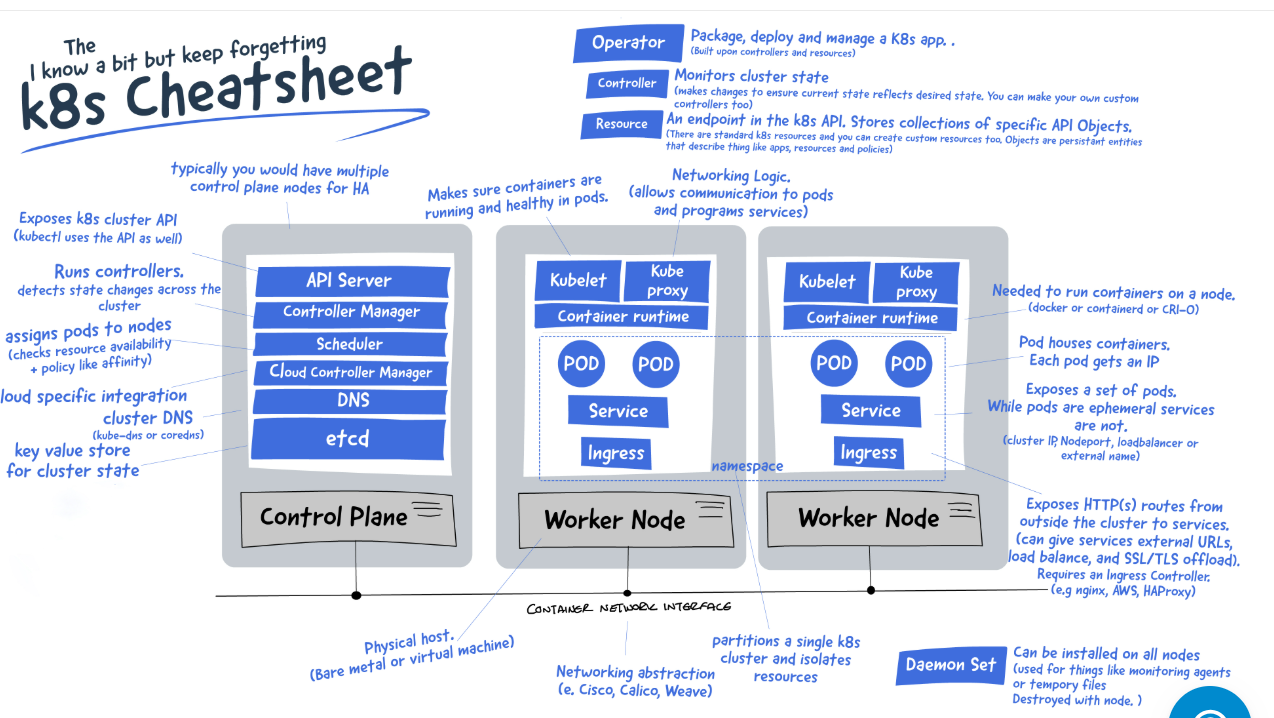
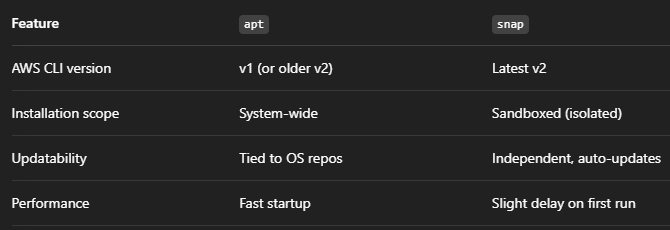
**Kubernetes**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Kops setup:

We create an ec2 instance named kops. This instance won’t be part of the cluster, we are just using it to launch the kubernetes cluster. Also, create an access key for the current IAM user since we need access to AWS CLI.

Now, ssh into the instance and then do **snap install aws-cli --classic.**

**Why not apt install aws-cli --classic?**  


**aws configure**

**ssh-keygen**

Search for install kops on your browser and download it in the instance.

**kops** #to check whether kops is installed or not.

Search for install kubectl on your browser and download it in the instance.

Create a S3 bucket (say xyz) with all the presets as they are.

Open, goDaddy.com and check your domain name (say sunnysb.xyz).

Create a public hosted zone in Route 53 with name as k8.sunnysb.xyz. Now see the record which contains NS, it denotes there are 4 authoritative DNS servers for k8.sunnysb.xyz

Now the DNS servers will check whether the domain sunnysb.xyz has a record named k8 or not, which by default there isn’t. So, we go to goDaddy.com. Then, we go to DNS and we add 4 ‘NS’ records with name as ‘k8’ and in the value the DNS servers mentioned in AWS.

**kops create cluster --name=k8.sunnysb.xyz --state=s3://myk8bucket675 --zones=us-east-1a,us-east-1b --node-count=2 --node-size=t3.small --control-plane-size=t3.medium --dns-zone=k8.sunnysb.xyz --node-volume-size=12 --control-plane-volume-size=12 --ssh-public-key ~/.ssh/id\_ed25519.pub**

**kops update cluster --name=k8.sunnysb.xyz --state=s3://myk8bucket675 --yes --admin**

**kops validate cluster --name=k8.sunnysb.xyz --state=s3://myk8bucket675** #check in 10 mins

Finally, cluster setup is done.

**ls –a**

**cat .kube/config**  #you will see server: [https://api.k8.sunnysb.xyz](https://api.k8.sunnysb.xyz/).This must be there for smooth working.

Now, go to Route 53 in AWS and you will see **api.k8.sunnysb.xyz** (corresponds to public ip of master node) and **api.internal.k8.sunnysb.xyz** (corresponds to private ip of master node). Moreover, a custom VPC is also made.

**kops delete cluster --name=k8.sunnysb.xyz --state=s3://myk8bucket675 --yes**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**kubectl get nodes -o wide ->** lists all the nodes in the cluster with more information.

**kubectl describe node <node\_name> ->** gives detailed information about a specific node in the cluster.

**kubectl drain <node> ->** It is used to safely evict all pods from a node, in preparation for maintenance.

**kubectl** **top nodes ->** shows real-time resource usage (CPU and memory) for all the nodes in the cluster.

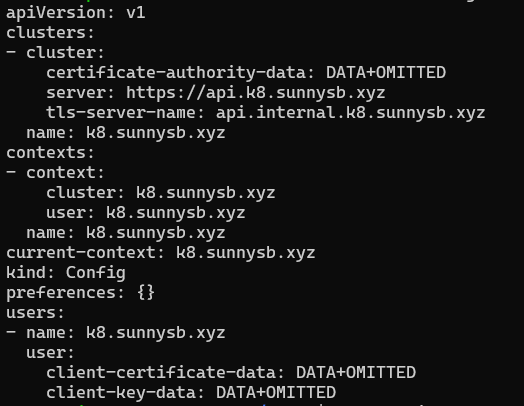
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The **kubeconfig file** is a critical config file for interacting with a Kubernetes cluster using tools like kubectl, Helm etc. By default, location is **~/.kube/config,** but you can specify your own with --kubeconfig tag.

The kubeconfig file contains cluster access information. Tells kubectl :

* Which cluster to connect to
* Which user credentials to use
* Which context (cluster-user-namespace combo) to use

-> Do either **cat ~/.kube/config** or **kubectl view config.** Output:



A **context** links:

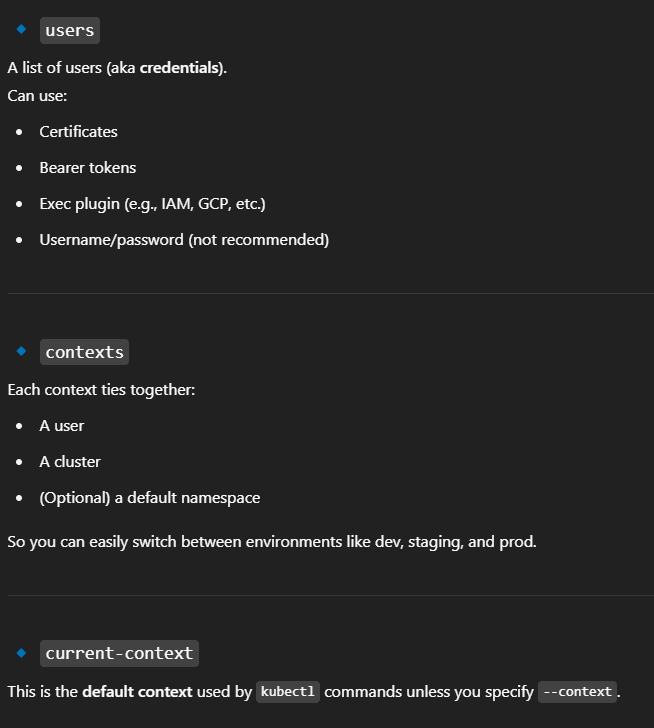
* A **cluster** (where to send requests)
* A **user** (how to authenticate)
* (Optional) A **namespace**

General format of kubeconfig file:



Explaining components obtained from above:

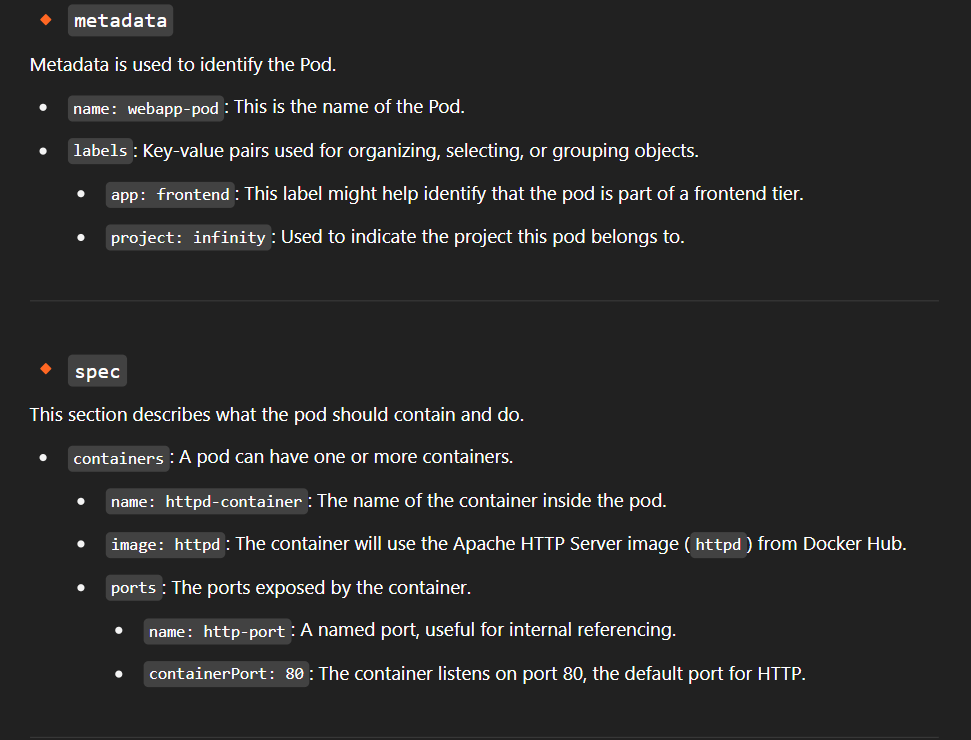
1.**clusters** -> A list of Kubernetes clusters you can talk to.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Kubernetes follow a one-container-per-pod policy. But there is something known as init containers in a pod making two or three containers per pod which is absolutely still fine.  
An **Init Container** is a special type of container that **runs before** the main application containers in a Pod. It's designed to **set up the environment** or do some **preliminary work** before the main containers start. It’s often used to avoid a "race condition" where a pod starts too early before the environment is ready.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Explanation of kubernetes pod-setup.yml file (pg 33 in other pdf notes):**



**kubectl create –f pod-setup.yml** -> creates the pod from manifest file. If the resource (pod) already exists, error would be thrown.

**kubectl apply –f pod-setup.yml** -> used to create or update the pod as per latest definition in yaml file. No chance of error.

**kubectl get pods** -> shows the available pods. The “Ready” column with 1/1 denotes that pod has 1 container and is in ready state. There is a ‘–A' tag which will tell pod and it’s namespace too. Also, **-o wide** and **–o yaml** options can be used.

**kubectl describe pod webapp-pod** -> get details about the pod.

**kubectl get pod webapp-pod** **-o yaml > webpod-definitions.yml** -> get details about the pod in yml format and save it in some yml file since sometimes we cannot directly edit the manifest file of pod. Now, you can delete the error-prone pod and launch a new one with correct manifest file. Note: Even though words “yaml” and “yml” are interchangeably used but still if in this command we write –o yml, error will be thrown as kubectl needs full “yaml” name as convention.

**kubectl exec -it webapp-pod -- sh ->** to log in inside the container.

**kubectl edit pod webapp-pod** -> used to edit the pod’s manifest file directly (sometimes does not work). However, editing the pod directly is not recommended, instead if the pod is part of the deployment, then updates should be made in the deployment. Moreover, if the pod is also not part of any deployment, then pod should be deleted and recreated.

**kubectl exec -it webapp-pod –c httpd-container -- /bin/bash ->** to log in inside specific container. This option is used when a pod has multiple containers.

**kubectl logs webapp-pod** -> get STDOUT of processes running in the container of your pod. Useful for debugging.

**kubectl delete pod webapp-pod** -> deletes the pod.

**kubectl delete pods --all -n <namespace>** -> deletes all the pod in particular namespace.

**kubectl delete –f pod-setup.yml -**> deletes the pod and all the associated resources mentioned in yml file.

**kubectl run <pod\_name> --image=<image\_name> --dry-run=client –o yaml ->** this command will not create a pod but will generate the pod’s manifest file in output.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In Kubernetes, *namespaces* provide a mechanism for isolating groups of resources within a single cluster. Namespace-based scoping is applicable only for namespaced objects (e.g. Deployments, Services, etc) and not for cluster-wide objects (e.g. StorageClass, Nodes, PersistentVolumes, etc).

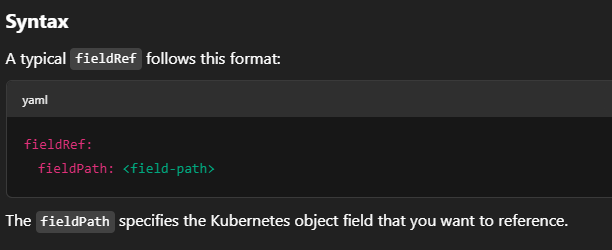
Remember, that if namespace name was not specified in resource manifest, the default namespace will be used. You can change resource's namespace after the resource was created.

**kubectl get namespaces** -> lists all the available namespaces in the cluster.

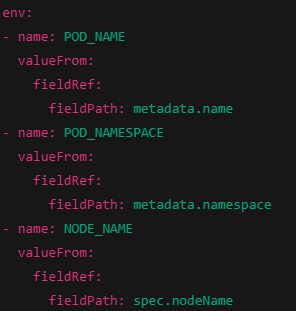
**kubectl create namespace dev** -> creates a new namespace named dev.

**kubectl delete namespace dev** -> deletes the namespace and all the resources inside it. Instead of whole word “namespace” “ns” can also be written.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In Kubernetes, **fieldRef** is a reference to a specific field of an object in the Kubernetes cluster. It allows you to extract values dynamically from the Kubernetes resource itself and use those values in configurations like environment variables.



Example:



The **fieldPath** can reference various fields in Kubernetes objects. Here are some commonly used paths:

* **metadata.name**: The name of the pod.
* **metadata.namespace**: The namespace in which the pod resides.
* **status.podIP**: The pod's IP address.
* **spec.nodeName**: The name of the node where the pod is running.
* **status.hostIP**: The IP address of the node.
* **metadata.uid**: The UID of the pod.
* **spec.serviceAccountName**: The service account used by the pod.

Kubernetes resources are dynamic in nature (e.g., pod names, pod IPs, node names). Using fieldRef allows your configuration to adapt to changes in these dynamic properties without hardcoding them.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Static Pods in Kubernetes:**

In Kubernetes, Static Pods are a special type of pod that is managed directly by the kubelet on a specific node, rather than by the Kubernetes control plane (e.g., the API server, scheduler, and controller manager). Unlike regular pods, which are part of a deployment, replicaset, or daemonset, static pods are defined by configuration files located directly on the node's filesystem. Static pods are managed in a way that is independent of the Kubernetes control plane, meaning the control plane is unaware of their existence (no API server interaction).

**Key Characteristics of Static Pods:**

* **Kubelet Managed**: The kubelet is responsible for creating, updating, and deleting static pods, but it does so directly without involving the Kubernetes API server.
* **No Controller**: Static pods are not controlled by a higher-level controller like Deployments or ReplicaSets. This means they do not benefit from features like auto-scaling, self-healing, or rolling updates provided by these controllers.
* **Direct Configuration**: Static pods are defined by YAML files that are placed on the node itself, and the kubelet reads these files to create and manage the pods.

**How Static Pods Work**

1. **Pod Definition File**: A static pod is defined by a YAML or JSON file located in a specific directory on the node. Typically, this directory is **/etc/kubernetes/manifests/**, but the location can be customized through the kubelet's configuration.
2. **Kubelet’s Role**: The kubelet constantly monitors the directory for changes to the static pod definition files. When a static pod is defined or updated, the kubelet will:
   1. Create or update the pod based on the file.
   2. Ensure that the pod is always running by restarting it if it crashes or is deleted.
   3. Periodically check the status of the pod and make sure it's healthy.
3. **Pod Visibility**: Static pods can be seen in the API server, but this is only because the kubelet periodically registers them as "API objects" (like a normal pod) to allow visibility within the Kubernetes cluster. However, they are not managed by the API server directly, unlike regular pods.

Static pods have exact same manifests file like regular pods, only difference is that the static pods manifest file is to be placed by ssh-ing into the node and in the directory mentioned above. Note: the manifest file of static pod must be in a namespace called static, if it does not exist, create it using kubectl create namespace static.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

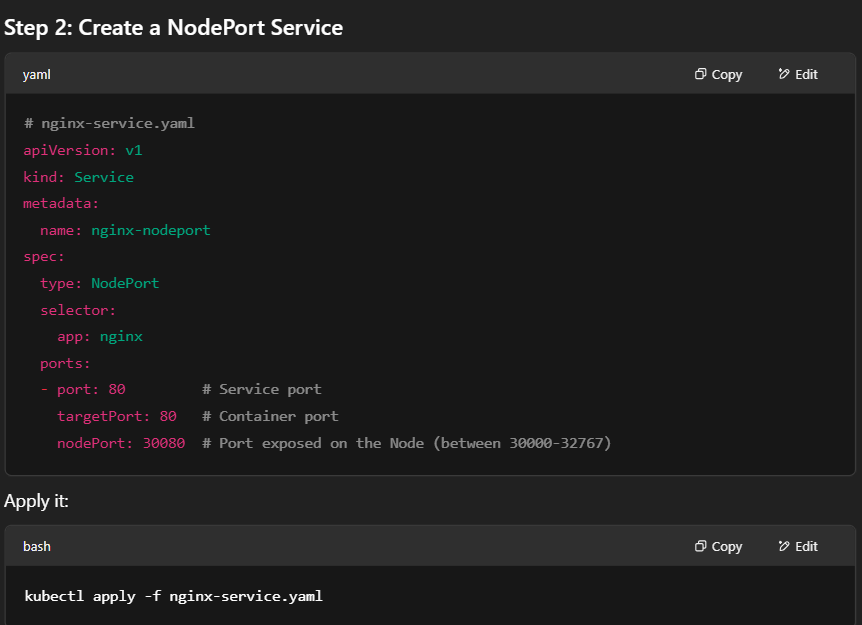
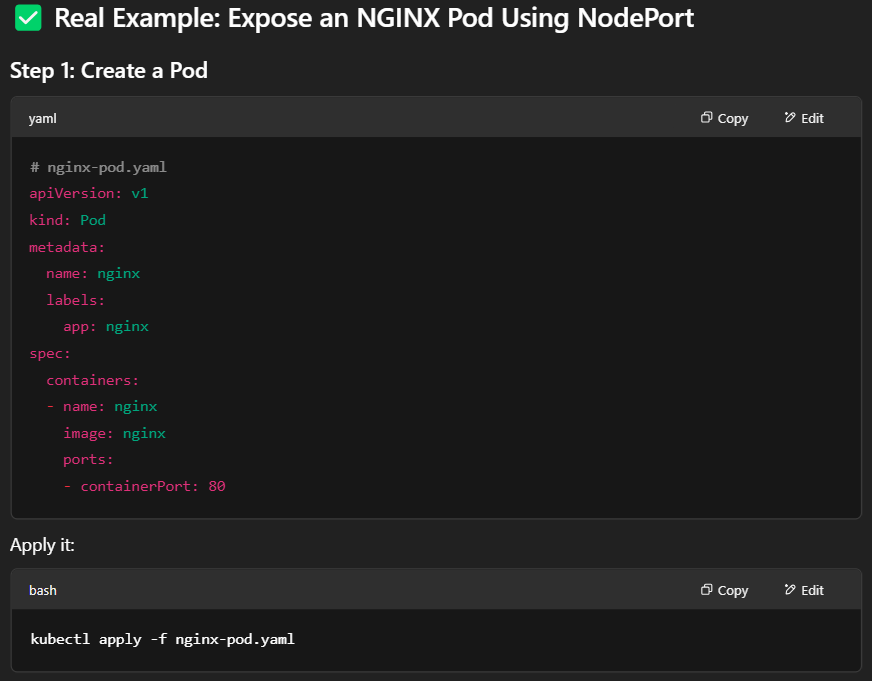
In Kubernetes, a **Service** is an abstraction which defines a logical set of Pods and a policy by which to access them — often via a stable IP address and DNS name. It is not a pod or a container or a node, so it’s across the whole cluster like some proxy rules. It is of 3 types: NodePort, ClusterIP and LoadBalancer.

Pods are ephemeral — they can die and restart, and their IP addresses can change. A **Service** solves this by giving clients a stable way to access them.

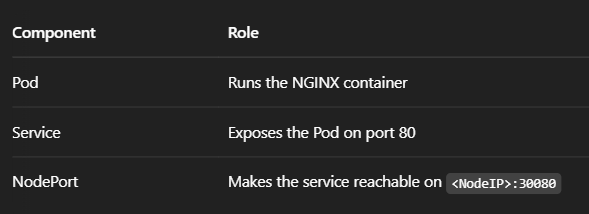
A **NodePort** service makes a pod accessible **outside the cluster** by exposing it on a static port **on each node in the cluster.** To check Nodeport service, just check with any of your node ip alongwith the nodeport on browser. (Make sure to allow all traffic from your ip).

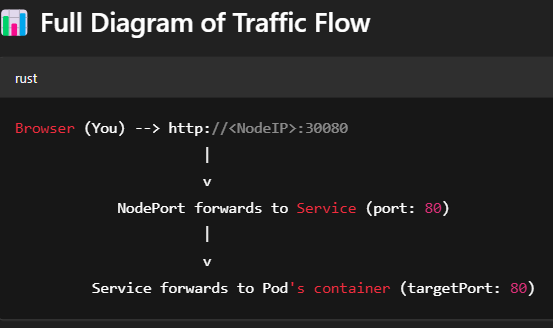
**📌 Characteristics:**

* Exposes the service on **<NodeIP>:<NodePort>**.
* Default range of NodePort is 30000-32767.
* Routes traffic from the NodePort to the **Service**, and then to matching **Pods**.



Now, you can access the service by http://<any-node-ip>:300080





**kubectl get svc** -> lists down the services.

**kubectl describe svc <service\_name>**-> explain the service features.

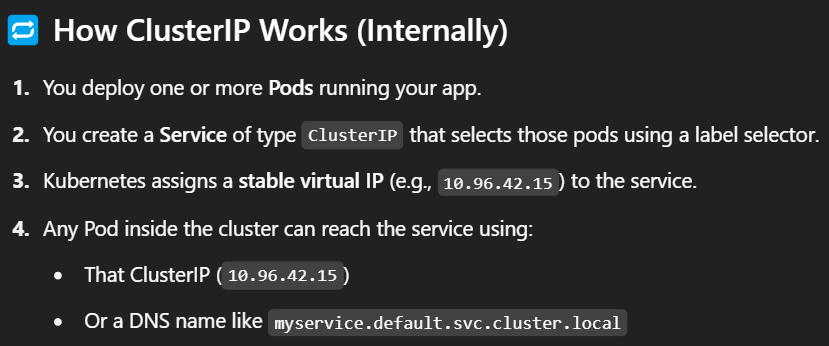
**kubectl delete svc <service\_name>** -> delete the service.

In kubernetes, you can write both pod definitions and service definitions in a single yaml file. Moreover, you can stack any no of k8 resources in a single yaml file with the yaml document separator “---”.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

A **ClusterIP** service is used to **expose a set of Pods internally within the cluster**. It is the **default type of Kubernetes Service**. It is used when you want to connect services inside the cluster. It:

* Creates a **virtual IP address** (ClusterIP).
* Makes your pods accessible **only within the cluster**.
* **Cannot be accessed from outside the cluster** (unlike NodePort or LoadBalancer).



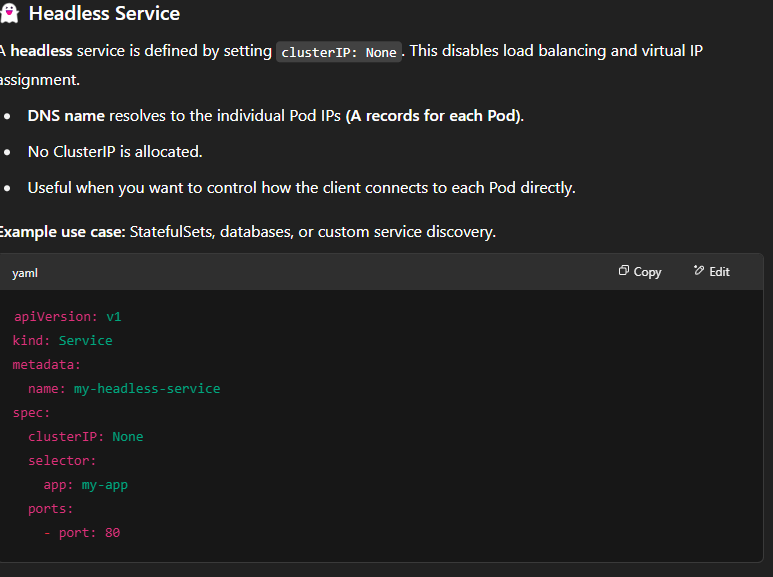
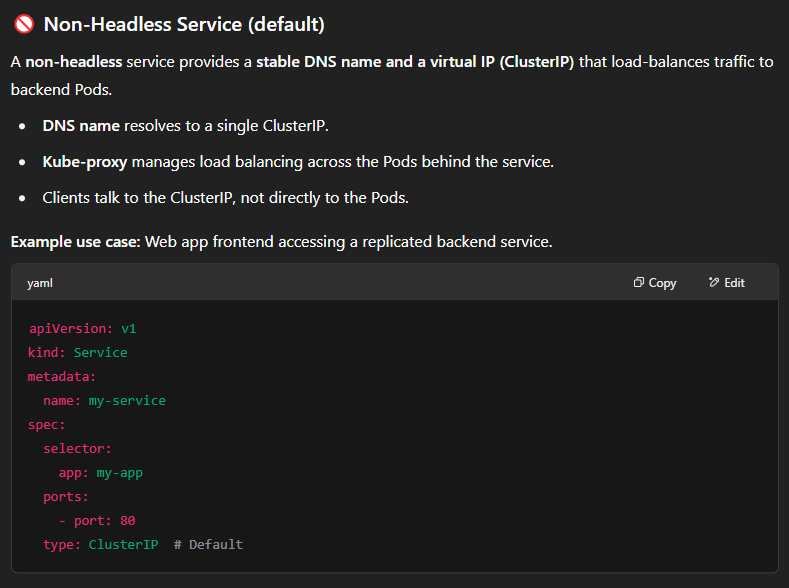
A **LoadBalancer service** exposes your application **externally using a cloud provider's load balancer**.

When you create a LoadBalancer service:

* Kubernetes provisions an **external IP address**.
* Incoming traffic hits that IP and is **load balanced across the Pods** behind the service.
* The service also behaves like a ClusterIP and a NodePort behind the scenes.

Creating a loadbalancer service: Check files in k8 folder. Apply kubectl create command and check on browser with DNS of LoadBalancer.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



A **ReplicaSet** in Kubernetes ensures that a specified number of **pod replicas** are running at any given time. It is used to maintain the desired state of a set of identical pods. If a pod crashes or is deleted, the ReplicaSet will automatically create a new one to replace it.

**kubectl get rs** -> view ReplicaSets  
**kubectl delete rs <ReplicaSet\_name>** -> delete the ReplicaSet

Refer yml file from notes.

Imp: The “template” directive is important in ReplicaSet yml file as it defines the blueprint of the pod that will be created and managed. Inside template.metadata.labels.app the value must be same as is the value mentioned above in selector.matchLabels.app .   
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

A **Deployment** in Kubernetes provides declarative updates for **Pods** and **ReplicaSets**. You describe your desired state (like how many replicas, what container image, labels, etc.), and the **Deployment controller** makes it happen and maintains that state over time.

**What Does a Deployment Do?**

**1. Creates Pods via a ReplicaSet**

* It creates a ReplicaSet.
* That ReplicaSet creates and manages the Pods.

**2. Monitors the Application**

* If a Pod fails or is deleted, the ReplicaSet ensures a new one is created.

**3. Handles Rolling Updates**

* Deployments can update the container image or configuration **without downtime** using a **rolling update** strategy.
* It gradually replaces old Pods with new ones.

**4. Rollbacks**

* You can roll back to a previous version with kubectl rollout undo deployment my-app.

**5. Version History**

* Each change creates a new ReplicaSet.
* Older ReplicaSets are retained for history unless explicitly removed.

Refer deployment yaml file from materials.

**kubectl apply -f deployment.yaml**

**kubectl get deploy ->** see the deployment.

**kubectl get rs ->** see the replica created from the deployment.

**kubectl get pods** -> you will see 3 pods and if you describe these pods you will see nginx version 1.14.2

**kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1 ->** used to change the image version through CLI. The first nginx in the command refers to the container name.

Imp: It is imperative way, hence we should always make changes in the deployment file only and then do kubectl apply. See below command.

**kubectl get deploy <deployment\_name> -o yaml > latest-deploy.yaml** -> get details in yaml format of a deployment and save it in a file. Used when doing changes in the image deployment.

You want to change in deploy.yaml file:

1-apply just above command.

2-make change in latest-deploy.yaml file (as making changes in deploy.yaml file only causes issues when doing kubectl apply)

3-**kubectl apply –f latest-deploy.yaml**

**kubectl get rs** -> you will see new replicaSet, thereby denoting new deployment.

#Now check pod description and it would show nginx with new image version.

**kubectl rollout undo deployment/nginx-deployment ->** reverts back to old deployment.

**kubectl rollout history deployment/nginx-deployment** -> shows previous deployments and their revision number.

**kubectl rollout undo deployment/nginx-deployment --to-revision=2 ->** rolls back to specific deployment.

Imp: If you delete a ReplicaSet that’s part of the deployment, k8 will automatically recreate it as it continuously monitors for desired state.

If you delete an old ReplicaSet (not the currently active), it won’t be recreated but you will lose the rollback ability for that revision.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

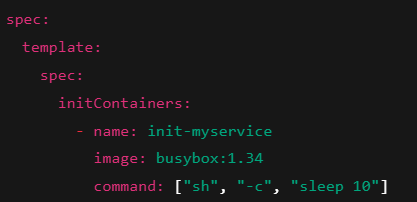
**Some Advanced directives used in deploy.yml files:**



The **maxSurge** field controls the maximum number of pods that can be created over the desired number of replicas during an update. It's specified as an integer or a percentage. Allows Kubernetes to temporarily scale up the number of pods while performing the update. In this case, Kubernetes can add up to 25% more pods than the desired number of replicas during the update. If you have 4 replicas, it will allow 1 additional pod (since 25% of 4 is 1) to be created during the update.

The **maxUnavailable** field controls the maximum number of pods that can be unavailable (not running) during the update. It is specified as an integer or a percentage. It helps control how many pods can be down during the update process.

If you don't specify rollingUpdate, the default value for maxSurge is 25% and for maxUnavailable is 25%.



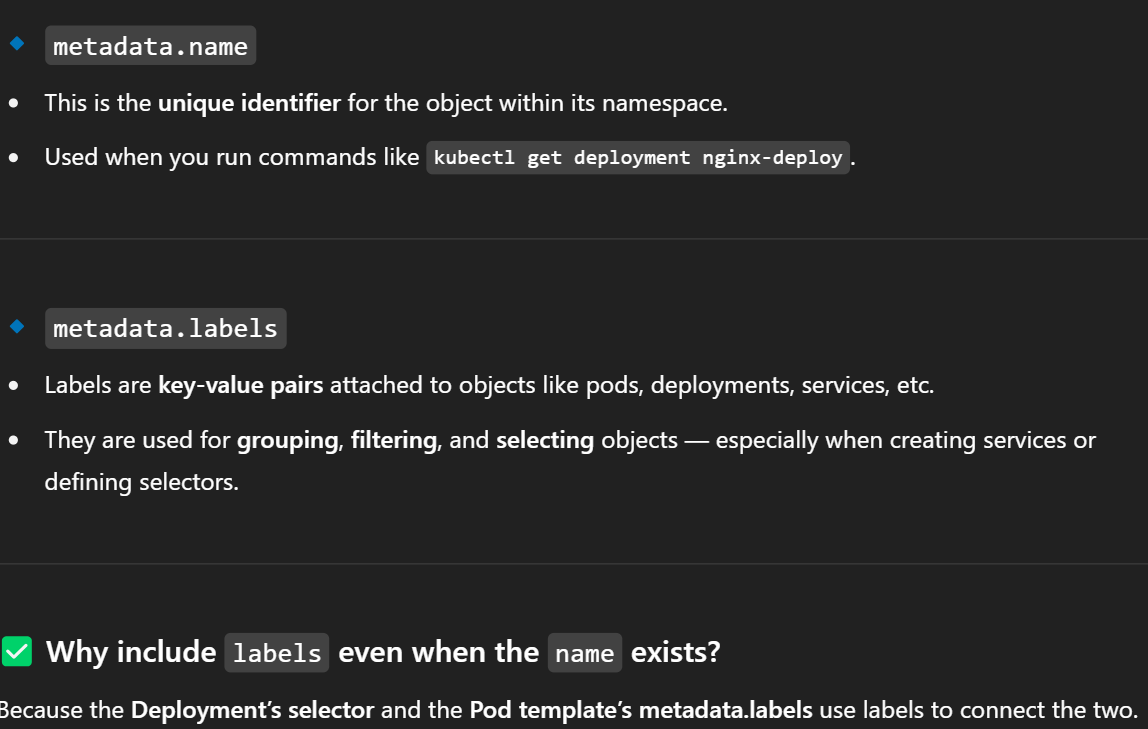


requests: The minimum resources the container needs.

limits: The maximum resources the container can use.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Refer volume.txt file in notes for k8 basic volume mapping.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In **deployment.yaml file**, we see three places where nginx is written. So what does it mean?

**metadata.name = nginx** -> denotes the name of deployment resource itself.

**spec.selector.matchLabels.app = nginx** -> what pods the deployment looks for.

**template.metadata.labels.app = nginx ->** what labels the deployment gives to the Pods it creates.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Explanation of **emptydir-volume.yml**:

### **What is emptyDir?**

* It is a **temporary** storage volume.
* It gets **created empty** when the Pod starts.
* It **exists as long as the Pod is running**.
* When the Pod is **deleted** (stopped, killed, rescheduled), **the emptyDir data is erased** permanently.

### **How it works:**

* Kubernetes allocates a folder **on the node's local storage** for that Pod.
* The container(s) inside the Pod can **read and write files** to it.

emptyDir: {} #uses disk storage

emptyDir: {}

Medium: Memory #Ram storage backed (very fast)

### **Common use cases:**

* Temporary file storage
* Sharing files between containers inside the same Pod
* Caching intermediate results

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Persistent Volumes (PV)**

A **Persistent Volume (PV)** is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using **StorageClasses.** It is a resource in the cluster just like a node is a cluster resource.

**PV Lifecycle**

1. **Provisioning**: A PV is created and its storage is allocated using the selected driver.
2. **Binding**: Kubernetes automatically watches for new PVCs and binds them to the PVs they reference.
3. **Reclaiming**: Users can delete the PVC to relinquish access to the PV. The cluster uses a reclaim policy to determine what to do with the volume after its claim is released.

**Reclaim Policies**

The reclaim policy on a PV dictates its fate after a PVC is released. There are two policies:

* **Delete**: The PV and its underlying storage are deleted when the PVC is deleted.
* **Retain**: When a PVC is deleted, the PV is not. Instead, it transitions to a Released state, but the data remains intact until manually deleted or repurposed.

**Persistent Volume Claims (PVC)**

A **Persistent Volume Claim (PVC)** is a request for storage by a user. It is similar to a Pod as Pods consume node resources and PVCs consume PV resources. PVCs allow users to request specific levels of resources (e.g., size and access modes) without needing to know the details of the underlying storage infrastructure.

**PVC Lifecycle**

1. **Provisioning**: Created by the user.
2. **Binding**: Binds to a suitable PV.
3. **Using**: Used by a Pod.
4. **Releasing**: Unbound from the Pod.
5. **Reclaiming**: PV is either deleted, or retained based on reclaim policy.

**Access Modes**

PVs provide different access modes according to the way they can be mounted:

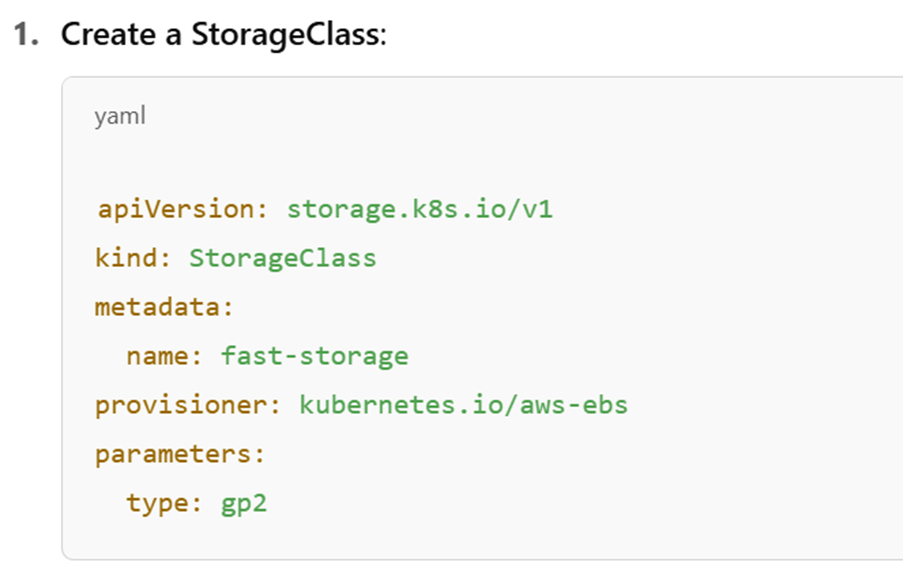
* **ReadWriteOnce (RWO)**: The volume can be mounted as read/write by only a single node.
* **ReadOnlyMany (ROX)**: The volume can be mounted as read-only by multiple nodes simultaneously.
* **ReadWriteMany (RWX)**: The volume can be mounted as read/write by many nodes.
* **ReadWriteOncePod (RWOP)**: The volume can be mounted as read/write by a single Pod.

Note: A PVC can only request a single access mode.

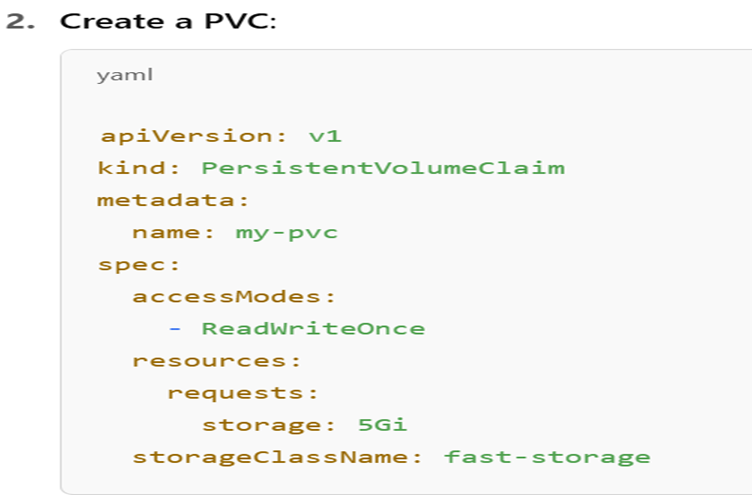
**Dynamic Provisioning with StorageClasses:**

To enable dynamic storage provisioning based on storage class, the cluster administrator needs to enable the DefaultStorageClass admission controller on the API server.

Dynamic provisioning allows storage volumes to be created automatically when a PVC is created. This eliminates the need for administrators to pre-provision storage resources.

**Example:**

This kubernetes.io/aws-ebs provisioner is **deprecated** in favor of the **CSI (Container Storage Interface)** driver: ebs.csi.aws.com. Will describe the alternative of this in ‘imp’ section below.



**3 Create a pod using PVC:**



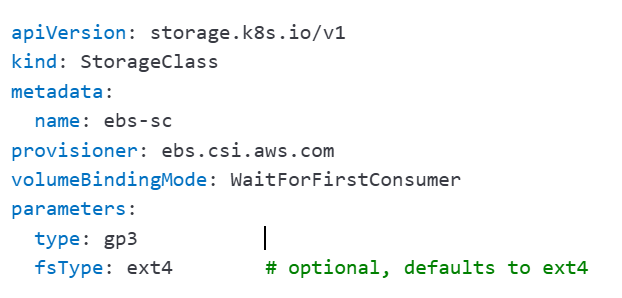
**kubectl get pvc my-pvc**

**kubectl describe pod mypod**

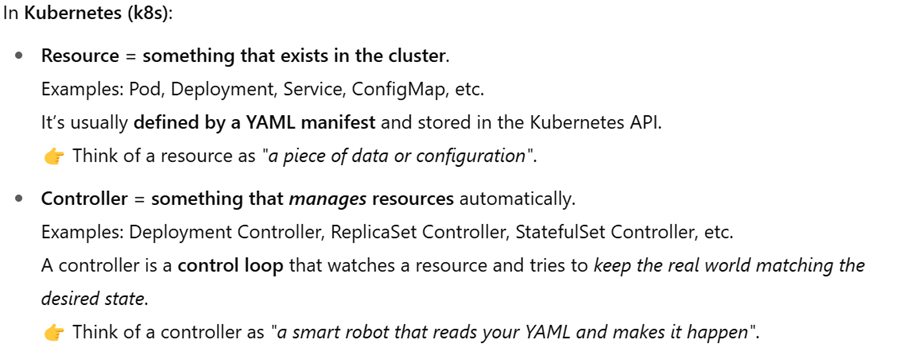
Imp: The CSI driver is maintained by AWS and has more features, better performance, and is future-proof.

**volumeBindingMode: WaitForFirstConsumer**

* This delays volume creation until a Pod using the PVC is scheduled.
* Ensures the EBS volume is created in the correct AZ (Availability Zone) matching the Pod.

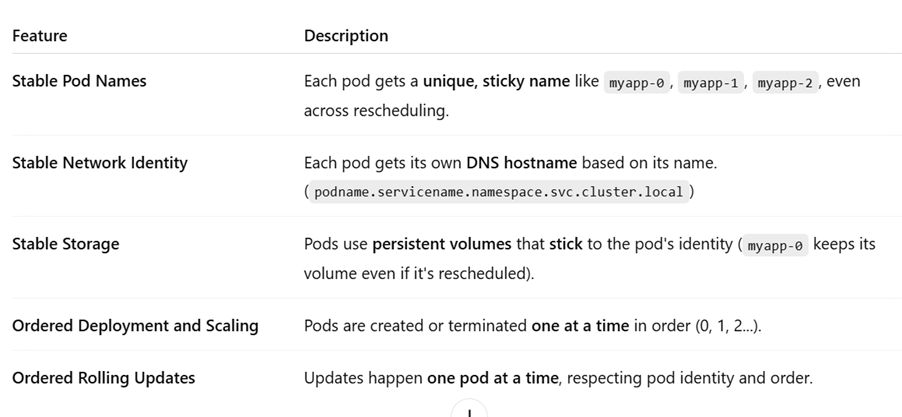


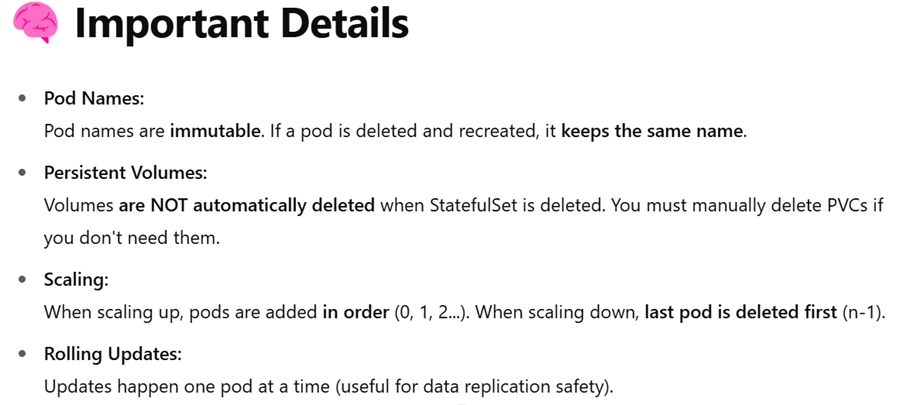
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



A **StatefulSet** is a **Kubernetes controller** used to manage **stateful applications**.  
In contrast to **Deployments**, which are meant for **stateless** apps, StatefulSets are for workloads where each pod:

* **Needs a stable identity** (name, network ID, etc.)
* **Persists data** across pod restarts
* **Has ordered operations** (create, scale, delete in a specific order)





**When to Use StatefulSets?**

✅ Databases (MySQL, PostgreSQL, Cassandra)  
 ✅ Distributed systems (Kafka, Zookeeper, etc.)  
 ✅ Any app that **requires fixed identities** and/or **persistent storage**  
 ✅ Stateful microservices (that cache or store something per instance)

Refer yaml files in notes directory.

**kubectl get statefulSet** -> lists all statefulSets in current namespace.

**kubectl describe statefulSet <name>** -> describes the statefulSet.

**kubectl get endpoints <statefulSet\_name>** ->lists ip of all the pods made by the set.

**In Kubernetes DNS names:**

<service-name>.<namespace>.svc.cluster.local

**Imp: 1-**Sometimes, updating the my-statefulSet.yml causes issues as Kubernetes forbids updates to certain fields of a StatefulSet once it’s been created — these fields include volumeClaimTemplates. Once you create a StatefulSet with a specific volumeClaimTemplates, you cannot modify that field (or change the configuration of the PVCs) without deleting and recreating the StatefulSet. So, delete existing one and create a new StatefulSet and re-apply.

2- Whenever you create a pod and want to enter into it and execute some commands, you wont be able to execute commands like nslookup, wget due to less size pods. So, the first way is to exec into the pod and apt install bind-tools, but this is not preferred as then pod size will increase. So what we do is we create a test pod and save files in it and then apply kubectl cp command as per required.

**kubectl run test-pod --image=busybox --restart=Never -- /bin/sh -c "sleep infinity"**

**kubectl exec –it test-pod – sh**

**kubectl cp <namespace>/<pod-name>:<location-in-pod> <location-in-local-machine>**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**What is a DaemonSet?**

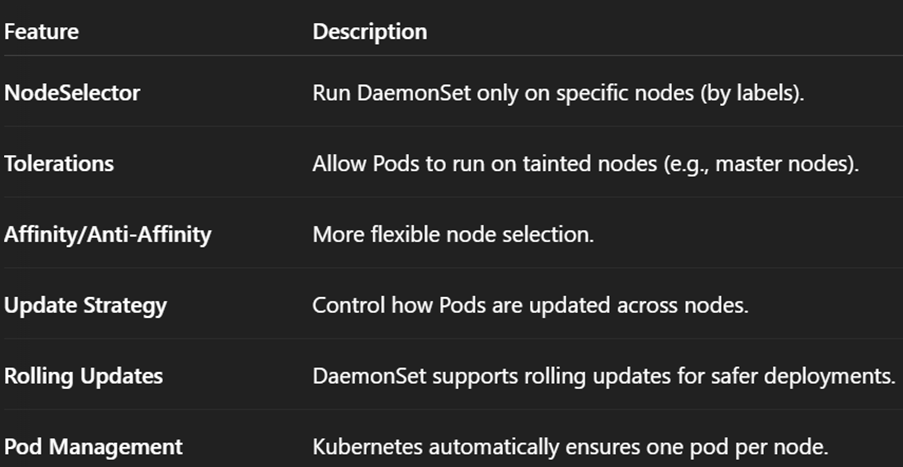
* A DaemonSet ensures that a copy of a Pod runs on all (or some) nodes in a Kubernetes cluster.
* Purpose: To automate the running of a background task on every node.

**Typical examples:**

* Running a **log collection agent** (e.g., Fluentd, Filebeat).
* Running a **metrics collector** (e.g., Prometheus Node Exporter).

**Basic Behavior of DaemonSets:**

* **When a Node is added**: DaemonSet automatically schedules a pod onto it.
* **When a Node is deleted**: DaemonSet automatically deletes the corresponding Pod.
* **When a DaemonSet is deleted**: It automatically deletes all Pods it created.
* **When Pods are manually deleted**: The DaemonSet controller recreates them.



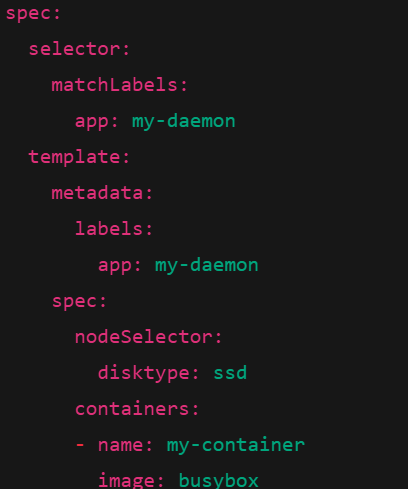
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

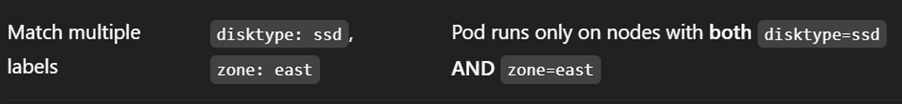
**1. NodeSelector**

You can make a DaemonSet run only on certain nodes.

If nodeSelector is mentioned, pod can run on any node.

In below yml part, **nodeSelector** ensures Pods run **only** on nodes with disktype=ssd.





\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**2. Taints and Tolerations:**

Taint: You mark a node as undesirable for normal pods.

Toleration: You configure a pod to accept (tolerate) that taint and still run on the tainted node.

DaemonSets **try to run on all nodes**, but if a node is **tainted**, **normal pods won't land there**. So, you must add **tolerations** to the DaemonSet pods **so they can land** on tainted nodes.

**kubectl taint nodes <node-name> <key>=<value>:<effect>** #you taint a node

key: the taint name value: optional value

effect: what happens

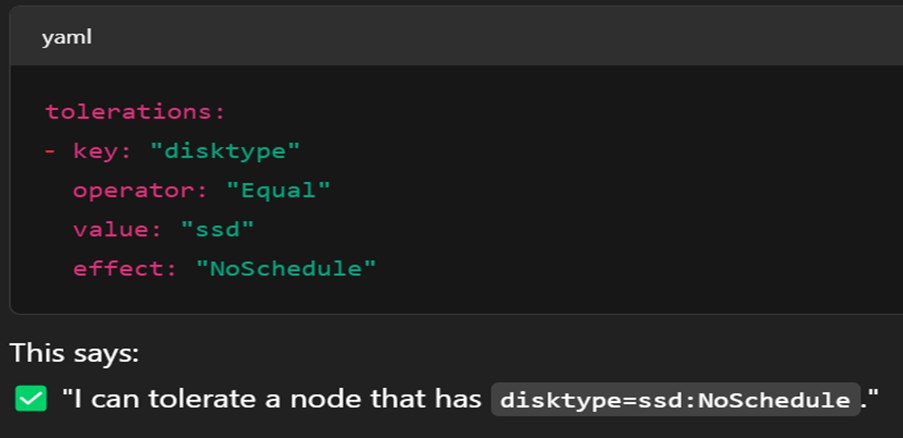
* **NoSchedule**: Do not schedule pods onto this node unless they tolerate the taint.
* **PreferNoSchedule**: Try to avoid scheduling pods onto this node, but it's not guaranteed.
* **NoExecute**: Existing pods are evicted unless they tolerate.

**NoSchedule** = "Don't allow new pods here." (Old pods stay.)

**NoExecute** = "Don't allow new pods here, and also remove old pods if they don't match!"

**Eg: kubectl taint nodes worker1 disktype=ssd:NoSchedule**

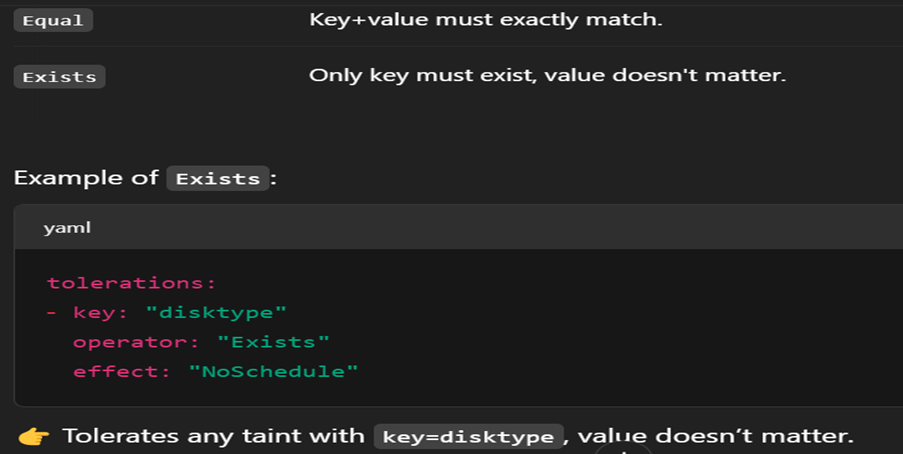
You add tolerations inside your DaemonSet or Pod spec:



**To untaint a node:**

**kubectl taint nodes <node-name> <key>=<value>:<effect>-**

Simply we put a minus sign at the end of the command we used to apply taint on .



Imp: To get the taint key of a node, simply **kubectl describe <node**> .

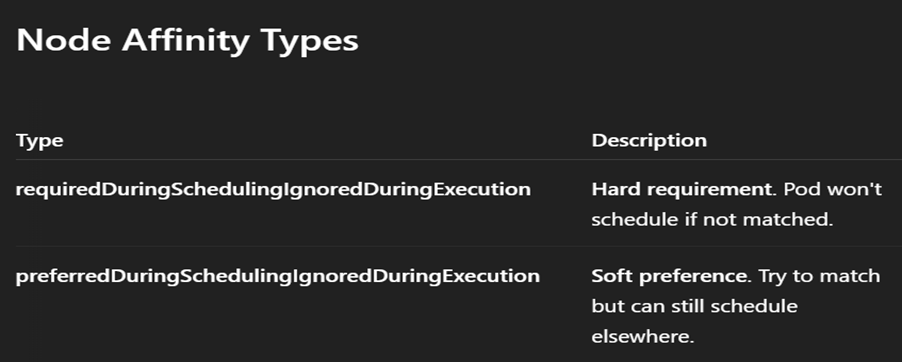
**3.Affinity/Anti-affinity:**

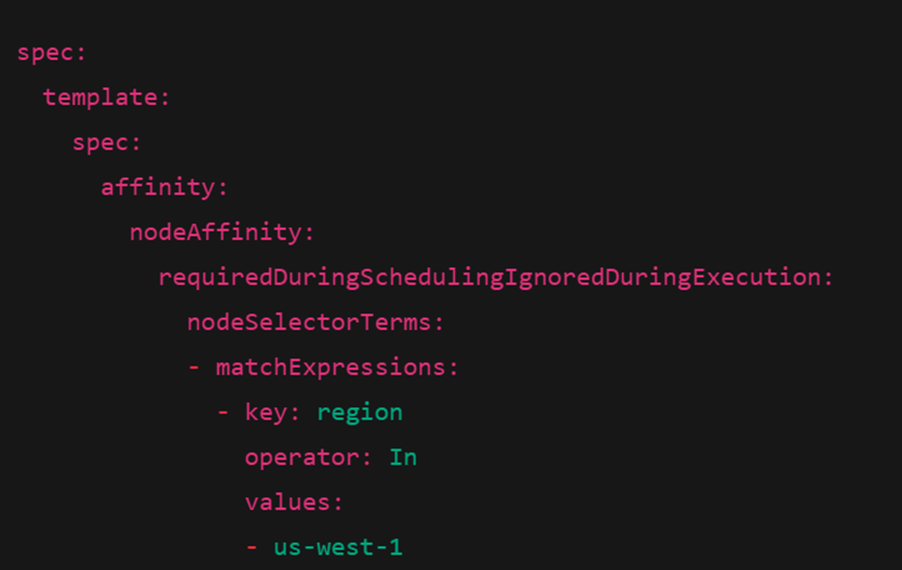
**Affinity** in Kubernetes = **smart scheduling rules**.  
 It **controls** *where* Pods are scheduled based on *node* or *pod* properties.

There are **two types**:

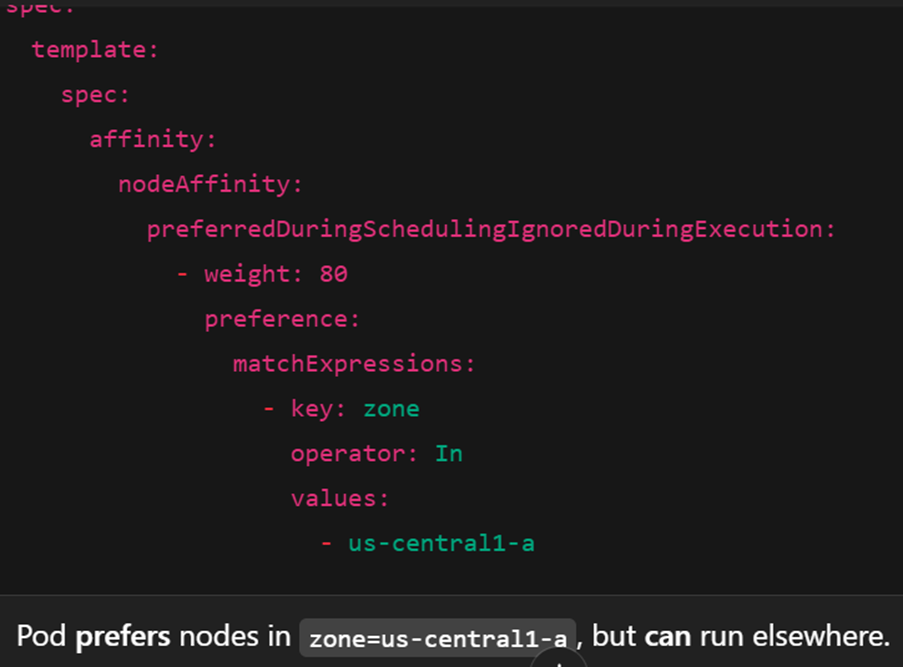
1. **Node Affinity** (match Pods to Nodes)
2. **Pod Affinity / Anti-Affinity** (match Pods to other Pods)

DaemonSets **most commonly use Node Affinity**, but they can use both.



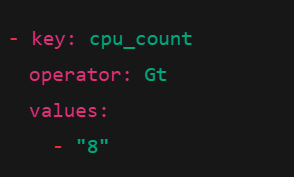


* **Pod must** be scheduled on a node labelled with region=us-west-1.
* Otherwise, Pod will stay **Pending** and won't run.



**Operators You Can Use**

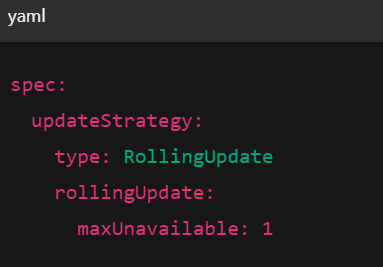
* **In** — Node must have the label key with one of listed values.
* **NotIn** — Node must NOT have listed values.
* **Exists** — Node must have the label key (value doesn't matter).
* **DoesNotExist** — Node must NOT have the label key.
* **Gt/Lt** — Greater than / Less than comparisons on **integer** label values.



Pod affinity / Pod anti-affinity is used very less, so read online for basics.

**4. Update Strategy**

DaemonSets can now support **rolling updates** (safe, controlled pod restarts).



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**ConfigMaps:**

A **ConfigMap** is a Kubernetes API object used to inject configuration data into your applications**.** It decouples configuration from application code, enabling better manageability and flexibility.  
**Multi-line strings**: Use | syntax in YAML.

Refer yml files in notes directory.

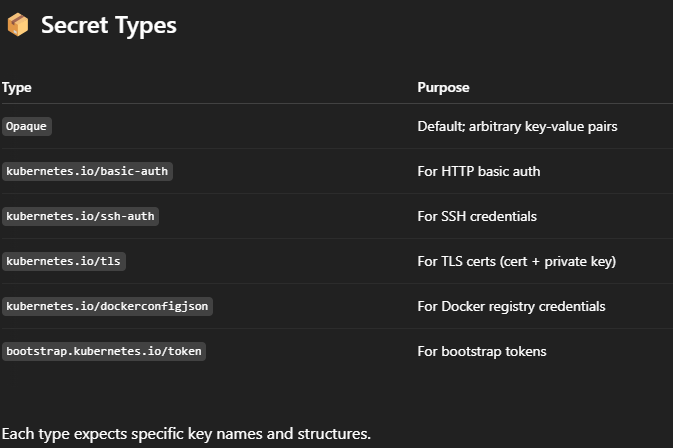
**kubectl get cm** -> get available configMaps.

Use **immutable: true** for performance (if configs don’t change) as they are needed to prevent accidental changes and also Kubelet stops asking API service for updates of such configmaps, which reduces load of API service.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Secrets:**

A **Secret** is a Kubernetes object used to store **sensitive data**, such as:

* Passwords
* API keys
* TLS certificates
* SSH keys
* OAuth tokens

  
A secret can be consumed by a pod via environment variables or volume mounts, but the best way is to be do via the k8 API server for more security,

### **Storage:**

* Secrets are stored in **etcd**, which must be **encrypted at rest**.
* Use encryption configuration with EncryptionConfiguration in API server.

### **Transmission:**

* Transmitted over the network using **TLS** between control plane and kubelets.

### **Access control:**

* Controlled via **RBAC**.
* Use fine-grained policies to restrict who/what can get, list, or watch Secrets.

Refer secrets directory for yml file (below lies explanation of those).

**kubectl apply -f secret.yaml**

**kops edit cluster --name=k8.sunnysb.xyz --state=s3://myk8bucket675**

Right under spec, add:

**encryptionConfig: true**

**#**This above configuration change enabled etcd (the key-value store used by Kubernetes) to encrypt Secrets and other sensitive resources at rest.

**kops update cluster --name=k8.sunnysb.xyz --state=s3://myk8bucket675 --yes**

**kops rolling-update cluster --name=k8.sunnysb.xyz --state=s3://myk8bucket675 --yes**

#Both kops update and rolling-update is done as both API server and current running nodes need to be restarted.

**kubectl apply -f secret-rbac.yaml**

**kubectl apply -f pod-secret-consumer.yaml**

**kubectl exec -it secret-consumer -- sh**

**cat /etc/secrets/username**

**cat /etc/secrets/password**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

SSL (Secure Sockets Layer) and TLS (Transport Layer Security) are cryptographic protocols designed to provide secure communication over a computer network, especially the internet. TLS is the modern, more secure successor to SSL. They provide encryption, authentication and data integrity.

HTTPS = HTTP over TLS.  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Ingress:**

**Ingress** is an API object in Kubernetes that manages external access to services within a cluster, typically via HTTP or HTTPS. It allows you to expose multiple services under the same IP address or load balancer, using routing rules based on hostnames or paths.

## **🔧 Why Use Ingress?**

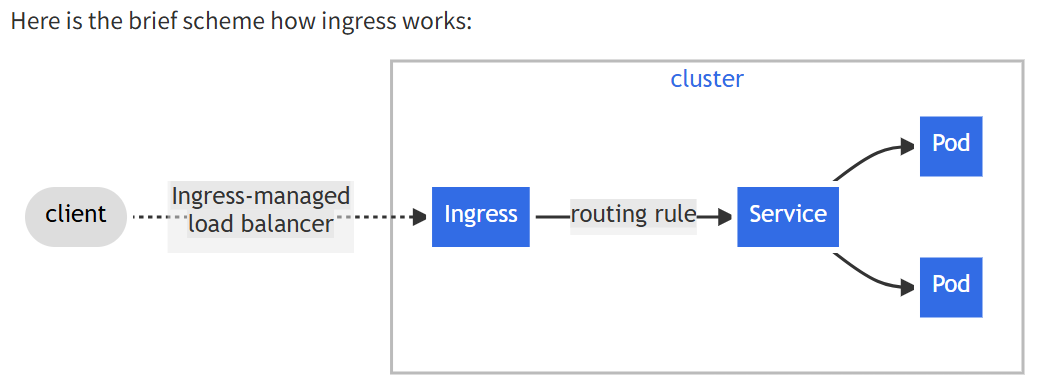
Before Ingress:

* You had to expose services with **NodePort** or **LoadBalancer** service types.
* LoadBalancer assigns a new external IP for each service (costly in cloud environments).

Ingress solves:

* Centralized routing of traffic to internal services.
* SSL/TLS termination at a central point.
* Path- or host-based routing.

You must install an Ingress Controller separately — it's not built into Kubernetes by default. You can check various types of ingress controllers from documentation. Mostly, nginx controller or AWS ingress controller is used.

Ingress is like a set of rules and services in k8 follow these rules.

#Download AWS ingress controller.

**kubectl apply -f** [**https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.1.3/deploy/static/provider/aws/deploy.yaml**](https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.1.3/deploy/static/provider/aws/deploy.yaml)

**kubectl apply –f vprodeployment.yml**

**kubectl apply –f vproservice.yml**

**kubectl apply –f vproingress.yml**

**kubectl get deploy**

**kubectl get ingress**  #copy the address of ELB made by this

Go to goDaddy.com, add a CNAME record with name as vprofile and value as obtained in the address section.

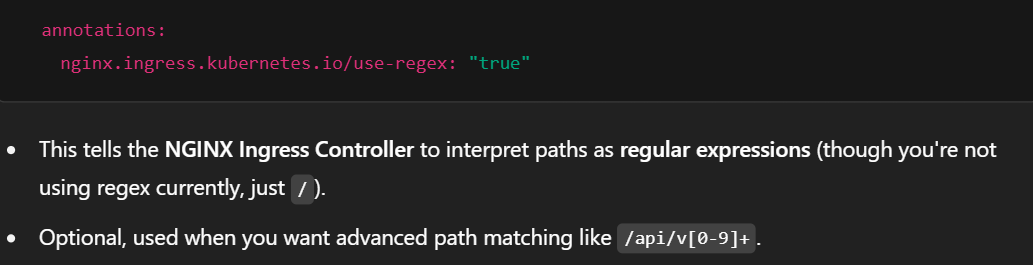
Open <http://vprofile.sunnysb.xyz/>

#Delete the ingress controller downloaded

kubectl delete -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.1.3/deploy/static/provider/aws/deploy.yaml

**1.Annotations:**

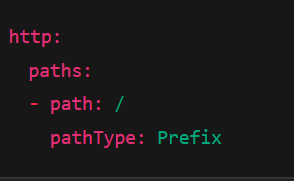
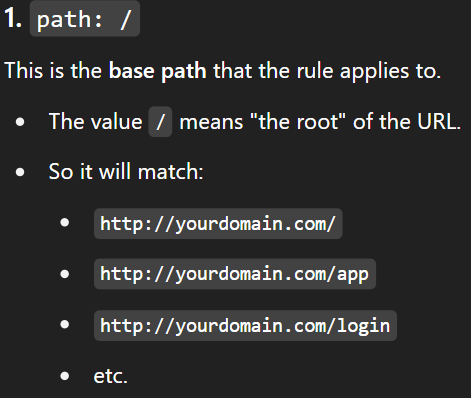
Annotations are key-value pairs in the metadata.annotations field of an Ingress resource.

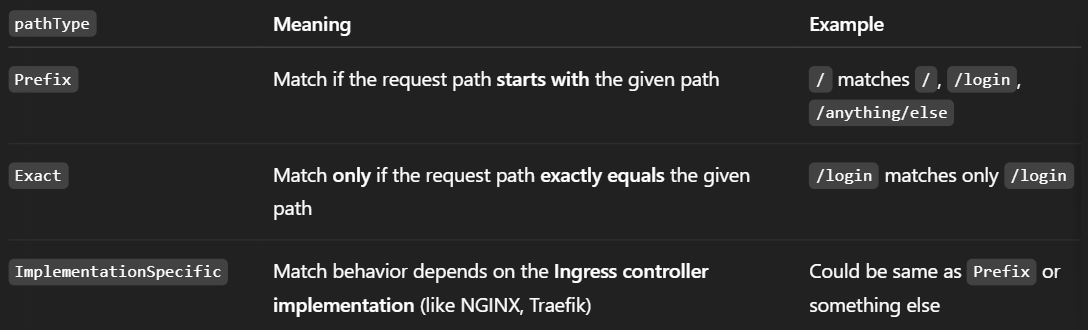
Kubernetes Ingress often matches on **partial URL paths**, like /app1/\*. But many backend services don’t understand URLs with extra path prefixes (e.g., /app1/login instead of /login).

So, we **rewrite** the path to what the backend expects by using annotation:  
**nginx.ingress.kubernetes.io/rewrite-target; /**

**nginx.ingress.kubernetes.io/auth-url**  #These two enable external   
**nginx.ingress.kubernetes.io/auth-signin** #authentication (eg. Oauth2)

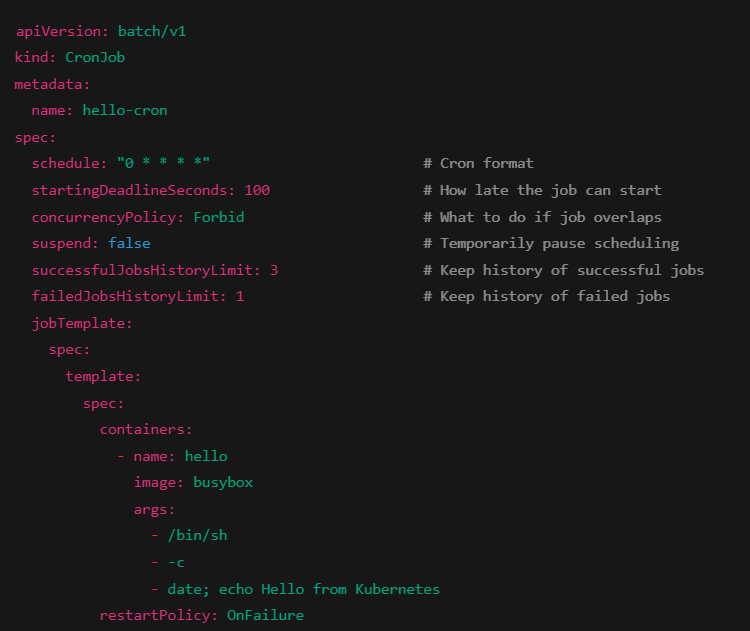
**2. Paths:**

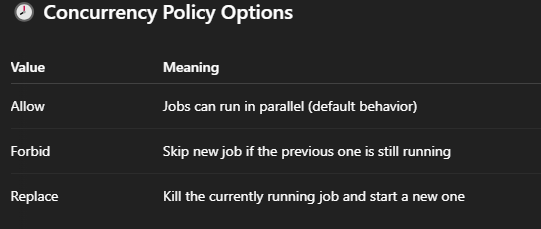
 

  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

A **CronJob** in Kubernetes is used to **run Jobs on a scheduled time**, just like the **cron** utility in Linux.

* It creates a **Job** resource on a defined **cron schedule**.
* The Job **creates Pods**, and the **Pods run to completion** (one-off tasks).



  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**RBACs:**

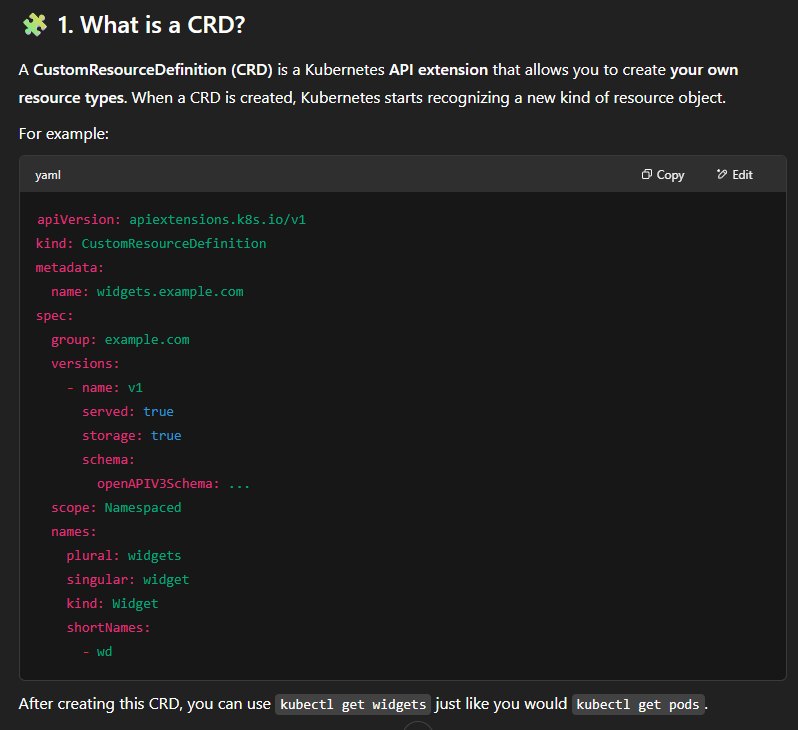
[**https://chatgpt.com/share/6818579a-cac4-8003-a078-1d014db6bd83**](https://chatgpt.com/share/6818579a-cac4-8003-a078-1d014db6bd83)

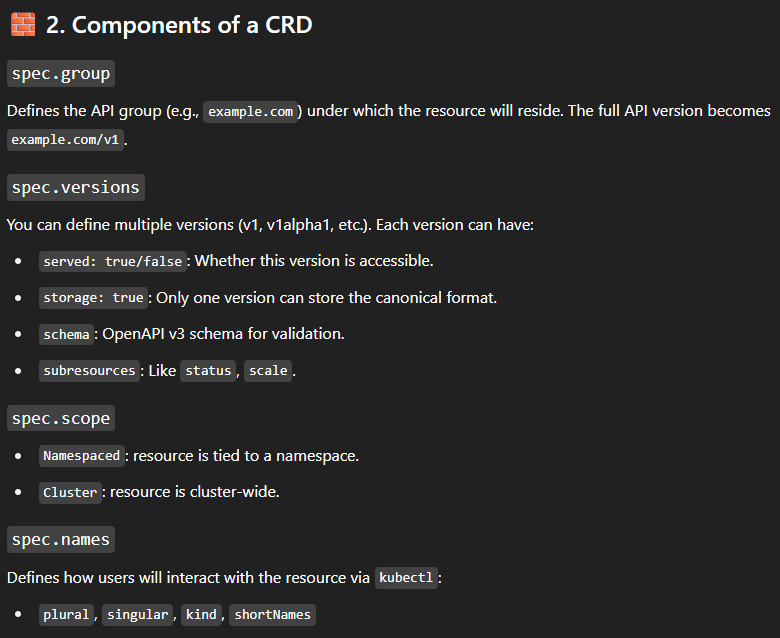
**kubectl auth reconcile -f my-rbac-rules.yaml --remove-extra-subjects --remove-extra-permissions**

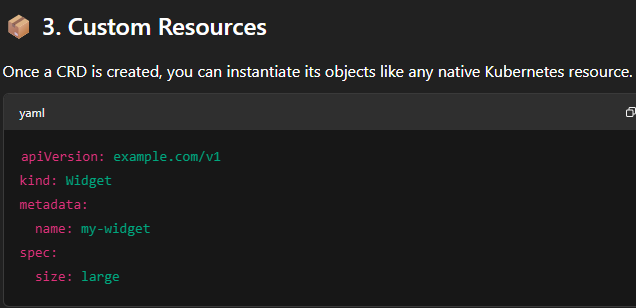
* **->** is used to synchronize (reconcile) your RBAC resources (such as Role, RoleBinding, ClusterRole, and ClusterRoleBinding) defined in a YAML file with the live cluster state.  
  This command does **not delete** resources — it only updates existing ones to match your file.
* It’s **idempotent**, making it safe for repeated use in automation and GitOps workflows.
* It’s often used to **enforce** RBAC rules in production and ensure consistency.

**kubectl auth can-i get configmap/my-configmap** -> checks whether your current user (or identity) is authorized to perform the action get on the specific resource configmap/my-configmap in the current namespace.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_





\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_