## **Container Orchestration**

Container orchestration is all about managing the life cycles of containers, especially in large, dynamic environments.

Container orchestration can be used to perform lot of tasks, some of them includes:

* Provisioning and deployment of containers
* Scaling up or removing containers to spread application load evenly
* Movement of containers from one host to another if there is a shortage of resources.
* Load balancing of service discovery between containers
* Health monitoring of containers and hosts

## **Monolithic Architecture**

Monolith means composed all in one piece. The **Monolithic** application describes a single-tiered **software** application in which different components combined into a single program from a single platform.

Packaged and deployed as single unit.

Components

* Presentation Layer

Front End [User Interface] Responsible for handling http request. Build with Web technologies like HTML, JAVASCRIPT, CSS etc.., It communicates other layer usng API calls.

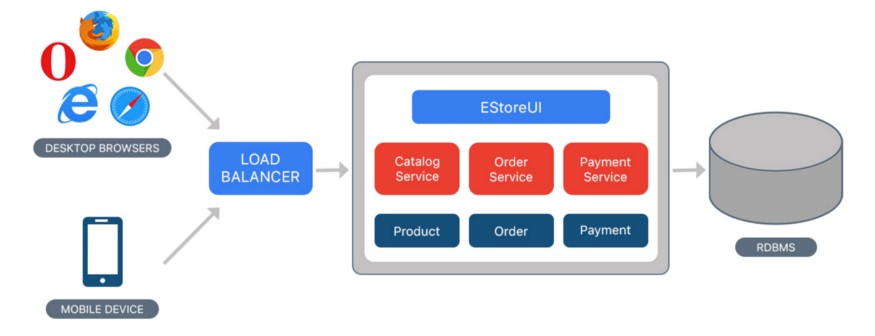
* Application Layer

Business logic often written in Java, .Net, C-sharp, Python

* Data Layer

Where all the data get stored and fetched using API calls. MYSQL, ORACLE, REDDIS, MONGODB

**Example**



* Consider an example of Ecommerce application, that authorizes customer, takes an order, check products inventory, authorize payment and ships ordered products. This application consists of several components including e-Store User interface for customers (Store web view) along with some backend services to check products inventory, authorize and charge payments and shipping orders.
* Despite having different components/modules/services, the application is built and deployed as one Application for all platforms (i.e. desktop, mobile and tablet) using RDBMS as a data source.

**Benefits**

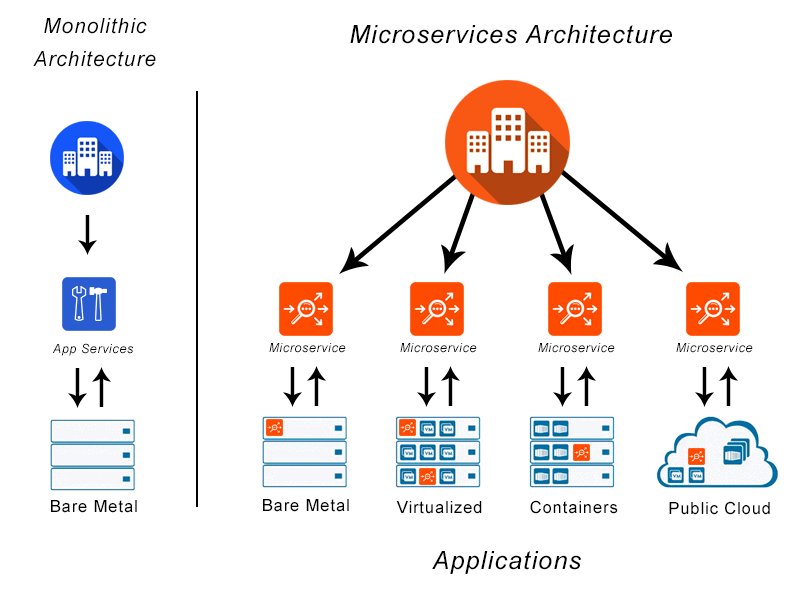
* Single unit of deployment
* Simple to develop — At the beginning of a project it is much easier to go with Monolithic Architecture.
* Simple to test. For example, you can implement end-to-end testing by simply launching the application and testing the UI with Selenium.
* Simple to deploy. You have to copy the packaged application to a server.
* Simple to scale horizontally by running multiple copies behind a load balancer.

**Drawbacks**

* A large code base can be significantly harder to understand.
* Technology dependent on initail decision
* All components impemetned using single development stack, Java or python.
* Frequent deployment are not practical
* Bug in any module (e.g. memory leak) can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug impact the availability of the entire application
* You must redeploy the entire application on each update.
* Need to scale entire Application stack

## **What is Microservices ??**

* Here in Microservices, The Application is composed of multiple services, which can be developed, deployed and maintained independently. Each of these services is responsible for discrete task and can communicate with other services through simple APIs to solve a large complex business problem.



## **When to use Microservices ?**

Ultimately, any size company can benefit from the use of a microservices architecture if they have applications that need frequent updates, experience dynamic traffic patterns, or require near real-time communication.

## **Microsservice Benefits**

* As the constituent services are small, they can be built by one or more small teams from the beginning separated by service boundaries which make it easier to scale up the development effort if need be.
* Once developed, these services can also be deployed independently of each other and hence its easy to identify hot services and scale them independent of whole application. Microservices also offer improved fault isolation whereby in the case of an error in one service the whole application doesn’t necessarily stop functioning. When the error is fixed, it can be deployed only for the respective service instead of redeploying an entire application.
* Another advantage which a microservices architecture brings to the table is making it easier to choose the technology stack (programming languages, databases, etc.) which is best suited for the required functionality (service) instead of being required to take a more standardized, one-size-fits-all approach.

Flexible Scaling Easy Deployment Technology Freedom Resuable code

Resilience

## **Disadvanages**

* First, communication between services can be complex. An application can include dozens or even hundreds of different services, and they all need to communicate securely.
* Second, debugging becomes more challenging with microservices. With an application consisting of multiple microservices and with each microservice having its own set of logs, tracing the source of the problem can be difficult.
* Up-front costs may be higher with microservices.
* For microservices architecture to work for your organization, you need sufficient hosting infrastructure with security and maintenance support, and you need skilled development teams who understand and manage all the services.

## **How are Microservies are deployed ?**

Deployment of microservices requires the following:

• Ability to scale simultaneously among many applications, even when each service has different amounts of traffic

• Quickly building microservices which are independently deployable from others

• Failure in one microservice must not affect any of the other services

Docker is a standard way to deploy microservices using the following steps:

• Package the microservice as a container image

• Deploy each service instance as a container

• Scaling is done based on changing the number of container instances

* Using Kubernetes with an orchestration system like Docker in deployment allows for management of a cluster of containers as a single system. It also lets enterprises run containers across multiple hosts while providing service discovery and replication control. Large scale deployments often rely on Kubernetes.

## **What is Kubernetes ??**

* Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation.
* It’s a container orchestration tool

## **Why you need Kubernetes and what it can do**

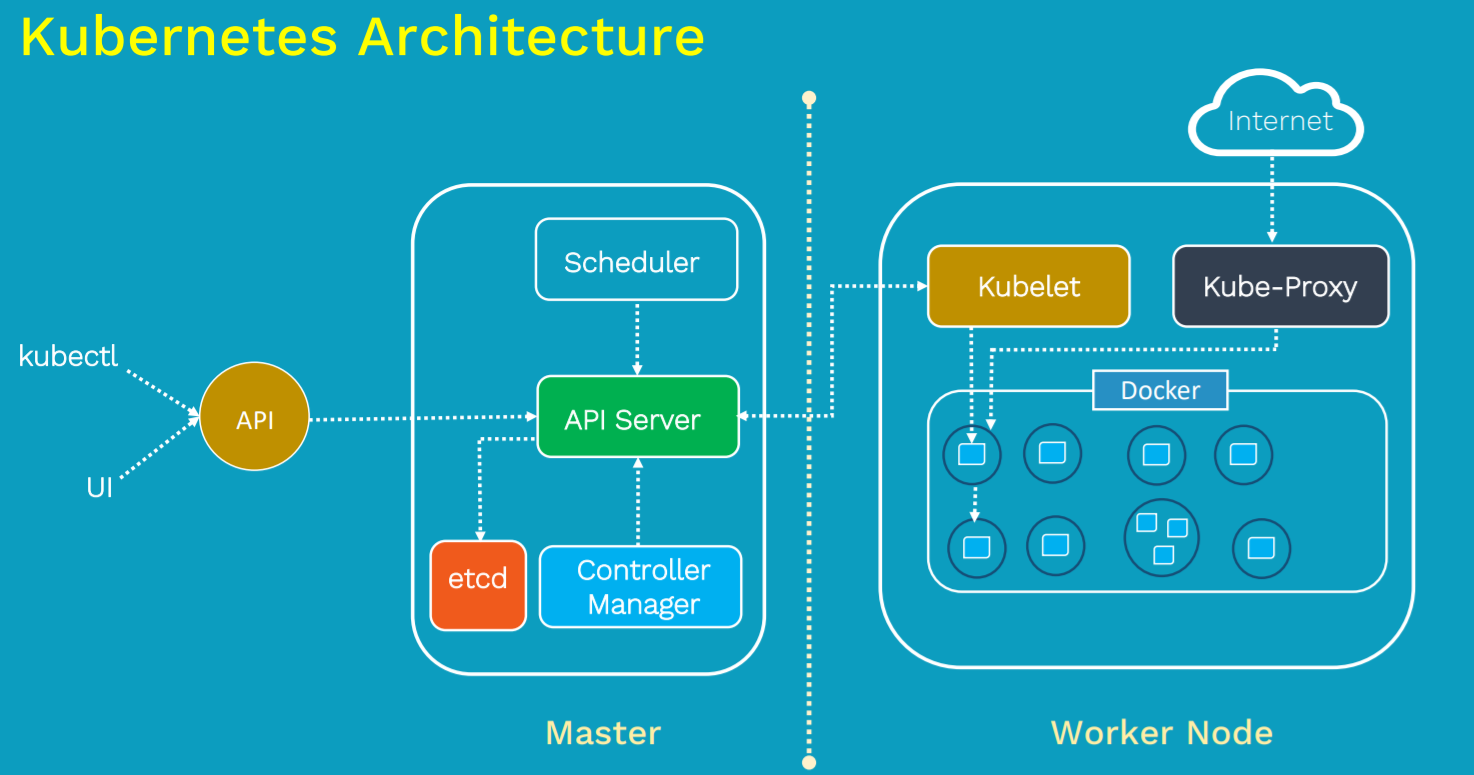
Containers are a good way to bundle and run your applications. In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime. For example, if a container goes down, another container needs to start. Wouldn't it be easier if this behavior was handled by a system?

That's how Kubernetes comes to the rescue! Kubernetes provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more. For example, Kubernetes can easily manage a canary deployment for your system.

**Kubernetes provides you with:**

* Service discovery and load balancing Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
* Storage orchestration Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
* Automated rollouts and rollbacks You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
* Automatic bin packing You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
* Self-healing Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
* Secret and configuration management Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.

## **Kubernetes Architecture**

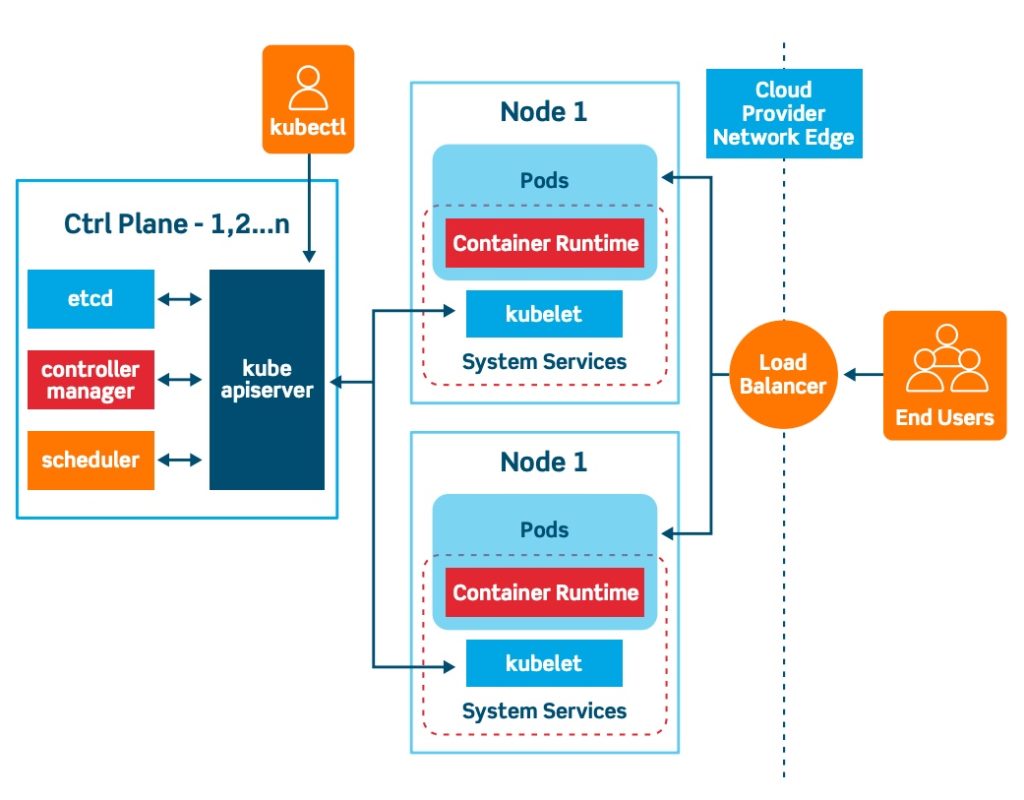


Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated



|  |  |
| --- | --- |
| **Control Panel | Master** | **Worker Node** |
| Kube-Api Server | Kubelet |
| Scheduler | Kube-proxy |
| Controller Manager | Pod |
| ETCD | Containers |

## **ETCD**

Kube API is the primary management component in Kubernetes. When you run a kubectl command the kubectl utility is infact reaching to the kube-api server. The kube-api server first authenticates the request and validates it, then retrieves the data from the ETCD cluster and responds back to with the request information.  
It’s a distributed reliable key-value store that is simple secure & fast.  
  
**ETC Distributed**  
Traditional Database  
  
Table

Description automatically generated

**What is Key Value Store ??**

* You put a key && value , it saves in the database. You get the key which returns the value.  
  You cannot have duplicate keys.  
    
  It is not replacement to tradional database, it is used to store and retrieve the small data such as configuration data which requires fast read and write data.

Graphical user interface, table

Description automatically generated

**Install ETCD**

Text

Description automatically generated

**ETCD Operation**

When you RUN etcd , its starts a service listens on port 2379 by default, then you can attach any client to ETCD service to store and retrieve the information.

**Default client for ETCD is etcdctl**

Graphical user interface, text

Description automatically generated

**ETCD in Kubernetes**

ETCD DataBase stores information regarding the clusters such as

**Nodes | PODS | Configs | Secrets | Accounts | Roles | Bindings | Others**

* Every Information you see when you run the kubectl get command is from the ETCD server.
* Every change you make to your cluster such as adding additional nodes, deploying pods or replica sets are updated in the ETCD server.

Depending on how you setup your cluster, ETCD is deployed differently

## **Kube-API server**

**Kube-API server act as Gateway server.**

Graphical user interface, text, application

Description automatically generated

Instead Kubectl you can also invoke the API-Server directltry using APIs by sending post request.

Graphical user interface, application

Description automatically generated

**Workflow**

* When you create POD,
  + Request is first authenticated && validated.
  + API server creates POD object without assigning it to a node, updates the information in the ETCD server and updates the user that POD is created.
  + The scheduler continoulsy monitor the API server and realizes the new pod is created with no POD assigned then it Identifies the right node to place the new POD on and communicate to the API server.
  + API server then updates the nformation in the ETCD cluster . Then pass it to the kubelet in approviare worker node.
  + Now Kubelet created the POD on the node and instructs the container runtime engine to deploy the application image.
  + Once done the kubelet updates the status back to the API server and the API server then updates the data back to ETCD server.
  + Everytime change is requested similar pattern is followed.

**Summary**  
KUBE-API SERVER responsible for

* + - Authenticate User
    - Validate request
    - Retrieve data
    - Update ETCD
    - Scheduler
    - Kublet updates

**WorkFlow 2**  
Diagram

Description automatically generated

* Kubectl writes to the API server (kubectl run mywebserver --image=nginx)
* API server will authenticate and authorize. Upon validation, it will write it to etcd.
* Upon write to etcd, API Server will invoke the scheduler.
* Scheduler decides which node the pod should run and return data to API server. API will in-turn write it back to etcd.
* API Server will invoke the kubelet in the node decided by the scheduler.
* Kubelet communicates to the docker daemon via Docker socket to create the container.
* Kubelet will update the status of the POD back to the API server.
* API Server will write the status details back to etcd.

## **Kube Controller Manager**

* It manages Various controller in Kubernetes. A controller is a department in the master node that have a own set of responsibilities.
* **Controller is a process that continuously monitors the state of various components within the system and works towards bring the whole system to the desired functioning state.**
* For example the node controller is reposnible for monitoring the status of the nodes and taking necessary actions to the application running. It does through the Kube-API server.
* **NODE Controller** checks the status of the node every 5 seconds. That way the node controller can monitor the health of the nodes if it stops receiving heartbeat from a node, then the node is marked as unreachable but it waits for 40 seconds before marking it unreachable.
* After a node is marked unreachable it gives it five min to come back up , if it doesn’t it removes the POD and provisioned them on the healthy ones.If the PODs are part of a replica set.
* **Replcation Controller** : It is responsible for monitoring the status replica sets and ensuring that the desired number of PODS are available at all times within the set. If POD dies it creates another one.
* Similar there are many controlles available, all are packaged into single process called kube controller Manager.

Diagram

Description automatically generated

**Some types of these controllers are:**

* Node controller: Responsible for noticing and responding when nodes go down.
* Job controller: Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
* Endpoints controller: Populates the Endpoints object (that is, joins Services & Pods).
* Service Account & Token controllers: Create default accounts and API access tokens for new namespaces.

Diagram

Description automatically generated

## **Cloud Controller Manager**

A Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster.

The cloud-controller-manager only runs controllers that are specific to your cloud provider. If you are running Kubernetes on your own premises, or in a learning environment inside your own PC, the cluster does not have a cloud controller manager.

As with the kube-controller-manager, the cloud-controller-manager combines several logically independent control loops into a single binary that you run as a single process. You can scale horizontally (run more than one copy) to improve performance or to help tolerate failures.

The following controllers can have cloud provider dependencies:

* Node controller: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
* Route controller: For setting up routes in the underlying cloud infrastructure
* Service controller: For creating, updating and deleting cloud provider load balancers

## **Kube Scheduler**

* Scheduler is only responsible for deciding which pod goes on which node. It doesn’t actually place the nodes, that’s the job of the kubelet.
* The scheduler decides which nodes the pods are placed on depending on certain criteria. You may have PODs with dif resource requirements. You can have nodes in the cluster dedicated to certain applications.

**How it will assign ??**

* The scheduler looks for each POD and tries to find best node for it.

For example, below one it has set of CPU and memory requirements. The scheduler goes in two phases to identify the best PODs.

A picture containing text, electronics, screenshot

Description automatically generated

* First phase it filter out the nodes that do not fit the profile for this POD. Here the nodes that do not have suffient CPU and memory resources requested by the pod.
* Second phase : Now schedules rank the the nodes to identiy the best fit for thr POD. It uses priority function to assign a scrore to the nodes on a scale of 0 to 10. Here in this scheduler calculates the amount of resources that free on the node after placing the pod on them.
* It can ble customized and you can create won scheduler

## **Kubelet**

* The kubelet in the Kubernetes worker node, registers the node with the Kubernetes cluster.
* When it receives instructions to load a container or a POD on the node, it requests the container run time engine which might be Docker , to pull the required the image and run an instance.
* The kubelet then continues to monitor the state of the POD and containers in it and report to the Kube-Api server on timely basis.
* You must always manually install the kubelet on your worker node.

## **Kube Proxy**

Within kubenetes cluster every POD can reach every other POD using **POD networking solution**.

A POD network is an internal virtual network that spans across all the nodes in the cluster to which all the POD connects to . Through this network are able to communicate with each other.

Diagram

Description automatically generated with medium confidence

There are many solutions available for deploying such a Network

**Example**.

* In this case I have a webapplciation deployed on the first node and a database application deployed on second node.
* The web app can reach the database, simply by using the IP of the database POD. But there is no guarantee that the IP of the database part Will always remain the same.
* A better way for th web application to access the database is using a **service**. So we create a service to expose the database application across the cluster.
* Now the web application can now access the database using the name of the service db.
* The service also gets an IP address assigned to it, whenever a POD tries to reach the service using its IP or name it forwards the traffic to the back end pod. In this case the database.

**But what is this service & how does it get an IP ?**

Does the service join the same POD network ?

No the service cannot join the pod network because the service is not actual thing. It is not a container like pod so it doesnt have any interfaces or an actively listening process.

It is a virtual component that only lives in the cabinet as memory. But then we also said that the service should be accessible across the cluster from any node .

That’s where kube-proxy comes in .

* Kube-proxy is a process that runs on each node in the Kubernetes cluster. Its job is to look for new services and every time a new service is created it creates the appropriate rules on each node to forward traffic to those services to the back end pods.

One way it does this is using IPTABLES rules.

In this case it creates an IP table on each node in the cluster to forward traffic heading to the IP of the service which is 10.96.0.12 to the IP of the actual pod which is 10.32.0.15. So that how kube-proxy configure the service.

## **Pods**

**What is POD ??**

* Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.
* A Pod (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled.
* It’s a Run time Environment using which we deploy our application.
* Here in POD containers are encapsulated inside the POD.
* A POD in Kubernetes represents a group of one or more application containers, and some shared resources [Volume] for those containers.
* A POD is always runs on a node.
* A node is a worker machine in Kubernetes
* Each node is managed by the Master
* A node can have Mutiple PODS

Graphical user interface, application

Description automatically generated

**Multi Container POD**  
  
Diagram

Description automatically generated

Sometimes we come across a scenario where you have a helper container that might be doing some kind of supporting task for me in the back, such as processing a user into data,or processing file uploaded bya user or etc.

And you want thic container to live along side of s container.In that case we can have both these container into same POD, so that when new container is created helper container also gets created and it dies when app container dies.

**POD/Object Creation Using Manifest/Yaml File**Most of the Kubernetes Objects consist of four Top Level fields, Even in POD manifest files it contains same 4 Top level fields.

apiVersion kind metadata spec

**apiVersion**

* It define the version number in which the Kubernetes objects belongs to ,
* If a **API version is V1** then the Kubernetes object is part of 1st stable release of kubernetes API. It container many of the core functions like
  + POD
  + ReplicationController
  + service.
* **Apps/v1** includes the functionality related to running apps on Kubernetes like
  + Deployment
  + Rolling updates
  + Replicaset
* **Batch/v1** containes objects related to batch processing and jobs like tasks.

**Kind**

Defines the kind of object you are creating

**metadata**

It has 2 fields Name of the objects we creating && labels is optionals.

Labels comes in handy when comes to filtering , Assume you have 1000 of PODS running and now you want to filter that related to nginx.

**spec**

Define container configuration.  
  
Graphical user interface

Description automatically generated with medium confidence

**POD Creation and Display**  
  
A picture containing graphical user interface

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated with medium confidence

## **Labels && Selectors** Without Labels

Graphical user interface, application, Teams

Description automatically generated

With Labels

Graphical user interface, application, Teams

Description automatically generated

**Selectors**

Selectors allows us to filter objects based on Labels.

Graphical user interface, application, Teams

Description automatically generated

**Practicle Examples Using Kubectl**

$ kubectl run nginx1 –image=nginx

$ kubectl run nginx2 –image=nginx

$ kubectl get pods

$ kubectl label pods <podName> env=dev

$ kubectl label pods <podName> env=prod

$ kubectl get pods –show-labels

$ kubectl get pods -l env=dev #Display all the pods which has labelled dev

$ kubectl get pods -l env!=prod

**Practicle Examples Using Kubectl YAML file**

apiVersion: v1

kind: Pod

metadata:

name: nginxwebserver

labels:

env: dev

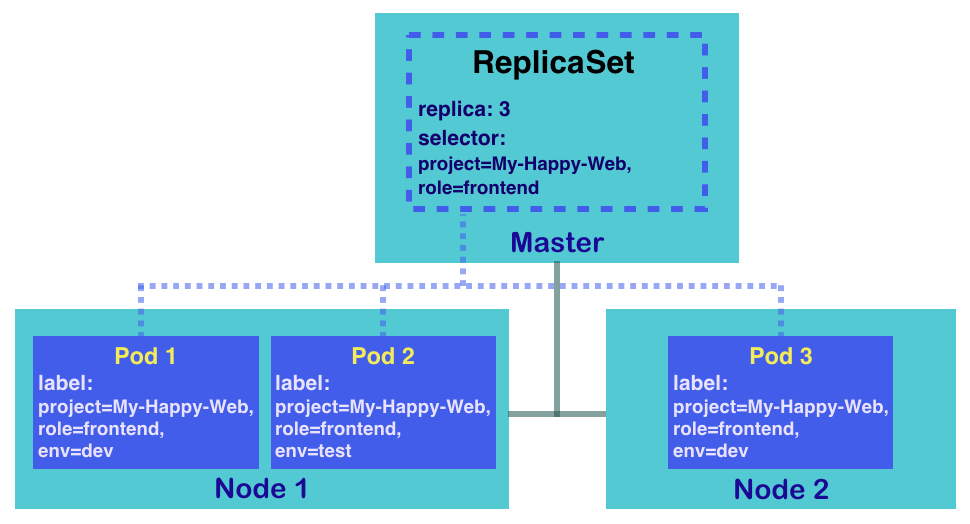
$ kubectl apply -f labels.yaml

## **Replication Controller [Old] | Replica Set [New]**

A ReplicaSet purpose is to maintain a stable set of replica Pods running at any given time.

Desired State -  The state of pods which is desired.

Current State - The actual state of pods that are running.



**Creating Our First ReplicaSets**

Replication controller helps to run multiple Instance of Single POD in a Kubernetes cluster. This provides high availability.  
 **High Availability**  
  
Even if you have single POD replication controller helps by automatically bring up new POD when existing one is failed.

It ensures that specified no. of POD running always at all time.

**Load Balancing && Scaling**  
 Another reason we need replication controller is to create multiple pods to share the load across them.

When no. of users increase we deploy additional pod to balance the load across the two pods. If the demand further increases and if we could deploy additional parts across the other nodes in the cluster.

Here **Replication controller spans across multiple nodes in the cluster**, It helps us balance the load across multiple pods on different nodes as well as scale our application when the demand increases.

Replication controller is the older tech that is being replaced by replicas set up. Replica set is the new recommended way to set up.

**Replica Set Manifest File**

Graphical user interface, text, application, email

Description automatically generated

**Create ReplicaSet**  
  
$ kubectl apply/create -f replicaset.yaml

$ kubectl get replicaset or $ kubectl get rs

$ kubectl delete replicaset ReplicaName

**Scaling the POD**

* # update the configuration file :: modify the replicas value : 6
* $ kubectl replace -f replicasetName.yaml #updated replicaset conf
* Another approach  
    
  $ kubectl scale –replicas=6 -f replicasetName.yaml or
* $ kubectl scale –replica=6 replicaset ReplicaSetName

**Updating the Replicaset**#Edit config file and save && Delete the existing pod so that it creates new pod with new configuration

**$ kubectl edit rs rsName**

$ kubectl replace -f replica.yaml

## **Deployments**

ReplicaSets works well in basic functionality like managing pods, scaling pods and similar.

Deployments provide replication functionality with the help of ReplicaSets, along with various additional capability like **rolling out of changes, rollback changes if required.**

Diagram

Description automatically generated

**Deployment Manifest**A picture containing table

Description automatically generated

**Create Deployment**

$ kubectl apply/create -f deployment.yaml

$ kubectl get deployments

$ kubectl get replicaset or # kubectl get rs

$ kubectl get pods

$ kubectl get all

$ kubectl create deployment httpd-frontend --image=httpd:2.4-alpine --replicas=3 --dry-run=client -o yaml > deploy.yaml

## **NameSpace**

* Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.
* Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all. Start using namespaces when you need the features they provide.
* Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. Namespaces cannot be nested inside one another and each Kubernetes resource can only be in one namespace.
* Namespaces are a way to divide cluster resources between multiple users (via [resource quota](https://kubernetes.io/docs/concepts/policy/resource-quotas/)).

**Kubernetes starts with four initial namespaces:**

* **default** The default namespace for objects with no other namespace
* **kube-system** The namespace for objects created by the Kubernetes system
* **kube-public** This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.
* **kube-node-lease** This namespace holds [Lease](https://kubernetes.io/docs/reference/kubernetes-api/cluster-resources/lease-v1/) objects associated with each node. Node leases allow the kubelet to send [heartbeats](https://kubernetes.io/docs/concepts/architecture/nodes/#heartbeats) so that the control plane can detect node failure

**NamesSpaces && DNS**

When you create a [Service](https://kubernetes.io/docs/concepts/services-networking/service/), it creates a corresponding [DNS entry](https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/).

This entry is of the form <service-name>.<namespace-name>.svc.cluster.local,

which means that if a container only uses **<service-name>,** it will resolve to the service which is local to a namespace.

**Command**

$ kubectl get pods #Display pods in default namespace.

$ kubectl get pods –ns=kube-system

$ kubectl get pods –all-namespace

* apiVersion: v1
* kind: Namespace
* metada:
  + name: dev

$ kubectl create -f name.yaml #Just creates the namespacekube

$ kubectl create namespace dev # Just creates the namespace

Along with POD definitaion file you can mention the namespace in metadata.

$ kubectl create -f pod-def.yaml or

$ kubectl create -f pod-def.yaml –namespace=dev # Creates in dev namespace

## **Services**

* Services helps us to connect application together with other applications.
* Kubernetes Service can act as an abstraction which can provide a single IP address and DNS through which pods can be accessed.
* Endpoints track the IP address of the objects that service can send traffic to.
* This layer of abstrcction allows us to perform lot of operations like load balancing, scaling of pods and others.

A picture containing diagram

Description automatically generated

Diagram

Description automatically generated**There are several types of Services which are available**

* **NodePort**
* **ClusterIP**
* **LoadBalancer**
* **ExternalName**

**Create Service and EndPoints**

* Create Service objects with **service port details && Target port details**.
* Internally it has Enpoinds configuration where we need to mention exact IP address of the target.
* Create **Endpoints** with service Name && Define target IP address. [Manual]
* You can use selector and pick the pod label so that IP address taken automatically.

**Port vs TargetPort**

Diagram, schematic

Description automatically generated

**NodePort [Good for Dev]**

Graphical user interface

Description automatically generated

* **It Exposes the POD to the outside world**
* From the name, we can identify that it has to do with opening a port on the nodes.
* If the service type is NodePort, then Kubernetes will allocate a port (default: 30000-32767) on every worker node.
* Each node will proxy that port into your service.

Graphical user interface, text

Description automatically generated

* + Port filed Is mandatory, If targetport not specified then it takes the same port as port value.
  + If nodeport value not defined then I takes any def port range.
  + Under Selectors :: Provide list of labels to identify the pod, pull that info from pod metadata labels

Selectors :

App: myapp

Type: front-end

**ClusterIP**

* **It Exposes a set of PODs to other objects in the cluster not outside world. Means One set of object can able to communicate to other objects inside the cluster.**
* **These services can be connected to Ingress service to get access to the outside world**
* Whenever the service type is ClusterIP, an internal cluster IP address is assigned to the service.
* Since an internal cluster IP is assigned, it can only be reachable from within the cluster.

**This is a default ServiceType.**

Diagram

Description automatically generated

Text

Description automatically generated

**Type: LoadBalancer**

Challenges with NodePort

We know that NodePort ServiceType will assign a node in all the worker node which can forward the traffic to the underlying service.

Challenge in NodePort:  We need to access it via IP/DNS:Port  
You might end up with dif IP with dif port to access the service by the client

Example:  google.com:31514

**LoadBalancer Service Type** will automatically deploy an external load balancer.

This load balancer takes care of routing requests to the underlying service.

Diagram

Description automatically generated

* The overall implementation of LoadBalancer depends on your Cloud Provider.
* If you plan to use it in bare-metal, then you will have to provide your own load balancer implementation.

## **Imperative Vs Declarative**

Imperative : From start to End have to provide the Direction.

* Provision a VM by the Name “Web-services”
* Install Nginx Software on it.
* Edit Conf file to use the port 8080
* Edit the conf file to web path /var/www/nginx
* Load web pages to /var/www/nginx from GIT Repo -X
* Start Nginx server

If objects already exist then you will error.

**Create Object**

$ kubectl run nginx –image=nginx

$ kubectl create deployment nginx –image=nginx

$ kubectl expose deployment nginx –port 80 # creating service to expose deployment

**Update Object**

$ kubectl edit deployment nginx

Edit will temporarlity changes in the config file and update. [ original file will be same]

$ kubectl scale deployment nginx –replicas=5

$ kubectl set image deployment nginx nginx=nginx:1.8 #Updating the image

$ kubectl create -f nginx.yaml

$ kubetl replace -f nginx.yaml

First Edit the local config file && then run replace command to update the objects.

$ kubectl delete -f nginx.yaml

**Declarative** : Provide Just Start location && End location. It decides the direction.

VM Name: web-server

Package: nginx

Port: 8080

Path : /var/www/nginx

Code: Git Repo -X

Create Objects:

$ kubectl apply -f nginx.yaml

$ kubectl appy -f /path/to/config # Directory so it creates all objects

Update the Object

Edit the config file && run

$ kubectl apply -f nginx.yaml #Even oject already exist it doesn’t thri error, it will updated the objects