

Commands / Topic	Description	Key word
kubectl get nodes kubectl get nodes -o wide	To check the workers running in the cluster. With -o wide I can see more information. (Internal IP, External IP and more.)	Check / Get / Nodes
kubectl get namespaces kubectl get ns	You can use namespaces to separete logically a cluster, in a cluster for example you can have a dev namespace, a test namespace and a production namespace. Also you use a namespace to limit differents things. To check the namespaces you use the command get. You can run the same commands as all the resources : get, logs, describe, delete, apply, etc	Check / 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 <div><div><pre>! namespace.yml x 1 apiVersion: v1 2 kind: Namespace 3 metadata: 4 name: curso-namespace</pre></div><div><pre>kind: Namespace apiVersion: v1 metadata: name: testing</pre></div></div>	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: <div><div><pre>! pod.yml x 1 apiVersion: v1 2 kind: Pod 3 metadata: 4 name: wildfly 5 spec: 6 containers: 7 - name: wildfly 8 image: jboss/wildfly 9</pre></div><div><pre>pod.yaml x apiVersion: v1 kind: Pod metadata: name: podtest2 spec: containers: - name: containertest image: nginx:alpine --- apiVersion: v1 kind: Pod metadata: name: podtest3 spec: containers: - name: containertest image: nginx:alpine</pre></div></div> If you want to use a image that you have in your local, you have to add a new line below image with: imagePullPolicy: IfNotPresent . If the image is not in your local , kubernetes will pull from dockerhub If you want to specify the namespace you have to add inside metadata: namespace: nameOfTheNamespace If you want to create many resources in the same yaml you have to separete the resources with: --- (This is not the best way, you can use replication)	Create / Pod

Limits	<p>You can limit Ram and CPU. Ram is in bytes, megabytes, gigabytes. And CPU in milicores, 1 CPU=1000m.</p> <p>Difference between request and limit: Request is the resources guarantee and limit is the possibility to increment resources temporarily.</p> <div><pre>apiVersion: v1 kind: Pod metadata: name: memory-demo spec: containers: - name: memory-demo-ctr image: polinux/stress resources: limits: memory: "200Mi" requests: memory: "100Mi" command: ["stress"] args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang", "1"]</pre></div> <div><pre>apiVersion: v1 kind: Pod metadata: name: cpu-demo spec: containers: - name: cpu-demo-ctr image: vish/stress resources: limits: cpu: "1000m" requests: cpu: "0.5" args: - -cpus - "2"</pre></div> <p>If the pod consume more memory than the limit kubernetes will restart the pod. (You can check with the command describe the event message)</p> <p>If the pod want to consume more CPU than the limit kubernetes won't restart the pod and Kubernetes gives to the pod just the limit and not more.</p> <p>If you assign in the request more than the capacity of the node the status of the pod will be in pending status all the time (You can check with the command describe the event message)</p> <p>Depends of the limits your pod has a QoS class: https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/</p>	Limits / Pod
kubectl get pods	To check the pods	Check / Get / Pod
kubectl get pod nameOfThePod -o yaml	To check the pod manifest, all the information with more detail	Check / Get / Pod
kubectl label pod nameOfThePod nameOfTheLabel=newValueForTheLabel --overwrite	To change a label in the pod.	Change
kubectl describe pod nameOfThePod	To check information about the pod	Check / Describe / Pod
kubectl delete -f pod.yml	To delete a pod, you can add the namespace or you can delete the pod with kubectl delete pod nameOfThePod	Delete / Pod
kubectl exec -it nameOfThePod -- /bin/bash kubectl exec -it nameOfThePod -- sh	To access to the pod and there you can run commands.	Access / Exec / Pod
kubectl port-forward <pod-name> 7000:<pod-port>	To access to the site configured in the pod when you are using virtual box and you cannot see the site in your pod through your pod IP because is in a different machine than your localhost, then you enter http://localhost:7000 and you can see the site in your pod.	Port
kubectl logs -f nameOfThePod	To check the logs of our pod	Logs / Pod

You create replicaset to create a number of pods and to maintain the number of pods in the quantity desired. You run the script with
kubectl apply -f yourYaml.yaml and you can check with kubectl get replicaset

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-test
  labels:
    app: rs-test
spec:
  replicas: 2
  selector:
    matchLabels:
      app: pod-label
  template:
    metadata:
      labels:
        app: pod-label
    spec:
      containers:
        - name: cont1
          image: python:3.6-alpine
          command: ['sh', '-c', 'echo cont1 > index.html && python -m http.server 8082']
        - name: cont2
          image: python:3.6-alpine
          command: ['sh', '-c', 'echo cont2 > index.html && python -m http.server 8083']
```

Replicaset information

Pod information

In the example, you can see a pod with 2 containers. And the replicaset mantain the number of pods in 2. The replicaset take in account just the pods with the same label.

The apiversion you can see running the command: kubectl **api-resources** , the column apigroup.
You can run the same commands as all the resources : get, logs, describe, delete, apply, etc
A rs can't update pods to change something.

Deployment

You use a deployment when you need to update a configuration in the pods or replicaset. You run a deployment with the apply command and get a deployment with kubectl get deployments.
Some examples:

deployment.yml ✕

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: wildfly-deployment
5  spec:
6    selector:
7      matchLabels:
8        app: wildfly
9    replicas: 1
10   template:
11     metadata:
12       labels:
13         app: wildfly
14     spec:
15       containers:
16         - name: wildfly
17           image: jboss/wildfly:15.0.0.Final
18           ports:
19             - containerPort: 8080
```

deploymentMariadb.yml ✕

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: mariadb-deployment
5  spec:
6    selector:
7      matchLabels:
8        app: mariadb
9    replicas: 1
10   template:
11     metadata:
12       labels:
13         app: mariadb
14     spec:
15       containers:
16         - name: mariadb
17           image: mariadb
18           ports:
19             - containerPort: 3306
20           env:
21             - name: MYSQL_ROOT_PASSWORD
22               value: "123"
```

```
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"
```

The apiversion you can see running the command: kubectl api-resources , the column apigroup.
As you can see, you can use environment variables and you can assign cpu and memory to each pod.
You can run the same commands as all the resources : get, logs, describe, delete, apply, etc

Deployment

Script and result: If you run this script you obtain this result:

```
apiVersion: apps/v1 #Deployment
kind: Deployment
metadata:
  name: deployment-test
  labels:
    app: front
spec: #Replica Set
  replicas: 3
  selector:
    matchLabels:
      app: front
  template: #Pod
    metadata:
      labels:
        app: front
    spec:
      containers:
      - name: nginx
        image: nginx:alpine
```

```
C:\Users\Gaston\Desktop\Gaston\Kubernetes\Scripts\Deployment>kubectl apply -f deployment.yaml
deployment.apps/deployment-test created

C:\Users\Gaston\Desktop\Gaston\Kubernetes\Scripts\Deployment>kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
deployment-test-64588d8b49-4xzhh    1/1     Running   0           9s
deployment-test-64588d8b49-njgrf    1/1     Running   0           9s
deployment-test-64588d8b49-pbd4t    1/1     Running   0           9s

C:\Users\Gaston\Desktop\Gaston\Kubernetes\Scripts\Deployment>kubectl get rs
NAME                                DESIRED   CURRENT   READY   AGE
deployment-test-64588d8b49          3         3         3       23s

C:\Users\Gaston\Desktop\Gaston\Kubernetes\Scripts\Deployment>kubectl get deployments
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
deployment-test  3/3      3            3           33s

C:\Users\Gaston\Desktop\Gaston\Kubernetes\Scripts\Deployment>kubectl rollout status deployment deployment-test
deployment "deployment-test" successfully rolled out
```

This command is to ckeck if the deployment worked fine: kubectl rollout **status** deployment nameOfTheDeployment

If you update a configuration and apply the deploy you can see that Kubernetes save the old version of the replicaset if you need to rollback, kubernets save 10 versions of a replicationset by default:

```
C:\Users\Gaston\Desktop\Gaston\Kubernetes\Scripts\Deployment>kubectl get rs
NAME                                DESIRED   CURRENT   READY   AGE
deployment-test-5d69f7646d          0         0         0       7m6s
deployment-test-64588d8b49          0         0         0       7m46s
deployment-test-cf7f9c68d           3         3         3       28s
```

You can run the command kubectl rollout history deployment nameOfTheDeployment and check the number of updates:

```
C:\Users\Gaston\Desktop\Gaston\Kubernetes\Scripts\Deployment>kubectl rollout history deployment deployment-test
deployment.apps/deployment-test
REVISION  CHANGE-CAUSE
2         <none>
3         <none>
4         <none>
```

You can see that the CHANGE-CAUSE is empty , you have many ways to complete this field:

#1 When you apply the script add the flag --record at the end, and you will see the command that you ran to deploy.

#2 You can add an annotation in the metadata of the deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    kubernetes.io/change-cause: "Changes port to 110"
  name: deployment-test
  labels:
```

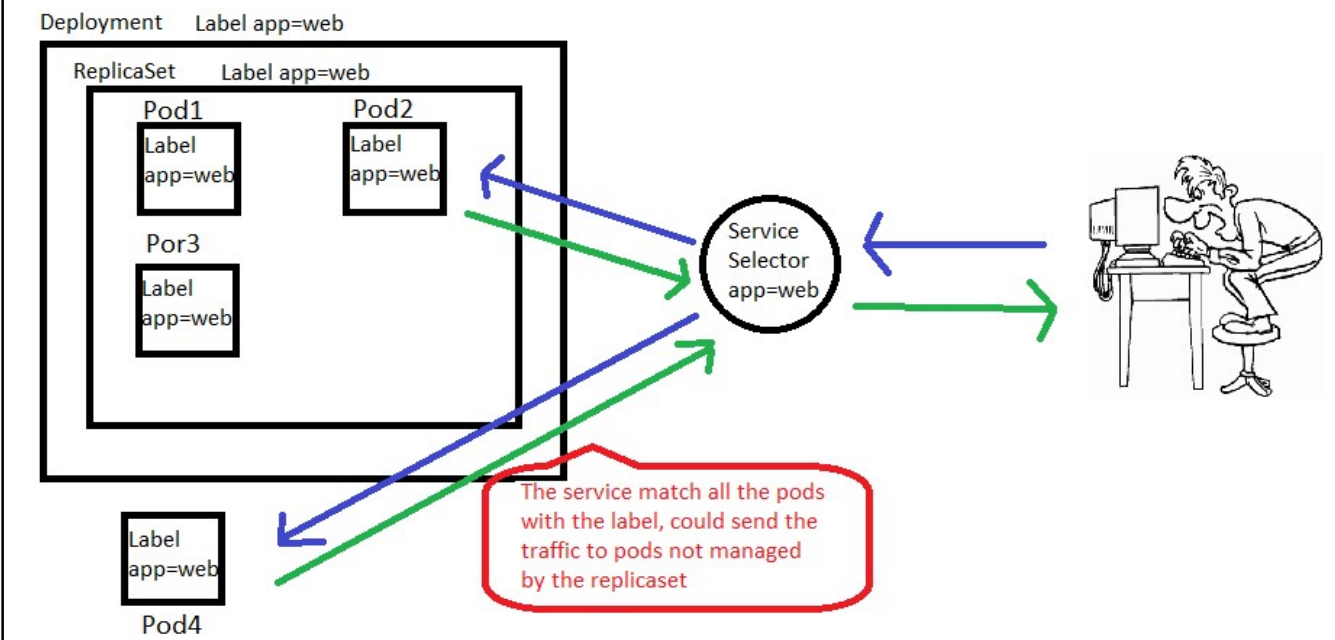
Also you can see in more details a specific revision adding a flag in the command: kubectl rollout **history** deployment nameOfTheDeployment --revision=numberOfRevision

To do a **roll back** you need to run the command kubectl rolout **undo** deployment nameOfTheDeploymenttt --to-revision=numberOfRevision

Service

You can use a service to access to your pods through different protocols (HTTP, TCP). The service match the pods with the same label and redirect the traffic. The service has an unique IP (and a DNS) , remember that pods are temporals and when a pod die , the new pod has a new IP. A service is like a load balancer.

If you want to call a service belonging to another namespace, you have to call the service in this way:
serviceName+namespaceName+svc.cluster.local .If you are in default namespace and you want to call the service my-service belonging to test. You can run from a pod belonging to default, the command: curl my-service.test.svc.cluster.local



Service

Some examples of a ClusterIP service, the service is clusterIP because it has only a virtual IP, the IP is internal cluster, we can access only from the cluster:

```
apiVersion: v1 #Servi
kind: Service
metadata:
  name: my-service
  labels:
    app: front
spec:
  selector:
    app: front
  ports:
    - protocol: TCP
      port: 8080 # Po
      targetPort: 80
```

```
! servicioMariadb.yml x
1 kind: Service
2 apiVersion: v1
3 metadata:
4   name: mariadb-service
5 spec:
6   selector:
7     app: mariadb
8   ports:
9     - protocol: TCP
10      port: 3306
11      targetPort: 3306
```

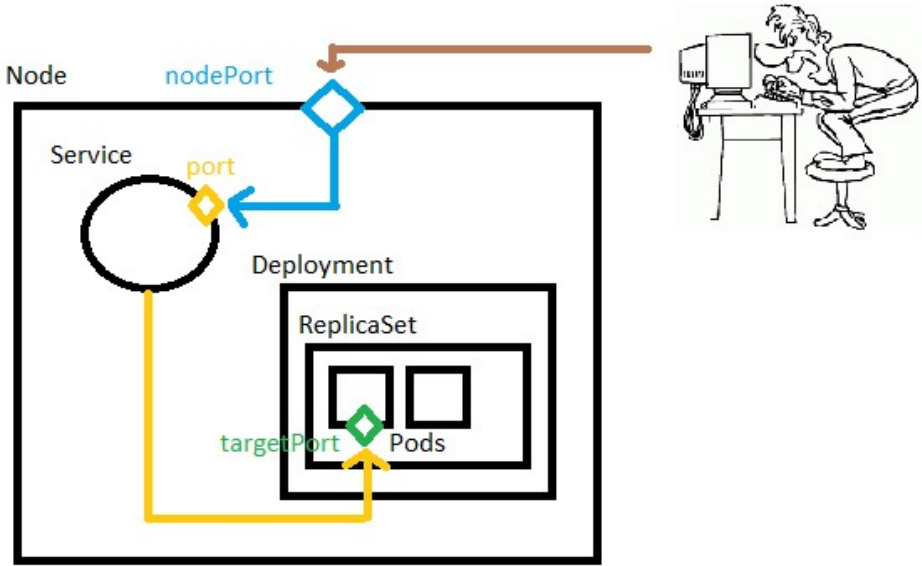
```
! servicioWildfly.yml x
1 kind: Service
2 apiVersion: v1
3 metadata:
4   name: wildfly-service
5 spec:
6   selector:
7     app: wildfly
8   ports:
9     - protocol: TCP
10      port: 8080
11      targetPort: 8080
```

```
C:\Users\Gaston\Desktop\Gaston\Kubernetes\Scripts\Services>kubectl get services -l app=front
NAME          TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)    AGE
my-service    ClusterIP   10.111.192.41 <none>         8080/TCP    46m
```

To create the service you apply the script, you can run the same commands as all the resources : get, logs, describe, delete, apply, etc

As you can see, the selector used is the same selector as in the deployment script. Port is the port in which the service will be listening and targetPort is the port of the pod that I will consume. If you don't specify the type of service, by default is ClusterIP

To check the service in your browser if you are using virtualbox: kubectl port-forward service/my-service 7000:8080 . Also you can enter to your cluster and check there with a curl IP:port

	<p>Other example: NodePort is a way to expose a service , so you can access from outside the cluster. NodePort create a clusterIP and also open a port to receive traffic.</p> <p>Port is the port in which the service will be listening</p> <p>TargetPort is the port of the pod that I will consume</p> <p>nodePort is the port in the node that point to the service. The port could be between 30000 and 32767. If you don't especify the port, kubernetes will select the port.</p> <p>This service will search the pods with the role wordpress and will send the traffc to these pods.</p> <pre>apiVersion: v1 kind: Service metadata: name: wordpress spec: type: NodePort ports: - port: 80 targetPort: 80 nodePort: 30000 selector: role: wordpress</pre> 	
	<p>Other example: LoadBalancer , with this, the service will conect to your cloud provider and create a load balancer. You can connect with AWS, Azure, GCP , etc. When you create a LoadBalancer you create a nodeport and a ClusterIP.</p> <pre>apiVersion: v1 kind: Service metadata: name: wordpress-lb spec: type: LoadBalancer ports: - protocol: TCP port: 80 targetPort: 80 name: http selector: role: wordpress</pre>	
kubectl get service kubectl get svc kubectl -n nameOfTheNamespace get svc	To check the services.	Check / Service
kubectl describe service nameOfTheService	To check information about the pod	Check / Describe / Service

HPA (horizontal pod autoscaling) Note: You run a script always in the same way.	<p>To control a metric , for example CPU:</p> <p>Pods with php-apache (check deployment yml) label will be created maintaining the CPU in 50%, min is 1 and max is 10.</p> <pre>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</pre>	HPA
Liveness and readiness probes	<p>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.</p> <p>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.</p> <pre>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</pre>	Liveness and readiness probes

	<p>Example 2: HTTP Get, if the file doesn't exist , the http get fail and Kubernetes restart the pod.</p> <pre>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</pre>	
	<p>Exmple 3: Check TCP port, you can check if a port is open</p> <pre>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</pre>	

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
```

Volumes	<p>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"</p> <pre>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</pre>	Volumes
	<p>Now I can create a service to access to the pod:</p> <pre>apiVersion: v1 kind: Service metadata: name: nginx spec: type: NodePort ports: - port: 80 targetPort: 80 nodePort: 30000 selector: app: nginx</pre>	

	<div>With a kubectl get all I can see the pods and service created:</div> <div><table><tr><th>NAME</th><th>READY</th><th>STATUS</th><th>RESTARTS</th><th>AGE</th></tr><tr><td>pod/nginx-01</td><td>1/1</td><td>Running</td><td>0</td><td>77s</td></tr><tr><td>pod/nginx-02</td><td>1/1</td><td>Running</td><td>0</td><td>50s</td></tr></table> <table><tr><th>NAME</th><th>TYPE</th><th>CLUSTER-IP</th><th>EXTERNAL-IP</th><th>PORT(S)</th><th>AGE</th></tr><tr><td>service/kubernetes</td><td>ClusterIP</td><td>10.96.0.1</td><td><none></td><td>443/TCP</td><td>42m</td></tr><tr><td>service/nginx</td><td>NodePort</td><td>10.101.171.246</td><td><none></td><td>80:30000/TCP</td><td>29s</td></tr></table></div>	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	
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																														
	<div>Other type of volume is "DownwardAPI" and you can use it to share data of the Kubernetes API with the pods through files</div> <div><pre>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</pre></div>																																		
	<div>And if you enter to the pod created you can check:</div> <div><pre>root@nginx-02:/etc/podinfo# ls annotations labels root@nginx-02:/etc/podinfo# cat labels app="nginx"root@nginx-02:/etc/podinfo#</pre></div>																																		

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:

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

This is the pod:

```
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)

	<p>You apply the pvc file and now you use the volume when you create a pod:</p> <pre>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</pre> <p>You can see that ClaimName is the name of the volume that we created in the pvc yaml</p> <p>If we enter to the pod we can see the directory lost+found that always this directory is created when is empty:</p> <pre>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# █</pre>	
	<p>And if you run the command mount you can see that the volume (digital ocean) is mounted :</p> <pre>shm on /dev/shm type tmpfs (rw,nosuid,nodev,noexec,relatime,size=65536k) /dev/disk/by-id/scsi-0D0_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)</pre> <p>with kubectI 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 kubectI delete pvc nameOfThePVC, In our case nginx-pvc</p>	

Environment variable

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: miscreto
type: Opaque
data:
  password: PasswordSecreta
```

And in the deployment yaml or pod yaml:

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

Environment variable