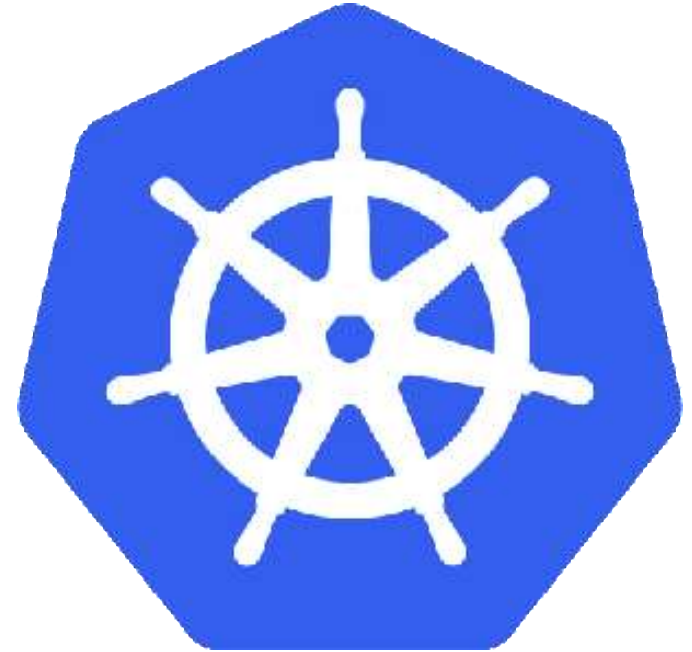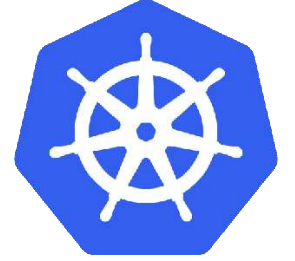# Kubernetes

Mithun Technologies

Kubernetes

Agenda

- What is Kubernetes?
- Advantages of Kubernetes
- Kubernetes Architecture
- Kubernetes Components
- Kubernetes Cluster Setup
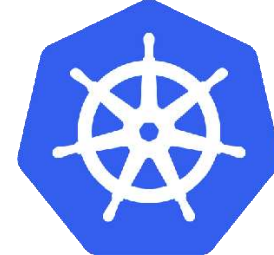- Kubernetes Objects
- Demo

What is Kubernetes?

- Kubernetes is an orchestration engine and open-source platform for managing containerized applications.

- Responsibilities include container deployment, scaling & descaling of containers & container load balancing.

- Actually, Kubernetes is not a replacement for Docker, But Kubernetes can be considered as a replacement for Docker Swarm, Kubernetes is significantly more complex than Swarm, and requires more work to deploy.

- Born in **Google** ,written in Go/Golang. Donated to CNCF(Cloud native computing foundation) in 2014.

- Kubernetes v1.0 was released on **July 21, 2015.**

- Current stable release v1.14.0.

Kubernetes Features

| 01	02	03	04
Automated Scheduling	**Self Healing Capabilities**	**Automated rollouts and rollback**	**Horizontal Scaling and Load Balancing**
Kubernetes provides advanced scheduler to launch container on cluster nodes	Rescheduling, replacing and restarting the containers which are died.	Kubernetes supports rollouts and rollbacks for the desired state of the containerized application	Kubernetes can scale up and scale down the application as per the requirements

The features of Kubernetes, are as follows:

•**Automated Scheduling:** Kubernetes provides advanced scheduler to launch container on cluster nodes based on their resource requirements and other constraints, while not sacrificing availability.

•**Self Healing Capabilities:** Kubernetes allows to replaces and reschedules containers when nodes die. It also kills containers that don't respond to user-defined health check and doesn't advertise them to clients until they are ready to serve.

•**Automated rollouts & rollback:** Kubernetes rolls out changes to the application or its configuration while monitoring application health to ensure it doesn't kill all your instances at the same time. If something goes wrong, with Kubernetes you can rollback the change.

•**Horizontal Scaling & Load Balancing:** Kubernetes can scale up and scale down the application as per the requirements with a simple command, using a UI, or automatically based on CPU usage.

Kubernetes Features
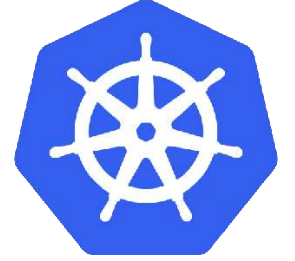
5. Service Discovery & Load balancing

With Kubernetes, there is no need to worry about networking and communication because Kubernetes will automatically assign IP addresses to containers and a single DNS name for a set of containers, that can load-balance traffic inside the cluster.
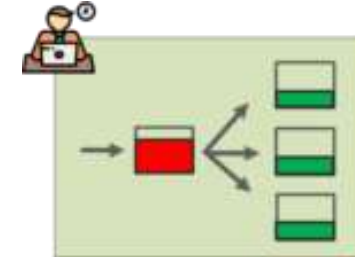
6. Storage Orchestration

With Kubernetes, you can mount the storage system of your choice. You can either opt for local storage, or choose a public cloud provider such as GCP or AWS, or perhaps use a shared network storage system such as NFS, iSCSI, etc.

Kubernetes Architecture

Kubernetes implements a cluster computing background, everything works from inside a ***Kubernetes Cluster***. This cluster is hosted by one node acting as the 'master' of the cluster, and other nodes as 'nodes' which do the actual 'containerization'. Below is a diagram showing the same.
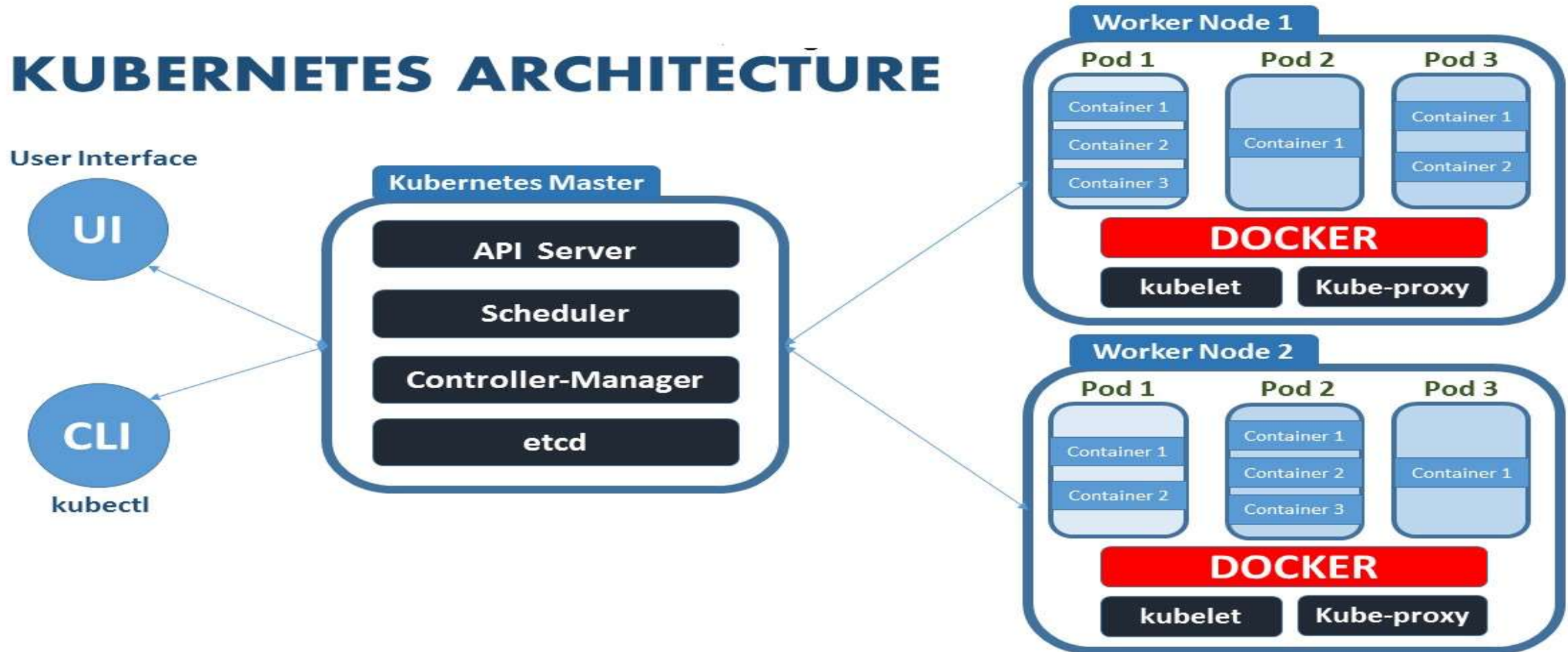
Kubernetes Components

Web UI (Dashboard)

Dashboard is a web-based Kubernetes user interface. You can use Dashboard to deploy containerized applications to a Kubernetes cluster, troubleshoot your containerized application, and manage the cluster itself along with its available resources.

Kubectl

Kubectl is a command line configuration tool (CLI) for Kubernetes used to interact with master node of kubernetes. Kubectl has a config file called kubeconfig, this file has the information about server and authentication information to access the API Server.

Master Node

The master node is responsible for the management of Kubernetes cluster. It is mainly the entry point for all administrative tasks. It handles the orchestration of the worker nodes. There can be more than one master node in the cluster to check for fault tolerance.
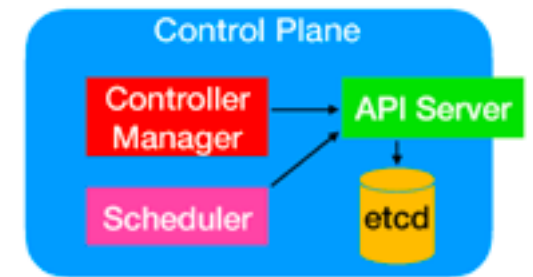
Master Components

It has below components that take care of communication, scheduling and controllers.

API Server:

Kube API Server interacts with API, Its a frontend of the kubernetes control plane. Communication center for developers, sysadmin and other Kubernetes components

Scheduler

Scheduler watches the pods and assigns the pods to run on specific hosts.

Kubernetes Components

Controller Manager:

Controller manager runs the controllers in background which runs different tasks in Kubernetes cluster. Performs cluster-level functions (replication, keeping track of worker nodes, handling nodes failures...).

Some of the controllers are,

1. Node controller - Its responsible for noticing and responding when nodes go down.

2. Replication controllers - It maintains the number of pods. It controls how many identical copies of a pod should be running somewhere on the cluster.

3. Endpoint controllers joins services and pods together.

4. Replicaset controllers ensure number of replication of pods running at all time.

5. Deployment controller provides declarative updates for pods and replicasets.

6. Daemonsets controller ensure all nodes run a copy of specific pods.

7. Jobs controller is the supervisor process for pods carrying out batch jobs

etcd

etcd is a simple distribute key value store. kubernetes uses etcd as its database to store all cluster datas. some of the data stored in etcd is job scheduling information, pods, state information and etc.

Kubernetes Components

Worker Nodes

- Worker nodes are the nodes where the application actually running in kubernetes cluster, it is also know as minion.

  These each worker nodes are controlled by the master node using kubelet process.

- Container Platform must be running on each worker nodes and it works together

  with kubelet to run the containers, This is why we use Docker engine

  and takes care of managing images and containers.

- We can also use other container platforms like CoreOS, Rocket.
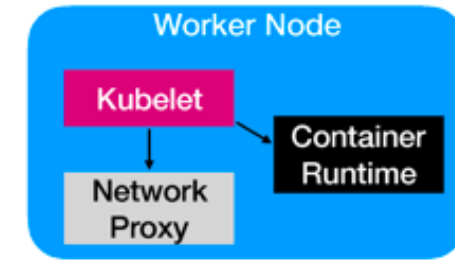
Node Components

Kubelet

- Kubelet is the primary node agent runs on each nodes and reads the container manifests which ensures that containers are running and healthy.

- It makes sure that containers are running in a pod. The kubelet doesn't manage containers which were not created by Kubernetes.

Kube-proxy

- kube-proxy enables the Kubernetes service abstraction by maintaining network rules on the host and performing connection forwarding.

- It helps us to have network proxy and load balancer for the services in a single worker node. Worker nodes can be exposed to internet via kubeproxy.

Container Runtime

- Each node must have a container runtime, such as Docker, rkt, or another container runtime to process instructions from the master server to run containers.

Installation

Different ways to install Kubernetes

Play-with-k8s   Google Kubernetes Engine(GKE)

https://labs.play-with-k8s.com.

Minikube   Amazon EKS

kubeadm   Azure Kubernetes Service (AKS)

Simplest   Most involved

Minikube   Google Container Engine (GKE)   AWS Provider   Manual install
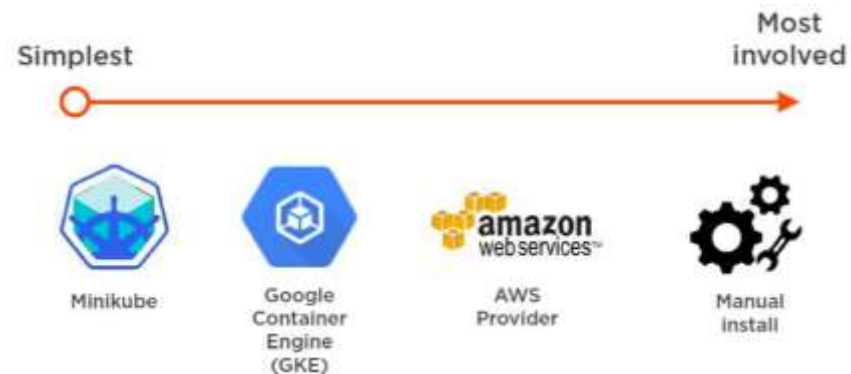
Check required ports

Control-plane node(s)

Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	6443*	Kubernetes API server	All
TCP	Inbound	2379-2380	etcd server client API	kube-apiserver, etcd
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	10251	kube-scheduler	Self
TCP	Inbound	10252	kube-controller-manager	Self

Worker node(s)

Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	30000-32767	NodePort Services**	All

Kubernetes Objects

- Kubernetes Objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster.

- A Kubernetes object is a "record of intent"–once you create the object, the Kubernetes system will constantly work to ensure that object exists.

- To work with Kubernetes objects–whether to create, modify, or delete them–you'll need to use the Kubernetes API. When you use the kubectl command-line interface, for example, the CLI makes the necessary Kubernetes API calls for you.

The basic Kubernetes objects include:

- Pod
- Replication Controller
- ReplicaSet
- DaemonSet
- Deployment
- Service
- Volume
- Job

Mithun Technologies

Kubernetes Objects

What is a Namespace?

You can think of a Namespace as a virtual cluster inside your Kubernetes cluster. You can have multiple namespaces inside a single Kubernetes cluster, and they are all logically isolated from each other. They can help you and your teams with organization, security, and even performance.

The first three namespaces created in a cluster are always **default**, **kube-system**, and **kube-public**. While you can technically deploy within these namespaces, I recommend leaving these for system configuration and not for your projects.

- **Default** is for deployments that are not given a namespace, which is a quick way to create a mess that will be hard to clean up if you do too many deployments without the proper information.

- **Kube-system** is for all things relating to the Kubernetes system. Any deployments to this namespace are playing a dangerous game and can accidentally cause irreparable damage to the system itself.

- **Kube-public** is readable by everyone, but the namespace is reserved for system usage.

Using namespaces for isolation

- I have used namespaces for isolation in a couple of ways. I use them most often to split many users' projects into separate environments. This is useful in preventing cross-project contamination since namespaces provide independent environments.
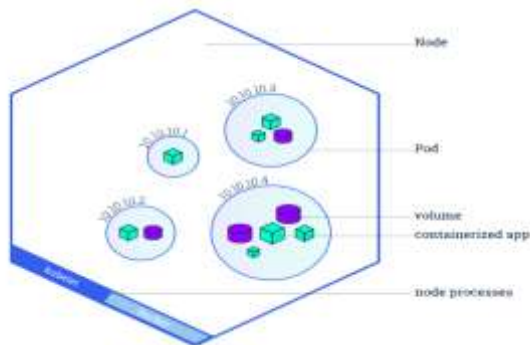
- kubectl get namespace

- kubectl create namespace test

https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-organizing-with-namespaces

Kubernetes Objects
POD


```
# nginx-pod.yaml
apiVersion: v1
kind: Pod
metadata:


spec:
```

Kind	apiVersion
Pod	v1
ReplicationController	V1
Service	v1
ReplicaSet	apps/v1
Deployment	apps/v1
DaemonSet	apps/v1
Job	batch/v1

- A Pod always runs on a **Node.**

- A pod is the smallest building block or basic unit of scheduling in Kubernetes.

- In a Kubernetes cluster, a pod represents a running process.

- Inside a pod, you can have one or more containers. Those containers all share a unique network IP, storage, network and any other specification applied to the pod.

- Pods abstract network and storage away from the underlying container.

- This lets you move containers around the cluster more easily.

- Each Pod has its unique IP Address within the cluster.

- Any data saved inside the Pod will disappear without

  a persistent storage.

```
apiVersion: v1
kind: Pod
metadata:
 name: nginx-pod
  labels:
   app: nginx
spec:
  containers:
   - name: first-container
     image: nginx
     ports:
     - containerPort: 80

Command
kubectl apply –f pod.yml
```
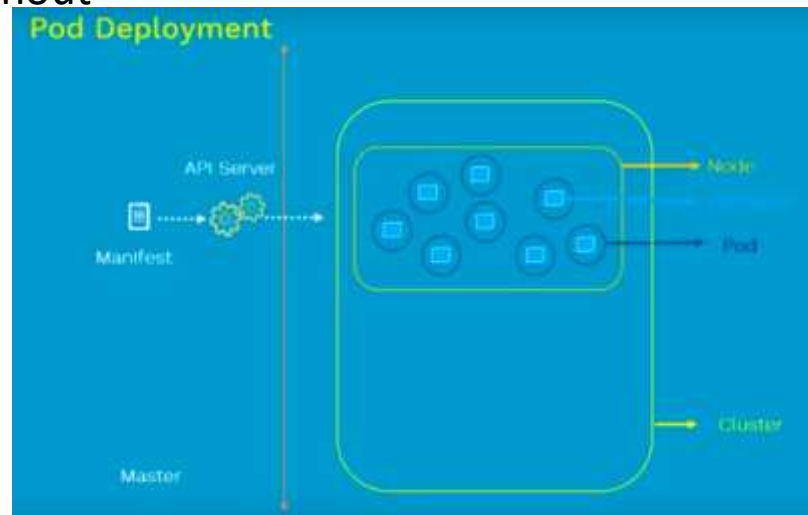
Kubernetes Objects

Pod Lifecycle

- Make a Pod reuqest to API server using a local pod defination file

- The API server saves the info for the pod in ETCD

- The scheduler finds the unscheduled pod and shedules it to node.

- Kubelet running on the node, sees the pod sheduled and fires up docker.

- Docker runs the container

- The entire lifecycle state of the pod is stored in ETCD.

Pod Concepts

- Pod is ephemeral(lasting for a very short time) and won't be rescheduled to a new node once it dies.

- You should not directly use Pod for deployment, Kubernetes have controllers like Replica Sets, Deployment, Deamon sets to keep pod alive.

Kubernetes Labels and Selectors

Labels

- When One thing in k8s needs to find another things in k8s, it uses labels.
- Labels are key/value pairs attached to Object
- You can make your own and apply it.
- it's like tag things in kubernetes

For e.g.

```
  labels:
    app: nginx
    role: web
    env: dev
```

Selectors

- Selectors use the label key to find a collection of objects matched with same value
- It's like Filter, Conditions and query to your labels

For e.g.

```
selectors:
  env = dev
  app != db
  release in (1.3,1.4)
```

- Labels and Selectors are used in many places like Services, Deployment and we will see now in Replicasets

Replication Controller

- Replication Controller is one of the key features of Kubernetes, which is responsible for managing the pod lifecycle. It is responsible for making sure that the specified number of pod replicas are running at any point of time.

- A Replication Controller is a structure that enables you to easily create multiple pods, then make sure that that number of pods always exists. If a pod does crash, the Replication Controller replaces it.

- Replication Controllers and PODS are associated with labels.

- Creating "RC" with count of 1 ensure that a POD is always available.
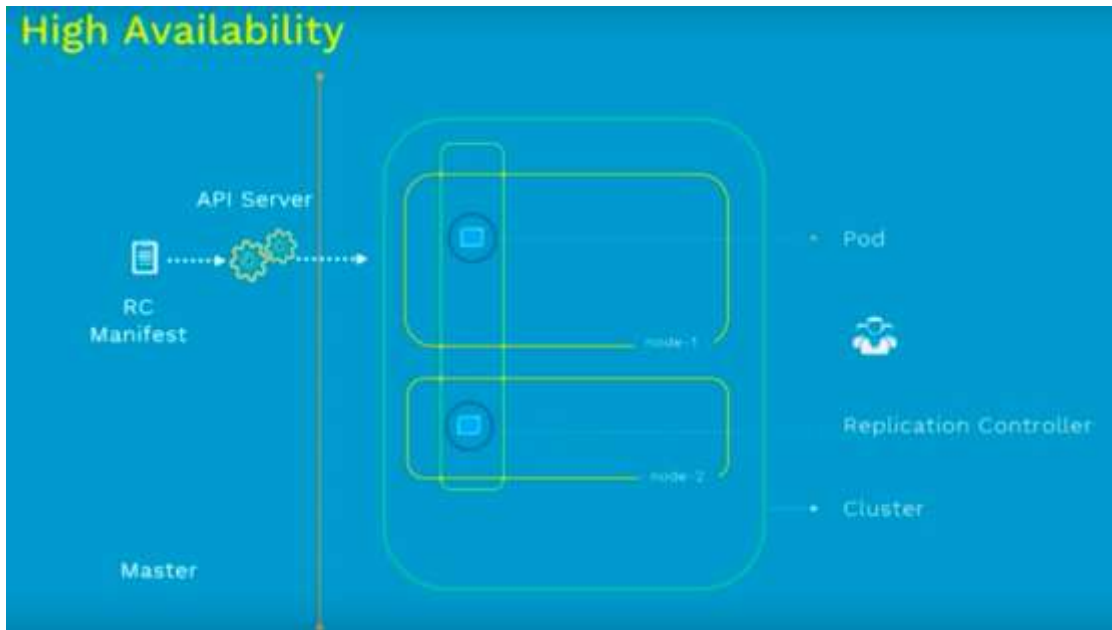
Command
kubectl apply –f rc.yml

```
apiVersion: v1
kind: ReplicationController
metadata:
 name: nginx-rc
spec:
 replicas: 2
 selector:
  app: nginx
 template:
  metadata:
   name: nginx
   labels:
    app: nginx
  spec:
   containers:
   - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

Mithun Technologies

ReplicaSet

- ReplicaSet is the next-generation Replication Controller.

- The only difference between a *ReplicaSet* and a *Replication Controller* right now is the selector support.

- ReplicaSet supports the new set-based selector requirements as described in the labels user guide whereas a Replication Controller only supports equality-based selector requirements.

  ```
  kubectl apply –f rs.yml
  kubectl scale rs nginx-replicaset --replicas 3
  ```

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx-rs-pod
    matchExpressions:
    - key: env
      operator: In
      values:
      - dev
  template:
    metadata:
      labels:
        app: nginx-rs-pod
        env: dev
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

DaemonSet

A *DaemonSet* make sure that all or some kubernetes Nodes run a copy of a Pod.

When a new node is added to the cluster, a Pod is added to it to match the rest of the nodes and when a node is removed from the cluster, the Pod is garbage collected.

Deleting a DaemonSet will clean up the Pods it created.

```
# Daemon Set
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx-ds
spec:
  selector:
    matchLabels:
      app: nginx-pod
  template:
    metadata:
      name: nginx-pod
      labels:
        app: nginx-pod
    spec:
      nodeSelector:
        name: node1
      containers:
        - name: webserver
          image: nginx
          ports:
          - containerPort: 80
```

Command
kubectl apply –f daemonset.yml

Mithun Technologies

Kubernetes Objects

Deployment

In Kubernetes, Deployment is the recommended way to deploy Pod or RS, simply because of the advance features it comes with.

Below are some of the key benefits.

- Deploy a RS.
- Updates pods (PodTemplateSpec).
- Rollback to older Deployment versions.
- Scale Deployment up or down.
- Pause and resume the Deployment.
- Use the status of the Deployment to determine state of replicas.
- Clean up older RS that you don't need anymore.

kubectl scale deployment *[DEPLOYMENT_NAME]* --replicas *[NUMBER_OF_REPLICAS]*

kubectl rollout status deployment nginx-deployment

kubectl get deployment

kubectl set image deployment nginx-deployment nginx-container=nginx:latest --record

kubectl get replicaset

kubectl rollout history deployment nginx-deployment

kubectl rollout history deployment nginx-deployment **--revision**=1

kubectl rollout undo deployment nginx-deployment --to-revision=1

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Mithun
Technologies

Kubernetes Objects

Deployment Strategies

There are several different types of deployment strategies you can take advantage of depending on your goal.

Rolling Deployment

The rolling deployment is the standard default deployment to Kubernetes. It works by slowly, one by one, replacing pods of the previous version of your application with pods of the new version without any cluster downtime.

A rolling update waits for new pods to become ready before it starts scaling down the old ones. If there is a problem, the rolling update or deployment can be aborted without bringing the whole cluster down. In the YAML definition file for this type of deployment, a new image replaces the old image.
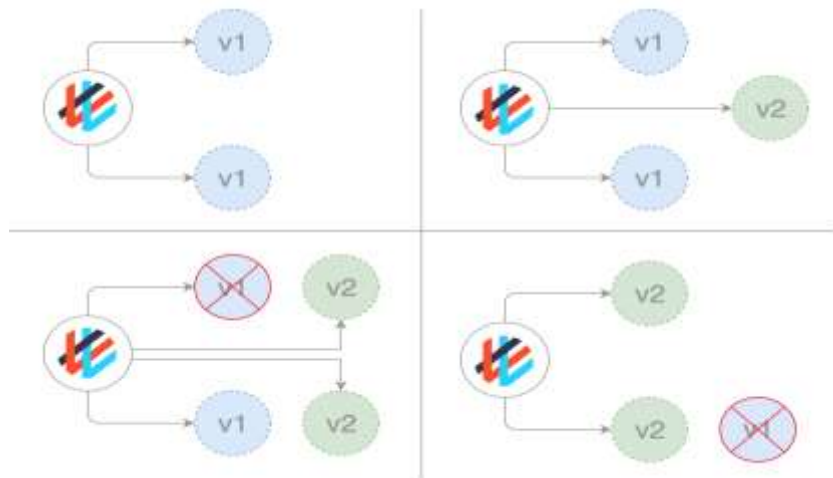
```
# Deployments  Rolling
Update
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  minReadySeconds: 30
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      name: nginx-pod
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```


Mithun
Technologies

Kubernetes Objects

Deployment Strategies

Recreate

- In this type of very simple deployment, all of the old pods are killed all at once and get replaced all at once with the new ones.
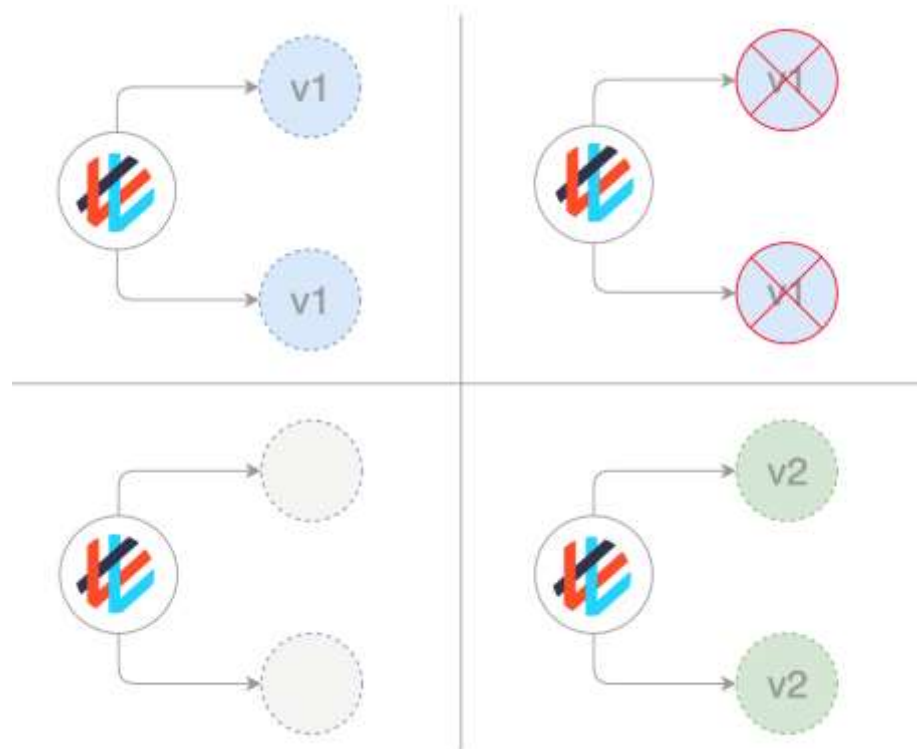

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Kubernetes Objects

Deployment Strategies

Blue/ Green (or Red / Black) deployments

In a blue/green deployment strategy (sometimes referred to as red/black) the old version of the application (green) and the new version (blue) get deployed at the same time. When both of these are deployed, users only have access to the green; whereas, the blue is available to your QA team for test automation on a separate service or via direct port-forwarding.

After the new version has been tested and is signed off for release, the service is switched to the blue version with the old green version being scaled down:

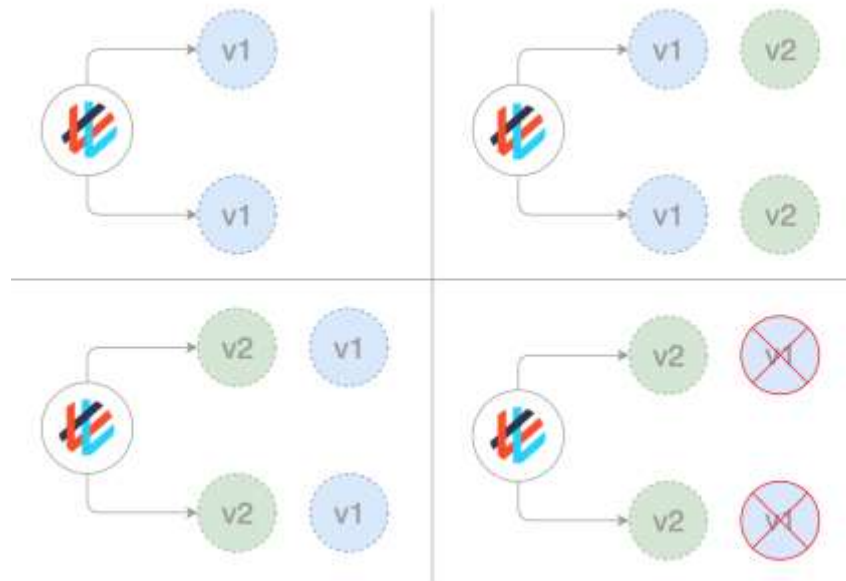apiVersion: v1

kind: Service

metadata:

  name: awesomeapp

spec:

  selector:

    app: awesomeapp

    version: "02"

Kubernetes Objects

Service

A service is responsible for making our Pods discoverable inside the network or exposing them to the internet. A Service identifies Pods by its LabelSelector.

Types of services available:

ClusterIP – Exposes the service on a cluster-internal IP. Service is only reachable from within the cluster. This is the default Type.


```
# Service Cluster IP
apiVersion: v1
kind: Service
metadata:
  name: my-internal-service
spec:
  selector:
    app: nginx
  type: ClusterIP
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
```
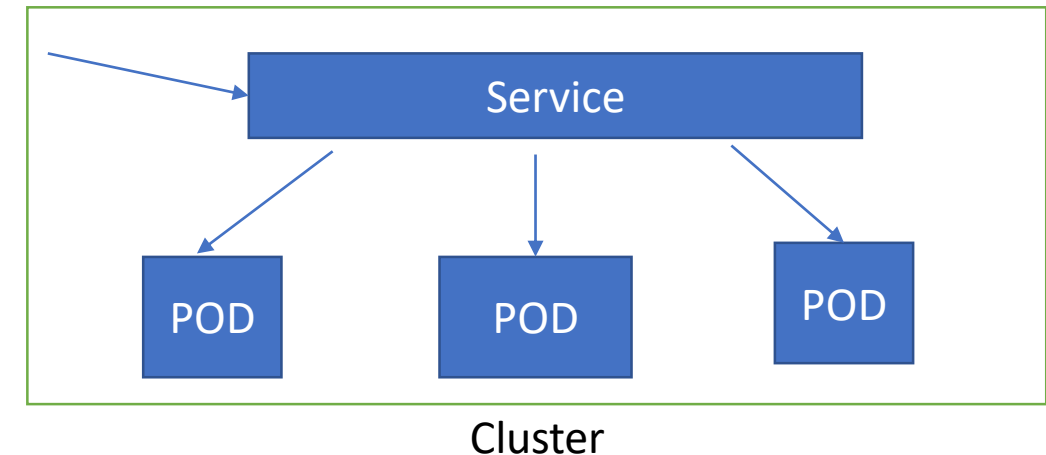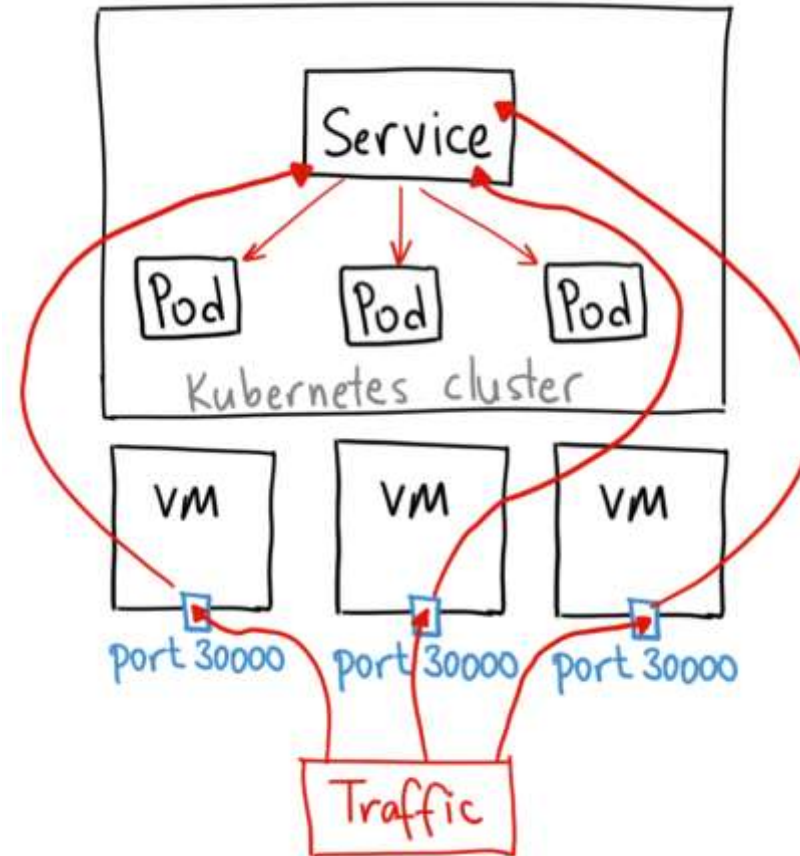
Service

POD   POD   POD

Cluster

Kubernetes Objects

NodePort – Exposes the service on each Node's IP at a static port. A ClusterIP service, to which the NodePort service will route, is automatically created. You'll be able to contact the NodePort service, from outside the cluster, by using "*<NodeIP>:<NodePort>*".

```
apiVersion: v1
kind: Service
metadata:
  name: my-nodeport-service
spec:
  selector:
    app: nginx-rs-pod
  type: NodePort
  ports:
  - name: http
    port: 80
    targetPort: 80
    nodePort: 30036
    protocol: TCP
```
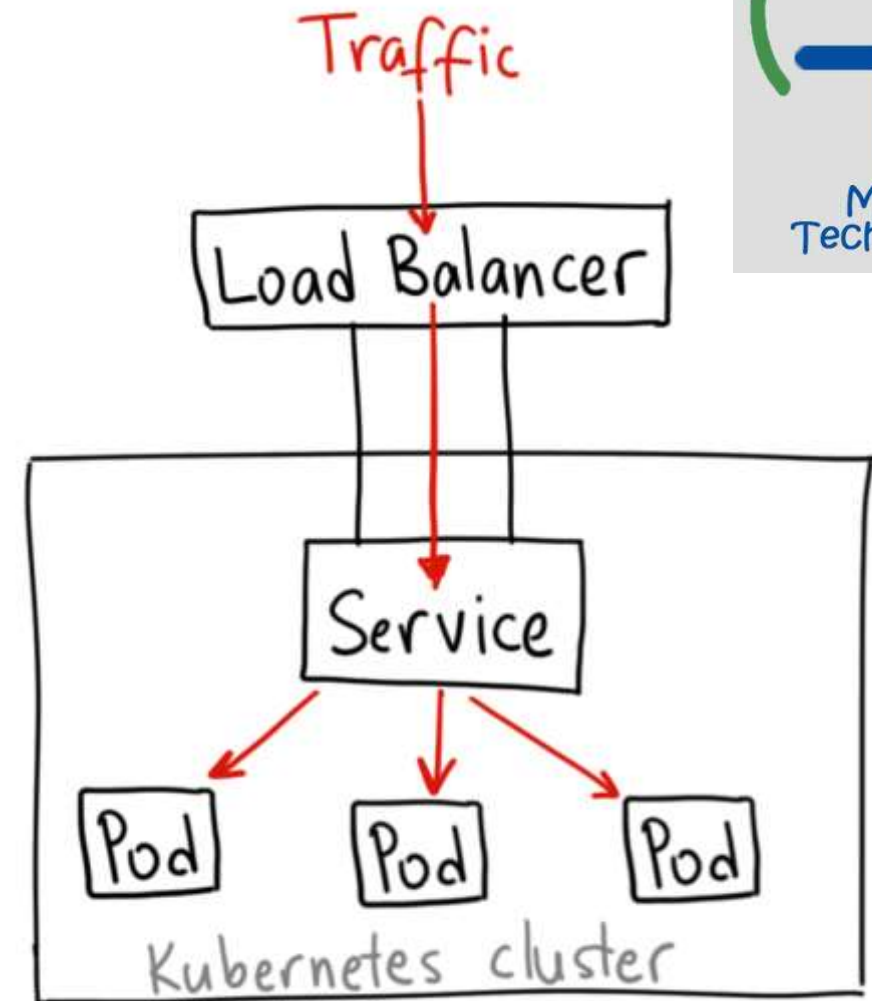
Kubernetes Objects

LoadBalancer – Exposes the service externally using a cloud provider's load balancer. NodePort and ClusterIP services, to which the external load balancer will route, are automatically created.

If you are using a custom Kubernetes Cluster
(using minikube, kubeadm or the like). In this case, there is no
LoadBalancer integrated (unlike AWS EKS or Google Cloud). With
this default setup, you can only use NodePort.
Configure Load Balancer Externally.

```
apiVersion: v1
kind: Service
metadata:
  name: nginxservice
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
  selector:
    app: nginx
  type: LoadBalancer
```

Kubernetes Objects

Ingress – *Kubernetes Ingress is a resource to add rules for routing traffic from external sources to the services in the kubernetes cluster.*

Kubernetes Ingress:

Kubernetes Ingress is a native kubernetes resource where you can have rules to route traffic from an external source to service endpoints residing inside the cluster. It requires an ingress controller for routing the rules specified in the ingress object.

Kubernetes Ingress Controller

Ingress controller is typically a proxy service deployed in the cluster. It is nothing but a kubernetes deployment exposed to a service. Following are the ingress controllers available for kubernetes.

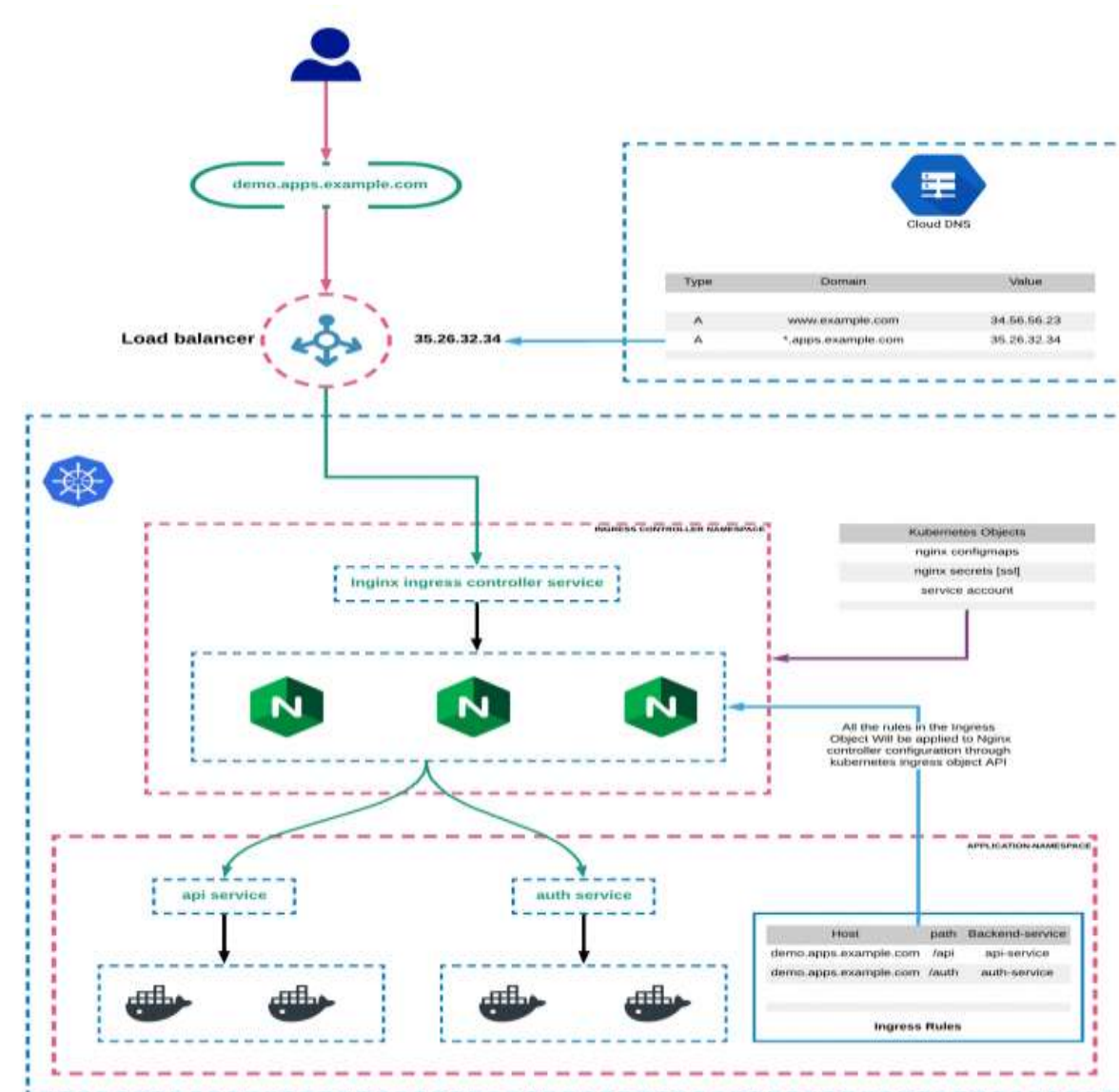Following are the ingress controllers available for kubernetes.
Nginx Ingress Controller (Community & From Nginx Inc)
Traefik
HAproxy
Contour
GKE Ingress Controller

Kubernetes Volumes

A Container's file system lives only as long as the Container does. So when a Container terminates and restarts, filesystem changes are lost. Volumes in Kubernetes are very easy to manage. It is basically a directory that gets mounted to a pod. After telling the container to use Volumes for storing evergreen information, you can safely modify the pods without ever losing your data.

Kubernetes supports many types of volumes, and a Pod can use any number of them simultaneously. For more consistent storage that is independent of the Container, we can use a Volume. This is especially important for stateful applications, such as databases.

To use a volume, a Pod specifies what volumes to provide for the Pod
(the .spec.volumesfield) and where to mount those into Containers
(the .spec.containers.volumeMounts field) as shown.

Kubernetes supports several types of Volumes:
awsElasticBlockStore
azureDisk
azureFile
configMap
emptyDir
gcePersistentDisk
gitRepo (deprecated)
hostPath
nfs
persistentVolumeClaim
secret

```yaml
# Mongo host path rc
apiVersion: v1
kind: ReplicationController
metadata:
  labels:
    name: mongo
  name: mongo-controller
spec:
  replicas: 1
  template:
    metadata:
      labels:
        name: mongo
    spec:
      containers:
      - image: mongo
        name: mongo
        ports:
        - name: mongo
          containerPort: 27017
          hostPort: 27017
        volumeMounts:
        - name: mongo-persistent-storage
          mountPath: /data/db
      volumes:
      - name: mongo-persistent-storage
        hostPath:
          path: /tmp/dbbackup
```

Kubernetes Volumes

Persistent Volumes

Persistent Volumes are simply a piece of storage in your cluster. Similar to how you have a disk resource in a server, a persistent volume provides storage resources for objects in the cluster. At the most simple terms you can think of a PV as a disk drive. It should be noted that this storage resource exists independently from any pods that may consume it. Meaning, that if the pod dies, the storage should remain intact assuming the claim policies are correct. Persistent Volumes are provisioned in two ways, Statically or Dynamically.

Static Volumes – A static PV simply means that some k8s administrator provisioned a persistent volume in the cluster and it's ready to be consumed by other resources.

Dynamic Volumes – In some circumstances a pod could require a persistent volume that doesn't exist. In those cases it is possible to have k8s provision the volume as needed if storage classes were configured to demonstrate where the dynamic PVs should be built. This post will focus on static volumes for now.

Persistent Volume Claims

Pods that need access to persistent storage, obtain that access through the use of a Persistent Volume Claim. A PVC, binds a persistent volume to a pod that requested it.

When a pod wants access to a persistent disk, it will request access to the claim which will specify the size , access mode and/or storage classes that it will need from a Persistent Volume. Indirectly the pods get access to the PV, but only through the use of a PVC.

Claim Policies

We also reference claim policies earlier. A Persistent Volume can have several different claim policies associated with it including:

Retain – When the claim is deleted, the volume remains.

Recycle – When the claim is deleted the volume remains but in a state where the data can be manually recovered.

Delete – The persistent volume is deleted when the claim is deleted.

The claim policy (associated at the PV and not the PVC) is responsible for what happens to the data on when the claim has been deleted.

Kubernetes Volumes
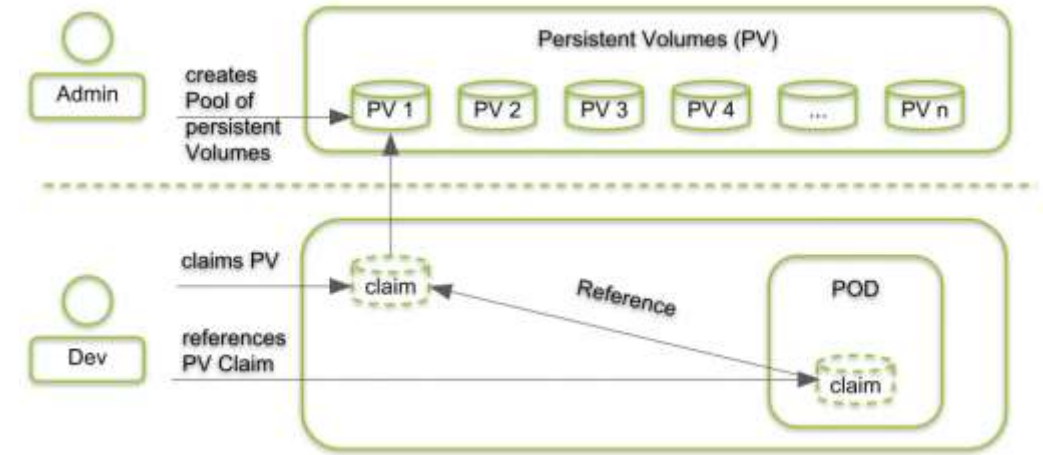
ConfigMaps and Secrets

Application's configuration can change between environments (e.g. development, staging, production, etc.) and that you want your applications to be portable. It is a good practice to make container images as reusable as possible. The same image should be able to be used for development, staging, and production.

Thus, you should store the config(like db connection details, passwords, API Keys ..etc) outside of the application itself.

With Docker and containers this means we should try to keep configuration out of the container image. Developer should not hard code configuration details instead they can refer/use **Environment Variables.**

Docker makes it easy to build containers with environment variables baked in. In your Dockerfile, you can specify them with the ENV directive.
You can also override the environment variables when running on the command line.

ConfigMaps and Secrets

Kubernetes allows you to provide configuration information to our applications through ConfigMaps or secret resources. The main differentiator between the two is the way a pod stores the receiving information and how the data is stored in the etcd data store.

Both, ConfigMaps and Secrets store data as a key value pair. The major difference is, **Secrets store data in base64** format meanwhile **ConfigMaps store data in a plain text**.

If you have some critical data like, keys, passwords, service accounts credentials, db connection string, etc then you should always go for **Secrets** rather than Configs.

And if you want to do some application configuration using environment variables which you don't want to keep secret/hidden like, app base platform url, etc then you can go for **ConfigMaps.**

Secret:
docker-registry
This is used by the kubelet when passed in a pod template if there is an **imagePullsecret** to provide the credentials needed to authenticate to a private Docker registry:

kubectl create secret docker-registry registryKey --docker-server <PrivateRepoURL> --docker-username <userName> --docker-password <password> --docker-email <email>
https://opensource.com/article/19/6/introduction-kubernetes-secrets-and-configmaps
https://medium.com/google-cloud/kubernetes-configmaps-and-secrets-68d061f7ab5b
https://www.oreilly.com/library/view/kubernetes-best-practices/9781492056461/ch04.html

Liveness And Readiness Probes

Liveness and Readiness probes are used to control the health of an application running inside a Pod's container.

Liveness Probe

- Suppose that a Pod is running our application inside a container, but due to some reason let's say memory leak, cpu usage, application deadlock etc the application is not responding to our requests, and stuck in error state.

- Liveness probe checks the container health as we tell it do, and if for some reason the liveness probe fails, it restarts the container.

Readiness Probe

- This type of probe is used to detect if a container is ready to accept traffic. You can use this probe to manage which pods are used as backends for load balancing services. If a pod is not ready, it can then be removed from the list of load balancers.

https://docs.okd.io/latest/dev_guide/application_health.html

https://www.weave.works/blog/resilient-apps-with-liveness-and-readiness-probes-in-kubernetes

Questions ?