** Kubernetes (k8s)**

Doc contains -

- Layman def for K8s & its keywords
- Technical def's
- Commands
- K8s Yaml Templates (k8s Manifest)

Overall something which can make applications run on K8s.

Kubernetes is the orchestration for your containerised application (Microservices), it can be difficult & error prone to manage huge number of containers manually which includes network, storage, scaling, portability, deployments, dependencies & more - So here Kubernetes help us with that & many more similar parts.

Layman explanation of Kubernetes components, the way we got Babar-Humayun-Akbar-Jhangir -

- Here in K8s,
  Master Node is the Father of Worker Node,
  Worker Node is the Father of Pods
  Pods is the father of container running within that, &
  Container is something where our application is running.

- **Ingress** is the gate to indoor & outdoor for application access within Kuberenetes cluster.

- With some Yaml files we can play around all kubernetes application related stuff -
  Service, deployments, replicas, ingress,
  configMap for variables, secret, pod level permissions etc.,
  known as **Kubernetes Manifest files (**Templates can be checked below.**)**

- For inter microservice communication on kubernetes we can use **Service Mesh** (Ex - Istio, Linkerd)

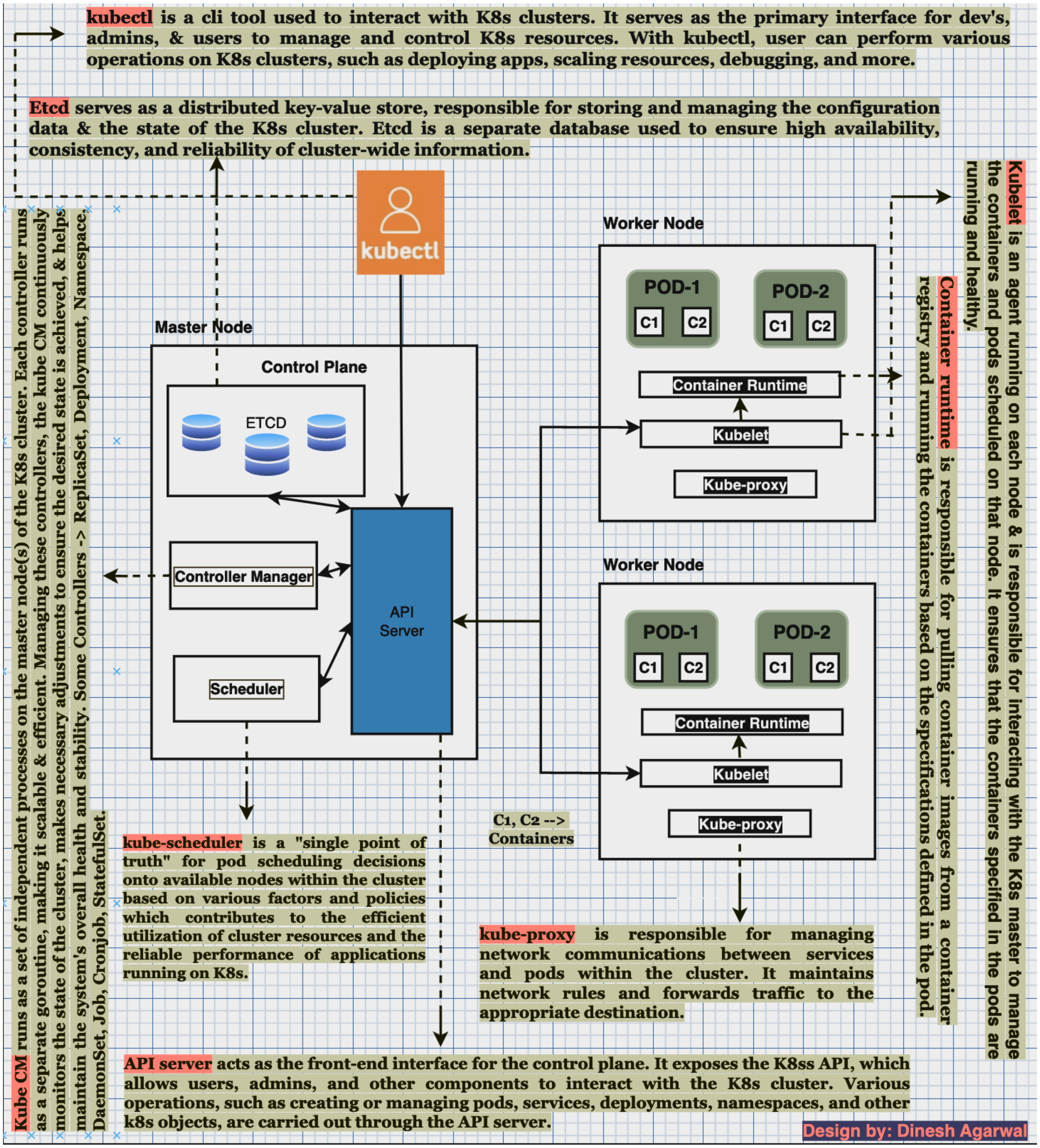
- **ConfigMap** to decouple environment specific configuration from our container images.

- Using **Service Account** we can fine-tune pods access to various resources and services, ensuring that each Pod operates with the least privilege principle.

- **Kustomization** can be used for env(dev, test, stage, prod) based configuration to make its management easy & reliable.

- **GitOps Tool** can be used for its continuous Delivery to Kubernetes. (Ex - ArgoCD, Flux)

Kubernetes Architecture



kubectl is a cli tool used to interact with K8s clusters. It serves as the primary interface for dev's, admins, & users to manage and control K8s resources. With kubectl, user can perform various operations on K8s clusters, such as deploying apps, scaling resources, debugging, and more.

Etcd serves as a distributed key-value store, responsible for storing and managing the configuration data & the state of the K8s cluster. Etcd is a separate database used to ensure high availability, consistency, and reliability of cluster-wide information.

Kube CM runs as a set of independent processes on the master node(s) of the K8s cluster. Each controller runs as a separate goroutine, making it scalable & efficient. Managing these controllers, the kube CM continuously monitors the state of the cluster, makes necessary adjustments to ensure the desired state is achieved, & helps maintain the system's overall health and stability. Some Controllers -> ReplicaSet, Deployment, Namespace, DaemonSet, Job, Cronjob, StatefulSet.

Kubelet is an agent running on each node & is responsible for interacting with the K8s master to manage the containers and pods scheduled on that node. It ensures that the containers specified in the pods are running and healthy.

Container runtime is responsible for pulling container images from a container registry and running the containers based on the specifications defined in the pod.

kube-scheduler is a "single point of truth" for pod scheduling decisions onto available nodes within the cluster based on various factors and policies which contributes to the efficient utilization of cluster resources and the reliable performance of applications running on K8s.

kube-proxy is responsible for managing network communications between services and pods within the cluster. It maintains network rules and forwards traffic to the appropriate destination.

API server acts as the front-end interface for the control plane. It exposes the K8ss API, which allows users, admins, and other components to interact with the K8s cluster. Various operations, such as creating or managing pods, services, deployments, namespaces, and other k8s objects, are carried out through the API server.

Kubectl

kubectl is a command-line tool used to interact with Kubernetes clusters. It serves as the primary interface for developers, administrators, and users to manage and control Kubernetes resources. With kubectl, we can perform various operations on Kubernetes clusters, such as deploying applications, scaling resources, debugging, and more.
Kubectl uses the Kubernetes API to communicate with the cluster, so it requires network connectivity to the cluster's API server.

Most Common request represented in Kubectl

- GET:	get a resource.
- DELETE:	delete a specified resource object.
- DESCRIBE:	retrieve information about a specified resource object.
- CREATE:	creating k8s resources.
- APPLY:	manages applications through files defining K8s resources.
- EDIT:	open a resource for editing.
- REPLACE:	replace a resource by filename or stdin
- EXEC:	execute a command in a container
- LOGS:	print the logs for a container in a pod or specific resource.

Components of a Kubernetes node are:

Master Node :

kube-api server - likely refers to the Kubernetes API server, which is a crucial component of a Kubernetes cluster and is typically deployed as part of the Kubernetes control plane.
The Kubernetes API server acts as the front-end interface for the Kubernetes control plane. It exposes the Kubernetes API, which allows users, administrators, and other components to interact with the Kubernetes cluster.
Various operations, such as creating or managing pods, services, deployments, namespaces, and other Kubernetes objects, are carried out through the API server.

kube-scheduler - It's a core component of Kubernetes responsible for scheduling pods onto available nodes within the cluster. As part of the Kubernetes control plane, it plays a critical role in optimizing resource utilization and ensuring that pods are placed on appropriate nodes based on various factors and policies.

kube-scheduler is a "single point of truth" for pod scheduling decisions. Once it schedules a pod to a specific node, it updates the Kubernetes API server with this information, which ensures that all components of the cluster have consistent knowledge about the pod's location.

By effectively assigning pods to appropriate nodes, the kube-scheduler contributes to the efficient utilization of cluster resources and the reliable performance of applications running on Kubernetes.

Etcd - Etcd is a critical component of the control plane that serves as a distributed key-value store. It is responsible for storing and managing the configuration data and the state of the Kubernetes cluster. Etcd is a separate database used to ensure high availability, consistency, and reliability of cluster-wide information. It ensures that the entire cluster maintains a consistent view of its configuration and state, thereby enabling the reliable and scalable operation of the platform.

Kube Controller Manager is one of the core components that make up the Kubernetes control plane & is designed to be highly available and runs as a set of independent processes on the master node(s) of the Kubernetes cluster. Each controller runs as a separate goroutine, making it scalable and efficient.

By managing these controllers, the Controller Manager continuously monitors the state of the cluster, makes necessary adjustments to ensure the desired state is achieved, and helps maintain the system's overall health and stability.

Some of the essential controllers managed by the Controller Manager include:

- ReplicaSet Controller
- Deployment Controller
- StatefulSet Controller
- DaemonSet Controller
- Job Controller
- CronJob Controller
- Namespace Controller

Worker Node

Kubelet -> The Kubelet is an agent running on each node and is responsible for interacting with the Kubernetes master to manage the containers and pods scheduled on that node. It ensures that the containers specified in the pods are running and healthy.

Container Runtime -> The container runtime is responsible for pulling container images from a container registry and running the containers based on the specifications defined in the pod.

Kubernetes Proxy (kube-proxy) -> The kube-proxy is responsible for managing network communications between services and pods within the cluster. It maintains network rules and forwards traffic to the appropriate destination.

Node Status -> Nodes report their status to the master, providing information about their availability and capacity. The Kubernetes scheduler uses this information to make decisions on where to place new pods based on resource availability and constraints.

Node

Nodes are the individual machines (physical or virtual) that form the underlying compute resources of a Kubernetes cluster or can say are the worker machines in a Kubernetes cluster where containers are deployed, and form the foundation for running and managing containerized applications within the Kubernetes ecosystem.

Node Commands

- ➔ kubectl get nodes
- ➔ kubectl get nodes -o wide

Pod

Pods are the smallest deployable units of computing that we can create & manage in Kubernetes.
A Pod is a group of one or more containers, with shared storage & network resources and a specification for how to run the containers.
A Pod's contents are always co-located & co-scheduled & run in a shared context. A Pod models an application specific "logical host": it contains one(recommended) or more application containers which are relatively tightly coupled.

Pod Commands

- ➔ kubectl run <desired-pod-name> --image <Image-Name>
- ➔ kubectl get pods/po
- ➔ kubectl get pods -o wide
- ➔ kubectl get pods --namespace/-n <namespace-name>
- ➔ kubectl delete pod <pod-name>
- ➔ kubectl exec -it <pod-name> -- <command Ex- cat /usr/share/nginx/html/index.html>
  [It Connects to container in a Pod]
  Ex: kubectl exec -it <pod-name> -- /bin/bash
- ➔ kubectl expose pod <pod-name> --type=LoadBalancer --port=80 --name=<service-name>
  [Expose Pod as a Service]
- ➔ Kubectl describe pod <pod-name>
- ➔ kubectl logs <pod-name>
- ➔ kubectl logs -f <pod-name> [stream pod logs with -f & access application to see live logs]

`pod.yaml`

```yaml
apiVersion: v1
kind: Pod
metadata:                            # Dictionary Object
 name: myapp-pod                    # pod-name
 namespace: <default/YourChoice>
 labels:
 app: myapp                         # key value pairs, myapp is container label
spec:
 container:                         # List
 - name: myapp                      # container-name
 image: <image-name>
 ports:
 - containerPort: 80
```

ReplicaSet

ReplicaSet purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.

ReplicaSet commands

- ➔ kubectl get replicaset/rs
- ➔ kubectl describe replicaset/rs <replicaSet-name>

`replicaset.yaml`

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:                               # Dictionary object
 name: myapp-rs
 labels: (optional)

spec:                                   # Dictionary Object
 replicas: 3                            # 3 pods
 selector:
   matchLabels:
     app: myapp                         # to identify pods, replicaset need to give pod labels here.
 template:                              # under template it covers 'pod specification'
   metadata:
     name: myapp-pod
     namespace: default/YourChoice
     labels:
       app: myapp

   spec:
     containers:
     - name: myapp
       image: <image-name>
       ports:
       - containerPort: 80
```

```
Note: deployment is a superset of replicaset, 99.99% of time we create deployment, not rs. Here
just change kind from ReplicaSet to Deployment - it will work.
```

```
Note: Difference between "name(mandatory)" & "label(optional)" fields in k8s manifest -
            The "name" field provides a unique identifier for a specific resource, while
            "labels" offer a way to add metadata and categorize resources based on common
     attributes, allowing for better organization, filtering, and querying of resources within the
     Kubernetes cluster.
```

Deployments

A Deployment Provides declarative updates for Pods & ReplicaSets.
We describe a desired state in a Deployment & the Deployment Controller changes the actual state to the desired state at a controlled rate.
We can define Deployments to create new ReplicaSets or to remove existing Deployments & adopt all their resources with new Deployments.

`deployment.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: myapp-deployment
 labels:                                          (optional)
   app: <microservice-name>
spec:
 replicas: 1
 selector:
   matchLabels:
     app: myapp
 template:
   metadata:
     name: myapp-pod
     namespace: <default/YourChoice>
     labels:
       app: myapp
 spec:
   serviceAccountName: <Service-Account-Name>
   containers:
   - name: myapp
     image: <image-name>
     imagePullPolicy: Always
     ports:
     - containerPort: 80
       name: http
     env:
     - name: SPRING_PROFILE_ACTIVE
       value: dev
     - name: SPRING_APPLICATION_JSON
       valueFrom:
         configMapKeyRef:
           name: <configMap-metadata-name>
           key: config.json
```

Service

An abstract way to expose an application running on a set of Pods as a network service.
With Kubernetes we don't need to modify our application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods their own IP addresses & a single DNS name for a set of Pods & can load balance across them.

Service Commands

- ➔ kubectl get service/svc
- ➔ kubectl get service/svc --namespace/-n <namespace-name>
- ➔ kubectl edit service <service-name>
- ➔ kubectl describe service <service-name>
- ➔ kubectl delete svc <svc-name>

`service.yaml`

```
apiVersion: v1
kind: Service
metadata:
 name: myapp-pod-service
 labels:                                                   #(optional in service)
   app: myapp
spec:
 type: LoadBalancer         # type can be -> #LoadBalancer # ClusterIP(default), # NodePort, here
     we are using LB
 selector:
   app: myapp
 ports:
 - name: http
   protocol: TCP
   port: 80                    # deployment Port
   targetPort: 80
```

```
Note:
 - tagertPort -> container Port, or port which you expose in your Dockerfile for the service.
 (`targetPort` is the port number on the backend pods that the Service forwards traffic to, while
     'port' is the port number on which the Service itself listens for incoming traffic.)
 - selector is -> to which POD you want to send the traffic coming to this respective service or
     backend to the service. Here you need to give the pod label.
```

Ingress

An API object that manages external access to the services in a cluster, typically HTTP.
Ingress may provide load balancing, SSL termination & name based virtual hosting.
Ingress exposes HTTP & HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the ingress resources.

Ingress Commands

- ➔ kubectl get ingress --namespace/-n <namespace-name>
- ➔ kubectl edit ingress <ingress-name>
- ➔ kubectl describe ingress <ingress-name>
- ➔ kubectl delete ingress <ingress-name>

`ingress.yaml`

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: <ingress-name>
 namespace: <namespace>
 annotation:
   nginx.ingress.kubernetes.io/ssl-redirect: "true"
   nginx.ingress.kubernetes.io/use-regex: "true"
spec:
 ingressClassName: nginx
 rules:
 - host: <hostname/elb>
   http:
     paths:
     - backend:
         service:
           name: <service-name>          # service -> metadata -> name_of_service
           port:
             number: 80                  # clusterIP port or service port need to be written
here, which is 'port' in service.yaml file, other port is container port (targetPort)
       path: /<your-path>(/|$)(.*)
       pathType: Prefix
```

INGRESS CLASS

In Kubernetes, an Ingress class is a way to specify which Ingress controller should be used to handle incoming traffic for a particular Ingress resource. Ingress controllers are responsible for implementing the Ingress rules and managing the incoming external traffic to services in the cluster.
`ingressClassName` helps in choosing the appropriate Ingress controller to process the Ingress rules and manage the traffic for that specific Ingress resource.
EX: ingressClassName: nginx # Specifies the Ingress class to use (e.g., "nginx")

Ingress Class Commands

➔ kubectl get ingressclass --all-namespace
➔ kubectl get ingressclass <ingressclass-name> -o yaml
➔ kubectl delete ingressclass nginx --all-namespace

`Ingressclass.yaml`

```
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
 name: nginx
spec:
 controller: nginx.org/ingress-controller
```

Secret

Kubernetes Secrets let you store & manage sensitive information such as passwords, API keys, ssh keys, OAuth tokens and other confidential data. Secrets are base64-encoded and can be used by applications deployed within a Kubernetes cluster, ensuring that sensitive information is kept secure and not exposed in plain text.
(FYI - base64 encoding is not encryption, as a result, it's crucial to manage access to Secrets carefully.)

Secrets are scoped to a specific namespace within a Kubernetes cluster. They can only be accessed by resources within the same namespace.

`secret.yaml`

```
apiVersion: v1
kind: Secret
metadata:
 name: my-secret
type: Opaque
data:
 username: dXNlcm5hbWU=   # base64-encoded value of "username"
 password: cGFzc3dvcmQ=   # base64-encoded value of "password"
```

```
Note: After applying we can reference this Secret in our pod specifications or deployments to
     securely access the sensitive data it holds.
     For example, we can use it as environment variables or mount it as volumes in our pods.
```

ConfigMap

A configMap is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as env variables, command-line arguments, or as configuration files in a volume.
A configMap allows you to decouple env specific configuration from our containers images, so that our applications are easily portable.
Note: It doesn't provide secrecy or encryption.

`configmap.yaml`

```
apiVersion: v1
kind: ConfigMap
metadata:
 name: <configMap-metadata-name>
data:
 config.json:
   '{
     "Key1_url": "Value1",
     "Key2_path": "Value2"
   }'
```

Namespaces

Namespaces are powerful tools for managing and organizing resources in a Kubernetes cluster, particularly in complex multi-tenant environments where multiple users or teams share the same infrastructure. Properly using namespaces helps maintain better control, isolation, and resource allocation within the cluster.

Namespace commands

- ➔ kubectl get namespace/ns
- ➔ kubectl create namespace <namespace-name>
- ➔ Kubectl config current-context
- ➔ kubectl config set-context --current --namespace=new-namespace
  [To switch namespace under same context like cluster or user]
- ➔ kubectl describe namespace my-namespace
- ➔ kubectl delete namespace <namespace-name>

`namespace.yaml`

```
apiVersion: v1
kind: Namespace
metadata:
 name: <namespace>
 annotations:
   linkerd.io/inject: enabled
```

```
Note: Above Annotations indicate the Linkerd service mesh that the pods deployed within this
     namespace should be automatically injected with Linkerd's sidecar proxy containers. This
```

```
    allows Linkerd to manage the service-to-service communication, observability, and security
    aspects of your applications.

    Annotations are used to attach arbitrary metadata to Kubernetes objects, providing additional
    information for various purposes, such as managing and monitoring resources.
```

Kustomization

With kustomization.yaml, we can manage variations of our Kubernetes resources easily, ensuring that each environment or use case has the appropriate customizations while keeping the base resources separate and maintainable.
For this we can create separate overlays for other environments like production or staging by creating additional directories with their respective kustomization.yaml files, containing specific patches or changes.

`kustomization.yaml`

```yaml
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- deployment.yaml
- configmap.yaml

namespace: dev

images:
- name: <ecr-domain>/<project-name>
 newTag: "20"

patches:
- path: patch.yaml
 target:
   kind: Deployment
   name: my-app
   version: apps/v1
```

```
Explanation: Above example create a kustomization.yaml to customize the image tag for diff
    environments.
        resources  - includes List of paths to the resource YAML files you want to include in your
    customization.
        patch.yaml - modifications will be applied to the deployment.yaml resource, and the
    resulting output will include the customized changes.
```

```
Note: This approach allows you to keep your modifications separate from the base resource
    definition, making it easier to manage different configurations for different environments.
```

DaemonSet

A DaemonSet ensures that all Nodes run a copy of a Pod. As nodes are added to the cluster, pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected.
Deleting a DamonSet will clean up the Pod it created.
DaemonSets are especially useful for running cluster-level services like
- log collectors (e.g., Fluentd or Filebeat),
- monitoring agents (e.g., Prometheus Node Exporter), or
- storage daemons (e.g., Ceph RBD),

where it is necessary to have an instance of the pod on every node for monitoring, logging, or storage purposes.

Job

A Job creates one or more Pods & will continue retry execution of the Pods until a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completion. When a specified number of successful completions is reached, the task(i.e., Job) is complete. Deleting a job will clean up the Pods it created.

CronJob

A CronJob creates a Job on a reporting schedule.

Service Account

- Service Accounts are commonly used to control access and permissions for Pods within the cluster. By associating specific Service Accounts with Pods, we can fine-tune their access to various resources and services, ensuring that each Pod operates with the least privilege principle.

- A Service Account provides an identity for processes that run in a Pod.
When we (human) access the cluster (Ex - Using kubectl), we are authenticated by the apiserver as a particular User Account (This is usually admin, unless our cluster admin has customized our cluster.)
Processes in containers inside pods can also contact the apiserver. When they do, they are authenticated as a particular Service Account.

- Kubernetes uses Role-Based Access Control (RBAC) to define what actions different Service Accounts are allowed to perform within the cluster. By assigning appropriate roles and permissions to Service Accounts, you can control access to resources and API endpoints.

Argo CD

- It's a declarative, GitOps continuous delivery tool for Kuberentes.
- It follows the GitOps pattern of using Git repositories as the source of truth for defining the desired application state. K8s manifest can be specified in one of many ways for different environments & patches can be applied for customization.

****************** END ******************