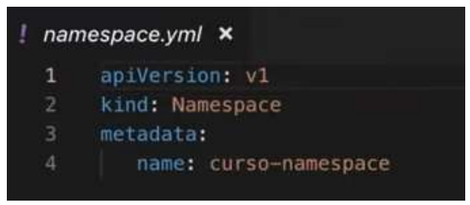
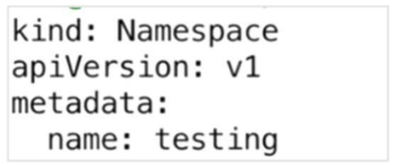
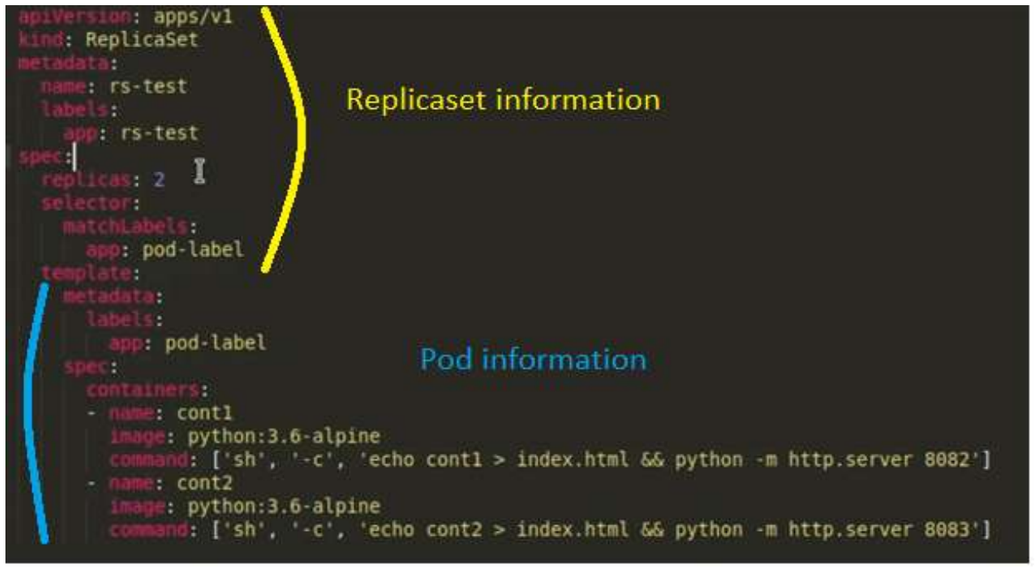
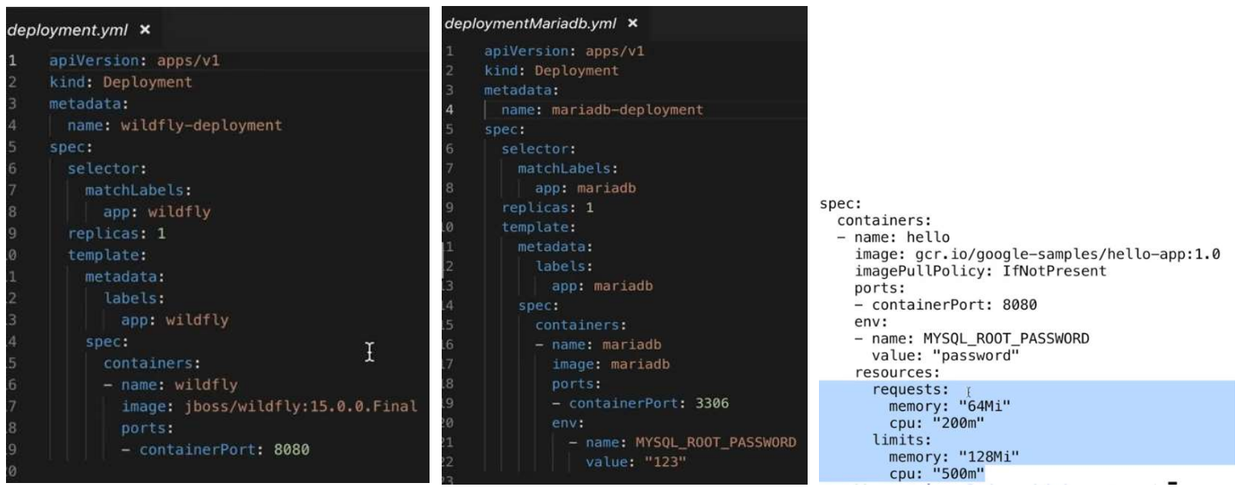
Commands / Topic	Description	Key word
kubectl get nodes kubectl get nodes -o wide	To check the workers running in the cluster. With -o wide I can see more information. (Internal IP, External IP and more.)	Check / Get / Nodes
kubectl get namespaces kubectl get ns	You can use namespaces to separete logically 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  ```yaml apiVersion: v1 kind: Namespace metadata: name: curso-namespace ```  ```yaml 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: ```yaml apiVersion: v1 kind: Pod metadata: name: wildfly spec: containers: - name: wildfly image: jboss/wildfly ```  ```yaml 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 ``` 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	You can limit Ram and CPU. Ram is in bytes, megabytes, gigabytes. And CPU in milicores, 1 CPU=1000m. Difference between request and limit: Request is the resources guarantee and limit is the possibility to increment resources temporarily. ```yaml 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"] ``` ```yaml 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" ``` If the pod consume more **memory** than the limit kubernetes will restart the pod. (You can check with the command describe the event message) 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. 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) Depends of the limits your pod has a QoS class: https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/	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 configureted 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

ReplicaSet	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 replicasets  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.	Replicasets

```yaml
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']
```

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

```yaml
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:

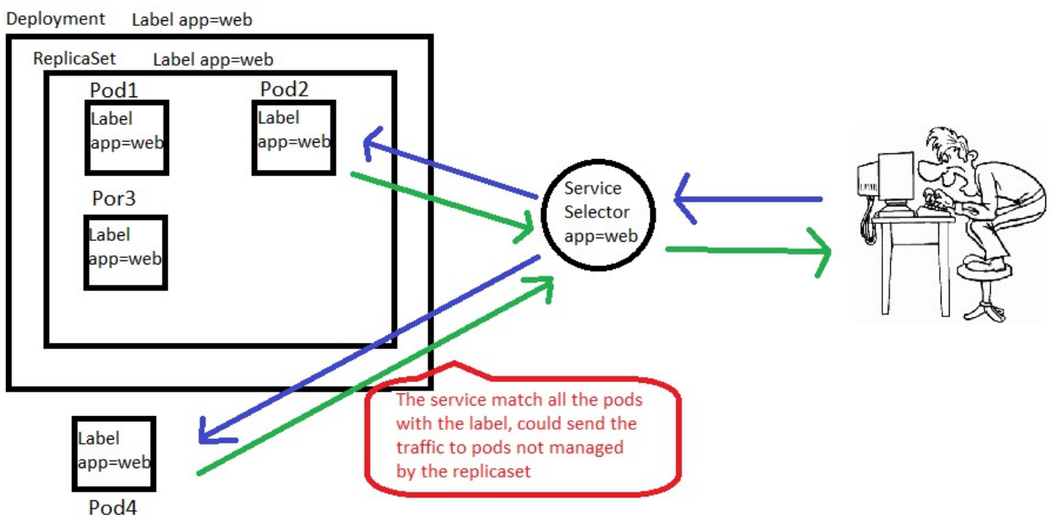
```
C:\Users\Gaston\Desktop\Gaston\Kubernetes\Scripts\Deployment>kubectl rollout history deployment deployment-test
deployment.apps/deployment-test
REVISION  CHANGE-CAUSE
2         <none>
3         <none>
4         <none>
```

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:

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

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 nameOfTheDeploymentt --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:

```yaml
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

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

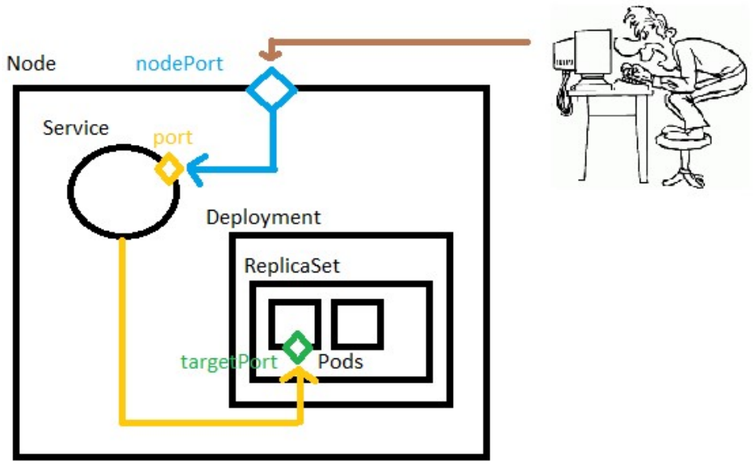
servicioWildfly.yml

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

```
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 especify 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

	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. Port is the port in which the service will be listening TargetPort is the port of the pod that I will consume 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. This service will search the pods with the role wordpress and will send the traffc to these pods. `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. You can connect with AWS, Azure, GCP , etc. When you create a LoadBalancer you create a nodeport and a ClusterIP. `apiVersion: v1` `kind: Service` `metadata:` `  name: wordpress-lb` `spec:` `  type: LoadBalancer` `  ports:` `  - protocol: TCP` `    port: 80` `    targetPort: 80` `    name: http` `  selector:` `    role: wordpress`	
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

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. ```yaml 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:

```yaml
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:

```yaml
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:

```yaml
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

```yaml
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 usded:

```
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)

```yaml
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	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. Readiness probe says to Kubernetes if your pod is ready to receive traffic. This probe is executed every certain time. 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. You can probe with: Command, HTTP, TCP. 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. During the first 30s the probe works fine. `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`	Startup, Liveness and readiness probes

	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 `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. Liveness probe checks every 20s , the probe waits 15s to run for the first time. `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. 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
	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`	

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

```yaml
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:

```yaml
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:

```yaml
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	

Environment variable	You have differents ways to manage environment variables, you can hardcode the key and value in the deployment or pod yaml: ```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): ```yaml apiVersion: v1 kind: Secret metadata: name: misecreto type: Opaque data: password: PasswordSecreta ``` And in the deployment yaml or pod yaml: ```yaml env: - name: ENV2 valueFrom: secretKeyRef: name: misecreto key: password ``` 	Environment variable