**Kubernetes** or k8s (built by google)

**Containers:** A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

**Container orchestration:** Automatically deploying and managing containers. Automatically scaling up and scaling down.

Kubernetes is a container orchestration technology. (docker swarm, mesos, etc.)

**Architecture**

Nodes (minions): A machine physical or virtual on which Kubernetes is installed.

Cluster: set of nodes group together. multiple nodes helps in sharing load as well.

Master: responsible for managing the cluster, info about members of the cluster stores, manage workload.

**Components**: api server, etcd, kubelet, container runtime, controller, scheduler.

Api server : acts as frontend.

etcd: distributed reliable key-value store. Stores data used to manage the cluster.

**Kubectl**: kube command line tool.

**Minikube**: Normally we have a worker node and a master node but minikube bundles the components of both into a single image providing a single node kubernetes cluster. The bundle is packaged into an ISO image and available online. Require virtual box or hyper v.

Kubeadm can be used to setup multi node kubernetes cluster on local environment.

**Pod:** When the load increases on a node, we create another instance of application into a separate pod instead of creating the instance on the same pod.

We can create helper container in the same pod. (They share the same network and storage space and can refer each other by localhost).

kubectl run nginx –image nginx

kubectl get pods –o wide

**Yaml in k8s**

pod-definition.yml

apiVersion: ----- String

POD – v1, Service – v1, ReplicaSet – apps/v1, Deployment – apps/v1

kind: Pod --------- String

metadata: ---------- dictionary

name: myapp-pod

labels:

app: myapp

spec: ----------- dictionary

containers: ----list/array

* name: nginx-container

image: nginx

**Replication controller:**

Helps us run multiple instances of a single pod in the k8s cluster thus provides high availability.

Even we have a single pod, replication controller helps in bringing up the new pod when the existing one fails. Thus, it helps in maintaining the no. of pods running all the time.

Also helps in Load balancing and scaling across multiple nodes.

Replication controller vs Replica set

Replication controller is the older technology, which is replaced by replica set.

we specify a pod template under spec which is used to create replicas.

under template we can use the yaml file that is used to create the pod.

apiVersion: v1

kind: ReplicationController

metadata:

spec:

template:

metadata:

name:

labels:

app:

type:

spec:

containers:

replicas: 3

kubectl create –f rc-definition.yaml

kubectl get replicationcontroller

**Replica Set**

apiVersion: apps/v1

kind: ReplicaSet

Major difference is replica set require a selector definiation.

selector helps to identify what pod falls under it.

There may be pods that are already created before replica set creation

replica set can be used to monitor the existing pods based on the labels. Labels acts as filters among the pods.

spec:

selector:

matchLabels:

tire: front-end

**Scale the application in replicaset**

1. We can also scale the application by changing the replicas in yaml file.

kubectl replace –f replicaset-definition.yaml

1. kubectl scale --replicas=6 -f replicaset-definition.yaml (does not modify the file).
2. kubectl scale --replicas=6 replicaset myapp-replicaset

type name

**\*\*Note:** let say currently we have no. of pods running same as replicas in replica set. Now if we try to create a new pod with the same label as we mentioned in the replica set. It will automatically be deleted and replica set does not allow to do that.

kubectl edit replicaset myapp-replicaset

**Deployments**

How we deploy our app in production environment, perform update.

(GOOGLE) Create or modify instances of the pods that hold a containerized application. Deployments can scale the number of replica pods, enable rollout of updated code in a controlled manner, or roll back to an earlier deployment version if necessary.

There is not much difference in replica set and deployment but there are many use cases for deployment.

(GOOGLE) Deployment is an object which can own ReplicaSets and update them and their Pods via declarative, server-side rolling updates. While ReplicaSets can be used independently, today they're mainly used by Deployments as a mechanism to orchestrate Pod creation, deletion and updates.

**update and rollback**

when we first create a deployment, it triggers a rollout. rollout creates a new deployment revision. In future when application is updated to a new version a new rollout is triggered and new deployment revision is created. This helps us to keep track of the changes we have made and we can rollback to previous versions.

kubectl rollout status deployment/myapp-deployment

kubectl rollout history deployment/myapp-deployment

**Deployment strategy:**

recreate: taking down all pods and recreating the new pods.

rolling update: taking down one pods and creating one pod and so on.

We can do this by changing the version in the yaml file.

kubectl apply –f deployment-definition.yaml

kubectl set image deployment/myapp-deployment nginx=nginx:1.9.1

Note: when we update the deployment, deployment object creates a new replica set and starts deploying the containers there. at the same time taking down the pods in the old replica set following a rolling update strategy.

Rollback: kubernetes allow you to rollback to previous revision.

kubectl rollout undo deployment/myapp-deployment

**Kubernetes Networking**

node has an ip address. this ip is used to access the kubernetes node

Each pod has an internal IP address (10.244.\*.\*). Each pod is connected to an internal network.

Kubernetes does not setup any kind of networking. If we implement a network, then we can manage the IP assigned to pods on different nodes.

**Services**

Enable communication b/w different components within and outside an application. It helps us connect different applications together and users.

Services enables loose coupling b/w microservices in our application.

(Without services) We can access the application internally within a node but we cannot access the application outside the node because it is in a separate network.

We need something in the middle to help us map request from our laptop through the node to the pod running the web container.

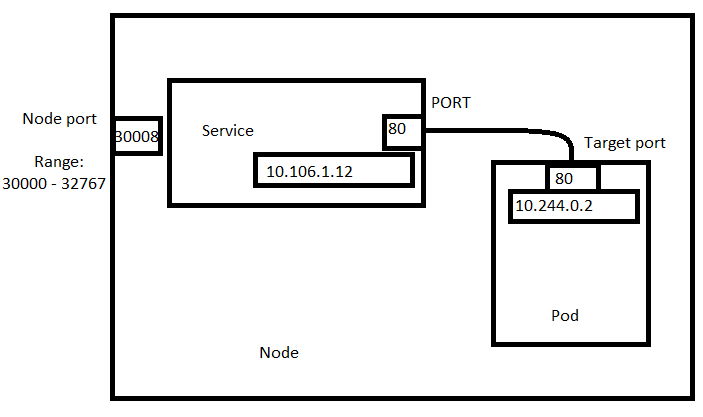
Service is an object just like deployment.

**Use case:** listen to a port on the node and forward request to a port on the pod running the web application. This type of service is node port service because the service listens to a port on the node and forwards request to the pods.

Service types:

1. Node port: service makes internal pod accessible on a port on the node.
2. Cluster IP: service creates a virtual IP inside the cluster to enable communication b/w different services such as a set of front end servers to a set of backend servers.
3. load balancer: it provisions a load balancer for our application in supported cloud providers.

Range of Node port: 30000 – 32767



Service-definition.yaml

apiVersion: v1

kind: Service

metadata:

name: myapp-service

spec:

type: NodePort

ports:

* targetPort: 80

port: 80

nodePort: 30008

selector:

app: myapp

tier: front-end

In case of multiple pods in a single node, all have the same labels as we specified in the selector. Therefore, the service will load balance among all the pods. Service can forward the request to any of the pod. It uses a random algorithm for load balancing.

In case of multiple pods in multiple nodes with different IP addresses. k8 creates a service that spans across all the nodes in the cluster and maps the target port on the same node port for all the nodes in the cluster. We can access the application using IP address of any node.

minikube service myapp-service –url

**kubectl expose pod redis --port=6379 --name redis-service --dry-run=client -o yaml**

(This will automatically use the pod's labels as selectors)

**kubectl create service clusterip redis --tcp=6379:6379 --dry-run=client -o yaml**

(This will not use the pods labels as selectors, instead it will assume selectors as app=redis. You cannot pass in selectors as an option. So it does not work very well if your pod has a different label set. So generate the file and modify the selectors before creating the service)

**Kubernetes Certified Application Developer (CKAD)**

**Namespaces:**

Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.

1. We can assign different policies and resources to different namespaces. Example: dev and production namespaces.
2. Objects in a namespace can refer each other simply by their names. To refer object in other namespace, we can use full dns name. example: **“db-service.dev.svc.cluster.local”**.

when a service is created a dns entry is created automatically in this format.

cluster.local = default domain name of k8s cluster.

svc = subdomain for service

dev = namespace

db-service = service name

1. **kubectl get pods --namespace=kube-system**

when a pod is created using yaml file, the pod is created in default namespace.

**kubectl create –f pod-def.yaml --namespace=dev**

we can also specify the namespace in yaml file.

**metadata:**

**namespace: dev**

1. create a new namespace.

**apiVersion: v1**

**kind: Namespace**

**metadata:**

**name: dev**

**kubectl create –f namespace.yaml**

or

**kubectl create namespace dev**

1. we can also switch to another namespace permanently.

**kubectl config set-context $(kubectl config current-context) --namespace=dev**

This command first identify the current context and then set the namespace to the desired one for the current context.

1. **kubectl get pods --all-namespaces**
2. we can also limit resource in a namespace using resource quota.

**compute-qouta.yaml**

**apiVersion: v1**

**kind: ResourceQuota**

**metadata:**

**name: compute-quota**

**namespace: dev**

**spec:**

**hard:**

**pods: “10”**

**requests.cpu: “4”**

**requests.memory: “5Gi”**

**limits.cpu: “10”**

**limits.memory: “10Gi”**

**kubectl create –f compute-quota.yaml**

**Imperative commands:**

**--dry-run**: If you simply want to test your command, use the --dry-run=client option. This will not create the resource. Instead, tell you whether the resource can be created and if your command is right.

**-o yaml**: This will output the resource definition in YAML format on the screen.

**Docker and kubernetes configuration**

Dockerfile

FROM ubuntu

ENTRYPOINT [“sleep”]

CMD [“5”]

pod.yaml

spec:

containers:

* name:

image:

command: [“sleep2.0”]

args: [“10”]

ENTRYPOINT is same as command in pod yaml file.

CMD is same as args in pod yaml file.

we can override the fields using this.

**Set Environment Variables**

containers:

* name :

image:

env:

* name: APP\_COLOR

value: pink

we can also set the environment variable using config maps and secret.

**ConfigMaps**

Two ways of creating the config maps:

1. Imperative

kubectl create configmap <config-name> --from-literal=<key>=<value>

kubectl create configmap app-config –from-literal=APP\_COLOR=blue

--from-literal=APP\_MODE=prod

we can also create config map from a file

kubectl create configmap <config-name> --from-file=<path-to-file>

kubectl create configmap app-config –from-file=app\_config.properties

1. Declarative

config-map.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: app-config

data:

APP\_COLOR: blue

APP\_MODE: prod

kubectl create –f config-map.yaml

kubectl get configmaps

**Inject config map in pods**

envFrom:

* configMapRef:

name: app-config

env:

* name: APP\_COLOR

valueFrom:

configMapKeyRef:

name: app-config

key: APP\_COLOR

volumes:

* name: app-config-volume

configMap:

name: app-config

**Secrets**

They are similar to configmaps except that they store data in encrypted format.

1. Creating a secret
2. imperative

kubectl create secret generic <secret-name> --from-literal=<key>=<value>

kubectl create secret generic <secret-name> --from-file=app\_secret.properties

1. declarative

apiVersion: v1

kind: Secret

metadata:

name: app-secret

data:

DB\_Host: mysql

DB\_User: root

We need to specify the data in hashed format.

echo –n ‘mysql’ | base64

echo –n ‘mysql’ | base64 --decode

kubectl create –f secret.yaml

1. Inject into pod

envFrom:

* secretRef:

name: app-secret

env:

* name: DB\_Password

valueFrom:

secretkeyRef:

name: app-secret

key: DB\_Password

**Docker Security**

A host has no. of processes running on it. Example: Docker daemon, SSH server, etc.

Unlike virtual machine, containers are not completely isolated from its host. Containers and host share the same kernel.

Containers are isolated using namespaces in Linux. Host and container both have their own namespace.

A container can see its own processes only, it cannot see anything outside of it or in any other namespace.

When we list the processes on the host, we see the processes including the docker container processes with different namespaces.

**docker run --user=1000 ubuntu sleep 3600**

We can specify that in the docker image at the time of creation.

**Dockerfile**

**FROM Ubuntu**

**USER 1000**

docker build –t my-ubuntu-image .

docker run my-ubuntu-image sleep 3600 (Now process will run with user id 1000)

If the root user inside the container is same as root user on the host and process on the container can do anything that root user on the host can do, then it will be dangerous.

Docker limits the abilities of the root user within the container. Docker uses Linux Capabilities to implement this.

Root user has all the permission on the kernel.

To provide privileges, we use --cap-add command.

**docker run --cap-add MAC\_USER ubuntu**

We can drop privileges using --cap-drop option.

**docker run --cap-drop KILL ubuntu**

For all privileges, use --privileged flag.

**docker run --privileged ubuntu**

**Security Context**

If we configure the security setting to the pod, then it will be set to all the containers in the pod.

Settings on the container will override the settings on the pod.

spec:

containers:

* name: ubuntu

securityContext:

runAsUser: 1000

capabilities:

add: [“MAC\_ADMIN”]

Note: We can specify securityContext under spec as well as under the containers.

To check user on the pod running the process inside the container.

kubectl exec [pod-name] -- [command]

kubectl exec ubuntu-sleeper – whoami

**Service Accounts**

There are two types of accounts: **user account, service account**

When you (a human) access the cluster (for example, using kubectl), you are authenticated by the apiserver as a particular User Account (currently this is usually admin, unless your cluster administrator has customized your cluster). Processes in containers inside pods can also contact the apiserver. When they do, they are authenticated as a particular Service Account (for example, default).

In order for application to query the kube-api server, it has to be authenticated and for that, we use a service account.

**kubectl create serviceaccount dashboard-sa**

**kubectl get serviceaccount**

**kubectl describe serviceaccount dashboard-sa**

When a service account is created, a token is also created automatically. A token is used by the external application while authenticating to the k8s API.

**Tokens: dashboard-sa-token-kbbdm**

Token is stored as a secret object and secret object is then linked to a service account.

**kubectl describe secret dashboard-sa-token-kbbdm**

curl https://<IP>/api -insecure--header “Authorization: Bearer <Token>”

Each namespace has its own **default** service account.

Whenever a pod is created the default service account and its token are automatically mounted to that pod as a volume mount.

A volume is automatically created named as default token.

**kubectl exec –it my-kubernetes-dashboard ls /var/run/secrets/kubernetes.io/serviceaccount**

Default service account is very restricted and can run only few queries.

We can modify the pod yaml file to use different service account.

**spec:**

**serviceAccount: dashboard-sa**

**spec:**

**automountServiceAccountToken: false**

We can add additional permissions for the newly created 'dashboard-sa' account using RBAC.

**Resource Requirements**

Each node has a set of CPU, Memory and Disk resources available. Every pod consumes a set of resources.

Kubernetes Scheduler decides which node a pod goes to. Based on the amount of resources required by a pod and resource available on the nodes.

If resource is not available on any node, then the pod will be in the pending state. Reason: Insuffecient CPU.

**Resource request**: minimum amount of memory and CPU requested by the container.

**spec:**

**resources:**

**requests:**

**memory: “1Gi”**

**cpu: 1**

0.1 CPU = 100 m (m = mili)

1 G (Gigabyte) = 1,000,000,000 bytes

1 Gi (Gibibyte) = 1,073,741,824 bytes

1 Ki (kibibyte) = 1,024 bytes

We can limit the resource usage for a pod.

**resources:**

**limits:**

**memory: “2Gi”**

**cpu: 2**

When a pod tries to exceed resources beyond its specified limit.

In case of CPU, k8 **throttles** the CPU that it does not go beyond its limit.

In case of memory, if a pod tries to consume more memory then the pod will be **terminated**.

When a pod is created, the containers are assigned a default CPU request of .5 and memory of 256Mi.

For the POD to pick up those defaults you must have first set those as default values for request and limit by creating a LimitRange in that namespace.

apiVersion: v1

kind: LimitRange

metadata:

name: mem-limit-range

spec:

limits:

- default:

memory: 512Mi

defaultRequest:

memory: 256Mi

type: Container

apiVersion: v1

kind: LimitRange

metadata:

name: cpu-limit-range

spec:

limits:

- default:

cpu: 1

defaultRequest:

cpu: 0.5

type: Container

**Note: The status CrashLoopBackOff indicates that it is failing because the pod is out of memory. Identify the memory limit set on the POD.**

**Taints and Tolerations**

Taints are placed on the nodes and tolerations are placed on the pods.

So that only those pods will be assigned to a tainted node which can tolerate that node.

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

**taint-effect:** what will happen to the pod that do not tolerate this taint.

NoSchedule, PreferNoSchedule, NoExecute

**kubectl taint node node1 app=blue:NoSchedule**

spec:

tolerations:

* key: “app”

operator: “Equal”

value: “blue”

effect: “NoSchedule”

**Note:** There is no pod scheduled on the master node. Because when the k8 cluster is first setup, a taint is set on the master node automatically that prevents any pod from being scheduled on this node.

**Node Selectors**

spec:

nodeSelector:

size: Large

Where does this size and large comes from? These are the key value pairs based on the labels on the node.

kubectl label nodes <node-name> <label-key>=<label-value>

**Node selector limitations:**

We cannot put conditions on the selector.

Example: Put the pod in Large or medium Node, Put the pod in a node that is not small.

For this Node, affinity and anti-affinity features are used.

**Node Affinity**

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoreDuringExecution: ……….Property

nodeSelectorTerms:

* matchExpressions:
  + key: size

operator: In

values:

* + - large

**Operator**: In, NotIn, Exists

**Property:**

requiredDuringSchedulingIgnoredDuringExecution

preferredDuringSchedulingIgnoredDuringExecution

Scheduling:

1. required: pod will not be scheduled if scheduler not able to find the affinity key in the node label.
2. preferred: scheduler try its best to find the matching label, if not found then place the pod in any node.

Execution: if any of the property change during execution of the pod.

1. Ignored: pod will continue to run and will ignore any change in the node affinity property.
2. required: pod will be removed from the node if the label is deleted from the node.

**Multi Container Pods**

Same lifecycle

Same network

Same storage

Multi container design patterns

1. SIDECAR
2. ADAPTER
3. AMBASSADOR

**SIDECAR:**

**example**-

webserver with a logging agent.

webserver collects the logs and send to the central log server.

Multiple applications may generate logs in different formats. So, before sending the logs to the central server we would like to convert the logs into a common format.

For this we deploy an **ADAPTER** container. ADAPTER container processes the logs before sending it to the central server.

**AMBASSADOR**

Our Application communicates to different database instances at different stages of development.

dev, testing, production databases.

We must modify the connectivity on the code depending on the environment we are deploying our application to.

We may outsource this logic to a separate container within a pod. so that our application can always refer to a database at localhost and the new container will proxy that request to the right database.

**Observability**

**Readiness Probe**

Pod Conditions

Ready- This condition indicates that the application inside the pod is running and is ready to accept the user traffic.

When we expose the pod using service, then service relies on the pod ready condition to route traffic.

K8 assumes that as soon as the container is created, it is ready to serve user traffic. So, it sets the value of the ready condition to true. But application took longer to get ready, so we end up hitting a port that is not running any live application.

We can setup different kind of tests or probe.

1. HTTP Test - /api/ready (for api)
2. TCP Test – 3306 (for database)
3. Exec Command to run a custom script.

spec:

containers:

- name:

readinessProbe:

httpGet:

path: /api/ready

port: 8080

initialDelaySeconds: 10

periodSeconds: 5

failureThreshold: 8

tcpSocket:

port: 3306

exec:

command:

- cat

- /app/is\_ready

**Liveness Probe**

When a pod crashes, k8 attempts to restart the container repeatedly to restore service to user.

In case when application is not working due to a bug in the code but the container stay alive, the application is stuck in an infinite loop.

Liveness probe is to check if the application in the pod is actually healthy. **livenessProbe:**

for i in {1..20}; do

kubectl exec --namespace=kube-public curl -- sh -c 'test=`wget -qO- -T 2 http://webapp-service.default.svc.cluster.local:8080/ready 2>&1` && echo "$test OK" || echo "Failed"';

echo ""

**Container Logging**

To see logs, **kubectl logs –f <pod-name> <container-name> (In case of multiple containers)**

**Monitoring**

One Metric server per k8 cluster.

Metric server: In memory monitoring solution.

On each node, there is a kubelet.

Kubelet contain a sub component cAdvisor or container advisor.

cAdvisor is responsible for receiving performance metrics from pods and exposing them through the kubelet api to make metrics available for the metrics server.

**minikube addons enable metrics-server**

**kubectl top node**

kubectl top pod

**Labels, Selectors and Annotations**

kubectl get pods --selector app=app1

Annotations: phone no., email ids, build version, etc.

**Jobs**

spec:

restartPolicy: Always, Never

-----------------------------------------------------------------------

**apiVersion: batch/v1**

**kind: Job**

**metadata:**

**name: math-add-job**

**spec:**

**completions: 3**

**template:**

**spec:**

**containers:**

**- name: math-add**

**image: ubuntu**

**command: [‘expr’, ‘3’, ‘+’, ‘2’]**

**restartPolicy: Never**

**kubectl create –f job.yaml**

Job tries to create new pods until it has three successful creation.

We can also create pods in parallel using

**spec:**

**parallelism: 3**

**CronJob**

**apiVersion: batch/v1beta1**

**kind: cronJob**

**metadata:**

**name: reporting-cron-job**

**spec:**

**schedule: “\*/1 \* \* \* \*”**

**jobTemplate:**

**spec:**

**completions: 3**

**parallelism: 3**

**template:**

**spec:**

**containers:**

**- name: reporting-tool**

**image: reporting-tool**

**restartPolicy: Never**

**kubectl create –f cron-job.yaml**

**kubectl get cronjob**

**Ingress Networking**

When we setup a small application, we have some services and pods, where a node port service is exposed so that user can access it using http://<node-ip>:38080.

But in production environment there are lot more things like a dns server to point to ip of the node.

<http://my-online-store.com:38080>

Now we want user not to remember the port no. either. so we bring in additional layer b/w dns server and cluster like a proxy server, that proxy the request on port 80 to port 38080.

What if we develop another service, which is independent of this service and we want to route traffic to it using <http://my-online-store.com:38080/video> and <http://my-online-store.com:38080/wear>.

Now we configure different load balancers for different services and both have different IP addresses.

The solution for this will be to configure an ingress service which will route traffic to different services using one IP address, it require a load balancer or node port service to expose the ingress service.

**Deploy:** reverse proxy or a load balancing solution like **Nginx, HAPROXY, traefik**.

deploy them on a kubernetes cluster and configure them to route traffic.

**configure:** defining URL routes, configuring SSL certificates.

**Ingress Controller**: Solution we deploy

**Ingress Resources**: Set of rules we configure.

**Ingress Controller:**

**apiVersion: extensions/v1beta1**

**kind: Deployment**

**metadata:**

**name: nginx-ingress-controller**

**spec:**

**replicas: 1**

**selector:**

**matchLabels:**

**name: nginx-ingress**

**template:**

**metadata:**

**labels:**

**name: nginx-ingress**

**spec:**

**containers:**

**- name: nginx-ingress-controller**

**image: quay.io/kubernetes-ingress-controller/nginx-ingress-controller:0.21.0**

**args:**

**- /nginx-ingress-controller**

**- --configmap=$(POD\_NAMESPACE)/nginx-configuration**

**env:**

**- name: POD\_NAME**

**valueFrom:**

**fieldRef:**

**fieldPath: metadata.name**

**- name: POD\_NAMESPACE**

**valueFrom:**

**fieldRef:**

**fieldPath: metadata.namespace**

**ports:**

**- name: http**

**containerPort: 80**

**- name: https**

**containerPort: 443**

err-log-path

keep-alive

ssl-protocols

In order to decouple these from nginx image, we need to create a **config map**.

**kind: configMap**

**apiVersion: v1**

**metadata:**

**name: nginx-configuration**

We also need to create a **service** to expose the controller to external load.

**apiVersion: v1**

**kind: Service**

**metadata:**

**name: nginx-ingress**

**spec:**

**type: NodePort**

**ports:**

**- port: 80**

**targerPort: 80**

**protocol: TCP**

**name: http**

**- port: 443**

**targetPort: 443**

**protocol: TCP**

**name: https**

**selector:**

**name: nginx-ingress**

**we also need a service account**

**Ingress Resource:**

**ingress-wear.yaml**

**apiVersion: extension/v1beta1**

**kind: Ingress**

**metadata:**

**name: ingress-wear**

**spec: spec:**

**backend: rules:**

**serviceName: wear-service - http:**

**servicePort: 80 paths:**

If it is a single backend then there is no root. **- path: /wear**

**backend:**

**serviceName:**

**servicePort:**

**kubectl create –f ingress-wear.yaml - path: /watch**

**kubectl get ingress backend:**

**Rules: serviceName:**

Rule1: [www.my-online-store.com](http://www.my-online-store.com)

Rule2: [www.wear.my-online-store.com](http://www.wear.my-online-store.com)

Rule3: [www.watch.my-online-store.com](http://www.watch.my-online-store.com)

Rule4: Everything else

**kubectl describe ingress ingress-wear-watch**

For different domain names

**spec:**

**rules:**

**- host: wear.my-online-store.com**

**http:**

**paths:**

**- backend:**

**serviceName:**

**servicePort:**

**- host: watch.my-online-store.com**

**http:**

**paths:**

**- backend:**

**serviceName:**

**servicePort:**

**Network Policy**

When we only want to allow traffic from certain pod, it can be ingress or egress.

**apiVersion: networking.k8s.io/v1**

**kind: NetworkPolicy**

**metadata:**

**name: db-policy**

**spec:**

**podSelector:**

**matchLabels:**

**role:db**

**policyTypes:**

**- Ingress**

**- Egress**

**ingress:**

**- from:**

**- podSelector:**

**matchLabels:**

**name: api-pod**

**namespaceSelector:**

**matchLabels:**

**name: prod**

**- ipBlock:**

**cidr: 192.168.5.10/32**

**ports:**

**- protocol: TCP**

**port: 3306**

**egress:**

**- to:**

**- ipBlock:**

**cidr: 192.168.5.10/32**

**ports:**

**- protocol: TCP**

**port: 80**

This will allow ingress traffic from api-pod on 3306 port.

Note: Flannel networking solution does not support network policy.

**State Persistence**

**Volume**

**Persistent volumes**

**Persistent volume claim**

**Storage Class**

**Stateful Sets**