

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>! namespace.yml x</div><div>1 apiVersion: v1 2 kind: Namespace 3 metadata: 4   name: curso-namespace</div><div>kind: Namespace apiVersion: v1 metadata:   name: testing</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>! pod.yml x</div><div>1 apiVersion: v1 2 kind: Pod 3 metadata: 4   name: wildfly 5 spec: 6   containers: 7   - name: wildfly 8     image: jboss/wildfly 9</div><div>pod.yml x</div><div>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</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: <b>imagePullPolicy:</b> 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: <b>namespace:</b> 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 <b>memory</b> 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 <b>CPU</b> 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: <a href="https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/">https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/</a></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:
```

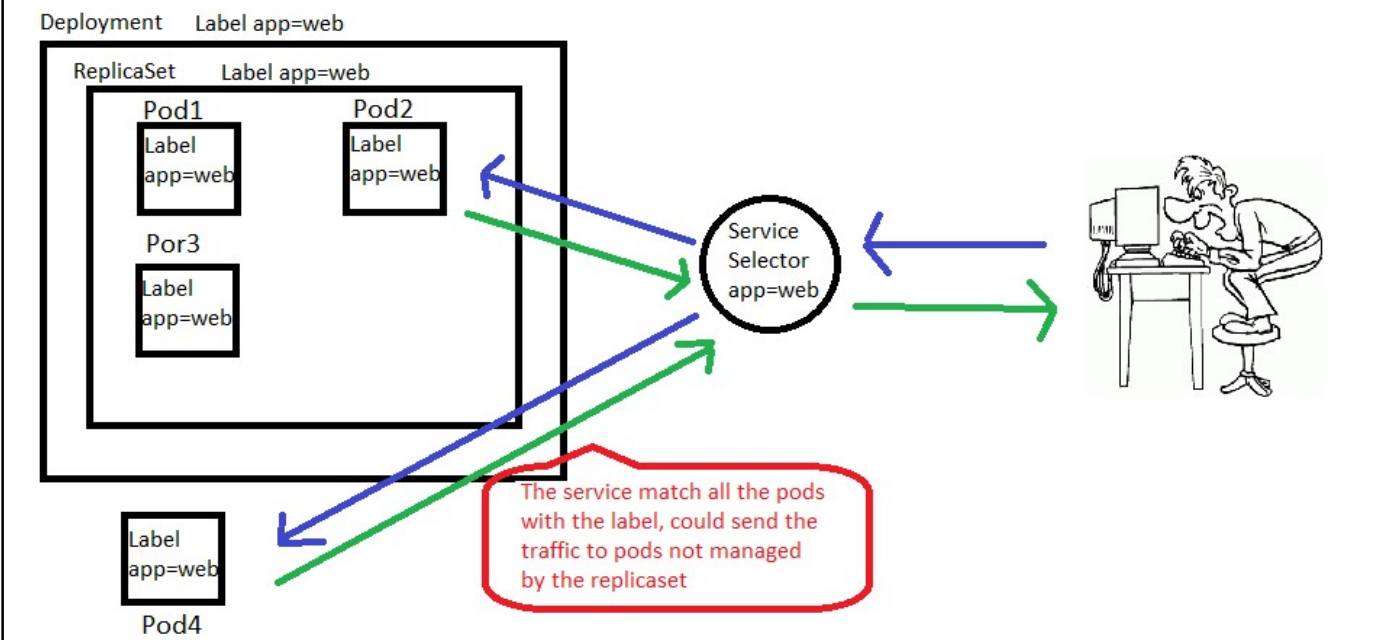
Also you can see in more details a specific revision adding a flag in the command: kubectl rollout **history** deployment nameOfTheDeployment --revision=numberOfRevision

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
```

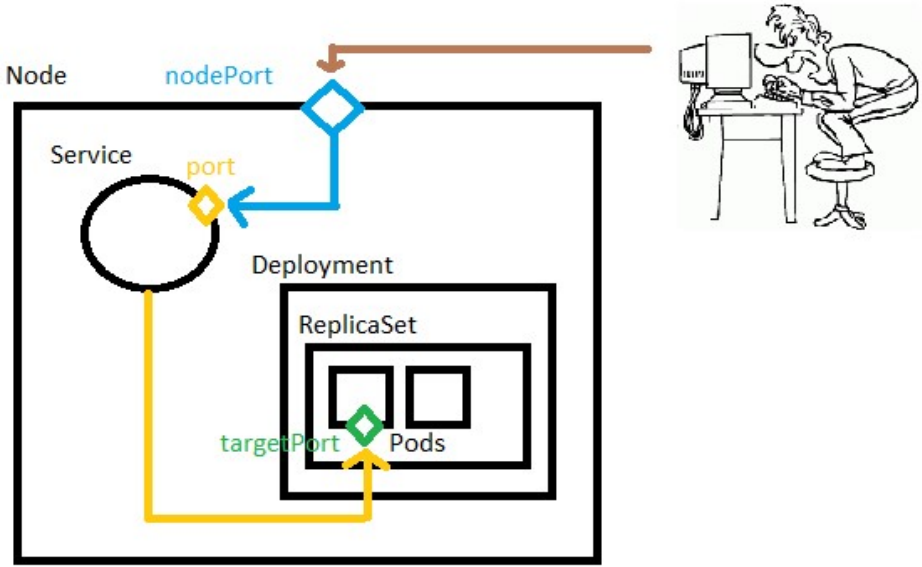
```
! servicioWildfly.yml x
1 kind: Service
2 apiVersion: v1
3 metadata:
4   name: wildfly-service
5 spec:
6   selector:
7     app: wildfly
8   ports:
9     - protocol: TCP
10      port: 8080
11      targetPort: 8080
```

```
C:\Users\Gaston\Desktop\Gaston\Kubernetes\Scripts\Services>kubect1 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: kubect1 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>	
<code>kubectl get service</code> <code>kubectl get svc</code> <code>kubectl -n nameOfTheNamespace get svc</code>	To check the services.	Check / Service
<code>kubectl describe service nameOfTheService</code>	To check information about the pod	Check / Describe / Service



limitRange

To control configurations or inject configurations at object level.

The limits in the limitrange will apply when we create a container without limits, this container will take the default values.  
If we create a container with limits, the limits of the container are your limits and not the limitrange limit.  
The limitrange only works in the namespace where it was created.

```
apiVersion: v1
kind: Namespace
metadata:
  name: dev
  labels:
    app: dev
---
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-cpu-limit-range
  namespace: dev #LimitRange in dev
spec:
  limits:
  - default:
      memory: 512Mi
      cpu: 1
    defaultRequest:
      memory: 256Mi
      cpu: 0.5
    type: Container
```

```
C:\Users\Gaston\Desktop\Gaston\Kubernetes\Scripts\LimitRange>kubectl get limitrange -n dev
NAME                               CREATED AT
mem-cpu-limit-range                2020-06-28T03:36:50Z
```

```
C:\Users\Gaston\Desktop\Gaston\Kubernetes\Scripts\LimitRange>kubectl describe limitrange mem-cpu-limit-range -n dev
Name:                               mem-cpu-limit-range
Namespace:                           dev
Type                                 Resource  Min  Max  Default Request  Default Limit  Max Limit/Request Ratio
----  -----  ---  ---  -
Container  cpu        -    -    500m             1               -
Container  memory     -    -    256Mi            512Mi           -
```

limitRange

Also you can define a min and max value:

```
apiVersion: v1
kind: Namespace
metadata:
  name: prod
  labels:
    name: prod
---
apiVersion: v1
kind: LimitRange
metadata:
  name: min-max
  namespace: prod
spec:
  limits:
    - max:
        memory: 1Gi
        cpu: 1
      min:
        memory: 100M
        cpu: 100m
      type: Container
```

Without error:

```
apiVersion: v1
kind: Pod
metadata:
  name: podtest3
  namespace: prod
  labels:
    app: backend
    env: dev
spec:
  containers:
    - name: cont1
      image: nginx:alpine
      resources:
        limits:
          memory: 500M
          cpu: 0.5
        requests:
          memory: 400M
          cpu: 0.3
```

With Error:

```
apiVersion: v1
kind: Pod
metadata:
  name: podtest3
  namespace: prod
  labels:
    app: backend
    env: dev
spec:
  containers:
    - name: cont1
      image: nginx:alpine
      resources:
        limits:
          memory: 2G
          cpu: 2
        requests:
          memory: 400M
          cpu: 0.3
```

And if you try to create a container with more cpu or memory than the max , an error appears, and the container won't be created.

```
Error from server (Forbidden): error when creating "min-max-limits.yaml": pods "podtest3" is forbidden: [maximum cpu usage per Container is 1, but limit is 2, maximum memory usage per Container is 1Gi, but limit is 2G]
```

If you try to create a container with a value of memory or cpu less than the min limits an error appears.

So, with limit ranges you control the min, max and a default value of resources in objects.

ResourceQuota

ResourceQuota is to apply limits at namespace level.

With limitrange you apply limits at container level , so you can apply a limit of 1 cpu , and if you create 200 containers you will use 200 cpus and you can obtain an error. With resourcequota you can limit at namespace level and you can say that the limit of the namespace is 50 CPU, so, you can't create 200 containers. The same with memory.

```
apiVersion: v1
kind: Namespace
metadata:
  name: uat
  labels:
    name: uat
---
apiVersion: v1
kind: ResourceQuota
metadata:
  name: res-quota
  namespace: uat
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory:
```

When you specify the deployment to create pods , you have to specify the memory and cpu (limit and request) in the pod section.

ResourceQuota

```
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: uat
  name: deployment-test
  labels:
    app: front
spec:
  replicas: 2
  selector:
    matchLabels:
      app: front
  template:
    metadata:
      labels:
        app: front
    spec:
      containers:
        - name: nginx
          image: nginx:alpine
          resources:
            requests:
              memory: 500M
              cpu: 500m
            limits:
              memory: 500M
              cpu: 500m
```

You can describe the namespace and check the resources used:

```
Resource Quotas
Name: res-quota
Resource      Used Hard
-----
limits.cpu    1     2
limits.memory 1G     2Gi
requests.cpu   1     1
requests.memory 1G     1Gi
```

With resourcequota you can also limit the number of pods in a namespace (In this case is the default namespace)

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pod-demo
spec:
  hard:
    pods: "3"
```

HPA (horizontal pod autoscaling )

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

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
```

HPA

Startup, Liveness and readiness probes	<p>Liveness probe says to Kubernetes if your pod is alive. This probe is executed every certain time. If the liveness file, kubernetes restart the pod.</p> <p>Readiness probe says to Kubernetes if your pod is ready to receive traffic. This probe is executed every certain time.</p> <p>Startup probe is used to wait until the application is deployed and configured and when application was uploaded the others probes (Readiness and Liveness) will be executed.</p> <p>You can probe with: Command, HTTP, TCP.</p> <p>Example 1: Exec, you can exec a command to know if there is a file. In this case we run <code>cat /tmp/healthy</code>, the liveness will fail because we remove the file after 30 seconds. During the first 30s the probe works fine.</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>	Startup, Liveness and readiness probes
--	---	--



	<p>Example 2: HTTP Get, if the file doesn't exist , the http get fail and Kubernetes restart the pod. The probe waits 3s and then test every 3s</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. Liveness probe checks every 20s , the probe waits 15s to run for the first time.</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. If the readiness probe is not ok, the traffic won't be sent to this 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

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
```

Volumes

Other type of volume is "DownwardAPI" and you can use it to share data of the Kuberne API with the pods through files , you can use environment variables (Explained below), but here you create 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. (This topic is explained in more details below)

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"
      }
    }
  }
```



	<p>PV and PVC with hostpath:</p> <p>You create a Persistent Volume and a Persistent Volume Claim and then assign the PVC with the deployment. Here you are using a hostpath, but this solution is not recommended if you are using more than 1 node.</p> <div><pre>apiVersion: v1 kind: PersistentVolume metadata:   name: test-pv   labels:     mysql: ready spec:   storageClassName: manual   capacity:     storage: 10Gi   accessModes:     - ReadWriteOnce   hostPath:     path: "/mysql" ---</pre><pre>apiVersion: v1 kind: PersistentVolumeClaim metadata:   name: test-pvc spec:   storageClassName: manual   accessModes:     - ReadWriteOnce   resources:     requests:       storage: 10Gi   selector:     matchLabels:       mysql: ready</pre><pre>apiVersion: apps/v1 kind: Deployment metadata:   name: mysql   labels:     app: mysql spec:   replicas: 1   selector:     matchLabels:       app: mysql   template:     metadata:       labels:         app: mysql     spec:       containers:         - name: mysql           image: mysql:5.7           env:             - name: MYSQL_ROOT_PASSWORD               value: "12345678"           volumeMounts:             - mountPath: "/var/lib/mysql"               name: vol-mysql       volumes:         - name: vol-mysql           persistentVolumeClaim:             claimName: test-pvc</pre></div> <p>/mysql in the host is related with the path /var/lib/mysql in the pods. PV and PVC are related through labels and Selectors. In the PV and PVC the storageClassName manual means that you will create the PV for the PVC, with dynamic provisioning you don't need to specify the PV, you only specify the PVC and the PV is created automatically. You can check this below (PVC in the cloud)</p>	
	<p>Other type of volume is "PersistentVolumeClaim (PVC)" in the cloud, is when you want to create a volume in your cloud provider (AWS, Digital ocean, etc):</p> <p>pvc yaml file:</p> <pre>apiVersion: v1 kind: PersistentVolumeClaim metadata:   name: nginx-pvc spec:   accessModes:     - ReadWriteOnce   resources:     requests:       storage: 5Gi   storageClassName: do-block-storage</pre> <p>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>	

	<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, this directory always is created when the volume 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 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</p>	

--	--	--

Environment variable

You have different 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"
```

Also you can use the values that you can find in the description (kubectl describe ):

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-envvars-fieldref
spec:
  containers:
  - name: test-container
    image: nginx:alpine
    env:
    - name: MY_NODE_NAME
      valueFrom:
        fieldRef:
          fieldPath: spec.nodeName
    - name: MY_POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    - name: MY_POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace
    - name: MY_POD_IP
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
```

If I run a ssh to the pod I can see the environment variable,  
echo \$MY\_POD\_NAME

Environment variable

ConfigMap

ConfigMap is used to separate the configurations from your pod, and you don't need to hardcode your configuration in your pod. The pod takes the configuration from the ConfigMap.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-map-name
  labels:
    app: front
data:
  nginx: |
    server {
      listen      80;
      listen  [::]:80;
      server_name localhost;

      location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
      }

      error_page  500 502 503 504  /50x.html;
      location = /50x.html {
        root   /usr/share/nginx/html;
      }
    }
```

```
apiVersion: apps/v1 #Deployment
kind: Deployment
metadata:
  name: deployment-test
  labels:
    app: front
spec: #Replica Set
  replicas: 1
  selector:
    matchLabels:
      app: front
  template: #Pod
    metadata:
      labels:
        app: front
    spec:
      containers:
        - name: nginx
          image: nginx:alpine
          volumeMounts:
            - name: nginx-volume # Reference of the volume
              mountPath: /etc/nginx/conf.d #Folder in the container
          volumes: #Volume
            - name: nginx-volume
              configMap:
                name: config-map-name # Reference of the configMap
                items:
                  - key: nginx # Key in the configMap
                    path: default.conf #Name of the config in the folder
```

I create the configMap with the key nginx, the value of the key is a configuration.  
You reference the configMap when you create the volume. And you reference the volume when you create the container.  
If you don't set the name of the file in the container (default.conf in our case), the file will take the name of the key, in our case "nginx", you can call the key as default.conf and also works.

ConfigMap

After apply both yamls you can check in the pod if the config was applied.

```
C:\Users\Gaston\Desktop\Gaston\Kubernetes\Scripts\ConfigMap>kubectl get pods
NAME                                READY  STATUS   RESTARTS  AGE
deployment-test-7c56485774-w85wz    1/1    Running  0          5m44s

C:\Users\Gaston\Desktop\Gaston\Kubernetes\Scripts\ConfigMap>kubectl exec -it deployment-test-7c56485774-w85wz sh
/ # cd ..
/ # cat etc/nginx/conf.d/default.conf
server {
  listen      80;
  listen  [::]:80;
  server_name localhost;

  location / {
    root   /usr/share/nginx/html;
    index  index.html index.htm;
  }

  error_page  500 502 503 504  /50x.html;
  location = /50x.html {
    root   /usr/share/nginx/html;
  }
}
```

And you can check the description of the configMap: `kubectl describe cm config-map-name`

```
C:\Users\Gaston\Desktop\Gaston\Kubernetes\Scripts\ConfigMap>kubectl describe cm config-map-name
Name:         config-map-name
Namespace:    default
Labels:       app=front
Annotations:  kubectl.kubernetes.io/last-applied-configuration:
              {"apiVersion":"v1","data":{"nginx":"server {\n  listen      80;\n  listen  [::]:80;\n  server_name  localhost;\n\n  location / {\n    r...

Data
====
nginx:
----
server {
  listen      80;
  listen  [::]:80;
  server_name  localhost;

  location / {
    root    /usr/share/nginx/html;
    index  index.html index.htm;
  }

  error_page  500 502 503 504  /50x.html;
  location = /50x.html {
    root    /usr/share/nginx/html;
  }
}

Events:  <none>
```

You can use environment variables in a configMap too and you define the variables in the pod script.

Then you can use the environment variables in the pod.

Also you can use volumes and environment variables together, and use the environment variables inside the config that you send to the pod using a volume.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: vars
  labels:
    app: front
data:
  db_host: dev.host.local
  db_user: dev_user
```

```
containers:
- name: nginx
  image: nginx:alpine
  env:
    - name: DB_HOST
      valueFrom:
        configMapKeyRef:
          name: vars
          key: db host
    - name: DB_USER
      valueFrom:
        configMapKeyRef:
          name: vars
          key: db_user
```



Secrets	<p>You can use Secret to save sensitive data.</p> <p>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):</p> <div><div>Here you write your username and password encoded.</div><pre>apiVersion: v1 kind: Secret metadata:   name: mysecret type: Opaque data:   username: YWRtaW4=   password: MWYyZDFlMmU2N2Rm</pre></div> <div><div>Here you write your username and password decoded.</div><pre>apiVersion: v1 kind: Secret metadata:   name: mysecret2 type: Opaque stringData:   username: usertest   password: passwordtest</pre></div> <p>When you run the command describe to a secret, kubernetes doesn't show you the content of the secret.</p> <p>If you use stringdata, kubernetes encode your values. The encode and decode is in base64. Check with kubectl get your secret and you can see the information.</p> <p>You can use apps like envsubst to create a new yaml script and replace the values of the variables with real values, you have to define your environment variables in your computer and then create a new file with the real values of the variables, you apply the new file and the you can remove it.</p> <p>If I run the command: envsubst &lt; mysecrete.yaml &gt; tmp.yaml</p> <div><pre>\$ export USER=Gaston \$ export PASSWORD=GastonPass \$ echo \$USER \$PASSWORD Gaston GastonPass</pre></div> <div><pre>apiVersion: v1 kind: Secret metadata:   name: mysecret4 type: Opaque stringData:   username: \$USER   password: \$PASSWORD</pre></div> <p>In your tmp.yaml you will have your yaml with the correct values</p>	Secrets
	<p>You can inject your secrets with volumes or with environment variables. This process is similar than ConfigMap</p> <div><pre>apiVersion: v1 kind: Secret metadata:   name: secret1 type: Opaque stringData:   username: admin   password: "12345678" ---</pre></div> <p>You can see in the file /opt/user.txt the value admin</p> <div><pre>apiVersion: v1 kind: Pod metadata:   name: mypod spec:   containers:     - name: mypod       image: nginx:alpine       volumeMounts:         - name: test           mountPath: "/opt"           readOnly: true       volumes:         - name: test           secret:             secretName: secret1             items:               - key: username                 path: user.txt</pre></div> <p>With environment variables and you can use them in your pod:</p> <div><pre>apiVersion: v1 kind: Pod metadata:   name: mypod spec:   containers:     - name: mypod       image: nginx:alpine       env:         - name: USERTEST           valueFrom:             secretKeyRef:               name: secret1               key: username         - name: PASSWORDTEST           valueFrom:             secretKeyRef:               name: secret1               key: password</pre></div>	