Kubernetes: Most Common Objects and Concepts

Introduction

Kubernetes is a powerful container orchestration platform that abstracts the underlying infrastructure and provides a unified interface for deploying, managing, and scaling containerized applications. This comprehensive reference guide covers the most frequently used Kubernetes objects and concepts that form the foundation of modern cloud-native deployments. Understanding these core components is essential for effectively building and managing production-grade Kubernetes environments.

1. Pods

Overview

A Pod is the smallest deployable unit in Kubernetes and represents a single instance of a running container or group of tightly coupled containers. Unlike Docker, where you directly manage containers, in Kubernetes you manage Pods, which wrap your containers with additional networking and storage capabilities.

Pod Characteristics

Container Encapsulation: Pods typically contain a single container but can hold multiple containers that need to work together. Containers within a Pod share network namespaces, meaning they share an IP address and can communicate via localhost.

Ephemeral Nature: Pods are temporary resources designed to be created and destroyed frequently. They don't have persistent identity by default, making them unsuitable for stateful applications without additional abstractions.

Networking: All containers in a Pod share the same network namespace, allowing communication via localhost. However, each Pod gets its own IP address within the cluster.

Basic Pod Example

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    app: my-app
spec:
  containers:
  - name: my-container
    image: nginx:latest
    ports:
    - containerPort: 80
```

Pod Operations

```
# Create a pod from YAML
kubectl apply -f pod.yaml

# Get pods in current namespace
kubectl get pods

# Describe pod details
kubectl describe pod my-pod

# View pod logs
kubectl logs my-pod

# Execute command in pod
kubectl exec -it my-pod -- /bin/bash

# Delete a pod
kubectl delete pod my-pod
```

2. ReplicaSets

Overview

A ReplicaSet maintains a stable set of replica Pods running at any given time. Its primary purpose is to ensure that a specified number of identical Pod replicas are always running. If a Pod fails, the ReplicaSet automatically creates a new one to maintain the desired state.

Key Features

Desired State Management: ReplicaSets continuously monitor the cluster and create or delete Pods to match the desired number of replicas specified in the configuration.

Pod Selector: Uses label selectors to identify which Pods to manage. This allows ReplicaSets to adopt existing Pods that match the selector criteria.

Automatic Replacement: When a Pod is deleted or fails, the ReplicaSet immediately creates a replacement Pod to maintain the desired replica count.

ReplicaSet Configuration

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
```

```
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
        ports:
        - containerPort: 80
```

ReplicaSet Management

```
# Create ReplicaSet
kubectl apply -f replicaset.yaml

# Scale ReplicaSet
kubectl scale replicaset frontend --replicas=5

# Get ReplicaSets
kubectl get replicasets

# Delete ReplicaSet and Pods
kubectl delete replicaset frontend
```

3. Deployments

Overview

A Deployment is a higher-level abstraction that manages ReplicaSets and provides declarative updates for Pods and ReplicaSets. It's the recommended way to deploy stateless applications in Kubernetes, offering rolling updates, rollbacks, and automatic rollout management.

Deployment Advantages

Declarative Updates: Define the desired state, and Kubernetes automatically handles the transition from the current state to the desired state.

Rolling Updates: Gradually replace old Pods with new ones, ensuring zero downtime deployments.

Automatic Rollback: Easily revert to previous versions if issues arise during an update.

Replica Management: Deployments automatically create and manage ReplicaSets.

Deployment Example

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
```

```
    app: nginx
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
          limits:
            cpu: 500m
            memory: 512Mi
```

Deployment Operations

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

# Check deployment status
kubectl rollout status deployment/nginx-deployment

# View deployment history
kubectl rollout history deployment/nginx-deployment

# Rollback to previous version
kubectl rollout undo deployment/nginx-deployment

# Scale deployment
kubectl scale deployment nginx-deployment --replicas=5

# Update deployment image
kubectl set image deployment/nginx-deployment \
  nginx=nginx:1.16.0 --record
```

4. Namespaces

Overview

Namespaces provide a mechanism to partition cluster resources between multiple users, teams, or applications. They offer logical isolation, allowing multiple virtual clusters to coexist within a single physical Kubernetes cluster.

Namespace Features

Logical Isolation: Namespaces create isolated environments where resources are separated by default.

Resource Quotas: Limit the amount of CPU, memory, storage, and other resources that a namespace can consume.

Access Control: Implement Role-Based Access Control at the namespace level to control who can access resources.

Multi-tenancy: Enable multiple teams or applications to share a single cluster while maintaining isolation.

Common Namespaces

```
# View system namespaces
kubectl get namespaces

# Default namespaces:
# - default: Where resources are created by default
# - kube-system: Kubernetes system components
# - kube-public: Publicly accessible cluster information
# - kube-node-lease: Node heartbeat data for availability
```

Namespace Management

```
apiVersion: v1
kind: Namespace
metadata:
  name: production
```

```
# Create namespace
kubectl create namespace production

# Set default namespace for context
kubectl config set-context --current --namespace=production

# Get resources in specific namespace
kubectl get pods -n production
```

```
# Create resource quota for namespace
kubectl apply -f - &lt;&lt;EOF
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-quota
  namespace: production
spec:
  hard:
    requests.cpu: "10"
    requests.memory: 20Gi
    limits.cpu: "20"
    limits.memory: 40Gi
    pods: "100"
EOF
```

5. Services

Overview

A Service is an abstract way to expose applications running on a set of Pods as a network service. Services provide stable network endpoints and load balancing for Pods, which have ephemeral IP addresses.

Service Types

ClusterIP: Default service type that exposes the service only within the cluster.

NodePort: Exposes the service on a static port on each node, allowing external access through any node's IP address.

LoadBalancer: Provisions an external load balancer to route traffic to the service.

ExternalName: Maps the service to an external DNS name without selecting any Pods.

ClusterIP Example

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

NodePort Example

```
apiVersion: v1
kind: Service
metadata:
  name: web-nodeport
spec:
  type: NodePort
  selector:
    app: web
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30080
```

Service Operations

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

# List services
kubectl get services

# Describe service
kubectl describe service web-service

# Port forwarding for testing
kubectl port-forward service/web-service 8080:80
```

6. Storage (PersistentVolumes, PersistentVolumeClaims, StorageClass)

PersistentVolumes

A PersistentVolume is a cluster-level resource that represents storage capacity provisioned by an administrator. PersistentVolumes exist independently of Pods and persist beyond a Pod's lifecycle.

PersistentVolumeClaims

A PersistentVolumeClaim is a request for storage by a user or application. PersistentVolumeClaims bind to available PersistentVolumes based on storage class, size, and access modes.

StorageClass

StorageClass enables dynamic provisioning of PersistentVolumes, allowing administrators to define classes of storage with different characteristics.

Storage Configuration Examples

```
# StorageClass for SSD storage
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-ssd
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer

---
# PersistentVolumeClaim
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-pvc
spec:
  storageClassName: fast-ssd
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi

---
# Pod using PersistentVolumeClaim
apiVersion: v1
kind: Pod
metadata:
  name: app-pod
spec:
  containers:
  - name: app
    image: my-app:latest
    volumeMounts:
    - name: data
      mountPath: /data
  volumes:
  - name: data
    persistentVolumeClaim:
      claimName: data-pvc
```

Storage Operations

```
# List storage classes
kubectl get storageclasses

# List PersistentVolumes
kubectl get pv

# List PersistentVolumeClaims
```

```
kubectl get pvc

# Describe PVC to check status
kubectl describe pvc data-pvc

# Delete PVC
kubectl delete pvc data-pvc
```

7. StatefulSets

Overview

StatefulSets manage stateful applications that require stable network identities and persistent storage. Unlike Deployments, which treat Pods as interchangeable, StatefulSets maintain unique, persistent identities for each Pod.

StatefulSet Characteristics

Stable Pod Identity: Pods are created sequentially with predictable names (for example, web-0, web-1, web-2).

Persistent Storage: Each Pod can have its own PersistentVolumeClaim, providing data persistence.

Ordered Operations: Scaling and updates happen in order, with ordered startup and termination.

Headless Service: Typically paired with a headless Service for stable DNS names.

StatefulSet Example

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  clusterIP: None
  selector:
    app: nginx
  ports:
  - port: 80
    name: web

---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 3
  selector:
    matchLabels:
      app: nginx
```

```
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "standard"
      resources:
        requests:
          storage: 1Gi
```

8. DaemonSets

Overview

A DaemonSet ensures that a copy of a Pod runs on all (or a subset of) nodes in the cluster. DaemonSets are typically used for cluster-wide services like logging agents, monitoring tools, and network plugins.

DaemonSet Characteristics

Node Coverage: Automatically adds Pods to new nodes and removes them from deleted nodes.

One Pod Per Node: By default, one Pod instance runs on each node.

Node Selection: Can be restricted to specific nodes using nodeSelector or affinity rules.

DaemonSet Example

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
```

```
      labels:
        name: fluentd-elasticsearch
    spec:
      containers:
      - name: fluentd-elasticsearch
        image: fluent/fluentd-kubernetes-daemonset:v1-debian-elasticsearch
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

DaemonSet Operations

```
# Create DaemonSet
kubectl apply -f daemonset.yaml

# List DaemonSets
kubectl get daemonsets

# Check DaemonSet status
kubectl rollout status daemonset/fluentd-elasticsearch

# Delete DaemonSet
kubectl delete daemonset fluentd-elasticsearch
```

9. Jobs and CronJobs

Jobs

A Job creates one or more Pods and ensures that a specified number of them successfully complete. Jobs are useful for batch processing, one-time tasks, and computation-based workloads.

CronJobs

CronJobs schedule Jobs to run at specific times using cron syntax.

Job Configuration

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi-job
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl:5.34
        command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
  parallelism: 2
  completions: 3
```

CronJob Configuration

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: backup-job
spec:
  schedule: "0 2 * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: backup
            image: busybox
            command: ["/bin/sh", "-c", "echo Backup started; sleep 10"]
          restartPolicy: OnFailure
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 1
```

Job Operations

```
# Create job
kubectl apply -f job.yaml

# List jobs
kubectl get jobs

# Check job status
kubectl describe job pi-job

# View job logs
kubectl logs job/pi-job
```

```
# Create CronJob
kubectl apply -f cronjob.yaml

# List CronJobs
kubectl get cronjobs

# Manually trigger CronJob
kubectl create job --from=cronjob/backup-job manual-backup
```

10. ConfigMaps and Secrets

ConfigMaps

ConfigMaps store non-confidential configuration data in key-value pairs, allowing decoupling of configuration from application code.

Secrets

Secrets store sensitive information like passwords and API keys with base64 encoding and stricter access controls.

ConfigMap Examples

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  database.url: "postgres://db:5432"
  debug: "false"
  log.level: "info"

---
# Using ConfigMap in Pod
apiVersion: v1
kind: Pod
metadata:
  name: config-pod
spec:
  containers:
  - name: app
    image: my-app:latest
    env:
    - name: DATABASE_URL
      valueFrom:
        configMapKeyRef:
          name: app-config
          key: database.url
    volumeMounts:
    - name: config
      mountPath: /etc/config
  volumes:
```

```
  - name: config
    configMap:
      name: app-config
```

Secret Examples

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  username: YWRtaW4=
  password: c2VjcmV0cGFzcw==

---
# Using Secret in Pod
apiVersion: v1
kind: Pod
metadata:
  name: secret-pod
spec:
  containers:
  - name: app
    image: my-app:latest
    env:
    - name: DB_USERNAME
      valueFrom:
        secretKeyRef:
          name: db-secret
          key: username
    - name: DB_PASSWORD
      valueFrom:
        secretKeyRef:
          name: db-secret
          key: password
```

11. Service Accounts

Overview

Service Accounts provide an identity for Pods running in the cluster. They enable authentication to the API server and allow Pods to interact with other Kubernetes resources securely. Every Pod runs under a Service Account, either explicitly specified or the default Service Account in the namespace.

Service Account Purpose

Pod Identity: Service Accounts identify Pods to the Kubernetes API server, allowing secure communication with the cluster.

Token Management: Each Service Account has an associated token that allows authentication to the API server.

RBAC Integration: Service Accounts can be bound to Roles and ClusterRoles to grant specific permissions to Pods.

Image Pull Secrets: Service Accounts can hold credentials for pulling container images from private registries.

Creating Service Accounts

```
# Create a service account
kubectl create serviceaccount my-service-account

# Get service accounts
kubectl get serviceaccounts

# Describe service account
kubectl describe serviceaccount my-service-account

# Delete service account
kubectl delete serviceaccount my-service-account
```

Service Account Configuration

```
# Define a Service Account
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app-service-account
  namespace: default

---
# Pod using a Service Account
apiVersion: v1
kind: Pod
metadata:
  name: app-pod
spec:
  serviceAccountName: app-service-account
  containers:
  - name: app
    image: my-app:latest
    volumeMounts:
    - name: token
      mountPath: /var/run/secrets/kubernetes.io/serviceaccount
  volumes:
  - name: token
    projected:
      sources:
      - serviceAccountToken:
```

```
          path: token
          expirationSeconds: 3600
```

Service Account with RBAC

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: pod-reader-sa
  namespace: default

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

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods-binding
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: pod-reader-role
subjects:
- kind: ServiceAccount
  name: pod-reader-sa
  namespace: default

---
# Pod that can read pods
apiVersion: v1
kind: Pod
metadata:
  name: pod-reader
spec:
  serviceAccountName: pod-reader-sa
  containers:
  - name: reader
    image: curlimages/curl:latest
    command:
    - /bin/sh
    - -c
    - |
      TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
      curl -H "Authorization: Bearer $TOKEN" \
```

```
        --cacert /var/run/secrets/kubernetes.io/serviceaccount/ca.crt \
        https://kubernetes.default.svc.cluster.local/api/v1/pods
```

Service Account Best Practices

Use Specific Service Accounts: Create dedicated Service Accounts for different applications or components rather than using the default Service Account.

Apply Least Privilege: Bind Service Accounts to Roles with only the minimum required permissions.

Avoid Default Service Account: Don't use the default Service Account in production applications; create specific ones with limited permissions.

Automate Token Rotation: Consider implementing token rotation mechanisms for enhanced security.

Monitor Service Account Usage: Regularly audit which Service Accounts exist and their associated permissions.

Service Account Operations

```
# Get tokens from service account
kubectl get secret -o yaml

# Mount service account token in pod
kubectl set serviceaccount deployment/my-app my-service-account

# Create service account with image pull secret
kubectl create serviceaccount my-app-sa \
  --dry-run=client -o yaml | kubectl apply -f -

# Add image pull secret to service account
kubectl patch serviceaccount my-app-sa \
  -p '{"imagePullSecrets": [{"name": "private-registry-creds"}]}'
```

12. RBAC (Role-Based Access Control)

Overview

RBAC is Kubernetes' native authorization mechanism that controls which users can access specific resources and perform specific actions.

RBAC Components

Role/ClusterRole: Define a set of permissions for specific resources and actions. Roles are namespace-scoped while ClusterRoles are cluster-wide.

RoleBinding/ClusterRoleBinding: Connect Roles to users, groups, or service accounts, granting the defined permissions.

Role and RoleBinding Examples

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

---
# RoleBinding to grant role to user
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: pod-reader
subjects:
- kind: User
  name: jane@example.com
  apiGroup: rbac.authorization.k8s.io

---
# ClusterRole for admin access
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-admin
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]

---
# ClusterRoleBinding for admin
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-admin-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: User
```

```
  name: admin@example.com
  apiGroup: rbac.authorization.k8s.io
```

RBAC Operations

```
# Create role and binding
kubectl apply -f rbac.yaml

# List roles
kubectl get roles

# List role bindings
kubectl get rolebindings

# Describe role
kubectl describe role pod-reader

# Test user permissions
kubectl auth can-i get pods --as=jane@example.com --namespace=default

# List cluster roles
kubectl get clusterroles

# Get cluster role bindings
kubectl get clusterrolebindings
```

13. Health Checks (Liveness, Readiness, and Startup Probes)

Overview

Health checks help Kubernetes determine if containers are running properly and ready to receive traffic. Three types of probes are available: liveness, readiness, and startup probes.

Probe Types

Liveness Probe: Detects if a container has entered a broken state and needs to be restarted.

Readiness Probe: Determines if a container is ready to accept traffic.

Startup Probe: Delays liveness and readiness checks until an application has finished startup.

Health Check Configuration

```
apiVersion: v1
kind: Pod
metadata:
  name: health-check-pod
spec:
  containers:
  - name: app
    image: my-app:latest
```

```
# HTTP Liveness Probe
livenessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 15
  periodSeconds: 20
  timeoutSeconds: 1
  failureThreshold: 3

# TCP Readiness Probe
readinessProbe:
  tcpSocket:
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 10

# Command Startup Probe
startupProbe:
  exec:
    command:
    - sh
    - -c
    - curl -f http://localhost:8080/startup || exit 1
  initialDelaySeconds: 0
  periodSeconds: 10
  failureThreshold: 30
```

Probe Protocol Options

HTTP GET: Makes an HTTP request to a specified path.

TCP Socket: Attempts to establish a TCP connection.

Exec: Executes a command inside the container.

Best Practices Summary

Area	Recommendation
Resource Management	Always set resource requests and limits; use namespaces for organization
Pod Health	Configure liveness and readiness probes for all containers
Configuration	Separate configuration using ConfigMaps; use Secrets for sensitive data
Storage	Use StorageClasses for dynamic provisioning; implement backup strategies
Security	Apply RBAC; use Service Accounts; implement container scanning
High Availability	Use Deployments with multiple replicas; configure Pod Disruption Budgets
Monitoring	Implement centralized logging and metrics collection
Networking	Use Services appropriately; match resource requests to actual needs

Area	Recommendation
Updates	Use rolling updates; maintain rollback capabilities
Service Accounts	Create dedicated Service Accounts; apply least privilege permissions

Conclusion

These thirteen core Kubernetes objects form the foundation of modern container orchestration. Pods provide the basic unit of deployment, while higher-level abstractions like Deployments, StatefulSets, DaemonSets, and Jobs orchestrate Pods for various workload patterns. Services expose applications, while storage solutions persist data. ConfigMaps and Secrets manage configuration, Service Accounts provide Pod identity, and RBAC provides security controls. Health checks ensure reliability, and understanding when and how to use each component is critical for building robust, scalable, and maintainable Kubernetes deployments.

Mastery of these objects enables you to design sophisticated, production-grade applications that leverage Kubernetes' powerful orchestration capabilities while maintaining security, reliability, and operational excellence.