**Kubernetes**

**Introduction to Kubernetes**

* Kubernetes (K8s): An open-source container orchestration platform for automating deployment, scaling, and management of containerized applications.
* Developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF).
* Helps manage clusters of nodes running containers.

**Key Concepts**

1. **Cluster**

* Master Node: Manages the Kubernetes cluster. Handles scheduling, scaling, and deployment.
* Worker Nodes: Run the containerized applications. Each node contains pods.

**2. Pod**

* Smallest deployable unit in Kubernetes.
* Encapsulates one or more containers, storage resources, a unique network IP, and options for running containers.

**3. Service**

* Exposes an application running on a set of Pods as a network service.
* Types: Cluster IP, Node Port, Load Balancer, External Name.

**4. Deployment**

* Manages the deployment and scaling of Pods.
* Ensures the desired state of applications is maintained.

**5. ConfigMap & Secret**

* ConfigMap: Manages configuration data for applications.
* Secret: Stores sensitive information like passwords and API tokens.

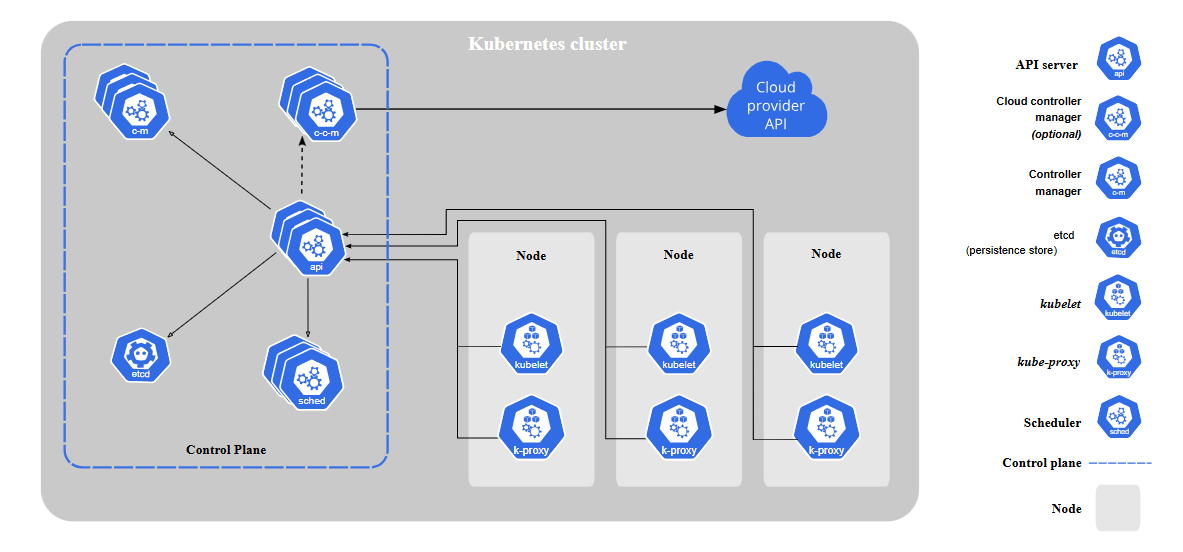
**6. Namespaces**

* Logical partitions in a Kubernetes cluster to divide resources among users and applications.

**7. Ingress**

* Manages external HTTP/HTTPS access to services.
* Provides routing rules for incoming traffic.

**Kubernetes – Architecture**

****

**1. Api Server / kube-api-server**

* It is the main management point of the cluster and is also called as brain of the cluster.
* All the components are directly connected to the API server, they communicate through the API server only and no other component will communicate directly with each other.
* This is the only component that connects and gets access to etcd.
* All the cluster requests are authenticated and authorized by the API server.
* The API server has a watch mechanism for watching the changes in the cluster.

**2. etcd**

* ectd is a distributed, consistent key-value store used for storing the complete cluster information/data.
* ectd contains data such as configuration management of cluster, distributed work and complete cluster information.

**3. scheduler / kube-scheduler**

* The scheduler always watches for a new pod request and decides which worker node this pod should be created.
* Based on the worker node load, affinity, and anti-affinity, the taint configuration pod will be scheduled to a particular node.

**4. Controller manager /control manager / kube-controller**

* It is a daemon that always runs and embeds core control loops known as controllers.
* K8s has some inbuild controllers such as Deployment, DaemonSet, ReplicaSet, Replication controller, node controller, jobs, cronjob, endpoint controller, namespace controller, etc.

**5. Cloud controller manager (CCM)**

* Cloud-controller-manager is a Kubernetes master component that embeds cloud-specific control logic which lets you link your cluster to your cloud provider services.
* Embedding infrastructure services without tight coupling between core components which separates the components that interact with that cloud platform from components that only interact with your cluster.
* These controllers help us to connect with the public cloud provider service and this component is maintained by cloud providers only.
* Cloud Controller Manager is a crucial component responsible for managing interactions between the Kubernetes cluster and the underlying cloud provider's infrastructure.
* Enable Kubernetes to seamlessly interact with various cloud providers without needing deep integration into the Kubernetes codebase itself.

**Worker node components**  
**1. kubelet**

* It is an agent that runs on every worker node and it always watches the API server for pod-related changes running in its worker node.
* kubelet always makes sure that the assigned pods to its worker nodes are running.
* kubelet is the one that communicates with the containerization tool (docker daemon) through docker API (CRI).
* The work of kubelet is to create and run the pods. Always report the status of the worker node and each pod to the API server. (uses a tool called cAdvisor)
* Kubelet is the one that runs probes.

**2. kube service proxy (in k8s service means networking)**

* The service proxy runs on every worker node and is responsible for watching the API server for any changes in service configuration (any network-related configuration).
* Based on the configuration service proxy manages the entire network of worker nodes.

**3. Container runtime interface (CRI)**

* This component initially identifies the container technology and connects it to the kubelet.

**pod**

* pods are the smallest deployable object in Kubernetes.
* pod should contain at least one container and can have n number of containers.
* If the pod contains more than one container all the containers share the same memory assigned to that pod.

[**LINK - ( When a request is sent to Kubernetes cluster using the kubectl )**](https://medium.com/jorgeacetozi/kubernetes-master-components-etcd-api-server-controller-manager-and-scheduler-3a0179fc8186)

**YAML (filetype .yaml or .yml)**

**YAML, which stands for "*Yet Another Markup Language*," is a human-readable data serialization format used for configuration files and data interchange.**

It is like XML and JSON but has easy and simple syntax. YAML is a superset of JSON

Previously we used XML, then JSON came, and now YAML.



* In YAML we write data in key-value pairs. The keys are defined by the tool - k8s and are defined in the camel case.
* To denote the structure Indentation of whitespace is used. We cannot use tabs inside YAML, use white space. (4 whitespace is equal to tab)

**<key>: <value>**

* + - * Normally space is important after the **colon (:)**
      * Camel case is a way of writing phrases without spaces, where the first letter of each word is capitalized, except for the first letter of the first word
      * Pascal Case is similar to Camel Case, but unlike Camel Case, the first letter of each word is capitalized

**YAML supports common types like**

**1. Data Structure:** Lists, Dictionaries

**2. Scalars**:

* **Integer, String, Boolean, floating-point numeric values.**
* Scalars are single values like strings, numbers, or booleans.
* Strings can be enclosed in single quotes ' or double quotes ".

# Scalar values

name: John Smith

age: 30

is\_student: false

# Lists

fruits: ["Apple", "Orange"]

fruits:

- Apple

- Banana

- Orange

# Dictionaries

person:

name: Alice

age: 25

contacts:

- type: email

address: alice@example.com

- type: phone

number: 123-456-7890

**Kubernetes YAML Pod configuration file**  
Other known names - manifest file, spec file, configuration file, yaml file etc.

#Sample pod yaml configuration/spec file

apiVersion: v1

kind: Pod

metadata:

name: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

**apiVersion: v1**

* This is the version of API used to create a k8s object.
* The fields are case-sensitive and YAML uses camelcase.
* The types of APIs are alpha, beta, and stable.

**kind: Pod**

* This field specifies the kind of resource being created. In this example, we are creating a Pod.
* Always object name's first letter is capitalised.

**metadata:**

* This field is used to provide information on the object which we are creating.
* Information such as name, labels, and annotations.

**spec:**

* This is used to do the actual configuration of the object.

**K8S commands**

* Create objects
* Apply configuration to objects
* List objects
* Delete objects

**To create/apply a configuration**  
    kubectl apply -f <file>.yml      
  
                  **(OR)**  
  
    kubectl apply -f <url>  
      
**To list objects**  
    kubectl get <object\_type>  
        ex: List pods - kubectl get pods   
              List deployment - kubectl get deployments  
              
**To delete objects**  
    kubectl delete <object\_type> <object\_name>  
  
                             **(OR)**  
  
    kubectl delete -f <url>  
  
                             **(OR)**  
  
    kubectl delete -f <file>.yml

**To further debug and diagnose cluster problems**  
    kubectl cluster-info dump

**To retrieve the status of the various components that make up the Kubernetes control plane**  
    kubectl get componentstatuses

**To print the supported API resources on the server**  
    kubectl api-resources

**Control Managers in Kubernetes**

Control managers are a set of processes that run inside the **Kubernetes control plane** to ensure the desired state of the cluster. They watch the cluster state, make decisions, and take actions to maintain the cluster's expected behavior. These managers operate as part of the **kube-controller-manager**, which is a single binary running multiple controllers.

**What is a Controller in Kubernetes?**

A **controller** in Kubernetes is a control loop that watches the state of the cluster and makes changes to bring it closer to the desired state defined in a Kubernetes object specification (YAML or API request). Controllers continuously monitor Kubernetes resources and ensure they match the specifications.

For example:

* If a pod crashes, a **ReplicaSet Controller** will detect the deviation from the desired state and create a new pod to replace it.
* If a node goes down, the **Node Controller** detects it and reschedules pods to healthy nodes.

**Types of Controllers in Kubernetes**

Kubernetes has several controllers, each responsible for managing a specific resource or functionality.

**1. Workload Controllers**

These controllers manage applications running in Kubernetes.

* **Deployment Controller** → Ensures that the desired number of pods are running and manages rolling updates.
* **ReplicaSet Controller** → Ensures the correct number of pod replicas are running.
* **StatefulSet Controller** → Manages stateful applications (e.g., databases) ensuring stable pod identities.
* **DaemonSet Controller** → Ensures a copy of a pod runs on every (or selected) node.
* **Job Controller** → Manages one-time batch jobs and ensures they complete successfully.
* **CronJob Controller** → Schedules jobs to run at specific times.

**2. Service and Networking Controllers**

These controllers handle networking and communication within the cluster.

* **Service Controller** → Manages service objects and integrates with cloud provider load balancers.
* **Ingress Controller** → Manages external access to services, such as HTTP routing.
* **EndpointSlice Controller** → Optimizes network endpoint management for better scalability.

**3. Node and Infrastructure Controllers**

These controllers manage cluster infrastructure.

* **Node Controller** → Detects and responds to node failures.
* **Persistent Volume (PV) Controller** → Manages persistent storage resources.
* **Cloud Controller Manager** → Manages cloud provider-specific interactions, like node lifecycle and storage provisioning.

**4. Security and Policy Controllers**

These controllers enforce security policies.

* **Role-Based Access Control (RBAC) Controller** → Enforces Kubernetes role-based access control policies.
* **Network Policy Controller** → Enforces network security policies between pods.

**5. Autoscaling Controllers**

These controllers automatically adjust resources based on demand.

* **Horizontal Pod Autoscaler (HPA)** → Adjusts the number of pod replicas based on CPU/memory utilization.
* **Vertical Pod Autoscaler (VPA)** → Adjusts resource requests and limits of individual pods.
* **Cluster Autoscaler** → Adjusts the number of nodes in the cluster based on workload demand.

**Key Control Manager in Kubernetes**

The main control manager in Kubernetes is:

**1. kube-controller-manager**

* Runs all core controllers within Kubernetes.
* Monitors the cluster and enforces the desired state.
* Handles pod scheduling, node management, job execution, etc.

**2. cloud-controller-manager**

* Separates cloud-specific controllers (e.g., AWS, GCP, Azure) from the main Kubernetes control plane.
* Manages cloud-based storage, networking, and compute resources.

**NODE CONTROLLER**

**Node Controller in Kubernetes**

The **Node Controller** is a critical component of the kube-controller-manager responsible for monitoring the health of Kubernetes nodes and taking corrective actions when nodes fail. It ensures that workloads remain available and that pods are rescheduled appropriately.

**1. Responsibilities of the Node Controller**

The Node Controller performs the following key functions:

**a. Detecting Node Failures**

* It continuously monitors the **heartbeat signals** from nodes.
* Nodes send heartbeats to the Kubernetes API server using:
  + **kubelet** (via NodeStatus updates)
  + **Node Lease** mechanism (lighter alternative introduced in Kubernetes 1.13)
* If a node stops sending heartbeats, the Node Controller marks it as **"NotReady"**.

**b. Handling Unhealthy Nodes**

If a node remains unresponsive for a specific duration (default **40 seconds**), the Node Controller:

* Changes the node's status to **"NotReady"**.
* If the node remains unreachable for a longer period (default **5 minutes**):
  + The node is **evicted** (Pods running on it are scheduled to other nodes if possible).
  + The node is marked as **"Unreachable"**.

**c. Managing Node Lifecycle**

* When a new node is added, the Node Controller registers it.
* When a node is deleted, it ensures workloads are safely rescheduled.
* If a cloud provider is used, it integrates with the **Cloud Controller Manager** to handle automatic node provisioning and termination.

**2. Node Controller Timers and Actions**

| **Timeout** | **Action Taken** |
| --- | --- |
| **Default: 40s** | Node marked as **NotReady** if no heartbeat received. |
| **Default: 5 min** | Pods on the node are **evicted** and scheduled on healthy nodes. |
| **Depends on cloud provider** | If using cloud infrastructure, Kubernetes may remove the failed node automatically. |

**3. Node Conditions Monitored by Node Controller**

The Node Controller monitors and updates node conditions:

| **Condition** | **Description** |
| --- | --- |
| Ready | Node is healthy and accepting pods. |
| MemoryPressure | Node is under high memory usage. |
| DiskPressure | Node's disk space is running low. |
| PIDPressure | Too many processes running on the node. |
| NetworkUnavailable | Network is down or misconfigured. |

If a condition is **critical**, Kubernetes might stop scheduling new workloads on the node.

**4. Node Eviction Process**

When a node is unresponsive:

1. It is marked as NotReady (40s timeout).
2. After **5 minutes**, the node is declared **unreachable**.
3. The Node Controller triggers **pod eviction**:
   * Pods with a **PodDisruptionBudget (PDB)** may not be evicted immediately.
   * If a node recovers, eviction stops.

**5. Node Controller and Cloud Integration**

For cloud-based Kubernetes clusters, the Node Controller interacts with the **Cloud Controller Manager** to:

* Detect if a node is permanently removed (e.g., terminated in AWS/GCP/Azure).
* Automatically remove such nodes from the cluster.

**6. Configuring the Node Controller**

You can configure Node Controller behavior using **flags** in kube-controller-manager:

| **Flag** | **Default** | **Description** |
| --- | --- | --- |
| --node-monitor-period | 5s | Frequency of checking node status. |
| --node-monitor-grace-period | 40s | Time before a node is marked NotReady. |
| --pod-eviction-timeout | 5m0s | Time before evicting pods from an unresponsive node. |

Example of modifying timeouts:

yaml

CopyEdit

spec:

template:

spec:

containers:

- name: kube-controller-manager

command:

- --node-monitor-grace-period=20s

- --pod-eviction-timeout=2m

This configuration:

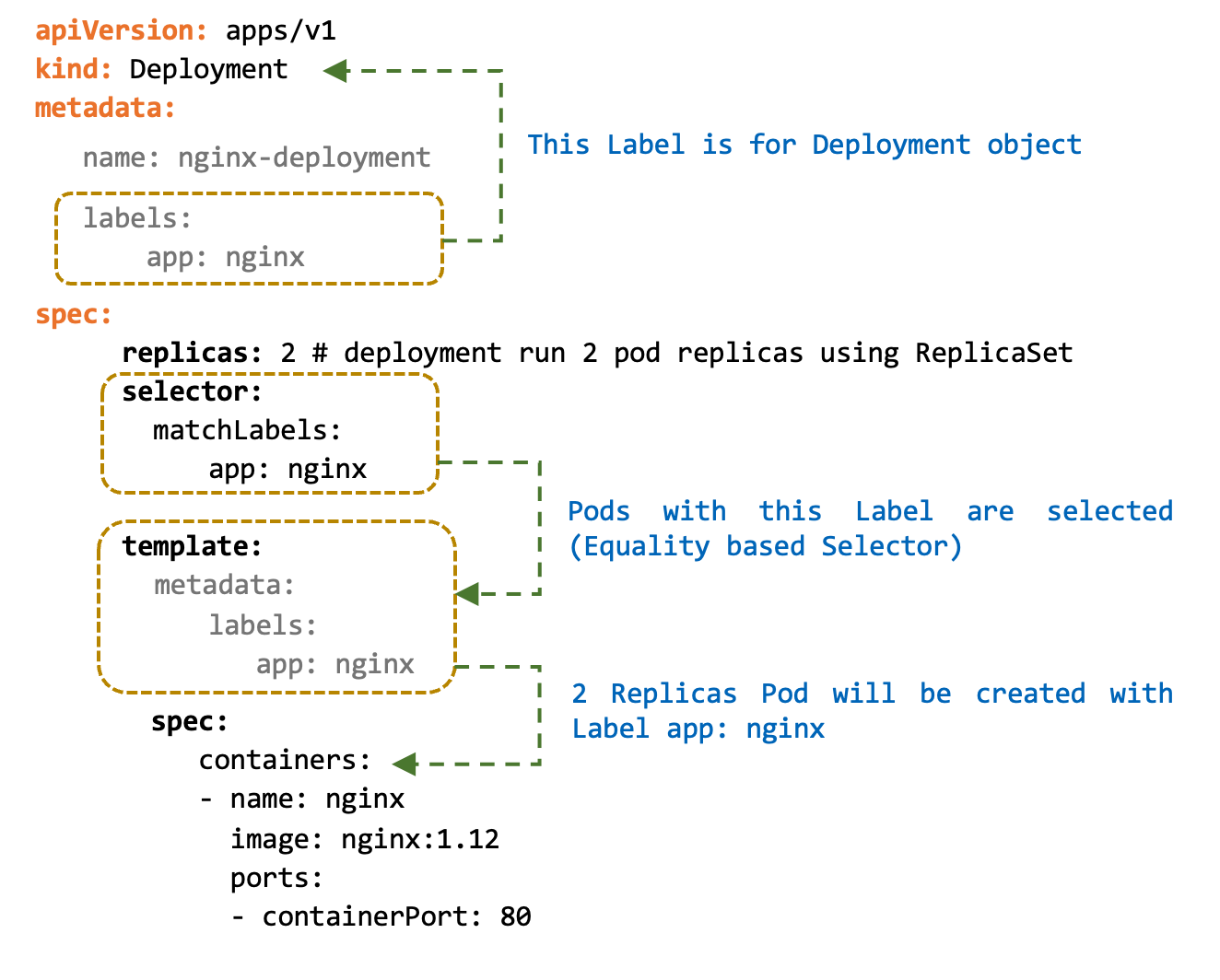
* Marks nodes **NotReady** in **20 seconds**.
* Evicts pods from unresponsive nodes in **2 minutes**.

**7. Best Practices for Node Health Management**

✅ **Use multiple replicas of critical workloads**: Prevent downtime if a node fails.  
✅ **Enable Node Lease feature**: Improves node health monitoring efficiency.  
✅ **Use taints & tolerations**: Prevents critical workloads from running on unreliable nodes.  
✅ **Monitor node conditions**: Use Prometheus/Grafana to track node health.  
✅ **Ensure cloud auto-scaling**: Kubernetes should automatically replace failed nodes.

**Deployment controller / Deployment / Deployment k8s**

* Deployment is used to create replicas of similar kinds of pods and It makes sure that at a given point in time, that number of replicas of pods is always running by using the ReplicaSet controller.
* If we update the configuration of deployment, it will automatically update in all the pod replicas.
* Rollout and Rollback of pod update is possible.
* We can use different deployment strategies for update,  by default it uses RollingUpdate. Other strategies: canary, Recreate, Ramped and Blue-Green
* we can pause the deployment whenever we need
* Deployment internally got its autoscaler which is of type horizontal smaller (hap).



**Deployment Controller in Kubernetes**

The **Deployment Controller** in Kubernetes manages the deployment of applications by ensuring the desired state of an application matches its actual state. It provides declarative updates for Pods and ReplicaSets.

**Key Features of Deployment Controller**

1. **Self-Healing** - If a Pod crashes, the Deployment Controller automatically replaces it.
2. **Rolling Updates** - Allows zero-downtime updates by gradually replacing old pods with new ones.
3. **Rollback** - Supports reverting to a previous version if a deployment fails.
4. **Scaling** - Easily scales applications up or down.
5. **Declarative Configuration** - Uses YAML/JSON manifests to define and update deployments.

**How Deployment Works**

A **Deployment** manages a **ReplicaSet**, which in turn manages Pods.

1. **User defines a Deployment** with a specified number of replicas.
2. **Kubernetes creates a ReplicaSet** to maintain the desired number of Pods.
3. **Deployment Controller monitors and ensures** the correct number of Pods are always running.
4. **During an update**, a new ReplicaSet is created, and the old Pods are replaced gradually.

**Basic Deployment Commands**

**1. Create a Deployment**

bash

CopyEdit

kubectl create deployment my-app --image=nginx --replicas=3

**2. View Deployments**

bash

CopyEdit

kubectl get deployments

kubectl describe deployment my-app

**3. Scale a Deployment**

bash

CopyEdit

kubectl scale deployment my-app --replicas=5

**4. Update a Deployment (Rolling Update)**

bash

CopyEdit

kubectl set image deployment/my-app nginx=nginx:latest

**5. Rollback a Deployment**

bash

CopyEdit

kubectl rollout undo deployment my-app

**6. Delete a Deployment**

bash

CopyEdit

kubectl delete deployment my-app

**Example Deployment YAML**

You can define a Deployment using a YAML file:

yaml

CopyEdit

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-container

image: nginx:latest

ports:

- containerPort: 80

Apply this Deployment using:

bash

CopyEdit

kubectl apply -f my-deployment.yaml

**Labels and selectors**

**Labels (Identifiable metadata):**

A **label** is a key-value pair assigned to Kubernetes objects (Pods, Nodes, Services, etc.) to categorize and organize them. Labels do not have any unique constraints and can be freely assigned or modified.

* We can provide labels to any object in K8S
* We can have the same label on multiple objects/resources in K8S.
* The label key must consist of alphanumeric characters, dashes (-), or periods (.), and it must not start or end with a dash or period.
* Labels are commonly used for organizing and categorizing resources. They help in grouping related resources for querying, filtering, and organizing purposes.
* K8S uses the kubernetes.io/ prefix for built-in labels, as it is reserved for Kubernetes, we should not use it.

**Key Rules:**

* Label keys are case-sensitive, and uppercase characters are not allowed.
* The maximum length for a label key is 63 characters.
* Labels are used to identify and group resources in K8S based on specific characteristics or attributes.

**Value Rules:**

* The label value can be an empty string ("").
* The label value can consist of any characters, including alphanumeric characters, dashes, periods, underscores, and colons.
* The maximum length for a label value is 63 characters.  
       **To list labels of any object:**kubectl get <object\_type> <object\_name> --show-labels

**Examples:**

* release: stable, release: canary
* environment: dev, environment: qa, environment: production
* tier: frontend, tier: backend, tier: cache
* partition: customerA, partition: customerB
* track: daily, track: weekly

**Selectors**

Selectors are used to select, filter and identify the labelled objects.

A **label selector** is a query that selects Kubernetes resources based on their labels

selectors:  
       matchLabels:  
             <label>

**Types of selectors**  
**1. equality-based**

* In this selector, we can use only one operator which is equal\_to (=, ==) or (!=) not\_equal
* It looks for an exact match for the label   
     app = nginx  or app: nginx  
     app != nginx

**2. set-based**

* Set-based selectors allow the selection of resources based on multiple values for a label key.
* It uses set operators such as in, not in, exists, and does not exist to perform the selection.
* The exists expression checks whether a label with a specific key is present on a resource, *regardless of its value*. It doesn't care what the value of the label is, only that the label key itself exists.
* Select pods with the label environment having values development or testing:  
  **Note:**Avoid using characters like comma (,), semicolon (;), or equals (=) in label keys or values used for selectors, as these characters have specific meanings in set-based selectors.

matchExpressions:

- key: <label\_key> # The label key to match

operator: <operator> # The operator to use (In, NotIn, Exists, DoesNotExist)

values: # Optional: A list of values for In and NotIn operators

- <value1>

- <value2>

...

matchExpressions:

- key: <label\_key>

operator: Exists

**Annotations**

* Kubernetes annotations are a way to attach arbitrary non-identifying metadata to Kubernetes objects.
* Annotations are not used to identify or select objects (we cannot use selectors on annotations), but rather to provide additional information that can be used by tools, libraries, or users.
* Annotations are key-value pairs, where both the key and value are strings.

**Common Uses of Annotations**

1. **Build/Release Information**: Annotations can store build/release information, such as Git commit hashes, timestamps, or version numbers.
2. **Configuration Details**: They can store configuration details that are not part of the object's spec.
3. **Tooling Information**: Annotations can be used by tools to store information about the object, such as the tool that created it or the last time it was modified.
4. **Custom Workflow Logic**: They can be used to trigger custom logic in controllers or operators.
5. **Audit Logging**: Annotations can store audit-related information, such as the user who last modified the object.

apiVersion: apps/v1

kind: Deployment

metadata:

name: my-app

annotations:

buildVersion: "1.2.3"

gitCommit: "a1b2c3d4e5f6"

description: "This deployment is for the production environment."

imageregistry: "https://registry.hub.docker.com"

spec:

**ReplicaSet vs Replication controller**

* Both ensure that at a given point in time, the specified number of replicas are always running.
* The replication controller is a very old controller now it is replaced by ReplicaSet.
* The only difference between them is Replication controller supports only equality-based selectors but ReplicaSet supports both set-based and equality-based selectors.

**ReplicaSet** and **ReplicationController** in Kubernetes serve the same fundamental purpose: ensuring that a specified number of pod replicas are running at all times. However, **ReplicaSet** is the more modern and preferred option. Here’s a breakdown of their differences:

**1. ReplicaSet vs. ReplicationController**

| **Feature** | **ReplicaSet** | **ReplicationController** |
| --- | --- | --- |
| **API Version** | apps/v1 | v1 |
| **Label Selector** | Supports **set-based** selectors and equality-based selectors | Only supports **equality-based** selectors |
| **Preferred Use Case** | Recommended for new deployments | Legacy feature, being phased out |
| **Usage with Deployments** | Used by Deployment objects | Not used by Deployment objects |
|  |  |  |
|  |  |  |

1. **Key Differences**

**Label Selector:**

* **ReplicationController** only supports **equality-based selectors**, meaning you can only match labels exactly (app=nginx).
* **ReplicaSet** supports both **equality-based** and **set-based selectors** (app in (nginx, apache)), making it more flexible.

**Deprecation & Modern Kubernetes:**

* **ReplicationController** is older and considered **deprecated** in favor of ReplicaSet.
* **Deployments** (which are widely used for managing pods) internally manage ReplicaSets, **not ReplicationControllers**.

**3. When to Use Which?**

* Use **ReplicaSet** for standalone replication needs.
* Use **Deployment**, which manages ReplicaSets automatically, for most real-world applications.
* Avoid **ReplicationController**, as it is outdated.

**namespaces (ns)**

In **Kubernetes (K8s)**, a **Namespace** is a way to divide cluster resources between multiple users or teams. It allows for logical separation within the same cluster, making it easier to manage large environments.

* k8s namespaces apply abstraction/isolation to support multiple virtual clusters of k8s objects within the same physical cluster.
* Every object in k8s must be in a namespace.
* If we don't specify a namespace, objects will be created in the default namespace of k8s.
* namespaces are cluster-level.
* Namespaces are only hidden from each other but not fully isolated because one service in a namespace can talk to another service in another namespace using the full name (service/<service\_name>) followed by the namespace name

usage: we can apply environment-based logical separation on a cluster.   
         
**Type of default NS**  
**1. default**      - This NS is used for all the objects which do not belong to any other namespace.  
      - If we don't specify any namespace while creating an object in k8s then that object will be created in the default namespace.

**2. kube-system**  
      - This namespace is always used for objects created by the k8s system.  
        
**3. kube-public**  
      - The objects in this namespace are available or accessible to all.  
      - All the objects in this namespace are made public.

**4. kube-node-lease**  
      - This namespace holds lease objects associated with each node.  
      - Node lease allows the kubelet to send heartbeats so that the control plane can detect node failure.

**To list namespaces**  
       kubectl get namespaces   
       kubectl get ns

**To list objects in a namespace**  
       kubectl get -n <namespace\_name> <object\_type>  
     
**To list objects in all namespaces**  
       kubectl get --all-namespaces <object\_type>  
       kubectl get -A <object\_type>

**To create a namespace**  
       kubectl create ns <namespace\_name>  
         
   **To create an object in a namespace**  
       1. In metadata:  
           namespace: <namespace\_name>

       2. While apply       
           kubectl apply -f <spec\_file>.yml -n <namespace\_name>

       Note: If we provide a namespace in both spec files and apply, apply command check and compare the namespace in the spec file if they are not the same k8s won't allow us to create the object.

**To List all namespaced objects/resources**kubectl api-resources --namespaced=true

**To List all non namespaced objects/resources**kubectl api-resources --namespaced=false

**Why do we need namespaces in K8S?**

1. **Isolation**: Namespaces provide a way to isolate resources within a Kubernetes cluster. Each namespace acts like a virtual cluster within the larger Kubernetes cluster.
2. **Resource Quotas**: Namespaces allow you to enforce resource quotas at a namespace level. This means you can limit the amount of CPU, memory, and other resources that can be consumed by containers within a specific namespace. This helps prevent one application from consuming all available resources and affecting other applications running in the cluster.
3. **Access Control**: Kubernetes RBAC (Role-Based Access Control) can be applied at the namespace level, allowing fine-grained control over who can access and manipulate resources within a specific namespace. This enables teams to have their own namespaces with restricted access, providing a multi-tenancy environment within the Kubernetes cluster.
4. **Organizational Structure**: Namespaces allow you to organize and group related resources together. For example, you can have separate namespaces for different environments (e.g., development, staging, production) or for different teams or projects.

**DaemonSet**

A **DaemonSet** in Kubernetes ensures that a specific pod runs on all (or some) nodes in a cluster. This is useful for deploying system-level services that should run on every node, such as log collectors, monitoring agents, or networking tools.

**Key Features of DaemonSet**

* Ensures a pod runs on all or specific nodes.
* Automatically adds pods when new nodes join the cluster.
* Removes pods when nodes leave or the DaemonSet is deleted.
* Uses a rolling update strategy to update pods gradually.

**Common Use Cases**

1. **Log Collection** (e.g., Fluentd, Filebeat)
2. **Monitoring Agents** (e.g., Prometheus Node Exporter)
3. **Network Services** (e.g., CNI plugins like Calico, Flannel)
4. **Security Tools** (e.g., Falco)

**Creating a DaemonSet**

Here’s an example YAML file for a simple DaemonSet running **Fluentd**:

yaml

CopyEdit

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: fluentd

labels:

app: fluentd

spec:

selector:

matchLabels:

app: fluentd

template:

metadata:

labels:

app: fluentd

spec:

containers:

- name: fluentd

image: fluent/fluentd:v1.14.0

resources:

limits:

memory: "200Mi"

cpu: "100m"

volumeMounts:

- name: varlog

mountPath: /var/log

volumes:

- name: varlog

hostPath:

path: /var/log

**Key Fields in DaemonSet**

* **selector**: Defines how to match pods with this DaemonSet.
* **template**: Specifies the pod template, similar to a Deployment.
* **hostPath volumes**: Used to access host files, useful for logging and monitoring.

**How to Manage DaemonSets**

**List all DaemonSets**

sh

CopyEdit

kubectl get daemonsets -n kube-system

**Describe a DaemonSet**

sh

CopyEdit

kubectl describe daemonset fluentd

**Delete a DaemonSet**

sh

CopyEdit

kubectl delete daemonset fluentd

**DaemonSet vs Deployment**

| **Feature** | **DaemonSet** | **Deployment** |
| --- | --- | --- |
| Pod Scheduling | Runs on all (or selected) nodes | Runs based on replicas |
| Use Case | System-wide services | Application workloads |
| Scaling | Scales with nodes | User-defined replicas |

**2-Tire vs 3-Tire**

**2-Tier (Client-Server) Architecture:**

In a 2-tier architecture, the client (presentation tier) directly interacts with the data store (data tier). This typically involves a direct database connection from the client application. Application logic can reside on either the client or the server, leading to potential challenges in maintainability and code reuse.

* **Key characteristics:** Direct client-database interaction. Limited scalability due to database connection limitations and potential single points of failure. Security vulnerabilities due to direct exposure of the database. Maintenance complexity due to distributed application logic.

**3-Tier Architecture:**

The 3-tier architecture introduces a middle tier – the application tier (or logic tier) – between the client and the data store. The client interacts with the application tier, which then communicates with the database. This separation of concerns allows for better organization and modularity. The application tier handles business logic, data validation, security, and transaction management.

* **Key characteristics:** Separation of concerns (presentation, logic, data). Improved scalability as the application tier can be scaled independently. Enhanced security by abstracting database access. Simplified maintenance with centralized application logic. Increased flexibility and code reusability.

**Technical Deep Dive (Optional, for more in-depth discussion):**

* You can mention specific technologies associated with each tier. For example, in a web application context:
  + **Presentation Tier:** HTML, CSS, JavaScript, React, Angular, etc.
  + **Application Tier:** Java, Python, Node.js, .NET, Spring Boot, etc.
  + **Data Tier:** Relational databases (PostgreSQL, MySQL, SQL Server), NoSQL databases (MongoDB, Cassandra), etc.
* Discuss the benefits of using an ORM (Object-Relational Mapper) in the application tier to interact with the database.
* Explain how caching mechanisms at the application tier can improve performance.
* Mention load balancing as a technique to distribute client requests across multiple instances of the application tier for scalability.
* Discuss API design (RESTful APIs, GraphQL) for communication between the client and application tiers.
* **3-tier:** The *building* has three floors (presentation, application, data).
* **MVC:** The *layout* of the top floor (presentation) is organized into different rooms (Model, View, Controller).

**How they relate:**

In a typical 3-tier web application:

1. The *presentation tier* is often implemented using the MVC pattern. The View handles displaying the UI, the Controller handles user input, and the Model manages the data for the UI.
2. The *application tier* (the middle tier) handles business logic and interacts with the data tier. This tier might have its own internal structure, but it's not usually described as MVC. It could use other design patterns.
3. The *data tier* is the database.

**Key differences:**

* **Scope:** 3-tier is a system-level architecture; MVC is a design pattern for a specific part of the system (usually the presentation tier).
* **Concerns:** 3-tier is about physical separation and deployment; MVC is about separating concerns within the user interface.
* **Level of abstraction:** 3-tier is a higher-level concept; MVC is a more detailed design pattern.

**In short:** You can have a 3-tier application where the presentation tier is *not* using MVC. You could also have a 2-tier application where the client-side *is* using MVC. However, the most common and effective approach for modern web applications is to combine them: use a 3-tier architecture for the overall system structure and use the MVC pattern to organize the presentation tier.

**Monolithic Application**

A monolithic application is a single, unified software system where all components—such as the user interface, business logic, and database access—are tightly integrated into one codebase. It is a traditional way of building applications where all functions are deployed as a single unit**.**

* In a monolithic architecture, the entire application is designed, built, deployed, and maintained as a single unit that has a single code base with multiple modules.
* All components of the application, such as the user interface, business logic, and data access layer, are tightly coupled together within a single codebase.
* Scaling a monolithic application can be challenging, as all components are scaled together, and even if only one component requires more resources, we need to scale the complete application.
* Monolithic architectures are often simpler to develop and deploy initially, but they can become unwieldy and difficult to maintain as the application grows.

**Microservices**

Microservices is an architectural style where an application is developed as a collection of small, independent services, each handling a specific business function. These services communicate with each other using APIs or messaging systems.

* Microservices architecture decomposes an application into a set of small, independent services, each running in its own main process, DB connection and communicating with lightweight mechanisms, often HTTP/REST or messaging protocols.
* Each service in a microservices architecture is focused on a specific business capability and can be developed, deployed, and scaled independently.
* Microservices enable better scalability, as resources can be allocated more efficiently to the parts of the application that need them most.
* However, microservices introduce additional complexity in terms of deployment, monitoring, and coordination between services.