CONCEPT

Concise Kubernetes

FUNDAMENTALS

What we Learn

- Kubernetes - Why it's essential
- Pods, Nodes, and Clusters.
- Kubernetes Manages Containers
- Pods- The smallest deployable unit.
- Scaling & managing workloads.
- Namespaces - Organizing resources
- Services- Exposing applications.

Part 1

Concise Kubernetes

Why is Kubernetes Essential

☞ **What is Kubernetes**?
Kubernetes is an open-source container **orchestration** platform that automates deploying, scaling, and managing applications in containers.

History:
Kubernetes was initially developed by Google to solve the challenges of managing containerized applications at scale. Google had an internal system called **Borg** that handled container orchestration.- Project 7 🚀
When a group of engineers started to work on open-sourcing a more streamlined, scalable orchestration platform based on their learnings from Borg, they code-named it "Project 7.

Project 7 evolved into a robust and open-source-ready platform, it was officially named "Kubernetes."

☞ **Why is it Essential in Modern DevOps**?
1.) **Efficient Resource Usage** ⚙️: Kubernetes optimizes resource use, helping teams save on hardware and cloud costs.
2.) **Environment Consistency**: It provides a consistent platform for development, testing, and production, ensuring smooth deployments.
3.) **Scalability** 📈: Automatically scales applications up or down to handle load changes efficiently.
4.) **Self-Healing**: Detects and replaces failed containers, maintaining uptime.
5.) **Security & Compliance** 🔐: Kubernetes offers built-in security features, like Role-Based Access Control (RBAC) and Secrets management.

CONCEPT

Concise Kubernetes

DAY#1

Know your Kubernetes Components

Cluster:
A Kubernetes cluster is a collection of nodes (machines) where Kubernetes manages and orchestrates containerized applications. A cluster represents a Kubernetes deployment as a whole, combining both worker nodes (where applications run) and a control plane (which manages and monitors the cluster).

Control Plane:
The control plane is responsible for the cluster's overall management. The cluster's control plane continuously monitors the actual state of the applications and resources, adjusting to match the desired state. This "desired state" is defined by the configurations (usually in YAML files) you apply to Kubernetes.

Control Plane comprises of several components that work together to manage cluster operations:
➲ **API Server:** Acts as the main access point for Kubernetes, handling communication between the user and the control plane.
➲ **Controller Manager:** Manages controllers, which ensure that the actual state matches the desired state (like ensuring the right number of pods).
➲ **Scheduler:** Assigns new pods to nodes based on available resources and other factors.
➲ **etcd:** A distributed key-value store that holds configuration data, cluster state, and metadata. It is similar to a database. It has the actual status of the pod.

Nodes:
A node is a single machine (either a virtual machine or a physical server) that runs in the Kubernetes cluster and hosts Pods. Nodes are the workhorses of Kubernetes; they perform all the tasks needed to keep applications running.

Types of Nodes:
Master Node (Control Plane Node): Manages the cluster and runs control plane components (API server, etcd, scheduler, controller manager).
Worker Node: Executes containerized applications and manages Pods.

Node Components:
➲ **Kubelet:** An agent that runs on each node, communicates with the API server, and ensures that containers in Pods are running as expected.
➲ **Kube-proxy:** Manages network rules and facilitates network communication between services, ensuring seamless Pod-to-Pod and Pod-to-external-traffic connections.
➲ **Container Runtime:** The software responsible for running the containers on each node (Docker, containerd, etc.).

In Kubernetes, nodes can be added or removed based on workload demands. Adding more worker nodes allows the cluster to run more applications or manage higher workloads, enhancing scalability.

How Kubernetes Manages Containers

Kubernetes manages containers by a process called container **orchestration**.
Container orchestration is the process of managing and automating the deployment, scaling, networking, and operation of containers across multiple servers.

How Kubernetes Manages Containers:

Automates Deployment 🚀: When you want to deploy a containerized application, Kubernetes handles all the steps automatically. You just tell Kubernetes what you want, and it takes care of all details.

Scales Up and Down 📈📉: Kubernetes can automatically add or remove containers based on the current demand. If your app is getting a lot of traffic, Kubernetes can spin up more containers to handle it.

Self-Healing ❤️: If a container crashes or has an issue, Kubernetes automatically detects it and replaces it with a new one.

Networking & Load Balancing 🌐: Kubernetes manages network connections between containers so they can communicate. It also distributes user requests across containers, balancing the load to prevent any single container from being overwhelmed.

Declarative Management 📜: You define the “desired state” of your application (like how many containers should be running, what version, etc.) in a configuration file. Kubernetes continuously monitors and makes sure the current state matches this desired state.

Pod: The Fundamental Building Block

A Pod is the smallest and most basic deployable unit in Kubernetes. It represents a single instance of a running process in the Kubernetes cluster. While containers are at the heart of modern application deployment, Kubernetes introduces Pods to provide an abstraction that manages one or more containers cohesively.

Why does Kubernetes use Pods?

Kubernetes could manage containers directly, but Pods provide:

1. Logical Grouping Containers in a Pod form a logical unit of deployment and scaling.
2. Enhanced Collaboration Shared storage and networking simplify interactions between tightly coupled containers.
3. Unified Resource Management Kubernetes can allocate resources, handle health checks, and manage container lifecycles more effectively by grouping them into Pods.

Key Features of a Pod

🔪 **Encapsulation**: A Pod encapsulates one or more containers.
Shared resources for those containers, including:

- Networking: All containers in a Pod share the same network namespace.
- Storage: Volumes are shared among the containers within a Pod.

✒️ **Lifecycle Management:** Kubernetes manages Pods rather than individual containers. This ensures containers in a Pod are always co-located, co-scheduled, and run in a tightly coupled manner.
✒️**Atomic Deployment Unit:** If a Pod fails, Kubernetes does not repair it but replaces it with a new Pod instance. Pods are designed to be ephemeral.

Components of a Pod
➡ **Containers**: Most Pods run a single container, but you can run multiple containers in a single Pod if they are tightly coupled.
➡**Shared Network Namespace:** Containers in the same Pod share:

- IP Address: Assigned at the Pod level.
- Ports: Containers within a Pod communicate via localhost.

➡**Volumes:** Storage shared among containers in the Pod.

Pod Lifecycle

1. **Pending**: The Pod is created but not yet running. This happens while Kubernetes schedules the Pod to a node.
2. **Running**: The Pod is successfully scheduled and all containers are running or in the process of starting.
3. **Succeeded**: All containers in the Pod have terminated successfully (exit code 0).
4. **Failed**: At least one container in the Pod has terminated with a non-zero exit code.
5. **Unknown**: The state of the Pod cannot be determined.

Nodes and Clusters - Scaling and managing workloads

A **node** is a single machine (virtual or physical) in a Kubernetes cluster that runs the actual workloads.

Node contains:

- **Kubelet**: Ensures that containers defined in the pod spec are running.
- **Kube-proxy**: Manages networking and communication for the pods.
- **Container Runtime:** Software to run containers (e.g. Docker, containerd).

Nodes can be worker nodes or control-plane nodes.

A **cluster** is a set of nodes working together, managed by Kubernetes, to run containerized applications.

- It includes a control plane that orchestrates and a group of worker nodes that handle workloads.
- Clusters ensure high availability and fault tolerance by distributing workloads across multiple nodes.

Scaling in Kubernetes

Kubernetes supports two types of scaling:

1. **Horizontal Pod Autoscaling (HPA):** Dynamically adjusts the number of pod replicas for a deployment or replica set. It monitors metrics like CPU, memory, or custom application metrics.Adds or removes pod replicas based on thresholds.
2. **Node Scaling:** Adding or removing nodes to/from the cluster.

Managed manually or automatically using tools like Cluster Autoscaler.
Cluster Autoscaler integrates with cloud providers to add remove virtual machines dynamically.

Managing Workloads in Kubernetes

Workloads: Workloads are the applications or services running on Kubernetes.

It is defined in Kubernetes using manifests (YAML or JSON files).

Types of Workloads:

- **Deployments**: For stateless applications; supports scaling and updates.
- **StatefulSets**: For stateful applications that require unique identities and stable storage (e.g., databases).
- **DaemonSets**: Ensures a copy of a pod runs on every node (e.g., log collectors).
- **Jobs and CronJobs**: For running one-time or scheduled tasks.

Other Features:

- **Load Balancing:** Kubernetes ensures workloads are balanced across the cluster using Services and Ingress.
- M**onitoring and Logging:** Tools like Prometheus, Grafana, and ELK Stack (Elasticsearch, Logstash, Kibana) help monitor workloads and log activities.
- **High Availability and Resilience:** Kubernetes automatically restarts failed pods and reschedules them to healthy nodes.

Benefits of Nodes and Clusters:

- Efficient Resource Utilization:
- Fault Tolerance
- Dynamic Scaling
- Centralized Management

Namespace in Kubernetes

Namespaces are a way to organize and isolate resources within a cluster. They provide a mechanism for dividing cluster resources between multiple users or teams. Namespaces are like virtual clusters within a single physical Kubernetes cluster.

You can think of Namespaces as separate "rooms" within a "house" (the cluster), where each room contains its own set of furniture (resources) and can be managed independently.

☛ **Key Features of Namespaces:**

🔷 **Isolation of environments** for different teams or projects. Resources within one namespace do not interact directly with resources in another.

🔷 **Resource Quota Management:** It define limits (e.g., CPU, memory) for resources within a namespace to avoid resource contention.

🔷 **Access Control:** Kubernetes Role-Based Access Control (RBAC) can be applied at the namespace level to restrict access to specific users or teams.

🔷 **Ease of Management:** Namespace is a group that manage resources related to a particular application or environment (e.g., dev, staging, prod) more effectively.

☛ **When to Use Namespaces?**

- For large-scale clusters with multiple teams or projects.
- To separate environments such as development, staging, and production.
- When implementing multi-tenancy to serve multiple customers.
- To enforce resource quotas and policies.

How Does a Namespace Organize Resources in Kubernetes?

♦ **Scoping Resources:** Resources like Pods, Services, ConfigMaps, and Secrets are created within a specific namespace. You can have a Pod named app in the **dev** namespace and another Pod with the same name in the **prod** namespace. Example:

kubectl get pods -n dev
kubectl get pods -n prod

♦ **Default Namespace:** If no namespace is specified, resources are created in the default namespace.

♦**System Namespaces:** Kubernetes reserves certain namespaces for system resources:

1. **kube-system:** For core Kubernetes components (e.g., the scheduler, controller manager).
2. **kube-public:** For publicly accessible resources.
3. **kube-node-lease**: For node heartbeat leases.

♦ **Resource Isolation:** Namespaces isolate resources logically. A Service in the dev namespace cannot directly resolve or access a Pod in the prod namespace without explicit configuration.

♦ **Cross-Namespace Operations:** Admin-level operations can view or manage resources across all namespaces using the --all-namespaces flag.
kubectl get pods --all-namespaces

Services in Kubernetes: Exposing to the world

Before getting started with Services in Kubernetes, the first question comes in our mind is "**Why do we need Services?**". Suppose we have a website where we have 3 pod replicas of front-end and 3 pod replicas of backend. These are the following scenarios we have to tackle:

1. How would the front-end pods be able to access the backend pods?
2. If the front-end pod wants to access the backend pod to which replica of the backend pod will the requests be redirected. Who makes this decision?
3. As the IP address of the pods can change, who keeps the track of the new IP addresses and inform this to the front-end pods?
4. As the containers inside the pods are deployed in a private internal network, which IP address will the users use to access the front-end pods?

To overcome the above mentioned cases the services object is created. Services enables **loose coupling** between microservices in our application. It enables communication between various components within and outside of the application.

In Kubernetes, Services are an abstraction that define a logical set of Pods and a policy for accessing them. Since Pods in Kubernetes are ephemeral (can be created and destroyed dynamically), their IP addresses are not static. This is where Services come in—to provide stable networking and access to the Pods.

Why Do We Need Services in Kubernetes?

- **Pods Are Ephemeral:** Pods are temporary and can be destroyed or recreated for reasons like scaling, updates, or failures. Each new Pod gets a different IP address, making direct communication with Pods unreliable.
- **Stable Communication:** Services provide a consistent way to access the Pods, regardless of changes in the underlying Pods or their IPs.
- **Load Balancing:** Services distribute network traffic among multiple Pods.
- **Discovery:** They simplify service discovery by acting as a single access point to a group of Pods.

Feature	ClusterIP	NodePort	LoadBalancer
Exposition	Exposes the Service on an internal IP in the cluster.	Exposing services to external clients	Exposing services to external clients
Cluster	This type makes the Service only reachable from within the cluster	A NodePort service, each cluster node opens a port on the node itself (hence the name) and redirects traffic received on that port to the underlying service.	A LoadBalancer service accessible through a dedicated load balancer, provisioned from the cloud infrastructure Kubernetes is running on
Accessibility	It is **default** service and Internal clients send requests to a stable internal IP address.	The service is accessible at the internal cluster IP-port, and also through a dedicated port on all nodes.	Clients connect to the service through the load balancer's IP.
Yaml Config	`type: ClusterIP`	`type: NodePort`	`type: LoadBalancer`
Port Range	Any public ip form Cluster	30000 - 32767	Any public ip form Cluster

CONCEPT

Concise Kubernetes

DAY#7

How Services Work Internally?

Label Selector: The Service identifies the set of Pods it manages using label selectors.

Endpoints Object: Kubernetes creates an Endpoints object, which keeps track of the IPs of Pods matching the Service's selector.

Service Proxying: Kubernetes uses kube-proxy to handle traffic to the Service and forward it to the appropriate Pod.kube-proxy uses methods like iptables or IPVS for load balancing.

Key Concepts of Services

- **Stable Endpoint:** A Service gives a consistent IP address (called a ClusterIP) and DNS name that remains constant, even as the underlying Pods change.
- **Discovery and Load Balancing:** Services allow clients to discover and communicate with the right Pods. They automatically distribute traffic among the Pods using label selectors and load-balancing mechanisms.
- **Selector and Labels:** A Service uses selectors to identify which Pods it should target. Pods with labels that match the selector will automatically become part of the Service.

WORKING WITH OBJECTS

What we Learn

- Understanding Kubernetes YAML Files
- Deployment-manage app updates
- ReplicaSets - Ensure high availability
- StatefulSets - Manage stateful app
- DaemonSets - Running on each node
- Jobs & CronJobs - Run & scheduled jobs
- ConfigMaps - Manage app config.

Part 2

Concise Kubernetes

Understanding Kubernetes YAML Files

Kubernetes has become a leading container orchestration platform, offering scalability, resilience, and portability.

There are two different ways to configure all components in Kuberneetes - **Declarative** and **Imperative**. Declarative way brings manifest file in the discussion which is written in either JSON or in YAML.

So, in general, YAML files are a fundamental aspect of defining Kubernetes resources. Key components of a YAML file, namely apiVersion, kind, metadata, and spec. By understanding these key elements, you will gain insights into how to create and configure Kubernetes resources effectively.

✍️ **apiVersion :**
The apiVersion field in a Kubernetes YAML file specifies the version of the Kubernetes API that the resource adheres to. It ensures compatibility between the YAML file and the Kubernetes cluster.
☞ **Core API Group**: Includes fundamental resources.
- Pods: apiVersion: v1
- Services: apiVersion: v1
- ConfigMaps: apiVersion: v1
- Secrets: apiVersion: v1

☞ **Apps API Group** : Used for managing workloads.
- Deployments: apiVersion: apps/v1
- DaemonSets: apiVersion: apps/v1
- StatefulSets: apiVersion: apps/v1
- ReplicaSets: apiVersion: apps/v1

CONCEPT

Concise Kubernetes

DAY#8

✍️ **kind :**
The kind field defines the type of resource being created or modified. It determines how Kubernetes interprets and manages the resource.
☞ Each kind has a specific purpose. For instance:
- Pod: Represents a single or multiple containers.
- Service: Exposes a set of Pods as a network service.
- Deployment: Manages rolling updates for applications.

✍️ **metadata :**
The metadata field contains essential information about the resource, such as its name, labels, and annotations. It helps identify and organize resources within the cluster.

- **name**: Specifies the name of the resource, allowing it to be uniquely identified within its namespace.
- **labels**: Enables categorization and grouping of resources based on key-value pairs. Labels are widely used for selecting resources when using selectors or applying deployments.
- **annotations**: Provides additional information or metadata about the resource. Annotations are typically used for documentation purposes, tooling integrations, or adding custom metadata.

✍️ **spec** :
The spec field describes the desired state of the resource. It outlines the configuration details and behavior of the resource. The structure and content of the spec field vary depending on the resource kind.

Deployment - How to manage application updates

A Deployment provides replication functionality with the help of ReplicaSets and various additional capabilities like rolling out of changes and rollback changes.

You describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

Key Features of Deployments

1. **Declarative Management:** You describe the desired state in a YAML or JSON file. Kubernetes takes care of the changes.
2. **Rolling Updates:** Updates occur incrementally, ensuring zero downtime.
3. **Rollback Capability:** Kubernetes can revert to previous versions if an update fails.
4. **Version History:** Deployments maintain a history of ReplicaSets, enabling easier rollbacks.
5. **Scaling:** Deployments manage the number of replicas dynamically.

Steps to Manage Application Updates

1. **Define the Deployment:** A Deployment describes the application's desired state, including the image version, number of replicas, and update strategy.

2. **Apply Updates Using kubectl:** You can update the application by changing the image tag in the Deployment file and applying it

3. Monitor the Update Progress: Use the following commands to observe the update:
Check Deployment status: ***kubectl rollout status deployment/my-app***
View update history: ***kubectl rollout history deployment/my-app***

4. **Rollback if Needed:** If the update causes issues, rollback to the previous version:
kubectl rollout undo deployment/my-app
You can also specify a specific revision to rollback to:
kubectl rollout undo deployment/my-app --to-revision=2

5. **Blue-Green Deployment (Optional)**
Instead of a rolling update, you can create a new Deployment with the updated version and use a Service to switch traffic between the old and new versions.

6. Canary Deployment (Optional)
This approach deploys the new version to a small subset of users first, allowing testing in a production-like environment. Gradually, the new version replaces the old one.

Challenge	Solution
Application Downtime	Use rolling updates or blue-green deployments.
Failed Updates	Rollback using `kubectl rollout undo`.
Long Update Durations	Adjust `maxUnavailable` and `maxSurge` parameters for faster updates.
Traffic Routing Issues	Use Service objects or ingress controllers to manage traffic properly.

Replicaset: How does it ensure high avaialbility

A ReplicaSet is a Kubernetes resource that ensures a **specified number of identical pod** replicas are running at any given time.

ReplicaSet is a process that run multiple instances of Pods. It constantly monitors the status of Pods and if any one fails or terminates then it restores by creating new instance of Pod and by deleting old one.

Key Features

1. **Desired State Management:** Maintains the desired number of replicas.
2. **Automatic Recovery:** Recreates pods that are deleted or fail.
3. **Selectors**: Matches pods using label selectors to manage them.

ReplicationController

A ReplicationController is an older Kubernetes resource with a similar purpose to ReplicaSets. It ensures that a specified number of pod replicas are running at all times.

How They Ensure High Availability

1. Maintaining Desired Pod Count: ReplicaSet and ReplicationController continuously monitor the pod count. If a pod fails or is deleted, they create new pods to match the desired state.
2. Automatic Rescheduling: If a node fails, pods are recreated on healthy nodes, ensuring availability.
3. Workload Distribution: Pods are distributed across nodes to prevent single points of failure.
4. Health Checks: Integration with liveness and readiness probes ensures only healthy pods serve traffic.

CONCEPT

Concise Kubernetes

DAY#10

Differences between ReplicaSet and ReplicationController:

➢ **Replicaset**:

- Selectors: Supports set-based selectors (more flexible).
- Use Case: Used with Deployments for modern applications.
- Efficiency: More advanced and flexible.

➢ **ReplicationController**

- Selectors: Supports only equality-based selectors.
- Use Case: Considered legacy, replaced by ReplicaSet.
- Efficiency: Limited to basic replication tasks.

- ReplicaSets are the preferred approach as part of Deployments for robust application lifecycle management.
- ReplicationControllers are rarely used now and have been largely replaced by ReplicaSets for advanced capabilities and flexibility.

StatefulSet in Kubernetes

A StatefulSet in Kubernetes is a resource designed to manage and deploy stateful applications— applications that require **persistent storage**, **stable** network identity, and **ordered deployment** or scaling.

Unlike Deployments, which handle stateless applications, StatefulSets ensure that each replica of an application has a unique identity and stable, consistent storage.

See, how we write YAML for a StatefulSet:

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

CONCEPT

Concise Kubernetes

DAY#11

Key Features of StatefulSets

♦ **Stable Network Identity:** Each pod in a StatefulSet gets a unique, persistent hostname that follows the pattern ***<statefulset-name>-<ordinal>.***

♦ **Stable Persistent Storage:** Each pod is associated with its own persistent volume (PV). Even if a pod is deleted or rescheduled, it retains its associated data by reattaching the same PV.

♦ **Ordered Deployment and Scaling:**

- Pods are created, updated, or deleted in a sequential order (e.g., Pod 0 before Pod 1).
- When scaling down, the highest-numbered pod is removed first.

♦ **Ordered Rolling Updates:** Updates to pods are performed in a controlled, sequential manner, ensuring minimal disruption to the application.

Feature	StatefulSet	Deployment
Pod Identity	Stable, unique for each pod	Dynamic, shared across replicas
Persistent Storage	Dedicated per pod	Shared or ephemeral storage
Scaling Behavior	Ordered scaling	All pods can scale simultaneously
Rolling Updates	Sequential updates	Parallel updates

DaemonSets: ensure Running a pod on each node

A DaemonSet is a Kubernetes resource that ensures a specific pod is running on all or selected nodes in a cluster. This is particularly useful for running background tasks, system monitoring, or log collection on every node.

Key Features of DaemonSets:

Pod Deployment to All Nodes:

- DaemonSets ensure that a copy of a pod runs on every eligible node in the cluster.
- When a new node is added, the DaemonSet automatically creates a pod on it.
- If a node is removed, the pod associated with it is also removed.

Selective Node Deployment:

- You can restrict DaemonSets to certain nodes using node selectors, node affinity, or taints and tolerations.

Automatic Updates:

- DaemonSets automatically maintain the desired pod specification on nodes. If you update the DaemonSet, the pods on the nodes will be updated accordingly.

How DaemonSets Ensure Running a Pod on Each Node:

Controller Mechanism: DaemonSet Controller monitors the cluster state. It ensures the desired number of pods (one per eligible node) is maintained by reconciling the cluster's actual state with the desired state defined in the DaemonSet.

Pod Scheduling:
When you create a DaemonSet, Kubernetes:
- Schedules the DaemonSet pod on all eligible nodes using the kube-scheduler.
- Ensures these pods run even if there are changes to the node pool, such as adding or removing nodes.

Node Addition or Removal:
New Node Added: When a new node is added, the DaemonSet controller detects it and schedules a pod on the node.
Node Removed: If a node is removed, the corresponding pod is deleted.

Immutable Management:
Each DaemonSet pod is managed as an individual unit, but Kubernetes ensures that every eligible node runs exactly one instance of the pod.

When to use DaemonSets:

- **Log Collection:** Running logging agents like Fluentd or Logstash on every node.
- **Monitoring:** Running system monitoring agents like Prometheus Node Exporter.
- **Networking:** Running network-related tools, such as a CNI plugin or a network proxy.

Jobs and CronJobs: help in Running jobs?

Jobs:
A Job is a Kubernetes resource that runs a specific task to completion. It's designed for one-time or on-demand tasks. Once the task is finished, the Job and its associated Pods are automatically cleaned up.

Key Characteristics of Jobs:

- **One-time execution:** Jobs are intended for tasks that need to be run once.
- **Parallelism:** Jobs can be configured to run multiple Pods concurrently to speed up processing.
- **Completion guarantee:** Kubernetes ensures that the Job completes successfully, even if Pods fail.
- **Automatic cleanup:** Once the task is finished, the Job and its Pods are removed.

Use Cases for Jobs:

- **Data processing:** Processing large datasets that require significant computational resources.
- **Batch jobs:** Running batch jobs like data imports, exports, or report generation.
- **One-time tasks:** Executing tasks that need to be run only once, such as database migrations or system updates.

CronJobs:
A CronJob is a Kubernetes resource that schedules Jobs to run periodically based on a specified schedule. It's like a time-based trigger that initiates Jobs at predefined intervals.

Key Characteristics of CronJobs:

- **Scheduled execution:** CronJobs define a schedule using a cron expression to determine when to run Jobs.
- **Recurring tasks:** They are ideal for tasks that need to be executed repeatedly, such as backups, log rotation, or data synchronization.
- **History:** CronJobs keep a history of past executions, allowing you to track Job completion and failures.

Use Cases for CronJobs:

- **Backup and restore:** Automating regular backups of databases, filesystems, or applications.
- **Log rotation:** Deleting old log files to save disk space.
- **Data synchronization:** Keeping data consistent across different systems or databases.
- **Monitoring and alerting:** Running scripts to monitor system health and send alerts.
- **Report generation:** Generating reports at regular intervals.

How Jobs and CronJobs Work Together

1. CronJob Scheduling: The CronJob controller monitors the cluster for CronJob objects. When a CronJob's schedule matches the current time, it creates a new Job.
2. Job Execution: The Job controller creates Pods to execute the task defined in the Job spec.
3. Task Completion: The Pods run the task and report their status to the Job controller.
4. Job Completion: Once all Pods associated with the Job complete successfully, the Job is considered finished.
5. Cleanup: The Job and its Pods are automatically deleted.

By effectively utilizing Jobs and CronJobs, you can automate routine tasks, optimize resource utilization, and ensure the reliability and efficiency of your Kubernetes applications.

Feature	Job	CronJob
Purpose	Executes tasks once or until complete.	Schedules recurring tasks.
Trigger	Manually or programmatically triggered.	Runs on a defined schedule (cron syntax).
Use Case	Database migration, file processing.	Nightly backups, scheduled reports.
Retries	Supports retries on failure.	Inherits retries from the Job template.

ConfigMaps - Managing application configurations

ConfigMaps in Kubernetes is a **key-value** store that allows you to manage application configuration data independently from the application code.

This separation of configuration from the application code adheres to the 12-factor app methodology and ensures flexibility and portability across environments (development, staging, production).

In Kubernetes, Configmap is an API object that is mainly used to store non-confidential data. The data that is stored in ConfigMap is stored as key-value pairs. **ConfigMaps** are configuration files that may be used by pods as command-line arguments, environment variables, or even as configuration files on a disc.

This feature allows us to decouple environment-specific configuration from our container images, after this is done, our applications are easily portable. The thing to be noted here is that ConfigMap does not provide any sort of secrecy or encryption, so it is advised to store non-confidential data only We can use secrets to store confidential data.

Working with Kubernetes ConfigMaps allows you to separate configuration details from containerized apps. ConfigMaps are used to hold non-sensitive configuration data that may be consumed as environment variables by containers or mounted as configuration files.

ConfigMaps - Managing application configurations

How ConfigMaps Help Manage Application Configurations:

✅ **Decouples Configuration from Code** ConfigMaps store configuration details outside of the application codebase, allowing you to:

- Update configurations without rebuilding or redeploying the application.
- Manage different configurations for different environments

✅ **Centralized Management** All configuration data can be stored in one place (ConfigMaps), making it easier to manage and update application settings.

✅ **Dynamic Updates** If a ConfigMap is mounted as a volume, changes to the ConfigMap automatically propagate to the pod's filesystem (though not all applications reload these changes dynamically without a restart).

✅ **Portability Across Environments** The same container image can be used across multiple environments by just updating the ConfigMap with environment-specific configurations.

✅ **Flexible Injection Methods** ConfigMaps provide multiple ways to inject configurations into your application:

- As environment variables.
- As files mounted into the container (via volumes).
- Through command-line arguments.

✅ **Simplifies Secret Management (with Limits)** ConfigMaps can hold non-sensitive application parameters, simplifying their management and avoiding exposure in application source code.