Kubernetes Application Développer Training

Nassim Niclas Youssef | 3/25/2023 – Présent | Frisange (Domicile)- Luxembourg

Contents

PODS

Definition

Pods are the basic execution units. In Pods we define what will be deployed and runed from containers, images, applications. Pod basically have IP address, storage, shared data, memory and CPU.

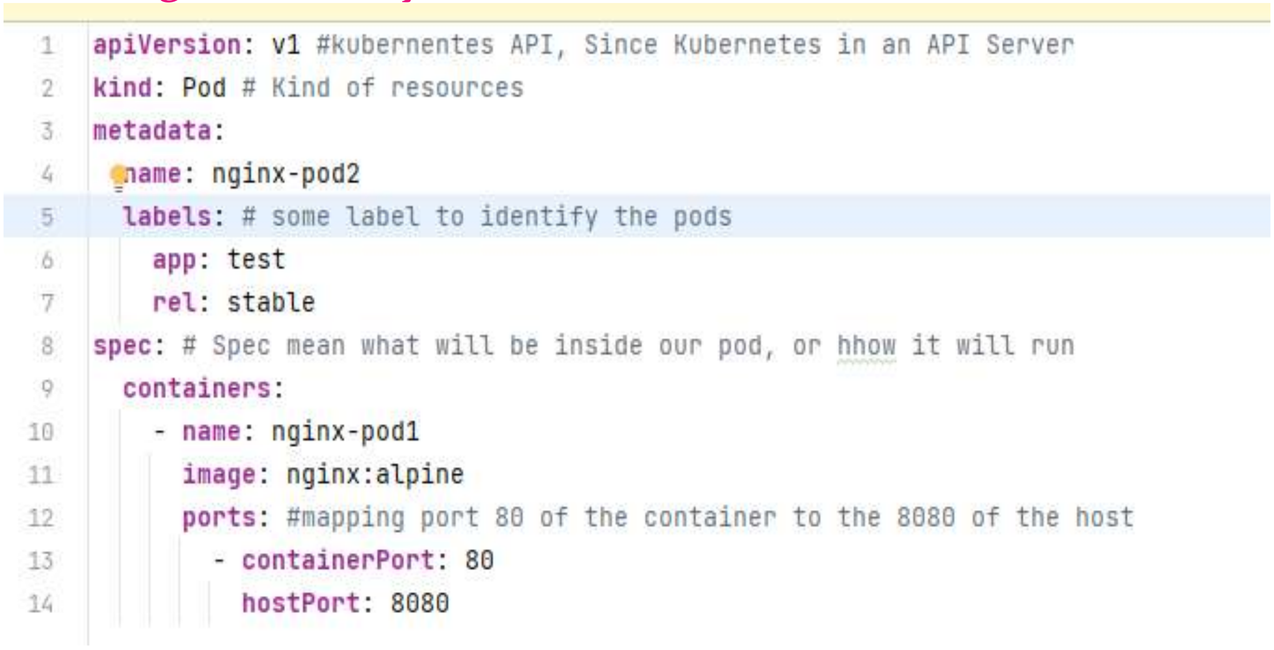
Pods have unique IP addresses and each container on a pod has its own port because multiple containers on a POD cannot use the same PORT.

Kubernetes have the option to create a unhealthy pod/or unlive and we call it replication

Creating Pod from CLI

To create a Pod we do kubectl run <pod-name> --image=<image-name>

Creating Pod from yml



```yaml
apiVersion: v1 #kubernentes API, Since Kubernetes in an API Server
kind: Pod # Kind of resources
metadata:
 name: nginx-pod2
  labels: # some label to identify the pods
    app: test
    rel: stable
spec: # Spec mean what will be inside our pod, or hhow it will run
  containers:
    - name: nginx-pod1
      image: nginx:alpine
      ports: #mapping port 80 of the container to the 8080 of the host
        - containerPort: 80
          hostPort: 8080
```

We need to create a yml file like the above

Then we need to issue a kubectl apply -f <path of the above yml file> or

Kubectl create -f filename.yml –dry-run –validate=true.

P.S we use --save-config on the create in order to track the configs of a specific pod that actually is running, this can be useful when we need to update the pod

Deleting a pod

Kubectl delete <pod-name>. Be aware a pod caused by a replication or a deployment maybe recreated or kubectl delete -f yml.file

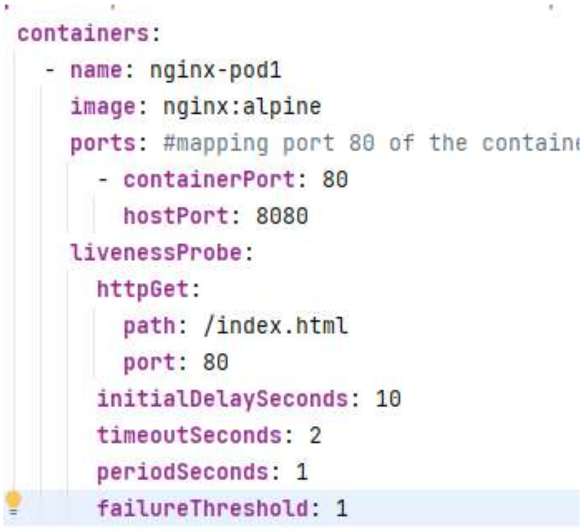
Networking and accessing

Each pod will be created with an IP address that is only accessible from within the cluster and not into the public, so only containers can access each other. In order to access pod IP we should use: kubectl port-forward <pod-name> public-port:pods-port

Accessing a pod internal kernel

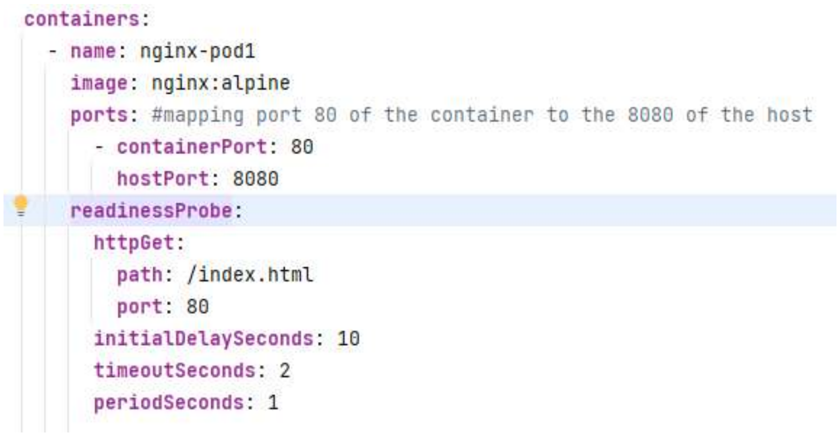
We simply type kubectl exec -it <pod-name> sh

Health Check: Probes



```
containers:
  - name: nginx-pod1
    image: nginx:alpine
    ports: #mapping port 80 of the containe
      - containerPort: 80
        hostPort: 8080
    livenessProbe:
      httpGet:
        path: /index.html
        port: 80
      initialDelaySeconds: 10
      timeoutSeconds: 2
      periodSeconds: 1
      failureThreshold: 1
```

The above is an image of the liveness of a specific container. Liveness mean that we are putting the condition under which is a prob is considered to be alive



```
containers:
  - name: nginx-pod1
    image: nginx:alpine
    ports: #mapping port 80 of the container to the 8080 of the host
      - containerPort: 80
        hostPort: 8080
    readinessProbe:
      httpGet:
        path: /index.html
        port: 80
      initialDelaySeconds: 10
      timeoutSeconds: 2
      periodSeconds: 1
```

Same as liveness, readiness is used to check when a prob is ready. It can be useful when we need to block traffic while the pod is running its container.

periodSeconds: the repetition period of the prob

timeOut: when we should end our probing

initialDelaySeconds: delay to launch our probing

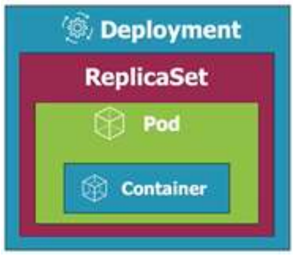
failureFreshHold: allowed number of failing, after it the pod will be considered dead (RIP)

P.S: httpGet is only a probing way with the objective of invoking an http url and expecting a status of 200, however we may choose other option (for more info check the kubernetes official docs

Deployment and ReplicatSet

Definition

ReplicatSet are simply a declarative way to manage pods by distributing/replicating them across nodes. ReplicatSet actually take care of re-creating a POD that is being destroyed or stopped for some reason. Deployment is simply a way to manage ReplicatSet.



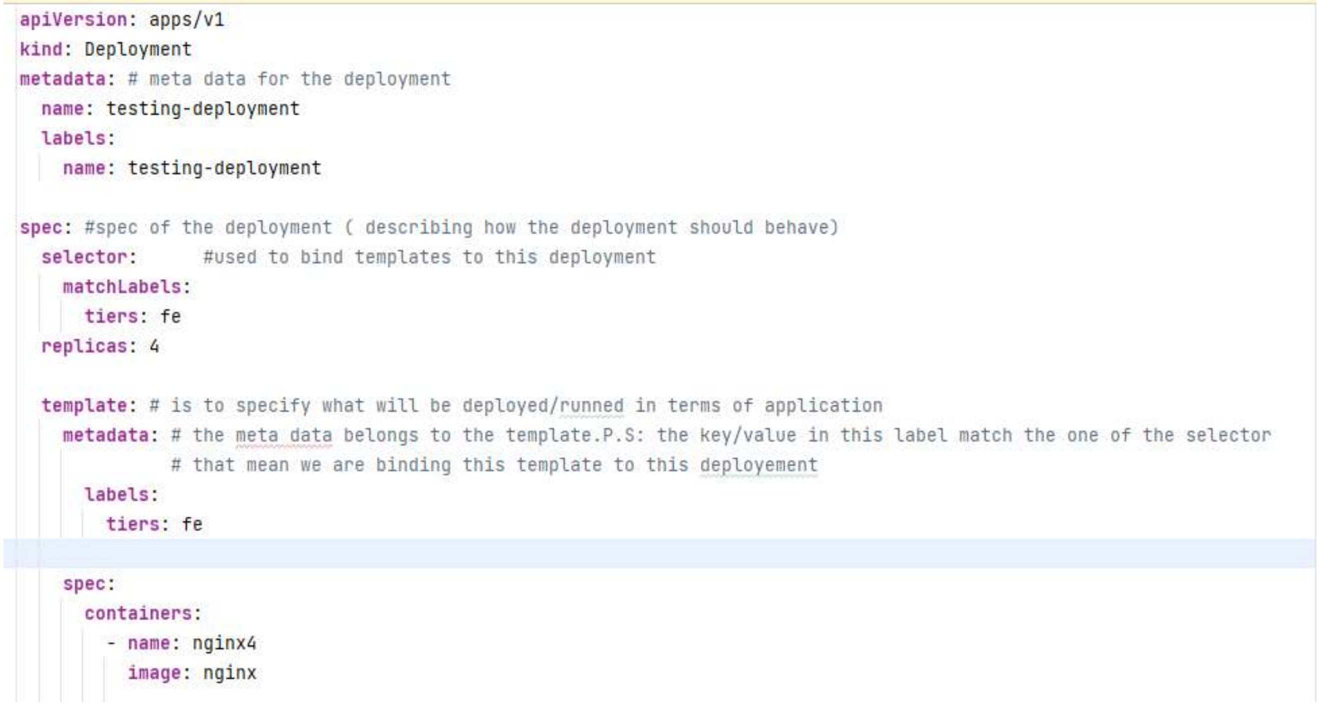
Role of ReplicatSet

1- Self-healing
2- Make sure we have the required number of pod running
3- Fault tolerance
4- Scaling
5- Relies on POD template

Role of Deployment

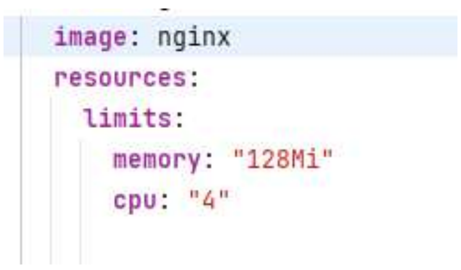
1- Management of replicatSet
2- Scaling
3- Rollback versions

Creating Deployment



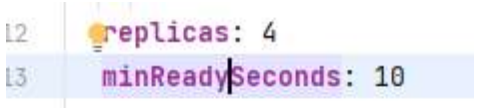
```yaml
apiVersion: apps/v1
kind: Deployment
metadata: # meta data for the deployment
  name: testing-deployment
  labels:
    name: testing-deployment

spec: #spec of the deployment ( describing how the deployment should behave)
  selector:      #used to bind templates to this deployment
    matchLabels:
      tiers: fe
  replicas: 4

  template: # is to specify what will be deployed/runned in terms of application
    metadata: # the meta data belongs to the template.P.S: the key/value in this label match the one of the selector
              # that mean we are binding this template to this deployement
      labels:
        tiers: fe

    spec:
      containers:
        - name: nginx4
          image: nginx
```

To simply create the deployment we run kubeclt create/apply -f filename.yml



```yaml
image: nginx
resources:
  limits:
    memory: "128Mi"
    cpu: "4"
```

The above image is to say what are the maximum resources allowed per container. When a container hits those limits it will be restarted.



```yaml
replicas: 4
minReadySeconds: 10
```

The above image specifies the time required for a pod to be fully functional

Scaling a specific deployment

Run kubectl scale -f filename.yml –replicas=<number_of_replicat>

P.S: wen we delete a replicated pod it will be re-created, so in order to remove we can delete the entire deployment or downscale

Deployment Options

1- Rolling Updates
2- Blue Green Deployment
3- Canary Deployment
4- RollBack

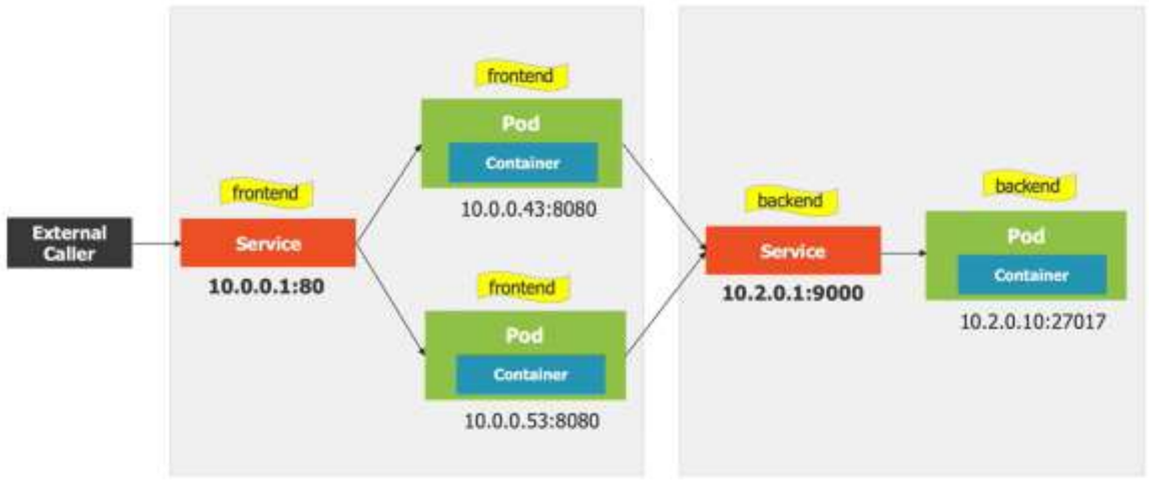
Zero Downtime

Basically a deployment will update versions of applications, and while doing this kubernetes will not terminate all the old versions instead it will manage the termination of some while creating the new version replicas. When all the new version replicas are deployed all of our old version will be terminated and by this kubernetes will assure zero downtime. Simply said we cannot terminate old versions before having fully operational new versions

Service

Definition

Service are considered to be single entry point for multiple pods. Its like an API gateway.its usefull when pods gets destroyed and re-created, in this case the service will be able to recognize new pods and to access them we only need a single entry point. Meaning the service is able to recognize a pod/container by its name rather then by it's IP and provide a route to it. Even multiple pods can rely on a service for internal communication and **those pods can communicate with each other via their names rather then their IP address**

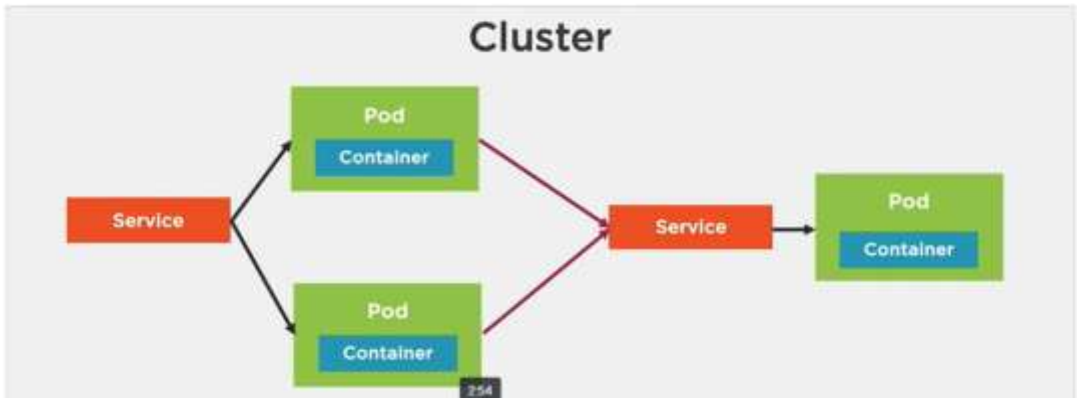


Service Type

ClusterIp Service

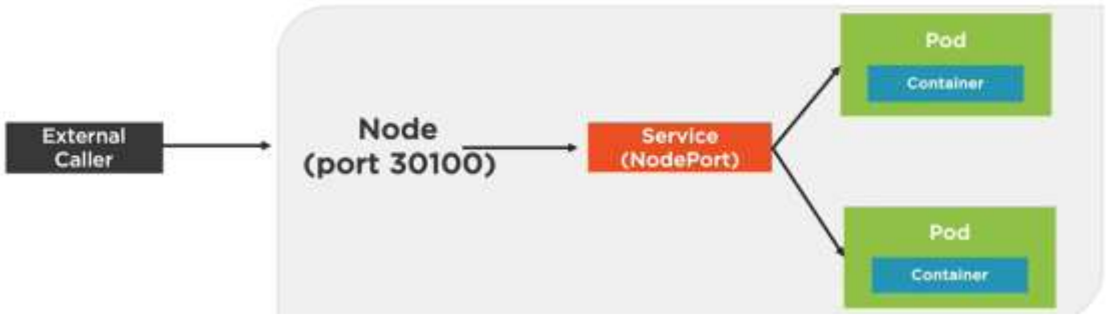
Service IP is exposed internally, so communication is restricted to the pods inside the clusters.

Only the pods inside the cluster can communicate to the service. The service also allows pods to communicate with each other even if those pods belong to 2 different service. Basically the service here is like routing point. Here the service can be used as a inter-service loadbalancing so basically when we need to speak to a pod outside the service we should pass by the service. P.S:



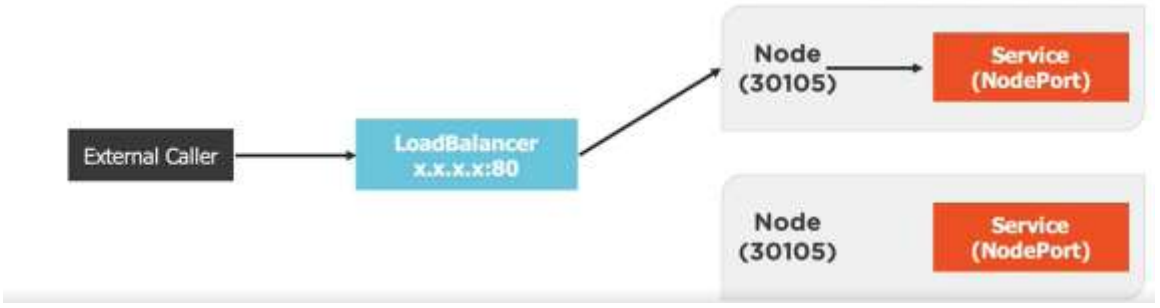
NodePort

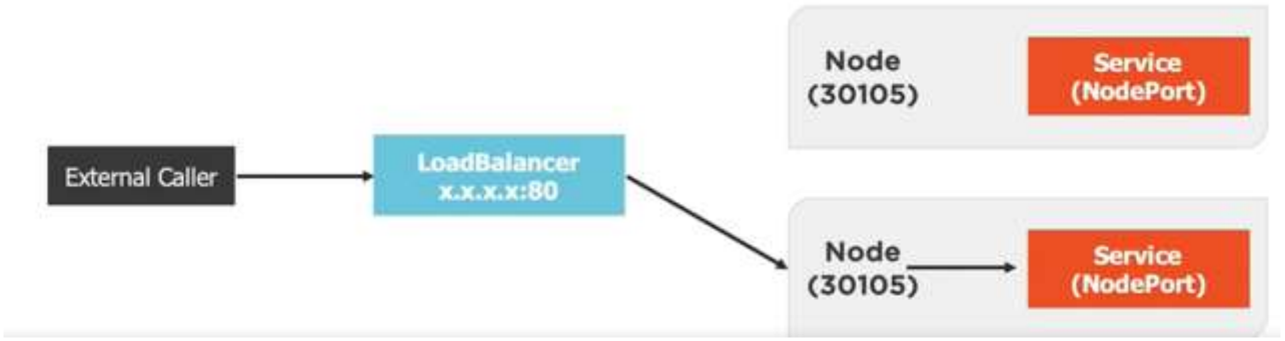
In NodePort we are exposing the pods and their port to their node ip with a random generated port



Load Balancer

Its role to balance the server charges. In this mode only the service IP is exposed to the public and its up to the service to do the re-routing based on load balancing algorithm





External Name

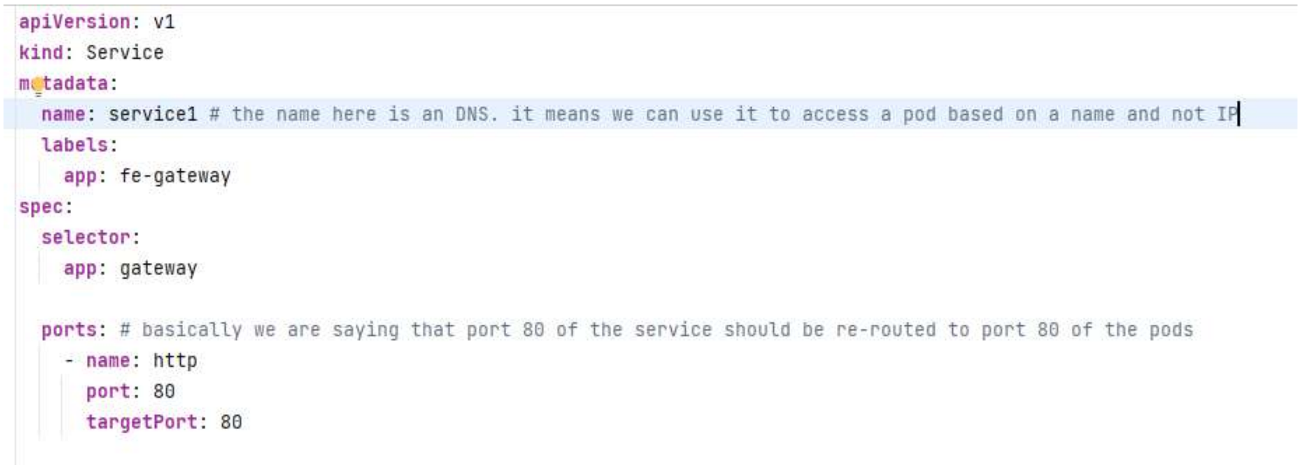
Simply it act as DNS for service. So instead of accessing a service by ip we can access by name

Service Creation

CLI

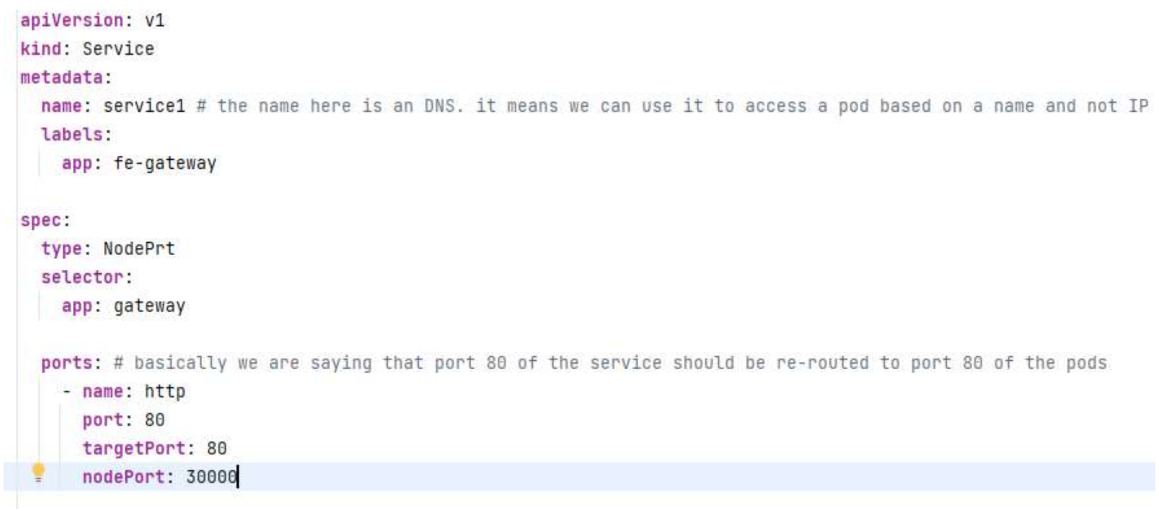
Kubectl port-forward deployment-name hostport:containerport, so basically we are only port forwarding the deployment that mean all the pods inside the deployment will be also port forwarded

YAML



```yaml
apiVersion: v1
kind: Service
metadata:
  name: service1 # the name here is an DNS. it means we can use it to access a pod based on a name and not IP
  labels:
    app: fe-gateway
spec:
  selector:
    app: gateway

  ports: # basically we are saying that port 80 of the service should be re-routed to port 80 of the pods
    - name: http
      port: 80
      targetPort: 80
```

Figure 1: ClusterIp Service



```yaml
apiVersion: v1
kind: Service
metadata:
  name: service1 # the name here is an DNS. it means we can use it to access a pod based on a name and not IP
  labels:
    app: fe-gateway

spec:
  type: NodePrt
  selector:
    app: gateway

  ports: # basically we are saying that port 80 of the service should be re-routed to port 80 of the pods
    - name: http
      port: 80
      targetPort: 80
      nodePort: 30000
```

Figure 2: NodePort Service

For load Balancer we use the same config as the ClusterIp but with type equalt: LoadBalancer

To create those services we simply use kubectl apply -f service-file.yml

Storage

Definition

Application often need to store data and state in a persistent way even after the application/pod is restarted. In kubernetes volume are used as way to save data and to map physical path (on the host hard drive) to a logical one represent it by the container.

Volumes

Volume are basically a storage location. A volume must have a unique name and its lifecycle could depend of the lifecycle of the pods depending on the volume type we are dealing with.

A volume mount is a mapping between physical and containerized location.

Volume Types

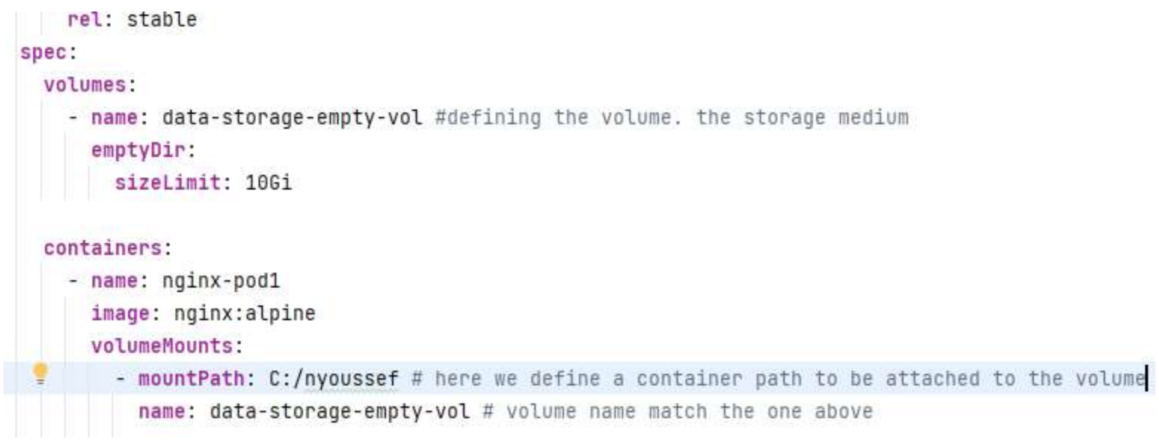
emptyDir: transient data, usefull for sharing data between containers. Its tied to the pod lifecycle, so if the pod get destroyed this directory will be destroyed as well

hostPath: is the actual physical path on the host machine.

Nfs: network file system, its when a pod is mounted on your network files system

ConfigMap/Secret: simply for storing configuration and secret

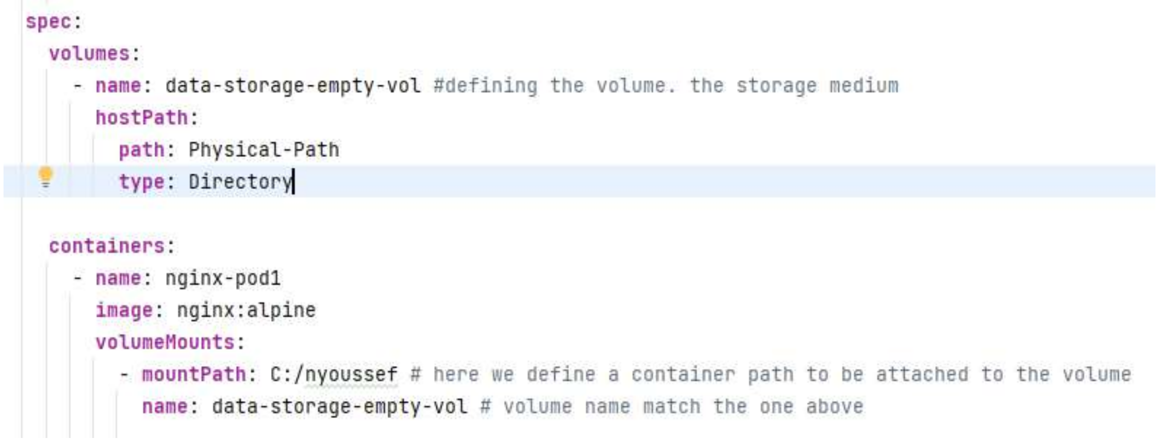
emptyDir



```yaml
    rel: stable
spec:
  volumes:
    - name: data-storage-empty-vol #defining the volume. the storage medium
      emptyDir:
        sizeLimit: 10Gi

  containers:
    - name: nginx-pod1
      image: nginx:alpine
      volumeMounts:
        - mountPath: C:/nyoussef # here we define a container path to be attached to the volume
          name: data-storage-empty-vol # volume name match the one above
```

What is happening here that we are biding a path in the container to emptyDir, so if another container reference the same volume it will be able to see the data, example container1 bind path X to emptyDir and container2 bind path Y to the same volume, what will happen that the data on X and Y are the same

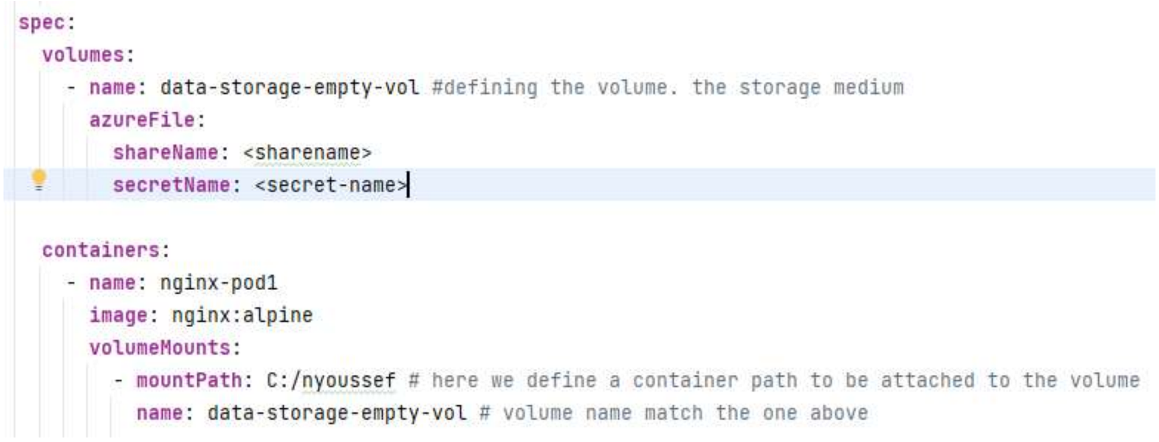
hostPath



```yaml
spec:
  volumes:
    - name: data-storage-empty-vol #defining the volume. the storage medium
      hostPath:
        path: Physical-Path
        type: Directory

  containers:
    - name: nginx-pod1
      image: nginx:alpine
      volumeMounts:
        - mountPath: C:/nyoussef # here we define a container path to be attached to the volume
          name: data-storage-empty-vol # volume name match the one above
```

Here we are biding the mountPath belonging to a container path to a physical path

Cloud:



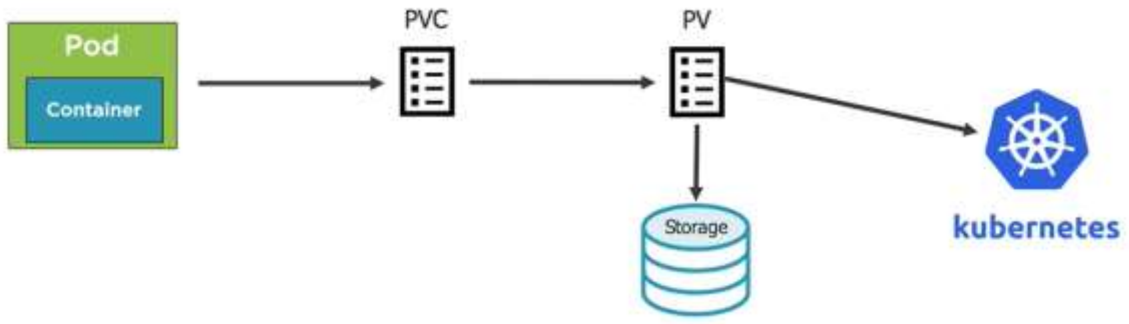
```
spec:
  volumes:
    - name: data-storage-empty-vol #defining the volume. the storage medium
      azureFile:
        shareName: <sharename>
        secretName: <secret-name>

  containers:
    - name: nginx-pod1
      image: nginx:alpine
      volumeMounts:
        - mountPath: C:/nyoussef # here we define a container path to be attached to the volume
          name: data-storage-empty-vol # volume name match the one above
```

Here simply we are saying that our container data will be stored in a cloud provider such as Azure or AWS

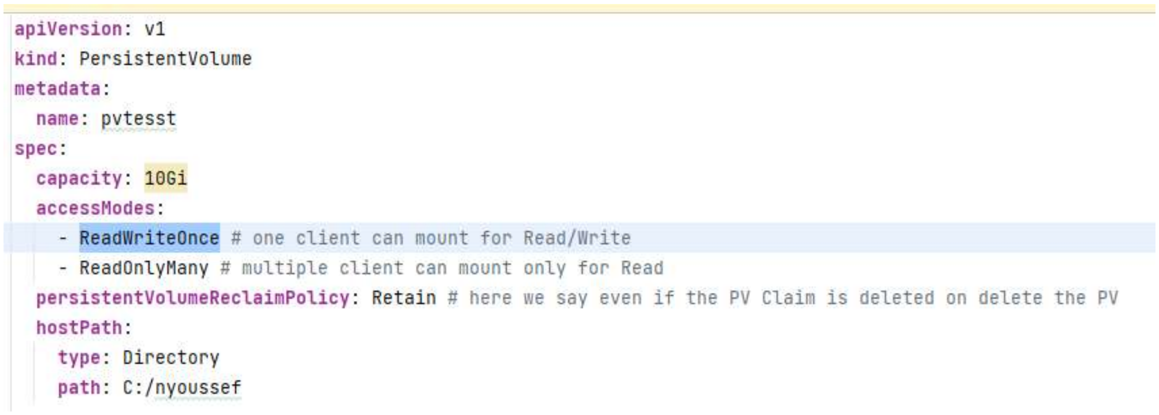
PersistentVolume and PersistentVolumeClaim

Definition

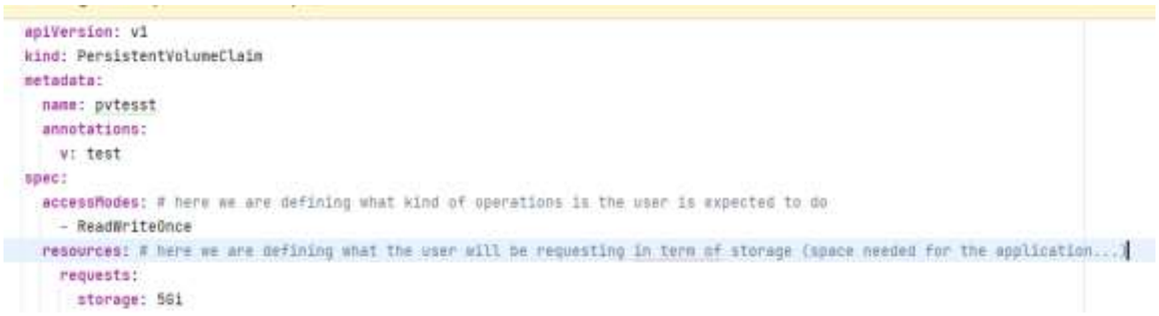
Persistent Volume Is a cluster wide storage. Its provisioned by the cluster administrator and its independent from the pods lifecycle. Meaning only the cluster administrator can control it. Persistent Volume is associated with PersistentVolumeClaim that resembles to some sort of permissions to acquire the storage/volume. Meaning that a PVC is bound to a PV. The user that have this PVC will be able to acquire the PV declared in the PVC. Note that in this case the user is only aware of the PVC and the PV , so here we are hiding the PV configuration and we are giving the user a PVC that will be exchanged with kubernetes and then kubernetes will simply bind the pod to the PV specified by the exchanged PVC



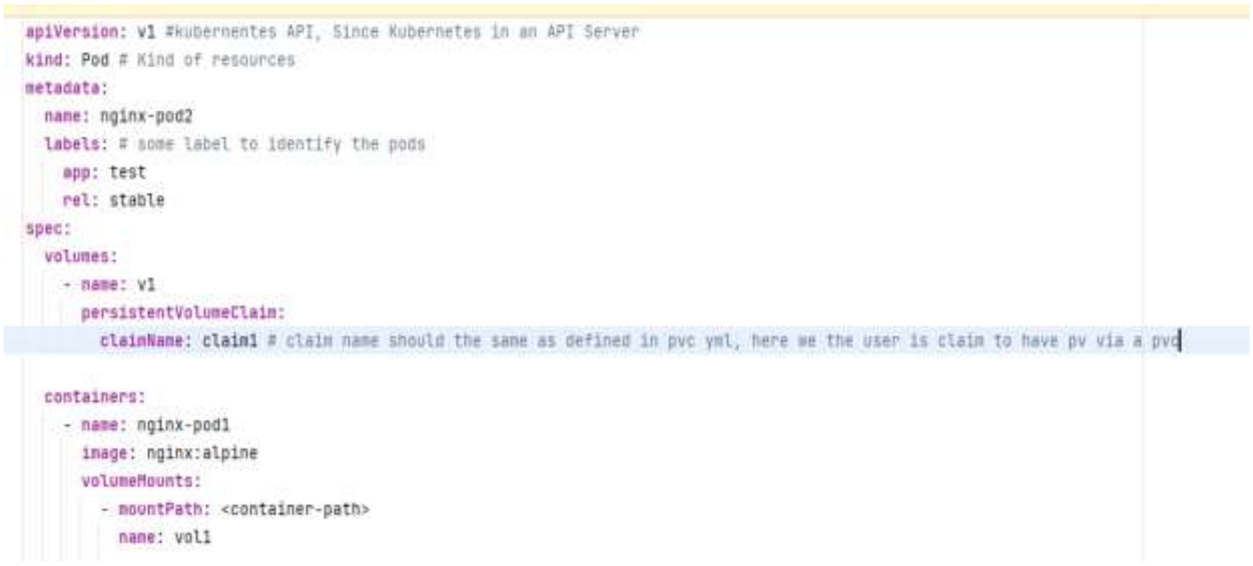
YAML



```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pvtesst
spec:
  capacity: 10Gi
  accessModes:
    - ReadWriteOnce # one client can mount for Read/Write
    - ReadOnlyMany # multiple client can mount only for Read
  persistentVolumeReclaimPolicy: Retain # here we say even if the PV Claim is deleted on delete the PV
  hostPath:
    type: Directory
    path: C:/nyoussef
```



```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvtesst
  annotations:
    v: test
spec:
  accessModes: # here we are defining what kind of operations is the user is expected to do
    - ReadWriteOnce
  resources: # here we are defining what the user will be requesting in term of storage (space needed for the application...)
    requests:
      storage: 5Gi
```



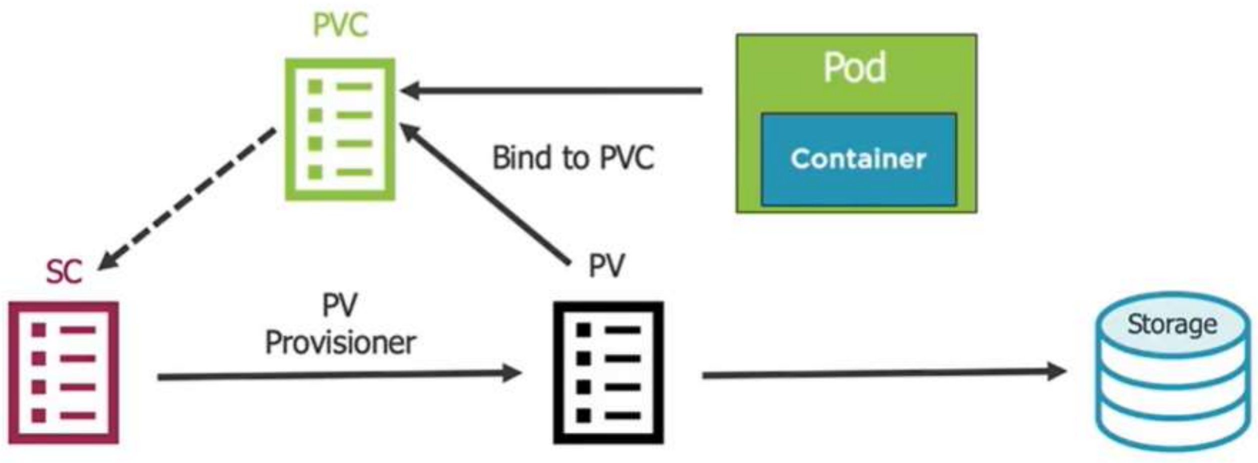
```yaml
apiVersion: v1 #kubernentes API, Since Kubernetes in an API Server
kind: Pod # Kind of resources
metadata:
  name: nginx-pod2
  labels: # some label to identify the pods
    app: test
    rel: stable
spec:
  volumes:
    - name: v1
      persistentVolumeClaim:
        claimName: claim1 # claim name should the same as defined in pvc yml, here we the user is claim to have pv via a pvc

  containers:
    - name: nginx-pod1
      image: nginx:alpine
      volumeMounts:
        - mountPath: <container-path>
          name: vol1
```

First, we need to create a PV by defining its storage capacity, paths on the physical or the cloud, the type of storage, provider and so on. Then we need to define a PVC that simply represent a claim to acquire a specific PV. After this the pod can acquire a Volume by simply referencing the claim.

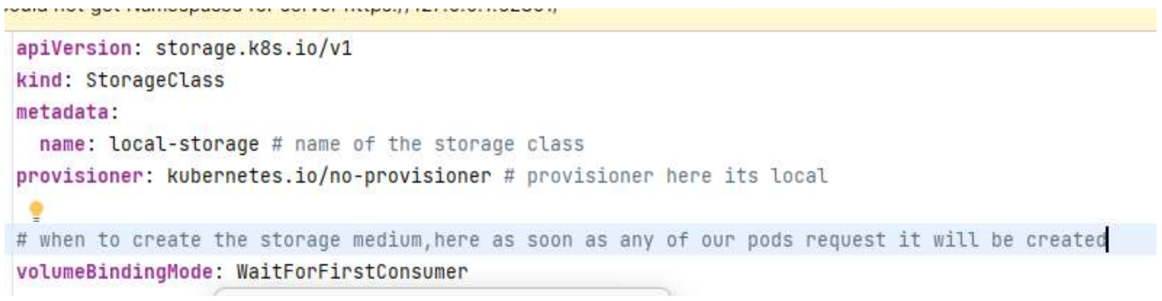
Storage Class

Definition

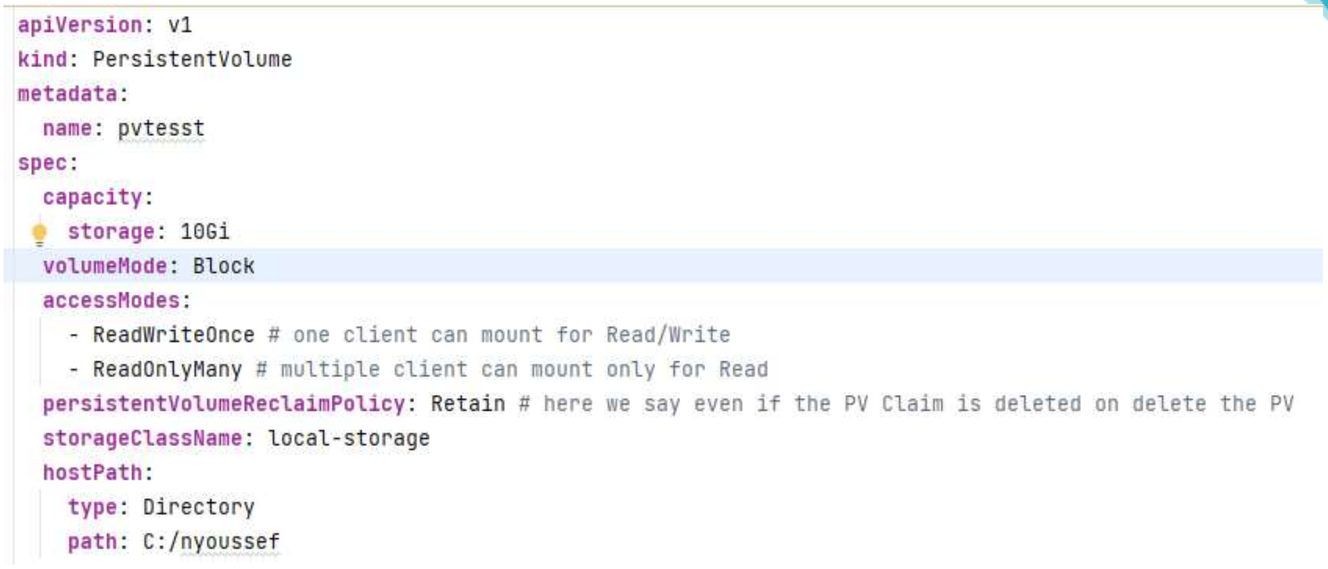
A StorageClass provides a way for administrators to describe the "classes" of storage they offer



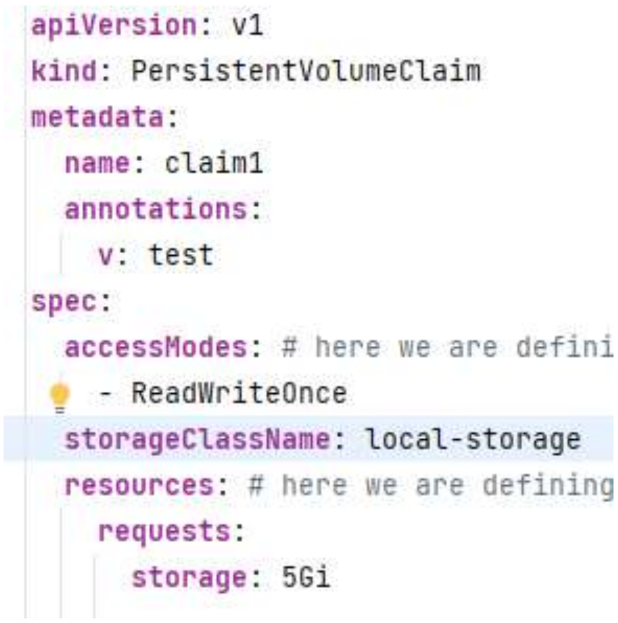
Yaml



```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local-storage # name of the storage class
provisioner: kubernetes.io/no-provisioner # provisioner here its local

# when to create the storage medium,here as soon as any of our pods request it will be created
volumeBindingMode: WaitForFirstConsumer
```



```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pvtesst
spec:
  capacity:
    storage: 10Gi
  volumeMode: Block
  accessModes:
    - ReadWriteOnce # one client can mount for Read/Write
    - ReadOnlyMany # multiple client can mount only for Read
  persistentVolumeReclaimPolicy: Retain # here we say even if the PV Claim is deleted on delete the PV
  storageClassName: local-storage
  hostPath:
    type: Directory
    path: C:/nyoussef
```



```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: claim1
  annotations:
    v: test
spec:
  accessModes: # here we are defini
    - ReadWriteOnce
  storageClassName: local-storage
  resources: # here we are defining
    requests:
      storage: 5Gi
```

Note: we need to setup PV and PVC with the created StorageClass in order to reference it.

Config and Secrets

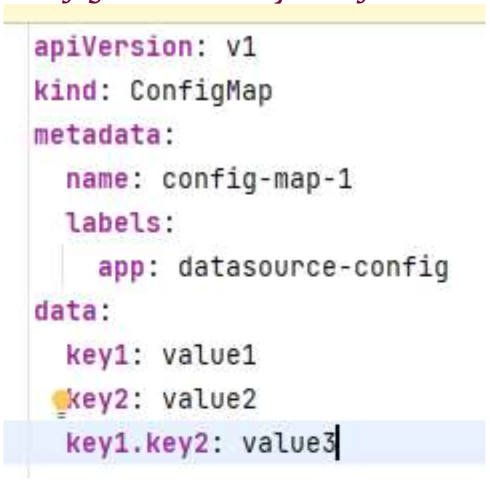
Config Map

Definition

It’s a way for storing configuration. Pods can access this configuration from anywhere on the cluster. Config Map are simply injected in containers

YAML

Configuration in yaml file



```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-map-1
  labels:
    app: datasource-config
data:
  key1: value1
  key2: value2
  key1.key2: value3
```

Config from properties file

Kubectl create configmap <name> --from-file=<path-file>

What will happened that this command will create a yaml similar to the one above with all the data from the file loaded in the file and by default the separator is specified to be | specified as

data: | in the above image

Or

Kubectl create configmap <name> --from-env-file=<path-file>

That will generate a yaml exactly like the image above

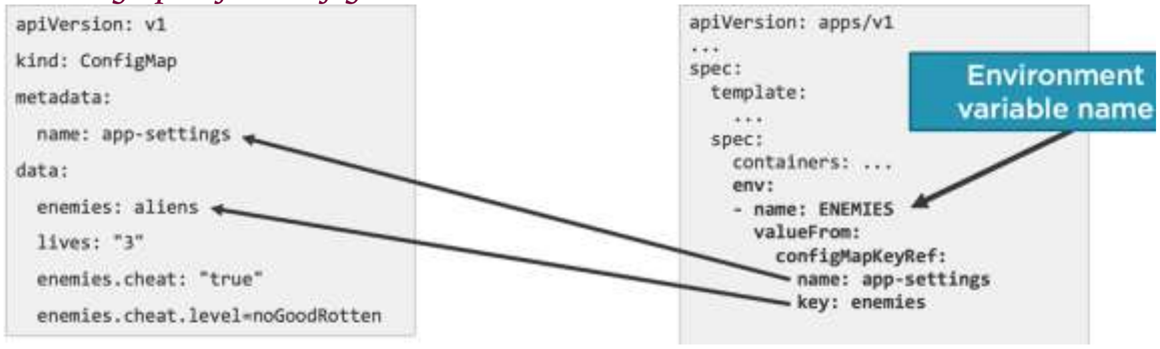
Or

Kubectl create configmap name –from-litteral=apiUrl=http....

Basical we will generate yaml file of type ConfigMap with config downloaded from the http

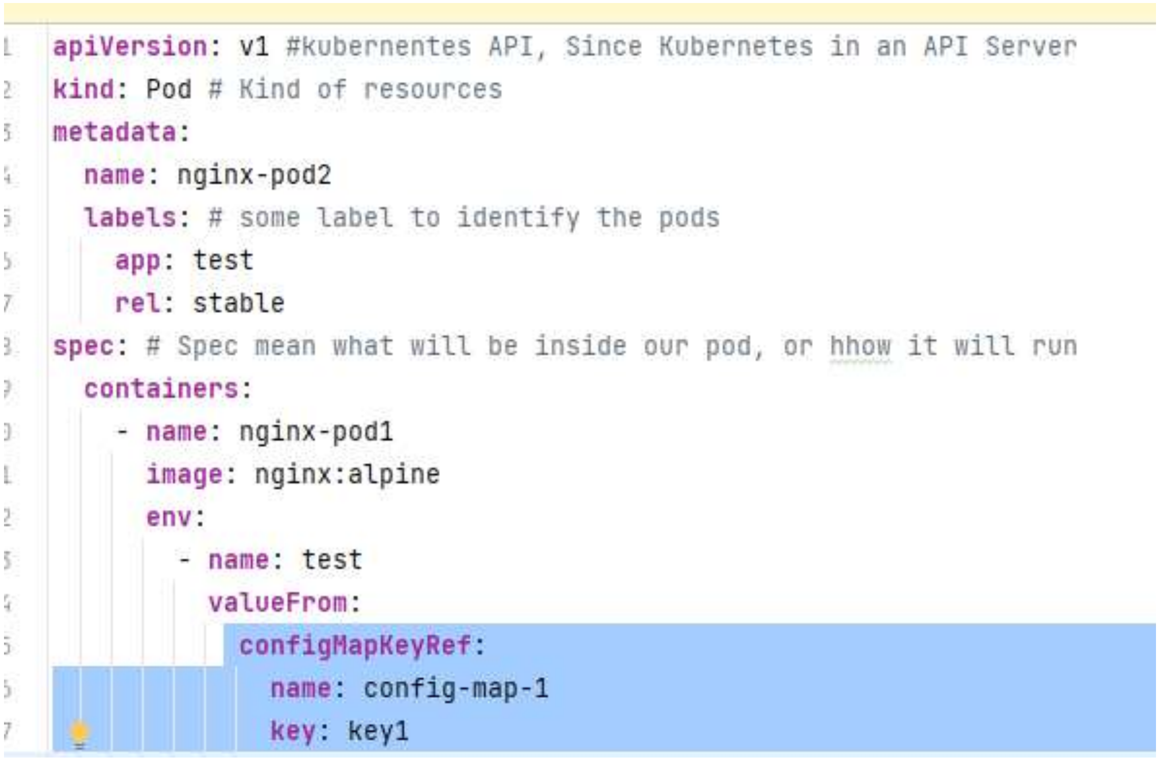
Loading Config into Pod

Loading specific config into container env varaibles



```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-settings
data:
  enemies: aliens
  lives: "3"
  enemies.cheat: "true"
  enemies.cheat.level=noGoodRotten
```

```
apiVersion: apps/v1
...
spec:
  template:
    ...
    spec:
      containers: ...
      env:
      - name: ENEMIES
        valueFrom:
          configMapKeyRef:
            name: app-settings
            key: enemies
```



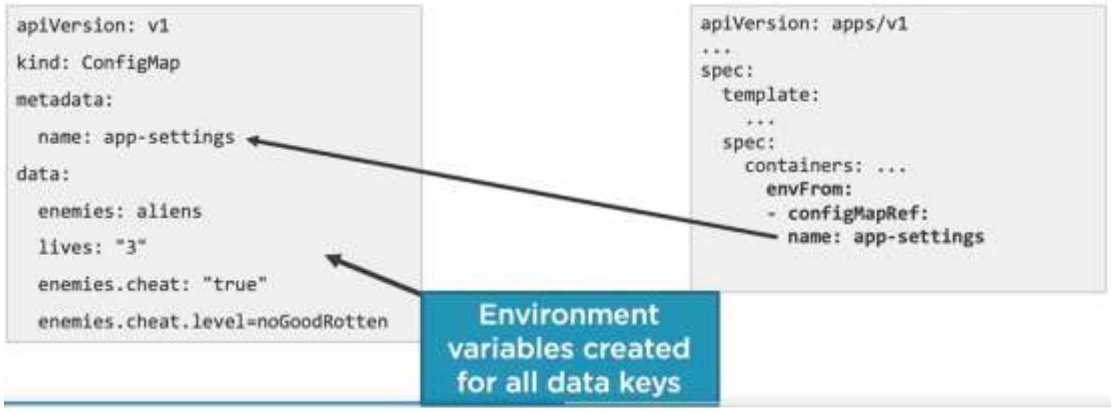
```
apiVersion: v1 #kubernentes API, Since Kubernetes in an API Server
kind: Pod # Kind of resources
metadata:
  name: nginx-pod2
  labels: # some label to identify the pods
    app: test
    rel: stable
spec: # Spec mean what will be inside our pod, or hhow it will run
  containers:
    - name: nginx-pod1
      image: nginx:alpine
      env:
        - name: test
          valueFrom:
            configMapKeyRef:
              name: config-map-1
              key: key1
```

From the above image first configMapKeyRef.name is referencing in config map resource

And configMapKeyRef.key is simply providing the value of a config with key = key1 into the env variable of the container defined by test (from name:test).

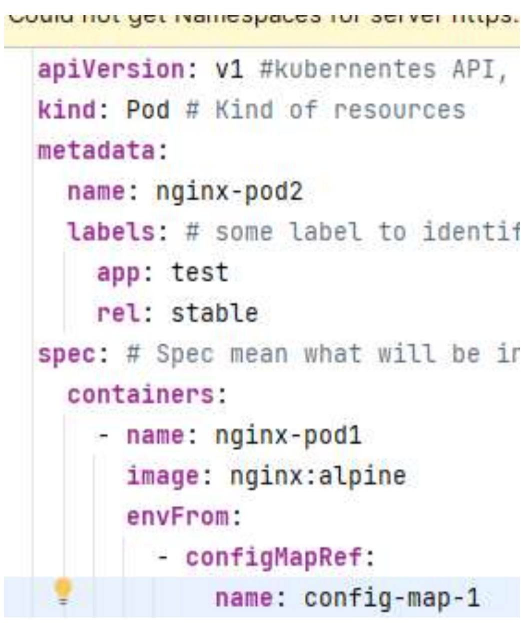
Loading All configs into env varaible

Basically we are telling to load all the key/value config of the config map into the containers environment variables.



```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-settings
data:
  enemies: aliens
  lives: "3"
  enemies.cheat: "true"
  enemies.cheat.level=noGoodRotten
```

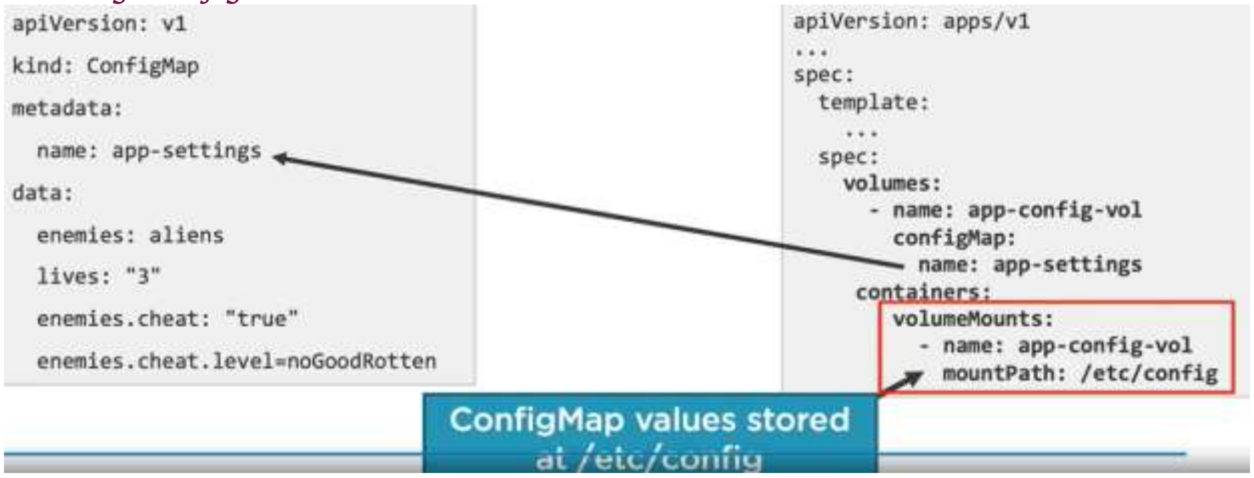
```
apiVersion: apps/v1
...
spec:
  template:
    ...
    spec:
      containers: ...
        envFrom:
        - configMapRef:
          name: app-settings
```



Could not get Namespaces for server https:

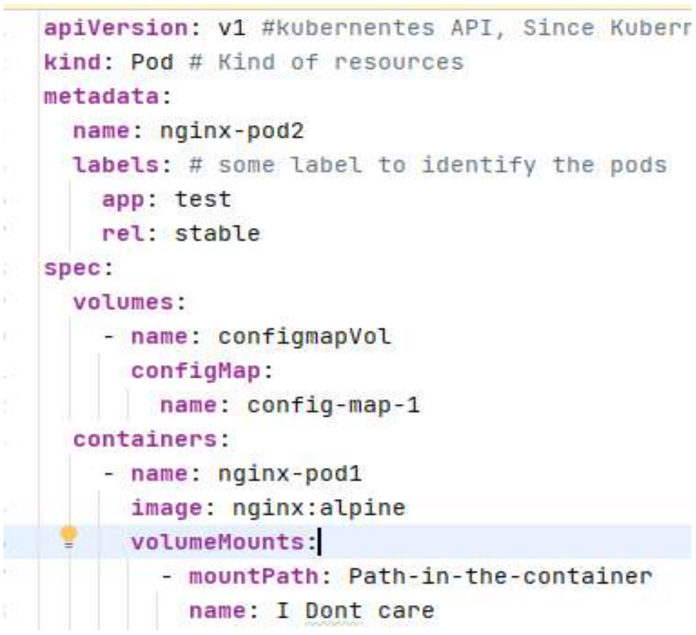
```
apiVersion: v1 #kubernentes API,
kind: Pod # Kind of resources
metadata:
  name: nginx-pod2
  labels: # some label to identif
    app: test
    rel: stable
spec: # Spec mean what will be ir
  containers:
    - name: nginx-pod1
      image: nginx:alpine
      envFrom:
        - configMapRef:
            name: config-map-1
```

Loading Config into volumes



```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-settings
data:
  enemies: aliens
  lives: "3"
  enemies.cheat: "true"
  enemies.cheat.level=noGoodRotten
```

```
apiVersion: apps/v1
...
spec:
  template:
    ...
    spec:
      volumes:
        - name: app-config-vol
          configMap:
            name: app-settings
      containers:
        volumeMounts:
          - name: app-config-vol
            mountPath: /etc/config
```



```
apiVersion: v1 #kubernentes API, Since Kuberr
kind: Pod # Kind of resources
metadata:
  name: nginx-pod2
  labels: # some label to identify the pods
    app: test
    rel: stable
spec:
  volumes:
    - name: configmapVol
      configMap:
        name: config-map-1
  containers:
    - name: nginx-pod1
      image: nginx:alpine
      volumeMounts:
        - mountPath: Path-in-the-container
          name: I Dont care
```

What will happen here that our volume is now referencing a config map instead of a physical file/folder. In volume mount we are simply saying that the config map content will be stored in the container path specified in the mountPath value

Secrets

Definition

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in a container image. Using a Secret means that you don't need to include confidential data in your application code. They can be mounted into container or passed as environment variables. They are written into tmpfs a temporary file that is not stored physically anywhere.

Creating Secrets

Generic way

From Key/value pair

kubectl.exe create secret generic sec1 --from-literal=key1=value1

here we are creating an un-encrypted secret using command line in the format of key/value pair

From Key/file path

kubectl.exe create secret generic sec2 --from-file=key1=<file-path>

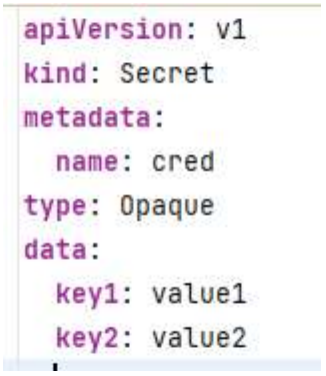
its like key/value pair, but the value is issued from a file

TLS Secret

Kubectl create secret tls tls-1 –cert=file-path1 –key=filepath2

Here we are creating a secret that hold a tls public/private key pair. File-path1 represent the public ket and file-path2 represent the private key. P.S: this secret is used mainly for https purposes

YAML

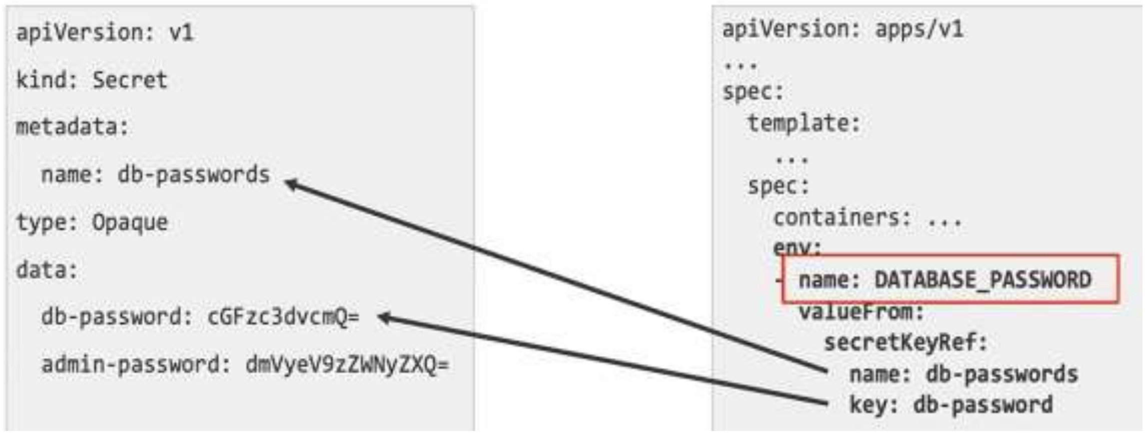


```yaml
apiVersion: v1
kind: Secret
metadata:
  name: cred
type: Opaque
data:
  key1: value1
  key2: value2
```

Be aware that those values should be encoded (base64) so basically they can be decoded if some have this file

Using Secrets

As Environment variable

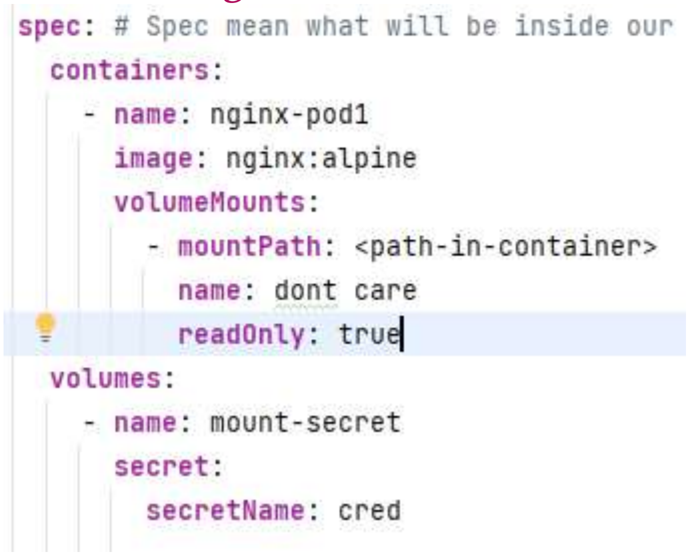


```yaml
apiVersion: v1
kind: Secret
metadata:
  name: db-passwords
type: Opaque
data:
  db-password: cGFzc3dvcmQ=
  admin-password: dmVyeV9zZWNyZXQ=
```

```yaml
apiVersion: apps/v1
...
spec:
  template:
    ...
  spec:
    containers: ...
    env:
    - name: DATABASE_PASSWORD
      valueFrom:
        secretKeyRef:
          name: db-passwords
          key: db-password
```

What will happen that first our container will reference a secret resource from kubernetes by its name, than it will bind the container's env defined by env.name (DATABASE_PASSWORD in this case) to the db-password defined in the Secert. The secretKeyRef take 2 args the name witch represent the resource we need to contact and the key that will reference a value from the secret

As Mounting on the container FS

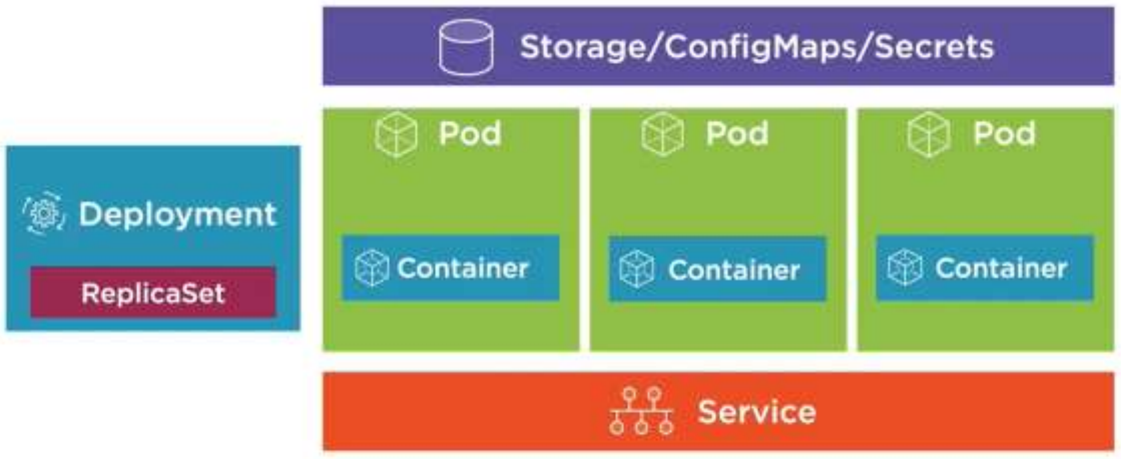


```
spec: # Spec mean what will be inside our
  containers:
    - name: nginx-pod1
      image: nginx:alpine
      volumeMounts:
        - mountPath: <path-in-container>
          name: dont care
          readOnly: true
  volumes:
    - name: mount-secret
      secret:
        secretName: cred
```

Simply in our volume declaration we say that this volume is pointing to a secret stored in kubernetes with the name defined in volume.secret.secretName. Then in our mounting we simply point this volume that contains the secret to the container path in this case our secret will be a file stored in the container

Connecting Everything together

Full Application Design



Step 1

We need to define our components (Backend technologies, frontend, databases)

This step is crucial in order to define what we will be having from connectivity, storage, configurations, secrets and networking.

Step 2

Since pods are a series of containers, we need to have images. Images can be created using docker. The objective here is to have a well defined image that will actually the container content from environment variable, file system path that our application will be aware of to read and write data and some configs and secret as environment variables

Step 3 (Storage and Volumes)

It is know that each application at some point need to persist data like database object , server logs, shared file system, So we need to define a storage entity for this by following these steps:

Step 3.1

We need to define our storageclass. This step is important because we need to have visibility and control on witch device or medium is our data in. We may choose from many option like host storage, cloud, nfs

Step 3.2

We need to create our persistentVolume with PersistentVolumeClaim in order to have an administrative storage (a storage that is abstracted from the developer)

Step 4 (ConfigMaps and Secrets)

After we defined our storages, we can now proceed in creating our configuration Maps and our secret resources. The objectives is to be able to centralized our configs to be able to share with multiple pods across the entire cluster and to isolate the secret (making them out of the reach of the developer). Normally we should consider injecting those configs and secret as environment variables that will be useable in our containers

P.S: here we can also consider connecting our configs and secret to a volume in case an application wants to access them via its own file system

Step 5 (Service)

In a normal application, we need to handle multiple layer of services/microservices, for example backend, frontend, computation, caching servers and databases. So there is some steps we need to consider

Networking is an essential part of communication between each container/pods/replicas of nodes. Plus we need to ensure that our users will have the ability to speak with our applications, however networking inside our cluster may become complex and unpredictable

(Dynamic IPs, pods re-creation, complex ports mapping). In this case it's highly advisable to use Services. A service is useful to have a single-entry point of a set of pods/container in our cluster. A service knows about our application Ips and ports even if there are changing or being re-created, plus it can play a role of load balancing in order to achieve optimal usage of the replication and also it plays the role of a router/proxy that will intercept the user request and re-routed to the correct service. A service also can help pods and containers to communicate with each other by name it means that our pods no longer have to know about IP in order to connect with other pods simply the name of the pod and the port of the container to reach the service of specific container.

Step 6 (Deployment and Replication)

In this final step we need to create our deployment with replications and pods creations. The deployment is useful for versioning, updating, rollback, achieving zero downtime, failure recovery and scaling. In Deployment we should create our pods/and container to be aligned with the Images specifications in Step 2 and all the configurations done in Step 3 to 5.

P.S: it's a good practice to considering readiness and liveness probes

Troubleshooting

Some Commands useful for logging

At some point a pod may not behave correctly, In this case kubernetes provide some way to troubleshoot what happing.

Kubectl logs pod-name -c container-name => this will display logs for a specific container in a pod

Kubectl logs pod-name => displaying cmd of the current pod

Kubectl logs -p pod-name => displaying logs of a previously running pod, its like a history check

Kubectl logs -f pod-name => for streaming logs of a specific container

Kubectl get res-name -o wide => to see whats happing in our resource/status

Some Probable cause of errors

1- Miss configuration, when an image/container is expecting an environmental variable that isn't available. This due to a configmap/secret mistake
2- Networking errors. Some time some application need to communicate with other pods via network, a possible mistake that this application in configured with the incorrect host or with IP or it could be firewall issue or even mistakes in service configuration

Summary of Commands

CMD	Description
Kubectl version	Get the current version of kubernetes
Kubectl cluster-info	Get some info about our cluster
Kubectl get all	Get all existing resources
Kubectl apply -f <yaml-file.yml>	To run a resource script and overwrite any exiting one
Kubectl create -f <yaml-file.yml>	Only create the resource
kubectl run <pod-name> --image=<image-name>	to create a pod
kubectl port-forward <pod-name> public-port:pods-port	To expose a pod to a public IP
Kubectl delete <pod-name>	To delete a specific pod
Kubelctl describe <kind> <resource-namee>	To display info about the a specific resource of a specific kind
kubectl exec -it <pod-name> <cmd>	We simply applying a command to pod.P.S cmd = sh will allow us to access the terminal
Kubectl get Deployment <deployment-name> --show-label	Getting info about a specific deployment with all labels attached to it
Kubectl get pod <pod-name> --show-label	Getting info about a specific pod with all its labels
Kubetctl get <kind> <res-name> -o <format>	Showing the details of a resource. Format can be: yaml,wide,json ....
Kubectl get <kind> <resource-name> -l key=value	Used to get resources with the label defined by key and value
kubectl scale -f filename.yml –replicas=<number_of_replicat>	Changing the replicas number for a specific deployment
Kubectl create configmap <name> --from-file=<path-file>	Creating a config map loaded from a file
Kubectl create configmap <name> --from-env-file=<path-file>	Creating a config map but from env file
Kubectl create configmap <name> --from-litteral=<options>	Creating config map from different source specified by options. It can be in the form of key value pair or http request or .. (Please refer to the doc for more options)
kubectl.exe create secret generic <res-name> --from-literal=key1=value1	Creating a generic secret using key/value pair
kubectl.exe create secret generic <res-name> --from-file=key1=<file-path>	Creating a generic secret using key/value pair, with the value issued from a file

Kubectl create secret tls tls-1 –cert=file-path1 –key=filepath2	Declaring a public/private key pairs mainly for a TLS communication
Kubectl apply -f <dir>	This command will simply create resource describe by a directory of yml file
Kubectl logs pod-name -c container-name	To display logs of a container in a specific pod
Kubectl logs pod-name	Logs of a specific pod
Kubectl logs -p pod-name	displaying logs of a previously running pod, its like a history check
Kubectl logs -f pod-name	Streaming logs of specific pod
Kubectl get <kind> --watch	A command to watch the creation of a specific resource kind in real time