PW SKILLS | DevOps and Cloud Computing

Kubernetes Objects, Networking, and Storage Management

Objective

- Understand how to manage rolling updates and rollbacks in Kubernetes deployments.
- Use Helm charts to package and deploy Kubernetes applications.
- Differentiate between service types and configure Ingress controllers.
- Implement DNS-based service discovery in Kubernetes.
- Manage persistent storage with PVs and PVCs.
- Deploy stateful applications using StatefulSets.
- Handle configuration using ConfigMaps and secure data using Secrets.

Explaining why rolling updates are used to update deployments with zero downtime.

Let’s see

- Rolling updates are used to update deployments with zero downtime because they incrementally replace old instances of an application with new ones, ensuring the service remains available throughout the update process.

Key reasons for using rolling updates for zero downtime include:

- Continuous Availability
- Risk Mitigation
- Seamless User Experience

Discussing how Kubernetes replaces Pods gradually to prevent service disruption

Let’s discuss

Kubernetes prevents service disruption during updates by using rolling updates, which gradually replace old Pods with new ones instead of updating all at once. Here’s how it works:

- Kubernetes creates new Pods running the updated application version and waits until they are healthy and ready.
- Only after the new Pods are ready does Kubernetes terminate the old Pods, ensuring that there is always a sufficient number of available Pods to handle user requests.

Let’s discuss

- The process is controlled by configurable parameters like maxUnavailable (maximum Pods that can be unavailable during the update) and maxSurge (maximum extra Pods allowed above the desired count during the update), allowing fine-tuning of update speed versus availability.
- Throughout the update, the Kubernetes Service only routes traffic to healthy, available Pods, so users do not experience downtime.

Demonstrating how rollback functionality reverts to a previous stable state in case of deployment failure.

Let's do it

Rollback functionality is a critical feature in deployment pipelines that allows systems to revert to a previous stable state if a deployment fails. Here's how it works in practice:

How Rollback Works

- Detection of Failure
- Initiation of Rollback
- Reversion to Previous State
- Restoration of Service

Introducing Helm as Kubernetes' package manager

Helm

Helm is the official package manager for Kubernetes, designed to simplify the deployment, management, and configuration of applications on Kubernetes clusters.

With Helm, you can:

- Install, upgrade, or uninstall applications
- Package and share applications
- Manage application releases
- Customize deployments

Explaining how Helm simplifies the deployment, upgrade, and rollback of Kubernetes applications.

Let’s see

- **Deployment:** Helm charts bundle all Kubernetes manifests and configurations, allowing applications to be deployed quickly and consistently without manually handling each resource.
- **Upgrade:** Helm tracks application versions and makes it easy to apply updates by upgrading to a new chart version, handling all underlying changes automatically.
- **Rollback:** If an upgrade fails, Helm can revert the application to a previous stable version with a simple rollback command, minimizing downtime and reducing manual intervention.

Discussing the structure of a Helm chart and how to use values.yaml for customization.

Let’s discuss

A Helm chart is organized into a specific directory structure that defines how Kubernetes applications are packaged and deployed. The key components of a Helm chart include:

- **Chart.yaml:** Contains metadata about the chart (name, version, description).
- **values.yaml:** Stores default configuration values for the chart's templates.
- **charts/:** Directory for chart dependencies (subcharts).
- **templates/:** Contains Kubernetes manifest templates (e.g., deployment.yaml, service.yaml) that are rendered using the values from values.yaml.

Let’s discuss

Using values.yaml for Customization:

- The values.yaml file provides default values for variables used in the chart’s templates.
- You can customize deployments by editing this file directly or by overriding its values at install/upgrade time using the --set flag or by providing a custom values file with the -f or --values flag.
- This allows you to tailor the deployment for different environments without modifying the chart templates themselves.

Explaining the role of Services in Kubernetes networking.

Role of services

Services in Kubernetes play a crucial role in networking by providing a stable way to expose and access a group of pods, regardless of their individual IP addresses, which can change over time.

Key functions of Services include:

- Service Discovery
- Load Balancing
- Stable Access

Describing and compare the service types:

Let's discuss & compare

Service Type	Accessibility	Use Case	Description
ClusterIP	Internal (within cluster)	Inter-service communication	Default type. Exposes the service on a cluster-internal IP. Not accessible from outside 1 2 3 4 .
NodePort	External (via node IP:port)	Basic external access for dev/test	Exposes the service on each node's IP at a static port (30000–32767). Accessible externally 2 3 4 .
LoadBalancer	External (via load balancer)	Production-grade external access	Provisions an external load balancer (usually via a cloud provider) to route traffic 2 [

Explaining how CoreDNS integrates with Kubernetes to allow service discovery via DNS.

Let’s see

- CoreDNS integrates with Kubernetes as the cluster’s DNS server, enabling automatic service discovery via DNS names.
- When a Kubernetes Service is created, CoreDNS-using its Kubernetes plugin-monitors the Kubernetes API and creates DNS records for each service, following the format service-name.namespace.svc.cluster.local.
- This allows pods to resolve and connect to services using standard DNS queries, regardless of changing pod IPs.
- As a result, applications within the cluster can reliably discover and communicate with each other by simply referencing service DNS names, streamlining connectivity and reducing manual configuration.

PW SKILLS

Describing how services are automatically registered and discoverable by name.

Let’s see

- When a new Service is created in Kubernetes, it is automatically registered and made discoverable by name through the cluster’s DNS system, managed by CoreDNS.
- CoreDNS continuously monitors the Kubernetes API for new or updated Services and dynamically creates DNS records for them in the format <service-name>.<namespace>.svc.cluster.local.
- Every Pod in the cluster is configured to use the CoreDNS service as its DNS resolver, so when a Pod tries to connect to a Service by its DNS name, CoreDNS resolves that name to the Service’s ClusterIP.

Take A 5-Minute **Break!**

- Stretch and relax
- Hydrate
- Clear your mind
- Be back in 5 minutes

PW SKILLS

Explaining how Ingress resources route external HTTP/S traffic to internal services.

Let’s see

Ingress resources in Kubernetes provide a centralized way to route external HTTP and HTTPS traffic to internal services based on rules you define. Here’s how the process works:

- **Ingress Resource Definition:** You create an Ingress resource as a YAML file, specifying routing rules based on hostnames and URL paths. These rules determine which internal service should receive traffic for specific domains or paths.
- **Ingress Controller:** An Ingress Controller (like NGINX or Traefik) must be running in the cluster. It watches for Ingress resources and implements the routing logic, acting as a smart reverse proxy and load balancer.

Let's see

- **Traffic Routing:** When external HTTP/S requests arrive at the cluster's entry point (often a cloud load balancer or node port), the Ingress Controller examines the request's host and path, matches it against the Ingress rules, and forwards the request to the appropriate internal service.
- **Features:** Ingress can also handle SSL/TLS termination, allowing encrypted traffic to be securely routed, and supports advanced routing features like path-based and host-based routing.

Discussing use cases for Ingress over traditional service types.

Let's discuss

Ingress resources offer several advantages over traditional Kubernetes service types (ClusterIP, NodePort, LoadBalancer) for managing external access to services. Here are key use cases where Ingress is preferred:

- Centralized Routing and Single Entry Point
- Advanced HTTP/S Routing
- SSL/TLS Termination
- Load Balancing
- Centralized Access Control and Security
- Reduced Complexity and Cost

Introducing Ingress Controllers (e.g., NGINX, Traefik) and their role in exposing services.

Let’s introduce

Ingress Controllers, such as NGINX and Traefik, are essential components in Kubernetes that implement the Ingress API to expose internal services to external HTTP/S traffic.

Roles:

- When you deploy an Ingress Controller, it continuously monitors Ingress resources and dynamically configures itself to direct traffic according to hostnames, paths, and other criteria.
- This enables centralized, flexible, and secure access to multiple services through a single entry point, simplifying external connectivity and traffic management in Kubernetes environments.

Explaining the need for persistent storage in a containerized environment.

Let’s see

Persistent storage is essential in a containerized environment because containers and pods are ephemeral by nature-their data is lost when they are terminated, restarted, or rescheduled.

- Persistent storage addresses this need by providing storage that is independent of the pod or container lifecycle.
- In Kubernetes, this is achieved using Persistent Volumes (PVs) and Persistent Volume Claims (PVCs), which allow data to persist even if the pod using it is deleted or recreated.

Describing how PVs and PVCs abstract the storage provisioning process.

Let’s see

Persistent Volume (PV): A PV is a cluster resource representing a piece of storage provisioned by an administrator or dynamically via a StorageClass.

- PVs can be backed by various storage technologies (e.g., NFS, cloud storage) and exist independently of any specific pod’s lifecycle.

Persistent Volume Claim (PVC): A PVC is a user’s request for storage with specific requirements such as size, access mode, and storage class.

- When a PVC is created, Kubernetes automatically matches and binds it to an available PV that meets the criteria, or triggers dynamic provisioning if no suitable PV exists.

Discussing binding, access modes, and reclaim policies.

Let’s discuss

Binding:

- Binding is the process where Kubernetes matches a Persistent Volume Claim (PVC) to a Persistent Volume (PV) that meets its requirements (such as size, access mode, and storage class).
- This process is automatic: when a PVC is created, Kubernetes searches for an available PV that satisfies the claim and binds them together. If no suitable PV exists at the time, the claim remains unbound until a matching PV is created.

Let’s discuss

Access Modes:

Access modes define how a volume can be mounted and used by pods. The main access modes are:

- ReadWriteOnce (RWO)
- ReadOnlyMany (ROX)
- ReadWriteMany (RWX)
- ReadWriteOncePod (RWO-Pod)

Let’s discuss

Reclaim Policies:

Reclaim policies determine what happens to a PV after its bound PVC is deleted:

- Retain
- Delete
- Recycle

Describing how StatefulSets differ from Deployments in terms of stability and identity.

Let’s see

Feature	StatefulSet	Deployment
Pod Identity	Unique, stable, predictable (e.g., `app-0`, `app-1`)	Random, interchangeable
Storage	Persistent, per-pod volume (via PVCs)	Ephemeral by default
Use Case	Stateful apps (databases, queues)	Stateless apps (web servers, APIs)
Pod Management	Ordered (sequential creation, update, deletion)	Unordered (parallel operations)
Network Identity	Stable DNS name per pod	No stable DNS per pod

Discussing use cases like databases and storage-intensive applications.

Let’s discuss

Databases: Applications like MySQL, PostgreSQL, MongoDB, and Elasticsearch require each pod to have a stable, unique identity and persistent storage.

- For example, in a replicated database setup, one pod might serve as the primary node (handling writes), while others act as read replicas.

Storage-Intensive Applications: StatefulSets are well-suited for distributed data stores, message queues, and consensus-based systems (like etcd or ZooKeeper) that need stable network identities and persistent volumes.

- Each pod in a StatefulSet is linked to its own PersistentVolume, ensuring that data is preserved across restarts or failures and that storage is not accidentally shared between pods.

Explaining the purpose of externalizing configuration from application containers.

Let’s see

Externalizing configuration from application containers means storing configuration data outside the container image, typically using tools like Kubernetes ConfigMaps and Secrets. This approach offers several key benefits:

- Flexibility and Dynamic Updates
- Portability
- Centralized Management
- Separation of Concerns
- Security

Demonstrating how to inject ConfigMaps as environment variables or mounted volumes.

Let’s do it

1. Injecting ConfigMaps as Environment Variables
    - Single Key as Environment Variable:

Reference a specific key from the ConfigMap in the Pod spec using env.valueFrom.configMapKeyRef.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
  - name: app
    image: busybox
    env:
    - name: SPECIAL_LEVEL_KEY
      valueFrom:
        configMapKeyRef:
          name: special-config
          key: special.how
```

Let’s do it

- All Keys as Environment Variables:

Use ‘envFrom’ to inject all key-value pairs from a ConfigMap as environment variables.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
  - name: app
    image: busybox
    envFrom:
    - configMapRef:
        name: special-config
```

Let’s do it

2. Mounting ConfigMaps as Volumes

- Mount ConfigMap Data as Files:

Reference the ConfigMap in the ‘volumes’ section and mount it into the container's filesystem using ‘volumeMounts’.

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
  - name: app
    image: busybox
    volumeMounts:
    - name: config-volume
      mountPath: /etc/config
  volumes:
  - name: config-volume
    configMap:
      name: special-config
```

Explaining why Kubernetes Secrets are used for storing credentials and sensitive data.

Let’s see

Kubernetes Secrets are used for storing credentials and sensitive data-such as passwords, tokens, and keys-to reduce the risk of exposing confidential information in application code, container images, or configuration files.

- Secrets provide a more secure and manageable way to handle sensitive data by keeping it separate from non-sensitive configuration, enabling access controls, and supporting encryption at rest.
- This helps ensure that only authorized applications or users can access critical credentials, improving security in containerized environments.

Comparing plain-text values, base64 encoding, and RBAC-controlled access.

Let’s compare

Aspect	Plain-text Values	Base64 Encoding	RBAC-Controlled Access
Purpose	Directly stores data as readable text	Encodes data to handle binary content and ensure safe storage	Restricts who can access, modify, or list secrets
Security	Highly insecure; easily readable and exposed	Not secure-base64 is reversible and not encryption	Strong security-enforces least privilege and auditability

Let’s compare

Usage in Kubernetes	Not recommended for sensitive data; use for non-secret config	All Kubernetes Secrets are stored as base64-encoded strings	RBAC policies define which users/pods can access which secrets
Protection Level	None	Minimal (obfuscation only)	High (with proper RBAC and encryption at rest enabled)
Best Practice	Avoid for credentials or sensitive info	Required by Kubernetes for Secrets, but not sufficient alone	Always use RBAC to limit secret access to only what is needed

Demonstrating using Secrets as environment variables or mounted files.

Let’s do it

1. Using Secrets as Environment Variables

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: env-pod
spec:
  containers:
  - name: secret-test
    image: nginx
    env:
    - name: USER
      valueFrom:
        secretKeyRef:
          name: database-credentials
          key: username.txt
    - name: PASSWORD
      valueFrom:
        secretKeyRef:
          name: database-credentials
          key: password.txt
```

Let’s do it

2. Mounting Secrets as Files in Volumes

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: volume-test-pod
spec:
  containers:
  - name: secret-test
    image: nginx
    volumeMounts:
    - name: secret-volume
      mountPath: /etc/config/secret
  volumes:
  - name: secret-volume
    secret:
      secretName: database-credentials
```

Time for case study!

Important

- Complete the post-class assessment
- Complete assignments (if any)
- Practice the concepts and techniques taught in this session
- Review your lecture notes
- Note down questions and queries regarding this session ar consult the teaching assistants

Thanks

!