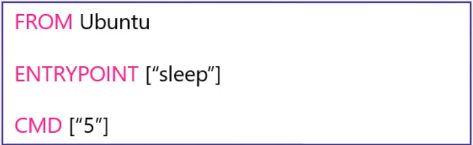
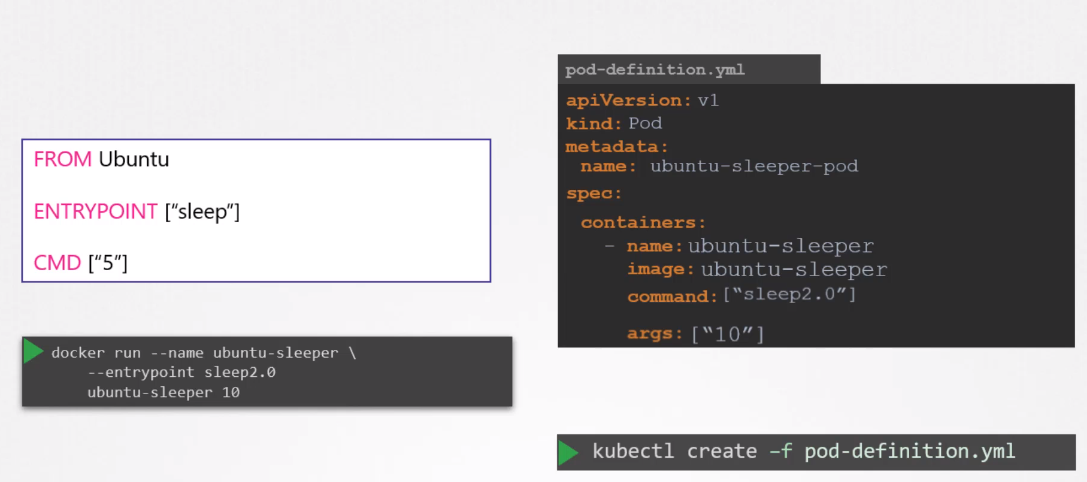
**Commands and Arguments in Docker:**

* $ docker built -t ubuntu-sleeper : to make “ubuntu-sleeper” image
* Now you can modify this image as per your need (ie. Image needs to start sleep cmd as beginning)
* 
* $ docker run ubuntu-sleeper 10 : now “ubuntu-sleeper” image will run for 10 sec (if we do not specify 10, then by default 5 sec as mention above)

**Commands and Arguments in Kubernetes:**

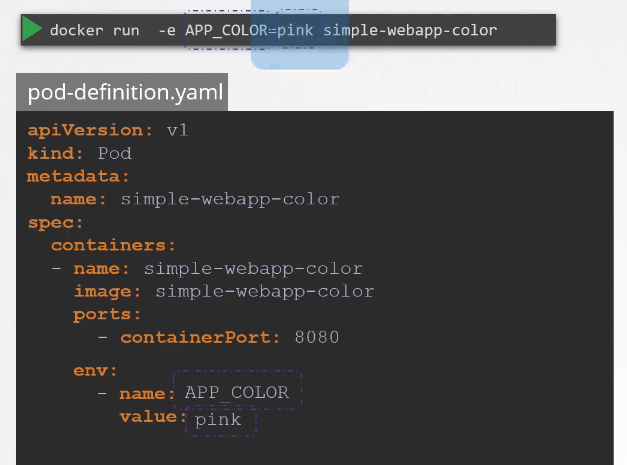
* We saw before how we can create docker image with specific args.
* Now we can create Kubernetes pods using this image (mention this image in the pod-definition.yaml file )as below.



* So in Kubernetes command overrides the docker entrypoint and Kubernetes args overrides docker CMD

|  |  |
| --- | --- |
| **Docker** | **Kubernetes** |
| Entrypoiny | Command |
| CMD | args |

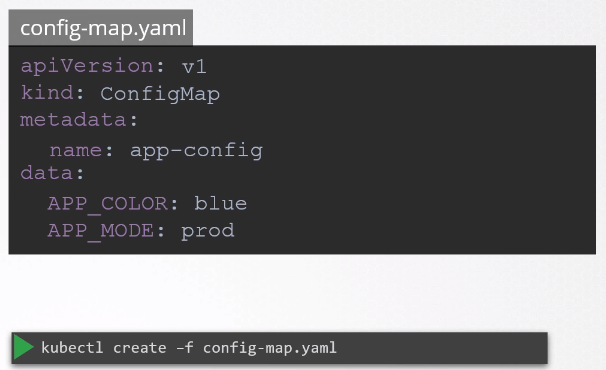
**Environment Variable:** specify “**env”** in pod-definition file as below.



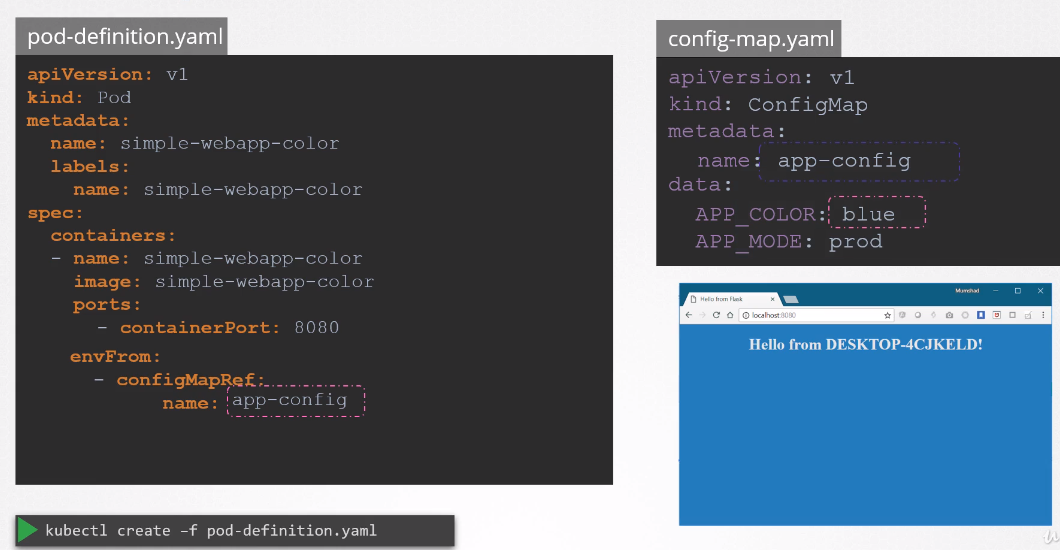
**Config Maps:**

* It becomes very hard to control all the environment variables when we have multiple pods. So we use configMaps
* You can create configMaps in two ways

1. Imperative : $ kubectl create configmap <config-name> --from-literal=<key>=<value>
2. Declarative : $ kubectl create -f config-map.yaml

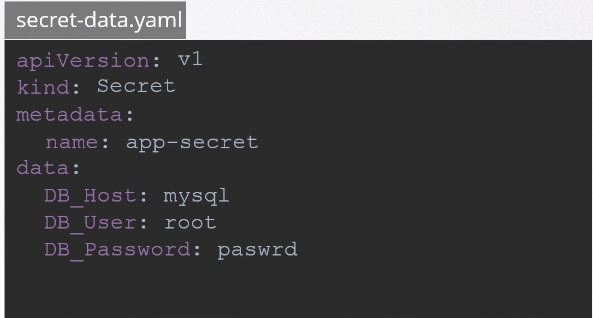
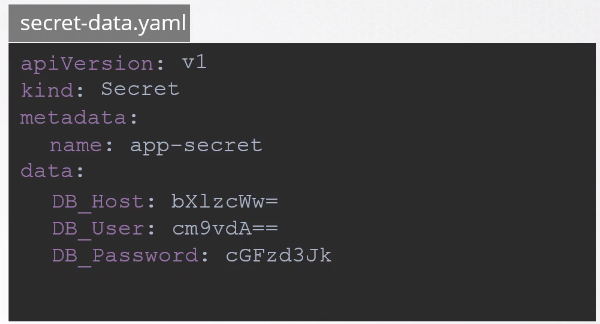


* Here is how to create a pod using this configmap



**Secrets:**

* It is used to store sensitive info (ie: credentials)
* Imperative : $ kubectl create secret generic <secret-name> --from-literal=<key>=<value>
* Declarative : $ kubectl create -f secret-data.yaml
* But here in data, you need to pass encoded values
* to encode values run this linux cmd : $ echo -n ‘mysql’ | base64
* to decode : $ echo -n ‘bxlzcWw=’ | base64 --decode

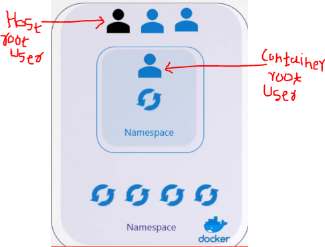
 



* to create pod using secret

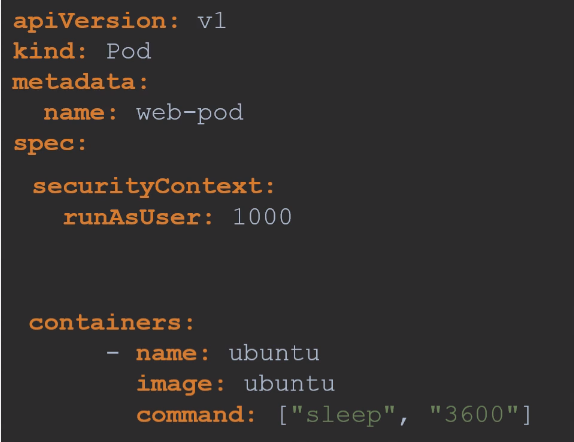


**Docker Security:**

* Docker puts security constraines on root user of container and root user of host which allows them to access only certain info.
* $ docker run --cap-add MAC\_ADMIN ubuntu : to run ubuntu image and provide MAC\_ADMIN access to user.

**Kubernetes Security:**

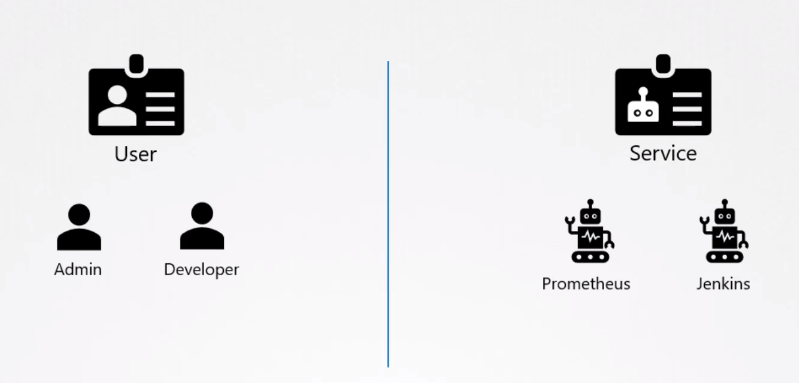
* The access can be configure in kubernetes as well
* You can configure this security settings at POD level, which will be apply to all the containers running inside of the POD

applying security at container level applying security at POD level

**Service Account :**

* User accounts are used by humans and service accounts are used by machines.
* Service accounts allows to do rest api.
* When you create a service account, by default it will store the token inside the secret.
* Each namespace has default serviceaccount, and when a new POD is created the token of default serviceaccount gets mounted on the POD automatically.



* $ kubectl create serviceaccount dashboard-sa : to create a service account named dashboard-sa

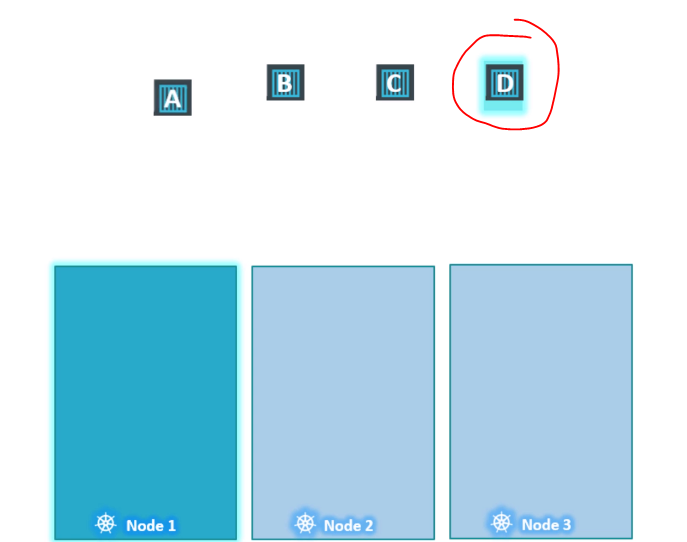
**Resource Requirement :**

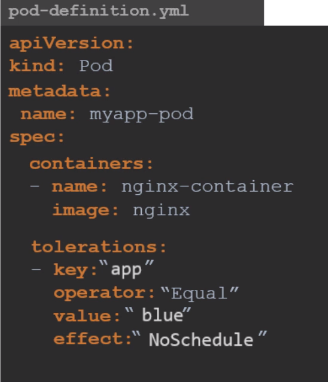
* When a POD gets deploy on NODE, it uses available resource of NODE (ie CPU, Memory, disk space)
* Scheduler is the one who decides which pod to be release on which node as per the POD’s resource requirement
* We can define these resources in pod-definition file.
* Under Resources, we can do “request” and also can put “limits”



**Taints & Tolerations:**

* Taints & Tolerations are only meant to restrict Nodes from accepting certain Pods
* Taint = Bug Repellent spray & Toleration = Bug’s tolerance power.
* Taint is applied at Nodes and Toleration is applied at Pods.



* In above scenario only Pod “D” can access the Node “1”
* Node:
* $ kubectl taint nodes “node-name” key=value:taint-effect : to add taint on node
* taint-effect is of three kind : NoSchedule | PreferNoSchedule | NoExecute
* $ kubectl taint nodes “node-name” key=value:taint-effect- : to remove taint from node (put minus sign in the end)
* Pods:
* 

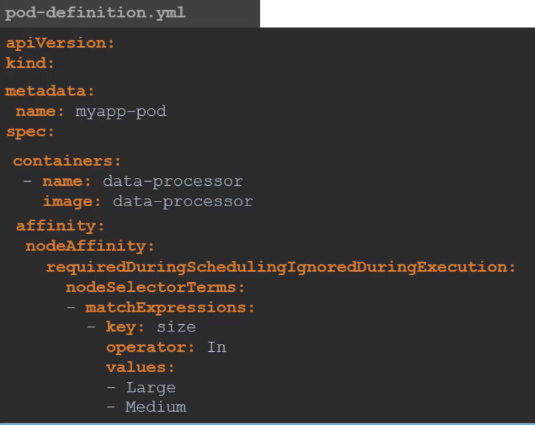
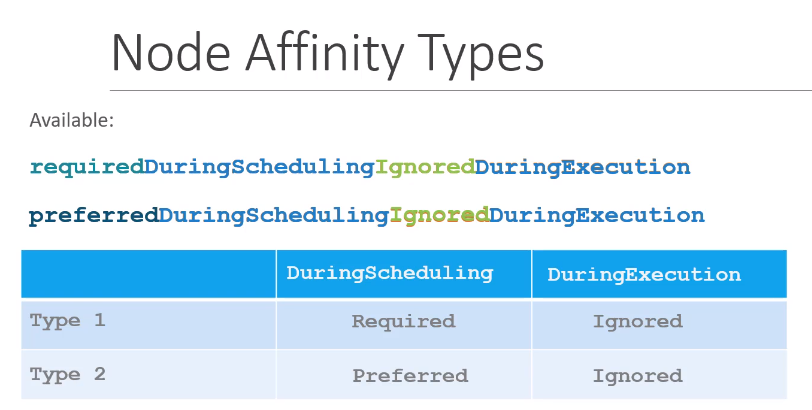
**Node-Selectors:**

* We can specify that particular pods (which requires higher horsepower for task to execute) deploys only on certain Nodes (with larger size).
* In order to do this first label the Node (key value pair) and then use that labels inside on pode-definition file to deploy that pod on specified node.
* $ kubectl label nodes “node-name” “key=value”
* $ kubectl create -f pod-definition.yaml



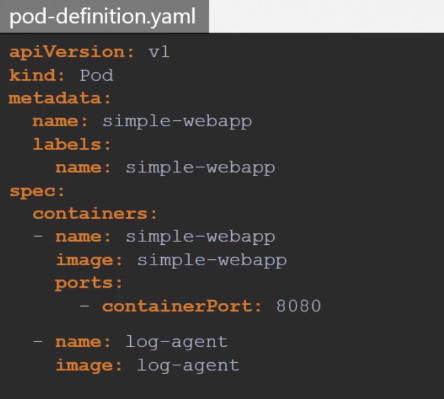
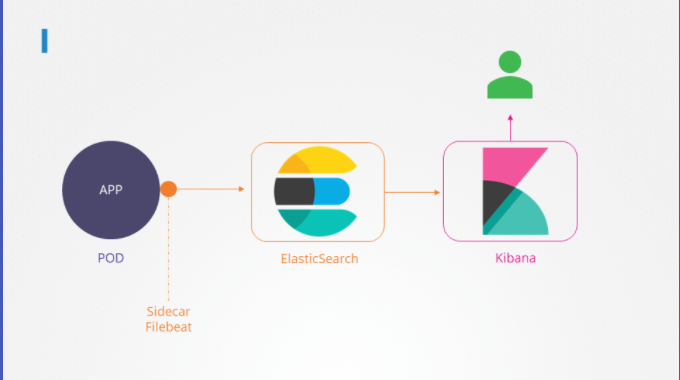
**Node Affinity:**

* Limitation of NodeSelector = you cannot use advance operators (ie: OR , NOT)
* It can be overcome by use of Node Affinity

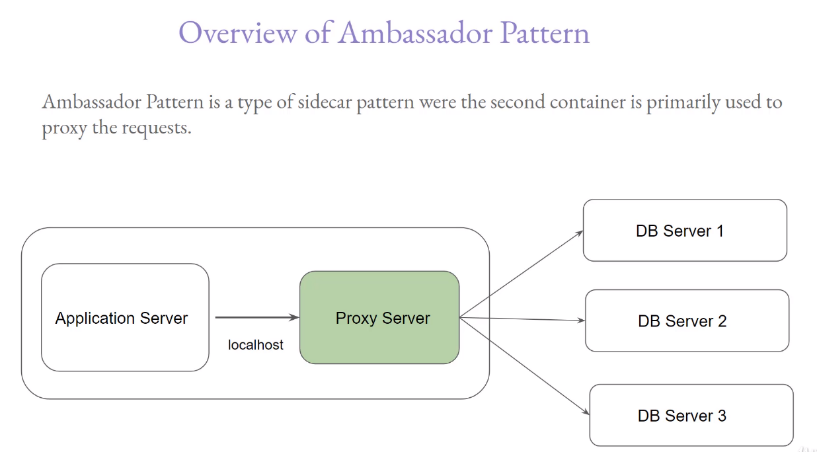
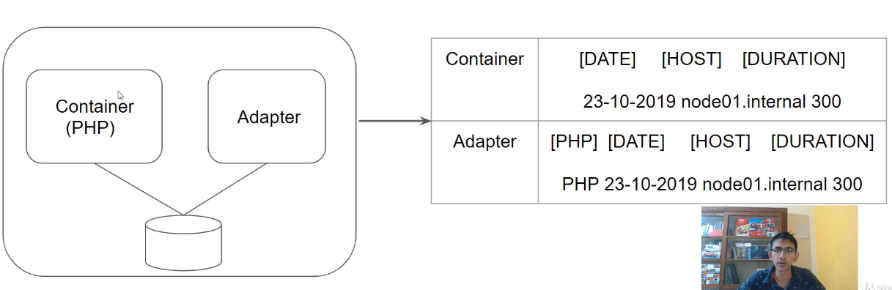
 

**Multi Container Pods:**

* Here Pod holds more than a one containers > it makes it easy to share services, Volume, Network etc..
* Add this container configs in the pod-definition file (under spec)
* example: we can add sidecar container in our pod which directs all the logs to ES

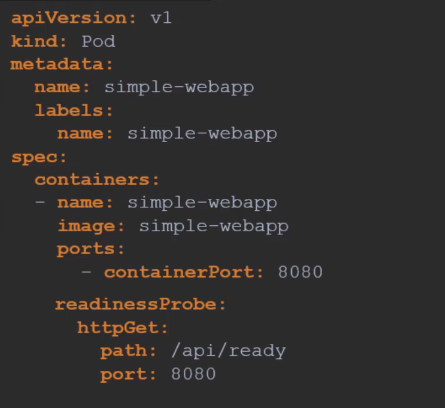
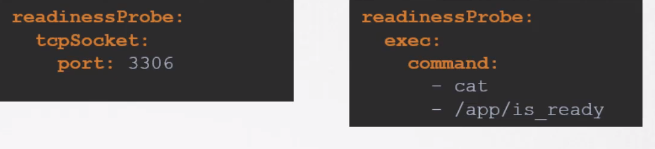
Three types :

1. sidecar container
2. Ambassador patern : Main conatiner + proxy server
3. Adapter Patern : main container + data x-form container (log data x-form to standarize it)

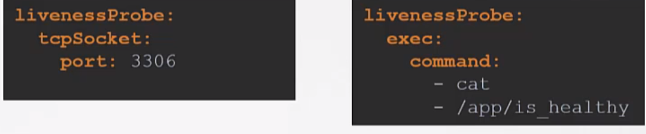
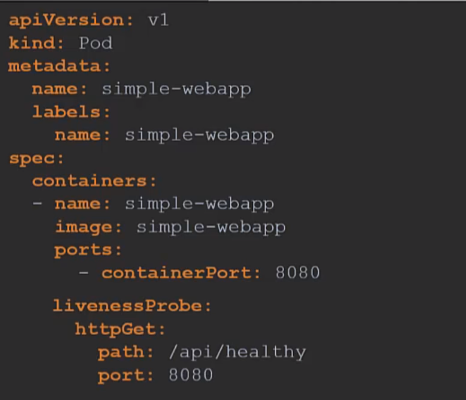
Ambassador Adapter

**Readiness Probes:**

* There are three states of Pod : Pending > Container Creating > Running
* We want to make sure that the container inside the pod is ready to serve the traffic before directing any traffic towards it. Way to do this is use readiness probe in pod-definition file as below.
* It has three types : 1) http , 2) tcpsocket , 3) exec



**Liveness Probes:**

* In docker if container application stops working then container will get deleted (cause it does not have orchestration ability)
* But if application inside Kubernetes pod gets down, Kubernetes will try to run one more pod and on and on…
* But what if application has bug in code and it goes into the malfunction (not stopped), so since application is not stopped Kubernetes wont restart or spin up a new the pod. In this scenario we need something to monitor the Liveness of the application, so Liveness Probe comes handy..

**Container Logging :**

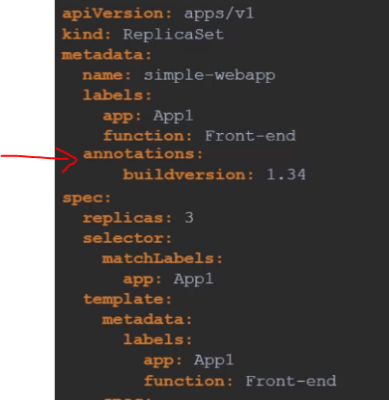
* $ kubectl logs -f “pod-name” : to view the logs of pod
* $ kubectl logs -f “pod-name” -c “container-name” : incase it has multiple container running in same pod

**Monitoring and debug application:**

* We can use third party application such as ELK , Prometheus etc. to do monitoring
* To enable metrics server on Kubernetes (it is monitoring tool of Kubernetes—Not used widely)
* git clone <https://github.com/kubernetes-incubator/metrics-server.git> : to clone the repo
* kubectl create -f deploy/1.8+/ : to create performance metrics
* kubectl top node : to see CPU , memory consumption of each node
* kubectl top pods : to see CPU , memory consumption of each pods

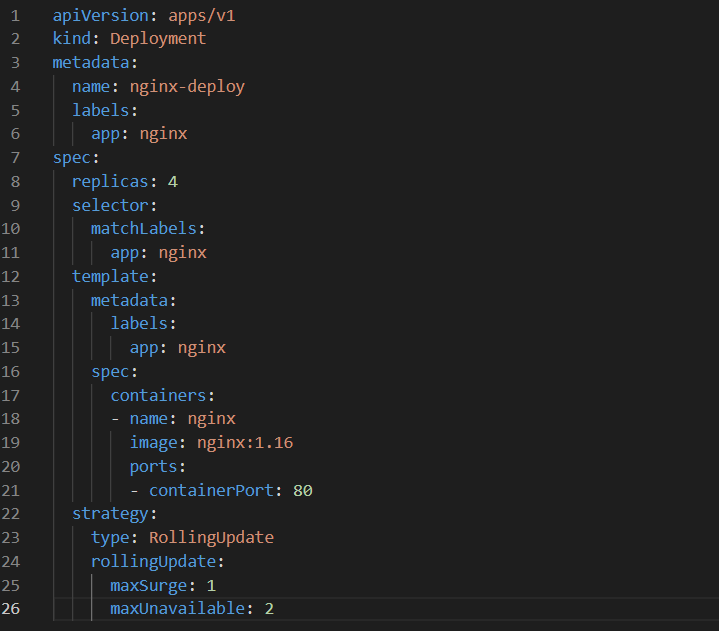
**POD Design:**

**Labels,selector,annotation :**

* To group or select objects based on certain criteria we need labels and selectors in kubernetes.
* Pode-definition.yaml > metadata > labels > key=value pair
* $ kubectl get pods --selector app=App1 : to get pods with specific labels
* Example of the how to use annotation :

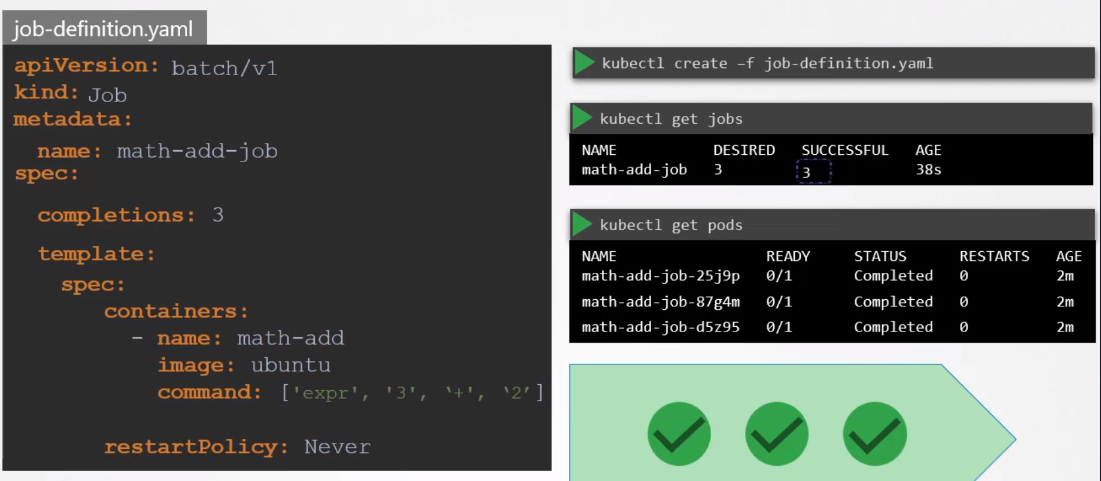
**Rolling Updates & Rollbacks :**

* $ kubectl rollout status “deployment-name”
* $ kubectl rollout history “deployment-name”
* $ kubectl rollout undo deployment/myapp-deployment : incase if we want to roll back to the previous version.
* $ kubectl create -f deployment-definition.yaml --record : when you put --record it will store the change cause which can be seen afterwards using rollout history cmd.



**Jobs:**

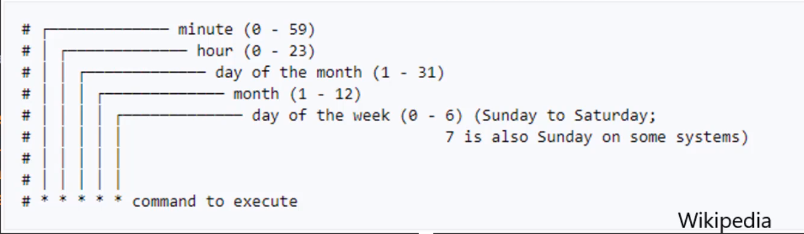
* A Job is used to run set of pods to perform a given task.



* Completion defines how many pods do we need for this job
* And as usual under template copy and paste pod-definition spec

**CronJobs:**

* Cronjobs can run on specified schedules.

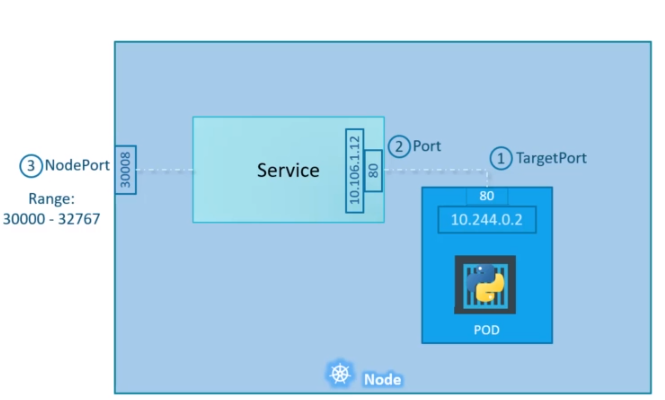
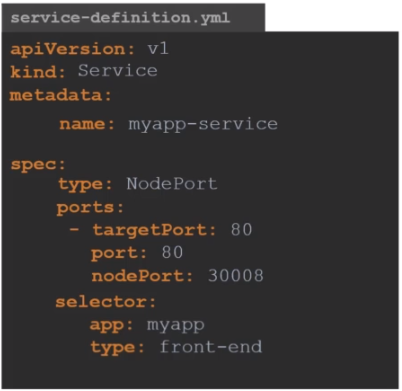


* Here under jobTemplate > copy and paste spec of Job
* So the first spec is of cronjob, second spec is of Jobs, and third spec is of Pod

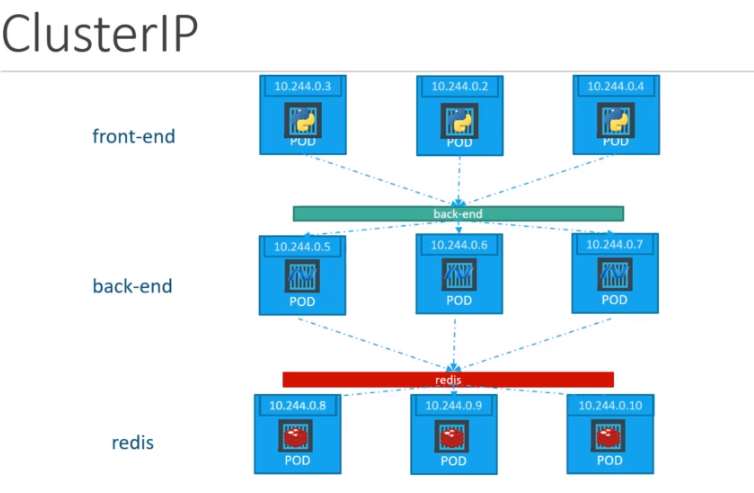
**Service and Networking:**

**Services :** “**service-definition.yaml”**

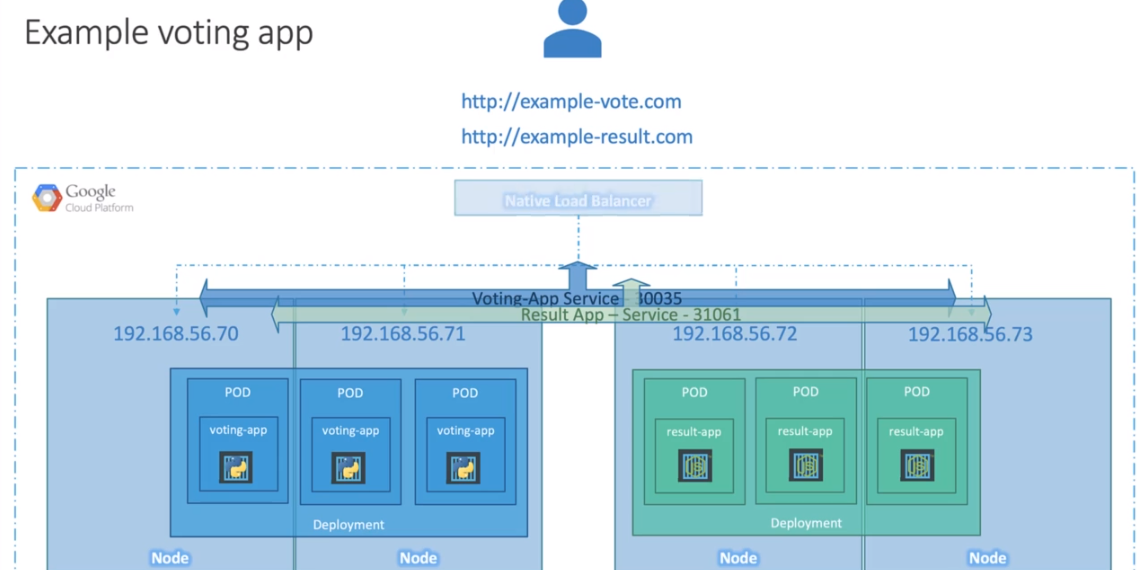
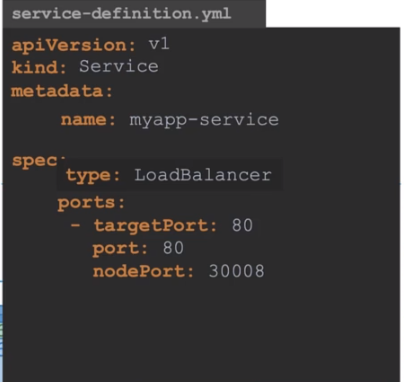
* It enables the connectivity b/w sets of pods.
* It Helps enabling communication b/w frontend application and Users.
* It helps enabling communication b/w frontend and backend pods.
* **NodePort service:**

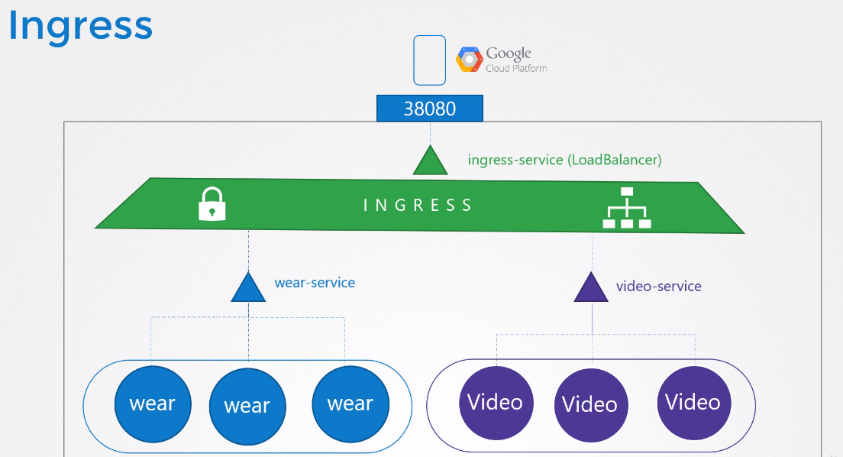
* $ kubectl create -f service-definition.yaml
* $ kubectl get services
* $ kubectl describe services myapp-service
* **Cluster IP:**

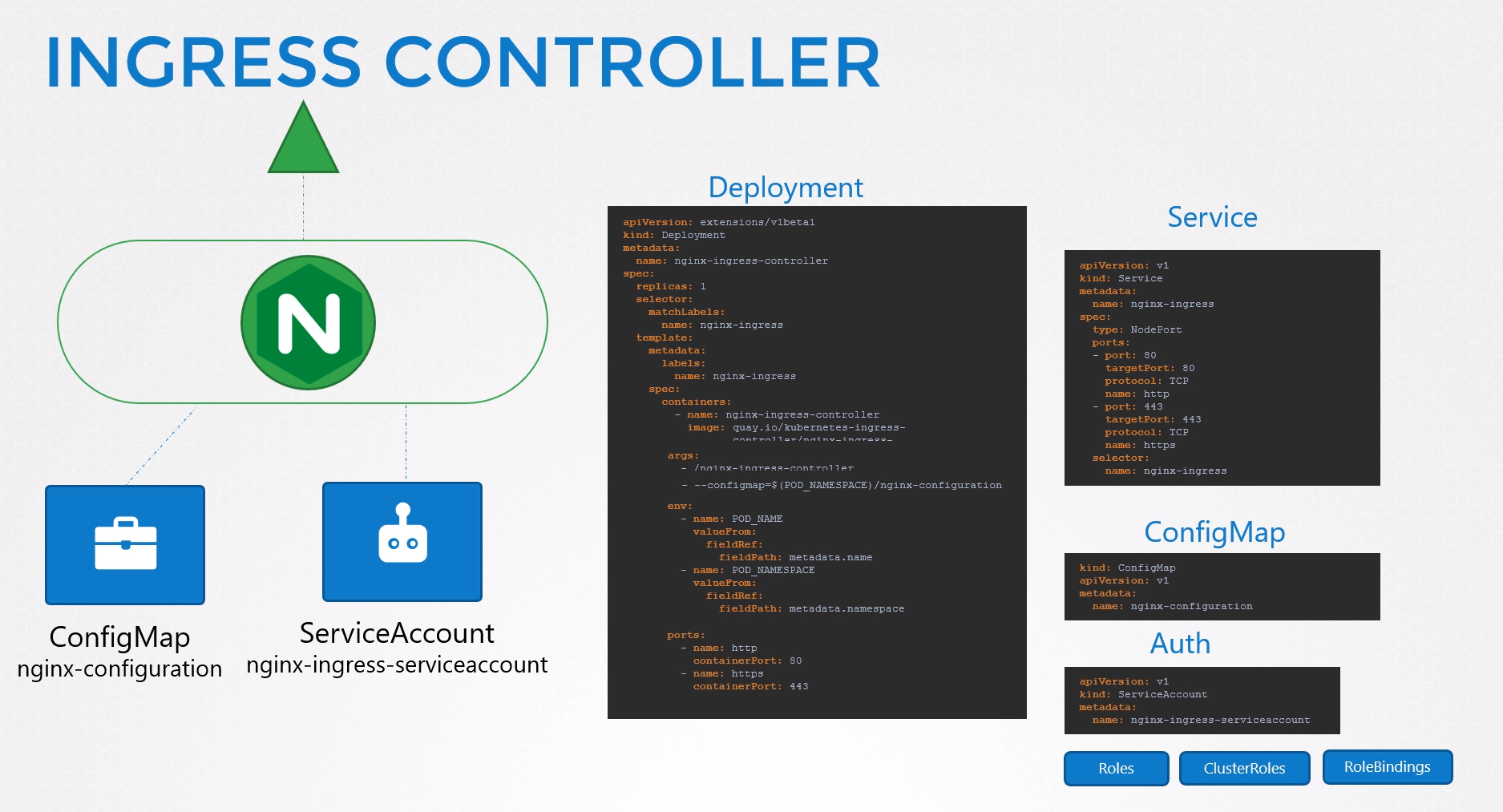


* **Load-Balancer:**

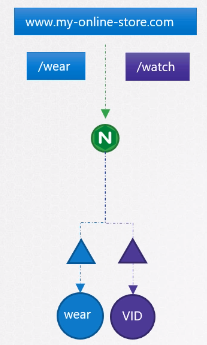


**Ingress Networking:**

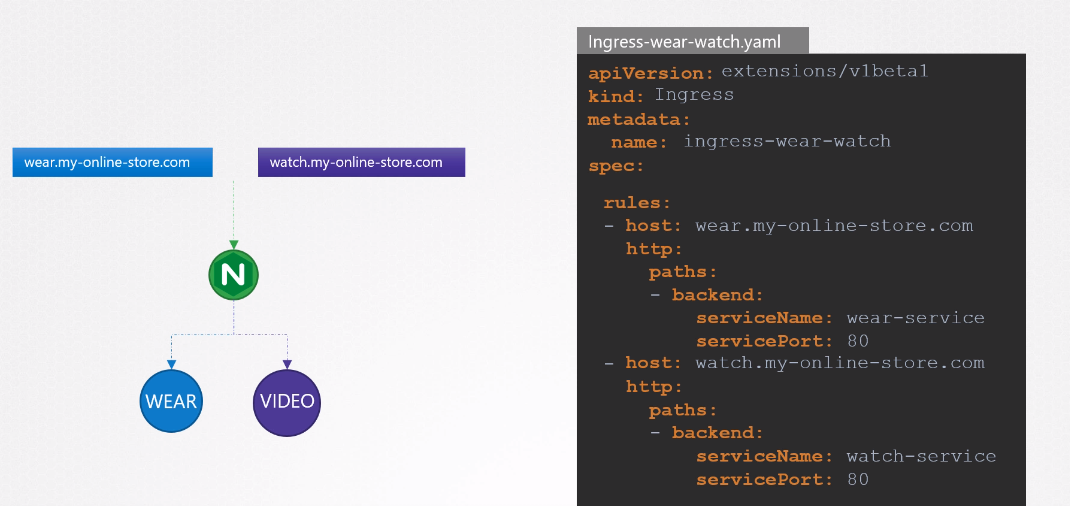
* Ingress helps your customer to access your application using single URL (DNS name) , which you can configure to route difference services within your cluster based on the URL path.
* It also adds SSL security (/https)
* Simply just think it as build in load balancer
* Triangle = Services
* Circle = Pods
* Long circles = Deployments
* **Ingress Controller :**
* Solution for reverse proxy is called ingress controller (which uses NGINX or HAPROXY or TRAFFIK internally)
* Kubernetes do not provide it by default, so you must deploy it first.
* Definition.yaml file you will need =
* ingress deployment : NGINX ingress image to deploy
* Service : to expose this deployment to outside users
* Config-map : to feed NGINX config data
* serviceaccount : with right permission to access all of the above objects.



* **Ingress resources :**
* Set of rules and configuration applied on ingress controller are called ingress resources
* They are created using definition.yaml file (same as pod,services,deployments …)
* $ Kubectl create -f ingress-wear.yaml
* $ kubectl get ingress
* ingress-wear.yaml with 1 DNS name and 2 paths (/watch or /wear)

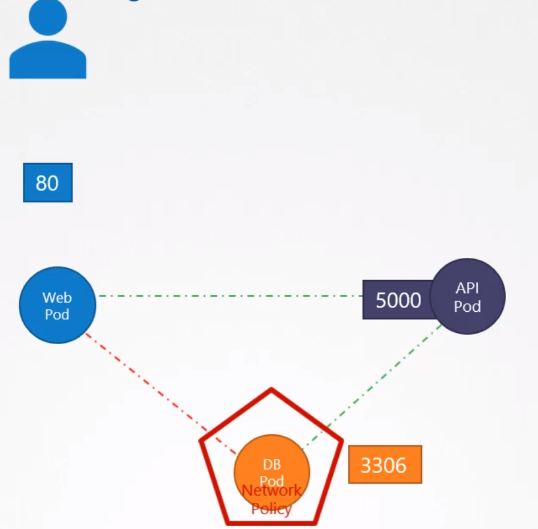
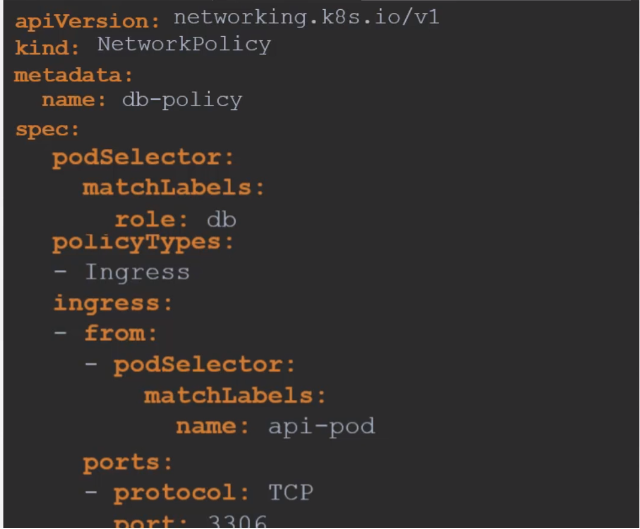
 

* ingress-wear.yaml with 2 DNS name and 1 path (path matches the DNS name)



**Network Policies: ”policy-definition.yaml”**

* By default each Pods can talk with any other pods in Kubernetes cluster.
* But what if we don’t wont communication b/w certain pods. For i.e : we do not want our front-end pod to talk with database pod directly , rathe it should talk like this frontend pod > API pod > DB pod
* This can be achieved by Network Policies.

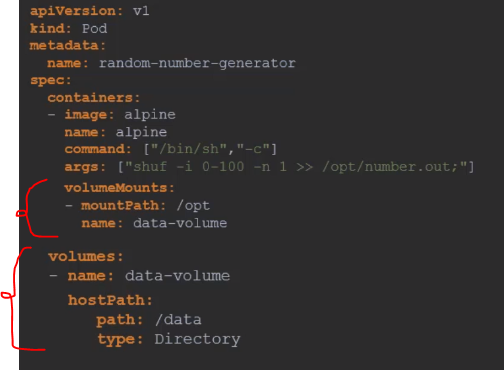
 

* $ kubectl create -f policy-definition.yaml

**State Persistence**

**Volume:**

* In docker data only last as long as the data processing process last. Once the process is done > data gets destroyed
* To have persistence data, we have attach volume with it. So even if container is deleted the data stays in volume.
* Same in Kubernetes, attach volume with pod, so even pods gets deleted the data remains.
* You need to mount this volume with pod.



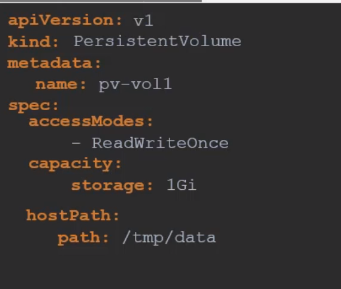
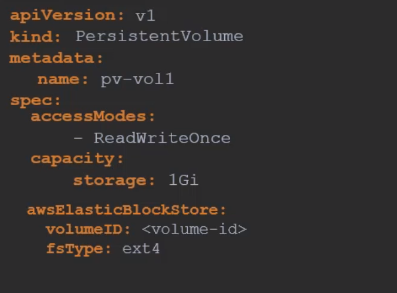


To use AWS EBS volume



**Persistance Volumes**  **“pv-definition.yaml”**

* It is a pool of volume.
* Every time user deploys a Pod , he has to configure the volume in pod-definition file > which gets tideous with large environment. > instead you can manage volume centrally. > so an administrator creates a large pool of volume and every user can use a piece of it as per requirement.

**Persistence-volume claim:**

* Administrator makes persistence volume (PV) and User makes persistence volume claim (PVC)
* Then Kubernetes binds PV and PVC based on the properties .

