**KUBERNETES**

what i need to focus while learning kubernetes as devops engineer ??

As a DevOps engineer, you should focus on the following areas while learning Kubernetes:

1. Kubernetes architecture: Gain a clear understanding of the Kubernetes architecture, including its core components, such as the Kubernetes API server, etcd, kube-scheduler, kube-controller-manager, and kubelet.
2. Deployment and scaling: Learn how to deploy and manage applications on Kubernetes, including how to scale applications horizontally or vertically as per demand.
3. Networking: Understand how networking works in Kubernetes, including how to configure Kubernetes services, ingress, and network policies.
4. Storage: Learn how to provision and manage storage for applications on Kubernetes, including how to use persistent volumes and persistent volume claims.
5. Security: Understand the security features of Kubernetes, including how to use Kubernetes RBAC (Role-Based Access Control) to control access to Kubernetes resources.
6. Monitoring and logging: Learn how to monitor Kubernetes clusters and applications using tools like Prometheus and Grafana, and how to aggregate logs from multiple containers running on Kubernetes using tools like Fluentd and Elasticsearch.
7. Continuous Integration and Deployment: Understand how to integrate Kubernetes with your CI/CD pipeline, including how to use tools like Jenkins, GitLab, and CircleCI to automate the deployment of applications on Kubernetes.

By focusing on these areas, you can become a skilled Kubernetes practitioner and effectively manage Kubernetes-based applications as a DevOps engineer.

What is Kubernetes and its components ?

Kubernetes is an open source container orchestration tool which is used to manage, scale and maintains the containers.

It has client server architecture which is distributed model. It has single master/multiple master for high availability and scalability but by default it has only single master and worker nodes which technically called as minions.

Master and worker node has different Kubernetes components to work properly i.e master node has administrative components where as worker node has follower components

Master node components : etcd storage, controller manager, api server, scheduler

Etcd storage: it’s a open source, scalable and available key value storage. It has cluster structure so that many storage devices will be there to provide. It has the data related to pods, configurations, api objects. Whenever there is a new pod or change in configuration of existing pod, then etcd will notify that to api server by using watchers.

Api server : api server is the heart of the Kubernetes master node, its is the face of master when any one worker node is able to see that there is a master . whenever any task to related to remaining three components(etcd storage, controller manager & kube scheduler) then api server transfers to that relevant component . kubeconfig is the package that comes with Kubernetes which is responsible for all the communication

Control manager: it is the main component which takes the request from the api server, whenever a worker node sends a request to create a new pod or to make changes in existing pod then control manager takes up that task and executes it.

Kube scheduler: it manages the work load of pods as per the resource utilization

1. what are the different yaml files used in kubernetes and their purpose in the entire process

* In Kubernetes, YAML files are used to define the specifications of Kubernetes resources such as pods, services, deployments, and more. Here are some of the different types of YAML files used in Kubernetes and their purposes:
* Pod YAML file - Defines the specifications for a pod, which is the smallest unit of deployment in Kubernetes. It includes the container images, ports, volumes, environment variables, and more.
* Deployment YAML file - Defines the specifications for a deployment, which manages the lifecycle of pods. It includes the number of replicas, the pod template to use, and more.
* Service YAML file - Defines the specifications for a service, which provides a stable IP address and DNS name for a set of pods. It includes the type of service, the selector to use, and more.
* ConfigMap YAML file - Defines the specifications for a ConfigMap, which provides a way to decouple configuration data from container images. It includes the key-value pairs of the configuration data.
* Secret YAML file - Defines the specifications for a Secret, which provides a way to store sensitive information such as passwords, tokens, and keys. It includes the key-value pairs of the sensitive information.
* PersistentVolumeClaim YAML file - Defines the specifications for a PersistentVolumeClaim, which is a request for storage resources. It includes the storage class, access mode, and more.
* PersistentVolume YAML file - Defines the specifications for a PersistentVolume, which is a piece of networked storage in the cluster. It includes the storage capacity, access modes, and more.
* Ingress YAML file - Defines the specifications for an Ingress, which provides a way to expose HTTP and HTTPS routes from outside the cluster to services within the cluster. It includes the rules for routing traffic to services.

These YAML files are used in the Kubernetes deployment process to define the desired state of the resources and to manage their lifecycle. They are typically created by developers or DevOps engineers and are stored in source control systems such as Git. They are then applied to the Kubernetes cluster using the kubectl command line tool or through CI/CD pipelines

**explain the storage concepts with commands in Kubernetes ?**

Persistent Volumes (PV): PV is a storage resource in Kubernetes cluster. A PV is a piece of storage in the cluster that has been provisioned by an administrator or dynamically created using Storage Classes.

To create a Persistent Volume, you can use the following command:

kubectl apply -f <pv-definition>.yaml

This will create a Persistent Volume based on the YAML file specified.

1. Persistent Volume Claims (PVC): PVC is a request for storage by a user or an application in Kubernetes cluster. PVCs are used to request a specific amount of storage and a specific type of access.

To create a Persistent Volume Claim, you can use the following command:

kubectl apply -f <pvc-definition>.yaml

This will create a Persistent Volume Claim based on the YAML file specified.

1. Storage Classes: A StorageClass is a way to define different classes of storage in a Kubernetes cluster. StorageClasses allow you to define different types of storage with different performance characteristics, and dynamically allocate storage to a PVC.

To create a Storage Class, you can use the following command:

kubectl apply -f <storage-class-definition>.yaml

This will create a Storage Class based on the YAML file specified.

1. StatefulSets: StatefulSets are used to manage stateful applications that require persistent storage. StatefulSets guarantee that the order of deployment, scaling, and deletion of pods is predictable, which is essential for applications that require a unique network identity or stable storage.

To create a StatefulSet, you can use the following command:

kubectl apply -f <statefulset-definition>.yaml

This will create a StatefulSet based on the YAML file specified.

1. Volume Mounts: A VolumeMount is a way to mount a volume into a container in a pod in Kubernetes cluster. VolumeMounts allow you to share data between containers in a pod, and also allow you to store data persistently.

To create a Volume Mount, you can use the following command:

**kubectl apply -f <volume-mount-definition>.yaml**

This will create a Volume Mount based on the YAML file specified.

These are just a few examples of the many storage commands used in Kubernetes. Understanding these commands is essential for managing storage in a Kubernetes environment.

**what will be the step by step process if any application is deployed in kubernetes cluster by argo ci cd from bit bucket or any source code management tool ?**

Here are the high-level steps for deploying an application in Kubernetes cluster using Argo CI/CD from a source code management tool such as Bitbucket:

1. Set up the Kubernetes cluster: The first step is to set up a Kubernetes cluster if you haven't already. You can use any cloud provider or on-premises solution for this, depending on your requirements.
2. Install Argo CD: Argo CD is a tool for deploying applications to Kubernetes. Install Argo CD on your Kubernetes cluster and configure it to manage your applications.
3. Configure your source code management tool: Connect your source code management tool, such as Bitbucket, to Argo CD by configuring webhooks or triggers. This will ensure that Argo CD is notified of any changes to your application code.
4. Create a deployment pipeline: Create a deployment pipeline in Argo CD that defines the steps needed to deploy your application. This pipeline can include tasks such as building the application, creating Docker images, and deploying to Kubernetes.
5. Define your application deployment manifest: Define your application deployment manifest in a Kubernetes YAML file. This file specifies the desired state of your application, including its replicas, resources, and services.
6. Store the deployment manifest in source control: Store the deployment manifest in your source control system, such as Bitbucket. This will ensure that Argo CD can access the manifest and deploy the application to Kubernetes.
7. Trigger the deployment pipeline: Make changes to your application code and commit them to your source control system. This will trigger the deployment pipeline in Argo CD, which will build and deploy your application to Kubernetes.
8. Monitor the deployment: Monitor the deployment of your application using Argo CD and Kubernetes dashboard. This will help you identify any issues or errors that may occur during the deployment process.
9. Rollback if necessary: If the deployment fails or there are issues with the application, roll back to the previous version using Argo CD.

These are the high-level steps involved in deploying an application in Kubernetes cluster using Argo CI/CD from a source code management tool. The exact process may vary depending on your specific requirements and tools used.

**explain the networking concept in kubernetes with commands used in real time ?**

Sure, here are some common commands used in Kubernetes networking:

1. Get pod IP address: To get the IP address of a pod in Kubernetes, you can use the following command:

kubectl get pods -o wide

This will return a list of pods running in the default namespace, along with their IP addresses.

1. Create a service: To create a service in Kubernetes, you can use the following command:

kubectl create service <service-type> <service-name> --tcp=<port>:<target-port> --dry-run=client -o yaml > <service-name>.yaml

This will create a YAML file with the service definition, which you can then apply to your cluster using the **kubectl apply -f <service-name>.yaml** command.

1. Expose a deployment: To expose a deployment in Kubernetes, you can use the following command:

kubectl expose deployment <deployment-name> --type=<service-type> --port=<port> --target-port=<target-port>

This will create a service that exposes the deployment on the specified port and target port.

1. Create a network policy: To create a network policy in Kubernetes, you can use the following command:

kubectl create -f <network-policy>.yaml

This will create a network policy based on the YAML file specified.

1. Get service IP address: To get the IP address of a service in Kubernetes, you can use the following command:

kubectl get services

This will return a list of services running in the default namespace, along with their IP addresses.

These are just a few examples of the many commands used in Kubernetes networking. Understanding these commands is essential for managing networking in a Kubernetes environment.

explain the stroage concept in Kubernetes ?

In Kubernetes, storage is a critical concept that enables persistent data storage for containerized applications. Kubernetes provides several storage options to meet the needs of different applications and workloads. Here are some of the key storage concepts in Kubernetes:

1. Volumes: Kubernetes volumes provide a way to store and share data between containers in a pod. Volumes are mounted into containers as a directory and can be backed by various storage types, such as hostPath, emptyDir, and persistentVolumeClaim.
2. Persistent Volumes: Persistent volumes (PVs) are network-attached storage resources that can be dynamically provisioned or pre-provisioned by a cluster administrator. PVs can be used to store data persistently and independently of the lifecycle of a pod.
3. Persistent Volume Claims: A Persistent Volume Claim (PVC) is a request for a specific amount and type of storage from a PV. PVCs enable developers to request storage resources without knowing the underlying storage details, such as the storage location or type.
4. Storage Classes: Storage Classes provide a way to define different storage profiles in a Kubernetes cluster. Storage classes enable cluster administrators to specify the type of storage backend to use for dynamic provisioning of Persistent Volumes.
5. StatefulSets: StatefulSets are a Kubernetes resource that provides a way to manage stateful applications that require stable network identities and persistent storage. StatefulSets enable developers to create and manage stateful applications, such as databases, with a unique identity and persistent storage.
6. CSI Drivers: Container Storage Interface (CSI) drivers provide a standardized interface for storage vendors to integrate with Kubernetes. CSI drivers enable the use of different storage backends with Kubernetes and allow for dynamic provisioning and management of storage resources.

Understanding Kubernetes storage is essential for managing stateful applications in a Kubernetes environment. Kubernetes provides a rich set of storage options to meet the needs of different applications and workloads, enabling developers to build and deploy applications with persistent data storage.

**Kubernetes components** :

ETCD : etcd is key value storage meaning it will store the information in individual files rather grouping the information with any relation like relational data base system. So change in any one individual data will not disturb remaining ones.

It listens to the port 2379

Etcdctl is the command line tool to interact wit etcd cluster. Eg : etcdctl set key1 value 1

The above command used to set one key value in etcd cluster

Etcdctl get key 1 command will gives you value of key 1 that you have saved previously

./etcdctl –version to get the version of etcdctl

Etcd stores the info like nodes,pods, configs, secretes, accounts, roles

Whatever I heard as of now about certs , these will be saved in etcd for that certificate needs to be generated and saved it in specific path

Api server : api server is the main component of the Kubernetes master node, it is the face to all the worker node. It listens to the worker nodes and provides the info as demanded

It first check with etcd cluster about the info asked by the Kubernetes worker node and provides

For eg: kubectl get pods > api server forward this request to the etcd because pod configuration information is in etcd and etcd provides it to the api server which will return to user

Control manager : It mainly has two controllers one is node controller and pod controller

Node controller actions >

it checks nodes and their working pods continuously meaning status of nodes by using api server. It monitors all the working nodes for every 5 secs and wait for the grace period to complete which is 40 sec then if any node is not responding then a new node will be created accordingly the configuration available in etcd cluster about that node

In the above state of node is called unreachable and pod eviction timeout is 5 mins (meaning to delete the unreachable node and to create a new node and pod )

Pod controller > it always maintains the pods as per the replica sets

There are other controllers as well like deployment, stateful set, job controller, cron job, name space, end point, service account, pv protection , pv binder, replication controllers

Considerations for large cluster

the latest version of Kubernetes (1.27 ) supports the following

the no of pods per node should not be more than 110

no of nodes per cluster must be less than 5000

no of pods per cluster 150000

no of containers per cluster 300000

kubernetes nodes do not automatically steer traffic towards control plane end points that are in the same failure zone. explain this in simple terms ?

Kubernetes is a system used for deploying and managing applications across multiple computer nodes. The control plane is a set of components that manage the Kubernetes cluster, and it's important that these components remain available and responsive.

In Kubernetes, nodes are the worker machines that run application workloads, and they communicate with the control plane to receive instructions and updates. However, if a control plane endpoint in a particular failure zone becomes unavailable, the nodes in that zone won't automatically steer traffic to another available endpoint in the same zone.

This means that if a control plane endpoint fails, the nodes in that zone will continue to send traffic to the failed endpoint, potentially causing downtime or errors. To prevent this, it's important to configure Kubernetes to automatically detect and handle these kinds of failures by implementing redundancy and failover mechanisms.

In simple terms, it's like having a team of workers (the nodes) who need to communicate with their boss (the control plane) to get their tasks done. If the boss suddenly becomes unavailable, the workers might not know what to do and could stop working altogether. To avoid this, the workers need to have backup instructions and someone else to take over if the boss is unavailable.

**what is failure zone with refers to kubernetes cluster** ?

In Kubernetes, a failure zone refers to a group of nodes that are susceptible to a single point of failure. For example, if all the nodes in a cluster are located in the same physical data center, that data center represents a failure zone.

If a failure occurs in that data center, such as a power outage or network interruption, it can affect all the nodes in that zone and cause downtime or service disruption. Therefore, it's important to spread out the nodes across multiple failure zones to increase availability and resiliency.

To achieve this, Kubernetes uses a feature called "node affinity" to schedule pods and services across different failure zones, ensuring that critical components of the cluster are not all located in the same zone. This can improve the cluster's overall reliability and fault tolerance, making it more resilient to failures.

From running in multiple zone concept, what I understand is

1. Kubernetes control plane components(api server, etcd cluster, control manager, kube scheduler) are all resources interchangeable, and they wont configure at one failure zone instead they will replicate into multiple zones to provide high availability
2. Kubelet in each node will assign label to each and every container so that kube control manager will know about the pods and nodes

Pods :

explain the pods that run multiple containers that need to work together in simple terms ?

1. In Kubernetes, a pod is the smallest deployable unit that can be managed. It represents a single instance of a running process in a cluster. A pod can contain one or more containers that are tightly coupled and need to work together to provide a complete application service.
2. When you have multiple containers that need to work together, you can run them in a single pod. Running multiple containers in a pod provides several benefits, including:
3. Shared network namespace: All containers in a pod share the same network namespace, which means they can communicate with each other using localhost or 127.0.0.1. This makes it easy for containers to interact with each other and share resources.
4. Shared storage: Containers in a pod can share the same volume, which makes it easy for them to read and write data to a shared location.
5. Atomic deployment: All containers in a pod are deployed and scaled together, which ensures that the application is always running in a consistent state.
6. Simplified configuration: Running multiple containers in a pod simplifies the configuration process because you only need to define the pod configuration once.
7. For example, let's say you have a web application that consists of a front-end container and a back-end container. The front-end container serves web pages to users, while the back-end container processes requests and communicates with a database. You can run both containers in a single pod, which allows them to share resources and communicate with each other easily.
8. In summary, pods that run multiple containers that need to work together provide a convenient way to deploy and manage complex applications in Kubernetes. By running multiple containers in a single pod, you can simplify configuration, share resources, and ensure that your application is always running in a consistent state.

what are init containers and explain them with practical examples and explain their importance in simple way ?

1.In Kubernetes, an init container is a special type of container that runs before the main container in a pod. The main purpose of an init container is to perform any initialization tasks that the main container needs before it can start running.

2.Init containers are useful when your application requires some additional setup before it can start running. For example, you might need to download some configuration files or install dependencies before the main container can start. By using an init container, you can perform these tasks before the main container starts running, which ensures that your application starts in a consistent and reliable state.

3.Here's an example of how you can use an init container to download a configuration file before the main container starts:

apiVersion: v1

kind: Pod

metadata:

name: my-pod

spec:

containers:

- name: main-container

image: my-image

command: ["./my-app"]

initContainers:

- name: init-container

image: busybox

command: ['sh', '-c', 'wget -O /config/my-config https://example.com/my-config']

volumeMounts:

- name: config-volume

mountPath: /config

volumes:

- name: config-volume

emptyDir: {}

In this example, we have two containers in a pod: init-container and main-container. The init-container is using the busybox image and running a wget command to download a configuration file from example.com and store it in a volume called config-volume. The main-container uses the my-image image and runs a command ./my-app. The main-container is dependent on the config/my-config file which is being downloaded by the init-container.

Here are some reasons why init containers are important in Kubernetes:

1. Init containers allow you to perform pre-start initialization tasks that your application needs, ensuring that it starts in a consistent state.
2. Init containers can be used to fetch configuration data or secrets from external services before the main container starts.
3. Init containers can help to minimize downtime during application updates, by allowing you to perform any required migrations or updates before the main container starts.
4. In summary, init containers are a useful feature in Kubernetes that allows you to perform initialization tasks before the main container in a pod starts running. By using init containers, you can ensure that your application starts in a consistent and reliable state, and minimize downtime during updates

* Deleting container or multiple containers doesn’t mean that deleting pod. Pod is just an environment in which container will run
* This means object inside that i.e processes get deleted but not the environment
* Pod will be deleted until explicitly deleted
* Pods are only supported by two types of os only i.e windows and linux as of now v1.27
* In spec, we should define .spec.os.name= windows/linux
* Os of the pod and os that the node is running should be same otherwise kubelet doesn’t allow a pod to be run

**what are workload resources and explain them in easiest wa**y ?

In Kubernetes, workload resources are used to deploy and manage applications in a cluster. There are three main types of workload resources:

1. Deployment: A deployment manages a set of replica pods, ensuring that a specified number of replicas are running at any given time. Deployments are typically used for stateless applications, such as web servers or microservices.
2. StatefulSet: A StatefulSet manages a set of replica pods, but provides guarantees about the ordering and uniqueness of each pod. StatefulSets are typically used for stateful applications, such as databases or storage systems.
3. DaemonSet: A DaemonSet ensures that a single instance of a pod is running on each node in the cluster. DaemonSets are typically used for system-level tasks, such as logging or monitoring.

Workload resources define the desired state of an application in Kubernetes, and the Kubernetes control plane ensures that the actual state of the application matches the desired state. This means that if a pod fails or a node goes down, the control plane will automatically spin up new pods or move them to healthy nodes, ensuring that the application is always running.

In summary, workload resources are an essential part of deploying and managing applications in Kubernetes. Deployments, StatefulSets, and DaemonSets provide different ways of managing replicas of pods in a cluster, ensuring that applications are always running and providing guarantees about the ordering and uniqueness of each pod.

can you brief the statement " guarantees about the ordering and uniqueness of each pod " in statefulset above ?

Sure! When we talk about "guarantees about the ordering and uniqueness of each pod" in a StatefulSet, it means that StatefulSets provide guarantees on how pods are created, scaled, and destroyed. Specifically, each pod in a StatefulSet has a unique and predictable name that is based on a specified naming convention. For example, if you have a StatefulSet with a name of "web", each pod in the StatefulSet might have a name like "web-0", "web-1", "web-2", and so on.

The unique and predictable naming of pods in a StatefulSet is important because it allows applications that require stable network identities, such as databases or distributed systems, to work correctly. For example, if you have a database running in a StatefulSet with multiple replicas, each replica can use its unique name to discover other replicas and join the cluster.

StatefulSets also provide guarantees about the ordering of pod creation, scaling, and deletion. Specifically, StatefulSets ensure that pods are created and scaled up or down in a deterministic order, based on their unique names. This is important for applications that require data consistency or have complex interdependencies between pods.

Overall, the guarantees provided by StatefulSets about the ordering and uniqueness of each pod make them a valuable tool for managing stateful applications in Kubernetes.

Top of Form

PROBES :

A probe is a diagnostic acitivity that kubelet does. Usually api server collects the information from all nodes via kubelet about the health condition of node and pods. For this activity , kubelet needs to check the pod those are

1. HTTP GET ACTION
2. EXECACTION
3. TCPSOCKETACTION

**POD LIFE CYCLE** :

Pod has different working modes, starting from pending, running, success and failed. If you start a container it will goes to pending state and then goes to running state. If it is successfully running then we treat it as success or failure.

For some errors, restarting a container provides solution or clears the error . those are given below

Restarting a container in a pod can help to resolve a variety of errors. Here are some common types of errors that might require a container to be restarted:

1. Configuration errors: If the container's configuration is incorrect or the application is misconfigured, restarting the container can help to fix the problem.
2. Resource errors: If the container is consuming too many resources such as CPU or memory, restarting the container can help to free up those resources.
3. Network errors: If the container is having trouble communicating with other containers or with the network, restarting the container can help to re-establish those connections.
4. Application errors: If the application running inside the container is crashing or behaving erratically, restarting the container can help to clear any state that might be causing the problem.
5. It's important to note that not all errors can be resolved by simply restarting a container. In some cases, more extensive troubleshooting and debugging may be required to identify and fix the root cause of the problem.

How the pods life is configured ?

Like individual application containers, Pods are considered to be relatively ephemeral (rather than durable) entities . explain this in simple terms and in more practical way .

In simple terms, Pods are considered to be relatively ephemeral because they are designed to be short-lived and disposable. Just like how flowers bloom and wither away, Pods are created, do their job, and are then deleted or replaced.

In a more practical way, this means that Pods are typically created to run a specific task or application, and once that task or application is completed, the Pod is no longer needed. For example, if you have an application that needs to process some data, you might create a Pod to run the application, and once the data processing is finished, the Pod is deleted.

This ephemeral nature of Pods makes them well-suited for use in containerized environments like Kubernetes, where workloads are dynamically managed and scaled based on demand. Instead of having to manually provision and manage individual servers to run applications, you can use Kubernetes to create and manage Pods, which are more lightweight and easier to scale up or down as needed.

However, it's important to note that while Pods are designed to be disposable, the data and state of your applications still need to be persisted in some way. This can be done using persistent volumes, which allow data to be stored independently of the Pods that are running the application. So even if a Pod is deleted or replaced, the data it was working on can still be accessed by other Pods that need it.

POD PHASES :

There are several phases of pod. Phase is nothing but it tells in which condition the pod is right now meaning from pending to failure

Pending > this means, pod is accepted by the cluster but still containers need to be created so time will take. This condition is called pending

Running > atleast one of the containers is running or restarted

Success > all the containers are running successfully and not restarted

Failure > any of the containers is failure to operate.

Unknown > it occurs when node is irresponsive

Every time a new pod created , there will be unique id (UID) will be assigned to that particular pod. Pod itself can not heal but if any pod is not working because of lack of resources meaning failure of node. Then new health node will be assigned in cluster and a similar new pod will be assigned to that node with different UID.

**Yaml file for Kubernetes :**

After configuration of cluster completes, we need to configure worker nodes using any infra set up tool like terraform or cloudformation in aws. Once these worker nodes are configured then we need to deploy our pods into worker nodes. For this we need to write yaml files

Basically there are four components in pod definition yaml file

1. Api version 2. Kind 3. Metadata 4. Spec

Api version > to know which version of api we need to use,

Use command “kubectl api-versions”

$ kubectl api-versions

admissionregistration.k8s.io/v1beta1

apiextensions.k8s.io/v1beta1

apiregistration.k8s.io/v1

apps/v1

authentication.k8s.io/v1

authentication.k8s.io/v1beta1

authorization.k8s.io/v1

authorization.k8s.io/v1beta1

autoscaling/v1

batch/v1

batch/v1beta1

certificates.k8s.io/v1

certificates.k8s.io/v1beta1

coordination.k8s.io/v1

coordination.k8s.io/v1beta1

...

This is the output we get if use that command

* **admissionregistration.k8s.io/v1beta1**: This API version is used for defining validating webhooks and mutating webhooks that intercept requests to the Kubernetes API server and modify them before they are persisted to etcd. You would use this API version if you need to define custom webhooks for your Kubernetes cluster.
* **apiextensions.k8s.io/v1beta1**: This API version is used for creating and managing custom resource definitions (CRDs) in Kubernetes. You would use this API version if you need to define your own custom resources for use in your Kubernetes cluster.
* **apiregistration.k8s.io/v1**: This API version is used for registering custom APIs with the Kubernetes API server. You would use this API version if you need to register a custom API with the Kubernetes API server.
* **apps/v1**: This API version is used for managing applications and workloads in Kubernetes, including deployments, statefulsets, and daemonsets. You would use this API version if you need to create or manage Kubernetes deployments or other workload objects.
* **authentication.k8s.io/v1**: This API version is used for authenticating users and services in Kubernetes, including configuring authentication plugins and defining authentication policies. You would use this API version if you need to manage authentication in your Kubernetes cluster.
* **authentication.k8s.io/v1beta1**: This is an older beta version of the **authentication.k8s.io/v1** API version. You should prefer using the **v1** version whenever possible.
* **authorization.k8s.io/v1**: This API version is used for authorizing access to Kubernetes resources, including configuring authorization plugins and defining authorization policies. You would use this API version if you need to manage authorization in your Kubernetes cluster.
* **authorization.k8s.io/v1beta1**: This is an older beta version of the **authorization.k8s.io/v1** API version. You should prefer using the **v1** version whenever possible.
* **autoscaling/v1**: This API version is used for autoscaling Kubernetes resources, including horizontal pod autoscalers (HPAs) and cluster autoscalers. You would use this API version if you need to configure autoscaling for your Kubernetes cluster.
* **batch/v1**: This API version is used for running batch jobs in Kubernetes, including jobs and cronjobs. You would use this API version if you need to create or manage batch jobs in your Kubernetes cluster.
* **batch/v1beta1**: This is an older beta version of the **batch/v1** API version. You should prefer using the **v1** version whenever possible.
* **certificates.k8s.io/v1**: This API version is used for managing certificates in Kubernetes, including generating certificates for use in TLS communication and configuring certificate authorities (CAs). You would use this API version if you need to manage certificates in your Kubernetes cluster.
* **certificates.k8s.io/v1beta1**: This is an older beta version of the **certificates.k8s.io/v1** API version. You should prefer using the **v1** version whenever possible.
* **coordination.k8s.io/v1**: This API version is used for managing coordination between different Kubernetes components, including leases and locks. You would use this API version if you need to manage coordination in your Kubernetes cluster.
* **coordination.k8s.io/v1beta1**: This is an older beta version of the **coordination.k8s.io/v1**
* **discovery.k8s.io/v1beta1**: This API version is used for discovering Kubernetes resources, including service endpoints and API servers. You would use this API version if you need to manage discovery in your Kubernetes cluster.
* **events.k8s.io/v1**: This API version is used for managing events in Kubernetes, including creating and querying events for Kubernetes resources. You would use this API version if you need to manage events in your Kubernetes cluster.
* **events.k8s.io/v1beta1**: This is an older beta version of the **events.k8s.io/v1** API version. You should prefer using the **v1** version whenever possible.
* **extensions/v1beta1**: This is an older API version used for managing Kubernetes extensions, including third-party extensions. This API version is deprecated and you should use the **apps/v1** API version instead.
* **networking.k8s.io/v1**: This API version is used for managing networking in Kubernetes, including configuring network policies, service load balancers, and ingresses. You would use this API version if you need to manage networking in your Kubernetes cluster.
* **networking.k8s.io/v1beta1**: This is an older beta version of the **networking.k8s.io/v1** API version. You should prefer using the **v1** version whenever possible.
* **node.k8s.io/v1**: This API version is used for managing Kubernetes nodes, including node status and configuration. You would use this API version if you need to manage nodes in your Kubernetes cluster.
* **node.k8s.io/v1beta1**: This is an older beta version of the **node.k8s.io/v1** API version. You should prefer using the **v1** version whenever possible.
* **policy/v1**: This API version is used for managing policies in Kubernetes, including pod security policies and pod disruption budgets. You would use this API version if you need to manage policies in your Kubernetes cluster.
* **policy/v1beta1**: This is an older beta version of the **policy/v1** API version. You should prefer using the **v1** version whenever possible.
* **rbac.authorization.k8s.io/v1**: This API version is used for managing role-based access control (RBAC) in Kubernetes, including creating and managing roles and role bindings. You would use this API version if you need to manage RBAC in your Kubernetes cluster.
* **rbac.authorization.k8s.io/v1beta1**: This is an older beta version of the **rbac.authorization.k8s.io/v1** API version. You should prefer using the **v1** version whenever possible.
* **scheduling.k8s.io/v1**: This API version is used for managing scheduling policies in Kubernetes, including configuring priority classes and preemption. You would use this API version if you need to manage scheduling in your Kubernetes cluster.
* **scheduling.k8s.io/v1beta1**: This is an older beta version of the **scheduling.k8s.io/v1** API version. You should prefer using the **v1** version whenever possible.
* **storage.k8s.io/v1**: This API version is used for managing storage in Kubernetes, including configuring persistent volumes (PVs) and persistent volume claims (PVCs). You would use this API version if you need to manage storage in your Kubernetes cluster.
* **storage.k8s.io/v1beta1**: This is an older beta version of the **storage.k8s.io/v1** API version. You should prefer using the **v1** version whenever possible.

Kind > in this field, we will mention whether it is deployment, statefulset, networking, what action or what is the outcome of this yaml file will mention here

Meta data: in meta data there are two fields, one is name which has string value and label which has child fields like name & type

Eg api version: v1

Kind: pod

Metadata:

Name: my-app

Labels:

App: myapp

Type: front-end

In the above eg, name & label are entered after some spaces when compare to metadata & remaining meaning name & lable are child fields of metadata and remaining all are sibilings of metadata. By providing the label type , you can group your pods under one category when you have multiple pods in your cluster and working for different faces like front end and backend

Services :