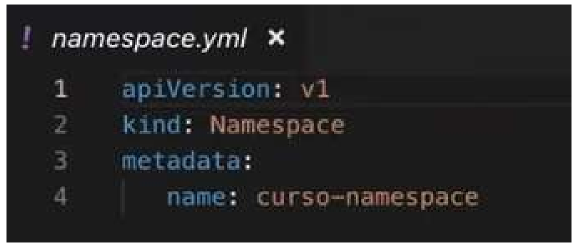
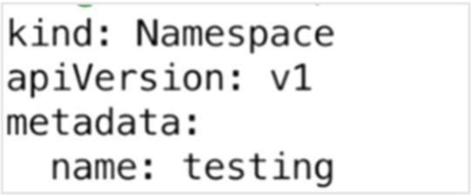
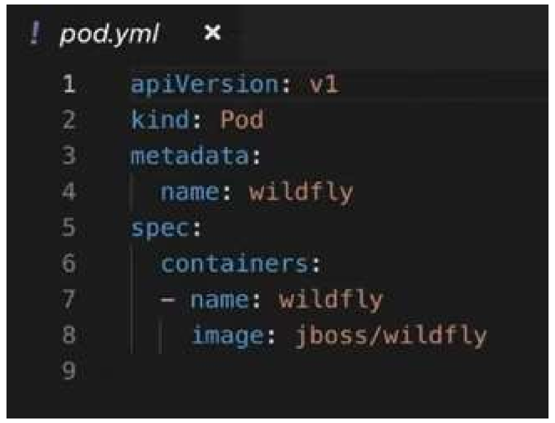
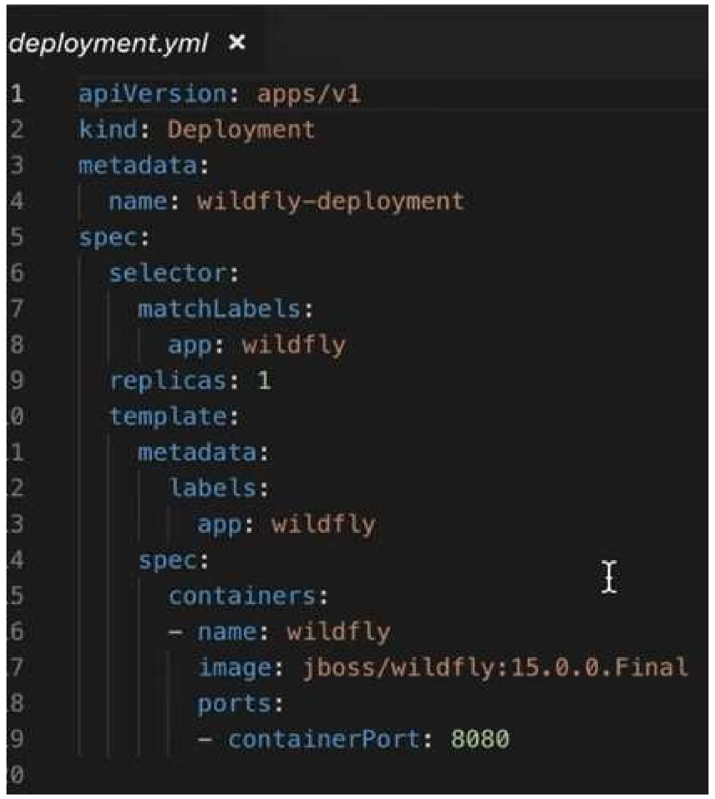
Command	Description	Key word
kubectl get nodes kubectl get nodes -o wide	To see the workers running in the cluster. With -o wide I can see more information. (Internal IP, External IP and more.)	See / Get / Nodes
kubectl get namespaces kubectl get ns	To see the namespaces (You can divide the cluster in multiple namespaces)	See / Get / Namespace
kubectl create namespace nameOfTheNamespace	To create a namespace. Also, I can create a yaml file and run the command: kubectl apply -f namespace.yml  ! namespace.yml × apiVersion: v1 kind: Namespace metadata: name: curso-namespace Other example:  kind: Namespace apiVersion: v1 metadata: name: testing	Create / Namespace
kubectl delete namespace nameOfTheNamespace	To delete a namespace. Also, I can delete with a yaml file and run the command: kubectl delete -f namespace.yml	Delete / Namespace
kubectl -n nameOfTheNamespace apply -f pod.yml Note: You run a script always in the same way.	To create a pod in a namespace:  ! pod.yml × apiVersion: v1 kind: Pod metadata: name: wildfly spec: containers: - name: wildfly image: jboss/wildfly	Create / Pod
kubectl get pods	To see the pods	See / Get / Pod
kubectl label pod nameOfThePod role=newRole --overwrite	To change the role in the pod.	Change
kubectl describe pod nameOfThePod	To see information about the pod	See / Describe / Pod
kubectl exec -it nameOfThePod -- /bin/bash	To access to the pod and there you can run commands.	Access / Exec / Pod
kubectl logs -f nameOfThePod	To check the logs of our pod	Logs / Pod

kubectl -n nameOfTheNamespace apply -f deployment.yml

Note: You run a script always in the same way.

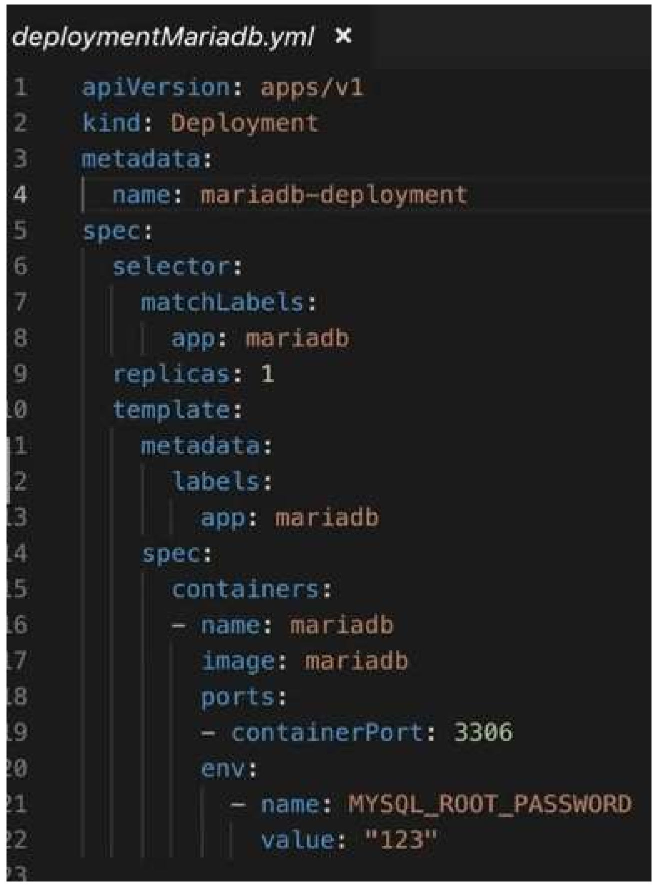
To deploy you need to run the command and the yml is:

You create pods with the deployment.



deployment.yml ×

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wildfly-deployment
spec:
  selector:
    matchLabels:
      app: wildfly
  replicas: 1
  template:
    metadata:
      labels:
        app: wildfly
    spec:
      containers:
      - name: wildfly
        image: jboss/wildfly:15.0.0.Final
        ports:
        - containerPort: 8080
```



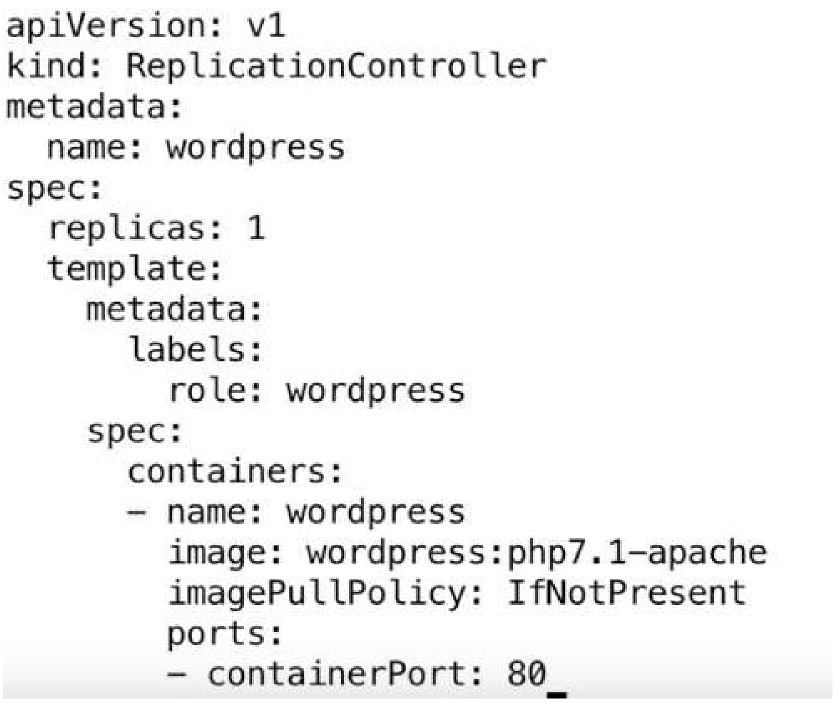
deploymentMariadb.yml ×

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mariadb-deployment
spec:
  selector:
    matchLabels:
      app: mariadb
  replicas: 1
  template:
    metadata:
      labels:
        app: mariadb
    spec:
      containers:
      - name: mariadb
        image: mariadb
        ports:
        - containerPort: 3306
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: "123"
```

You can see environments variables in the mariadb yaml

Other example:

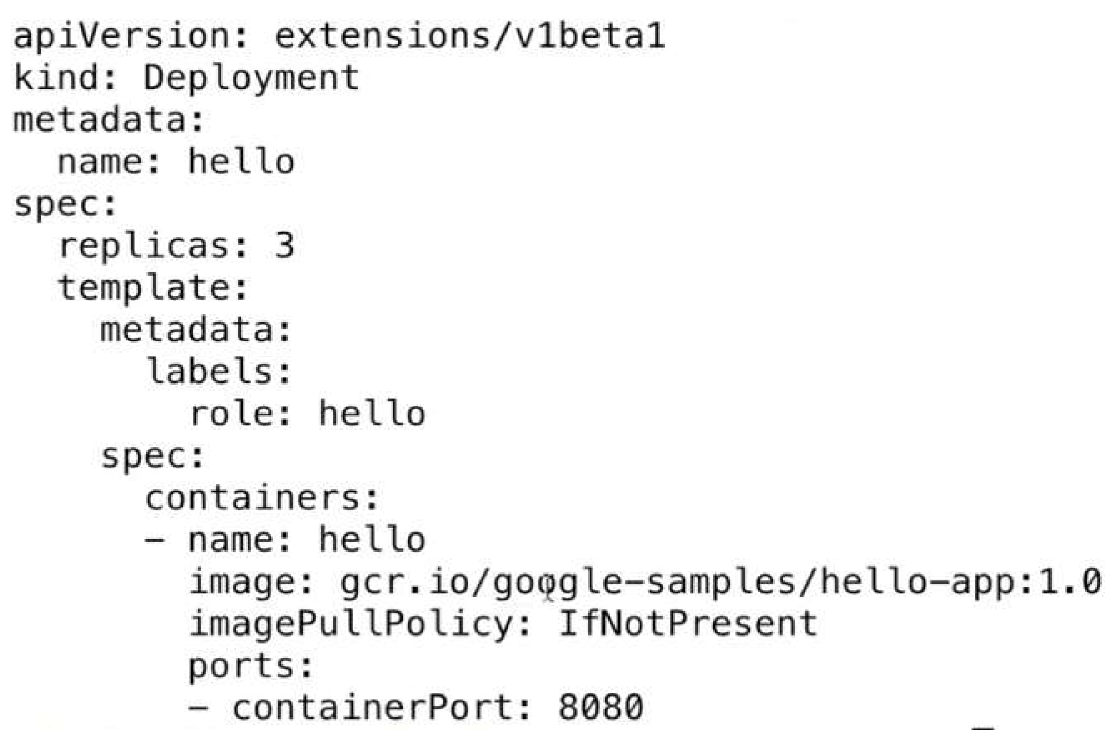
Deploy /
Repplication Controller



```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: wordpress
spec:
  replicas: 1
  template:
    metadata:
      labels:
        role: wordpress
    spec:
      containers:
      - name: wordpress
        image: wordpress:php7.1-apache
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
```

<table>
<tr>
<td></td>
<td>

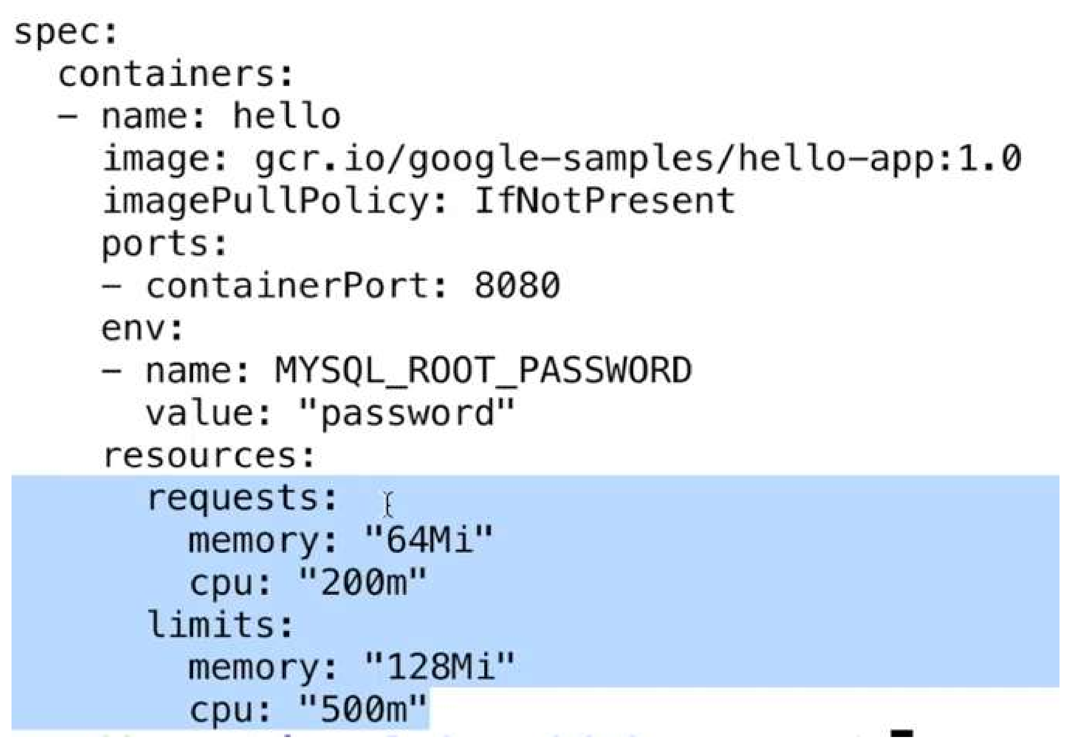
Check that kind is different, check the kubernete documentation to know the differentes controllers, but with deployment you change something and create a new version and all the pods in the old version will be replaced with new pods with the new version



```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: hello
spec:
  replicas: 3
  template:
    metadata:
      labels:
        role: hello
    spec:
      containers:
      - name: hello
        image: gcr.io/google-samples/hello-app:1.0
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
```

</td>
<td></td>
</tr>
<tr>
<td></td>
<td>

Also you can assign CPU and Memory to each pod:



```
spec:
  containers:
  - name: hello
    image: gcr.io/google-samples/hello-app:1.0
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 8080
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: "password"
    resources:
      requests:
        memory: "64Mi"
        cpu: "200m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

</td>
<td></td>
</tr>
<tr>
<td>kubectl get deployments</td>
<td>To see the deployments</td>
<td>See / Deploy</td>
</tr>
<tr>
<td>kubectl delete -f deployment.yml</td>
<td>To delete a deployment.</td>
<td>Delete / Deploy</td>
</tr>
</table>

<table>
<tr>
<td>

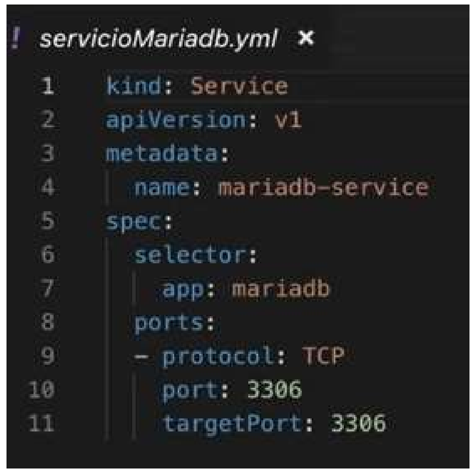
kubectl -n nameOfTheNamespace apply -f servicioMariadb.yml

Note: You run a script always in the same way.

</td>
<td>

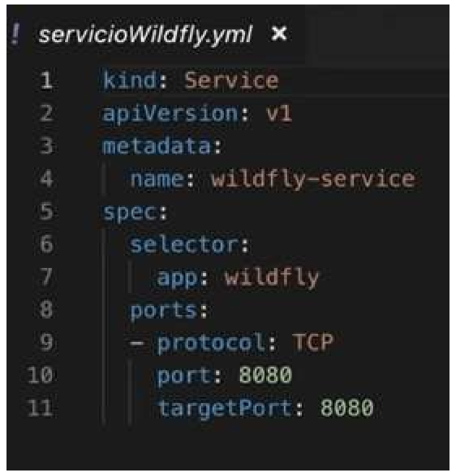
You can use a service to access to your pods through different protocols (HTTP, TCP).

To create a service:



servicioMariadb.yml

```yaml
kind: Service
apiVersion: v1
metadata:
  name: mariadb-service
spec:
  selector:
    app: mariadb
  ports:
  - protocol: TCP
    port: 3306
    targetPort: 3306
```



servicioWildfly.yml

```yaml
kind: Service
apiVersion: v1
metadata:
  name: wildfly-service
spec:
  selector:
    app: wildfly
  ports:
  - protocol: TCP
    port: 8080
    targetPort: 8080
```

As you can see, the selector used is the same selector than in the deployment script. In this way I relate the service with the pods created with the deployment script.

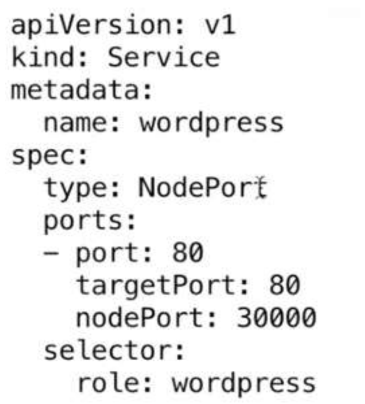
Other example: NodePort is a way to expose a port, and in this way you create a port in the worker that point to the container that we want.
targetPort is the port in the container.
nodePort is the port in the node that point to the point 80 in the container.
This service will search the pods with the role wordpress and will send the traffc to these pods.

</td>
<td>

Create / Service

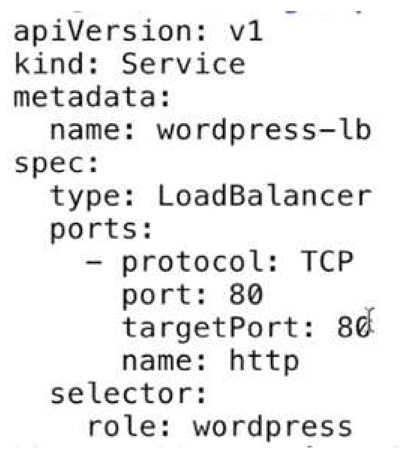
</td>
</tr>
</table>

<table>
<tr><td></td><td>



```yaml
apiVersion: v1
kind: Service
metadata:
  name: wordpress
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
    nodePort: 30000
  selector:
    role: wordpress
```

Other example: LoadBalancer , with this, the service will conect to your cloud provider and create a load balancer.



```yaml
apiVersion: v1
kind: Service
metadata:
  name: wordpress-lb
spec:
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      name: http
  selector:
    role: wordpress
```

</td><td></td></tr>
<tr><td>kubectl get service
kubectl get svc
kubectl -n nameOfTheNamespace get svc</td><td>To see the services.</td><td>See / Service</td></tr>
<tr><td>kubectl describe service nameOfTheService</td><td>To see information about the pod</td><td>See / Describe / Service</td></tr>
</table>

<table>
<tr>
<td>

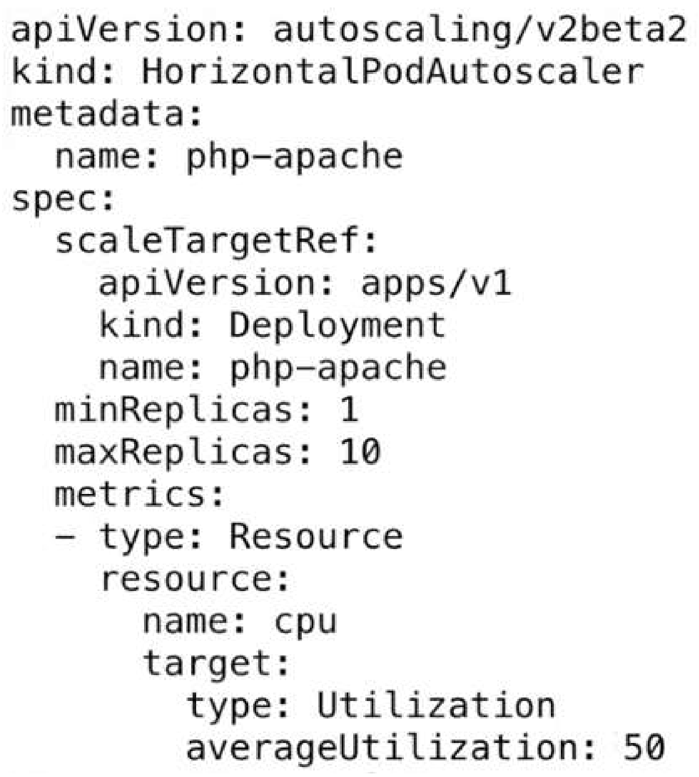
HPA (horizontal pod autoscaling)

Note: You run a script always in the same way.

</td>
<td>

To control a metric , for example CPU:

Pods with php-apache (check deployment yml) label will be created maintaining the CPU in 50%, min is 1 and max is 10.



```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

</td>
<td>

HPA

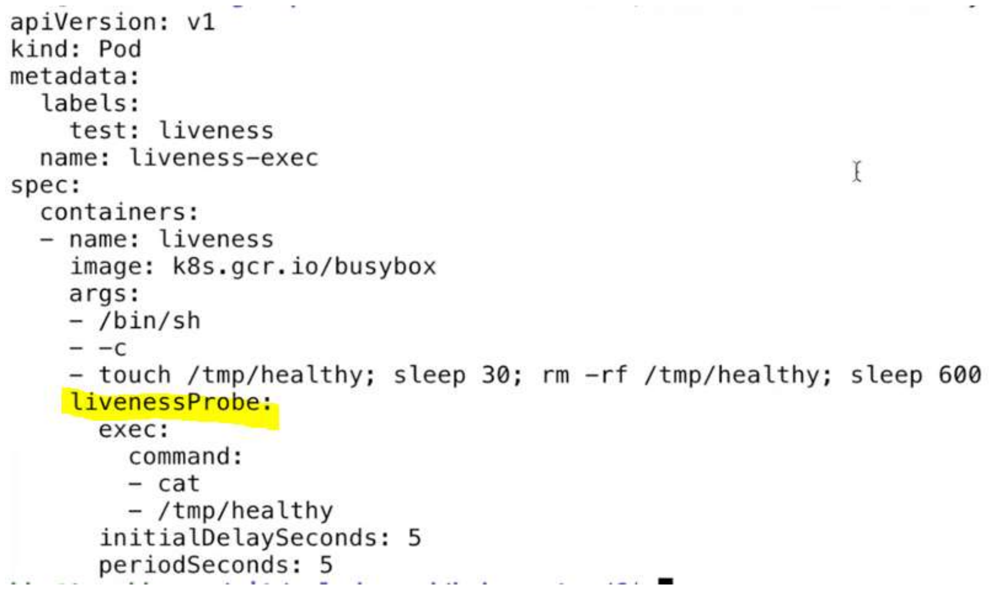
</td>
</tr>
<tr>
<td>

Liveness and readiness probes

</td>
<td>

Liveness probe is for Kubernetes knows if your pod is alive. If the liveness file, kubernetes restart the pod.
Readiness probe is for Kubernetes knows if your pod is ready to receive traffic.

Example 1: Exec, you can exec a command to know if there is a file. In this case we run cat /tmp/healthy, the liveness will fail because we remove the file after 30 seconds.



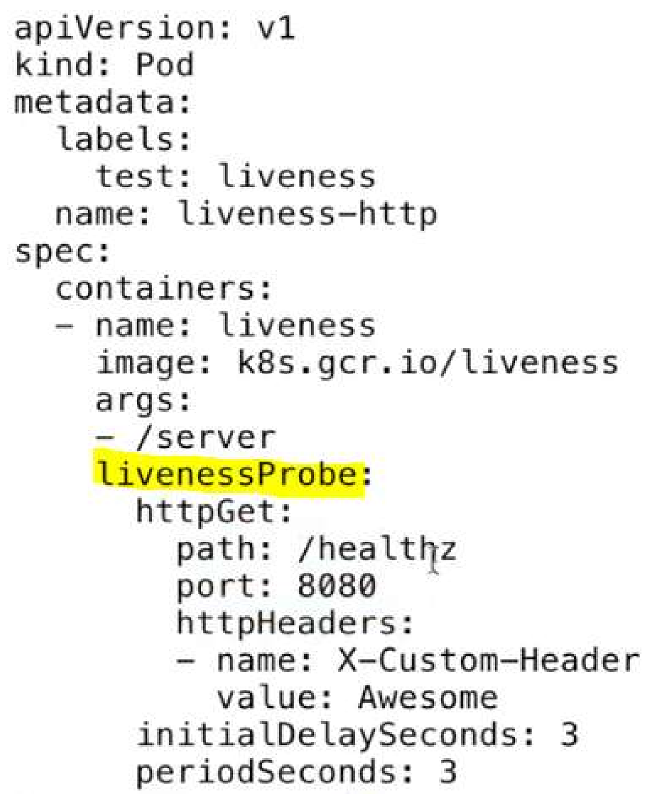
```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

</td>
<td>

Liveness and readiness probes

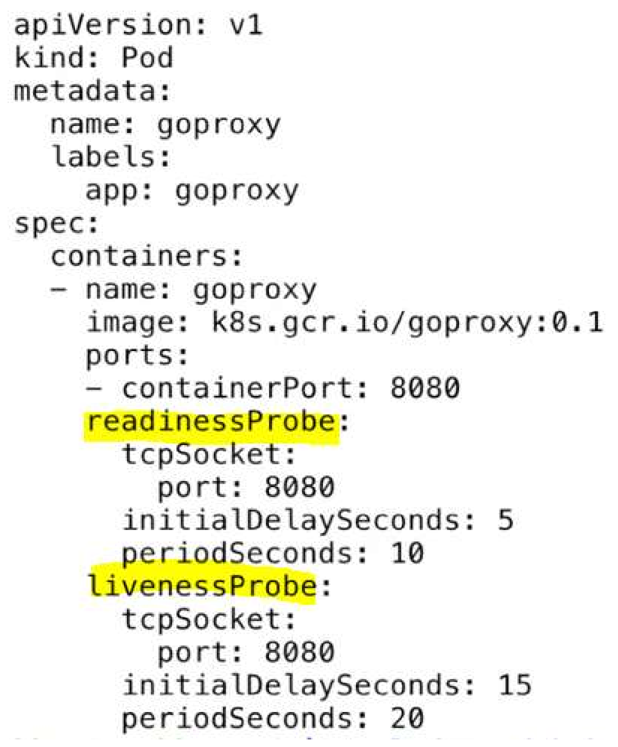
</td>
</tr>
</table>

Example 2: HTTP Get, if the file doesn’t exist , the http get fail and Kubernetes restart the pod.



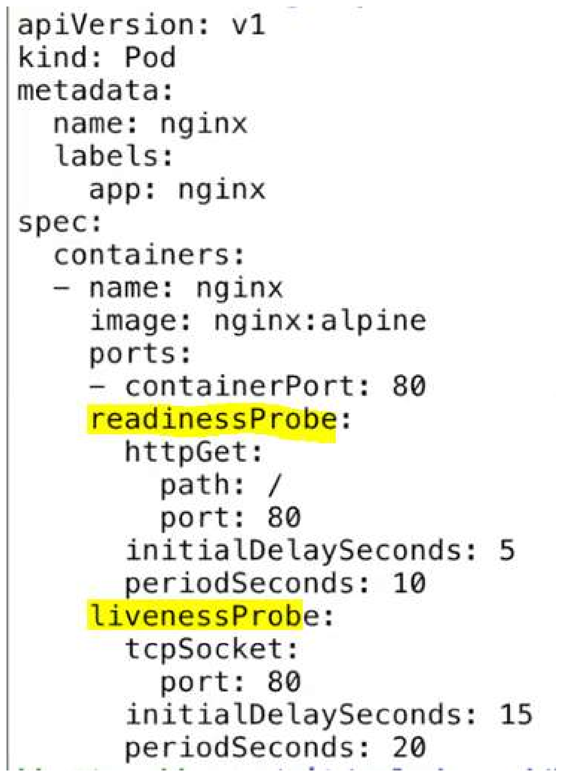
```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/liveness
    args:
    - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

Exmple 3: Check TCP port, you can check if a port is open



```
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
  - name: goproxy
    image: k8s.gcr.io/goproxy:0.1
    ports:
    - containerPort: 8080
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

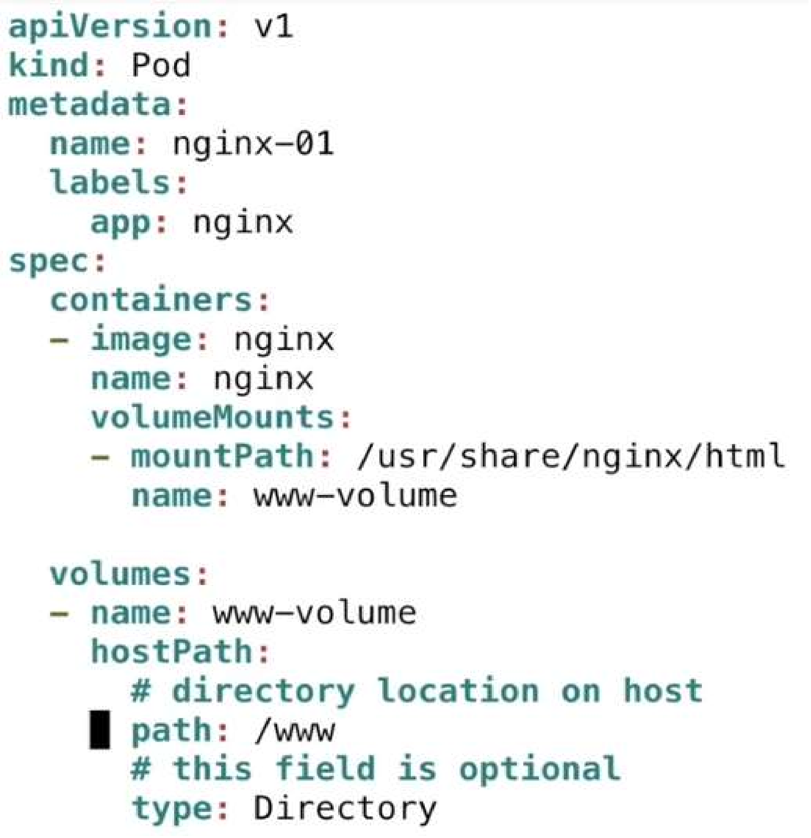
Example 4: In this case we run a nginx and check the port , if the port is open the liveness probe will be passed, for the readiness probe is a HTTP request to check if nginx is ready to receive request. If the liveness probe is Ok but the readiness probe is not OK kubernetes won't restart the pod.



```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:alpine
    ports:
    - containerPort: 80
    readinessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 80
      initialDelaySeconds: 15
      periodSeconds: 20
```

<table>
<tr>
<td>Volumes</td>
<td>

You can create many pods (nginx-01, nginx-02, etc) with a yml script and you can see the volume in the host that will be mounted in the pod:
I can check the directory in the host and upload a file there. This volumes is a "HostPath Volume"

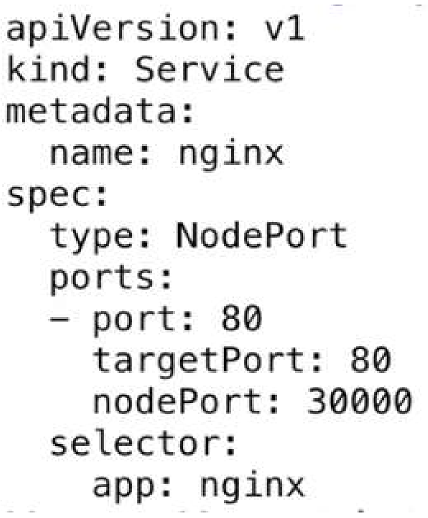


```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-01
  labels:
    app: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: www-volume

  volumes:
  - name: www-volume
    hostPath:
      # directory location on host
      path: /www
      # this field is optional
      type: Directory
```

</td>
<td>Volumes</td>
</tr>
<tr>
<td></td>
<td>

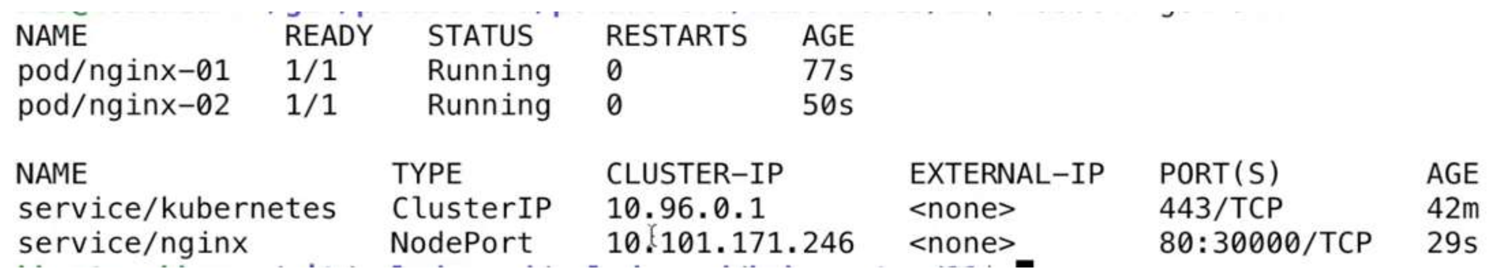
Now I can create a service to access to the pod:



```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
    nodePort: 30000
  selector:
    app: nginx
```

</td>
<td></td>
</tr>
</table>

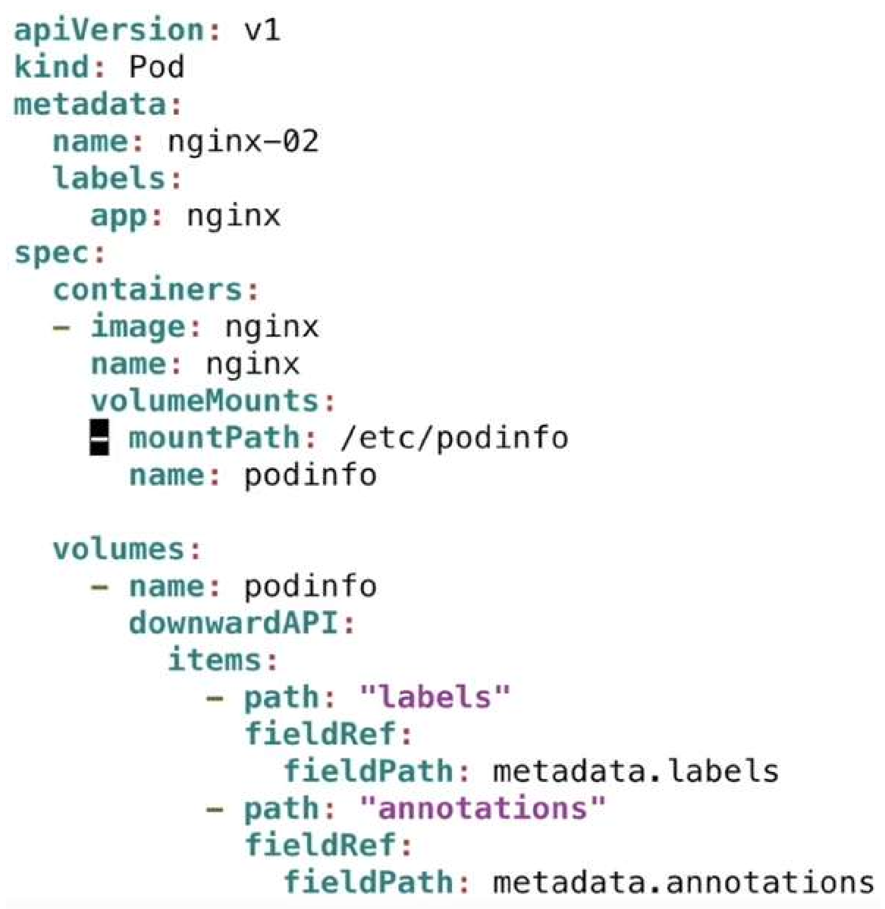
With a kubectl get all I can see the pods and service created:



```
NAME           READY   STATUS    RESTARTS   AGE
pod/nginx-01   1/1     Running   0          77s
pod/nginx-02   1/1     Running   0          50s

NAME                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
service/kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP        42m
service/nginx        NodePort    10.101.171.246   <none>        80:30000/TCP   29s
```

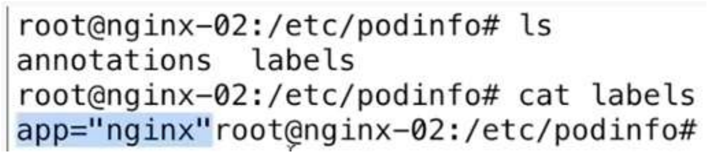
Other type of volume is "DownwardAPI" and you can use it to share data of the Kubernete API with the pods through files



```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-02
  labels:
    app: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /etc/podinfo
      name: podinfo

  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "labels"
            fieldRef:
              fieldPath: metadata.labels
          - path: "annotations"
            fieldRef:
              fieldPath: metadata.annotations
```

And if you enter to the pod created you can check:



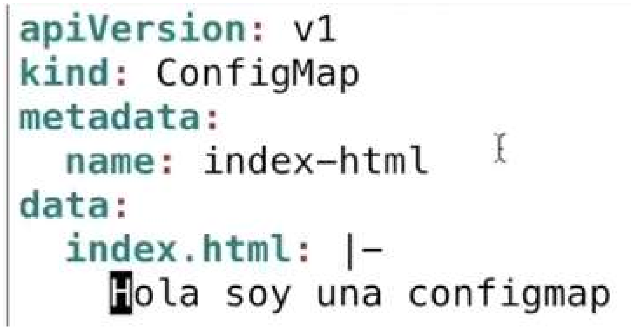
```
root@nginx-02:/etc/podinfo# ls
annotations  labels
root@nginx-02:/etc/podinfo# cat labels
app="nginx"root@nginx-02:/etc/podinfo#
```

Other type of volume is "ConfigMap" and you can use to send configuration or files to pods, if you can't send the configuration with an environment variable you can use ConfigMap.

First you apply the configMap and then the pods.

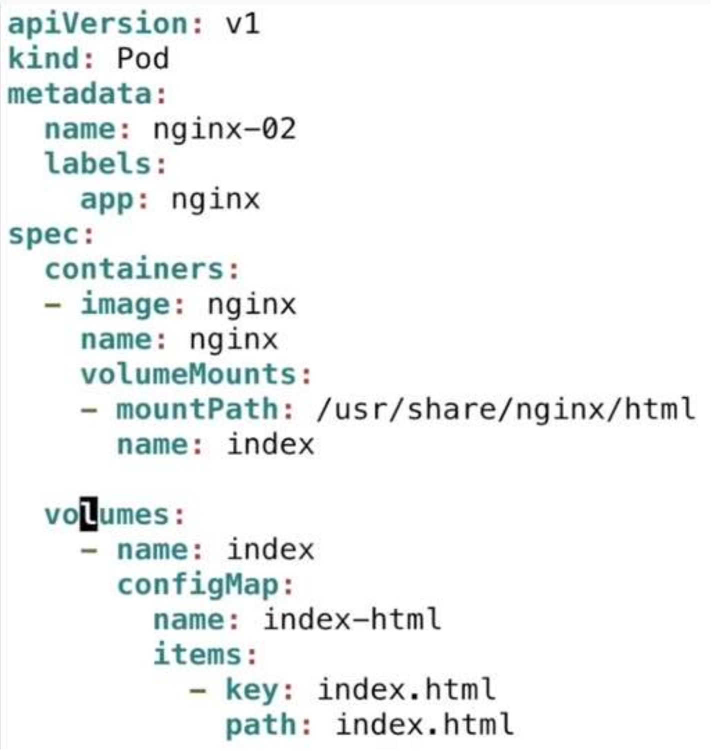
For this case you share the file index.html

This is the configMap:



```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: index-html
data:
  index.html: |-
    Hola soy una configmap
```

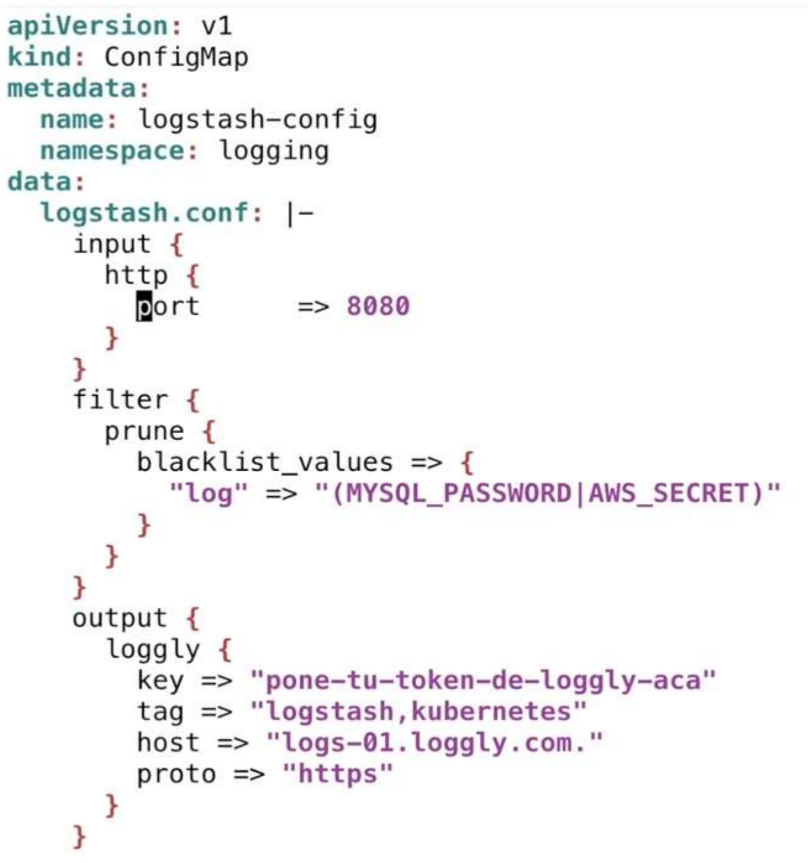
This is the pod:



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-02
  labels:
    app: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: index

  volumes:
    - name: index
      configMap:
        name: index-html
        items:
          - key: index.html
            path: index.html
```

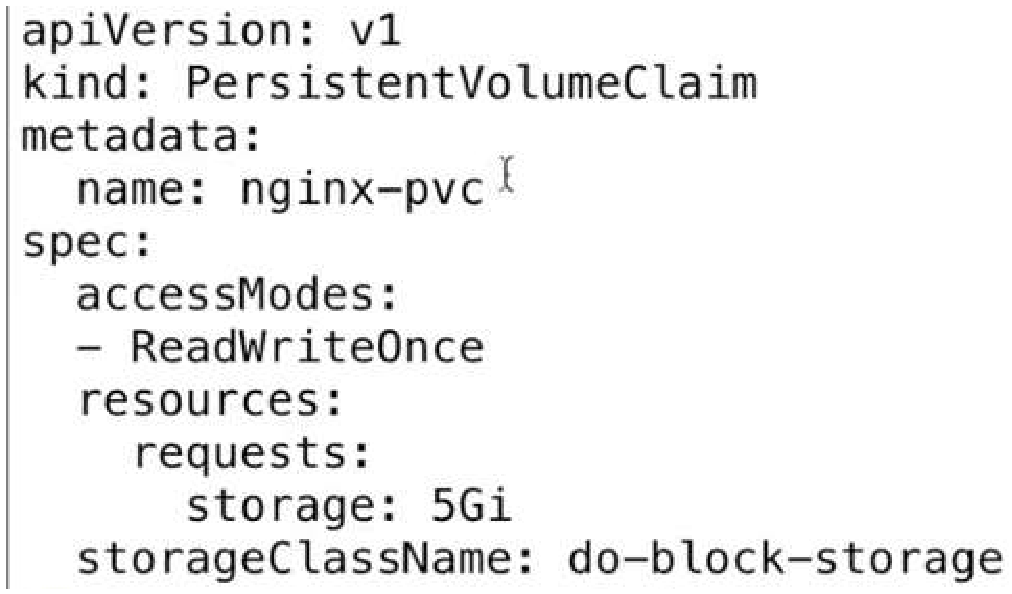
Also you can share a config file, for example:



```
apiVersion: v1
kind: ConfigMap
metadata:
  name: logstash-config
  namespace: logging
data:
  logstash.conf: |-
    input {
      http {
        port      => 8080
      }
    }
    filter {
      prune {
        blacklist_values => {
          "log" => "(MYSQL_PASSWORD|AWS_SECRET)"
        }
      }
    }
    output {
      loggly {
        key => "pone-tu-token-de-loggly-aca"
        tag => "logstash,kubernetes"
        host => "logs-01.loggly.com."
        proto => "https"
      }
    }
```

Other type of volume is "PersistentVolumeClaim" is when you want to create a volume in your cloud provider (AWS, Digital ocean, etc):

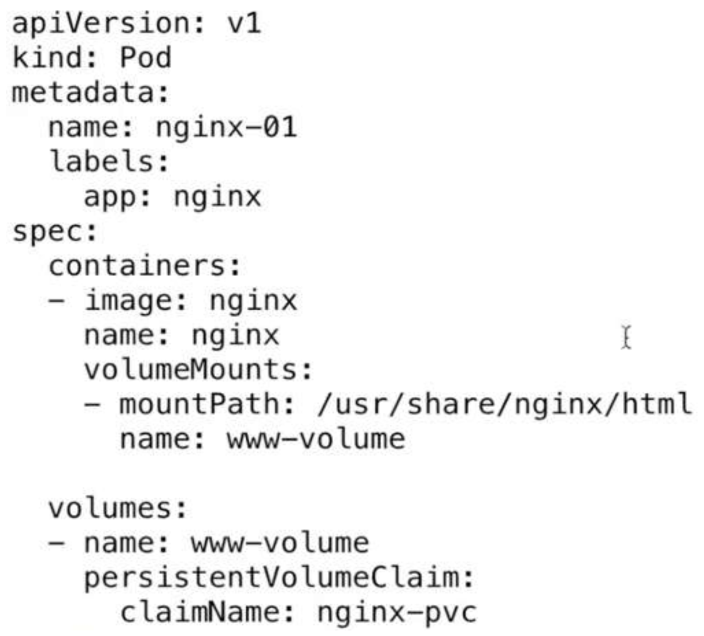
pvc yaml file:



```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nginx-pvc
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: do-block-storage
```

This is a volume of 5 gb with ReadWriteOnce permission and the storageClassName is the library used by Kubernetes to connect to the API of our cloud provider and create the volume (In our case is digital ocean)

You apply the pvc file and now you use the volume when you create a pod:

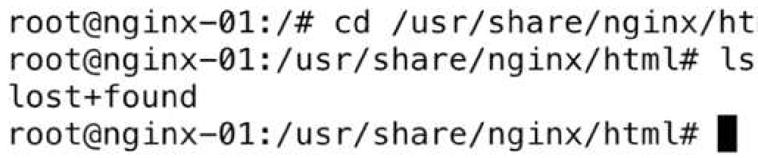


```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-01
  labels:
    app: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: www-volume

  volumes:
  - name: www-volume
    persistentVolumeClaim:
      claimName: nginx-pvc
```

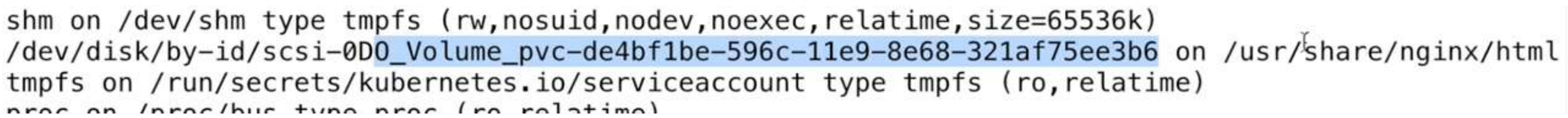
You can see that ClaimName is the name of the volume that we created in the pvc yaml

I f we enter to the pod we can see the directory lost+found that always this directory is created when is empty:



```
root@nginx-01:/# cd /usr/share/nginx/html/
root@nginx-01:/usr/share/nginx/html# ls
lost+found
root@nginx-01:/usr/share/nginx/html# █
```

And if you run the command mount you can see that the volume (digital ocean) is mounted :

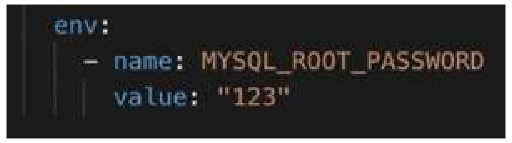


```
shm on /dev/shm type tmpfs (rw,nosuid,nodev,noexec,relatime,size=65536k)
/dev/disk/by-id/scsi-0DO_Volume_pvc-de4bf1be-596c-11e9-8e68-321af75ee3b6 on /usr/share/nginx/html
tmpfs on /run/secrets/kubernetes.io/serviceaccount type tmpfs (ro,relatime)
proc on /proc/bus type proc (ro,relatime)
```

with kubectl get vpc you can see all the volumes , to delete a volume you need to delete before the pods that are using the volume , and then you can delete the volume with the command kubectl delete pvc nameOfThePVC, In our case nginx-pvc

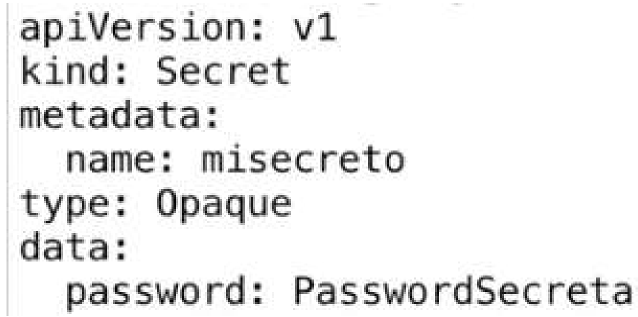
<table>
<tr>
<td>Environment variable</td>
<td>

You have differents ways to manage environment variables, you can hardcode the key and value in the deployment or pod yaml:



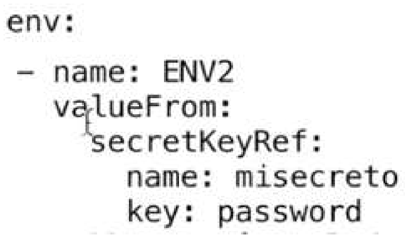
```
env:
  - name: MYSQL_ROOT_PASSWORD
    value: "123"
```

Or you can use a secret yaml, you run the secret yaml in the namespace (you can have a secret in each environment, so you have a different password in each environment):



```
apiVersion: v1
kind: Secret
metadata:
  name: misecreto
type: Opaque
data:
  password: PasswordSecreta
```

And in the deployment yaml or pod yaml:



```
env:
 - name: ENV2
   valueFrom:
     secretKeyRef:
       name: misecreto
       key: password
```

</td>
<td>Environment variable</td>
</tr>
</table>