running at any given time.
    - **Importance**: Provides high availability.
4. **Deployments**:
    - **Role**: Provides declarative updates for Pods and ReplicaSets.
    - **Importance**: Manages and maintains desired states.
5. **ConfigMaps** and **Secrets**:
    - **Role**: ConfigMaps store configuration data in key-value pairs. Secrets store sensitive information.
    - **Importance**: Manage configuration and sensitive data separately from application code.

Networking in Kubernetes

Kubernetes networking allows for communication between different components within the cluster and with the outside world.

1. **Cluster Networking**:
    - **Role**: Allows communication between Pods within the same cluster.
    - **Importance**: Ensures that each Pod can communicate with any other Pod without NAT.
2. **Service Networking**:
    - **Role**: Exposes services to external networks.
    - **Importance**: Allows external users to access services within the cluster.
3. **Network Policies**:
    - **Role**: Define how groups of Pods can communicate with each other and with other network endpoints.
    - **Importance**: Provide fine-grained control over network traffic.

Kubernetes Architecture Diagram

```
Kubernetes Cluster
├── Control Plane
│   ├── etcd
│   │   └── Key-value store for cluster data
│   ├── kube-apiserver
│   │   └── Front-end for the Kubernetes control plane
│   ├── kube-scheduler
│   │   └── Schedules pods on nodes
│   ├── kube-controller-manager
│   │   ├── Node Controller
│   │   ├── Replication Controller
│   │   ├── Endpoints Controller
│   │   └── Service Account & Token Controllers
│   └── cloud-controller-manager (if using a cloud provider)
│       ├── Node Controller
│       ├── Route Controller
│       └── Service Controller
├── Node Components
│   ├── kubelet
│   │   └── Ensures containers are running in a Pod
│   ├── kube-proxy
│   │   └── Maintains network rules on nodes
│   └── Container Runtime
│       └── Examples: Docker, containerd, CRI-O
├── Persistent Storage Components
│   ├── PersistentVolume (PV)
│   │   └── Storage provisioned by an admin or dynamically
│   ├── PersistentVolumeClaim (PVC)
│   │   └── Request for storage by a user
│   └── Storage Class
│       └── Defines types of storage classes available
├── Kubernetes Objects
│   ├── Pods
│   │   └── Smallest and simplest Kubernetes object
│   ├── Services
│   │   └── Abstract way to expose an application
│   ├── ReplicaSets
│   │   └── Ensures a specified number of pod replicas are running
│   ├── Deployments
│   │   └── Provides declarative updates for Pods and ReplicaSets
│   ├── ConfigMaps
│   │   └── Stores configuration data in key-value pairs
│   └── Secrets
│       └── Stores sensitive information
├── Networking Components
│   ├── Cluster Networking
│   │   └── Allows communication between Pods within the cluster
│   ├── Service Networking
│   │   └── Exposes services to external networks
│   └── Network Policies
│       └── Defines how groups of Pods can communicate
│
```

Setting Up a Kubernetes Cluster

Minikube

Minikube is a tool that allows you to run Kubernetes locally. It's perfect for development and testing.

1. **Install Minikube**:
2.
```
curl -LO
https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

3. **Start Minikube**:

```
minikube start
```

4. **Verify Installation**:

```
kubectl get nodes
```

Kubernetes on Cloud

You can set up a Kubernetes cluster on various cloud providers like Google Kubernetes Engine (GKE), Amazon Elastic Kubernetes Service (EKS), or Azure Kubernetes Service (AKS). Here's an example for GKE:

1. **Install Google Cloud SDK** and authenticate:
2.
```
curl https://sdk.cloud.google.com | bash
exec -l $SHELL
gcloud init
gcloud auth login
```

5. **Create a GKE cluster**:

```
gcloud container clusters create my-cluster --zone us-central1-a
```

6. **Get authentication credentials**:

```
gcloud container clusters get-credentials my-cluster --zone us-central1-a
```

7. **Verify cluster**:

```
kubectl get nodes
```

Deploying Applications

Creating a Deployment

A Deployment provides declarative updates to applications. Here’s how to create one:

1. **Create a Deployment YAML file** (`deployment.yaml`):

```
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
```

2. **Apply the Deployment**:

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

3. **Verify the Deployment**:

```
kubectl get deployments
kubectl get pods
```

Exposing a Deployment

To expose your Deployment so that it can be accessed from outside the Kubernetes cluster, you need to create a Service.

1. **Create a Service YAML file** (`service.yaml`):

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

2. **Apply the Service**:

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

3. **Verify the Service**:

```
kubectl get services
```

Scaling Applications

Scaling applications in Kubernetes is straightforward.

1. **Scale the Deployment**:

   ```
   kubectl scale deployment nginx-deployment --replicas=5
   ```

2. **Verify the Scaling**:

   ```
   kubectl get deployments
   kubectl get pods
   ```

Updating Applications

Updating an application involves updating the image version.

1. **Update the Deployment**:

   ```
   kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1
   ```

2. **Verify the Update**:

   ```
   kubectl rollout status deployment/nginx-deployment
   kubectl get pods
   ```

ConfigMaps and Secrets

ConfigMaps

ConfigMaps allow you to decouple environment-specific configuration from your container images.

1. **Create a ConfigMap**:

   ```
   kubectl create configmap example-config --from-literal=key1=value1 --
   from-literal=key2=value2
   ```

2. **Use ConfigMap in a Pod**:

   ```
   apiVersion: v1
   kind: Pod
   metadata:
     name: configmap-pod
   spec:
     containers:
     - name: mycontainer
       image: busybox
       command: [ "sh", "-c", "env && sleep 3600" ]
       env:
   ```

```
    - name: SPECIAL_KEY
      valueFrom:
        configMapKeyRef:
          name: example-config
          key: key1
```

3. **Apply the Pod configuration**:

```
kubectl apply -f pod-configmap.yaml
```

4. **Verify the Pod**:

```
kubectl exec -it configmap-pod -- env
```

Secrets

Secrets are used to store sensitive information, such as passwords, OAuth tokens, and ssh keys.

1. **Create a Secret**:

```
kubectl create secret generic db-user-pass --from-
literal=username=admin --from-literal=password='S3cr3t!'
```

2. **Use Secret in a Pod**:

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-pod
spec:
  containers:
  - name: mycontainer
    image: busybox
    command: [ "sh", "-c", "env && sleep 3600" ]
    env:
    - name: DB_USERNAME
      valueFrom:
        secretKeyRef:
          name: db-user-pass
          key: username
    - name: DB_PASSWORD
      valueFrom:
        secretKeyRef:
          name: db-user-pass
          key: password
```

3. **Apply the Pod configuration**:

```
kubectl apply -f pod-secret.yaml
```

4. **Verify the Pod**:

```
kubectl exec -it secret-pod -- env
```

Persistent Storage

Kubernetes supports different types of persistent storage. Here's how to use a PersistentVolume and PersistentVolumeClaim.

1. **Create a PersistentVolume** (`pv.yaml`):

   ```
   apiVersion: v1
   kind: PersistentVolume
   metadata:
     name: pv-volume
   spec:
     capacity:
       storage: 1Gi
     accessModes:
       - ReadWriteOnce
     hostPath:
       path: "/mnt/data"
   ```

2. **Create a PersistentVolumeClaim** (`pvc.yaml`):

   ```
   apiVersion: v1
   kind: PersistentVolumeClaim
   metadata:
     name: pv-claim
   spec:
     accessModes:
       - ReadWriteOnce
     resources:
       requests:
         storage: 1Gi
   ```

3. **Apply the PersistentVolume and PersistentVolumeClaim**:

   ```
   kubectl apply -f pv.yaml
   kubectl apply -f pvc.yaml
   ```

4. **Use PersistentVolumeClaim in a Pod**:

   ```
   apiVersion: v1
   kind: Pod
   metadata:
     name: pv-pod
   spec:
     containers:
     - name: mycontainer
       image: busybox
       command: [ "sh", "-c", "echo 'Hello World' > /mnt/data/hello.txt
   && sleep 3600" ]
       volumeMounts:
       - mountPath: "/mnt/data"
         name: mypvc
     volumes:
     - name: mypvc
       persistentVolumeClaim:
         claimName: pv-claim
   ```

5. **Apply the Pod configuration**:

```
kubectl apply -f pod-pv.yaml
```

6. **Verify the Pod**:

```
kubectl exec -it pv-pod -- cat /mnt/data/hello.txt
```

Networking in Kubernetes

Kubernetes networking allows communication between Pods, services, and external resources.

Network Policies

Network Policies allow you to control the traffic flow between Pods.

1. **Create a Network Policy** (`network-policy.yaml`):

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-nginx
spec:
  podSelector:
    matchLabels:
      app: nginx
  ingress:
  - from:
    - podSelector:
        matchLabels:
          access: "true"
```

2. **Apply the Network Policy**:

```
kubectl apply -f network-policy.yaml
```

3. **Verify the Network Policy**:

```
kubectl get networkpolicies
```

Monitoring and Logging

Monitoring and logging are crucial for maintaining the health and performance of your applications.

Prometheus and Grafana

1. **Install Prometheus and Grafana using Helm**:

```
helm repo add prometheus-community https://prometheus-
community.github.io/helm-charts
helm repo update
helm install prometheus prometheus-community/prometheus
helm install grafana prometheus-community/grafana
```

2. **Access Grafana**:

```
kubectl get svc --namespace default -w grafana
```

3. **Set up Prometheus as a data source in Grafana and create dashboards to visualize metrics.**

ELK Stack (Elasticsearch, Logstash, Kibana)

1. **Install Elasticsearch and Kibana using Helm**:

```
helm repo add elastic https://helm.elastic.co
helm repo update
helm install elasticsearch elastic/elasticsearch
helm install kibana elastic/kibana
```

2. **Configure Logstash to collect logs from your applications and send them to Elasticsearch.**
3. **Access Kibana**:

```
kubectl get svc --namespace default -w kibana
```

4. **Use Kibana to visualize and analyze logs.**

Helm: Kubernetes Package Manager

Helm simplifies the deployment and management of applications on Kubernetes.

1. **Install Helm**:

```
curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 | bash
```

2. **Add a Helm Repository**:

```
helm repo add stable https://charts.helm.sh/stable
helm repo update
```

3. **Install an Application**:

```
helm install my-release stable/nginx
```

4. **Upgrade an Application**:

```
helm upgrade my-release stable/nginx
```

5. **Uninstall an Application**:

```
helm uninstall my-release
```

CI/CD with Kubernetes

Integrating CI/CD with Kubernetes enables automated deployments and continuous delivery of applications.

Jenkins

1. **Install Jenkins using Helm**:

   ```
   helm repo add jenkins https://charts.jenkins.io
   helm repo update
   helm install jenkins jenkins/jenkins
   ```

2. **Access Jenkins**:

   ```
   kubectl get svc --namespace default -w jenkins
   ```

3. **Configure Jenkins to build and deploy your applications to Kubernetes.**

GitLab CI/CD

1. **Create a `.gitlab-ci.yml` file**:

   ```
   stages:
     - build
     - deploy

   build:
     stage: build
     script:
       - docker build -t my-app:$CI_COMMIT_SHA .

   deploy:
     stage: deploy
     script:
       - kubectl apply -f deployment.yaml
   ```

2. **Push the configuration to your GitLab repository and set up runners to execute the CI/CD pipeline.**

Best Practices for Kubernetes

1. **Namespace Isolation**: Use namespaces to isolate environments (e.g., dev, test, prod).
2. **Resource Requests and Limits**: Define resource requests and limits for your Pods to ensure efficient resource utilization.
3. **Readiness and Liveness Probes**: Use readiness and liveness probes to monitor the health of your applications.
4. **Auto-scaling**: Use Horizontal Pod Autoscaler (HPA) to automatically scale your applications based on metrics.

5. **Security**: Implement Role-Based Access Control (RBAC) and network policies to secure your cluster.
6. **Backup and Disaster Recovery**: Regularly back up your etcd database and implement disaster recovery procedures.
7. **Logging and Monitoring**: Ensure comprehensive logging and monitoring to maintain the health and performance of your applications.

Conclusion

Kubernetes is a powerful platform for managing containerized applications. This guide has provided a comprehensive overview of Kubernetes, from setting up a cluster to deploying applications, scaling, updating, and managing configurations and secrets. By following these hands-on examples and best practices, you can effectively utilize Kubernetes to streamline your development and operations workflows.