.NET Microservices – Azure DevOps and AKS

Section 13: AKS - Notes

Introduction to Kubernetes

Kubernetes (K8s) is an open-source platform designed to automate the deployment, scaling, and operation of containerized applications. It provides a robust framework for managing microservices architectures and large-scale applications.

What is Kubernetes?

- **Definition**: Kubernetes is a container orchestration platform that enables developers to automate the management of containerized applications. Containers encapsulate the application and its dependencies, making them portable and consistent across environments. Kubernetes helps to manage these containers in a clustered environment.
- **Goals**:
    - **Automate Deployment**: Kubernetes automates the deployment of containers to ensure that applications are distributed across the cluster as intended.
    - **Scaling**: Automatically scale applications based on demand, adding or removing instances of services as needed.
    - **Maintain Availability**: Ensure applications are highly available and resilient to failures.

Key Concepts

- **Cluster**:
    - A Kubernetes cluster consists of a set of nodes that run containerized applications. The cluster is managed by a master node (or control plane) which coordinates the worker nodes that run the application containers.
    - **Components**:
        - **Master Node**: Controls the Kubernetes cluster. It includes:
            - **API Server**: Manages API requests, validates, and processes them.
            - **Controller Manager**: Manages controllers that handle routine tasks, such as scaling deployments or managing replicas.
            - **Scheduler**: Assigns Pods to nodes based on resource availability.
            - **etcd**: A distributed key-value store used for storing cluster state and configuration.

        - **Worker Nodes**: Run application containers and include:
            - **Kubelet**: An agent that ensures containers are running as expected.
            - **Kube Proxy**: Handles network routing and load balancing for Pods.
            - **Container Runtime**: Software (like Docker) that runs containers.

- **Pod**:
    - The smallest deployable unit in Kubernetes. A Pod can contain one or more containers that share the same network namespace and storage. Pods are designed to run a single instance of a specific application.
- **Service**:
    - An abstraction that defines a logical set of Pods and a policy for accessing them. Services provide a stable endpoint for accessing Pods, which might be dynamic and change over time.
- **Deployment**:
    - Manages the deployment and scaling of Pods. It ensures the desired number of Pods are running and can perform rolling updates and rollbacks to maintain application consistency.
- **ConfigMap and Secret**:
    - **ConfigMap**: Manages non-sensitive configuration data. It allows you to decouple configuration artifacts from container images.
    - **Secret**: Manages sensitive data such as passwords, OAuth tokens, and SSH keys. Secrets are encoded in Base64 to prevent casual inspection.

Benefits of Kubernetes

- **High Availability**:
    - Kubernetes ensures that applications are always available by automatically distributing the load and replicating Pods across multiple nodes. If a node fails, Kubernetes reschedules Pods to other healthy nodes.
- **Self-Healing**:
    - Automatically replaces or restarts failed containers. It ensures that the desired state of the application is maintained even if failures occur.
- **Automated Rollouts and Rollbacks**:
    - Provides mechanisms for deploying application updates gradually and rolling back changes if something goes wrong. This helps in managing application updates with minimal disruption.

- **Scalability**:
    - Kubernetes allows horizontal scaling of applications by adding or removing Pods based on load. It supports both manual and automatic scaling based on predefined metrics.
- **Declarative Configuration**:
    - Allows you to describe the desired state of the system using YAML or JSON files. Kubernetes ensures that the current state matches the desired state specified in these configuration files.
- **Service Discovery and Load Balancing**:
    - Kubernetes automatically assigns a unique IP address and DNS name to each Service. It handles load balancing between Pods and provides service discovery mechanisms.

Use Cases

- **Microservices Architecture**:
    - Kubernetes is particularly suited for managing microservices-based applications where services are decoupled and interact through APIs. It helps in managing the deployment, scaling, and monitoring of these services.
- **CI/CD Pipelines**:
    - Integrates well with continuous integration and continuous deployment (CI/CD) systems. Kubernetes can automate the deployment of applications as part of the CI/CD pipeline.
- **Hybrid Cloud Deployments**:
    - Supports hybrid cloud environments where applications are deployed across on-premises and cloud-based infrastructure.

Key Points to Remember (For Interview Preparation)

1. **Kubernetes Overview**: Understand the role of Kubernetes in container orchestration and its ability to automate deployment, scaling, and operation of applications.
2. **Cluster Architecture**: Be familiar with the components of a Kubernetes cluster, including the master node, worker nodes, and their roles.
3. **Core Concepts**: Know the basic Kubernetes objects such as Pods, Services, and Deployments, and how they interact.
4. **Benefits**: Recognize the advantages of using Kubernetes, including high availability, self-healing, scalability, and automated updates.
5. **Use Cases**: Be able to explain typical scenarios where Kubernetes is beneficial, such as microservices architecture and CI/CD pipelines.

Problems with Kubernetes

While Kubernetes provides powerful capabilities for managing containerized applications, it comes with its own set of challenges and limitations. Understanding these issues is crucial for effectively implementing and operating Kubernetes in a real-world environment.

1. Complexity and Learning Curve

- **Steep Learning Curve**:
  - Kubernetes introduces many new concepts and components (like Pods, Services, Deployments, etc.), which can be overwhelming for newcomers. The complexity of Kubernetes architecture demands significant time and effort to understand and operate effectively.
- **Complex Configuration**:
  - Kubernetes configurations (typically defined in YAML files) can be intricate and prone to human error. Debugging issues related to configuration can be time-consuming.
- **Diverse Ecosystem**:
  - The Kubernetes ecosystem includes various tools and add-ons (e.g., Helm for package management, Prometheus for monitoring). Managing and integrating these tools can add to the overall complexity.

2. Resource Consumption

- **High Resource Overhead**:
  - Kubernetes itself requires resources to run, including the master components and agents on each node. This overhead can be substantial, particularly in smaller clusters or on resource-constrained environments.
- **Efficient Resource Utilization**:
  - Ensuring efficient resource utilization (CPU, memory) within the cluster requires careful configuration of resource requests and limits. Poorly configured resource management can lead to inefficient use of cluster resources.

3. Operational Overhead

- **Cluster Management**:
  - Managing and maintaining a Kubernetes cluster involves regular updates, security patches, and backups. This operational overhead can be significant, especially for self-managed clusters.

- **Troubleshooting**:
    - Diagnosing issues in a Kubernetes cluster can be challenging due to the distributed nature of the system. Problems with Pods, Services, or networking can be hard to trace and resolve.

4. Security Concerns

- **Access Control**:
    - Kubernetes security requires proper configuration of Role-Based Access Control (RBAC) and Network Policies. Misconfigured access control can lead to security vulnerabilities.
- **Secrets Management**:
    - While Kubernetes provides mechanisms for managing sensitive information (Secrets), ensuring that secrets are handled securely and appropriately can be complex.
- **Pod Security**:
    - Securing Pods and containerized applications involves configuring security contexts and Pod Security Policies. Inadequate security configurations can expose applications to risks.

5. Networking Challenges

- **Complex Networking**:
    - Kubernetes networking involves managing internal and external traffic, Service discovery, and load balancing. Configuring and troubleshooting network issues can be complex.
- **Service Discovery and Load Balancing**:
    - Ensuring that Services are discovered correctly and load-balanced effectively across Pods can be challenging, especially in large or dynamic environments.

6. Persistent Storage

- **Stateful Applications**:
    - Managing persistent storage for stateful applications in Kubernetes can be complex. Kubernetes supports Persistent Volumes (PVs) and Persistent Volume Claims (PVCs), but configuring and managing storage can require additional effort.
- **Data Migration and Backup**:
    - Handling data migration and backup for applications running in Kubernetes needs to be carefully planned. Ensuring data durability and availability across cluster operations is crucial.

7. Integration with Existing Systems

- **Legacy Systems**:
    - Integrating Kubernetes with existing legacy systems and infrastructure can be challenging. Migrating applications from traditional environments to Kubernetes may require significant changes.
- **Interoperability**:
    - Ensuring interoperability between Kubernetes and other systems (e.g., databases, message brokers) can involve additional configuration and management.

8. Cost Management

- **Cloud Costs**:
    - Running Kubernetes on cloud platforms can incur substantial costs due to the need for multiple nodes, storage, and network resources. Effective cost management and optimization are essential.
- **Operational Costs**:
    - The operational costs of managing a Kubernetes cluster, including personnel, tools, and maintenance, can add up. Balancing these costs with the benefits is important for overall efficiency.

Key Points to Remember (For Interview Preparation)

1. **Complexity**: Be aware of the complexity of Kubernetes and its steep learning curve. Understand that configuring and managing Kubernetes requires significant expertise.
2. **Resource Overhead**: Recognize that Kubernetes introduces resource overhead and requires careful resource management to avoid inefficiencies.
3. **Operational Overhead**: Acknowledge the operational overhead involved in managing and maintaining a Kubernetes cluster, including troubleshooting and security.
4. **Security**: Understand the security concerns associated with Kubernetes, including access control, secrets management, and Pod security.
5. **Networking**: Be familiar with the challenges related to Kubernetes networking, Service discovery, and load balancing.
6. **Persistent Storage**: Know the complexities of managing persistent storage for stateful applications and data migration in Kubernetes.
7. **Integration and Costs**: Be aware of the challenges in integrating Kubernetes with existing systems and managing costs associated with cloud and operational resources.

Introduction to AKS

Azure Kubernetes Service (AKS) is a managed container orchestration service provided by Microsoft Azure that simplifies the deployment, management, and scaling of containerized applications using Kubernetes. AKS abstracts much of the complexity associated with Kubernetes management, allowing you to focus on developing and running your applications.

1. Overview of AKS

- **Managed Kubernetes Service**:
  - AKS provides a managed Kubernetes environment, handling many of the operational aspects of Kubernetes, such as the control plane and cluster maintenance, which are managed by Azure.
- **Integration with Azure Ecosystem**:
  - AKS integrates seamlessly with other Azure services, such as Azure Container Registry (ACR) for container image storage, Azure Monitor for logging and monitoring, and Azure Active Directory (AAD) for identity and access management.
- **Simplified Cluster Management**:
  - AKS reduces the operational burden of managing Kubernetes clusters by automating tasks like upgrades, scaling, and security patching.

2. Key Features of AKS

- **Automated Cluster Management**:
  - **Automatic Upgrades**: AKS automates Kubernetes version upgrades, ensuring that your cluster is always up to date with the latest features and security patches.
  - **Scaling**: AKS supports both manual and automatic scaling of nodes and pods to accommodate varying workloads.
- **Integrated Monitoring and Diagnostics**:
  - **Azure Monitor**: Provides comprehensive monitoring and logging capabilities, allowing you to gain insights into cluster performance and diagnose issues.
  - **Azure Log Analytics**: Collects and analyzes log data from your AKS cluster to help with troubleshooting and performance optimization.
- **Security and Compliance**:
  - **Azure Active Directory Integration**: Allows you to manage Kubernetes access and permissions using Azure AD identities and roles.
  - **Network Policies**: AKS supports Kubernetes Network Policies to control traffic between pods and enhance security.

- **Container Registry Integration**:
    - **Azure Container Registry (ACR)**: AKS integrates with ACR, allowing you to store and manage container images securely and efficiently.
- **Developer Productivity**:
    - **DevOps Integration**: AKS integrates with Azure DevOps and GitHub Actions for continuous integration and deployment (CI/CD) pipelines, enabling automated application deployments.
- **Cost Management**:
    - **Pay-as-You-Go**: AKS is billed based on the resources used, including virtual machines and storage. You only pay for the compute resources you consume, with no charge for the Kubernetes control plane.

3. Common Use Cases

- **Microservices Architectures**:
    - AKS is well-suited for deploying and managing microservices applications, providing scalability and isolation for different service components.
- **Dev/Test Environments**:
    - Developers and testers can use AKS to quickly spin up Kubernetes clusters for development and testing purposes, leveraging the managed nature of AKS to reduce setup and maintenance overhead.
- **Production Workloads**:
    - AKS supports running production-grade workloads with high availability and reliability, thanks to its integration with Azure's infrastructure and services.
- **Hybrid and Multi-Cloud Scenarios**:
    - AKS can be part of a hybrid cloud or multi-cloud strategy, allowing you to manage workloads across on-premises, Azure, and other cloud environments.

4. Getting Started with AKS

- **Creating an AKS Cluster**:
    - You can create an AKS cluster using the Azure Portal, Azure CLI, or ARM templates. The process involves specifying cluster configuration parameters such as node size, number of nodes, and Kubernetes version.
- **Accessing the Cluster**:
    - Once created, you can access the AKS cluster using kubectl, the Kubernetes command-line tool. You can configure kubectl to communicate with your AKS cluster using the Azure CLI or the Azure Portal.

- **Deploying Applications**:
    - Deploying applications to AKS involves creating Kubernetes manifests (Deployment, Service, ConfigMap, etc.) and applying them using kubectl. AKS provides a scalable and resilient environment for running containerized applications.

Key Points to Remember (For Interview Preparation)

1. **Managed Service**: AKS is a managed Kubernetes service that simplifies cluster management by handling operational aspects such as upgrades and scaling.
2. **Integration**: AKS integrates with other Azure services, including Azure Container Registry, Azure Monitor, and Azure Active Directory, enhancing its functionality and security.
3. **Automated Management**: Features like automatic upgrades, scaling, and integrated monitoring reduce operational overhead and improve cluster management.
4. **Cost-Effective**: AKS follows a pay-as-you-go pricing model, charging only for the resources you use without additional costs for the Kubernetes control plane.
5. **Use Cases**: AKS is ideal for microservices architectures, development and testing environments, production workloads, and hybrid/multi-cloud scenarios.
6. **Getting Started**: Creating and managing an AKS cluster involves using the Azure Portal, Azure CLI, or ARM templates, and deploying applications using Kubernetes manifests and kubectl.

Kubernetes Architecture

Understanding Kubernetes architecture is essential to grasp how AKS works, as AKS leverages this architecture to manage containerized applications at scale. Kubernetes architecture consists of several core components that enable it to schedule, manage, and orchestrate containers efficiently.

1. Key Components of Kubernetes

The Kubernetes architecture is divided into two primary sections: the **Control Plane** and the **Worker Nodes**. Let's explore each of these in detail.

1. Control Plane:

The control plane is responsible for managing the overall cluster and orchestrating the worker nodes. It consists of several components, each with a specific role in managing the Kubernetes cluster.

- **API Server (kube-apiserver)**:
    - The API Server is the entry point for all Kubernetes-related operations. It serves as the primary interface for users, applications, and administrators to interact with the Kubernetes cluster. All requests to the cluster (e.g., to create or delete resources like pods and services) go through the API server.
- **Etcd**:
    - Etcd is the key-value store used by Kubernetes to store all cluster data. This includes information about nodes, pods, services, configuration data, secrets, and more. Since Etcd stores the cluster's entire state, it plays a crucial role in ensuring the reliability and availability of the cluster.
- **Controller Manager (kube-controller-manager)**:
    - The controller manager runs multiple controllers that are responsible for monitoring the state of the cluster and making necessary adjustments to reach the desired state. For example, the ReplicaSet controller ensures that the correct number of pod replicas are running at any given time.
- **Scheduler (kube-scheduler)**:
    - The Kubernetes scheduler is responsible for assigning pods to nodes based on resource availability, policies, and constraints. It evaluates each pod’s resource requirements and schedules them on the appropriate worker nodes.

2. Worker Nodes:

Worker nodes are the machines (virtual or physical) where containers are deployed. Each worker node runs several components that allow it to communicate with the control plane and manage the running containers.

- **Kubelet**:
    - The kubelet is the primary agent running on each worker node. It is responsible for communicating with the API server, receiving instructions, and ensuring that the containers specified in the pod definitions are running as expected on the node.
- **Kube-Proxy**:
    - The kube-proxy is a network proxy that maintains network rules on each worker node. It enables communication between different services and pods, both within the node and across nodes. It also ensures that the networking environment supports Kubernetes services.
- **Container Runtime**:
    - The container runtime (e.g., Docker or containerd) is the software that actually runs containers on the worker node. It handles the downloading of container images and manages the lifecycle of containers.

2. Interaction Between Components

The control plane components work in tandem with the worker node components to ensure that the desired state of the cluster is always achieved and maintained. Here’s how these interactions typically work:

- **Cluster State**:
  - The desired state of the cluster (e.g., the number of replicas, services, and configurations) is stored in the etcd database.
- **API Requests**:
  - Administrators and developers interact with the Kubernetes API server to submit requests to the cluster, such as deploying new applications or scaling existing ones. The API server processes these requests and updates the etcd database.
- **Controllers**:
  - The controller manager continuously monitors the state stored in etcd. If the actual state of the cluster does not match the desired state (e.g., a pod is down or missing), the appropriate controller will take corrective actions, such as creating a new pod.
- **Scheduling**:
  - The scheduler monitors resource availability on the worker nodes and assigns pods to nodes that have the necessary resources to run the application. The kubelet on each worker node ensures that the assigned pods are running and communicates their status back to the API server.
- **Networking**:
  - Kube-proxy manages the network rules and ensures that services and pods can communicate with each other across nodes. It supports service discovery and load balancing.

3. Benefits of Kubernetes Architecture in AKS

- **Scalability**: Kubernetes can easily scale both horizontally (by adding more nodes) and vertically (by increasing node resources). AKS leverages this architecture to scale applications dynamically, ensuring they can handle varying loads.
- **Resilience**: By automatically rescheduling failed pods, AKS ensures high availability and resilience. The controller manager and scheduler work together to maintain the desired state of the application.
- **Modularity**: Kubernetes’ modular architecture allows you to use different components interchangeably. For example, you can switch between container runtimes or networking plugins as needed, which makes AKS adaptable to different use cases.
- **Automation**: The control plane components in Kubernetes automate most of the operational tasks, such as load balancing, scaling, and self-healing, which AKS extends to its managed services, reducing the administrative overhead.

4. AKS and Kubernetes Architecture

In AKS, the control plane is managed by Azure, meaning that Azure takes responsibility for maintaining the API server, etcd, scheduler, and controller manager. The worker nodes, however, are managed by you, allowing you to configure and scale them as necessary.

- **Managed Control Plane**:
    - AKS abstracts the complexity of managing the control plane, automating tasks like patching, upgrading, and scaling the control plane components.
- **Customizable Worker Nodes**:
    - You have full control over the worker nodes in AKS. You can specify the size, number, and configuration of these nodes based on your application requirements. You can also scale the nodes manually or use autoscaling features provided by AKS.

Key Points to Remember (For Interview Preparation)

1. **Control Plane vs Worker Nodes**: Kubernetes architecture is divided into the control plane (which manages the cluster) and worker nodes (where containers are deployed).
2. **Control Plane Components**: Key components like the API server, etcd, controller manager, and scheduler play crucial roles in ensuring the cluster's desired state is maintained.
3. **Worker Node Components**: The kubelet, kube-proxy, and container runtime (e.g., Docker) ensure that containers are running as expected on the worker nodes.
4. **AKS Managed Control Plane**: AKS simplifies Kubernetes management by offering a fully managed control plane while giving you control over worker nodes.
5. **Resilience and Automation**: Kubernetes' architecture enables self-healing, scalability, and automation, reducing the need for manual intervention in operational tasks.

5. Creating AKS Cluster (using CLI)

Creating an AKS (Azure Kubernetes Service) cluster using the Azure CLI involves a series of steps to set up and configure a Kubernetes cluster that is managed by Azure. This process allows you to deploy and manage containerized applications efficiently.

1. Prerequisites

Before creating an AKS cluster, ensure that you have the following:

- **Azure CLI**: Make sure the Azure CLI (az) is installed and up-to-date on your machine.
- **Azure Subscription**: An active Azure subscription to create and manage resources.
- **Resource Group**: An existing or new Azure resource group where your AKS cluster will be created.

Install Azure CLI:

```
# For Windows:

msiexec /i https://aka.ms/installazurecliwindows


# For macOS:

brew install azure-cli


# For Linux:

curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash
```

2. Authenticate to Azure

Before creating resources, log in to your Azure account using the Azure CLI:

```
az login
```

This command opens a web browser for authentication. Follow the prompts to log in.

3. Create a Resource Group

A resource group is a logical container for Azure resources. If you don't already have a resource group, create one using the Azure CLI:

```
az group create --name myResourceGroup --location eastus
```

Replace myResourceGroup with your desired resource group name and eastus with your preferred Azure region.

4. Create an AKS Cluster

To create an AKS cluster, use the az aks create command. This command allows you to specify various parameters for the cluster, including the name, resource group, and node configurations.

```
az aks create \
  --resource-group myResourceGroup \
  --name myAKSCluster \
  --node-count 3 \
  --enable-addons monitoring \
  --generate-ssh-keys
```

Parameters:

- --resource-group: The resource group where the AKS cluster will be created.
- --name: The name of the AKS cluster.
- --node-count: The number of nodes in the cluster.
- --enable-addons: Optional. Enables specific add-ons like monitoring.
- --generate-ssh-keys: Generates SSH keys for node access.

5. Verify Cluster Creation

After creating the AKS cluster, you can verify its status using:

```
az aks show --resource-group myResourceGroup --name myAKSCluster
```

To get detailed information about your cluster, including its version and configuration, use this command.

6. Connect to the AKS Cluster

To manage your AKS cluster, configure kubectl to use the cluster:

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

This command downloads the cluster credentials and configures kubectl to interact with your AKS cluster.

7. Verify kubectl Configuration

Ensure that kubectl is correctly configured to interact with your AKS cluster:

```
kubectl get nodes
```

This command lists all nodes in the cluster, confirming that your kubectl configuration is working.

Key Points to Remember (For Interview Preparation)

1. **Azure CLI Commands**: Familiarize yourself with the az aks create command and its parameters for creating and configuring AKS clusters.
2. **Resource Group**: Understand the role of resource groups in organizing Azure resources and how to create them.
3. **Cluster Configuration**: Be aware of key parameters like node count and add-ons when creating an AKS cluster.
4. **Connecting to AKS**: Know how to use az aks get-credentials to configure kubectl for managing your AKS cluster.
5. **Verification**: Use commands like az aks show and kubectl get nodes to verify the cluster creation and connectivity.

All Important Commands of kubectl

kubectl is the command-line tool for interacting with Kubernetes clusters. It allows you to manage your applications and clusters efficiently. Below is a detailed overview of some essential kubectl commands, categorized by their use cases:

1. Cluster Management

- **Check Cluster Info**

```
kubectl cluster-info
```

Displays information about the Kubernetes cluster, including the master and services URLs.

- **Get Cluster Nodes**

```
kubectl get nodes
```

Lists all nodes in the cluster, providing their status and roles.

- **Get Cluster Version**

```
kubectl version
```

Shows the Kubernetes client and server versions.

2. Pod Management

- **List Pods**

```
kubectl get pods
```

Lists all pods in the current namespace. Use --all-namespaces to list pods across all namespaces.

- **Describe Pod**

```
kubectl describe pod <pod-name>
```

Provides detailed information about a specific pod, including events and status.

- **Get Pod Logs**

```
kubectl logs <pod-name>
```

Retrieves logs from a specific pod. Use --container <container-name> if the pod has multiple containers.

- **Execute Command in Pod**

```
kubectl exec -it <pod-name> -- <command>
```

Executes a command inside a specific pod. The -it flag is used for interactive commands.

- **Delete Pod**

```
kubectl delete pod <pod-name>
```

Deletes a specific pod. This is useful for restarting a pod by forcefully removing it.

3. Deployment Management

- **Create Deployment**

```
kubectl create deployment <deployment-name> --image=<image-name>
```

Creates a new deployment with the specified name and container image.

- **List Deployments**

```
kubectl get deployments
```

Lists all deployments in the current namespace.

- **Scale Deployment**

```
kubectl scale deployment <deployment-name> --replicas=<number>
```

Scales the number of replicas for a deployment to the specified number.

- **Update Deployment**

```
kubectl apply -f <deployment-file>.yaml
```

Applies changes to a deployment based on a YAML configuration file.

- **Delete Deployment**

```
kubectl delete deployment <deployment-name>
```

Deletes a specific deployment along with its pods.

4. Service Management

- **List Services**

```
kubectl get services
```

Lists all services in the current namespace.

- **Describe Service**

```
kubectl describe service <service-name>
```

Provides detailed information about a specific service, including its endpoints.

- **Create Service**

```
kubectl expose deployment <deployment-name> --port=<port> --target-port=<target-port> --name=<service-name>
```

Creates a service that exposes a deployment on the specified ports.

- **Delete Service**

```
kubectl delete service <service-name>
```

Deletes a specific service.

5. Namespace Management

- **List Namespaces**

```
kubectl get namespaces
```

Lists all namespaces in the cluster.

- **Create Namespace**

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

Creates a new namespace.

- **Delete Namespace**

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

Deletes a specific namespace and its associated resources.

6. Config Management

- **View Config Maps**

```
kubectl get configmaps
```

Lists all config maps in the current namespace.

- **Describe Config Map**

```
kubectl describe configmap <configmap-name>
```

Provides detailed information about a specific config map.

- **Create Config Map**

```
kubectl create configmap <configmap-name> --from-literal=<key>=<value>
```

Creates a config map with the specified key-value pair.

- **Delete Config Map**

```
kubectl delete configmap <configmap-name>
```

Deletes a specific config map.

7. Secret Management

- **List Secrets**

```
kubectl get secrets
```

Lists all secrets in the current namespace.

- **Create Secret**

```
kubectl create secret generic <secret-name> --from-literal=<key>=<value>
```

Creates a new secret with the specified key-value pair.

- **Describe Secret**

```
kubectl describe secret <secret-name>
```

Provides detailed information about a specific secret.

- **Delete Secret**

```
kubectl delete secret <secret-name>
```

Deletes a specific secret.

8. Troubleshooting and Debugging

- **Get Pod Events**

```
kubectl get events
```

Lists recent events in the cluster that can help in debugging issues.

- **Get Pod Metrics**

kubectl top pod

Shows resource usage (CPU and memory) of pods. Requires metrics-server to be installed.

- **Describe Node**

kubectl describe node <node-name>

Provides detailed information about a specific node, including resource usage and status.

Key Points to Remember (For Interview Preparation)

1. **Basic Commands**: Understand the basic kubectl commands for managing pods, deployments, and services.
2. **Pod Management**: Be able to list, describe, and interact with pods, and retrieve logs.
3. **Deployment Management**: Familiarize yourself with creating, scaling, and deleting deployments.
4. **Service Management**: Know how to expose deployments via services and manage them.
5. **Namespace and Config Management**: Understand how to manage namespaces and config maps/secrets.
6. **Troubleshooting**: Be aware of commands for troubleshooting issues in the cluster, including viewing events and metrics.

Deployment Manifests

A deployment manifest is a YAML file that defines the desired state of your application and how Kubernetes should manage it. Here's a detailed breakdown of key components and how to create a deployment manifest:

1. Basic Structure of a Deployment Manifest

A deployment manifest typically includes the following sections:

1. **apiVersion**: Specifies the Kubernetes API version to use for the deployment. For deployments, this is usually apps/v1.
2. **kind**: Defines the type of Kubernetes resource being created. For deployments, this is Deployment.
3. **metadata**: Contains metadata about the deployment, such as its name, labels, and namespace.

4. **spec**: Defines the desired state of the deployment, including the pod template and strategy for managing updates.

Example:

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

2. Key Sections Explained

- **apiVersion**:
  - Specifies the version of the Kubernetes API you're using. For deployments, use apps/v1.
- **kind**:
  - Indicates the type of resource. For deploying applications, use Deployment.

- **metadata**:
    - **name**: Name of the deployment.
    - **labels**: Labels used to identify and select resources. These are crucial for service selectors and pod management.
- **spec**:
    - **replicas**: The number of pod replicas to maintain. Kubernetes ensures that this number of pods is always running.
    - **selector**: A label selector used to identify the pods managed by this deployment. It must match the labels defined in the pod template.
    - **template**:
        - **metadata**:
            - **labels**: Labels for the pods created by this deployment. They must match the selector.
        - **spec**:
            - **containers**: Defines the containers within each pod.
                - **name**: The name of the container.
                - **image**: The Docker image to use for the container.
                - **ports**: Ports exposed by the container.

3. Example Deployment Manifest

Here is a complete example of a deployment manifest:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-web-app
  labels:
    app: my-web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-web-app
```

```
 template:
  metadata:
   labels:
    app: my-web-app
  spec:
   containers:
   - name: web
    image: my-web-app:1.0
    ports:
    - containerPort: 80
    env:
    - name: ASPNETCORE_ENVIRONMENT
     value: "Production"
```

Explanation:

- **metadata.name**: my-web-app - The name of the deployment.
- **spec.replicas**: 3 - Three replicas of the pod will be maintained.
- **spec.selector.matchLabels**: app: my-web-app - This selector will match the pods with the label app: my-web-app.
- **spec.template.metadata.labels**: app: my-web-app - Labels for the pods to match the selector.
- **spec.template.spec.containers**:
  - **name**: web - Name of the container.
  - **image**: my-web-app:1.0 - Docker image used.
  - **ports**: Exposes port 80 on the container.
  - **env**: Defines an environment variable ASPNETCORE_ENVIRONMENT with value "Production".

4. Managing Deployments

- **Apply Deployment Manifest**

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

Applies the deployment manifest to create or update resources in the cluster.

- **View Deployment Status**

kubectl rollout status deployment/my-web-app

Checks the rollout status of a deployment.

- **Update Deployment** Modify the deployment manifest and reapply it with kubectl apply -f deployment.yaml to update the deployment.
- **Rollback Deployment**

kubectl rollout undo deployment/my-web-app

Rolls back to the previous version of the deployment if there is an issue with the new version.

5. Best Practices

- **Version Tags**: Always use version tags for images to ensure that your deployments are predictable and reproducible.
- **Health Checks**: Consider adding livenessProbe and readinessProbe to ensure your application is running correctly and is ready to serve traffic.
- **Resource Limits**: Define resource limits and requests to ensure that your containers don't consume more resources than available.

Key Points to Remember (For Interview Preparation)

1. **Manifest Structure**: Understand the key sections of a deployment manifest (apiVersion, kind, metadata, spec).
2. **Pod Management**: Know how to configure replicas and manage pod templates.
3. **Updating Deployments**: Be familiar with updating and rolling back deployments.
4. **Best Practices**: Be aware of best practices for defining and managing deployments.

Service Manifests

A service manifest defines how to expose and access your applications running in Kubernetes. Services provide a stable IP address and DNS name to access your application, enabling communication between different parts of your application.

1. Basic Structure of a Service Manifest

A service manifest typically includes the following sections:

1. **apiVersion**: Specifies the Kubernetes API version to use for the service. For services, this is usually v1.
2. **kind**: Defines the type of Kubernetes resource being created. For services, this is Service.
3. **metadata**: Contains metadata about the service, such as its name and labels.
4. **spec**: Defines the desired state of the service, including type, selector, and ports.

Example:

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

2. Key Sections Explained

- **apiVersion**:
  - Specifies the version of the Kubernetes API you're using. For services, use v1.
- **kind**:
  - Indicates the type of resource. For exposing applications, use Service.
- **metadata**:
  - **name**: Name of the service.
  - **labels**: Labels used to identify and manage the service.

- **spec**:
  - **selector**: A label selector used to identify the pods that the service will route traffic to. It must match the labels defined in the pod template.
  - **ports**:
    - **protocol**: Protocol used by the service, typically TCP.
    - **port**: The port on which the service is exposed.
    - **targetPort**: The port on the container to which traffic is forwarded.
  - **type**: Type of service. Common types include:
    - **ClusterIP** (default): Exposes the service on a cluster-internal IP. Only accessible within the cluster.
    - **NodePort**: Exposes the service on each node's IP at a static port. Accessible externally via <NodeIP>:<NodePort>.
    - **LoadBalancer**: Exposes the service externally using a cloud provider's load balancer. Provides a stable external IP address.
    - **ExternalName**: Maps the service to an external DNS name.

3. Example Service Manifest

Here's a more detailed example of a service manifest:

```
apiVersion: v1

kind: Service

metadata:

  name: my-web-service

  labels:

    app: my-web-app

spec:

  selector:

    app: my-web-app

  ports:

    - protocol: TCP

      port: 80

      targetPort: 8080

  type: NodePort
```

Explanation:

- **metadata.name**: my-web-service - The name of the service.
- **spec.selector**: app: my-web-app - The service routes traffic to pods with the label app: my-web-app.
- **spec.ports**:
    - **port**: 80 - The port exposed by the service.
    - **targetPort**: 8080 - The port on the container where the service directs traffic.
- **spec.type**: NodePort - The service is exposed on a static port on each node.

4. Managing Services

- **Apply Service Manifest**

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

Applies the service manifest to create or update services in the cluster.

- **View Service Details**
- kubectl get services

```
kubectl describe service/my-web-service
```

Lists all services and describes a specific service to show details.

- **Expose Deployment as a Service**

```
kubectl expose deployment my-web-app --type=LoadBalancer --port=80 --target-port=8080
```

Creates a service for an existing deployment.

- **Delete Service**

```
kubectl delete service/my-web-service
```

Deletes the specified service.

5. Best Practices

- **Use Labels and Selectors**: Ensure the selector in your service manifest matches the labels of the pods to route traffic correctly.
- **Choose Service Type Wisely**: Select the service type based on your needs (e.g., ClusterIP for internal services, LoadBalancer for external access).
- **Security Considerations**: Be aware of the security implications of exposing services, especially with types like NodePort and LoadBalancer.

Key Points to Remember (For Interview Preparation)

1. **Service Structure**: Understand the key sections of a service manifest (apiVersion, kind, metadata, spec).
2. **Service Types**: Know the different service types (ClusterIP, NodePort, LoadBalancer, ExternalName) and when to use each.
3. **Service Management**: Be familiar with creating, updating, and managing services.
4. **Selector and Labels**: Understand how selectors are used to route traffic to the appropriate pods.

Troubleshooting Pods

Troubleshooting Kubernetes pods involves diagnosing and resolving issues related to pod creation, execution, and functionality. Here's a comprehensive guide to help you troubleshoot pods effectively.

1. Checking Pod Status

- **List Pods**:

```
kubectl get pods
```

This command shows the list of all pods and their statuses. Look for pods that are not in the Running state.

- **Describe Pod**:

```
kubectl describe pod <pod-name>
```

Provides detailed information about a specific pod, including events and error messages.

- **View Pod Logs**:

```
kubectl logs <pod-name>
```

Displays the logs from a container within the pod. Useful for debugging application-level issues.

For a specific container in a pod:

```
kubectl logs <pod-name> -c <container-name>
```

2. Common Pod Issues and Solutions

- **Pod Statuses**:
    - **Pending**: The pod is waiting for resources to be allocated.
        - **Check Events**: Use kubectl describe pod <pod-name> to view events and see why the pod is pending.
        - **Check Resource Requests**: Ensure that your cluster has enough resources to fulfill the pod's requests.
    - **CrashLoopBackOff**: The pod is repeatedly crashing and restarting.
        - **Examine Logs**: Use kubectl logs <pod-name> to check the logs for errors.
        - **Check Liveness and Readiness Probes**: Misconfigured probes can cause restarts.
    - **Error**: The pod is unable to start due to errors.
        - **Check Configuration**: Review the pod's YAML configuration for mistakes.
        - **Examine Logs**: Logs can provide clues about the issue.
- **Inspecting Container State**:

```
kubectl get pod <pod-name> -o=jsonpath='{.status.containerStatuses[*].state}'
```

This command provides the state of all containers within the pod.

- **Checking Resource Utilization**:

```
kubectl top pod <pod-name>
```

Shows the CPU and memory usage of a pod. High resource usage can cause performance issues.

3. Debugging Tools

- **kubectl exec**:

```
kubectl exec -it <pod-name> -- /bin/sh
```

Opens a shell inside a running pod, allowing you to run commands and inspect the file system.

- **kubectl port-forward**:

```
kubectl port-forward <pod-name> <local-port>:<container-port>
```

Forwards a port from your local machine to a pod's port, useful for accessing applications running inside the pod.

- **kubectl cp**:

```
kubectl cp <local-file> <pod-name>:<container-path>
```

Copies files between your local machine and a container in a pod.

4. Troubleshooting Techniques

- **Logs Analysis**: Start by checking the logs for error messages or stack traces that can provide clues about the problem.
- **Events Inspection**: Use kubectl describe to view events related to the pod, which can help identify issues like scheduling problems or resource constraints.
- **Configuration Review**: Ensure that the pod's configuration (e.g., resource requests, environment variables, volumes) is correct and matches your expectations.
- **Resource Monitoring**: Check resource usage to ensure that the pod has enough CPU and memory. Use kubectl top and monitor metrics from the cluster.

5. Example Scenarios

- **Pod Not Starting**:
  - **Problem**: Pod is stuck in Pending state.
  - **Solution**: Check for resource availability. Ensure there are no unsatisfied resource requests. Review events with kubectl describe pod.
- **Application Crashing**:
  - **Problem**: Container is repeatedly crashing (CrashLoopBackOff).
  - **Solution**: Examine application logs with kubectl logs. Check for application errors or misconfigurations. Adjust liveness and readiness probes if necessary.
- **Network Issues**:
  - **Problem**: Application is not accessible.
  - **Solution**: Verify service and endpoint configuration. Use kubectl port-forward to test connectivity. Check network policies and firewall rules.

Key Points to Remember (For Interview Preparation)

1. **Pod Statuses**: Understand the meaning of different pod statuses (Pending, Running, CrashLoopBackOff, Error).
2. **Troubleshooting Commands**: Be familiar with kubectl get pods, kubectl describe pod, kubectl logs, and kubectl exec.
3. **Common Issues**: Know common pod issues and how to diagnose them (e.g., pending pods, crashing containers).
4. **Resource Management**: Monitor and manage pod resources to prevent issues related to resource constraints.

Zero Downtime Rollout

Zero Downtime Rollout refers to deploying new versions of an application without interrupting the service or causing downtime for users. This approach ensures continuous availability and minimizes disruption.

1. What is Zero Downtime Rollout?

Zero Downtime Rollout is a deployment strategy where new versions of applications are rolled out gradually, ensuring that the system remains available and operational during the update. This technique is crucial for high-availability systems where even short downtimes can affect user experience and business operations.

2. Strategies for Zero Downtime Rollout

1. Blue-Green Deployment

- **Concept**: Involves running two identical production environments (Blue and Green). At any time, only one environment serves the live traffic.
- **Process**:
  - Deploy the new version of the application to the idle environment (e.g., Green).
  - Test the new version in the Green environment.
  - Switch traffic from the Blue environment to the Green environment.
  - The Blue environment becomes idle and can be used for the next deployment.
- **Advantages**:
  - Easy rollback: Switch back to the previous environment if issues arise.
  - Minimal disruption: Users experience no downtime during the switch.

2. Rolling Updates

- **Concept**: Gradually updates instances of an application one by one.
- **Process**:
  - Update a small subset of instances (e.g., one pod) at a time.
  - Ensure that the new version is healthy before updating the next subset.
  - Continue this process until all instances are updated.

- **Advantages**:
    - Gradual deployment reduces risk by isolating issues to a small number of instances.
    - Continuous availability of the service.

3. Canary Releases

- **Concept**: Deploy the new version to a small subset of users or instances before a full rollout.
- **Process**:
    - Deploy the new version to a small portion of the production environment (e.g., 5% of instances).
    - Monitor the performance and user feedback.
    - Gradually increase the rollout percentage if the new version is stable.
- **Advantages**:
    - Limited exposure of potential issues.
    - Allows for real-world testing with a controlled group.

4. Feature Toggles (Flags)

- **Concept**: Use feature flags to control the visibility of new features in the application.
- **Process**:
    - Deploy the new version with feature flags turned off.
    - Gradually enable the new features by toggling flags on.
- **Advantages**:
    - Enables feature-level rollouts without deploying new code.
    - Provides the ability to roll back specific features without affecting the entire deployment.

3. Implementing Zero Downtime in Kubernetes

- **Rolling Updates in Kubernetes**:
    - Kubernetes supports rolling updates natively through its deployment objects.
    - **Deployment Manifest Example**:

      apiVersion: apps/v1

      kind: Deployment

```
metadata:
  name: my-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app-container
        image: my-app:latest
        ports:
- containerPort: 80
```

  - Kubernetes updates pods one at a time to ensure that at least some instances are always available.

- **Readiness Probes**:
  - Use readiness probes to ensure that new pods are only added to the service once they are ready.
  - **Readiness Probe Example**:

```
readinessProbe:
  httpGet:
    path: /health
    port: 80
  initialDelaySeconds: 5
  periodSeconds: 10
```

- **Rolling Update Strategy**:
  - Define the rolling update strategy in the deployment manifest.
  - **Strategy Example**:

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1
    maxUnavailable: 1
```

4. Monitoring and Rollback

- **Monitoring**:
  - Monitor application metrics and logs during the rollout to detect issues early.
  - Use tools like Prometheus and Grafana for real-time monitoring.
- **Rollback**:
  - Ensure that rollback procedures are in place to revert to the previous version if issues are detected.
  - Kubernetes supports rollback commands:

```
kubectl rollout undo deployment/my-app
```

5. Best Practices for Zero Downtime Rollout

- **Automated Testing**: Perform automated tests to verify that the new version works correctly before deploying.
- **Monitoring and Alerts**: Set up monitoring and alerts to detect and respond to issues quickly.
- **Incremental Rollouts**: Use canary releases or rolling updates to deploy changes incrementally.
- **Backup and Recovery**: Ensure that you have backup and recovery procedures in place in case of deployment failures.

Key Points to Remember (For Interview Preparation)

1. **Deployment Strategies**: Understand different deployment strategies like Blue-Green Deployment, Rolling Updates, Canary Releases, and Feature Toggles.
2. **Kubernetes Rollouts**: Familiarize yourself with how Kubernetes handles rolling updates and readiness probes.
3. **Monitoring and Rollback**: Be aware of tools and practices for monitoring deployments and rolling back if necessary.

Kubernetes Secrets

Kubernetes Secrets are used to store and manage sensitive information, such as passwords, OAuth tokens, SSH keys, and other confidential data. Using Secrets helps keep sensitive information separate from application code and configuration.

1. What are Kubernetes Secrets?

Kubernetes Secrets allow you to securely store and manage sensitive data within your Kubernetes cluster. Secrets are encoded and can be referenced in your pods and deployments without exposing the actual values in plaintext.

2. Types of Secrets

1. Opaque Secrets

- General-purpose secrets that contain arbitrary user-defined data.
- Example use case: storing database passwords.

2. Docker Registry Secrets

- Used for authenticating with private Docker registries.
- Example use case: pulling private Docker images.

3. TLS Secrets

- Used to store TLS certificates and private keys.
- Example use case: securing communication between services.

4. Service Account Tokens

- Automatically created for service accounts.
- Example use case: providing credentials for API access.

3. Creating Secrets

Secrets can be created using kubectl commands or YAML manifests.

1. Creating Secrets with kubectl

- **Command Example**:

```
kubectl create secret generic my-secret --from-literal=username=myuser --from-literal=password=mypassword
```

- my-secret: Name of the Secret.
- --from-literal: Adds literal values to the Secret.

- **Creating a Secret from a File**:

```
kubectl create secret generic my-secret --from-file=ssh-privatekey=/path/to/private.key
```

2. Creating Secrets using YAML

- **YAML Example**:
- apiVersion: v1
- kind: Secret
- metadata:
-   name: my-secret
- type: Opaque
- data:
-   username: bXl1c2Vy

  password: bXlwYXNzd29yZA==

  - data: The Secret data in base64 encoding. username and password are encoded base64 values.

4. Using Secrets in Pods

Secrets can be used in Pods as environment variables or mounted as files.

1. Using Secrets as Environment Variables

- **YAML Example**:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    env:
```

```
- name: DB_USERNAME
  valueFrom:
    secretKeyRef:
      name: my-secret
      key: username
- name: DB_PASSWORD
  valueFrom:
    secretKeyRef:
      name: my-secret
     key: password
```

2. Mounting Secrets as Files

- **YAML Example**:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    volumeMounts:
    - name: secret-volume
      mountPath: /etc/secret
      readOnly: true
  volumes:
  - name: secret-volume
    secret:
      secretName: my-secret
```

5. Managing Secrets

1. Viewing Secrets

- **Command Example**:

```
kubectl get secrets
```

- **Viewing Secret Details**:

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

- **Decoding Secret Data**:

```
kubectl get secret my-secret -o jsonpath='{.data.username}' | base64 --decode
```

2. Updating Secrets

- **Updating with kubectl**:

```
kubectl create secret generic my-secret --from-literal=username=newuser --from-
literal=password=newpassword --dry-run=client -o yaml | kubectl apply -f -
```

- **Updating with YAML**: Modify the existing YAML file and apply the changes:

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

3. Deleting Secrets

- **Command Example**:

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

6. Best Practices for Managing Secrets

- **Avoid Hardcoding Secrets**: Use Kubernetes Secrets to manage sensitive data instead of hardcoding them in your application code.
- **Use RBAC**: Implement Role-Based Access Control (RBAC) to restrict access to secrets.
- **Limit Secret Scope**: Ensure that secrets are only accessible to the necessary components.
- **Regularly Rotate Secrets**: Regularly update and rotate secrets to minimize security risks.

Key Points to Remember (For Interview Preparation)

1. **Types of Secrets**: Understand the different types of Kubernetes Secrets and their use cases.
2. **Creating and Managing Secrets**: Familiarize yourself with creating, viewing, updating, and deleting secrets.
3. **Using Secrets in Pods**: Know how to use secrets as environment variables and mounted files.
4. **Best Practices**: Be aware of best practices for managing secrets securely in Kubernetes.