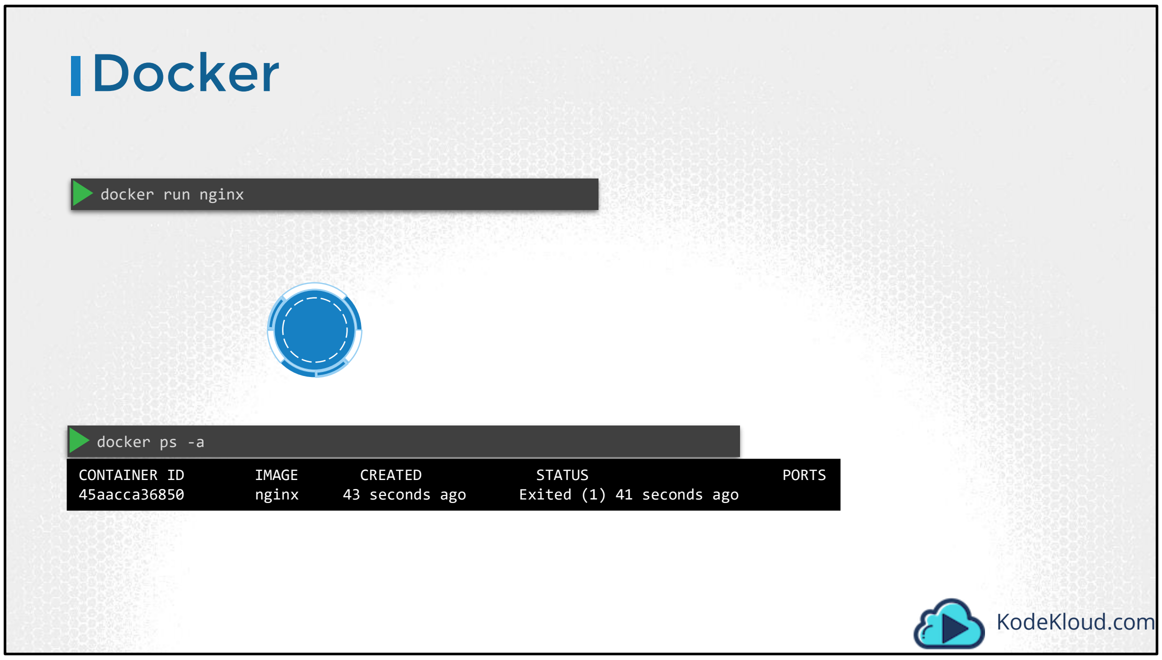
Liveness Probes

KodeKloud.com

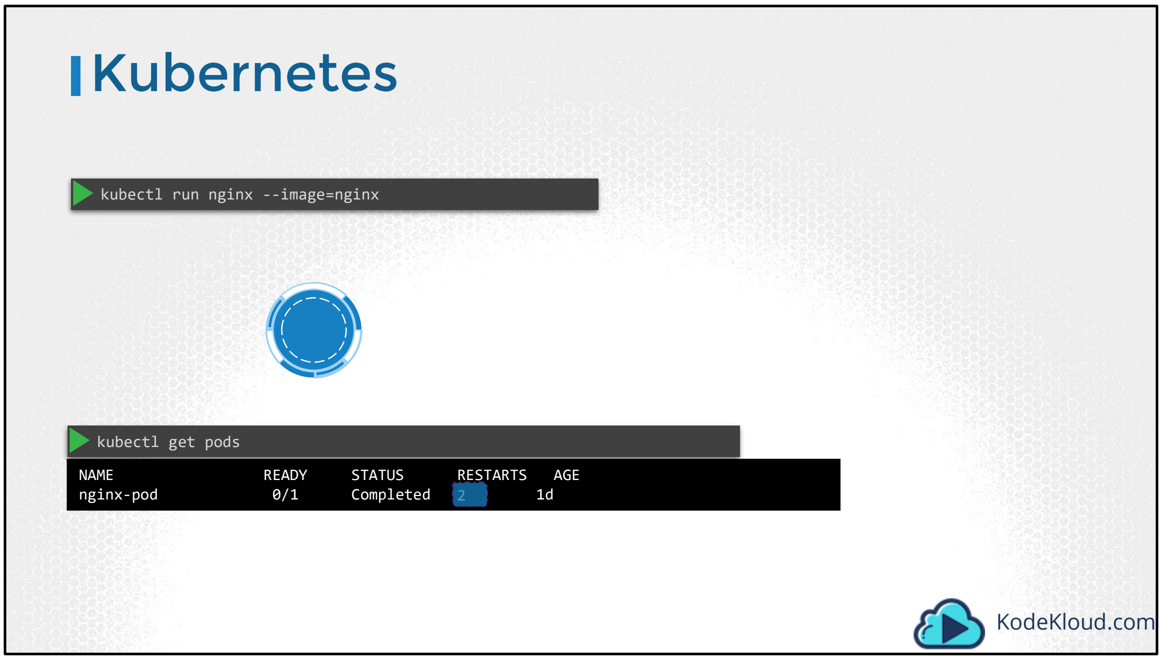


Docker

```
docker run nginx
```

```
docker ps -a
CONTAINER ID        IMAGE       CREATED              STATUS                      PORTS
45aacca36850        nginx       43 seconds ago       Exited (1) 41 seconds ago
```

Let's start from the basics. You run an image of NGINX using docker and it starts to serve users. For some reason the web server crashes and the nginx process exits. The container exits as well. And you can see the status of the container when you run the docker ps command. Since docker is not an orchestration engine, the container continues to stay dead and deny services to users, until you manually create a new container.



Kubernetes

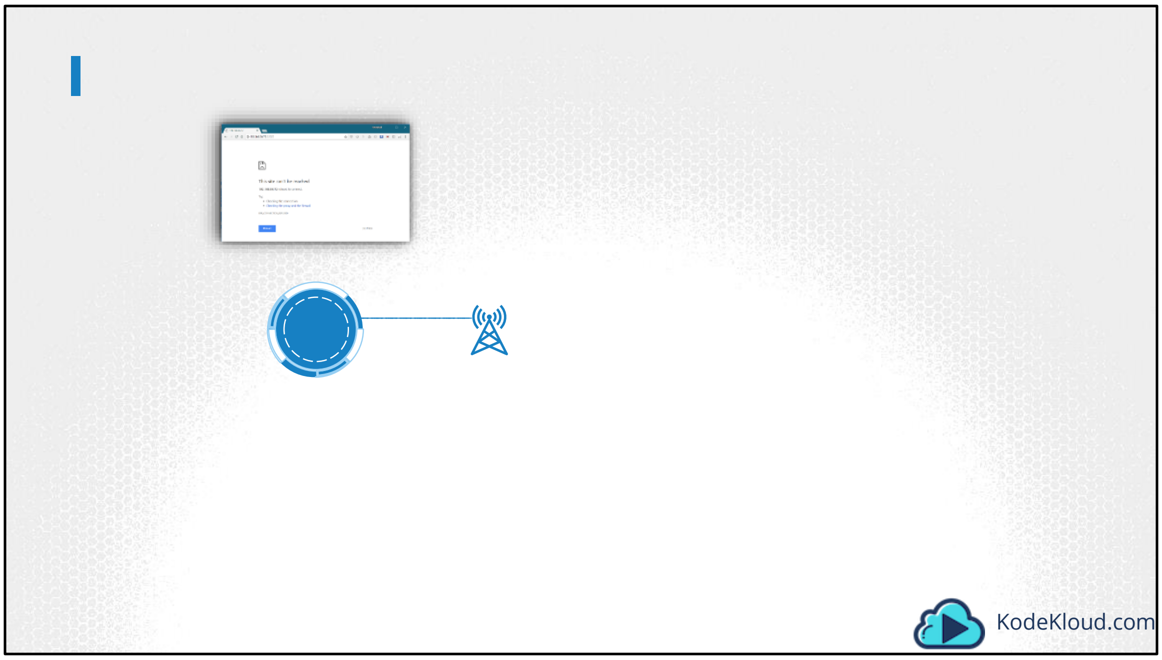
```
kubectl run nginx --image=nginx
```

```
kubectl get pods
```

```
NAME                 READY      STATUS       RESTARTS    AGE
nginx-pod             0/1       Completed    2           1d
```

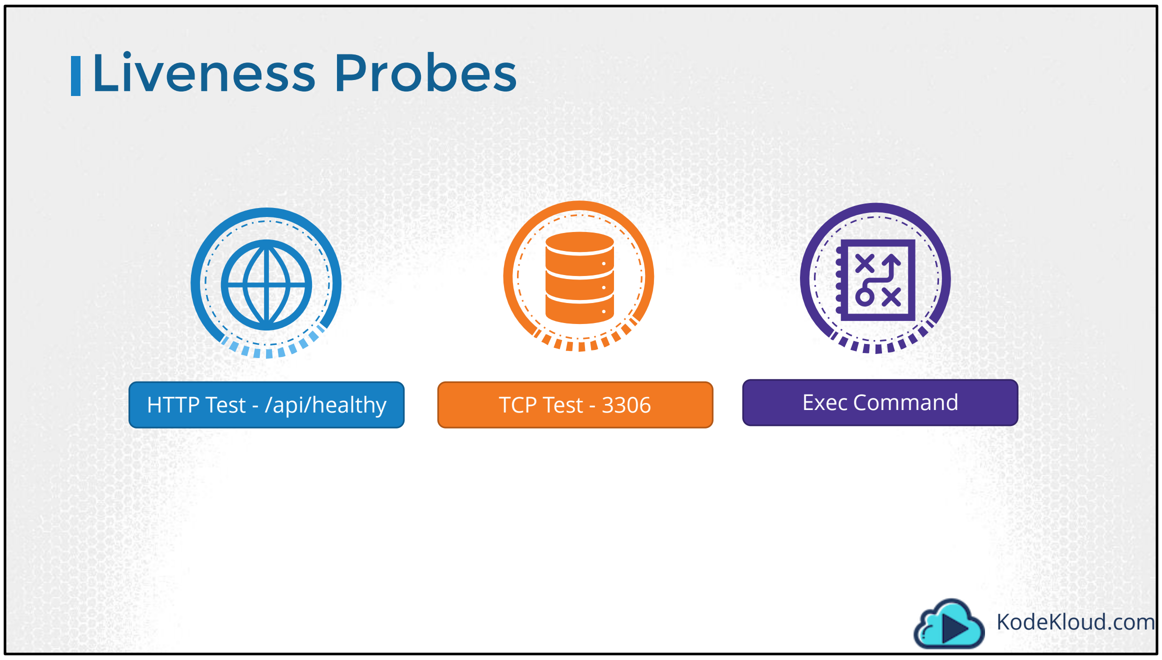
KodeKloud.com

Enter Kubernetes Orchestration. You run the same web application with kubernetes. Every time the application crashes, kubernetes makes an attempt to restart the container to restore service to users. You can see the count of restarts increase in the output of kubectl get pods command. Now this works just fine.



However, what if the application is not really working but the container continues to stay alive? Say for example, due to a bug in the code, the application is stuck in an infinite loop. As far as kubernetes is concerned, the container is up, so the application is assumed to be up. But the users hitting the container are not served. In that case, the container needs to be restarted, or destroyed and a new container is to be brought up.   That is where the liveness probe can help us. A liveness probe can be configured on the container to periodically test whether the application within the container is actually healthy. If the test fails, the container is considered unhealthy and is destroyed and recreated.

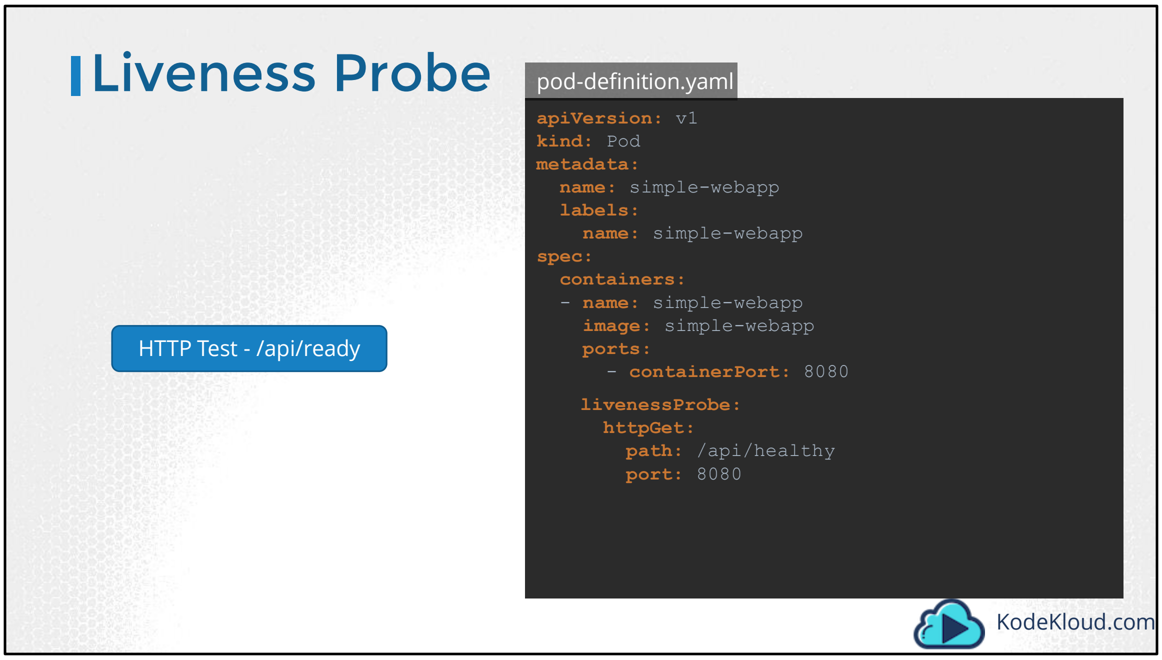
But again, as a developer, you get to define what it means for an application to be healthy.



Liveness Probes

HTTP Test - /api/healthy
TCP Test - 3306
Exec Command
KodeKloud.com

In case of a web application it could be when the API server is up and running. In case of database, you may test to see if a particular TCP socket is listening. Or You may simply execute a command to perform a test.



Liveness Probe

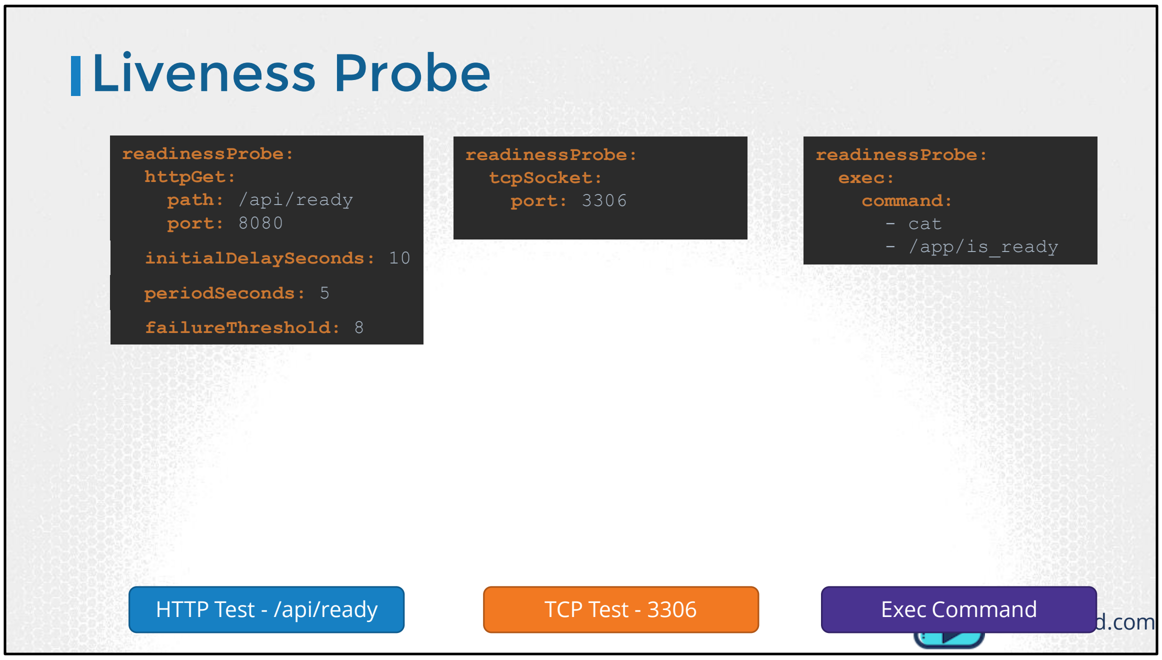
HTTP Test - /api/ready

pod-definition.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp
  labels:
    name: simple-webapp
spec:
  containers:
  - name: simple-webapp
    image: simple-webapp
    ports:
      - containerPort: 8080

    livenessProbe:
      httpGet:
        path: /api/healthy
        port: 8080
```

KodeKloud.com

The liveness probe is configured in the pod definition file as you did with the readinessProbe. Except here you use liveness instead of readiness.



Liveness Probe

```
readinessProbe:
  httpGet:
    path: /api/ready
    port: 8080

  initialDelaySeconds: 10

  periodSeconds: 5

  failureThreshold: 8
```

```
readinessProbe:
  tcpSocket:
    port: 3306
```

```
readinessProbe:
  exec:
    command:
      - cat
      - /app/is_ready
```

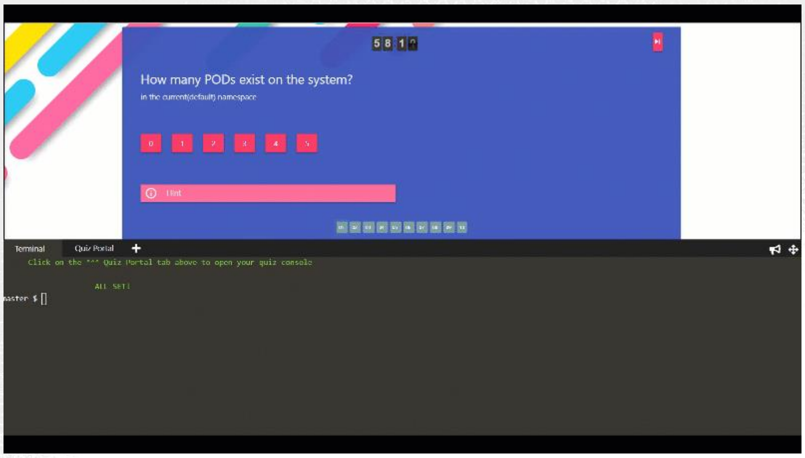
Similar to readiness probe you have httpGet option for apis, tcpSocker for ports and exec for commands. As well as additional options like initialDelay before the test is run, periodSeconds to define the frequency and success and failure thresholds.

Well that's it for this lecture. Head over and practice what you learned in the coding exercises section.

We have some fun and challenging exercises were you will be required to configure probes as well as troubleshoot and fix issues with existing probes.

See you in the next lecture.

Practice Test


How many PODs exist on the system?

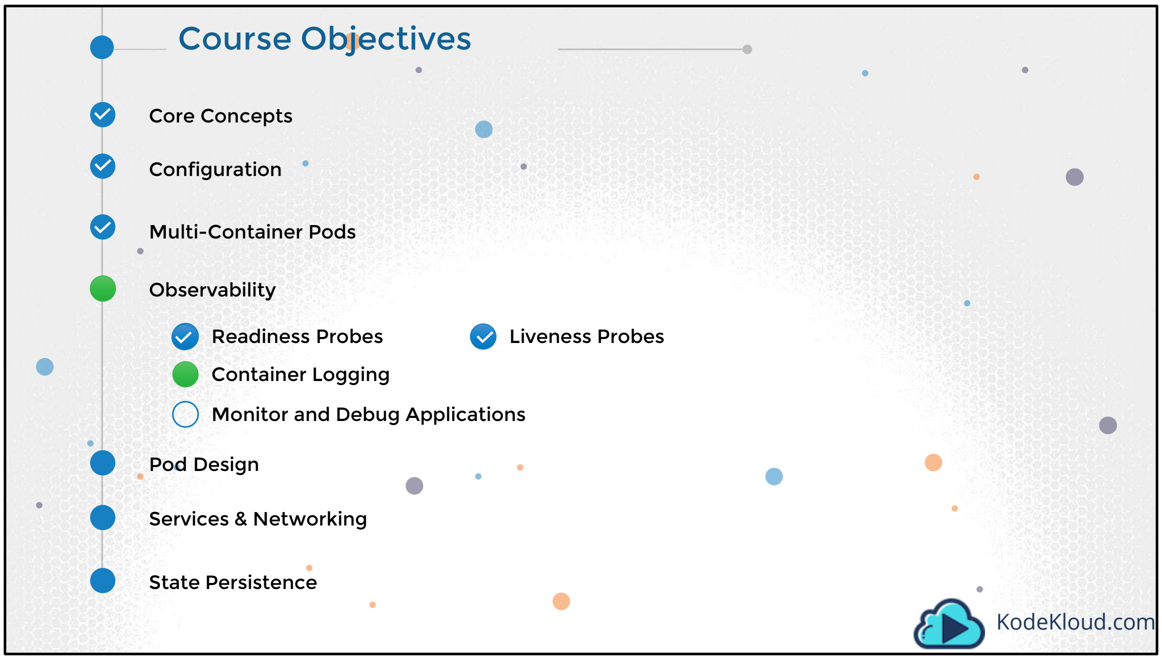
in the current(default) namespace

```
    Click on the ^^^ Quiz Portal tab above to open your quiz console

                ALL SET!
master $ []
```

Check Link Below!

KodeKloud.com

Access Test Here: https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743663



Course Objectives

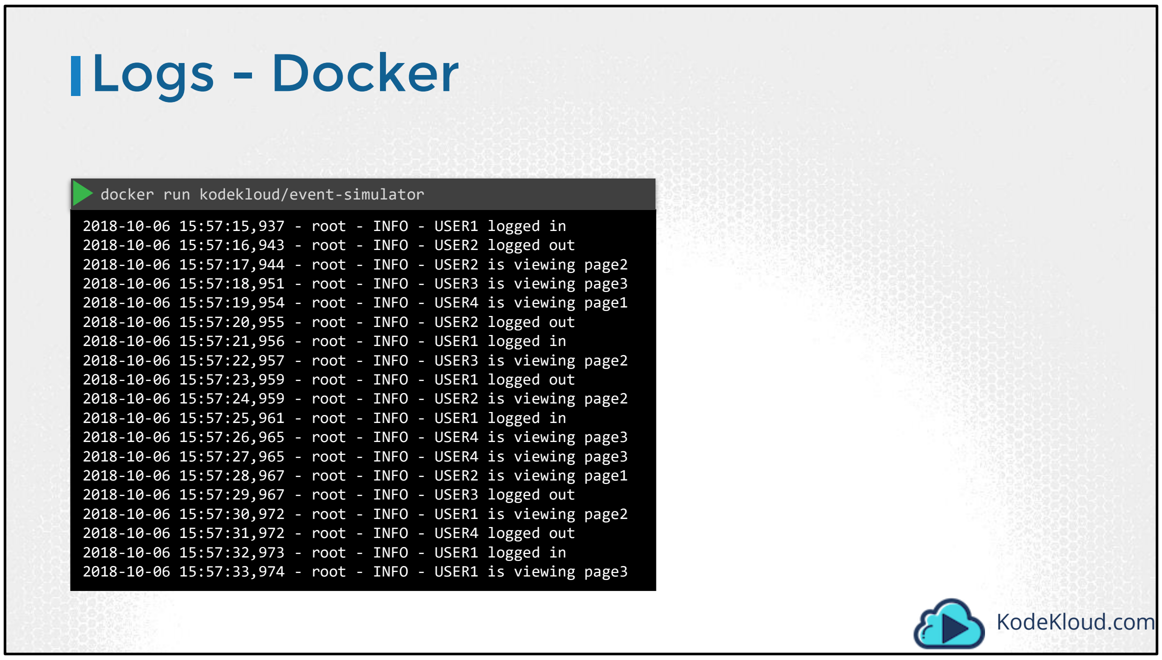
- ✓ Core Concepts
- ✓ Configuration
- ✓ Multi-Container Pods
- ● Observability
  - ✓ Readiness Probes
  - ✓ Liveness Probes
  - ● Container Logging
  - ○ Monitor and Debug Applications
- ● Pod Design
- ● Services & Networking
- ● State Persistence

KodeKloud.com

Hello and welcome to this lecture. In this lecture we will talk about various Logging mechanisms in kubernetes.

Container Logging

KodeKloud.com

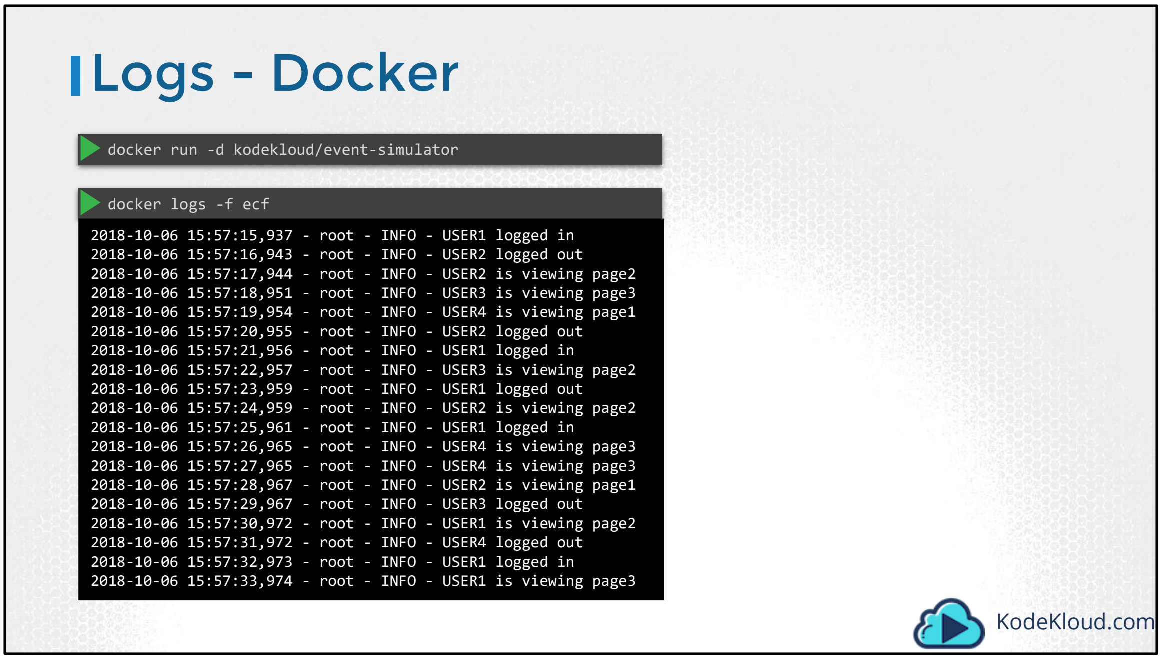


Logs - Docker

```
docker run kodekloud/event-simulator
2018-10-06 15:57:15,937 - root - INFO - USER1 logged in
2018-10-06 15:57:16,943 - root - INFO - USER2 logged out
2018-10-06 15:57:17,944 - root - INFO - USER2 is viewing page2
2018-10-06 15:57:18,951 - root - INFO - USER3 is viewing page3
2018-10-06 15:57:19,954 - root - INFO - USER4 is viewing page1
2018-10-06 15:57:20,955 - root - INFO - USER2 logged out
2018-10-06 15:57:21,956 - root - INFO - USER1 logged in
2018-10-06 15:57:22,957 - root - INFO - USER3 is viewing page2
2018-10-06 15:57:23,959 - root - INFO - USER1 logged out
2018-10-06 15:57:24,959 - root - INFO - USER2 is viewing page2
2018-10-06 15:57:25,961 - root - INFO - USER1 logged in
2018-10-06 15:57:26,965 - root - INFO - USER4 is viewing page3
2018-10-06 15:57:27,965 - root - INFO - USER4 is viewing page3
2018-10-06 15:57:28,967 - root - INFO - USER2 is viewing page1
2018-10-06 15:57:29,967 - root - INFO - USER3 logged out
2018-10-06 15:57:30,972 - root - INFO - USER1 is viewing page2
2018-10-06 15:57:31,972 - root - INFO - USER4 logged out
2018-10-06 15:57:32,973 - root - INFO - USER1 logged in
2018-10-06 15:57:33,974 - root - INFO - USER1 is viewing page3
```

KodeKloud.com

Let us start with logging in Docker. I run a docker container called event-simulator and all that it does is generate random events simulating a web server. These are events streamed to the standard output by the application.



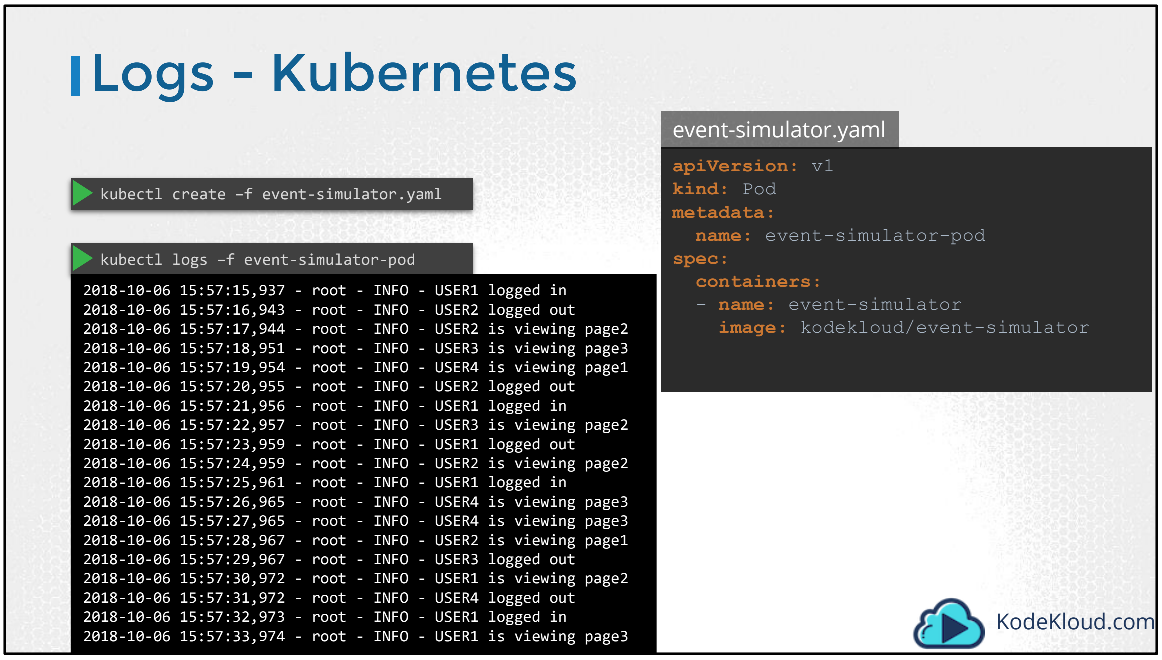
Logs - Docker

```
docker run -d kodekloud/event-simulator
```

```
docker logs -f ecf
2018-10-06 15:57:15,937 - root - INFO - USER1 logged in
2018-10-06 15:57:16,943 - root - INFO - USER2 logged out
2018-10-06 15:57:17,944 - root - INFO - USER2 is viewing page2
2018-10-06 15:57:18,951 - root - INFO - USER3 is viewing page3
2018-10-06 15:57:19,954 - root - INFO - USER4 is viewing page1
2018-10-06 15:57:20,955 - root - INFO - USER2 logged out
2018-10-06 15:57:21,956 - root - INFO - USER1 logged in
2018-10-06 15:57:22,957 - root - INFO - USER3 is viewing page2
2018-10-06 15:57:23,959 - root - INFO - USER1 logged out
2018-10-06 15:57:24,959 - root - INFO - USER2 is viewing page2
2018-10-06 15:57:25,961 - root - INFO - USER1 logged in
2018-10-06 15:57:26,965 - root - INFO - USER4 is viewing page3
2018-10-06 15:57:27,965 - root - INFO - USER4 is viewing page3
2018-10-06 15:57:28,967 - root - INFO - USER2 is viewing page1
2018-10-06 15:57:29,967 - root - INFO - USER3 logged out
2018-10-06 15:57:30,972 - root - INFO - USER1 is viewing page2
2018-10-06 15:57:31,972 - root - INFO - USER4 logged out
2018-10-06 15:57:32,973 - root - INFO - USER1 logged in
2018-10-06 15:57:33,974 - root - INFO - USER1 is viewing page3
```

KodeKloud.com

Now, if I were to run the docker container in the background, in a detached mode using the –d option, I wouldn't see those logs. If I wanted to view the logs, I could use the docker logs command followed by the container ID. The –f option helps us see the live log trail.



Logs - Kubernetes

```
kubectl create –f event-simulator.yaml
```

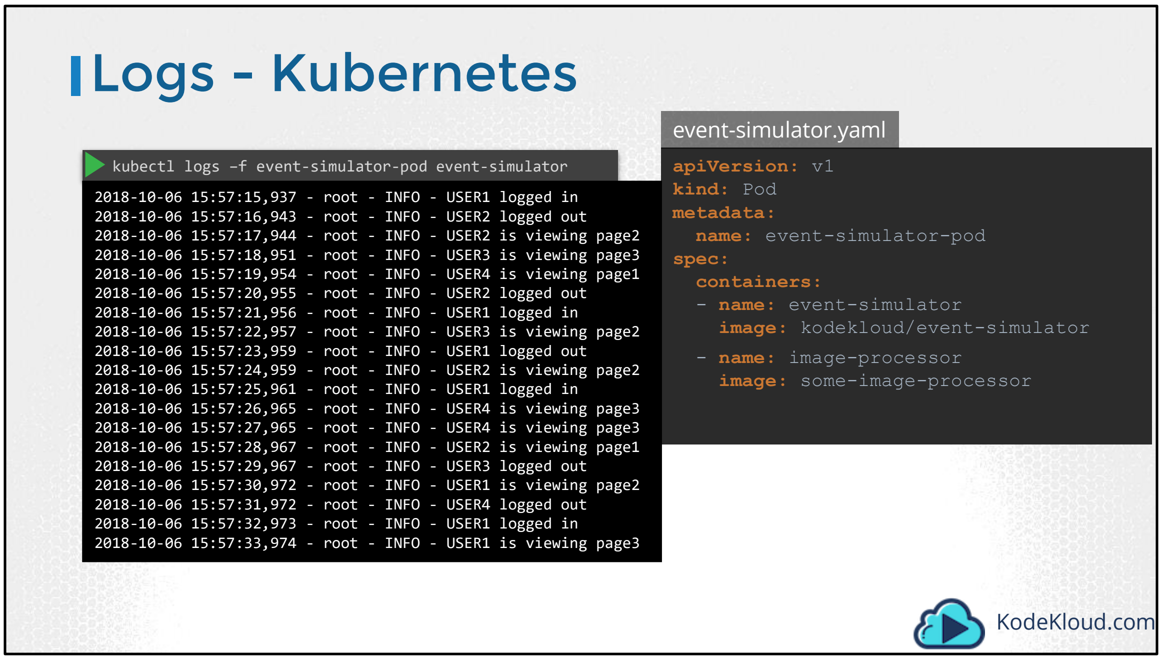
```
kubectl logs –f event-simulator-pod
2018-10-06 15:57:15,937 - root - INFO - USER1 logged in
2018-10-06 15:57:16,943 - root - INFO - USER2 logged out
2018-10-06 15:57:17,944 - root - INFO - USER2 is viewing page2
2018-10-06 15:57:18,951 - root - INFO - USER3 is viewing page3
2018-10-06 15:57:19,954 - root - INFO - USER4 is viewing page1
2018-10-06 15:57:20,955 - root - INFO - USER2 logged out
2018-10-06 15:57:21,956 - root - INFO - USER1 logged in
2018-10-06 15:57:22,957 - root - INFO - USER3 is viewing page2
2018-10-06 15:57:23,959 - root - INFO - USER1 logged out
2018-10-06 15:57:24,959 - root - INFO - USER2 is viewing page2
2018-10-06 15:57:25,961 - root - INFO - USER1 logged in
2018-10-06 15:57:26,965 - root - INFO - USER4 is viewing page3
2018-10-06 15:57:27,965 - root - INFO - USER4 is viewing page3
2018-10-06 15:57:28,967 - root - INFO - USER2 is viewing page1
2018-10-06 15:57:29,967 - root - INFO - USER3 logged out
2018-10-06 15:57:30,972 - root - INFO - USER1 is viewing page2
2018-10-06 15:57:31,972 - root - INFO - USER4 logged out
2018-10-06 15:57:32,973 - root - INFO - USER1 logged in
2018-10-06 15:57:33,974 - root - INFO - USER1 is viewing page3
```

event-simulator.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: event-simulator-pod
spec:
  containers:
  - name: event-simulator
    image: kodekloud/event-simulator
```

KodeKloud.com

Now back to Kubernetes. We create a pod with the same docker image using the pod definition file. Once it’s the pod is running, we can view the logs using the kubectl logs command with the pod name. Use the –f option to stream the logs live.



Logs - Kubernetes

```
kubectl logs -f event-simulator-pod event-simulator
2018-10-06 15:57:15,937 - root - INFO - USER1 logged in
2018-10-06 15:57:16,943 - root - INFO - USER2 logged out
2018-10-06 15:57:17,944 - root - INFO - USER2 is viewing page2
2018-10-06 15:57:18,951 - root - INFO - USER3 is viewing page3
2018-10-06 15:57:19,954 - root - INFO - USER4 is viewing page1
2018-10-06 15:57:20,955 - root - INFO - USER2 logged out
2018-10-06 15:57:21,956 - root - INFO - USER1 logged in
2018-10-06 15:57:22,957 - root - INFO - USER3 is viewing page2
2018-10-06 15:57:23,959 - root - INFO - USER1 logged out
2018-10-06 15:57:24,959 - root - INFO - USER2 is viewing page2
2018-10-06 15:57:25,961 - root - INFO - USER1 logged in
2018-10-06 15:57:26,965 - root - INFO - USER4 is viewing page3
2018-10-06 15:57:27,965 - root - INFO - USER4 is viewing page3
2018-10-06 15:57:28,967 - root - INFO - USER2 is viewing page1
2018-10-06 15:57:29,967 - root - INFO - USER3 logged out
2018-10-06 15:57:30,972 - root - INFO - USER1 is viewing page2
2018-10-06 15:57:31,972 - root - INFO - USER4 logged out
2018-10-06 15:57:32,973 - root - INFO - USER1 logged in
2018-10-06 15:57:33,974 - root - INFO - USER1 is viewing page3
```

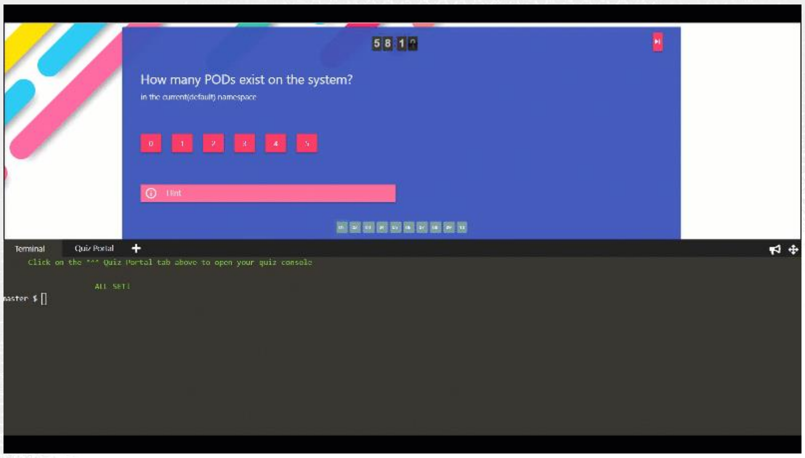
event-simulator.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: event-simulator-pod
spec:
  containers:
  - name: event-simulator
    image: kodekloud/event-simulator
  - name: image-processor
    image: some-image-processor
```

Now, these logs are specific to the container running inside the POD. As we learned before, Kubernetes PODs can have multiple docker containers in them. In this case I modify my pod definition file to include an additional container called image-processor. If you ran the kubectl logs command now with the pod name, which container's log would it show? If there are multiple containers within a pod, you must specify the name of the container explicitly in the command, otherwise it would fail asking you to specify a name. In this case I will specify the name of the first container event-simulator and that prints the relevant log messages.

Now, that is the simple logging functionality implemented within Kubernetes. And that is all that an application developer really needs to know to get started with Kubernetes. However, in the next lecture we will see more about advanced logging configuration and 3rd party support for logging in Kubernetes.

Practice Test



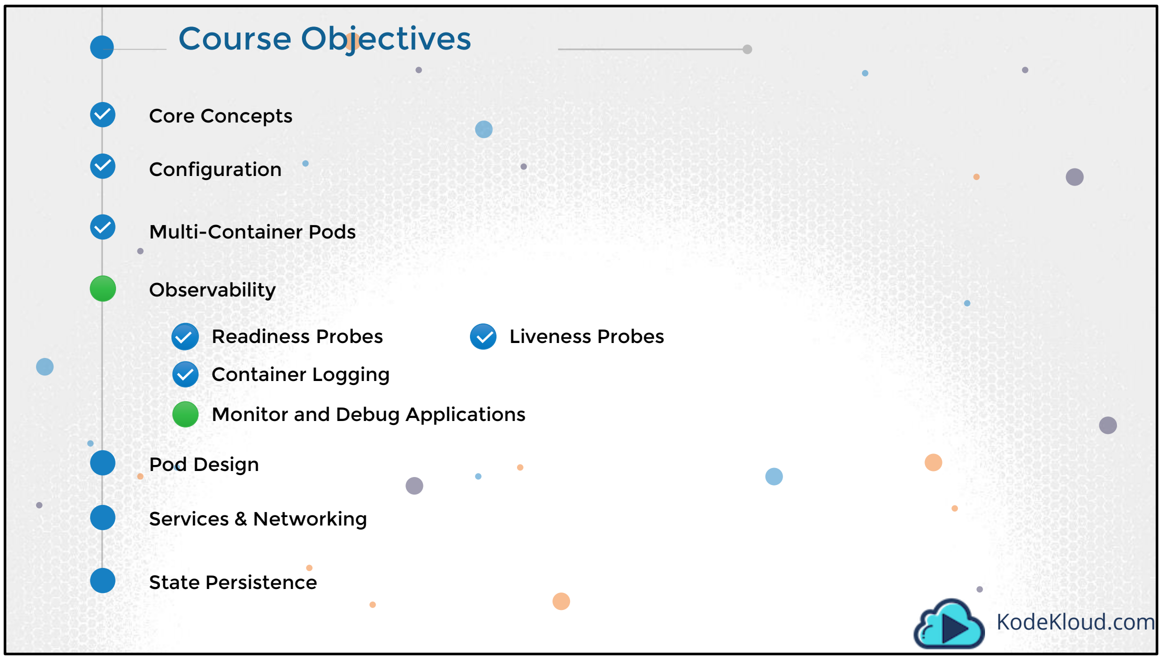
How many PODs exist on the system?
in the current(default) namespace

```
Click on the *** Quiz Portal tab above to open your quiz console

           ALL SET!
master $ []
```

Check Link Below!

KodeKloud.com

Access Video Here: https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743665



Course Objectives

- ✓ Core Concepts
- ✓ Configuration
- ✓ Multi-Container Pods
- ● Observability
  - ✓ Readiness Probes
  - ✓ Liveness Probes
  - ✓ Container Logging
  - ● Monitor and Debug Applications
- ● Pod Design
- ● Services & Networking
- ● State Persistence

KodeKloud.com

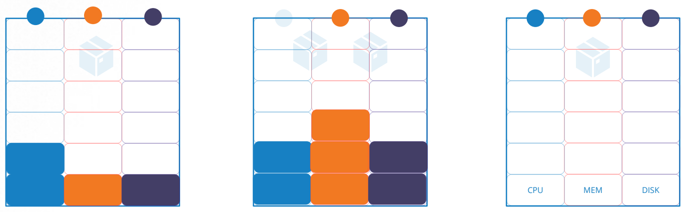
Hello and welcome to this lecture. In this lecture we will talk about the various monitoring and debugging options available.

Monitoring Kubernetes

211
KodeKloud.com

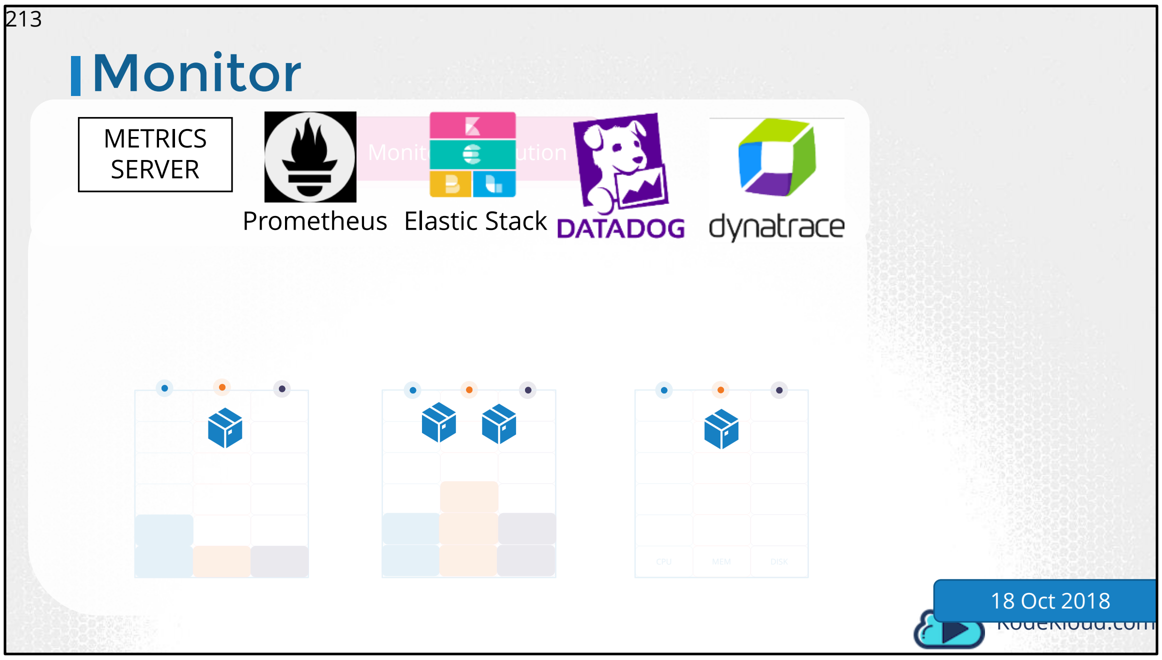
Hello and welcome to this lecture. In this lecture we talk about Monitoring a Kubernetes cluster.

Monitor



KodeKloud.com

So how do you monitor resource consumption on Kubernetes? Or more importantly what would you like to monitor? I'd like to know Node level metrics such as the number of nodes in the cluster, how many of them are healthy as well as performance metrics such as CPU. Memory, network and disk utilization.


213

Monitor

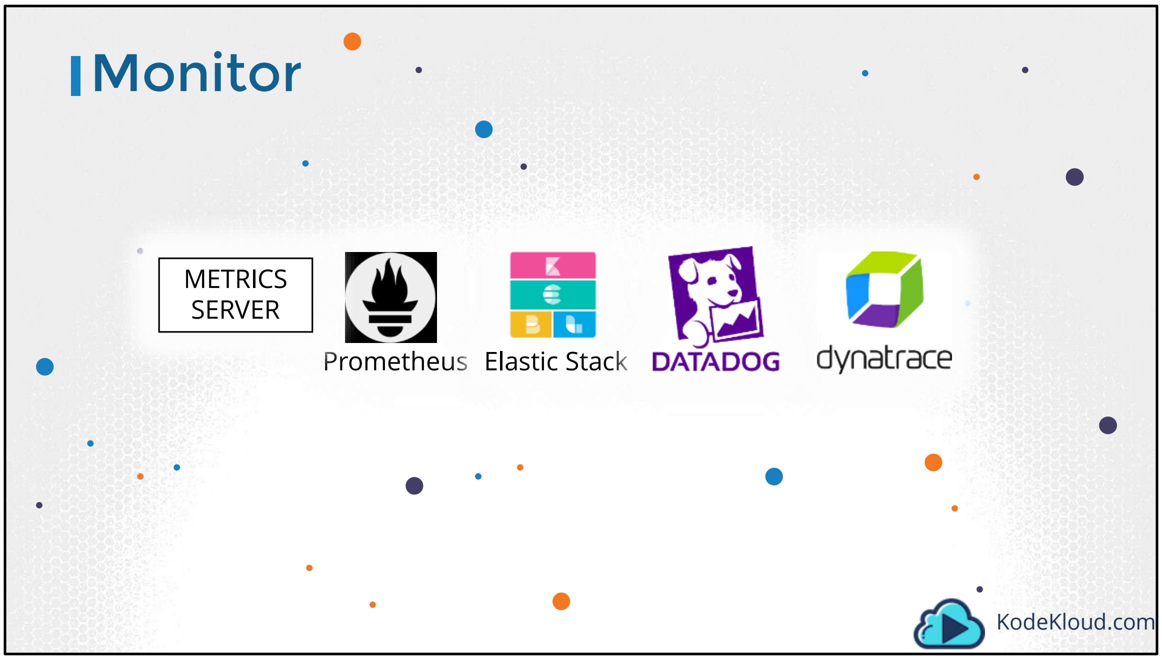
METRICS SERVER

Prometheus Elastic Stack DATADOG dynatrace

18 Oct 2018

As well as POD level metrics such as the number of PODs, and performance metrics of each POD such the CPU and Memory consumption.   So we need a solution that will monitor these metrics, store them and provide analytics around this data.

As of this recording , Kubernetes does not come with a full featured built-in monitoring solution. However, there are a number of open-source solutions available today, such as the Metrics-Server, Prometheus, the Elastic Stack, and proprietary solutions like Datadog and Dynatrace.

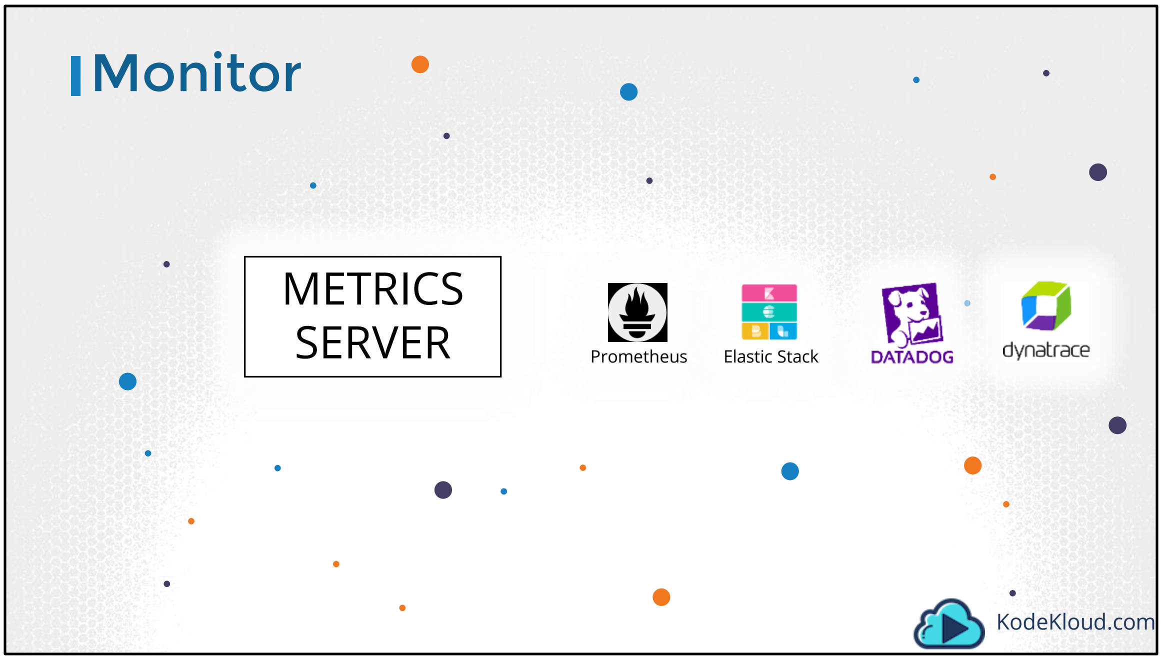


Monitor

METRICS SERVER

Prometheus
Elastic Stack
DATADOG
dynatrace
KodeKloud.com

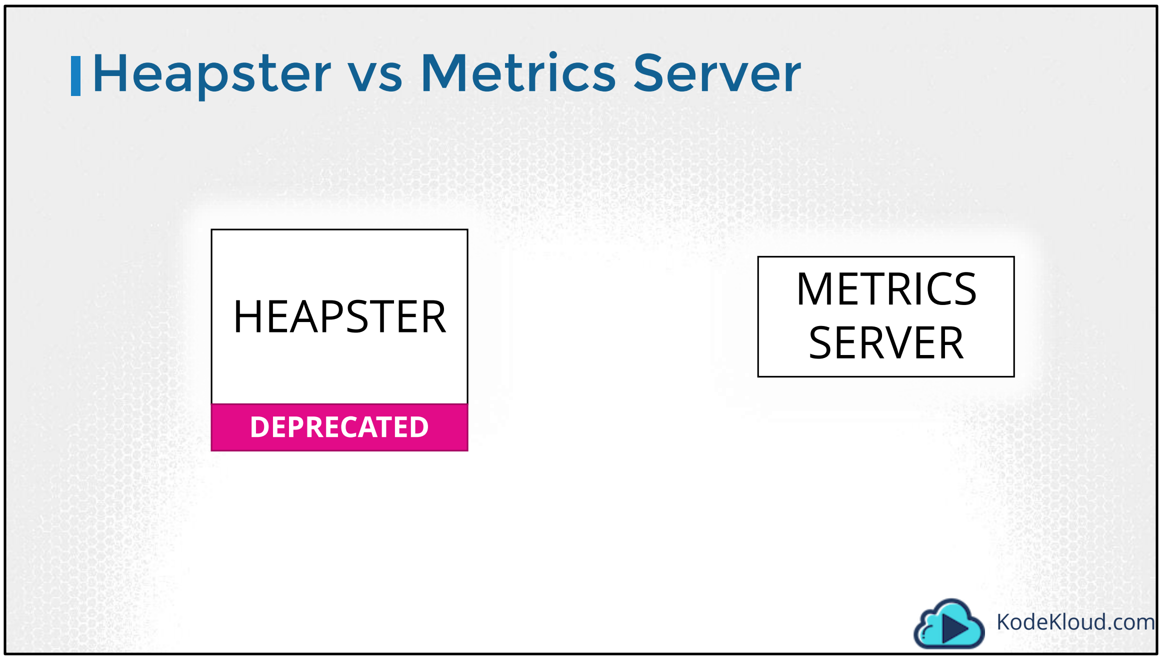
The kubernetes for developers course as well as the certification, requires only a minimal knowledge of monitoring kubernetes.


Monitor

METRICS SERVER

Prometheus
Elastic Stack
DATADOG
dynatrace
KodeKloud.com

So in the scope of this course, we will discuss about the Metrics Server only. The other solutions will be discussed in the Kubernetes for administrators course.



Heapster vs Metrics Server

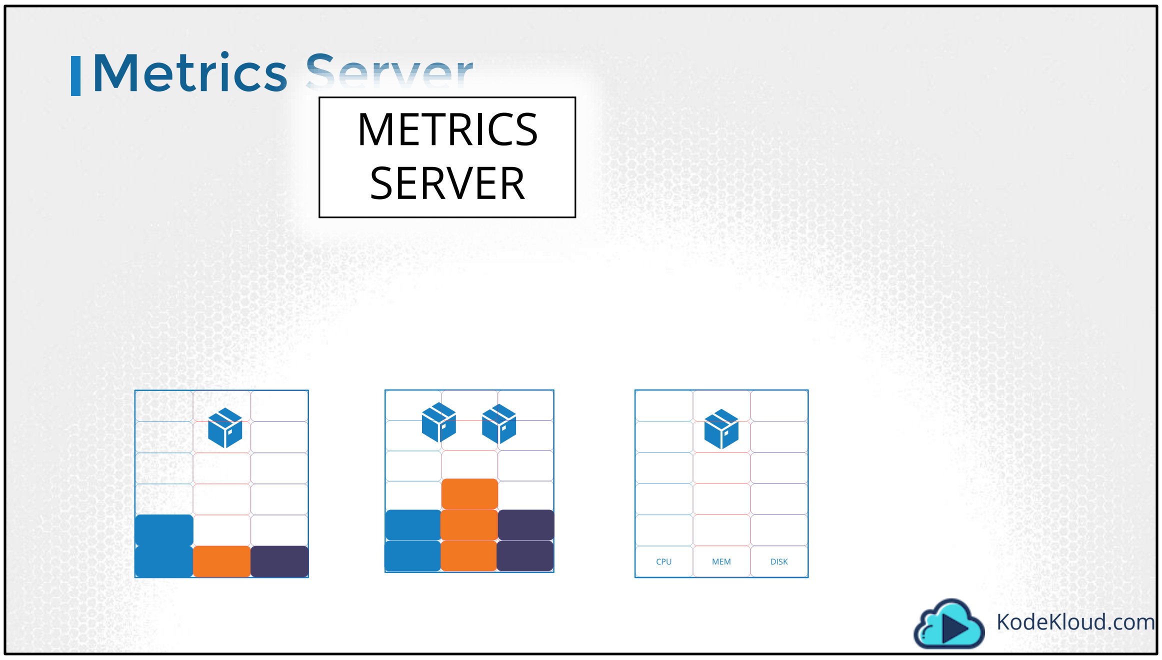
HEAPSTER

DEPRECATED

METRICS SERVER

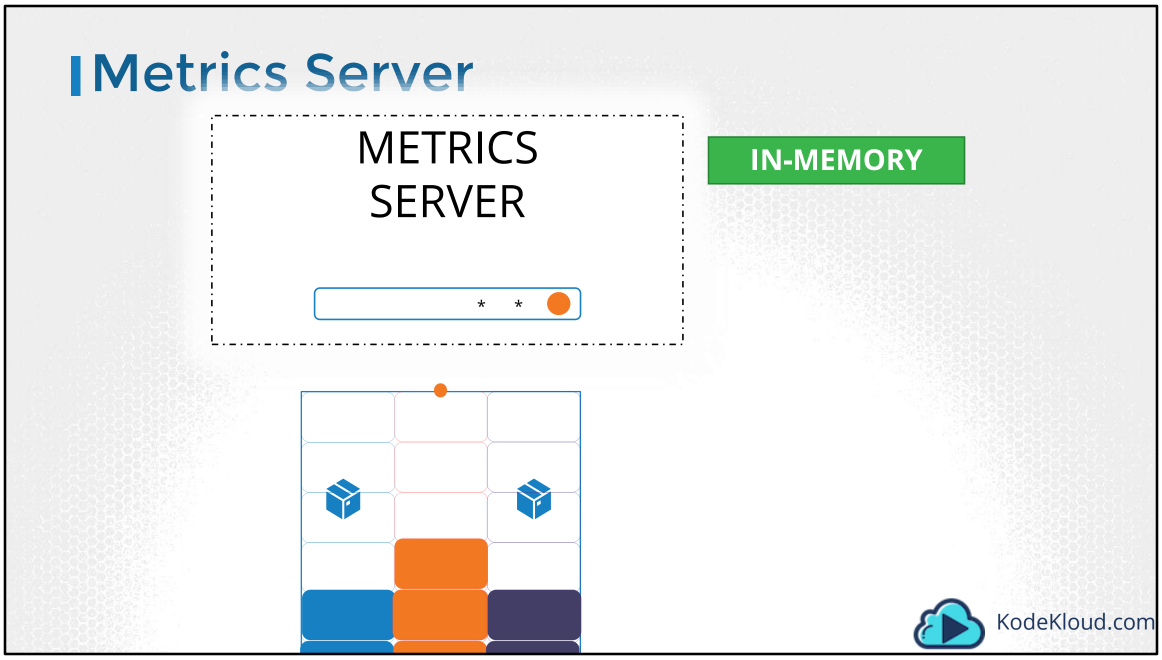
KodeKloud.com

Heapster was one of the original projects that enabled monitoring and analysis features for Kubernetes. You will see a lot of reference online when you look for reference architectures on monitoring Kubernetes. However, Heapster is now Deprecated and a slimmed down version was formed known as the Metrics Server.



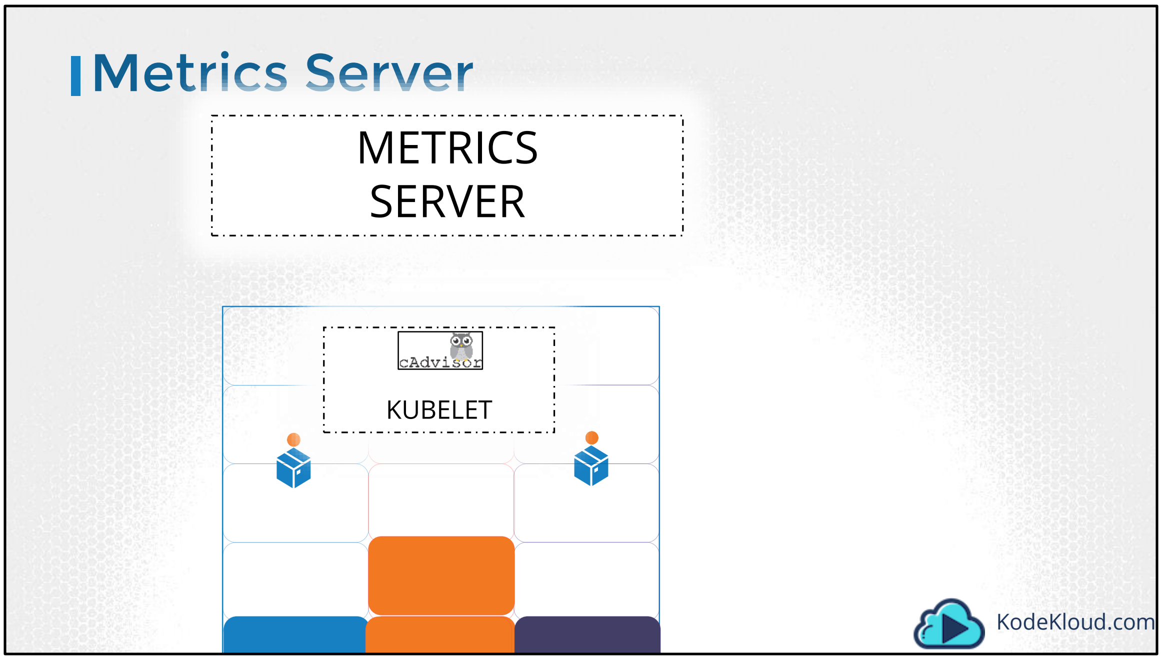
Metrics Server

You can have one metrics server per kubernetes cluster.



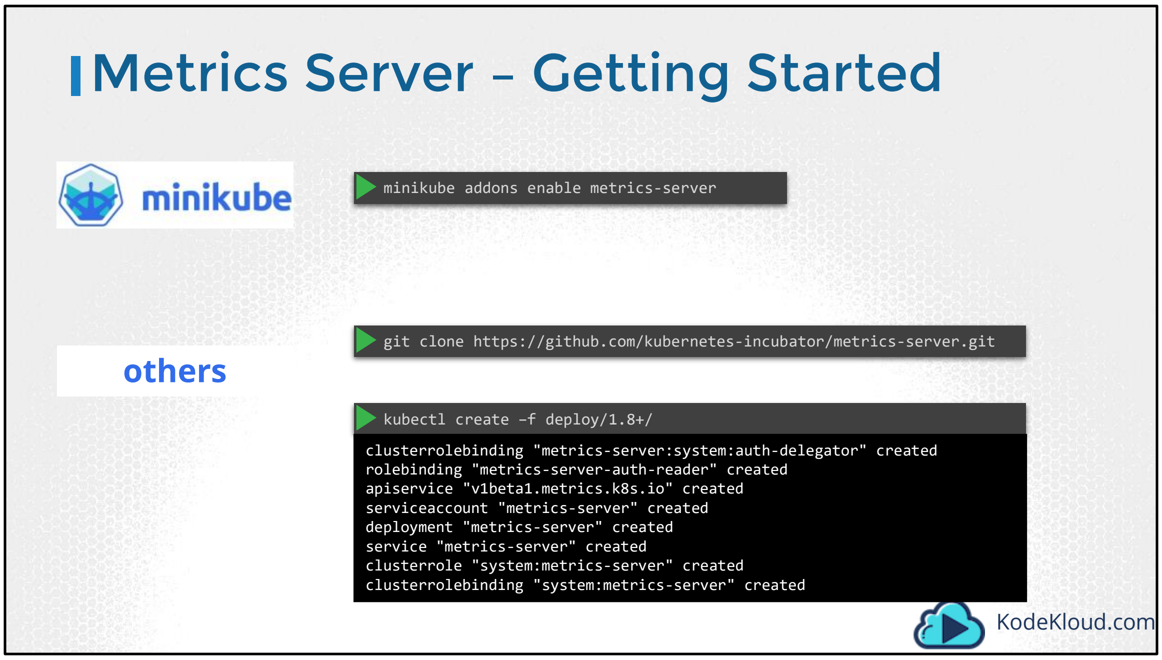
Metrics Server

The metrics server retrieves metrics from each of the kubernetes nodes and pods, aggregates them and stores them in memory. Note that the metrics server is only an in-memory monitoring solution and does not store the metrics on the disk, and as a result you cannot see historical performance data. For that you must rely on one of the advanced monitoring solutions we talked about earlier in this lecture.



Metrics Server

So how are the metrics generated for the PODs on these nodes? Kubernetes runs an agent on each node known as the kubelet, which is responsible for receiving instructions from the kubernetes API master server and running PODs on the nodes. The kubelet also contains a subcomponent known as as cAdvisor or Container Advisor. cAdvisor is responsible for retrieving performance metrics from pods, and exposing them through the kubelet API to make the metrics available for the Metrics Server.



Metrics Server – Getting Started

minikube

```
minikube addons enable metrics-server
```

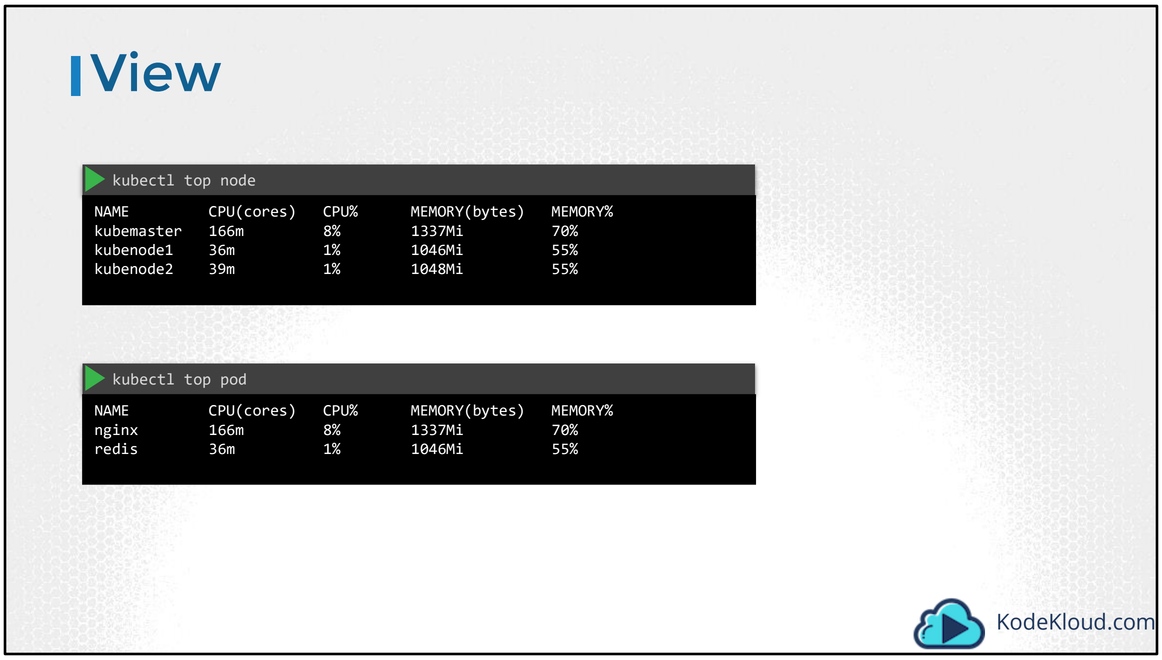
others

```
git clone https://github.com/kubernetes-incubator/metrics-server.git
```

```
kubectl create –f deploy/1.8+/
clusterrolebinding "metrics-server:system:auth-delegator" created
rolebinding "metrics-server-auth-reader" created
apiservice "v1beta1.metrics.k8s.io" created
serviceaccount "metrics-server" created
deployment "metrics-server" created
service "metrics-server" created
clusterrole "system:metrics-server" created
clusterrolebinding "system:metrics-server" created
```

KodeKloud.com

If you are using minikube for your local cluster, run the command minikube addons enable metrics-server. For all other environments deploy the metrics server by cloning the metrics-server deployment files from the github repository. And then deploying the required components using the kubectl create command. This command deploys a set of pods, services and roles to enable metrics server to poll for performance metrics from the nodes in the cluster.



View

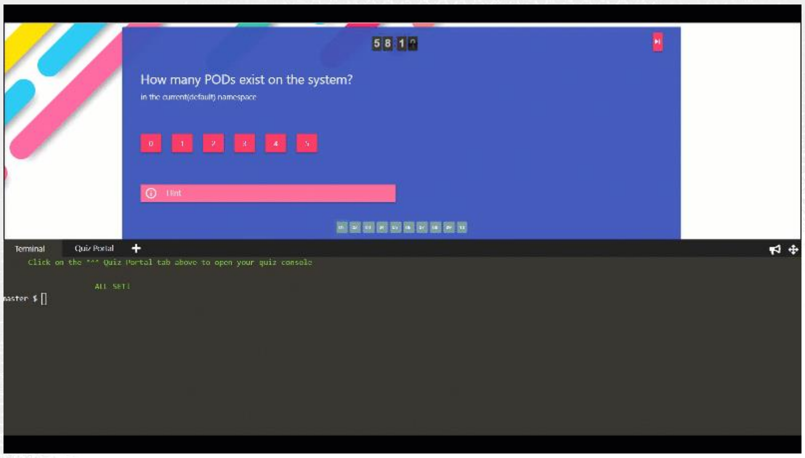
```
kubectl top node
NAME          CPU(cores)    CPU%      MEMORY(bytes)    MEMORY%
kubemaster    166m          8%        1337Mi           70%
kubenode1     36m           1%        1046Mi           55%
kubenode2     39m           1%        1048Mi           55%
```

```
kubectl top pod
NAME          CPU(cores)    CPU%      MEMORY(bytes)    MEMORY%
nginx         166m          8%        1337Mi           70%
redis         36m           1%        1046Mi           55%
```

Once deployed, give the metrics-server some time to collect and process data. Once processed, cluster performance can be viewed by running the command kubectl top node. This provides the CPU and Memory consumption of each of the nodes. As you can see 8% of the CPU on my master node is consumed, which is about 166 milli cores.

Use the kubectl top pod command to view performance metrics of pods in kubernetes.

Practice Test



How many PODs exist on the system?

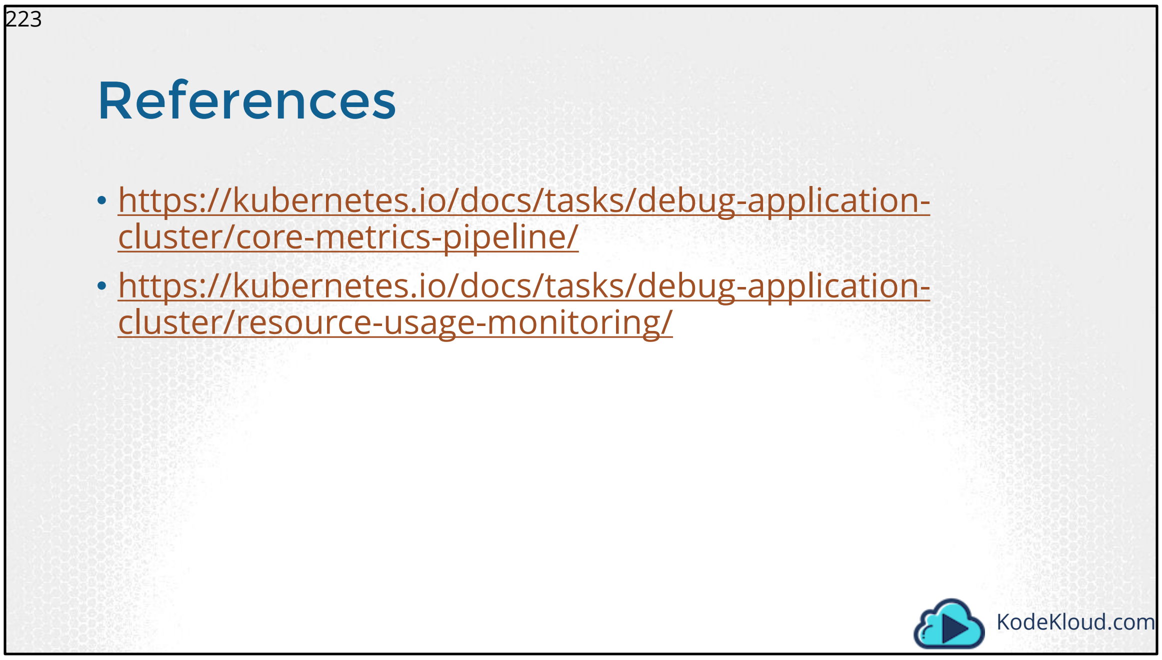
in the current(default) namespace

```
   Click on the ^^^ Quiz Portal tab above to open your quiz console

               ALL SET!
master $ []
```

Check Link Below!

KodeKloud.com

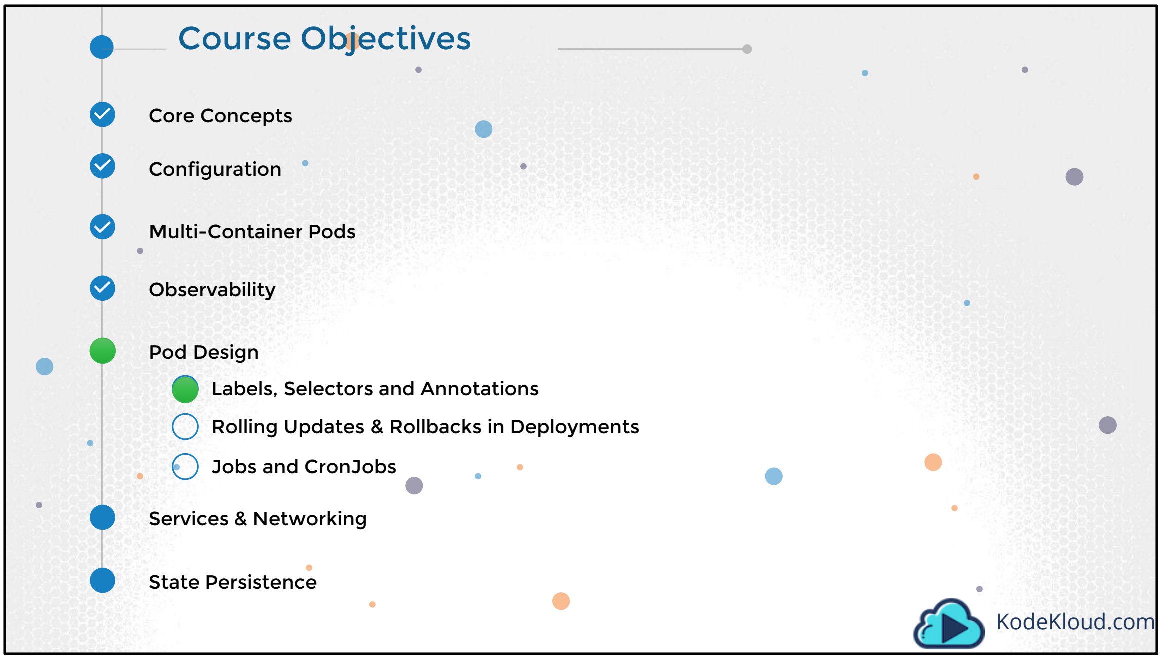
Access Test Here: https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743690



References

- https://kubernetes.io/docs/tasks/debug-application-cluster/core-metrics-pipeline/
- https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/

KodeKloud.com



Course Objectives

- ✓ Core Concepts
- ✓ Configuration
- ✓ Multi-Container Pods
- ✓ Observability
- ● Pod Design
  - ● Labels, Selectors and Annotations
  - ○ Rolling Updates & Rollbacks in Deployments
  - ○ Jobs and CronJobs
- ● Services & Networking
- ● State Persistence

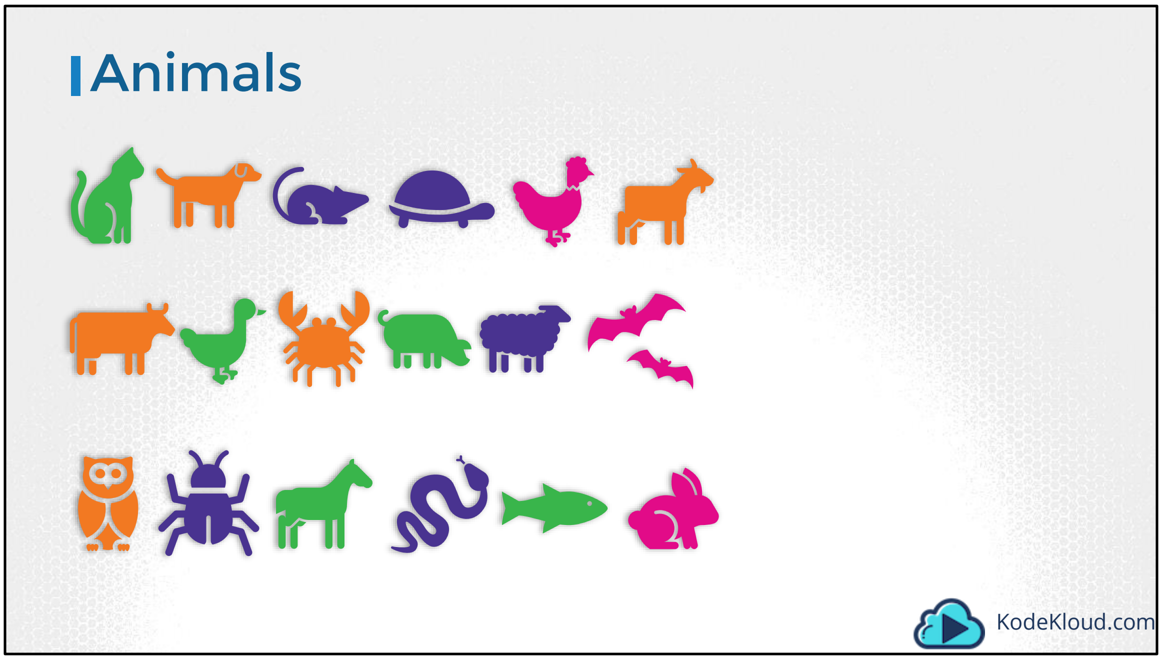
KodeKloud.com

We then move on to Labels & Selectors. And then Rolling updates and rollbacks in deployments. We will learn about why you need Jobs and CronJobs and how to schedule them.

Labels, Selectors & Annotations

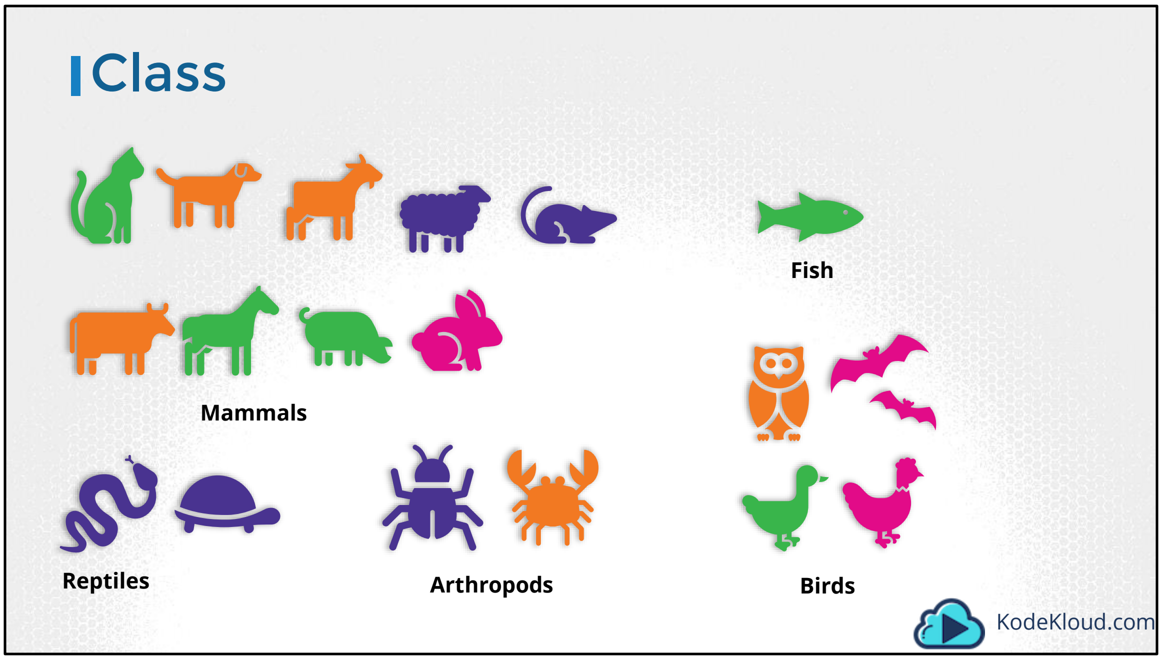
KodeKloud.com

Let us start with Labels and Selectors. What do we know about Labels and Selectors already?



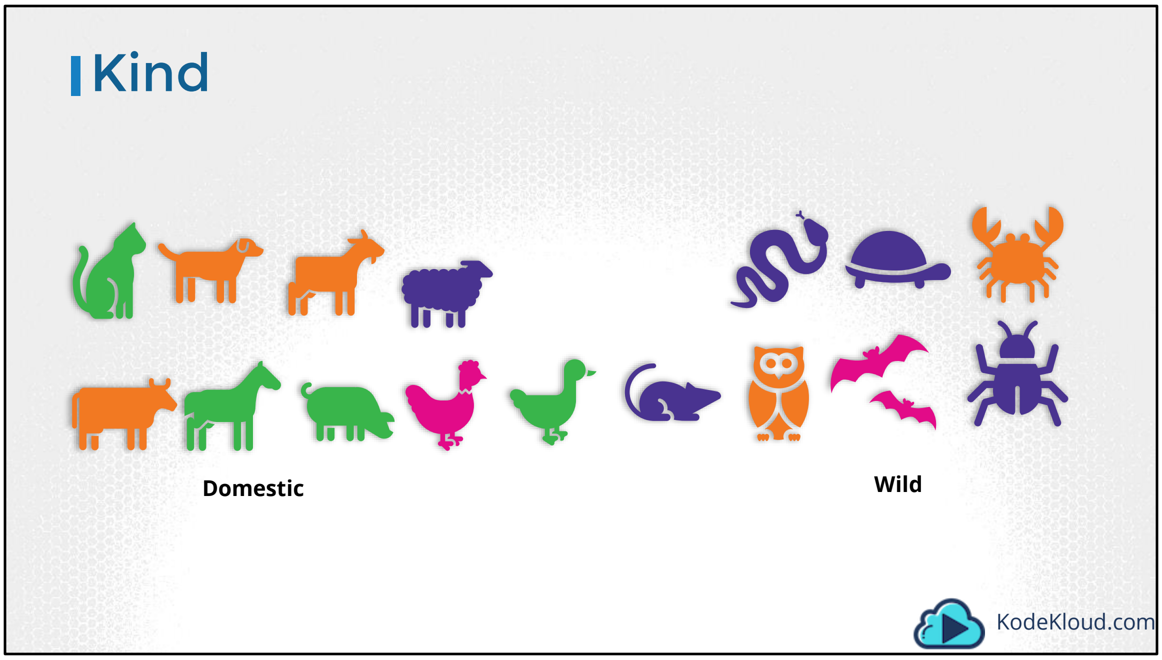
Animals

Labels and Selectors are a standard method to group things together. Say you have a set of different species. A user wants to be able to filter them based on different criteria.



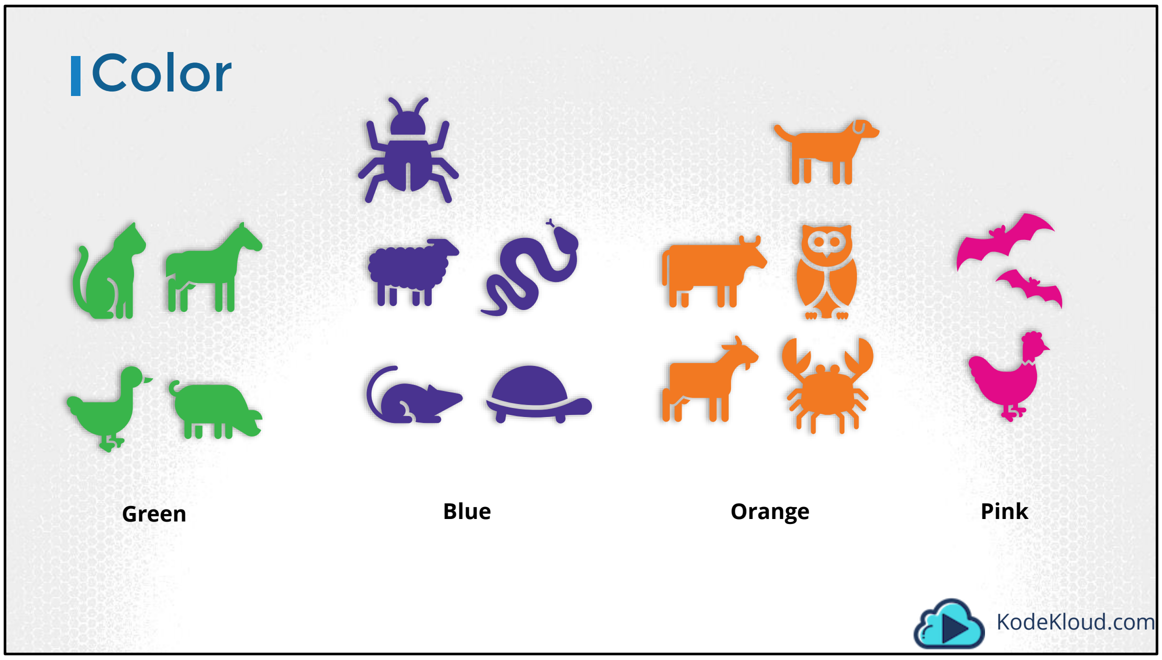
Class

Such as based on their class.



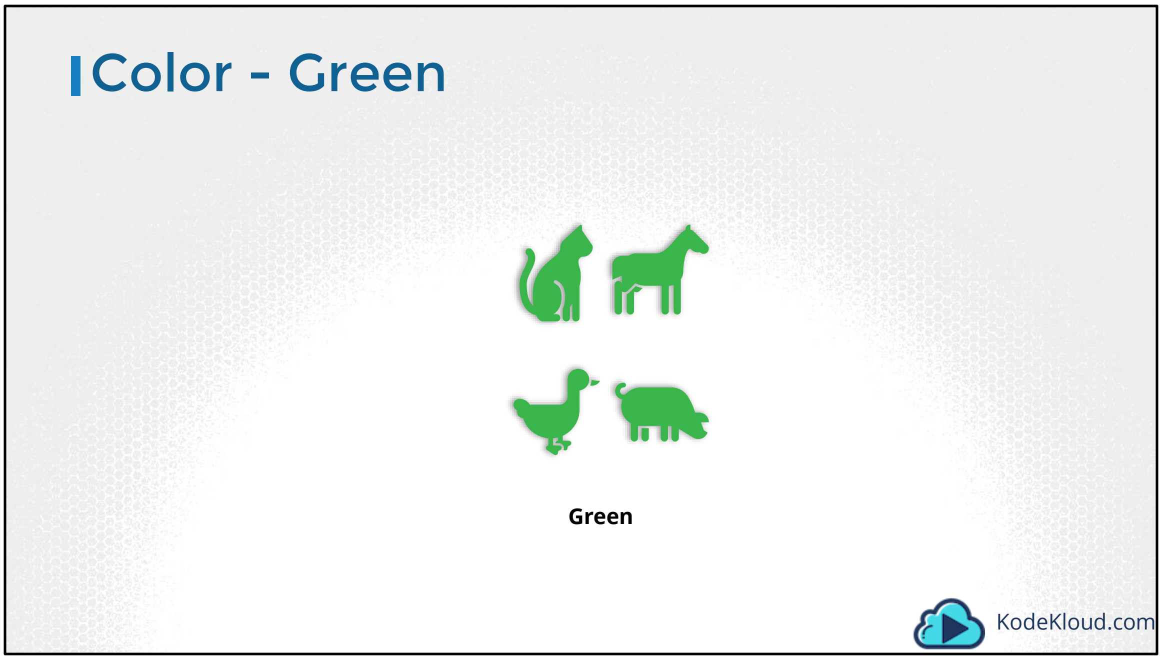
Kind

Or based on their type – domestic or wild.

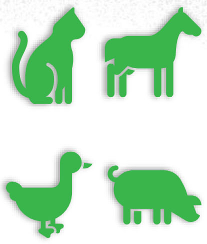


Color

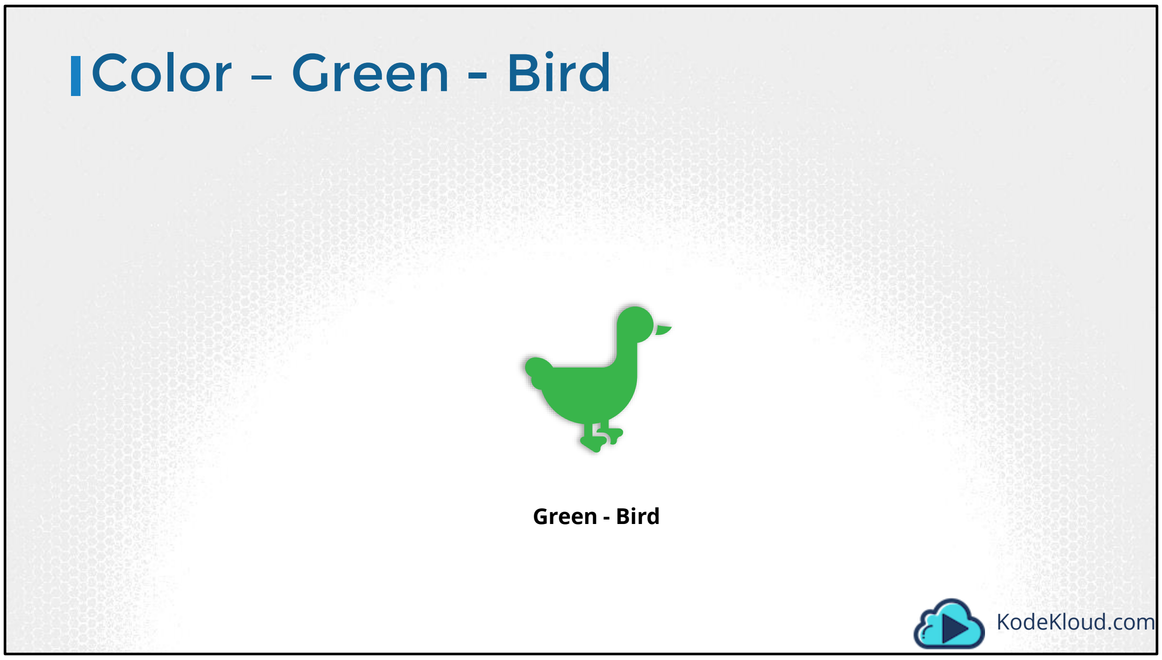
Or say by color.



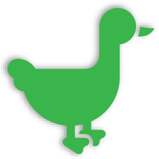
Color - Green



And not just group, you want to be able to filter them based on a criteria. Such as all green animals



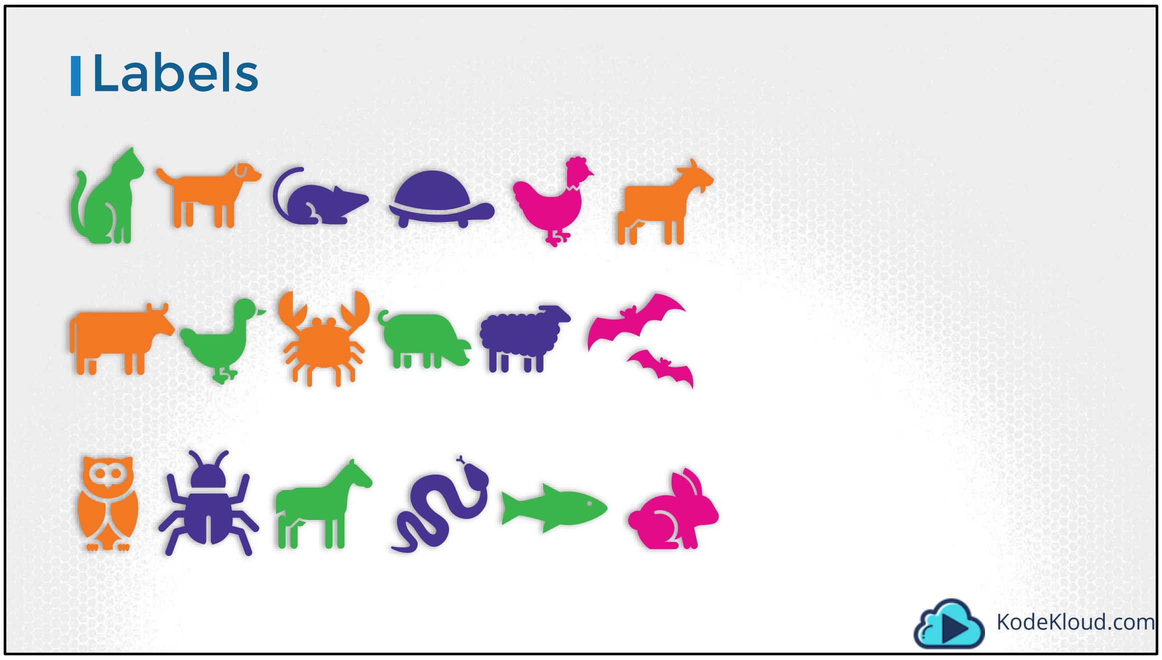
Color – Green - Bird



Green - Bird

KodeKloud.com

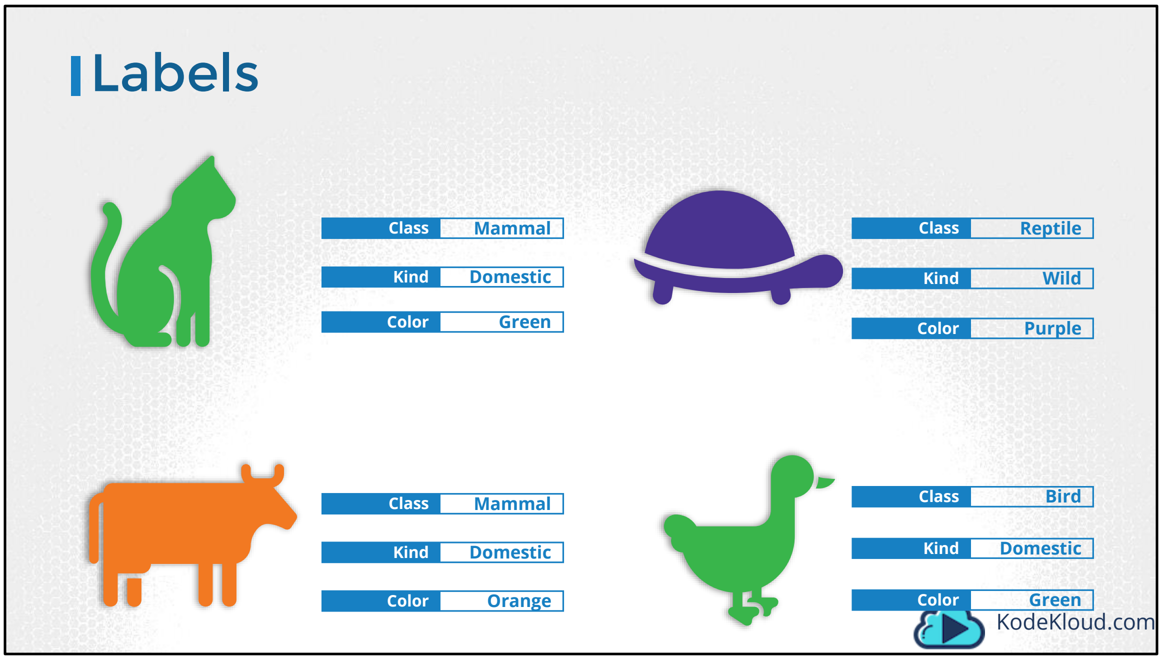
Or with multiple criteria such as everything green that is also a bird. Whatever that classification may be you need the ability to group things together and filter them based on your needs. And the best way to do that, is with labels.



Labels

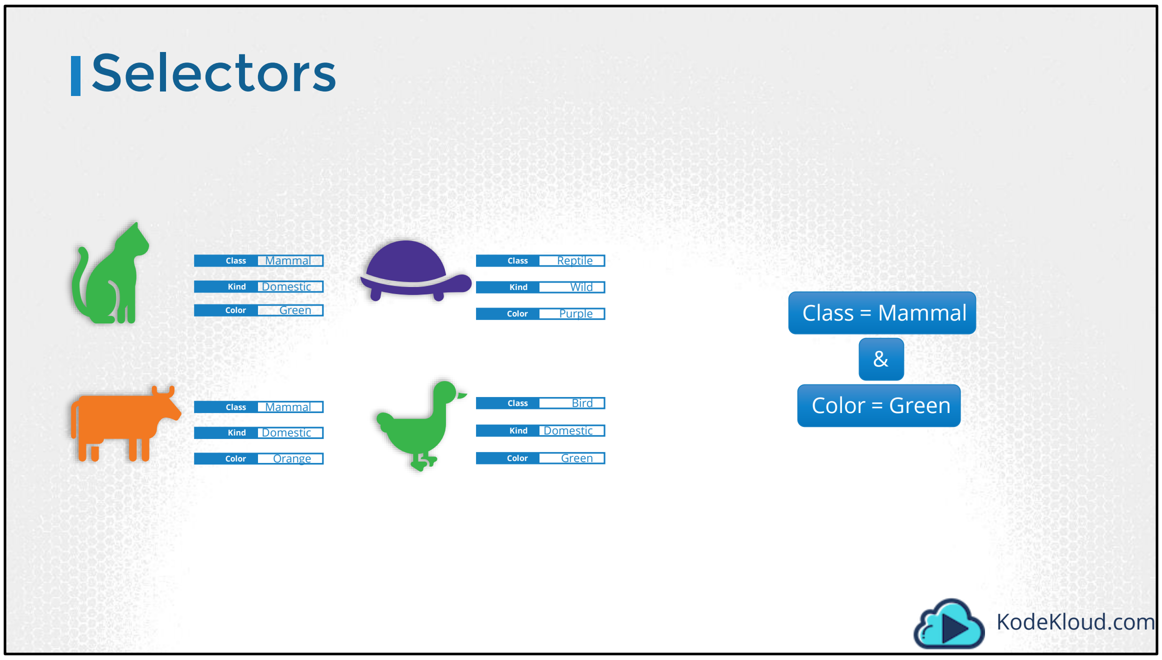
KodeKloud.com

Labels are properties attached to each item.



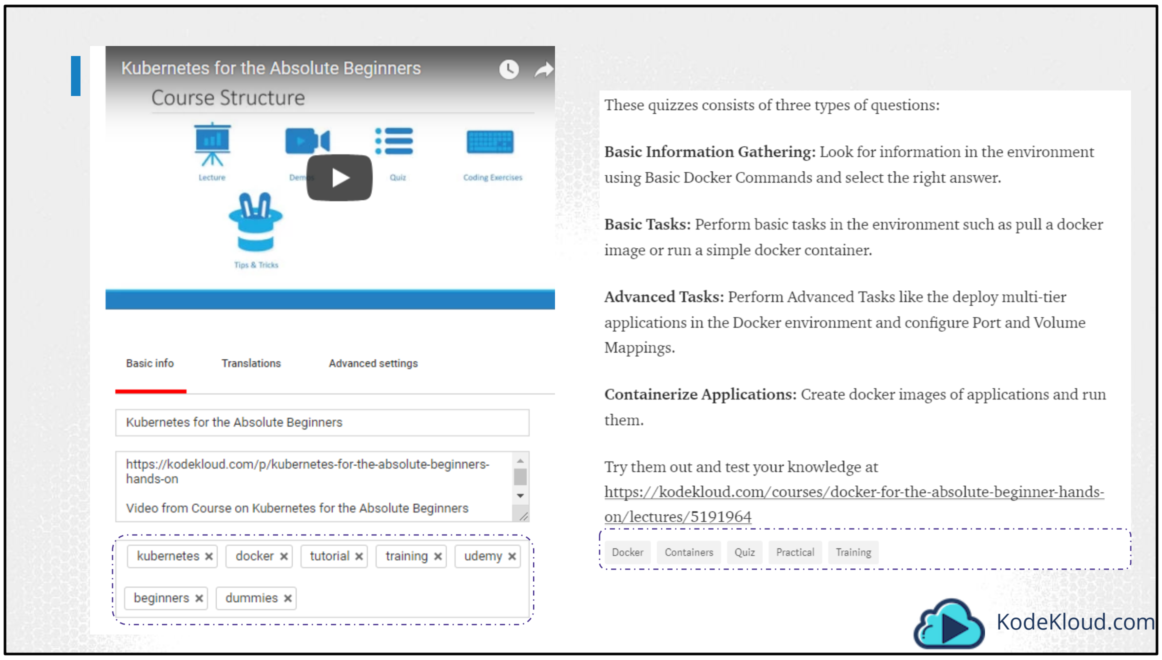
Labels

Labels are properties attached to each item. So you add properties to each item for their class, kind and color.



Selectors

Selectors help you filter these items. For example, when you say class equals mammal, we get a list of mammals. And when you say color equals green, we get the green mammals.



Kubernetes for the Absolute Beginners

https://kodekloud.com/p/kubernetes-for-the-absolute-beginners-hands-on

Video from Course on Kubernetes for the Absolute Beginners

kubernetes ✕ docker ✕ tutorial ✕ training ✕ udemy ✕ beginners ✕ dummies ✕

These quizzes consists of three types of questions:

Basic Information Gathering: Look for information in the environment using Basic Docker Commands and select the right answer.

Basic Tasks: Perform basic tasks in the environment such as pull a docker image or run a simple docker container.

Advanced Tasks: Perform Advanced Tasks like the deploy multi-tier applications in the Docker environment and configure Port and Volume Mappings.

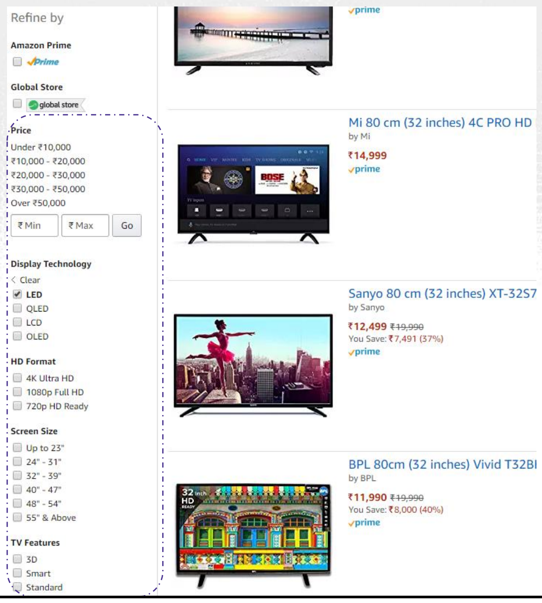
Containerize Applications: Create docker images of applications and run them.

Try them out and test your knowledge at https://kodekloud.com/courses/docker-for-the-absolute-beginner-hands-on/lectures/5191964

Docker Containers Quiz Practical Training

KodeKloud.com

We see labels and selectors used everywhere, such as the keywords you tag to youtube videos or blogs that help users filter and find the right content.



Display Technology: [✓ LED] QLED LCD OLED

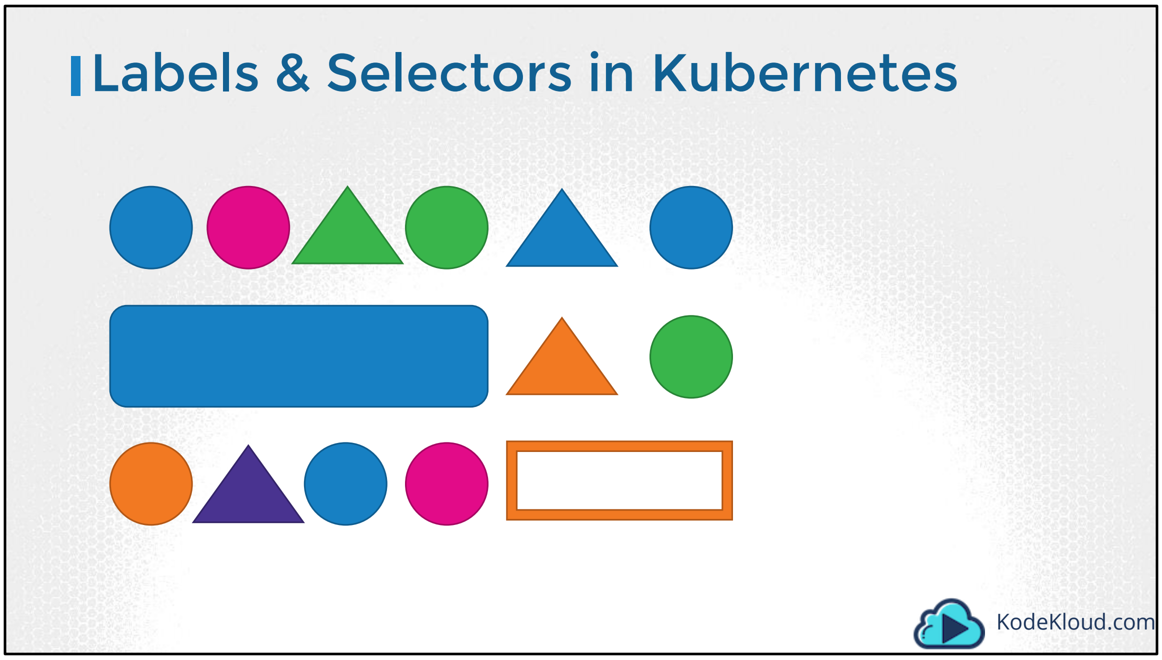
prime

Mi 80 cm (32 inches) 4C PRO HD
by Mi
₹14,999
prime

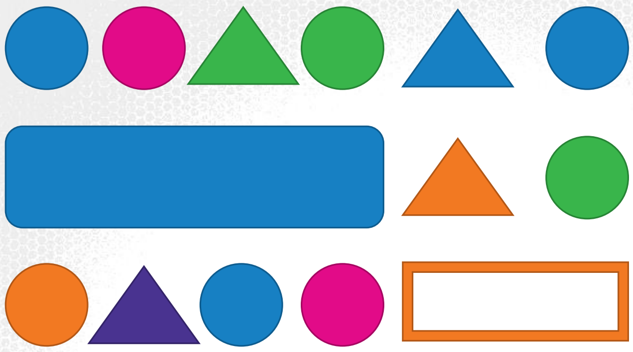
Sanyo 80 cm (32 inches) XT-32S7
by Sanyo
₹12,499 ~~₹19,990~~
You Save: ₹7,491 (37%)
prime

BPL 80cm (32 inches) Vivid T32BI
by BPL
₹11,990 ~~₹19,990~~
You Save: ₹8,000 (40%)
prime

We see labels added to items in an online store that help you add different kinds of filters to view your products.

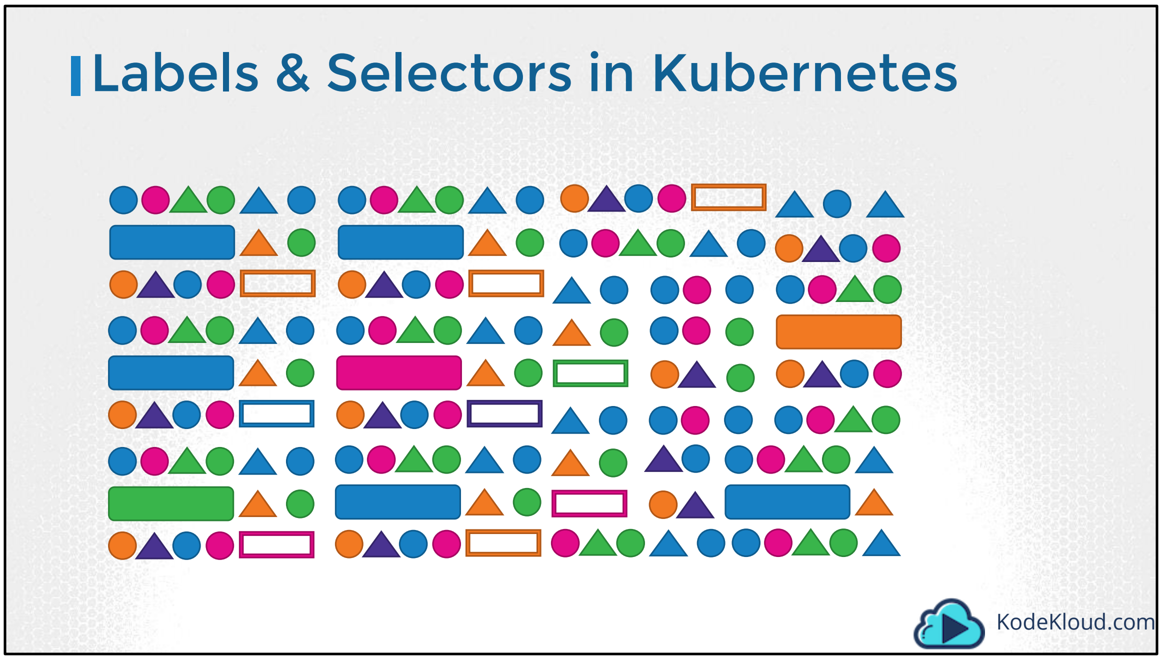


Labels & Selectors in Kubernetes

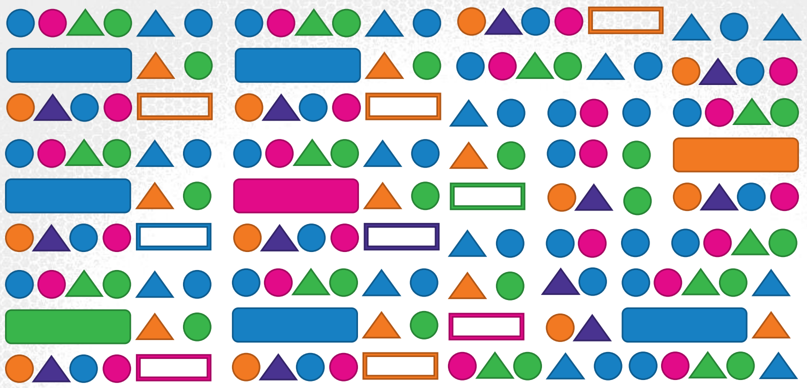


KodeKloud.com

So how are labels and selectors used in Kubernetes? We have created a lot of different types of Objects in Kuberentes. Pods, Services, ReplicaSets and Deployments. For Kubernetes, all of these are different objects. Over time you may end up having 100s and 1000s of these objects in your cluster. Then you will need a way to filter and view different objects by different categories. Like

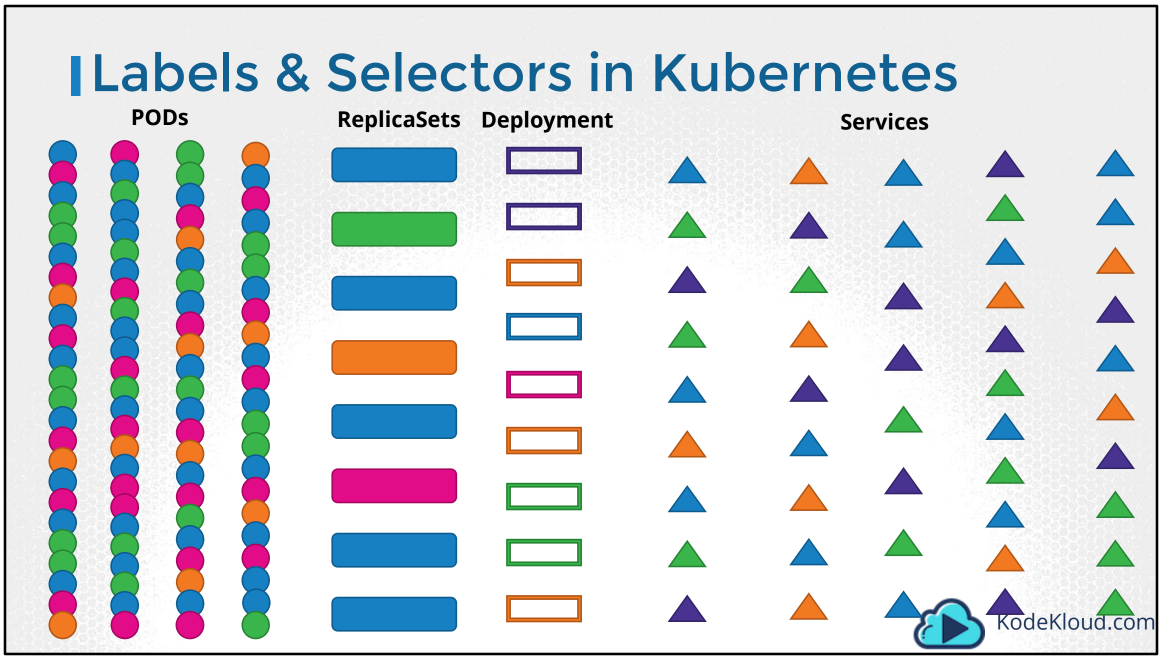


Labels & Selectors in Kubernetes



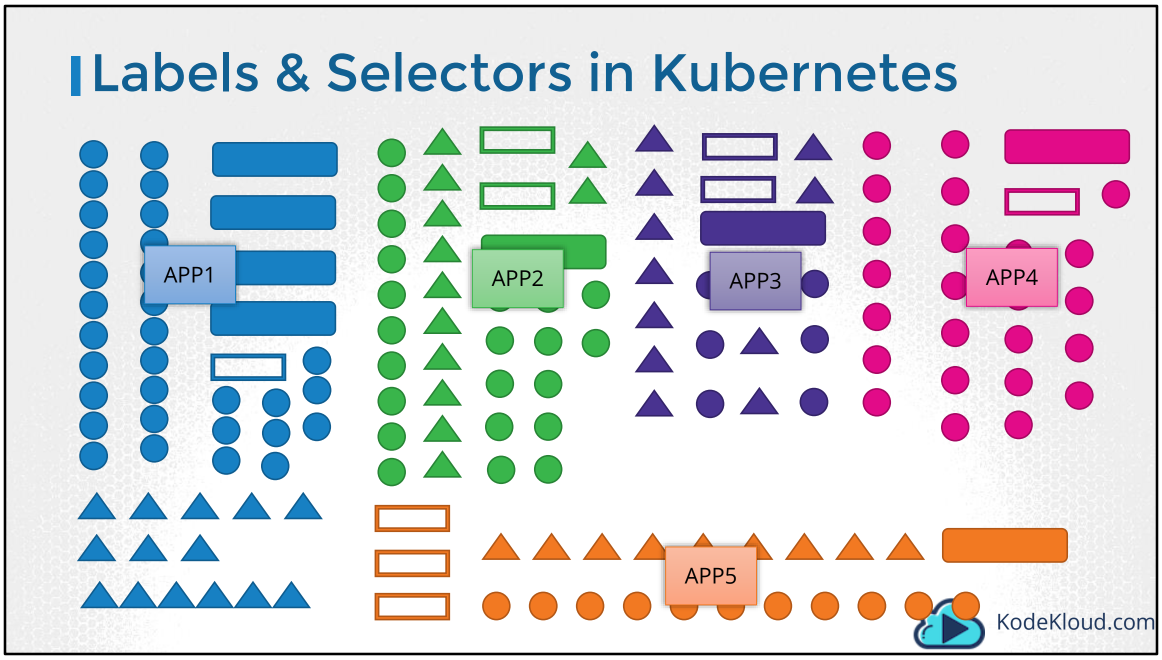
KodeKloud.com

Over time you may end up having 100s and 1000s of these objects in your cluster. Then you will need a way to group, filter and view different objects by different categories.



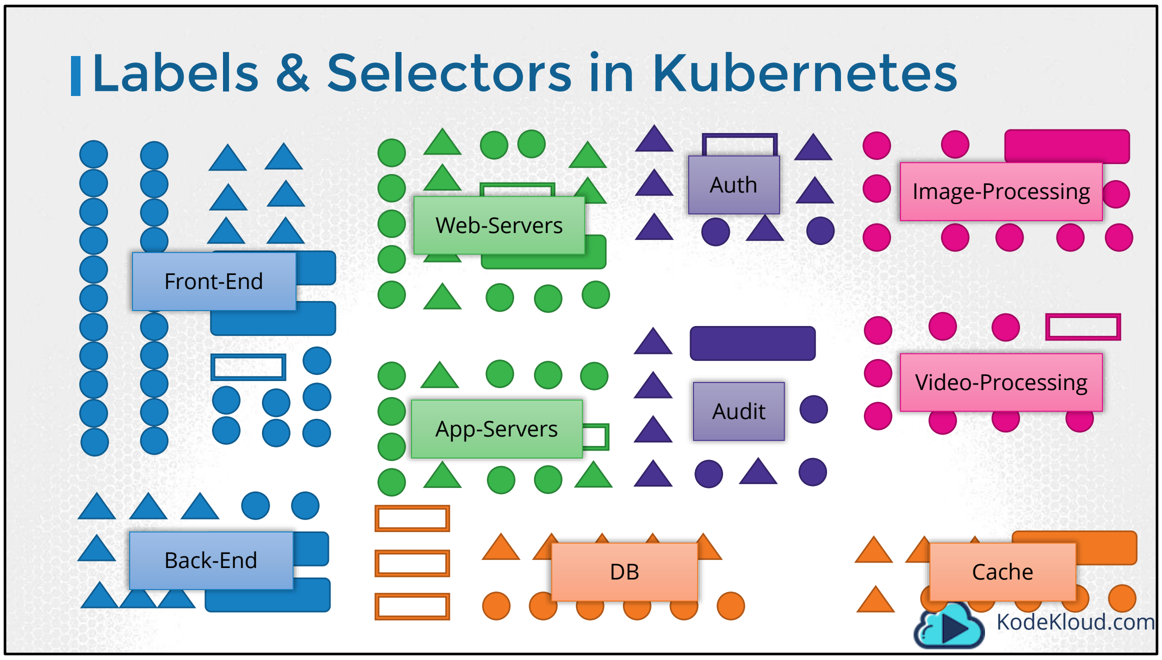
Labels & Selectors in Kubernetes

Such as to group objects by their type.

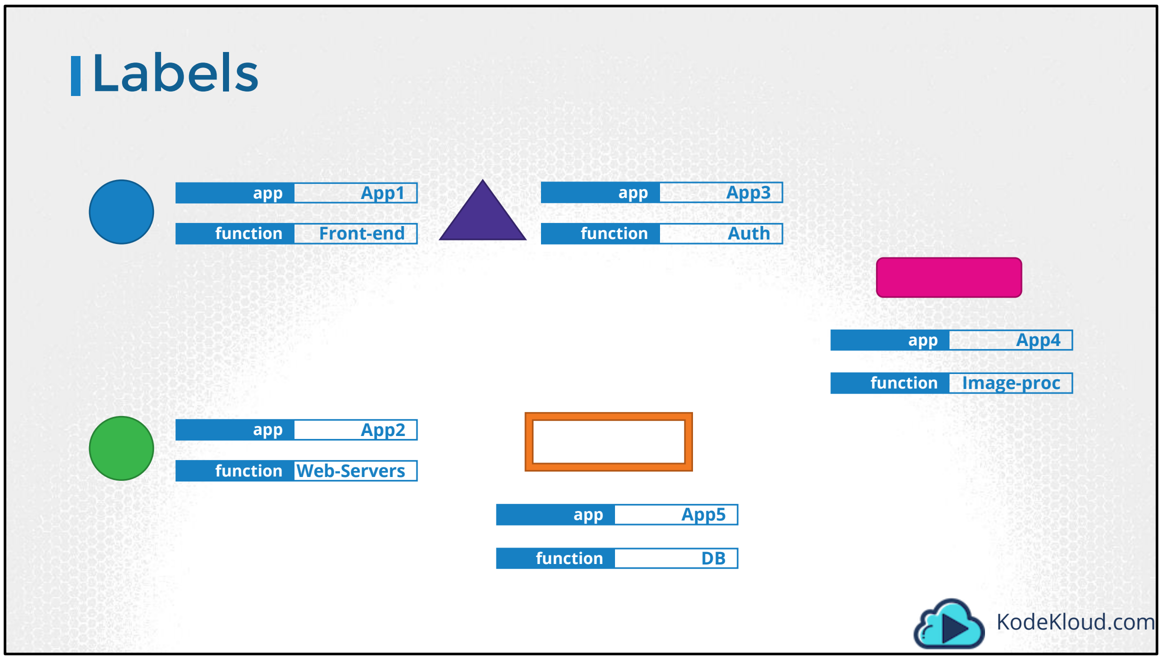


Labels & Selectors in Kubernetes

Or view objects by application.

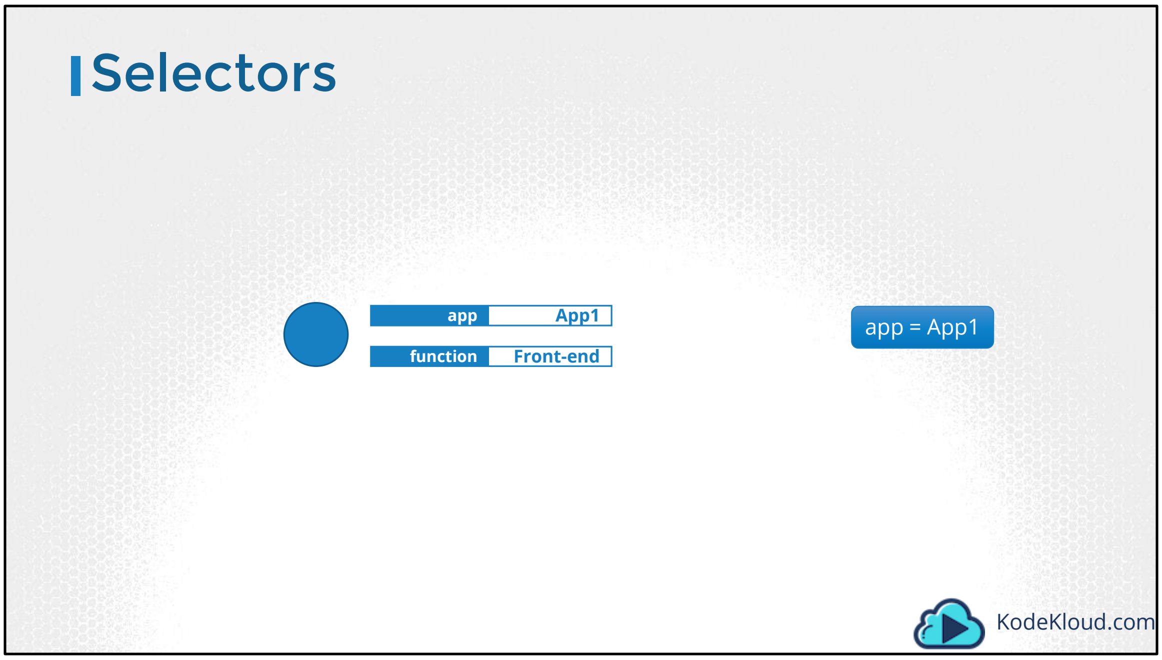

Labels & Selectors in Kubernetes

Or by their functionality. Whatever it may be, you can group and select objects using labels and selectors.



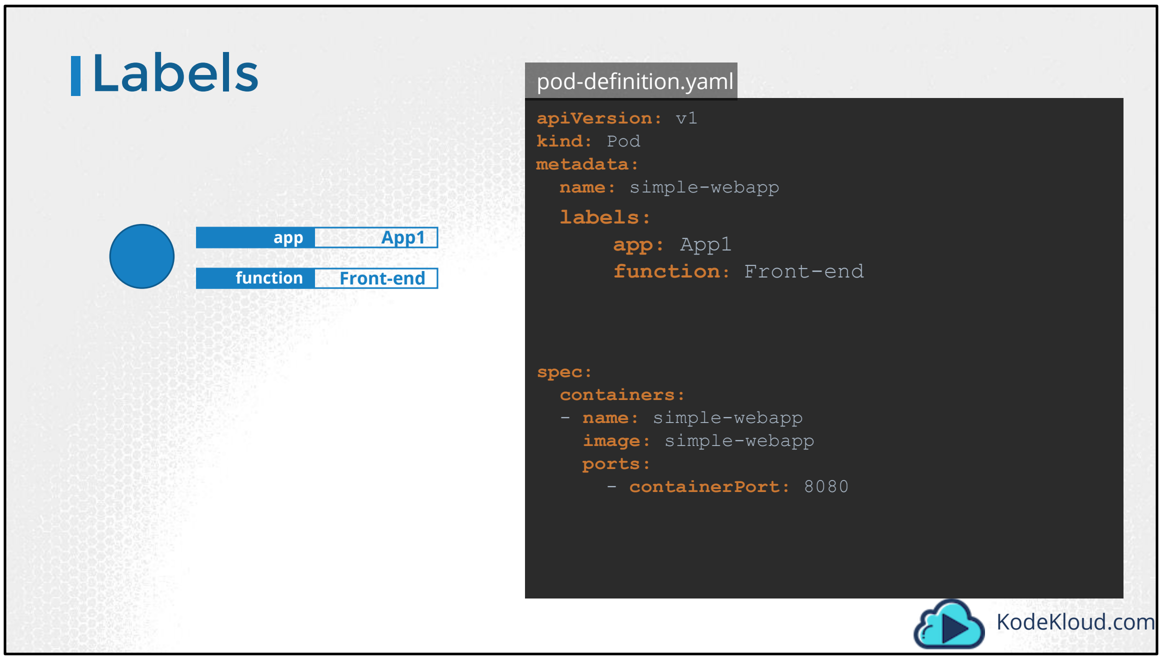
Labels

For each object attach labels as per your needs, like app, function etc.

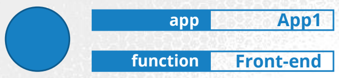


Selectors

Then while selecting, specify a condition to filter specific objects. For example app == App1.



Labels

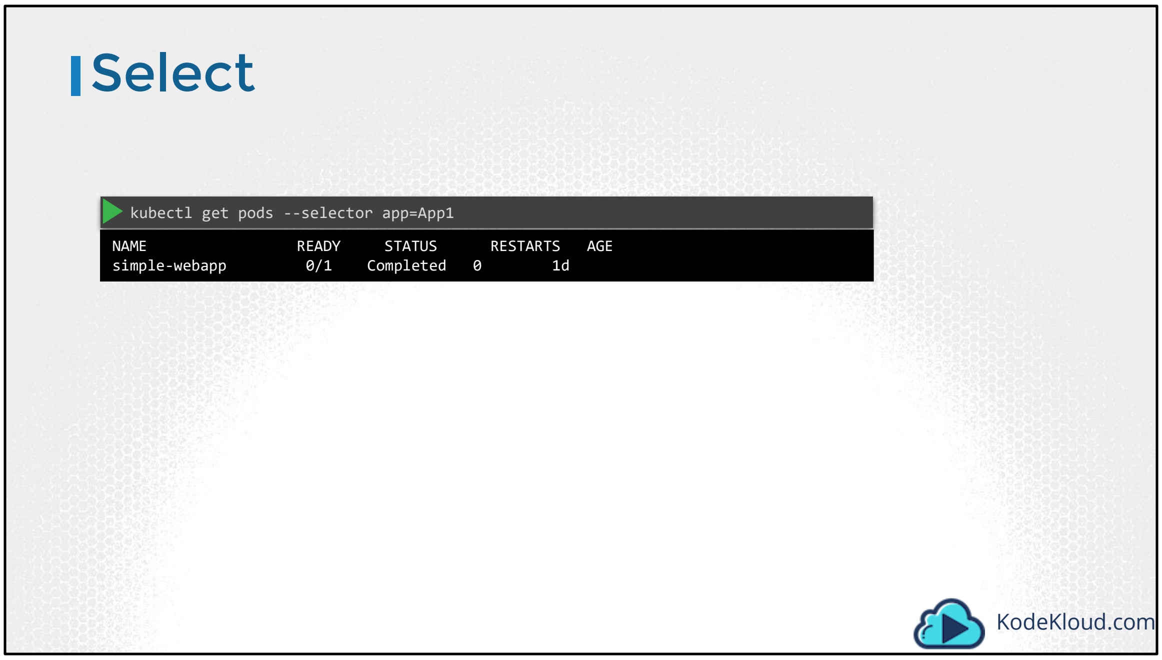


pod-definition.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp
  labels:
      app: App1
      function: Front-end



spec:
  containers:
  - name: simple-webapp
    image: simple-webapp
    ports:
      - containerPort: 8080
```

KodeKloud.com

So how exactly do you specify labels in kubernetes. In a pod-definition file, under metadata, create a section called labels. Under that add the labels in a key value format like this. You can add as many labels as you like.

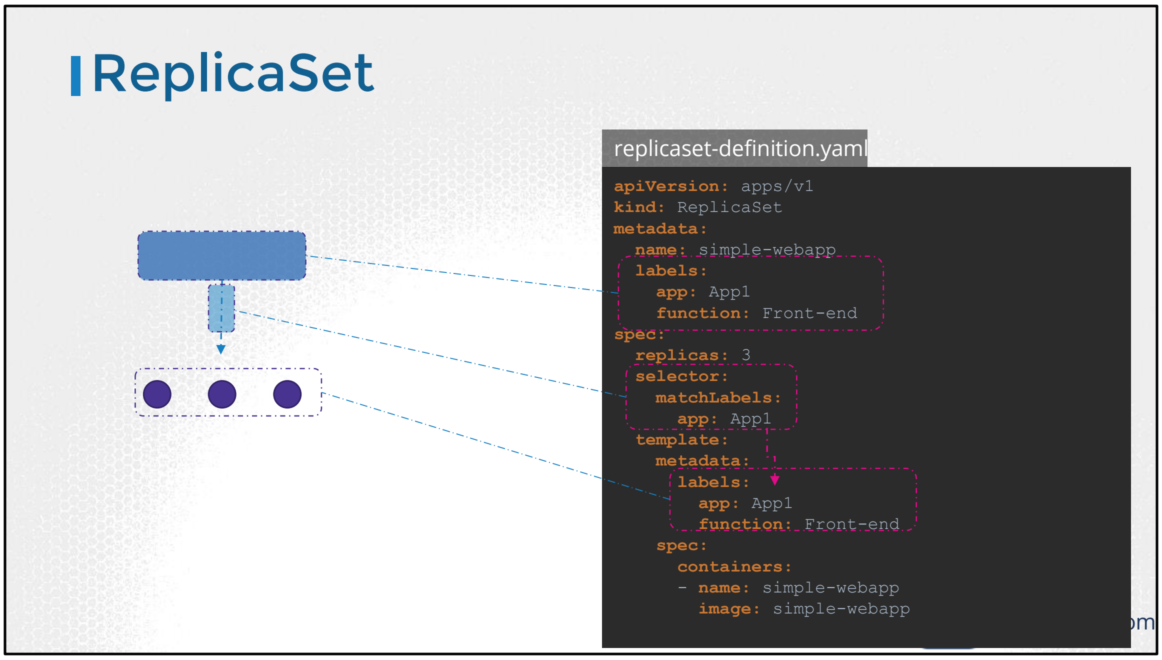


Select

```
kubectl get pods --selector app=App1
NAME                  READY     STATUS       RESTARTS   AGE
simple-webapp          0/1     Completed    0          1d
```

KodeKloud.com

Once the pod is created, to select the pod with the labels use the kubectl get pods command along with the selector option, and specify the condition like app=App1.

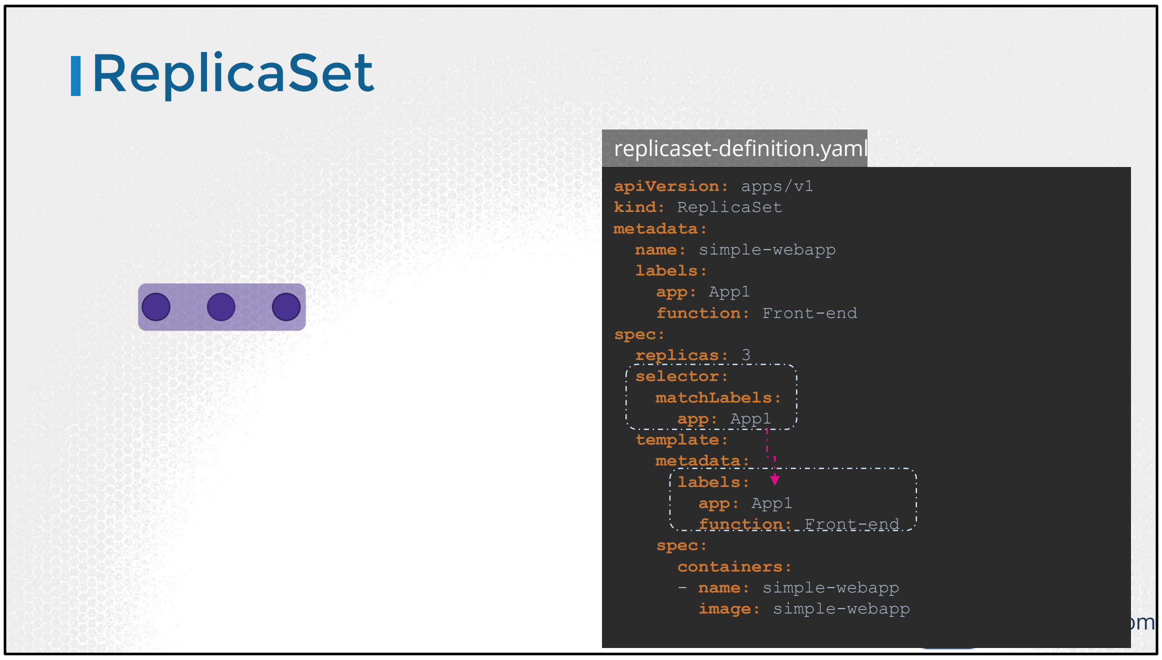


ReplicaSet

replicaset-definition.yaml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: simple-webapp
  labels:
    app: App1
    function: Front-end
spec:
  replicas: 3
  selector:
    matchLabels:
      app: App1
  template:
    metadata:
      labels:
        app: App1
        function: Front-end
    spec:
      containers:
      - name: simple-webapp
        image: simple-webapp
```

Now this is one use case of labels and selectors. Kubernetes objects use labels and selectors internally to connect different objects together. For example to create a replicaset consisting of 3 different pods, we first label the pod definition and use selector in a replicaset to group the pods . In the replica-set definition file, you will see labels defined in two places. Note that this is an area where beginners tend to make a mistake.   The labels defined under the template section are the labels configured on the pods. The labels you see at the top are the labels of the replica set. We are not really concerned about that for now, because we are trying to get the replicaset to discover the pods. The labels on the replicaset will be used if you were configuring some other object to discover the replicaset. In order to connect the replica set to the pods, we configure the selector field under the replicaset specification to match the labels defined on the pod. A single label will do if it matches correctly. However if you feel there could be other pods with that same label but with a different function, then you could specify both the labels to ensure the right pods are discovered by the replicaset.

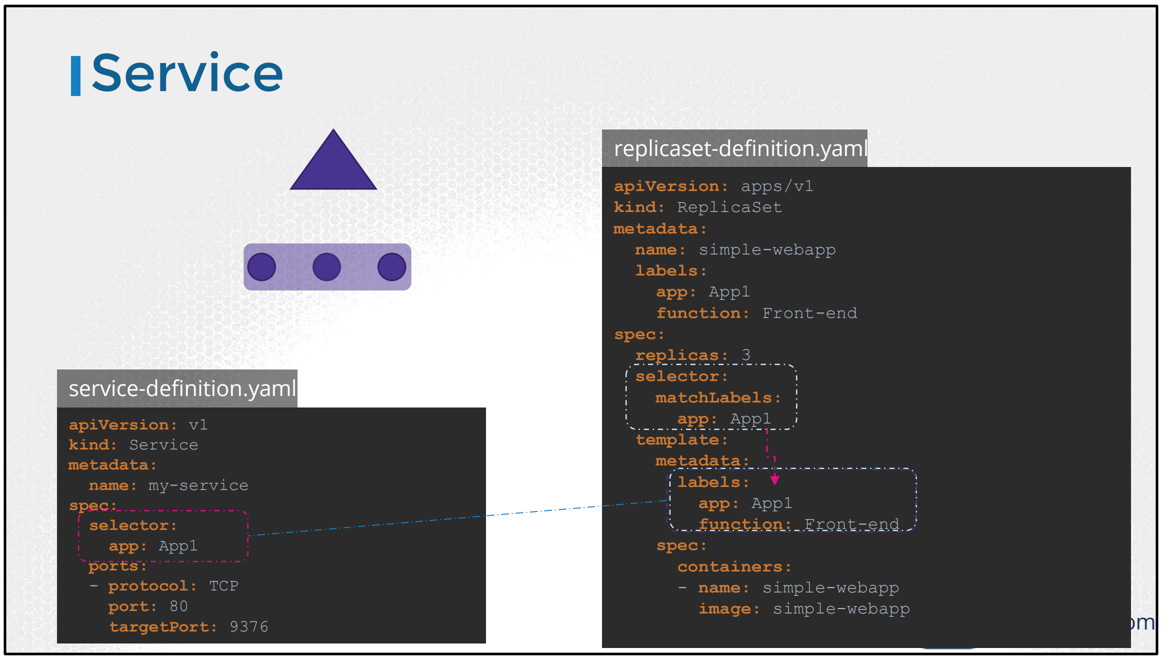


ReplicaSet

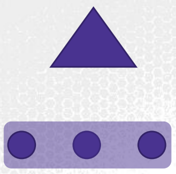
replicaset-definition.yaml

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: simple-webapp
  labels:
    app: App1
    function: Front-end
spec:
  replicas: 3
  selector:
    matchLabels:
      app: App1
  template:
    metadata:
      labels:
        app: App1
        function: Front-end
    spec:
      containers:
      - name: simple-webapp
        image: simple-webapp
```

On creation, if the labels match, the replicaset is created successfully.



Service



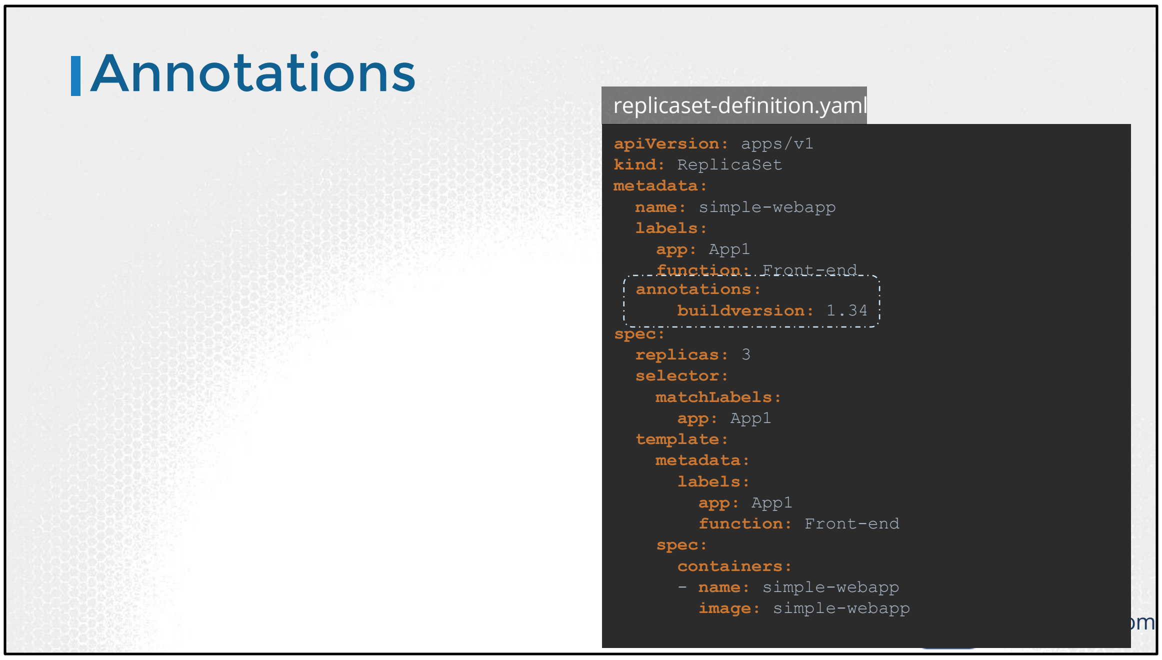
service-definition.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: App1
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

replicaset-definition.yaml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: simple-webapp
  labels:
    app: App1
    function: Front-end
spec:
  replicas: 3
  selector:
    matchLabels:
      app: App1
  template:
    metadata:
      labels:
        app: App1
        function: Front-end
    spec:
      containers:
        - name: simple-webapp
          image: simple-webapp
```

It works the same for other objects like a service. When a service is created, it uses the selector defined in the service definition file to match the labels set on the pods in the replicaset-definition file.



Annotations

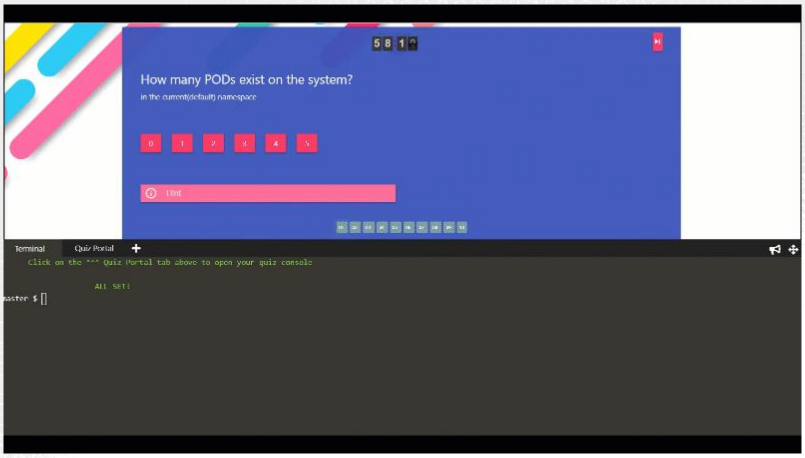
replicaset-definition.yaml

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: simple-webapp
  labels:
    app: App1
    function: Front-end
  annotations:
      buildversion: 1.34
spec:
  replicas: 3
  selector:
    matchLabels:
      app: App1
  template:
    metadata:
      labels:
        app: App1
        function: Front-end
    spec:
      containers:
      - name: simple-webapp
        image: simple-webapp
```

Finally let's look at annotations. While labels and selectors are used to group and select objects, annotations are used to record other details for informatory purpose. For example tool details like name, version build information etc or contact details, phone numbers, email ids etc, that may be used for some kind of integration purpose.

Well, that's it for this lecture on Labels and Selectors. Head over to the coding exercises section and practice working with labels and selectors.

Practice Test


How many PODs exist on the system?

in the current(default) namespace

0 1 2 3 4 5

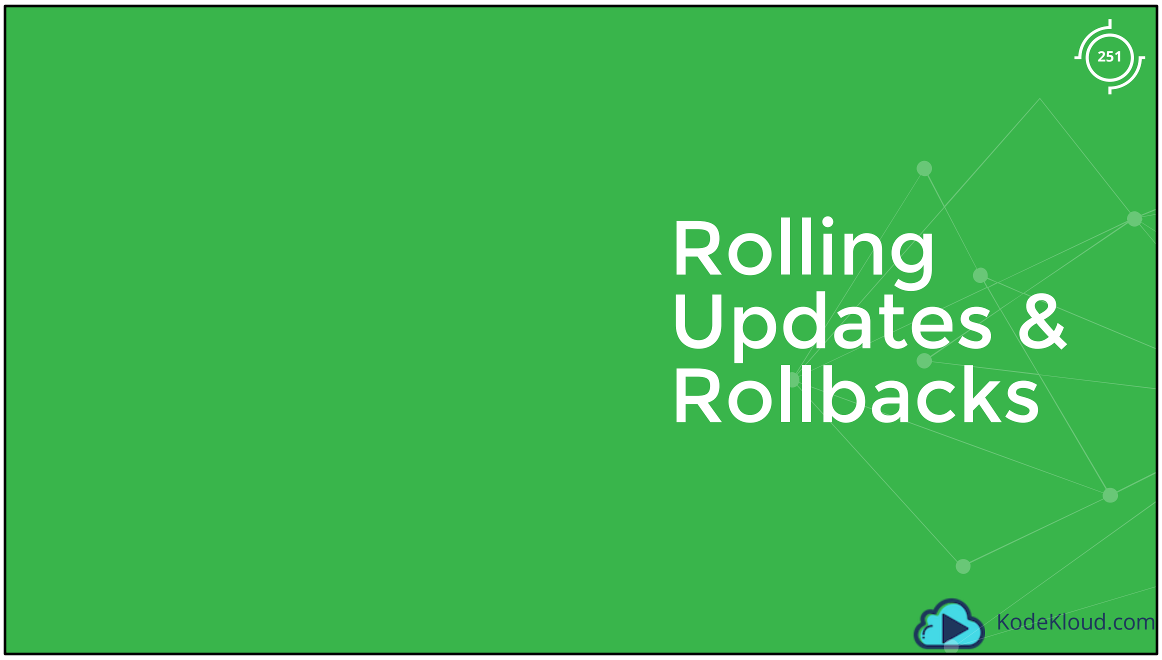
Hint

```
Click on the ^^^ Quiz Portal tab above to open your quiz console

            ALL SET!
master $
```

Check Link Below!

KodeKloud.com

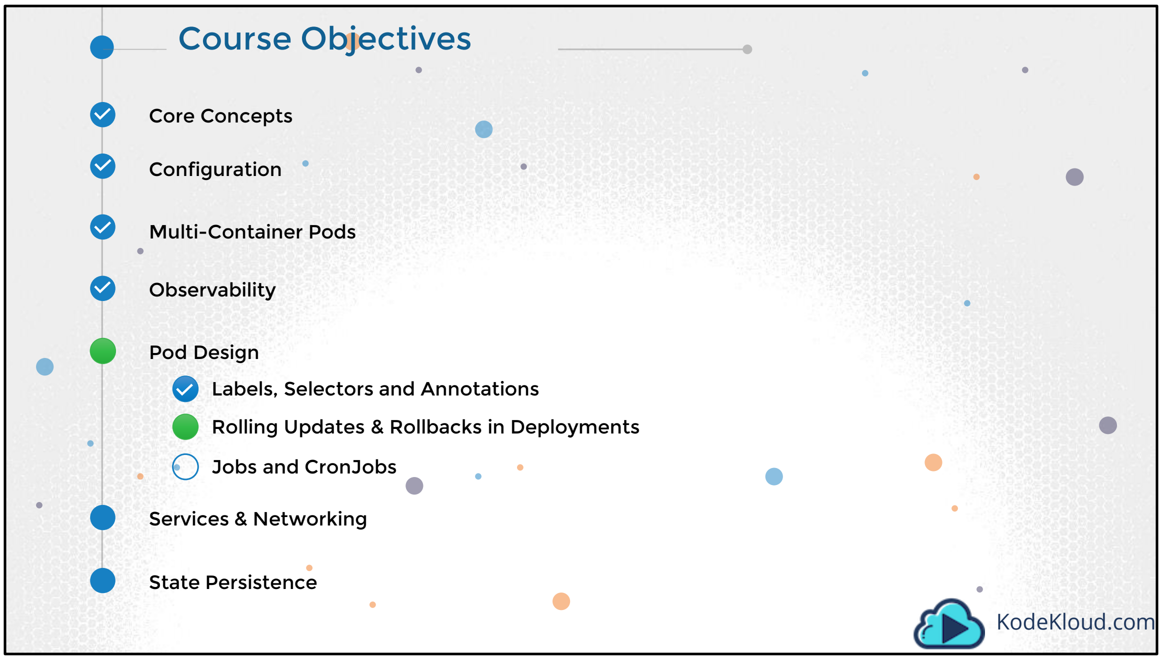
Access Test Here: https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743692



Rolling Updates & Rollbacks

KodeKloud.com

Let us start with Labels and Selectors. What do we know about Labels and Selectors already?



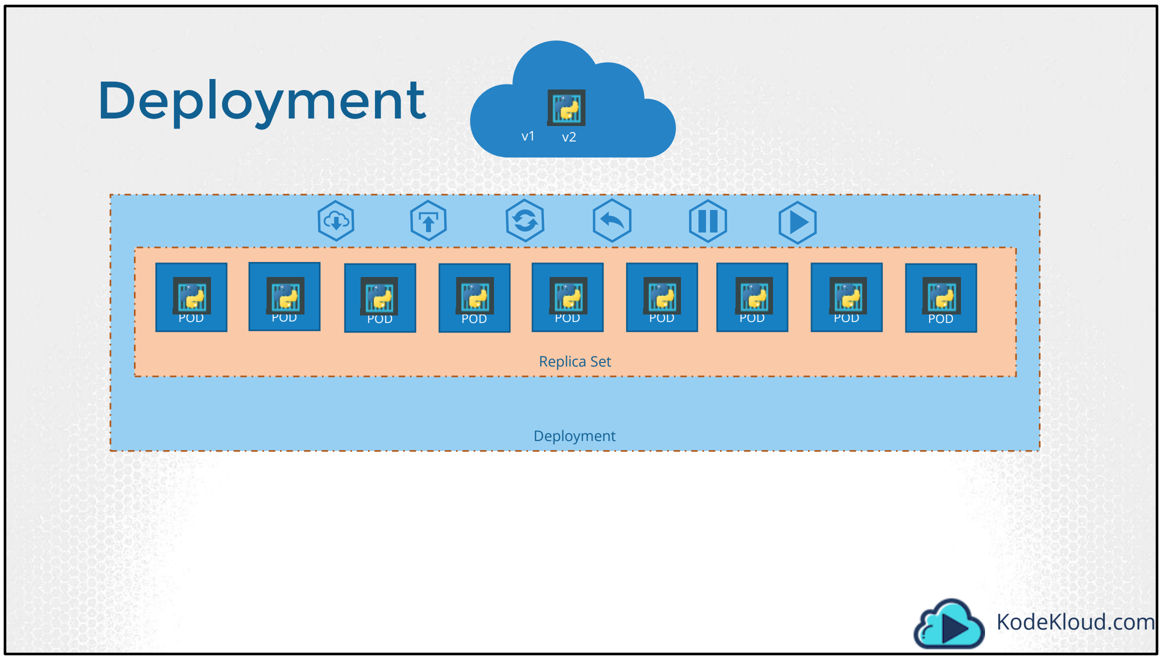
Course Objectives

- ✓ Core Concepts
- ✓ Configuration
- ✓ Multi-Container Pods
- ✓ Observability
- ● Pod Design
  - ✓ Labels, Selectors and Annotations
  - ● Rolling Updates & Rollbacks in Deployments
  - ○ Jobs and CronJobs
- ● Services & Networking
- ● State Persistence

KodeKloud.com

Let us now look at Rolling Updates & Rollbacks in Deployments.

Deployment



For a minute, let us forget about PODs and replicasets and other kubernetes concepts and talk about how you might want to deploy your application in a production environment. Say for example you have a web server that needs to be deployed in a production environment. You need not ONE, but many such instances of the web server running for obvious reasons.

Secondly, when newer versions of application builds become available on the docker registry, you would like to UPGRADE your docker instances seamlessly.

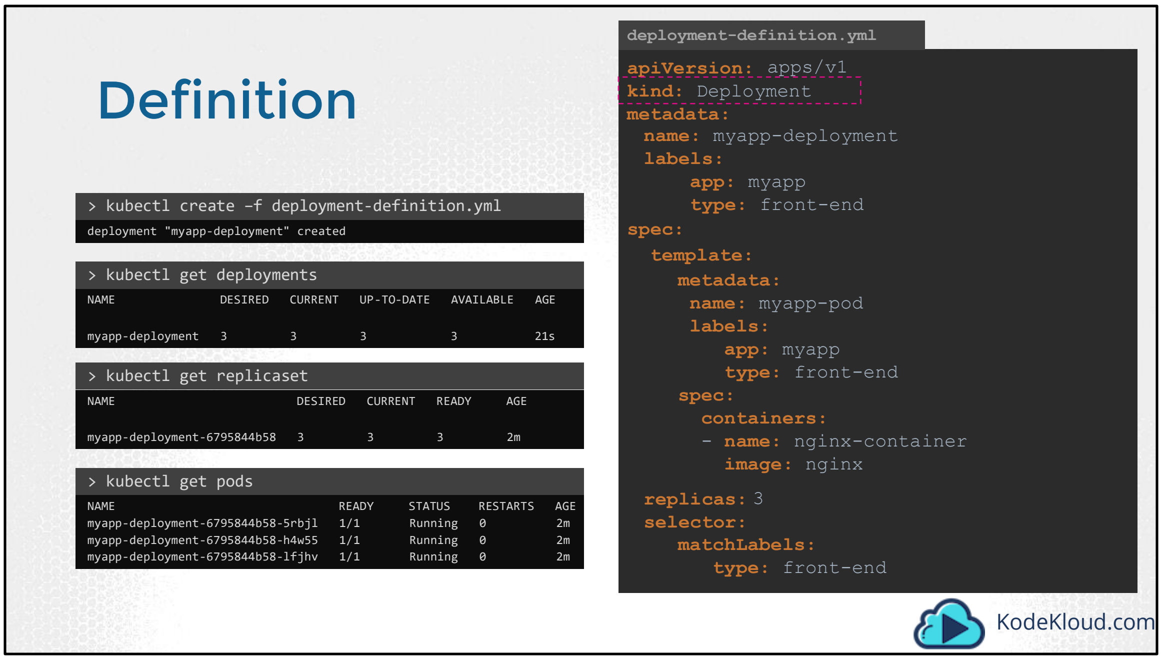
However, when you upgrade your instances, you do not want to upgrade all of them at once as we just did. This may impact users accessing our applications, so you may want to upgrade them one after the other. And that kind of upgrade is known as Rolling Updates.

Suppose one of the upgrades you performed resulted in an unexpected error and you are asked to undo the recent update. You would like to be able to rollBACK the changes that were recently carried out.

Finally, say for example you would like to make multiple changes to your environment such as upgrading the underlying WebServer versions, as well as scaling your environment and also modifying the resource allocations etc. You do not want to apply each change immediately after the command is run, instead you would like to apply a pause to your environment, make the changes and then resume so that all changes are rolled-out together.

All of these capabilities are available with the kubernetes Deployments.

So far in this course we discussed about PODs, which deploy single instances of our application such as the web application in this case. Each container is encapsulated in PODs. Multiple such PODs are deployed using Replication Controllers or Replica Sets. And then comes Deployment which is a kubernetes object that comes higher in the hierarchy. The deployment provides us with capabilities to upgrade the underlying instances seamlessly using rolling updates, undo changes, and pause and resume changes to deployments.



Definition

```
> kubectl create –f deployment-definition.yml
deployment "myapp-deployment" created
```

```
> kubectl get deployments
NAME              DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE

myapp-deployment  3         3         3            3           21s
```

```
> kubectl get replicaset
NAME                         DESIRED   CURRENT   READY   AGE

myapp-deployment-6795844b58  3         3         3       2m
```

```
> kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
myapp-deployment-6795844b58-5rbjl  1/1     Running   0          2m
myapp-deployment-6795844b58-h4w55  1/1     Running   0          2m
myapp-deployment-6795844b58-lfjhv  1/1     Running   0          2m
```

deployment-definition.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
          app: myapp
          type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx

  replicas: 3
  selector:
    matchLabels:
        type: front-end
```

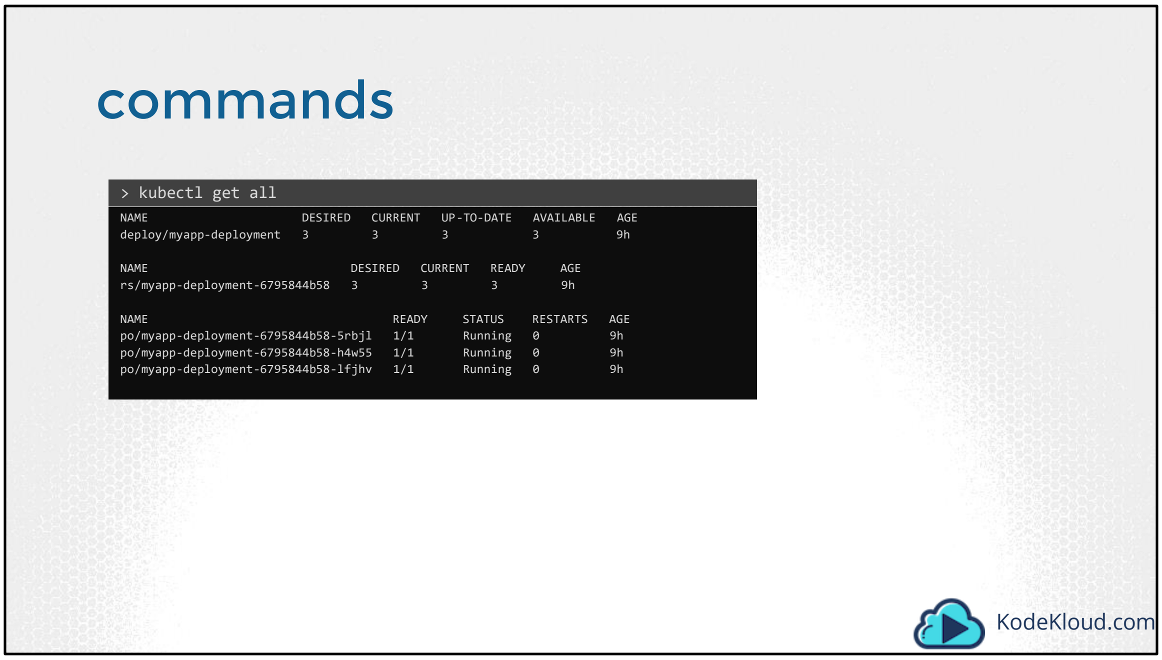
KodeKloud.com

So how do we create a deployment. As with the previous components, we first create a deployment definition file. The contents of the deployment-definition file are exactly similar to the replicaset definition file, except for the kind, which is now going to be Deployment.

If we walk through the contents of the file it has an apiVersion which is apps/v1, metadata which has name and labels and a spec that has template, replicas and selector. The template has a POD definition inside it.

Once the file is ready run the kubectl create command and specify deployment definition file. Then run the kubectl get deployments command to see the newly created deployment. The deployment automatically creates a replica set. So if you run the kubectl get replcaset command you will be able to see a new replicaset in the name of the deployment. The replicasets ultimately create pods, so if you run the kubectl get pods command you will be able to see the pods with the name of the deployment and the replicaset.

So far there hasn't been much of a difference between replicaset and deployments, except for the fact that deployments created a new kubernetes object called deployments. We will see how to take advantage of the deployment using the use cases we discussed in the previous slide in the upcoming lectures.

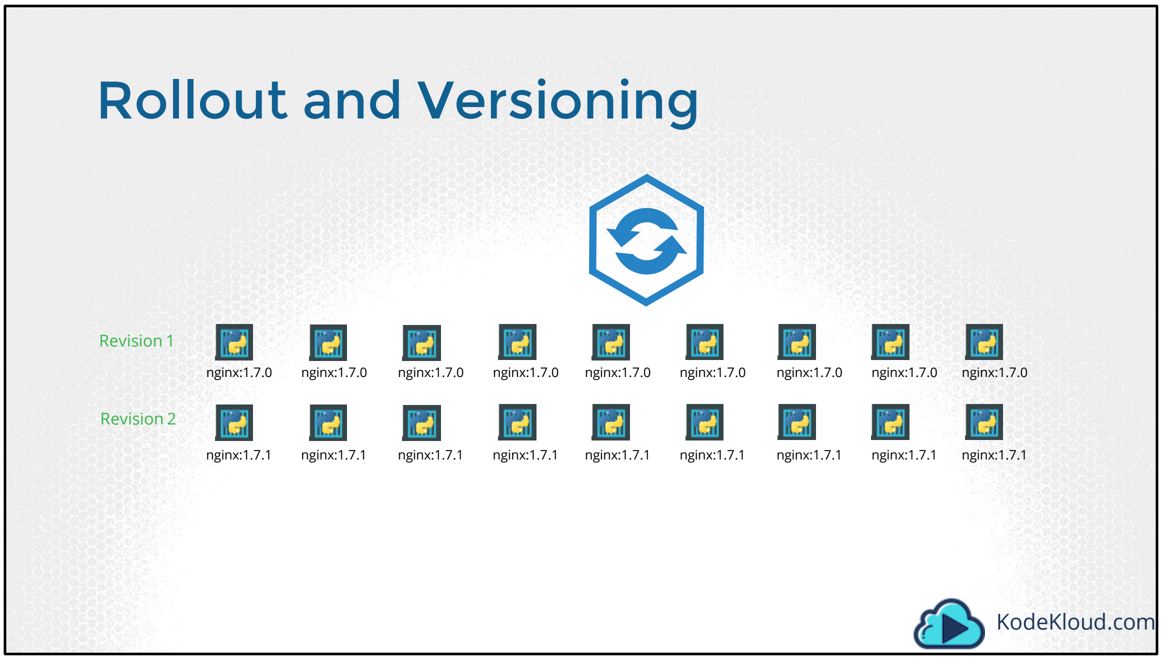


commands

```
> kubectl get all
NAME                     DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/myapp-deployment  3         3         3            3           9h

NAME                               DESIRED   CURRENT   READY     AGE
rs/myapp-deployment-6795844b58     3         3         3         9h

NAME                                     READY     STATUS    RESTARTS   AGE
po/myapp-deployment-6795844b58-5rbjl    1/1       Running   0          9h
po/myapp-deployment-6795844b58-h4w55    1/1       Running   0          9h
po/myapp-deployment-6795844b58-lfjhv    1/1       Running   0          9h
```

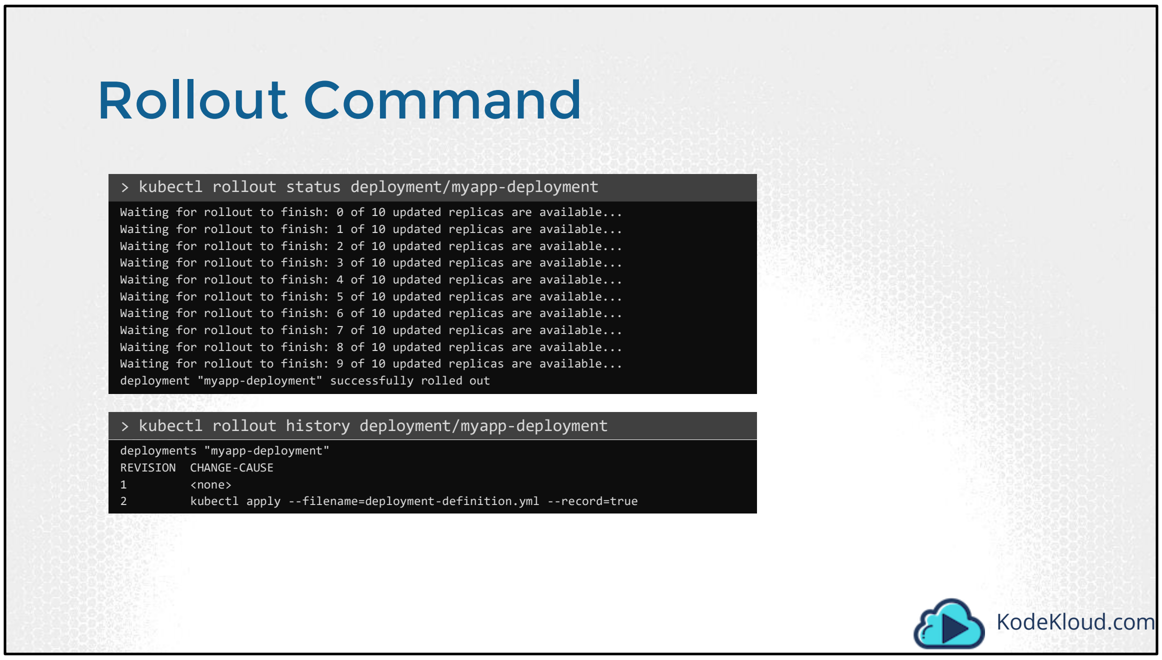
KodeKloud.com

To see all the created objects at once run the kubectl get all command.



Rollout and Versioning

Before we look at how we upgrade our application, let's try to understand Rollouts and Versioning in a deployment. Whenever you create a new deployment or upgrade the images in an existing deployment it triggers a Rollout. A rollout is the process of gradually deploying or upgrading your application containers. When you first create a deployment, it triggers a rollout. A new rollout creates a new Deployment revision. Let's call it revision 1. In the future when the application is upgraded – meaning when the container version is updated to a new one – a new rollout is triggered and a new deployment revision is created named Revision 2. This helps us keep track of the changes made to our deployment and enables us to rollback to a previous version of deployment if necessary.



Rollout Command

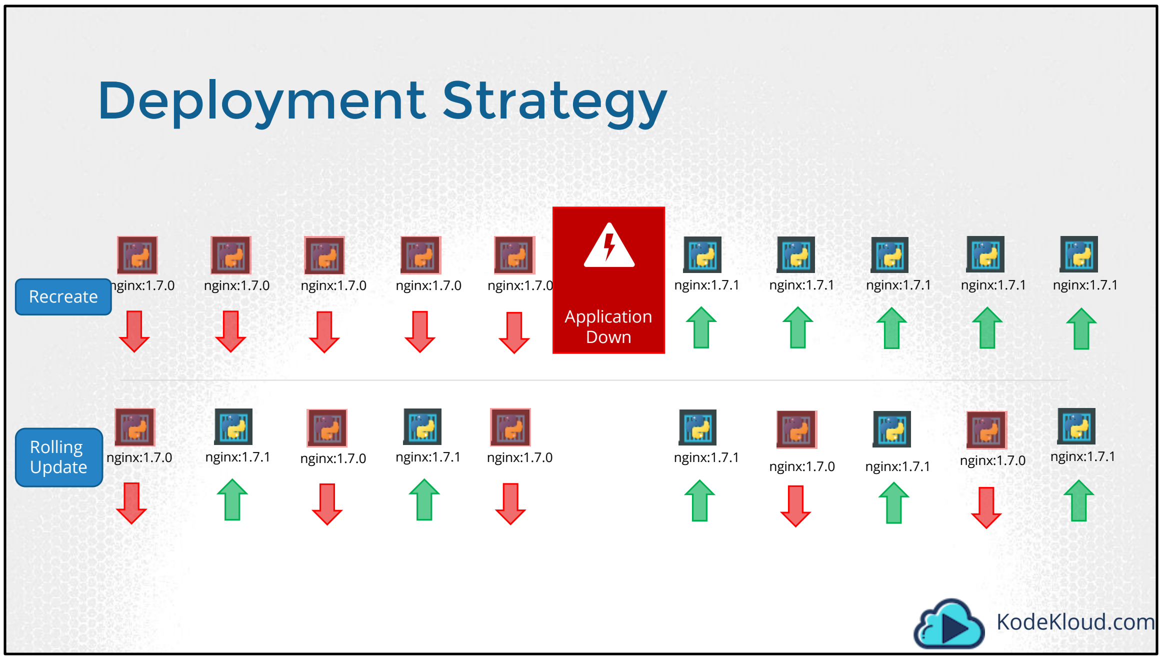
```
> kubectl rollout status deployment/myapp-deployment
Waiting for rollout to finish: 0 of 10 updated replicas are available...
Waiting for rollout to finish: 1 of 10 updated replicas are available...
Waiting for rollout to finish: 2 of 10 updated replicas are available...
Waiting for rollout to finish: 3 of 10 updated replicas are available...
Waiting for rollout to finish: 4 of 10 updated replicas are available...
Waiting for rollout to finish: 5 of 10 updated replicas are available...
Waiting for rollout to finish: 6 of 10 updated replicas are available...
Waiting for rollout to finish: 7 of 10 updated replicas are available...
Waiting for rollout to finish: 8 of 10 updated replicas are available...
Waiting for rollout to finish: 9 of 10 updated replicas are available...
deployment "myapp-deployment" successfully rolled out
```

```
> kubectl rollout history deployment/myapp-deployment
deployments "myapp-deployment"
REVISION  CHANGE-CAUSE
1         <none>
2         kubectl apply --filename=deployment-definition.yml --record=true
```

KodeKloud.com

You can see the status of your rollout by running the command: kubectl rollout status followed by the name of the deployment.

To see the revisions and history of rollout run the command kubectl rollout history followed by the deployment name and this will show you the revisions.

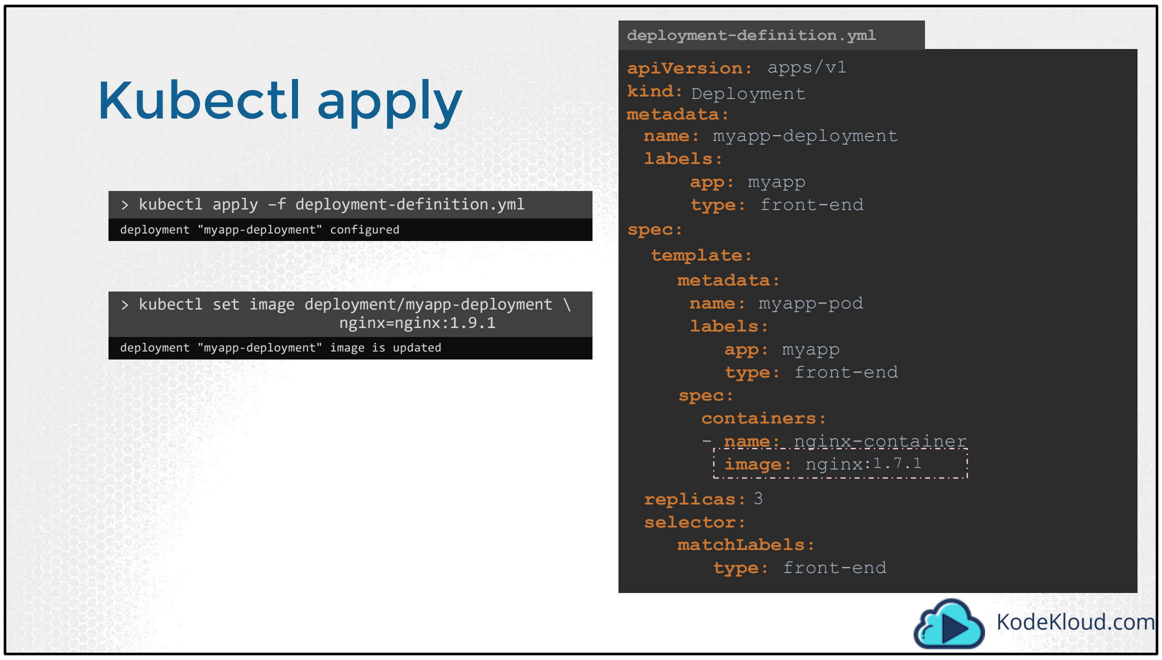


Deployment Strategy

There are two types of deployment strategies. Say for example you have 5 replicas of your web application instance deployed. One way to upgrade these to a newer version is to destroy all of these and then create newer versions of application instances. Meaning first, destroy the 5 running instances and then deploy 5 new instances of the new application version. The problem with this as you can imagine, is that during the period after the older versions are down and before any newer version is up, the application is down and inaccessible to users. This strategy is known as the Recreate strategy, and thankfully this is NOT the default deployment strategy.

The second strategy is were we do not destroy all of them at once. Instead we take down the older version and bring up a newer version one by one. This way the application never goes down and the upgrade is seamless.

Remember, if you do not specify a strategy while creating the deployment, it will assume it to be Rolling Update. In other words, RollingUpdate is the default Deployment Strategy.



Kubectl apply

```
> kubectl apply -f deployment-definition.yml
deployment "myapp-deployment" configured
```

```
> kubectl set image deployment/myapp-deployment \
                    nginx=nginx:1.9.1
deployment "myapp-deployment" image is updated
```

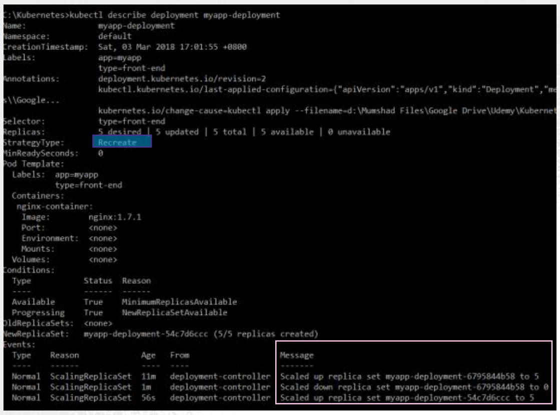
deployment-definition.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
      app: myapp
      type: front-end
spec:
  template:
     metadata:
       name: myapp-pod
       labels:
           app: myapp
           type: front-end
     spec:
       containers:
       - name: nginx-container
         image: nginx:1.7.1
  replicas: 3
  selector:
     matchLabels:
         type: front-end
```

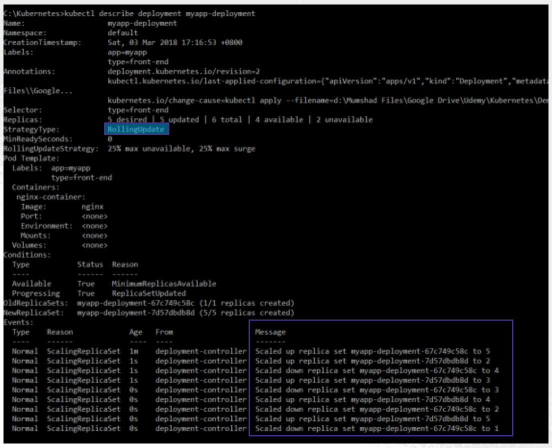
KodeKloud.com

So we talked about upgrades. How exactly DO you update your deployment? When I say update it could be different things such as updating your application version by updating the version of docker containers used, updating their labels or updating the number of replicas etc. Since we already have a deployment definition file it is easy for us to modify this file. Once we make the necessary changes, we run the kubectl apply command to apply the changes. A new rollout is triggered and a new revision of the deployment is created.

But there is ANOTHER way to do the same thing. You could use the kubectl set image command to update the image of your application. But remember, doing it this way will result in the deployment-definition file having a different configuration. So you must be careful when using the same definition file to make changes in the future.



```
C:\Kubernetes>kubectl describe deployment myapp-deployment
Name:                   myapp-deployment
Namespace:              default
CreationTimestamp:      Sat, 03 Mar 2018 17:01:55 +0800
Labels:                 app=myapp
                        type=front-end
Annotations:            deployment.kubernetes.io/revision=2
                        kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"apps/v1","kind":"Deployment","me
s\\Google...
                        kubernetes.io/change-cause=kubectl apply --filename=d:\Mumshad Files\Google Drive\Udemy\Kubernet
Selector:               type=front-end
Replicas:               5 desired | 5 updated | 5 total | 5 available | 0 unavailable
StrategyType:           Recreate
MinReadySeconds:        0
Pod Template:
  Labels:  app=myapp
           type=front-end
  Containers:
   nginx-container:
    Image:        nginx:1.7.1
    Port:         <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   myapp-deployment-54c7d6ccc (5/5 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  11m   deployment-controller  Scaled up replica set myapp-deployment-6795844b58 to 5
  Normal  ScalingReplicaSet  1m    deployment-controller  Scaled down replica set myapp-deployment-6795844b58 to 0
  Normal  ScalingReplicaSet  56s   deployment-controller  Scaled up replica set myapp-deployment-54c7d6ccc to 5
```



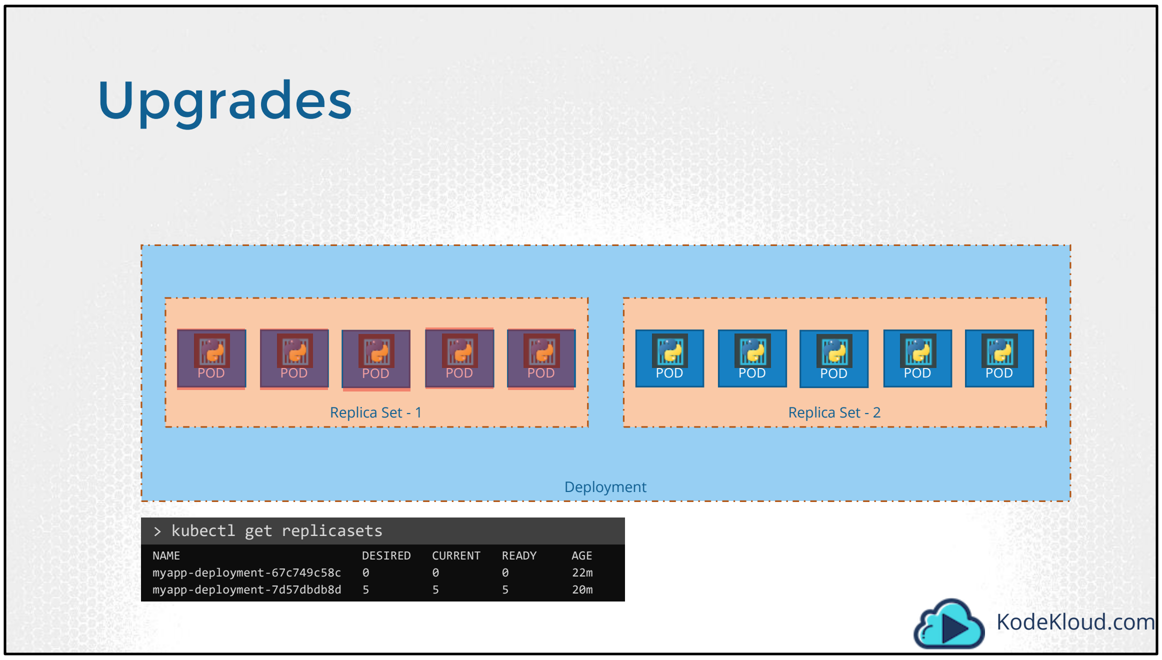
```
C:\Kubernetes>kubectl describe deployment myapp-deployment
Name:                   myapp-deployment
Namespace:              default
CreationTimestamp:      Sat, 03 Mar 2018 17:16:53 +0800
Labels:                 app=myapp
                        type=front-end
Annotations:            deployment.kubernetes.io/revision=2
                        kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"apps/v1","kind":"Deployment","metadata
Files\\Google...
                        kubernetes.io/change-cause=kubectl apply --filename=d:\Mumshad Files\Google Drive\Udemy\Kubernetes\De
Selector:               type=front-end
Replicas:               5 desired | 5 updated | 6 total | 4 available | 2 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=myapp
           type=front-end
  Containers:
   nginx-container:
    Image:        nginx
    Port:         <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    ReplicaSetUpdated
OldReplicaSets:  myapp-deployment-67c749c58c (1/1 replicas created)
NewReplicaSet:   myapp-deployment-7d57dbdb8d (5/5 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  1m    deployment-controller  Scaled up replica set myapp-deployment-67c749c58c to 5
  Normal  ScalingReplicaSet  1s    deployment-controller  Scaled up replica set myapp-deployment-7d57dbdb8d to 2
  Normal  ScalingReplicaSet  1s    deployment-controller  Scaled down replica set myapp-deployment-67c749c58c to 4
  Normal  ScalingReplicaSet  1s    deployment-controller  Scaled up replica set myapp-deployment-7d57dbdb8d to 3
  Normal  ScalingReplicaSet  0s    deployment-controller  Scaled down replica set myapp-deployment-67c749c58c to 3
  Normal  ScalingReplicaSet  0s    deployment-controller  Scaled up replica set myapp-deployment-7d57dbdb8d to 4
  Normal  ScalingReplicaSet  0s    deployment-controller  Scaled down replica set myapp-deployment-67c749c58c to 2
  Normal  ScalingReplicaSet  0s    deployment-controller  Scaled up replica set myapp-deployment-7d57dbdb8d to 5
  Normal  ScalingReplicaSet  0s    deployment-controller  Scaled down replica set myapp-deployment-67c749c58c to 1
```

Recreate

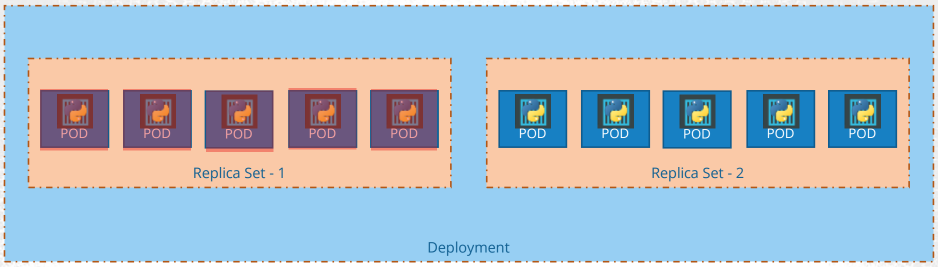
RollingUpdate

KodeKloud.com

The difference between the recreate and rollingupdate strategies can also be seen when you view the deployments in detail. Run the kubectl describe deployment command to see detailed information regarding the deployments. You will notice when the Recreate strategy was used the events indicate that the old replicaset was scaled down to 0 first and the new replica set scaled up to 5. However when the RollingUpdate strategy was used the old replica set was scaled down one at a time simultaneously scaling up the new replica set one at a time.



Upgrades

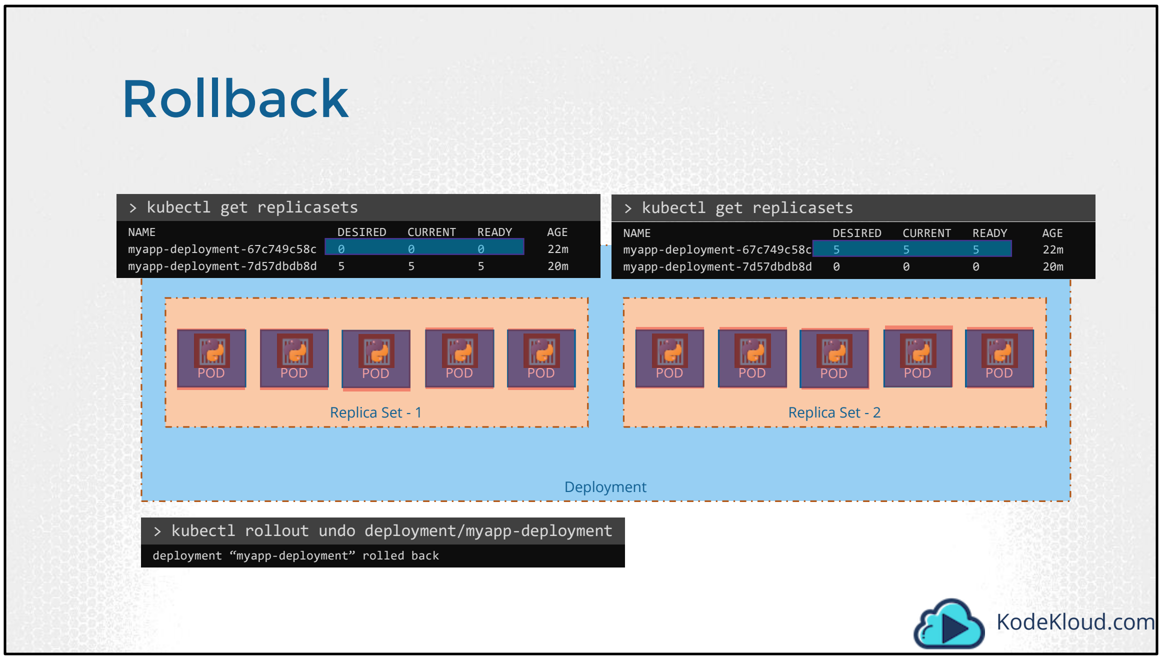


```
> kubectl get replicasets
NAME                          DESIRED   CURRENT   READY    AGE
myapp-deployment-67c749c58c   0         0         0        22m
myapp-deployment-7d57dbdb8d   5         5         5        20m
```

KodeKloud.com

Let's look at how a deployment performs an upgrade under the hoods. When a new deployment is created, say to deploy 5 replicas, it first creates a Replicaset automatically, which in turn creates the number of PODs required to meet the number of replicas. When you upgrade your application as we saw in the previous slide, the kubernetes deployment object creates a NEW replicaset under the hoods and starts deploying the containers there. At the same time taking down the PODs in the old replica-set following a RollingUpdate strategy.

This can be seen when you try to list the replicasets using the kubectl get replicasets command. Here we see the old replicaset with 0 PODs and the new replicaset with 5 PODs.



Rollback

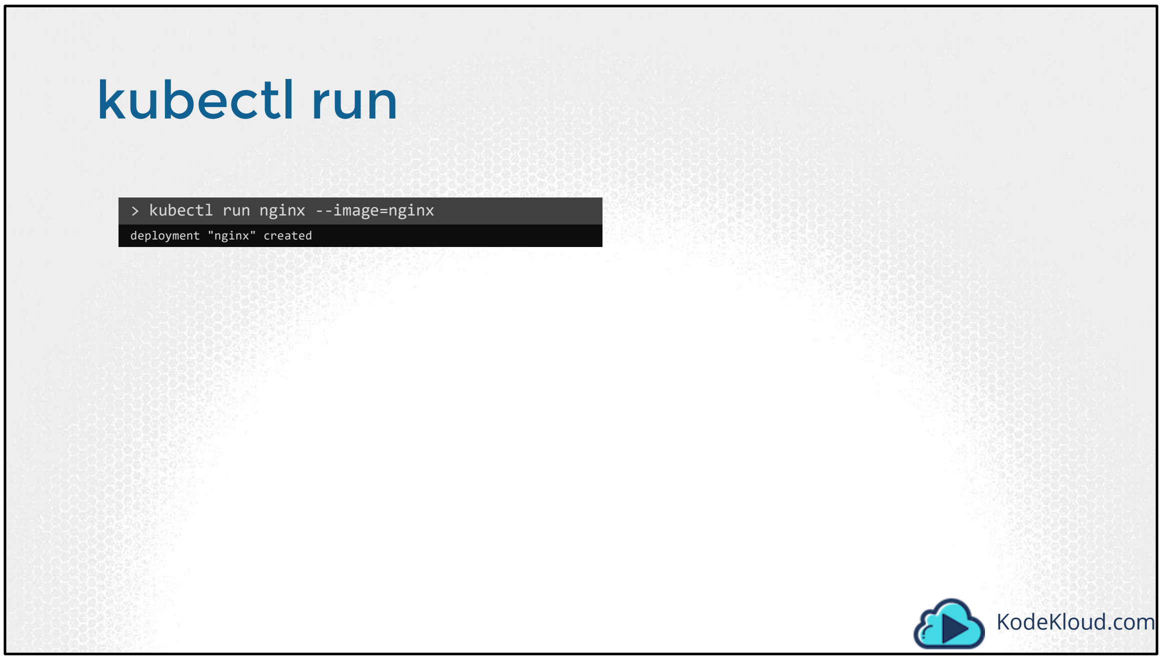
```
> kubectl get replicasets
NAME                           DESIRED   CURRENT   READY   AGE
myapp-deployment-67c749c58c    0         0         0       22m
myapp-deployment-7d57dbdb8d    5         5         5       20m
```

```
> kubectl get replicasets
NAME                           DESIRED   CURRENT   READY   AGE
myapp-deployment-67c749c58c    5         5         5       22m
myapp-deployment-7d57dbdb8d    0         0         0       20m
```

```
> kubectl rollout undo deployment/myapp-deployment
deployment "myapp-deployment" rolled back
```

Say for instance once you upgrade your application, you realize something isn't very right. Something's wrong with the new version of build you used to upgrade. So you would like to rollback your update. Kubernetes deployments allow you to rollback to a previous revision. To undo a change run the command kubectl rollout undo followed by the name of the deployment. The deployment will then destroy the PODs in the new replicaset and bring the older ones up in the old replicaset. And your application is back to its older format.

When you compare the output of the kubectl get replicasets command, before and after the rollback, you will be able to notice this difference. Before the rollback the first replicaset had 0 PODs and the new replicaset had 5 PODs and this is reversed after the rollback is finished.

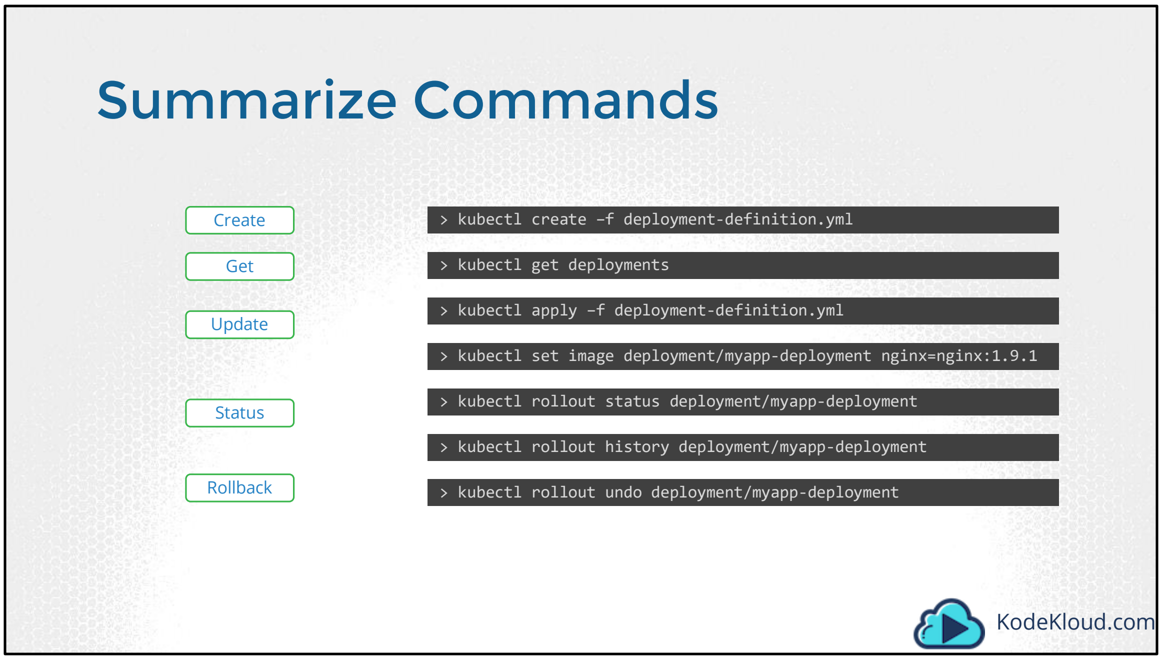


kubectl run

```
> kubectl run nginx --image=nginx
deployment "nginx" created
```

KodeKloud.com

And finally let's get back to one of the commands we ran initially when we learned about PODs for the first time. We used the kubectl run command to create a POD. This command infact creates a deployment and not just a POD. This is why the output of the command says Deployment nginx created. This is another way of creating a deployment by only specifying the image name and not using a definition file. A replicaset and pods are automatically created in the backend. Using a definition file is recommended though as you can save the file, check it into the code repository and modify it later as required.



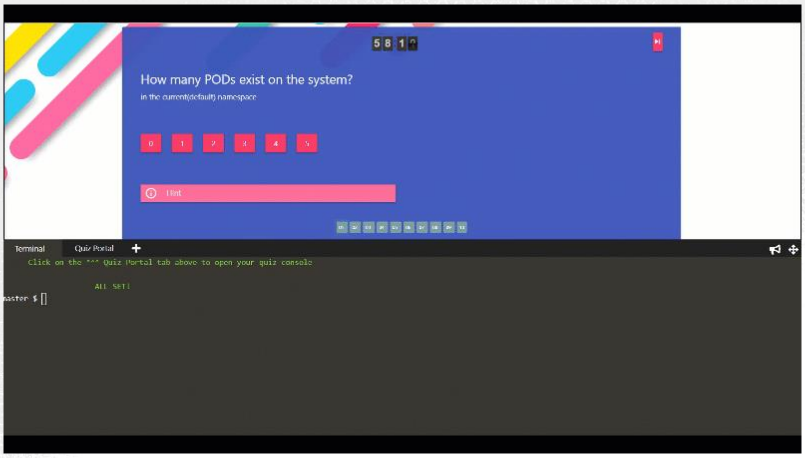
Summarize Commands

Create
Get
Update
Status
Rollback
KodeKloud.com

```
> kubectl create –f deployment-definition.yml
> kubectl get deployments
> kubectl apply –f deployment-definition.yml
> kubectl set image deployment/myapp-deployment nginx=nginx:1.9.1
> kubectl rollout status deployment/myapp-deployment
> kubectl rollout history deployment/myapp-deployment
> kubectl rollout undo deployment/myapp-deployment
```

To summarize the commands real quick, use the kubectl create command to create the deployment, get deployments command to list the deployments, apply and set image commands to update the deployments, rollout status command to see the status of rollouts and rollout undo command to rollback a deployment operation.

Practice Test



How many PODs exist on the system?

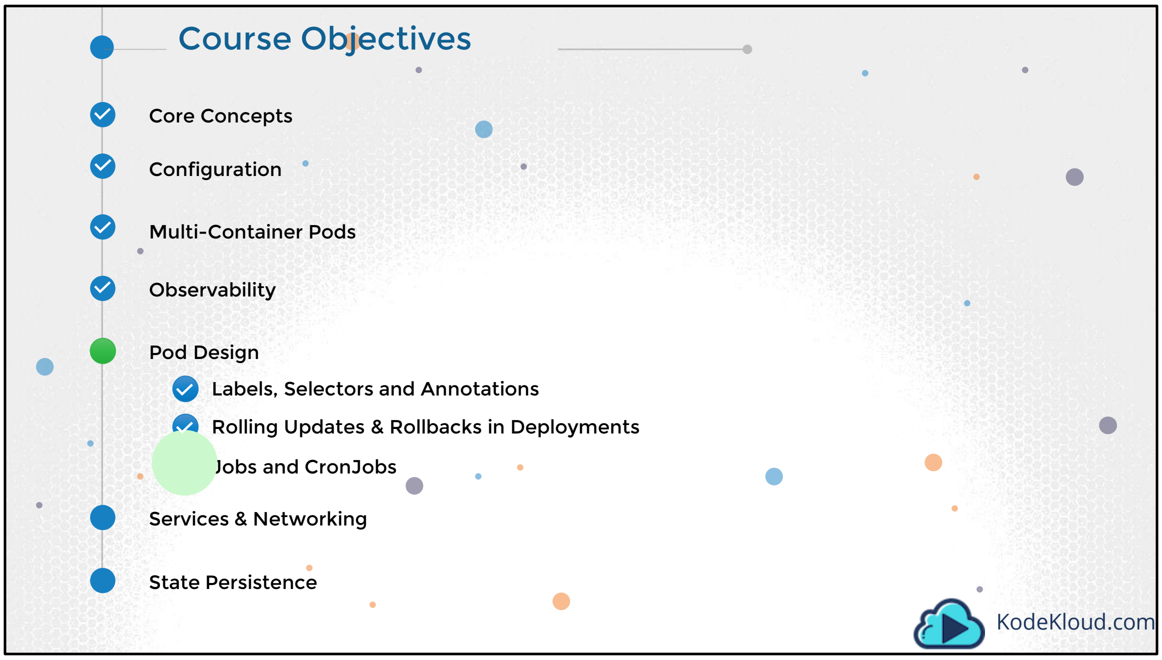
in the current(default) namespace

```
  Click on the ^^^ Quiz Portal tab above to open your quiz console

                 ALL SET!
master $ []
```

Check Link Below!

KodeKloud.com

Access Test Here: https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743696



Course Objectives

- ✓ Core Concepts
- ✓ Configuration
- ✓ Multi-Container Pods
- ✓ Observability
- ● Pod Design
  - ✓ Labels, Selectors and Annotations
  - ✓ Rolling Updates & Rollbacks in Deployments
  - Jobs and CronJobs
- ● Services & Networking
- ● State Persistence

KodeKloud.com

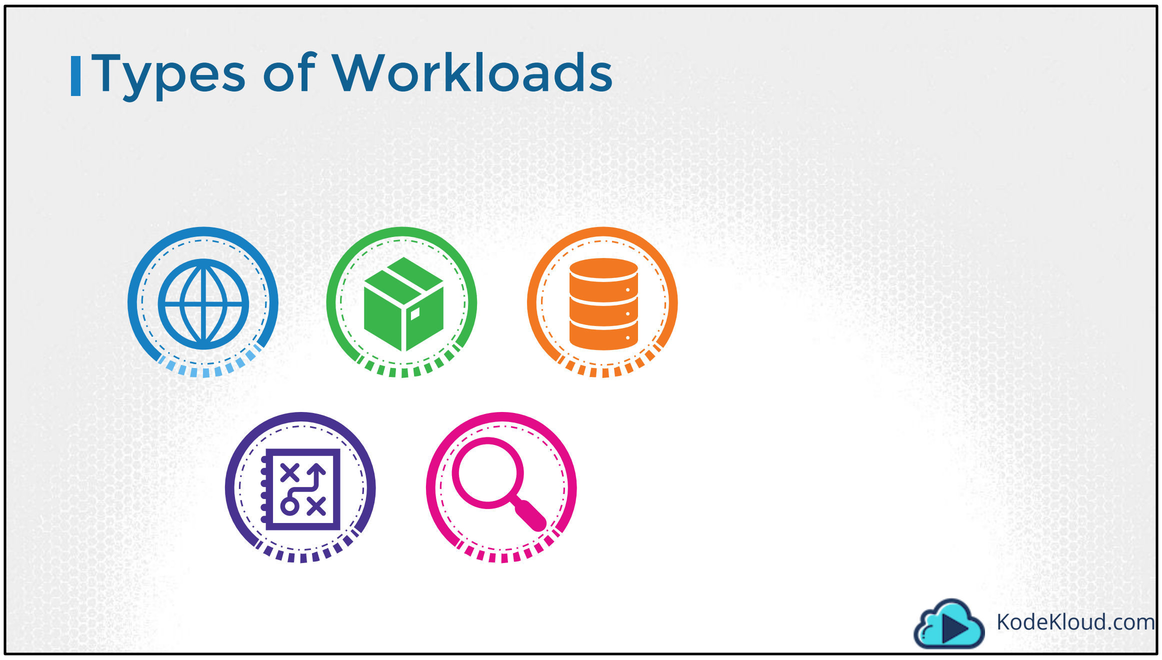
Let us now look at Rolling Updates & Rollbacks in Deployments.

Jobs

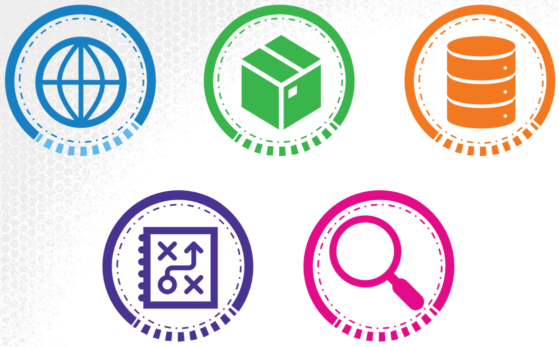
267

KodeKloud.com

Let us start with Jobs in Kubernetes.

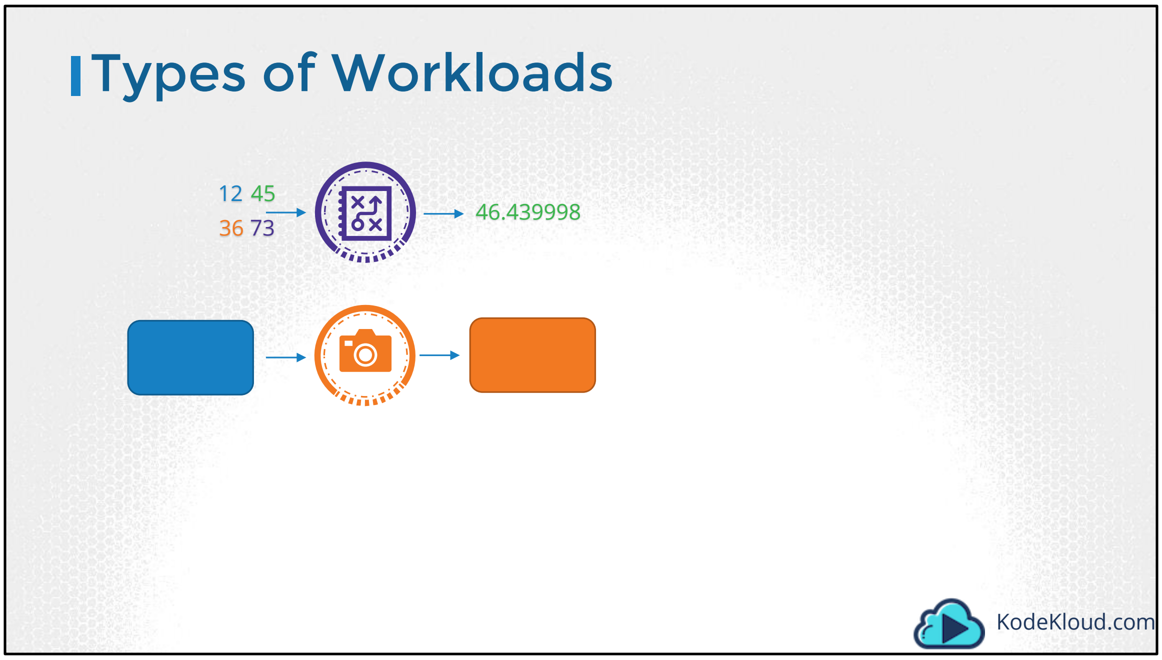


Types of Workloads



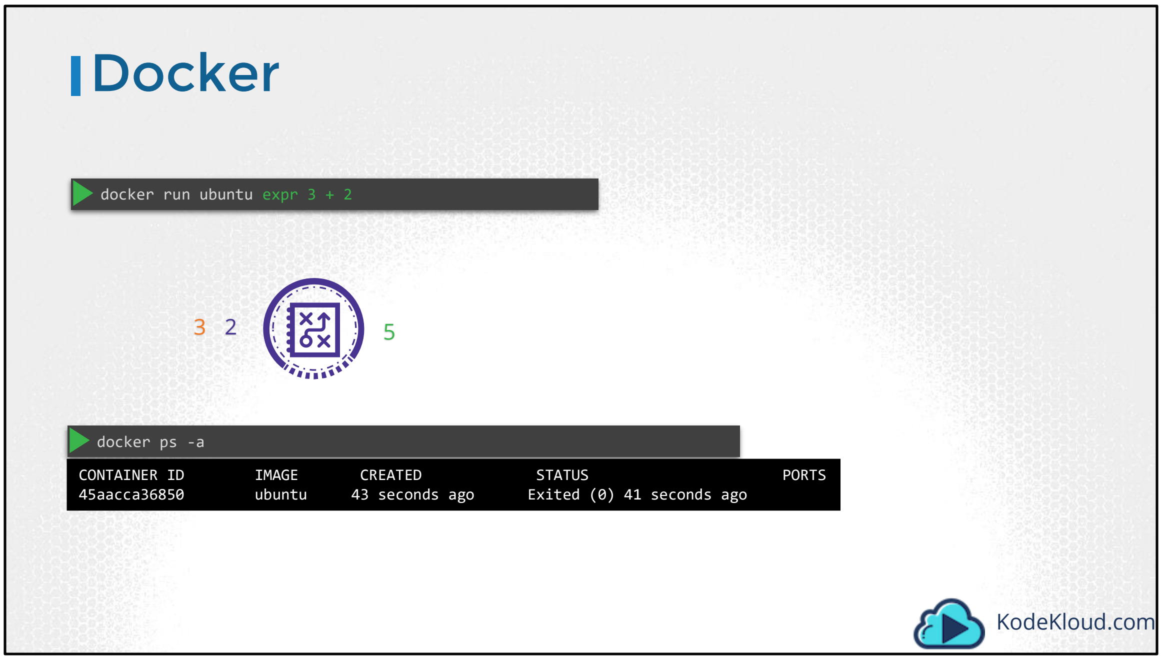
KodeKloud.com

There are different types of workloads that a container can serve. A few that we have seen through this course are Web, application and database. We have deployed simple web servers that serve users. These workloads are meant to continue to run for a long period of time, until manually taken down. There are other kinds of workloads such as batch processing, analytics or reporting that are meant to carry out a specific task and then finish.



Types of Workloads

For example, performing a computation, processing an image, performing some kind of analytics on a large data set, generating a report and sending an email etc. These are workloads that are meant to live for a short period of time, perform a set of tasks and then finish.



Docker

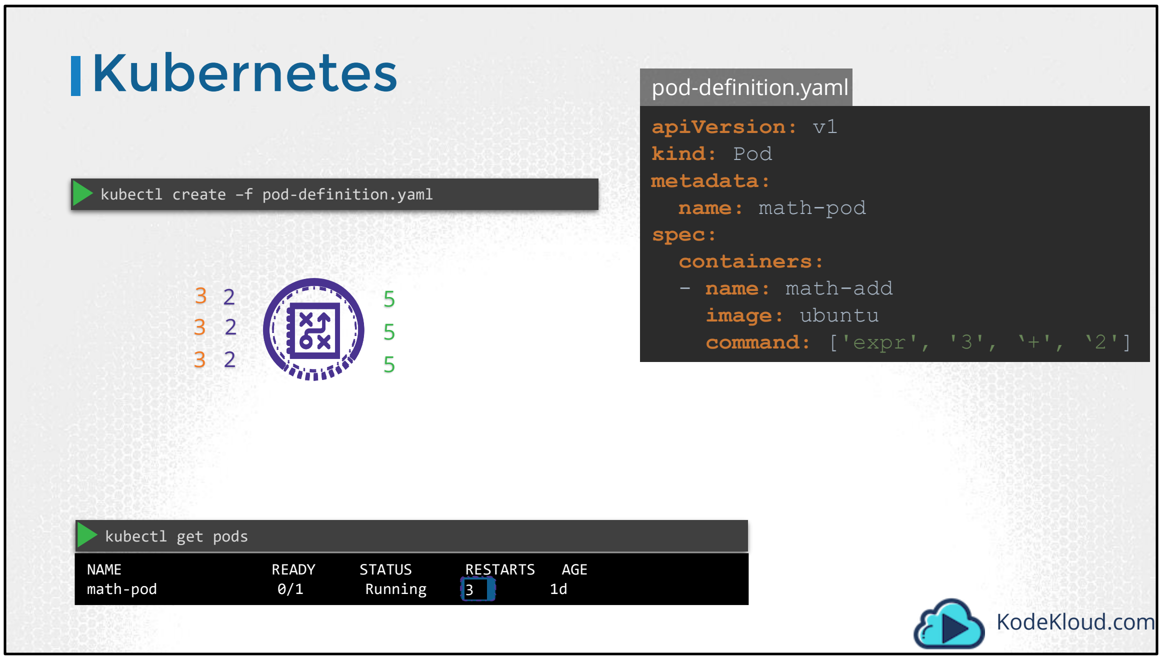
```
docker run ubuntu expr 3 + 2
```

3
2
5

```
docker ps -a
CONTAINER ID        IMAGE       CREATED              STATUS                        PORTS
45aacca36850        ubuntu      43 seconds ago       Exited (0) 41 seconds ago
```

KodeKloud.com

Let us first see how such a work load works in Docker and then we will relate the same concept to Kubernetes. So I am going to run a docker container to perform a simple math operation. To add two numbers. The docker container comes up, performs the requested operation, prints the output and exits. When you run the docker ps command, you see the container in an exited state. The return code of the operation performed is shown in the bracket as well. In this case since the task was completed successfully, the return code is zero.



Kubernetes

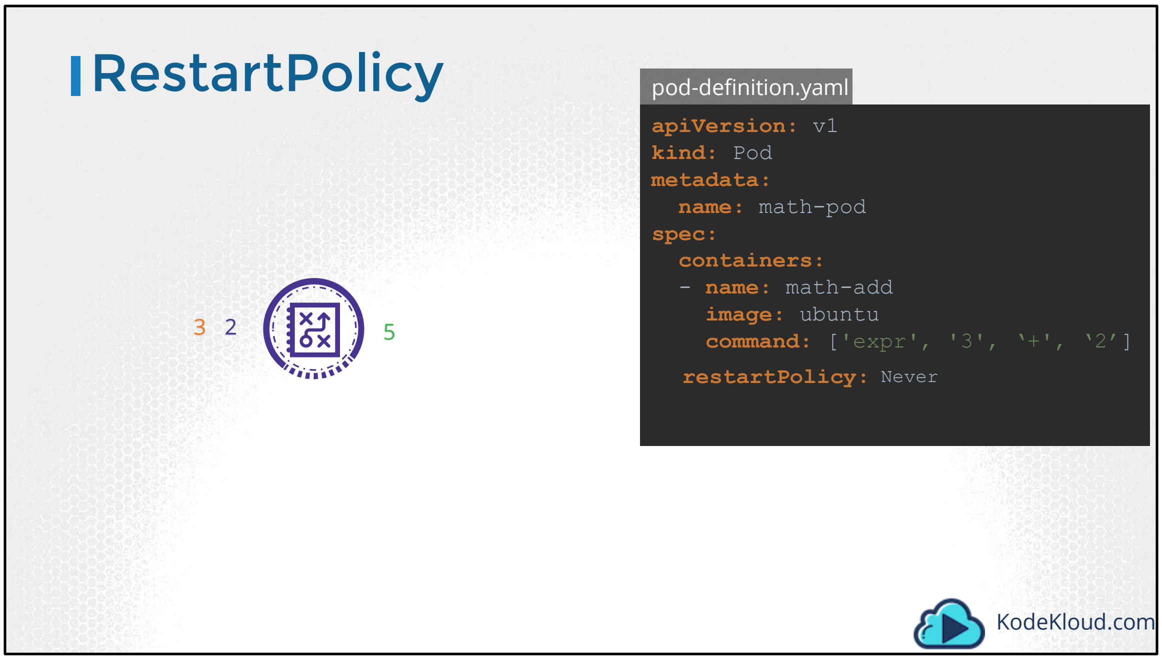
```
kubectl create -f pod-definition.yaml
```

pod-definition.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: math-pod
spec:
  containers:
  - name: math-add
    image: ubuntu
    command: ['expr', '3', '+', '2']
```

```
kubectl get pods
NAME                READY     STATUS      RESTARTS    AGE
math-pod             0/1       Running     3            1d
```

Let us replicate the same with Kubernetes. We will create a pod definition to perform the same operation. When the pod is created, it runs a container performs the computation task and exits and the pod goes into a Completed state. But, It then recreates the container in an attempt to leave it running. Again the container performs the required computation task and exits. And kubernetes brings it up again. And this continuous to happen until a threshold is reached. So why does that happen?



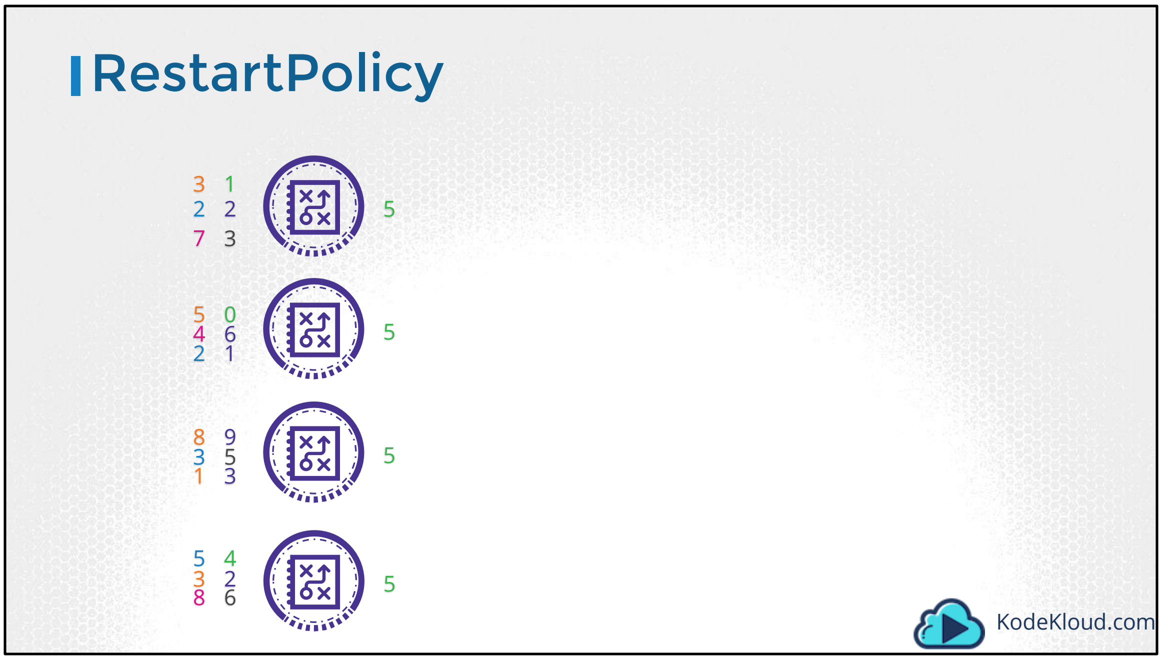
RestartPolicy

pod-definition.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: math-pod
spec:
  containers:
  - name: math-add
    image: ubuntu
    command: ['expr', '3', '+', '2']

  restartPolicy: Never
```

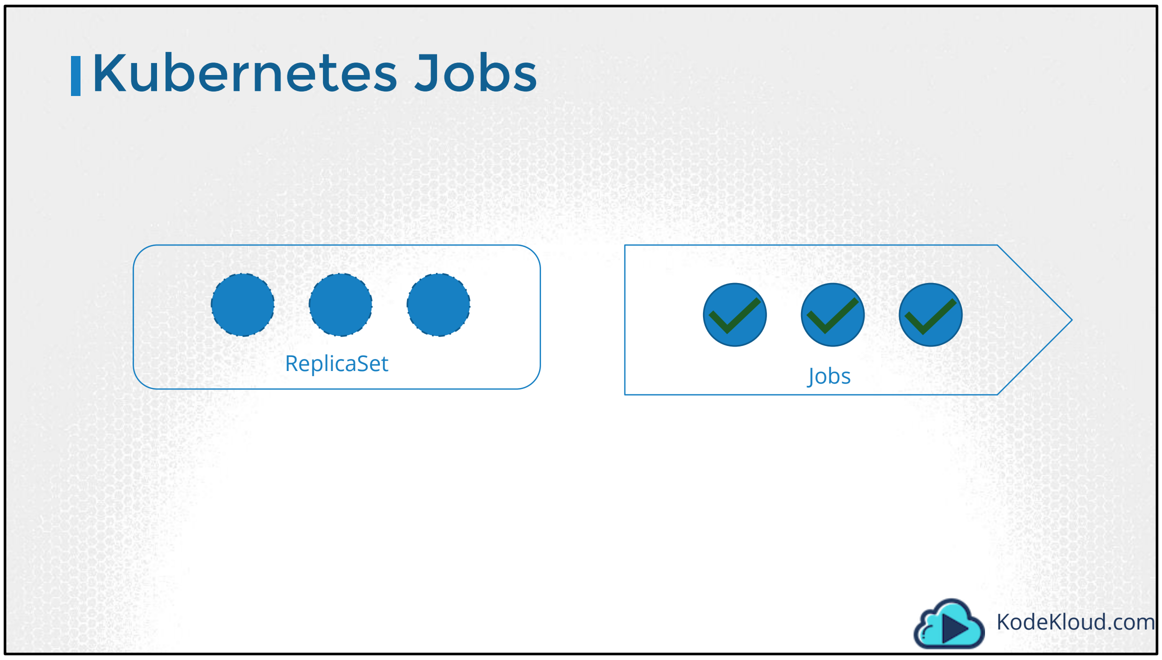
KodeKloud.com

Kubernetes wants your applications to live forever. The default behavior of PODs is to attempt to restart the container in an effort to keep it running. This behavior is defined by the property restartPolicy set on the POD, which is by default set to Always. And that is why the POD ALWAYS recreates the container when it exits. You can override this behavior by setting this property to Never or OnFailure. That way Kubernetes does not restart the container once the job is finished. Now, that works just fine.

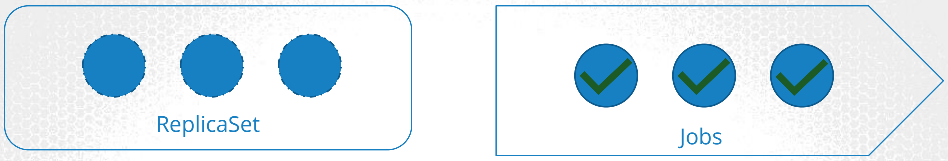


RestartPolicy

We have new use cases for batch processing. We have large data sets that requires multiple pods to process the data in parallel. We want to make sure that all PODs perform the task assigned to them successfully and then exit. So we need a manager that can create as many pods as we want to get a work done and ensure that the work get done successfully.

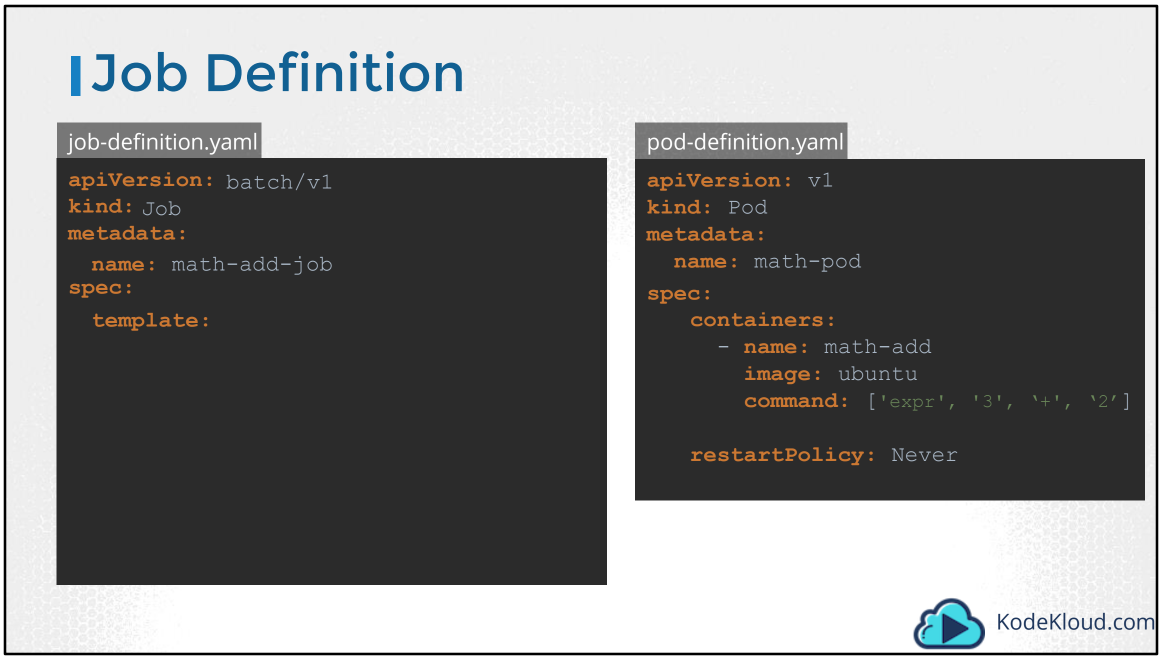


Kubernetes Jobs



KodeKloud.com

That is what JOBs in Kubernetes do. But we have learned about ReplicaSets helping us creating multiple PODs. While a ReplicaSet is used to make sure a specified number of PODs are running at all times, a Job is used to run a set of PODs to perform a given task to completion. Let us now see how we can create a job.



Job Definition

job-definition.yaml

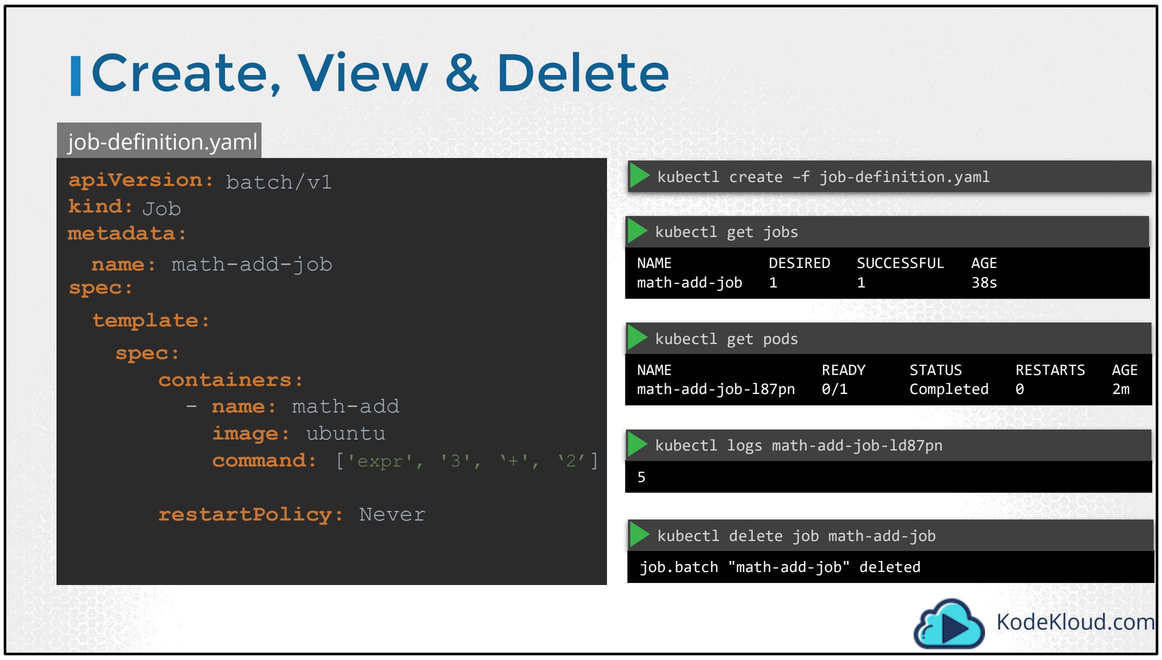
```
apiVersion: batch/v1
kind: Job
metadata:
  name: math-add-job
spec:
  template:
```

pod-definition.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: math-pod
spec:
    containers:
      - name: math-add
        image: ubuntu
        command: ['expr', '3', '+', '2']

    restartPolicy: Never
```

KodeKloud.com

We create a JOB using a definition file. So we will start with a pod definition file. To create a job using it, we start with the blank template that has apiVersion, kind, metadata and spec. The apiVersion is batch/v1 as of today. But remember to verify this against the version of Kubernetes release that you are running on. The kind is Job of course. We will name it math-add-job. Then under the spec section, just like in replicasets or deployments, we have template. And under template we move all of the content from pod definition specification.



Create, View & Delete

job-definition.yaml

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: math-add-job
spec:
  template:
    spec:
      containers:
        - name: math-add
          image: ubuntu
          command: ['expr', '3', '+', '2']

      restartPolicy: Never
```

```
▶ kubectl create -f job-definition.yaml
```

```
▶ kubectl get jobs
NAME           DESIRED   SUCCESSFUL   AGE
math-add-job   1         1            38s
```

```
▶ kubectl get pods
NAME                 READY     STATUS      RESTARTS    AGE
math-add-job-187pn   0/1       Completed   0           2m
```

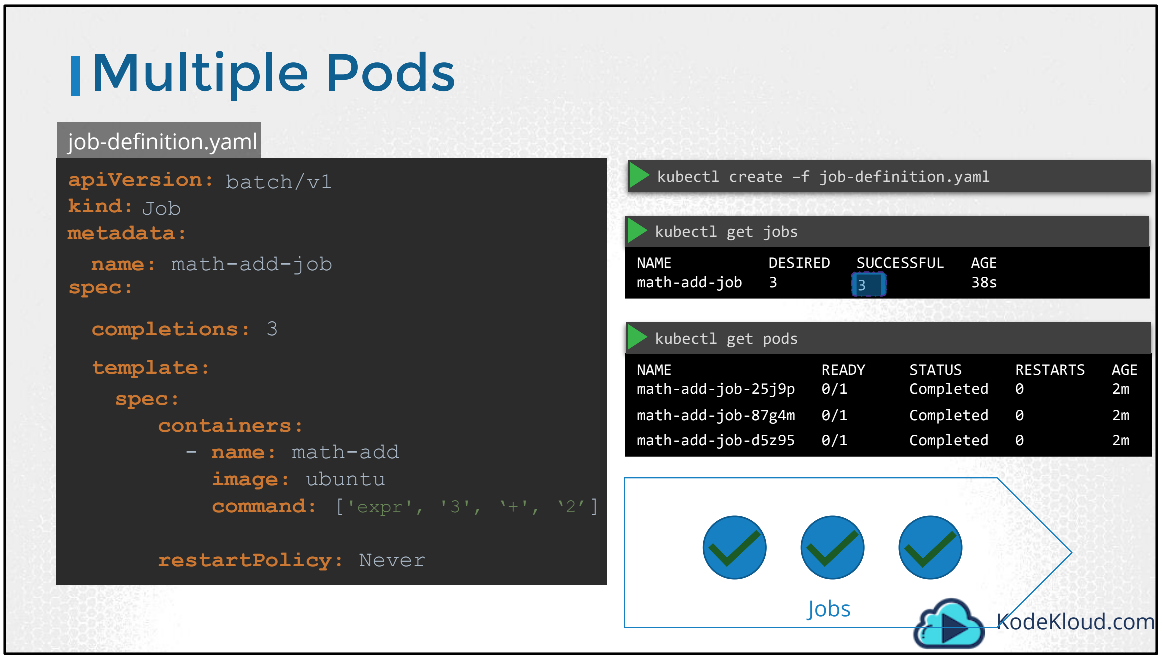
```
▶ kubectl logs math-add-job-ld87pn
5
```

```
▶ kubectl delete job math-add-job
job.batch "math-add-job" deleted
```

KodeKloud.com

Once done create the job using the kubectl create command. Once created, use the kubectl get jobs command to see the newly created job. We now see that the job was created and was completed successfully. To see the pods created by the kubectl get pods command you run kubectl get pods command. We see that it is in a completed state with 0 Restarts, indicating that kubernetes did not try to restart the pod. Perfect! But, what about the output of the job? In our case, we just had the addition performed on the command line inside the container. So the output should be in the pods standard output. The standard output of a container can be seen using the logs command. So we run the kubectl logs command with the name of the pod to see the output. Finally, to delete the job, runthe kubecl delete job command. Deleting the job will also result in deleting the pods that were created by the job.

Now, I hope you realize that this example was made simple so we understand what jobs are and of course this is not typically how jobs are implemented in the real world. For example, if the job was created to process an image, the processed image stored in a persistent volume would be the output or if the job was to generate and email a report, then the email with the report would be the result of the job. So I hope you get the gist of it. And for the sake of understanding jobs, we will continue with this example.



Multiple Pods

job-definition.yaml

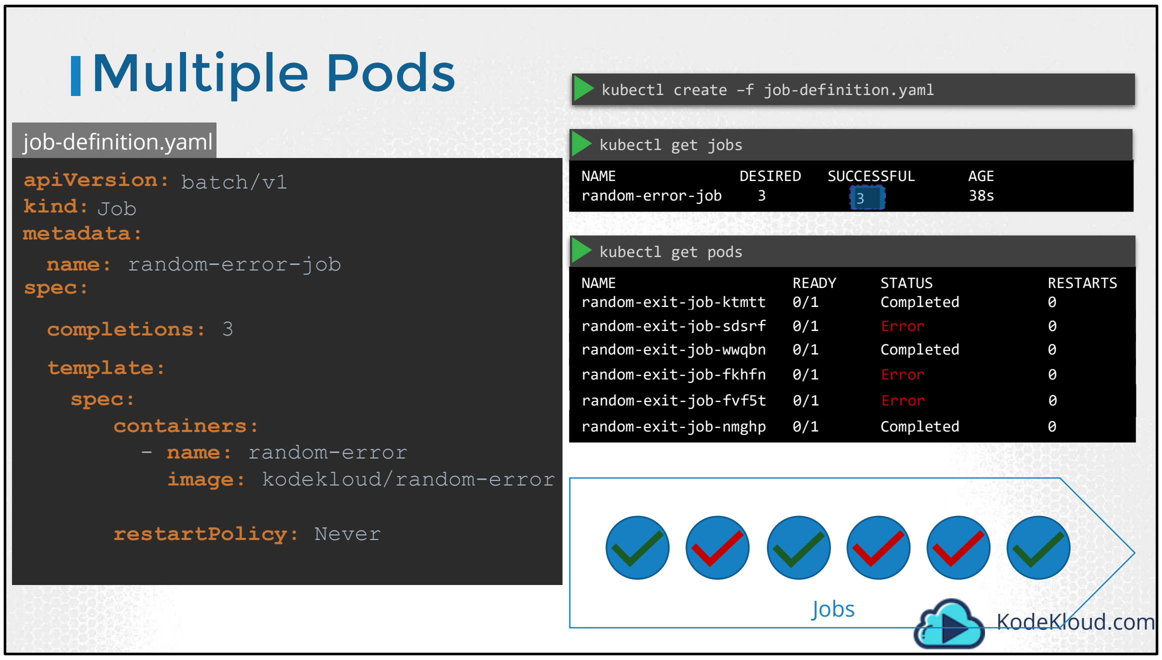
```
apiVersion: batch/v1
kind: Job
metadata:
  name: math-add-job
spec:
  completions: 3
  template:
    spec:
      containers:
        - name: math-add
          image: ubuntu
          command: ['expr', '3', '+', '2']

      restartPolicy: Never
```

```
▶ kubectl create –f job-definition.yaml
```

```
▶ kubectl get jobs
NAME            DESIRED   SUCCESSFUL   AGE
math-add-job    3         3            38s
```

```
▶ kubectl get pods
NAME                 READY    STATUS      RESTARTS   AGE
math-add-job-25j9p   0/1      Completed   0          2m
math-add-job-87g4m   0/1      Completed   0          2m
math-add-job-d5z95   0/1      Completed   0          2m
```

So we just ran one instance of the pod in the previous example. To run multiple pods, we set a value for completions under the job specification. And we set it to 3 to run 3 PODs. This time, when we create the job, We see the Desired count is 3, and the successful count is 0. Now, by default, the PODs are created one after the other. The second pod is created only after the first is finished.



Multiple Pods

job-definition.yaml

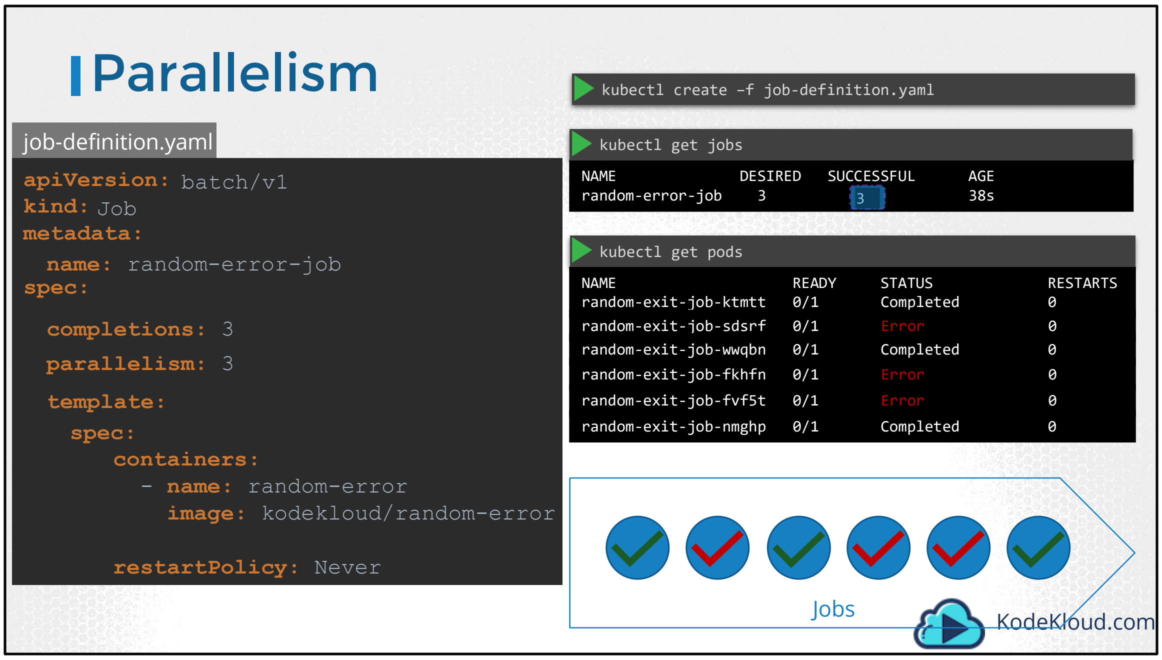
```
apiVersion: batch/v1
kind: Job
metadata:
  name: random-error-job
spec:

  completions: 3

  template:

    spec:

        containers:

          - name: random-error

            image: kodekloud/random-error


        restartPolicy: Never
```

```
▶ kubectl create –f job-definition.yaml
```

```
▶ kubectl get jobs
NAME             DESIRED   SUCCESSFUL     AGE
random-error-job    3          3          38s
```

```
▶ kubectl get pods
NAME                     READY     STATUS          RESTARTS
random-exit-job-ktmtt    0/1       Completed       0
random-exit-job-sdsrf    0/1       Error           0
random-exit-job-wwqbn    0/1       Completed       0
random-exit-job-fkhfn    0/1       Error           0
random-exit-job-fvf5t    0/1       Error           0
random-exit-job-nmghp    0/1       Completed       0
```

That was straight forward. But what if the pods fail? For example, I am now going to create a job using a different image called random-error. It's a simple docker image that randomly completes or fails. When I create this job, first pod completes successfully, the second one fails, so a third one is created and that completes successfully and the fourth one fails, and so does the fifth one and so to have 3 completions, the job creates a new pod which happen to complete successfully. And that completes the job.



Parallelism

job-definition.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: random-error-job
spec:
  completions: 3
  parallelism: 3
  template:
    spec:
      containers:
        - name: random-error
          image: kodekloud/random-error

      restartPolicy: Never
```

```
▶ kubectl create –f job-definition.yaml
```

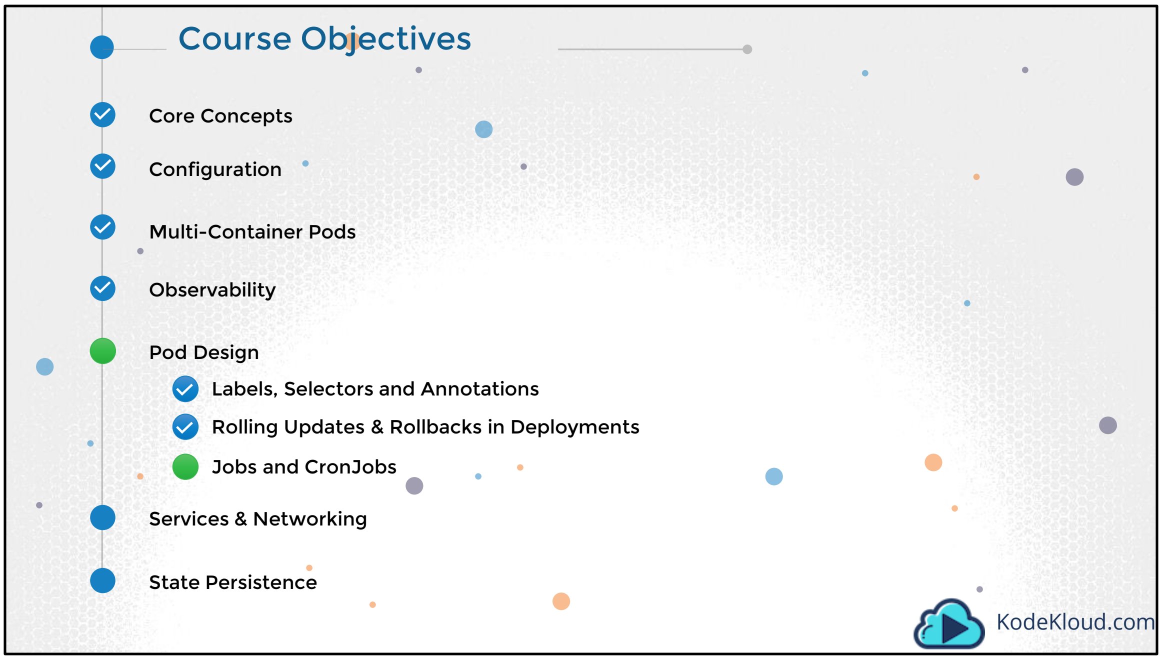
```
▶ kubectl get jobs
NAME              DESIRED   SUCCESSFUL   AGE
random-error-job    3         3          38s
```

```
▶ kubectl get pods
NAME                      READY     STATUS      RESTARTS
random-exit-job-ktmtt     0/1       Completed   0
random-exit-job-sdsrf     0/1       Error       0
random-exit-job-wwqbn     0/1       Completed   0
random-exit-job-fkhfn     0/1       Error       0
random-exit-job-fvf5t     0/1       Error       0
random-exit-job-nmghp     0/1       Completed   0
```

Instead of getting the pods created sequentially we can get them created in parallel. For this add a property called parallelism to the job specification. We set it to 3 to create 3 pods in parallel. So the job first creates 3 pods at once. Two of which completes successfully. So we only need one more, so it's intelligent enough to create one pod at a time until we get a total of 3 completed pods.

<Mention demo if built>

Well that's it for this lecture. Head over to the coding quiz and have fun playing around with jobs. I will see you in the next lecture.



Course Objectives

- ✓ Core Concepts
- ✓ Configuration
- ✓ Multi-Container Pods
- ✓ Observability
- ● Pod Design
  - ✓ Labels, Selectors and Annotations
  - ✓ Rolling Updates & Rollbacks in Deployments
  - ● Jobs and CronJobs
- ● Services & Networking
- ● State Persistence

KodeKloud.com

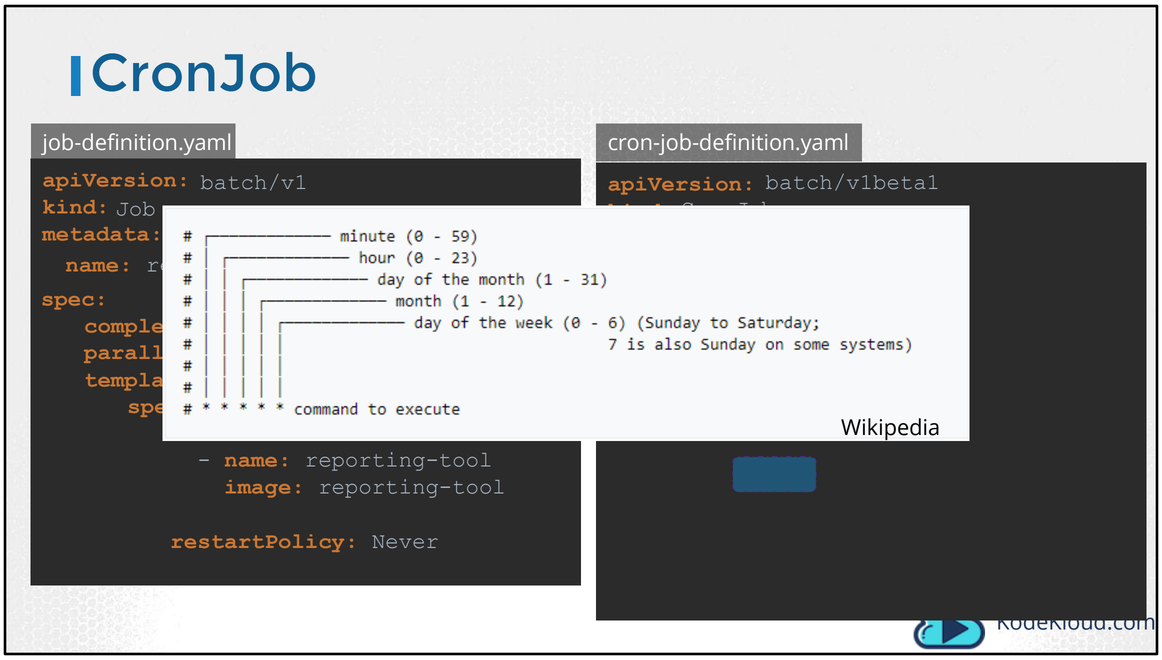
Let us now look at Rolling Updates & Rollbacks in Deployments.

CronJobs

281

KodeKloud.com

Let us now look at CronJobs in Kubernetes.



CronJob

job-definition.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: r
spec:
    comple
    parall
    templa
       spe
          - name: reporting-tool
            image: reporting-tool

         restartPolicy: Never
```

cron-job-definition.yaml

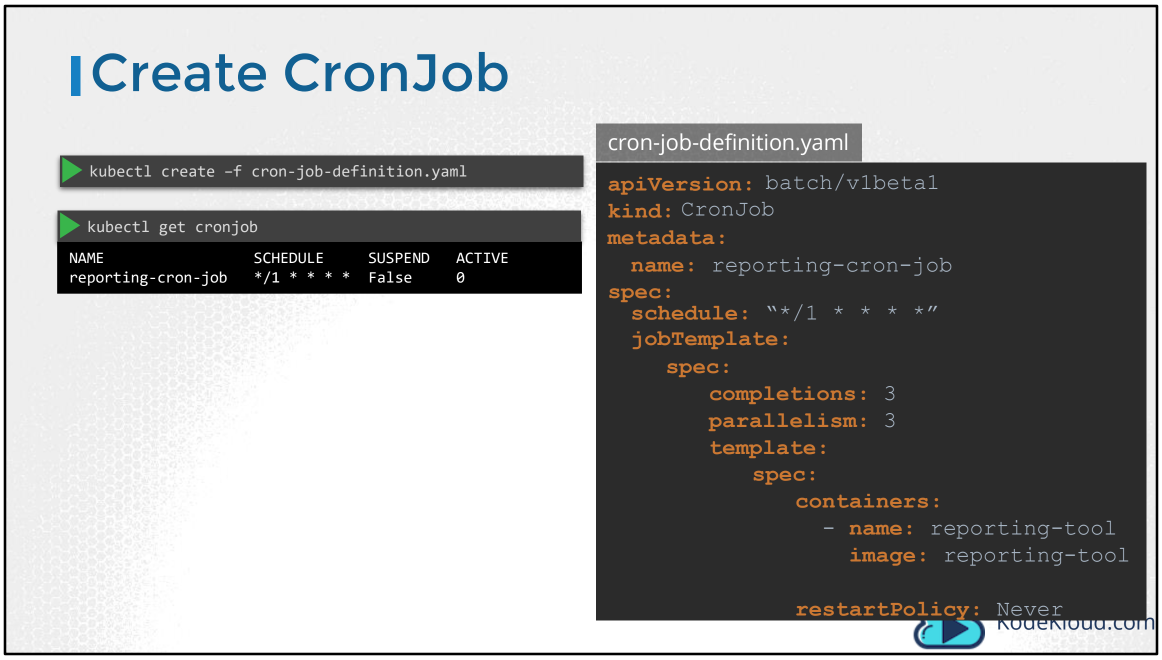
```
apiVersion: batch/v1beta1
```

```
# ┌───────────── minute (0 - 59)
# │ ┌───────────── hour (0 - 23)
# │ │ ┌───────────── day of the month (1 - 31)
# │ │ │ ┌───────────── month (1 - 12)
# │ │ │ │ ┌───────────── day of the week (0 - 6) (Sunday to Saturday;
# │ │ │ │ │                                   7 is also Sunday on some systems)
# │ │ │ │ │
# │ │ │ │ │
# * * * * * command to execute
```

Wikipedia

KodeKloud.com

A cronjob is a job that can be scheduled. Just like cron tab in Linux, if you are familiar with it. Say for example you have a job that generates a report and sends an email. You can create the job using the kubectl create command, but it runs instantly. Instead you could create a cronjob to schedule and run it periodically. To create a cronjob we start with a blank template. The apiVersion as of today is batch/v1beta1. The kind is CronJob with a capital C and J. I will name it reporting-cron-job. Under spec you specify a schedule. The schedule option takes a cron like format string where you can specify the time when the job is to be run. Then you have the Job Template, which is the actual job that should be run. Move all of the content from the spec section of the job definition under this. Notice that the cron job definition now gets a little complex. So you must be extra careful. There are now 3 spec sections, one for the cron-job, one for the job and one for the pod.



Create CronJob

```
kubectl create –f cron-job-definition.yaml
```

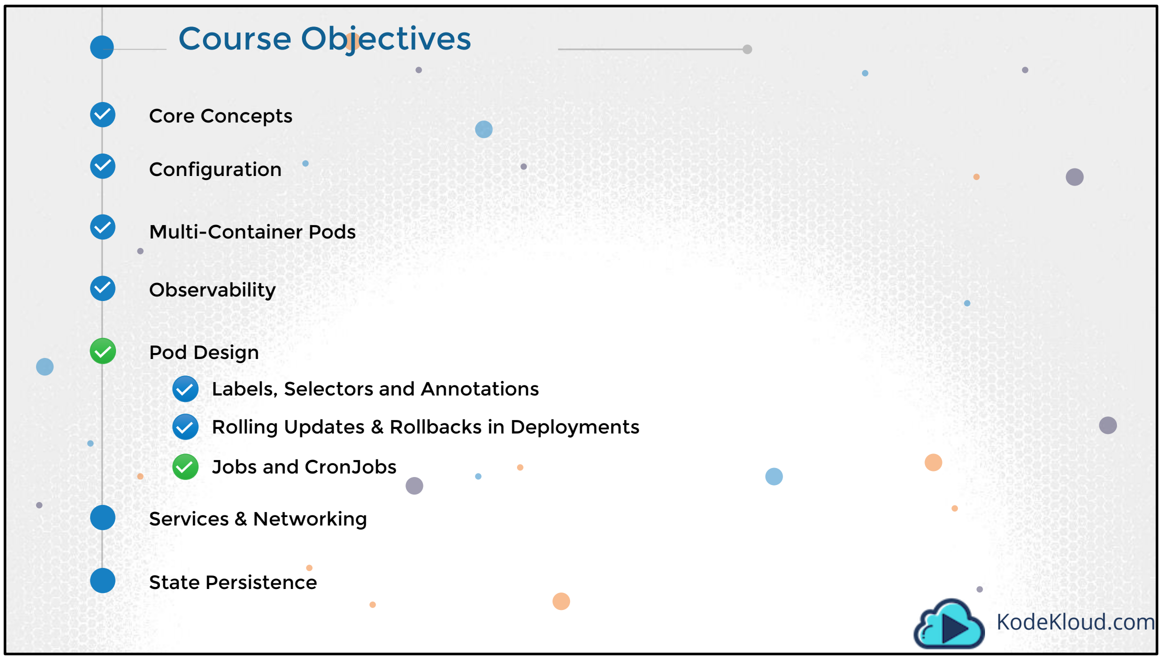
```
kubectl get cronjob
NAME                SCHEDULE     SUSPEND   ACTIVE
reporting-cron-job  */1 * * * *  False     0
```

cron-job-definition.yaml

```yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: reporting-cron-job
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
     spec:
        completions: 3
        parallelism: 3
        template:
           spec:
              containers:
                - name: reporting-tool
                  image: reporting-tool

              restartPolicy: Never
```

KodeKloud.com

Once the file is ready run the kubectl create command to create the cron-job and run the kubectl get cronjob command to see the newly created job. It would inturn create the required jobs and pods.



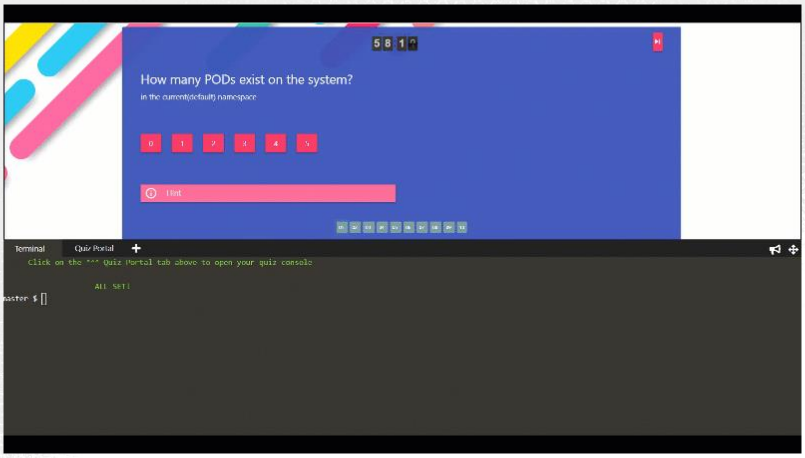
Course Objectives

- ✓ Core Concepts
- ✓ Configuration
- ✓ Multi-Container Pods
- ✓ Observability
- ✓ Pod Design
  - ✓ Labels, Selectors and Annotations
  - ✓ Rolling Updates & Rollbacks in Deployments
  - ✓ Jobs and CronJobs
- Services & Networking
- State Persistence

KodeKloud.com

Well that’s it for this lecture, and I will see you in the next lecture.

Practice Test



5 8 1 ?

How many PODs exist on the system?

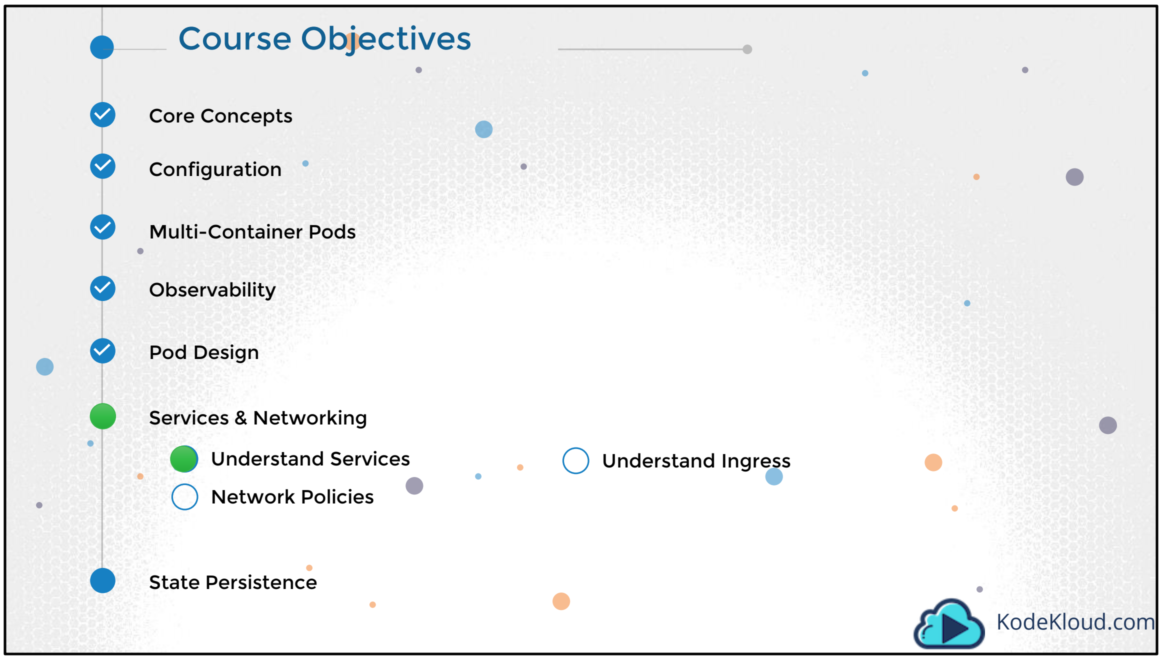
in the current(default) namespace

```
   Click on the ^^^ Quiz Portal tab above to open your quiz console

                 ALL SET!
master $ []
```

Check Link Below!

KodeKloud.com

Access Test Here: https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743697



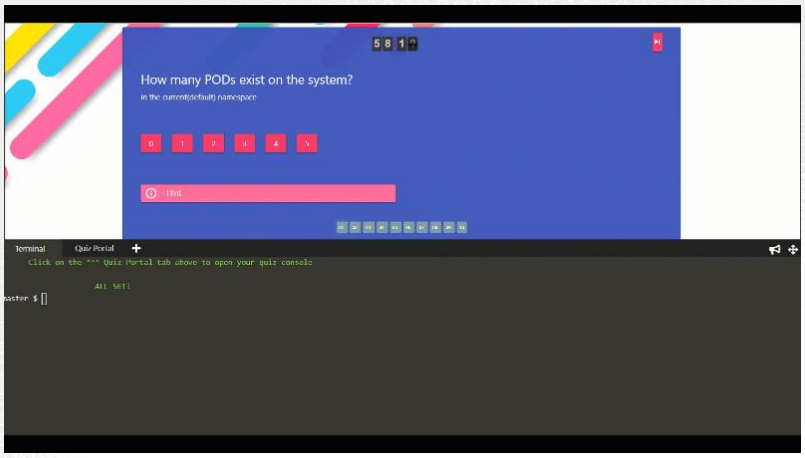
Course Objectives

- ✓ Core Concepts
- ✓ Configuration
- ✓ Multi-Container Pods
- ✓ Observability
- ✓ Pod Design
- ● Services & Networking
  - ● Understand Services
  - ○ Network Policies
  - ○ Understand Ingress
- ● State Persistence

KodeKloud.com

Hello and welcome to this lecture. We are going through the Certified Kubernetes Application Developers course. Services were discussed in the beginners course, so nothing more to add here. Checkout the practice tests.

Practice Test



How many PODs exist on the system?

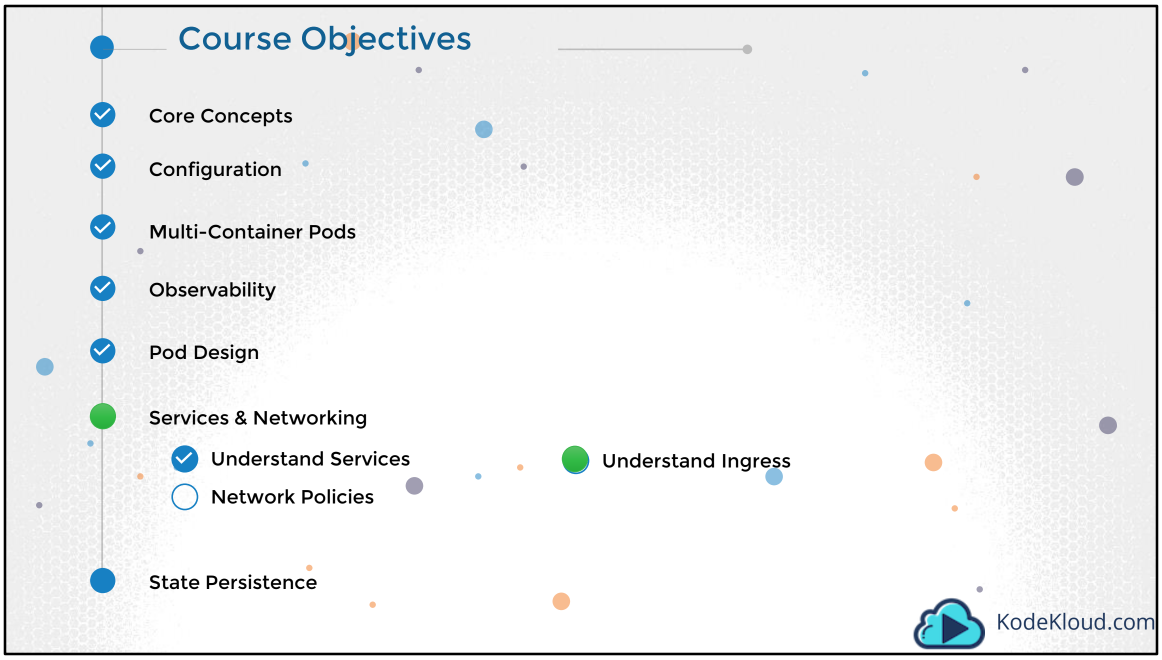
in the current(default) namespace

```
   Click on the ^^^ Quiz Portal tab above to open your quiz console

               ALL SET!
master $ 
```

Check Link Below!

KodeKloud.com

Access Test Here: https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743699



Course Objectives

- ✓ Core Concepts
- ✓ Configuration
- ✓ Multi-Container Pods
- ✓ Observability
- ✓ Pod Design
- ● Services & Networking
  - ✓ Understand Services
  - ○ Network Policies
  - ● Understand Ingress
- ● State Persistence

KodeKloud.com

Hello and welcome to this lecture. We are going through the Certified Kubernetes Application Developers course and in this lecture we will discuss about Ingress in Kubernetes.

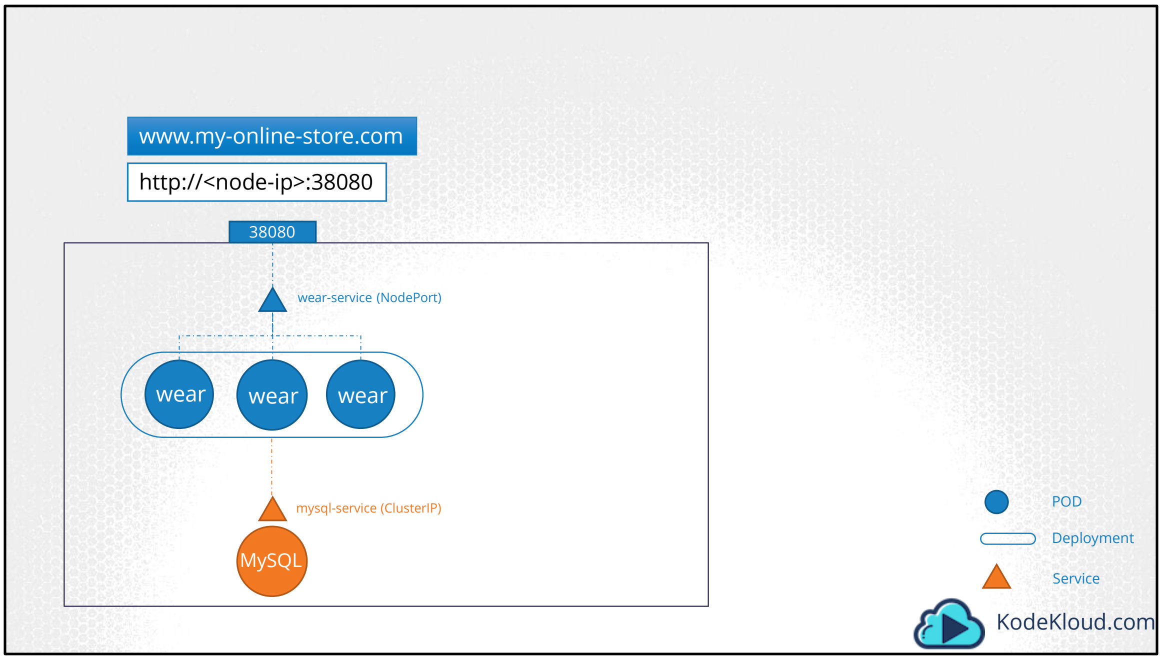
INGRESS

289

KodeKloud.com

So let's get started

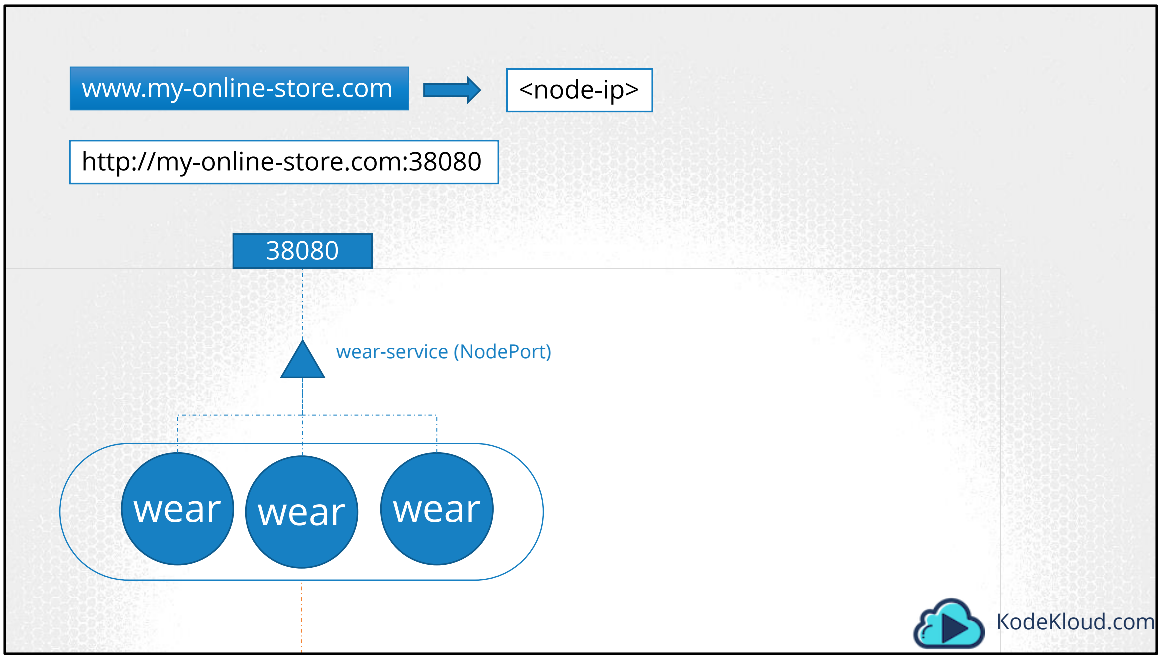
Let us first briefly revisit services and work our way towards ingress. We will start with a simple scenario. You are deploying an application on Kubernetes for a company that has an online store selling products. Your application would be available at say my-online-store.com.



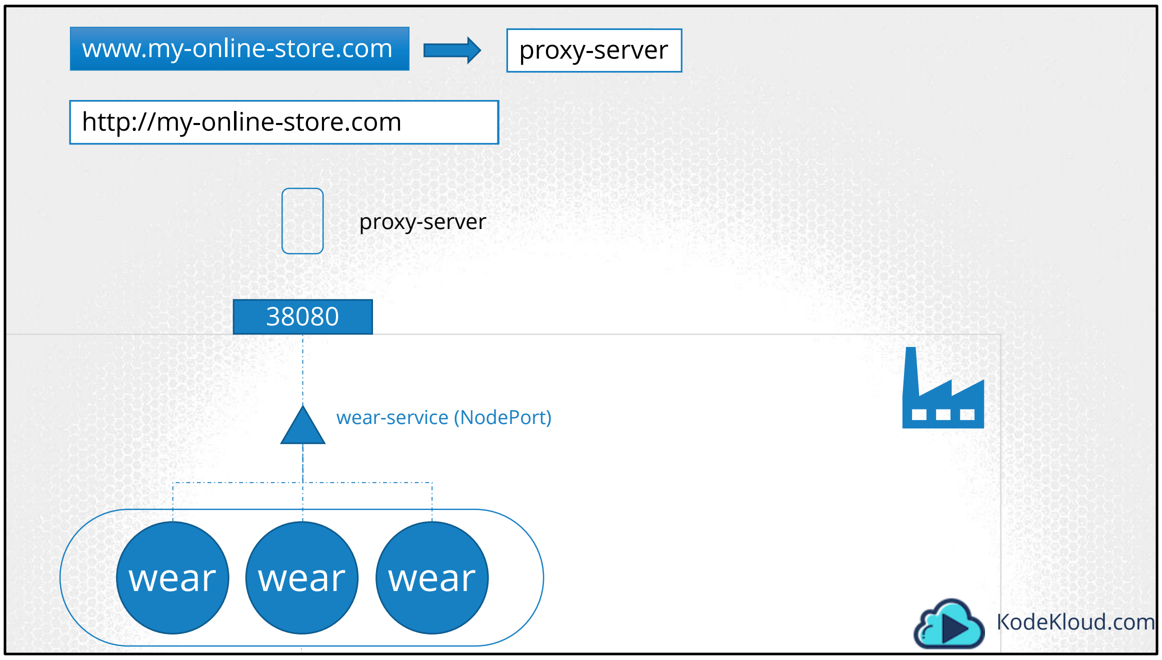
You build the application into a Docker Image and deploy it on the kubernetes cluster as a POD in a Deployment. Your application needs a database so you deploy a MySQL database as a POD and create a service of type ClusterIP called mysql-service to make it accessible to your application. Your application is now working. To make the application accessible to the outside world, you create another service, this time of type NodePort and make your application available on a high-port on the nodes in the cluster. In this example a port 38080 is allocated for the service. The users can now access your application using the URL: http, colon, slash slash IP of any of your nodes followed by port 38080. That setup works and users are able to access the application.

Whenever traffic increases, we increase the number of replicas of the POD to handle the additional traffic, and the service takes care of splitting traffic between the PODs.

<pause>
However, if you have deployed a production grade application before, you know that there are many more things involved in addition to simply splitting the traffic between the PODs.

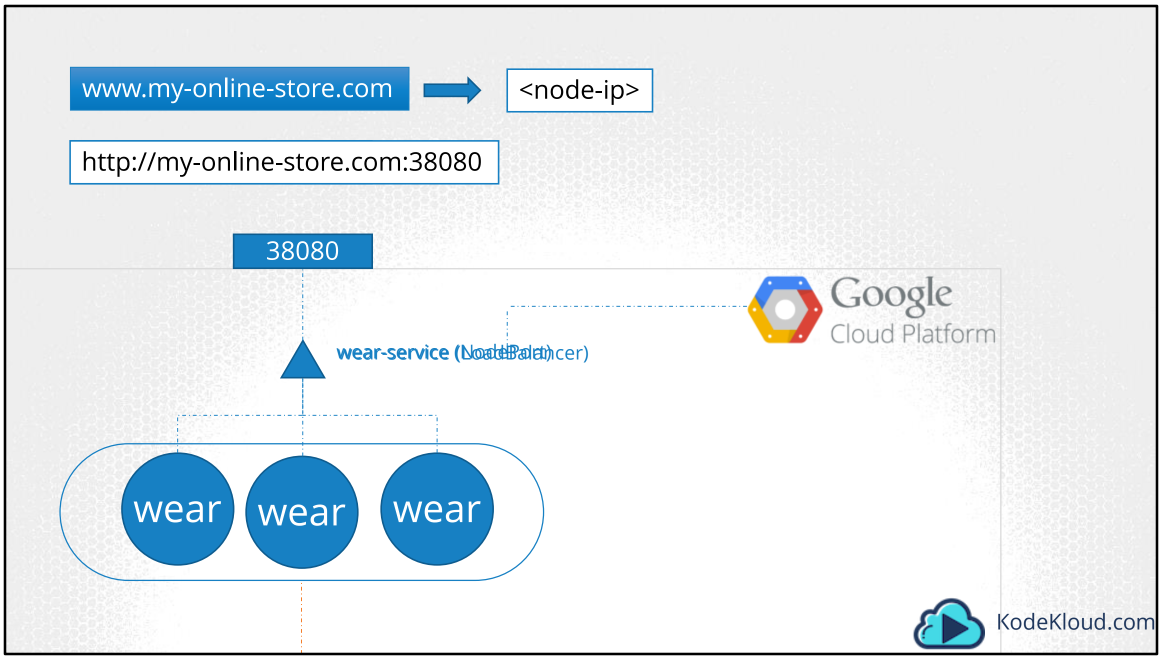


For example, we do not want the users to have to type in IP address everytime. So you configure your DNS server to point to the IP of the nodes. Your users can now access your application using the URL http://my-online-store.com:38080.



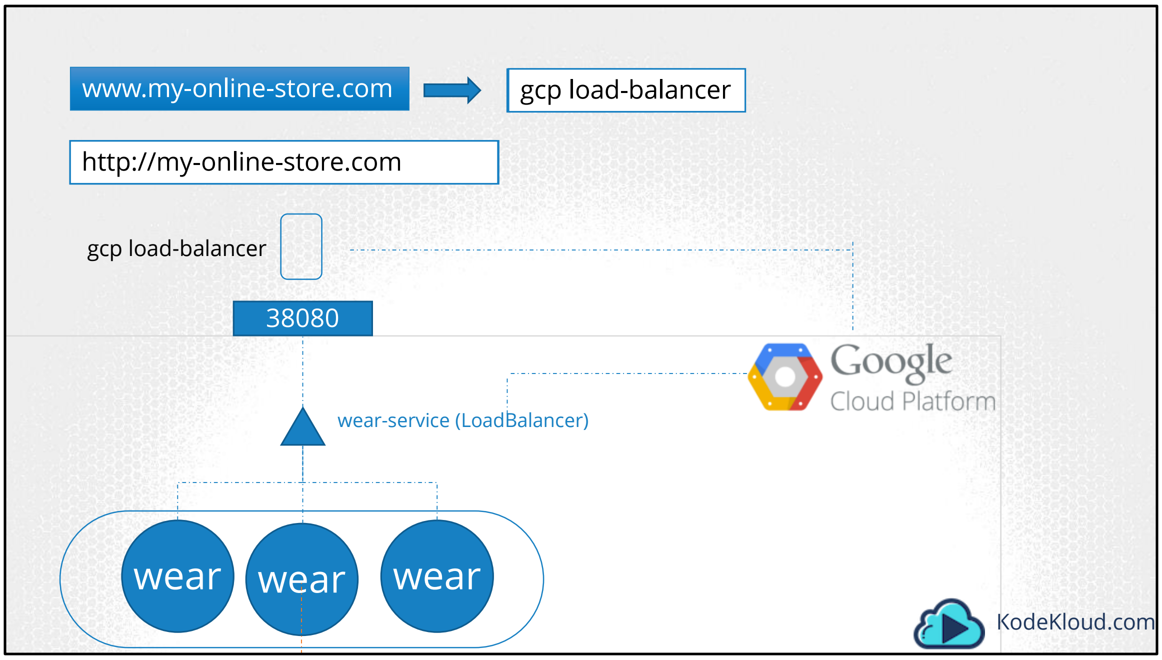
Now, you don’t want your users to have to remember port number either. However, Service NodePorts can only allocate high numbered ports which are greater than 30,000. So you then bring in an additional layer between the DNS server and your cluster, like a proxy server, that proxies requests on port 80 to port 38080 on your nodes. You then point your DNS to this server, and users can now access your application by simply visiting my-online-store.com.

Now, this is if your application is hosted on-prem in your datacenter.



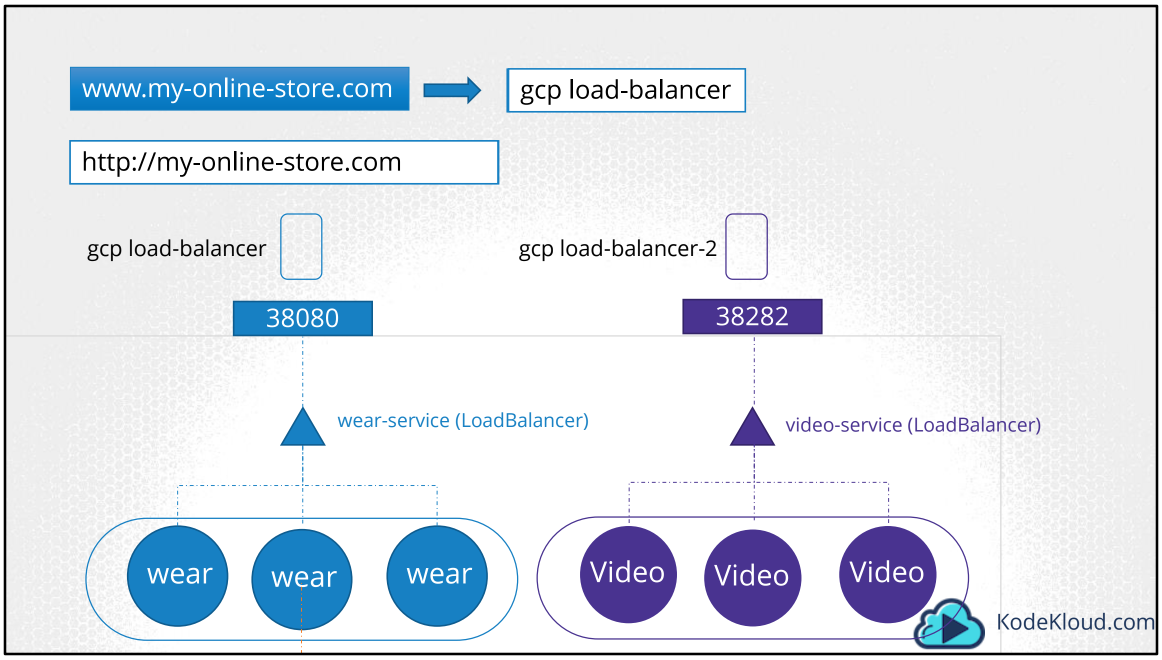
Let’s take a step back and see what you could do it you were on a public cloud environment like Google Cloud Platform.

In that case, instead of creating a service of type NodePort for your wear application, you could set it to type LoadBalancer. When you do that Kubernetes would still do everything that it has to do for a NodePort, which is to provision a high port for the service, but in addition to that kubernetes also sends a request to Google Cloud Platform to provision a native load balancer for this service. GCP would then automatically deploy a LoadBalancer configured to route traffic to the service ports on all the nodes and return its information to kubernetes. The LoadBalancer has an external IP that can be provided to users to access the application. In this case we set the DNS to point to this IP and users access the application using the URL.



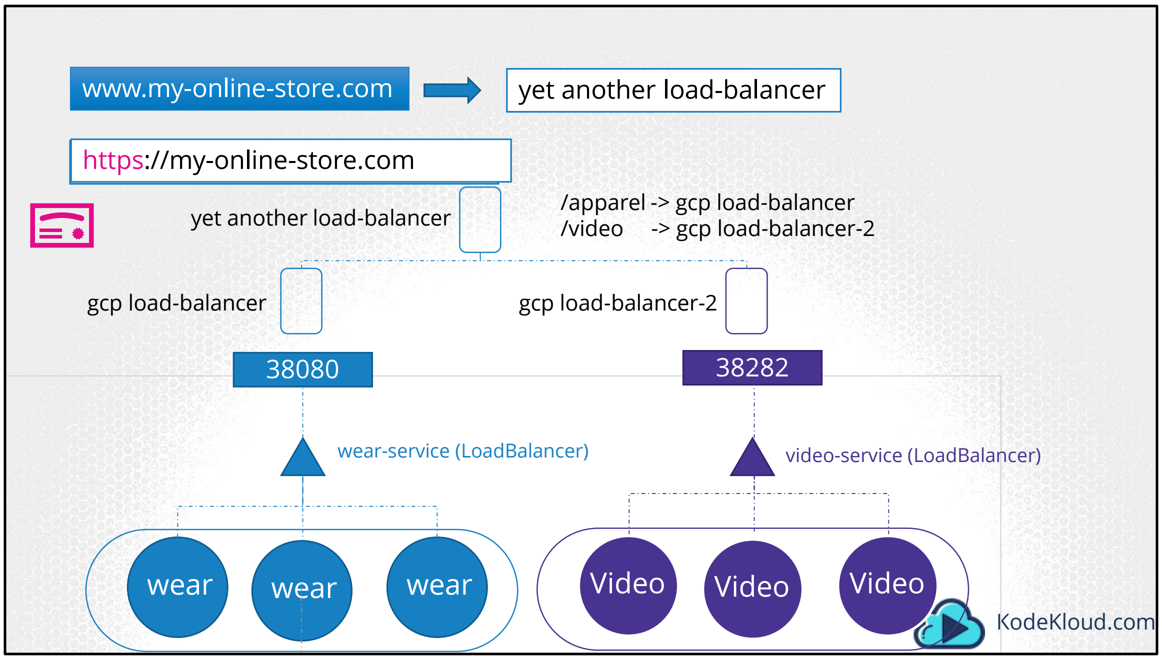
On receiving the request, GCP would then automatically deploy a LoadBalancer configured to route traffic to the service ports on all the nodes and return its information to kubernetes. The LoadBalancer has an external IP that can be provided to users to access the application. In this case we set the DNS to point to this IP and users access the application using the URL my-online-store.com. Perfect!!

Your companies business grows and you now have new services for your customers. For example, a video streaming service. You want your users to be able to access your new video streaming service by going to my-online-store.com/watch. You'd like to make your old application accessible at my-online.store.com / wear.



Your developers developed the new video streaming application as a completely different application, as it has nothing to do with the existing one. However to share the cluster resources, you deploy the new application as a separate deployment within the same cluster. You create a service called video-service of type LoadBalancer. Kubernetes provisions port 38282 for this service and also provisions a cloud native LoadBalancer. The new load balancer has a new IP. Remember you must pay for each of these load balancers and having many such load balancers can inversely affect your cloud bill.

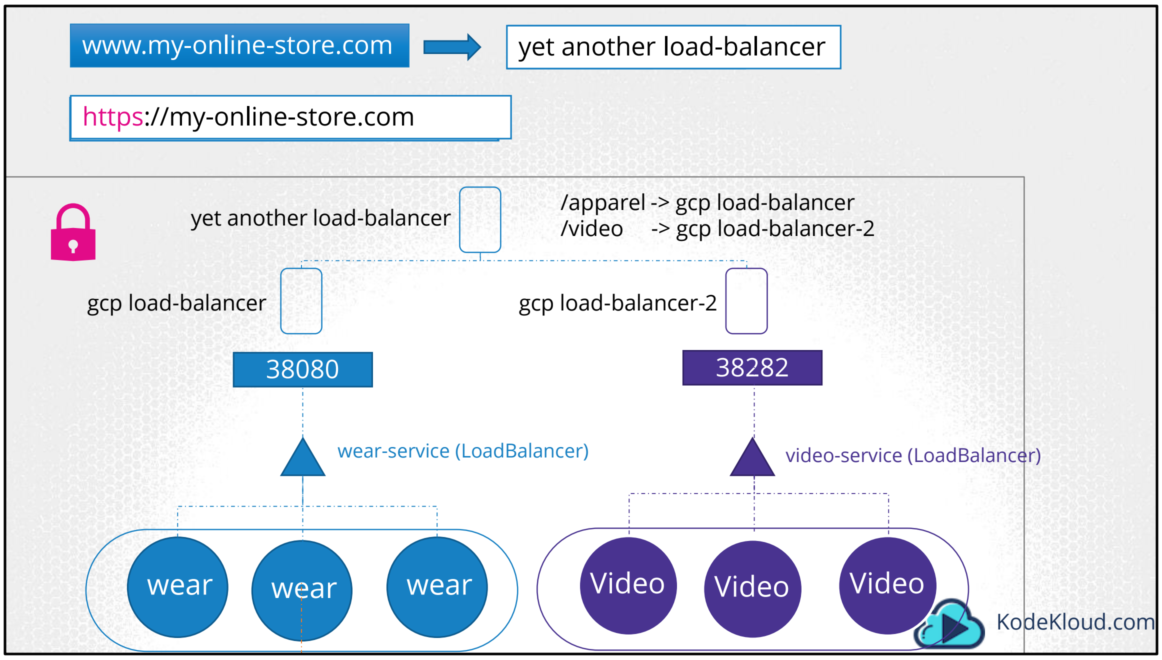
So how do you direct traffic between each of these load balancers based on URL?


www.my-online-store.com
yet another load-balancer
https://my-online-store.com
yet another load-balancer
/apparel -> gcp load-balancer
/video -> gcp load-balancer-2
gcp load-balancer
gcp load-balancer-2
38080
38282
wear-service (LoadBalancer)
video-service (LoadBalancer)
wear
wear
wear
Video
Video
Video
KodeKloud.com

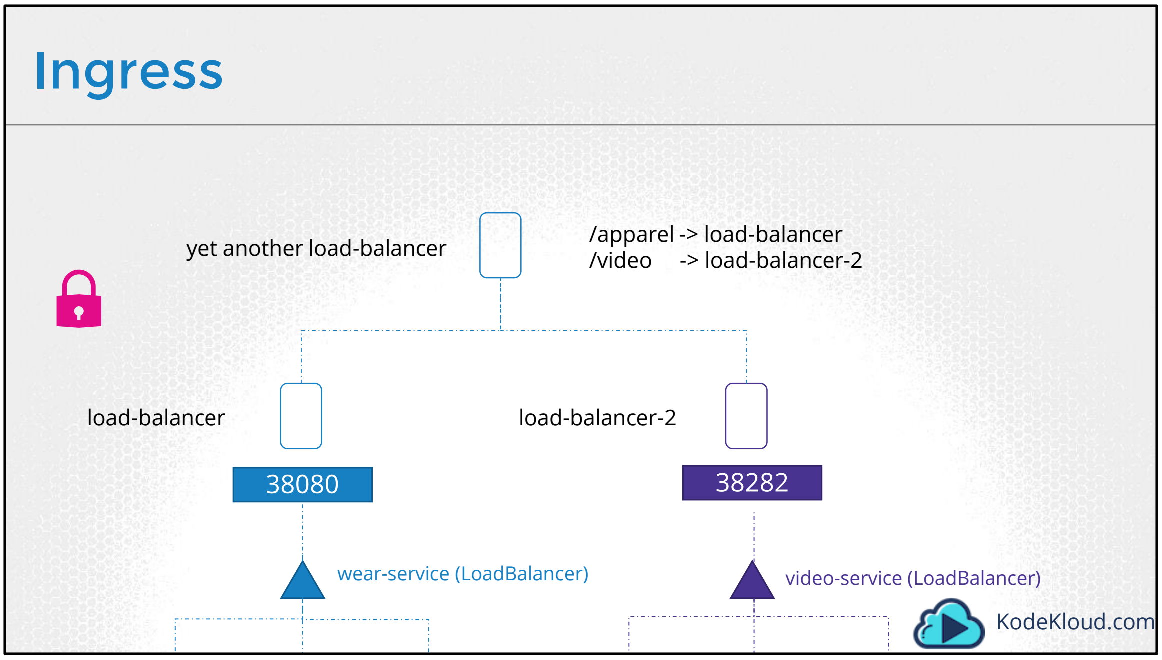
You need yet another proxy or load balancer that can re-direct traffic based on URLs to the different services. Every time you introduce a new service you have to re-configure the load balancer.

And finally you also need to enable SSL for your applications, so your users can access your application using https. Where do you configure that?

Now that's a lot of different configuration and all of these becomes difficult to manage when your application scales. It requires involving different individuals in different teams. You need to configure your firewall rules for each new service. And its expensive as well, as for each service a new cloud native load balancer will be provisioned.


www.my-online-store.com
yet another load-balancer
https://my-online-store.com
yet another load-balancer
/apparel -> gcp load-balancer
/video -> gcp load-balancer-2
gcp load-balancer
gcp load-balancer-2
38080
38282
wear-service (LoadBalancer)
video-service (LoadBalancer)
wear
wear
wear
Video
Video
Video
KodeKloud.com

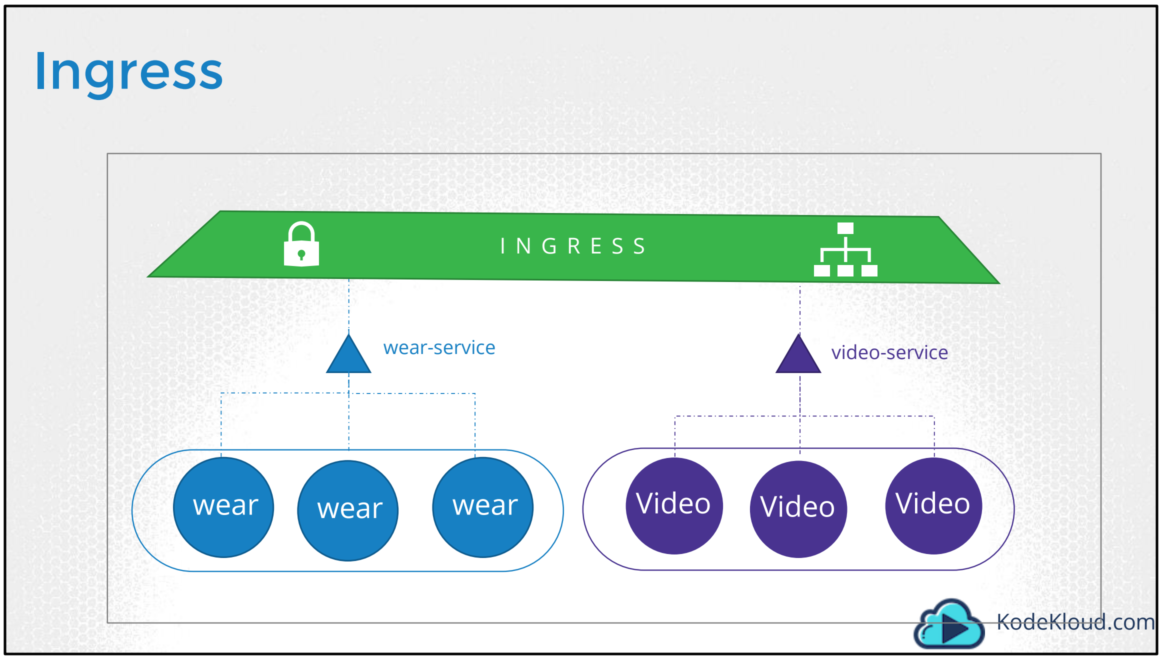
Wouldn’t it be nice if you could manage all of that within the Kubernetes cluster, and have all that configuration as just another kubernetes definition file, that lives along with the rest of your application deployment files?



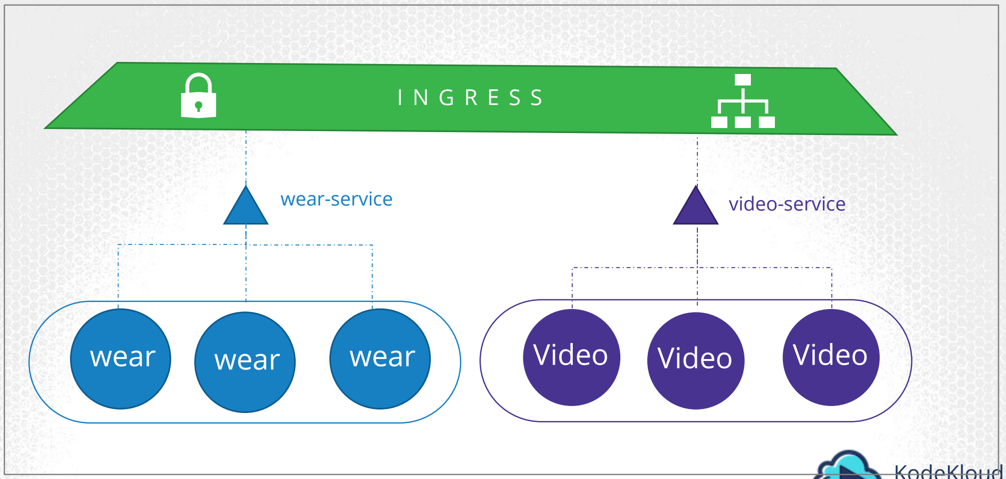
Ingress

yet another load-balancer
/apparel -> load-balancer
/video -> load-balancer-2
load-balancer
load-balancer-2
38080
38282
wear-service (LoadBalancer)
video-service (LoadBalancer)
KodeKloud.com

That’s where Ingress comes into play. Ingress helps your users access your application using a single Externally accessible URL, that you can configure to route to different services within your cluster based on the URL path, at the same time terminate TLS.

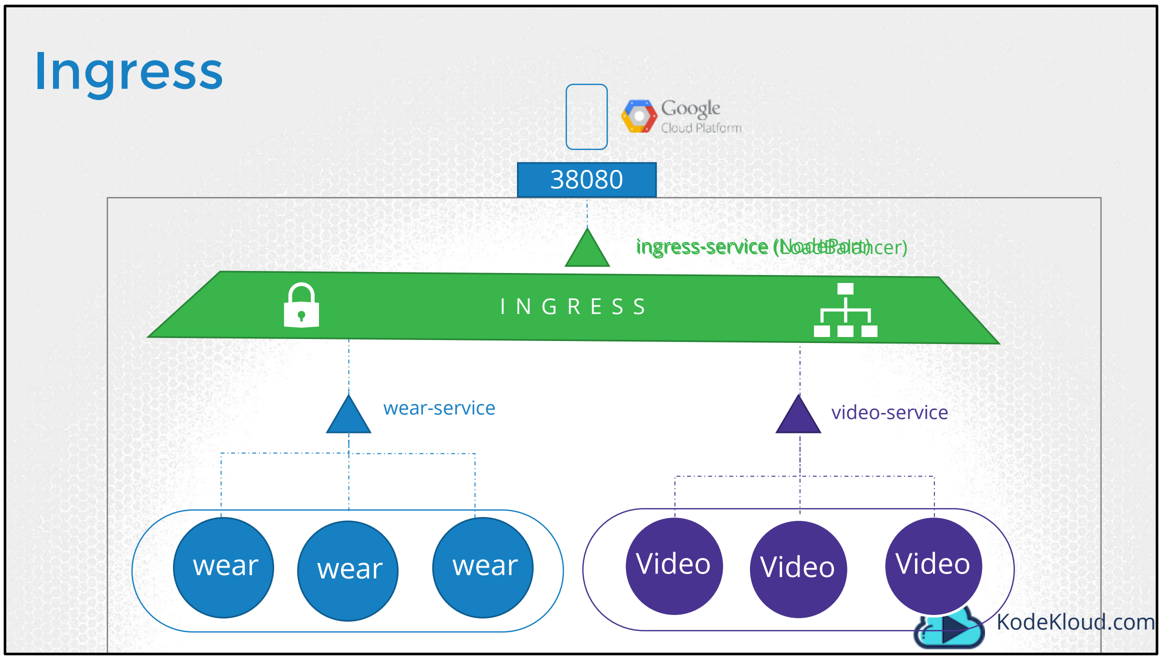


Ingress

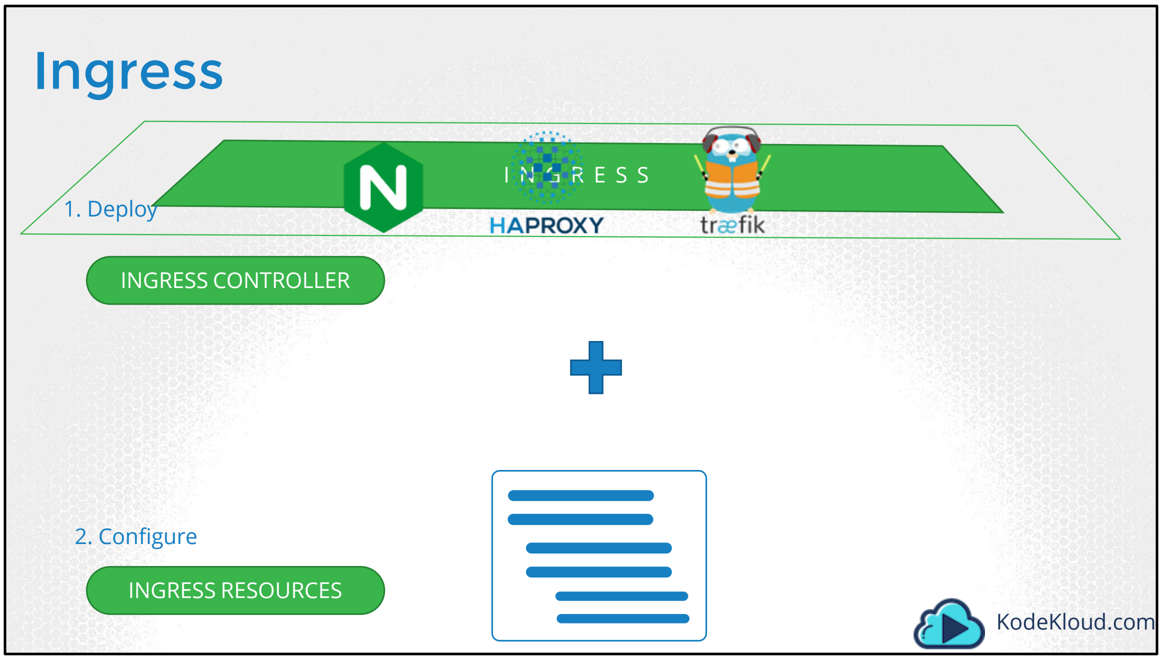


Simply put, think of ingress as a layer 7 load balancer built-in to the kubernetes cluster that can be configured using native kubernetes primitives just like any other object in kubernetes.

Ingress



Now remember, even with Ingress you still need to expose it to make it accessible outside the cluster. So you still have to either publish it as a NodePort or with a Cloud Native LoadBalancer. But that is just a one time thing. Going forward you are going to perform all your load balancing, Auth, SSL and URL based routing configurations on the Ingress controller.



Ingress

So how does it work? What is it? Where is it? How can you see it? How can you configure it?

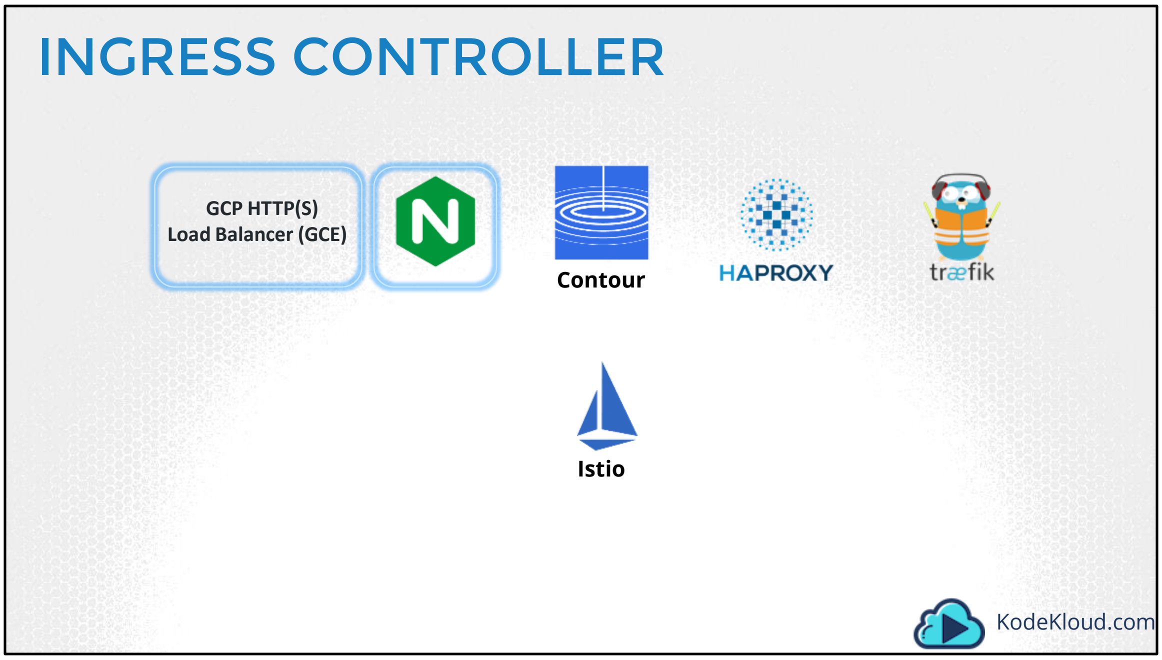
So how does it load balance? How does it implement SSL?

Without ingress, how would YOU do all of these? I would use a reverse-proxy or a load balancing solution like NGINX or HAProxy or Traefik. I would deploy them on my kubernetes cluster and configure them to route traffic to other services. The configuration involves defining URL Routes, SSL certificates etc.

Ingress is implemented by Kubernetes in the same way. You first deploy a supported solution, which happens to be any of these listed here, and then specify a set of rules to configure Ingress. The solution you deploy is called as an Ingress Controller. And the set of rules you configure is called as Ingress Resources. Ingress resources are created using definition files like the ones we used to create PODs, Deployments and services earlier in this course.

Now remember a kubernetes cluster does NOT come with an Ingress Controller by default. If you setup a cluster following the demos in this course, you won't have an ingress controller. So if you simply create ingress resources and expect them to work, they wont.

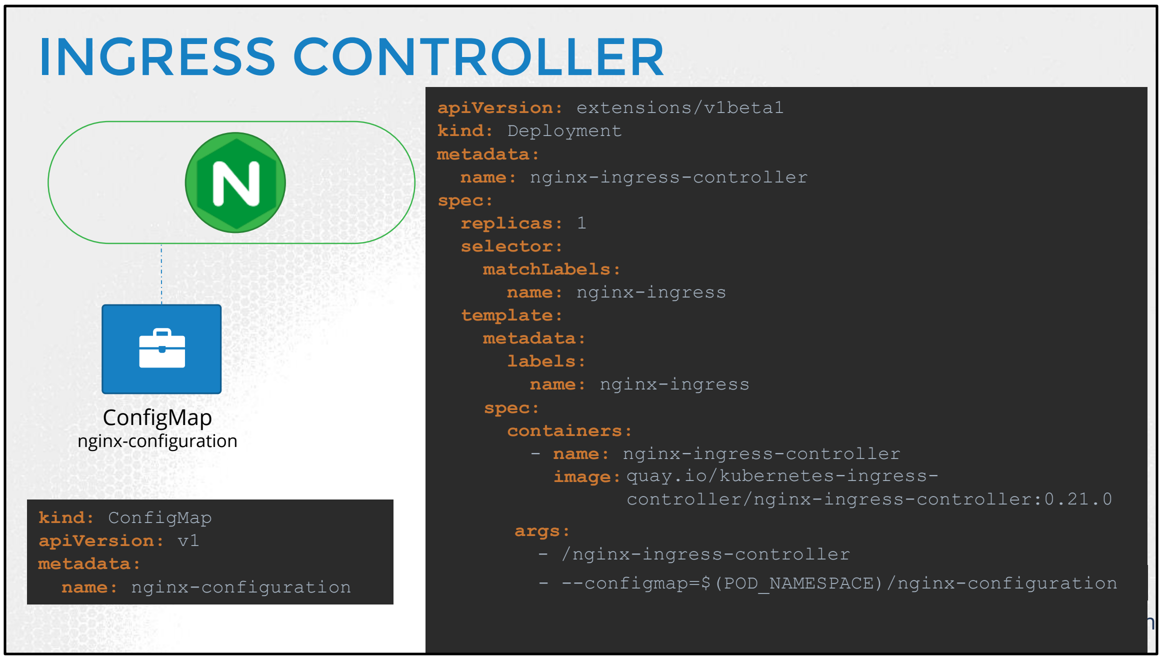
Let’s look at each of these in a bit more detail now.



INGRESS CONTROLLER

GCP HTTP(S)
Load Balancer (GCE)
Contour
HAPROXY
træfik
Istio
KodeKloud.com

Let’s look at each of these in a bit more detail now. As I mentioned you do not have an Ingress Controller on Kubernetes by default. So you MUST deploy one. What do you deploy? There are a number of solutions available for Ingress, a few of them being GCE - which is Googles Layer 7 HTTP Load Balancer. NGINX, Contour, HAPROXY, TRAFIK and Istio. Out of this, GCE and NGINX are currently being supported and maintained by the Kubernetes project. And in this lecture we will use NGINX as an example.

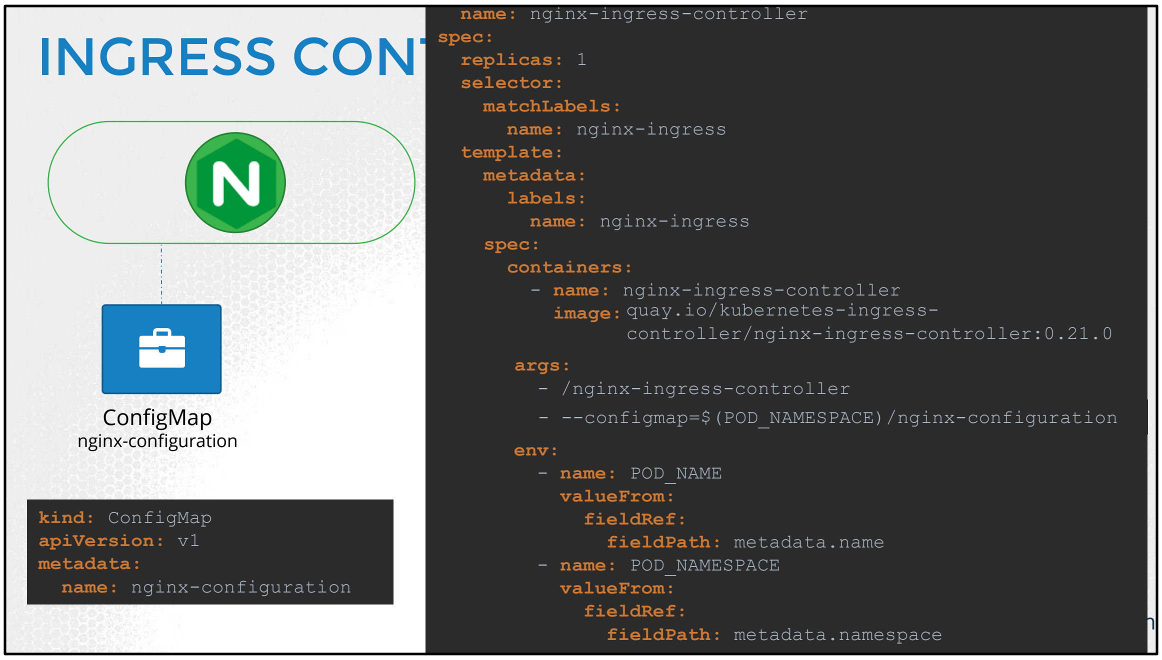


INGRESS CONTROLLER

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: nginx-configuration
```

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-ingress-controller
spec:
  replicas: 1
  selector:
    matchLabels:
      name: nginx-ingress
  template:
    metadata:
      labels:
        name: nginx-ingress
    spec:
      containers:
        - name: nginx-ingress-controller
          image: quay.io/kubernetes-ingress-
                 controller/nginx-ingress-controller:0.21.0

         args:
          - /nginx-ingress-controller
          - --configmap=$(POD_NAMESPACE)/nginx-configuration
```

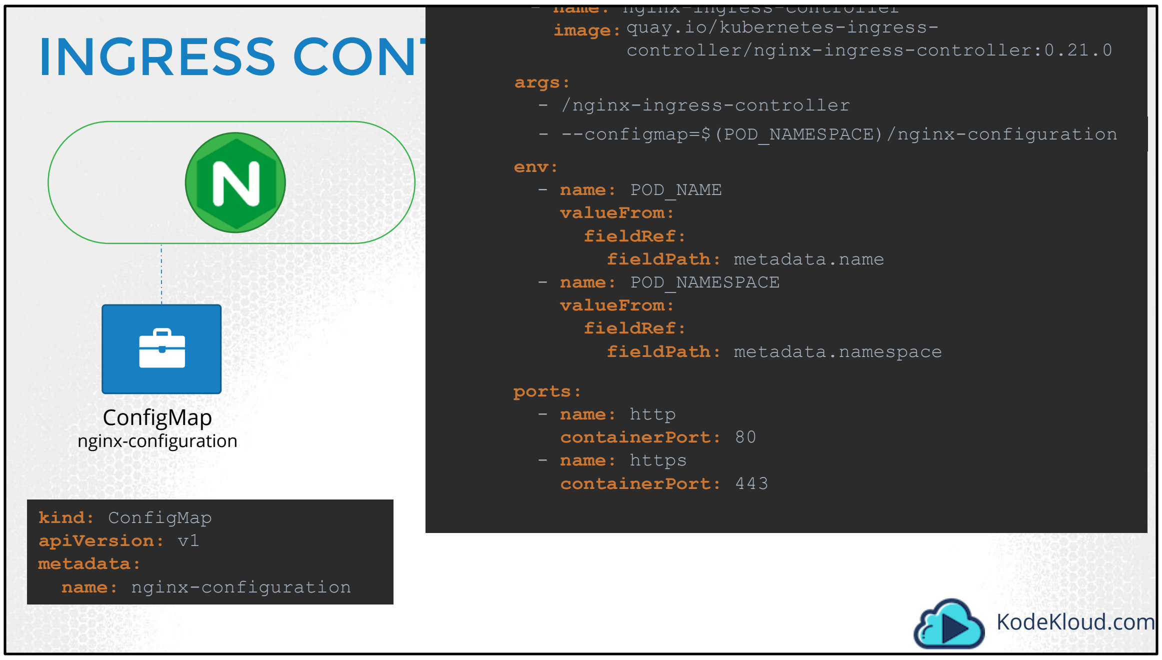
And in this lecture we will use NGINX as an example. An NGINX Controller is deployed as just another deployment in Kubernetes. So we start with a deployment file definition, named nginx-ingress-controller. With 1 replica and a simple pod definition template. We will label it nginx-ingress and the image used is nginx-ingress-controller with the right version. This is a special build of NGINX built specifically to be used as an ingress controller in kubernetes. So it has its own requirements. Within the image the nginx program is stored at location /nginx-ingress-controller. So you must pass that as the command to start the nginx-service. If you have worked with NGINX before, you know that it has a set of configuration options such as the path to store the logs, keep-alive threshold, ssl settings, session timeout etc. In order to decouple these configuration data from the nginx-controller image, you must create a ConfigMap object and pass that in. Now remember the ConfigMap object need not have any entries at this point. A blank object will do. But creating one makes it easy for you to modify a configuration setting in the future. You will just have to add it in to this ConfigMap.



```
kind: ConfigMap
apiVersion: v1
metadata:
  name: nginx-configuration
```

```
  name: nginx-ingress-controller
spec:
  replicas: 1
  selector:
    matchLabels:
      name: nginx-ingress
  template:
    metadata:
      labels:
        name: nginx-ingress
    spec:
      containers:
        - name: nginx-ingress-controller
          image: quay.io/kubernetes-ingress-
                 controller/nginx-ingress-controller:0.21.0

        args:
          - /nginx-ingress-controller
          - --configmap=$(POD_NAMESPACE)/nginx-configuration

        env:
          - name: POD_NAME
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
          - name: POD_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace
```

You must also pass in two environment variables that carry the POD's name and namespace it is deployed to. The nginx service requires these to read the configuration data from within the POD.

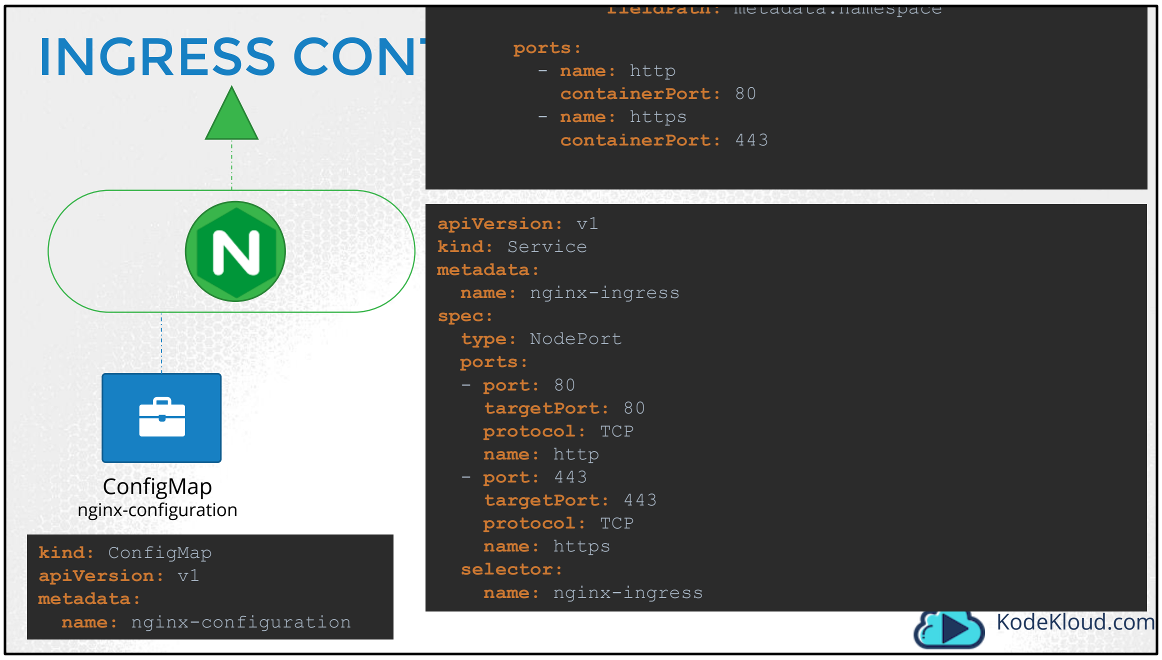


INGRESS CON

```
    image: quay.io/kubernetes-ingress-
              controller/nginx-ingress-controller:0.21.0

  args:
    - /nginx-ingress-controller
    - --configmap=$(POD_NAMESPACE)/nginx-configuration

  env:
    - name: POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    - name: POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace

  ports:
    - name: http
      containerPort: 80
    - name: https
      containerPort: 443
```

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: nginx-configuration
```

And finally specify the ports used by the ingress controller.



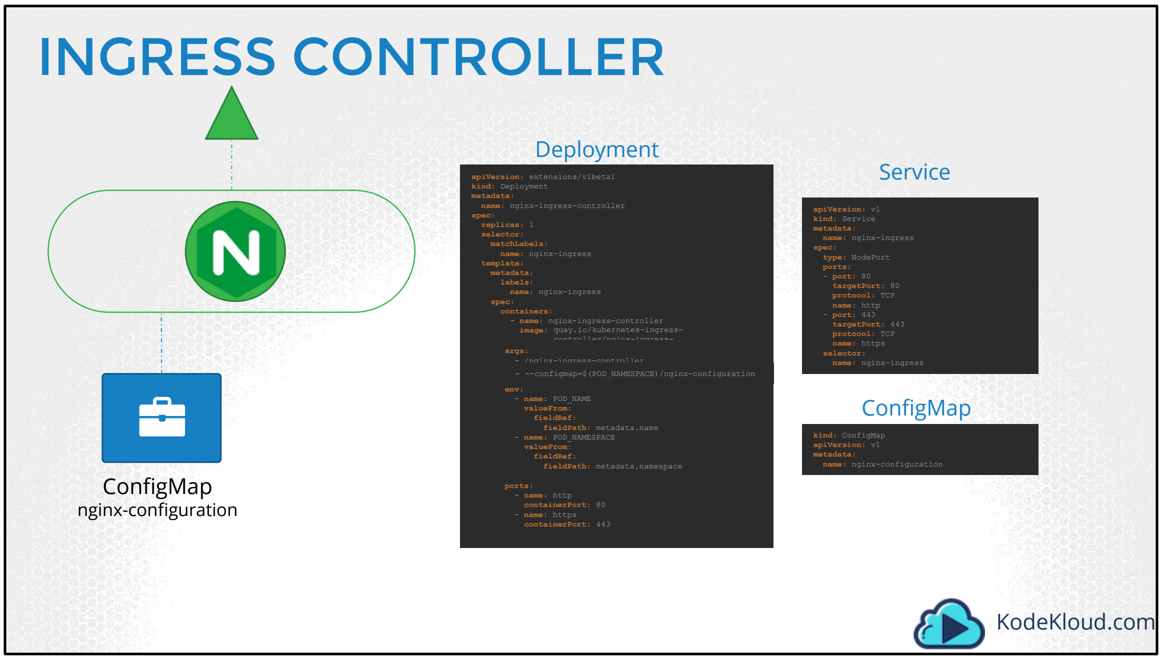
INGRESS CON

```
    fieldPath: metadata.namespace

        ports:
          - name: http
            containerPort: 80
          - name: https
            containerPort: 443
```

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    targetPort: 443
    protocol: TCP
    name: https
  selector:
    name: nginx-ingress
```

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: nginx-configuration
```

We then need a service to expose the ingress controller to the external world. So we create a service of type NodePort with the nginx-ingress label selector to link the service to the deployment. So with these three objects we should be ready with an ingress controller in its simplest form.



INGRESS CONTROLLER

Deployment

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-ingress-controller
spec:
  replicas: 1
  selector:
    matchLabels:
      name: nginx-ingress
  template:
    metadata:
      labels:
        name: nginx-ingress
    spec:
      containers:
        - name: nginx-ingress-controller
          image: quay.io/kubernetes-ingress-
                 controller/nginx-ingress-

       args:
         - /nginx-ingress-controller

         - --configmap=$(POD_NAMESPACE)/nginx-configuration

       env:
         - name: POD_NAME
           valueFrom:
             fieldRef:
               fieldPath: metadata.name
         - name: POD_NAMESPACE
           valueFrom:
             fieldRef:
               fieldPath: metadata.namespace

       ports:
         - name: http
           containerPort: 80
         - name: https
           containerPort: 443
```

Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    targetPort: 443
    protocol: TCP
    name: https
  selector:
    name: nginx-ingress
```

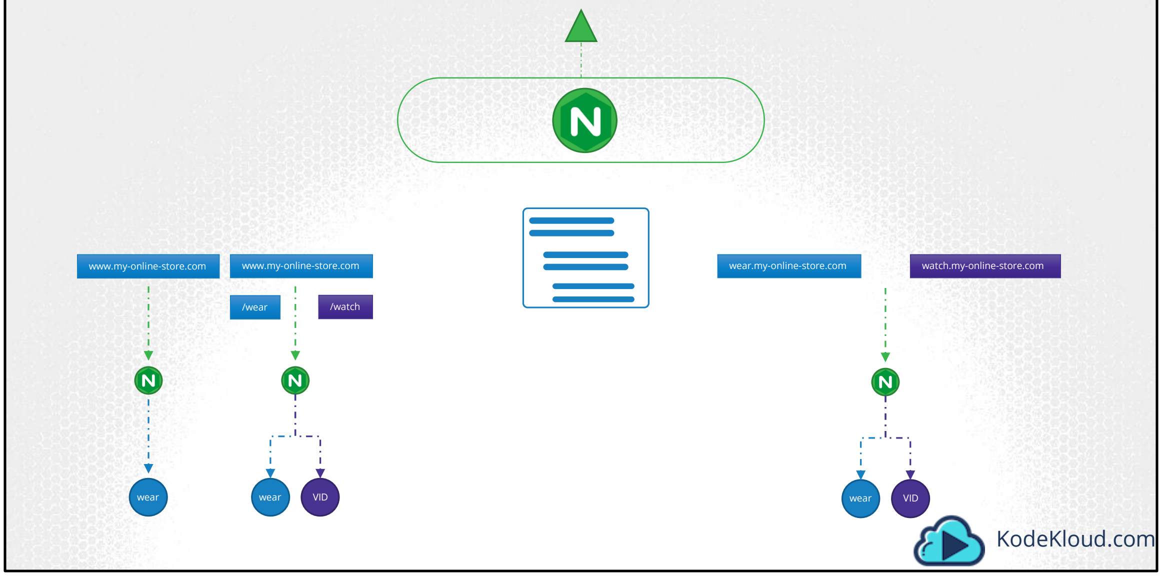
ConfigMap

```yaml
kind: ConfigMap
apiVersion: v1
metadata:
  name: nginx-configuration
```

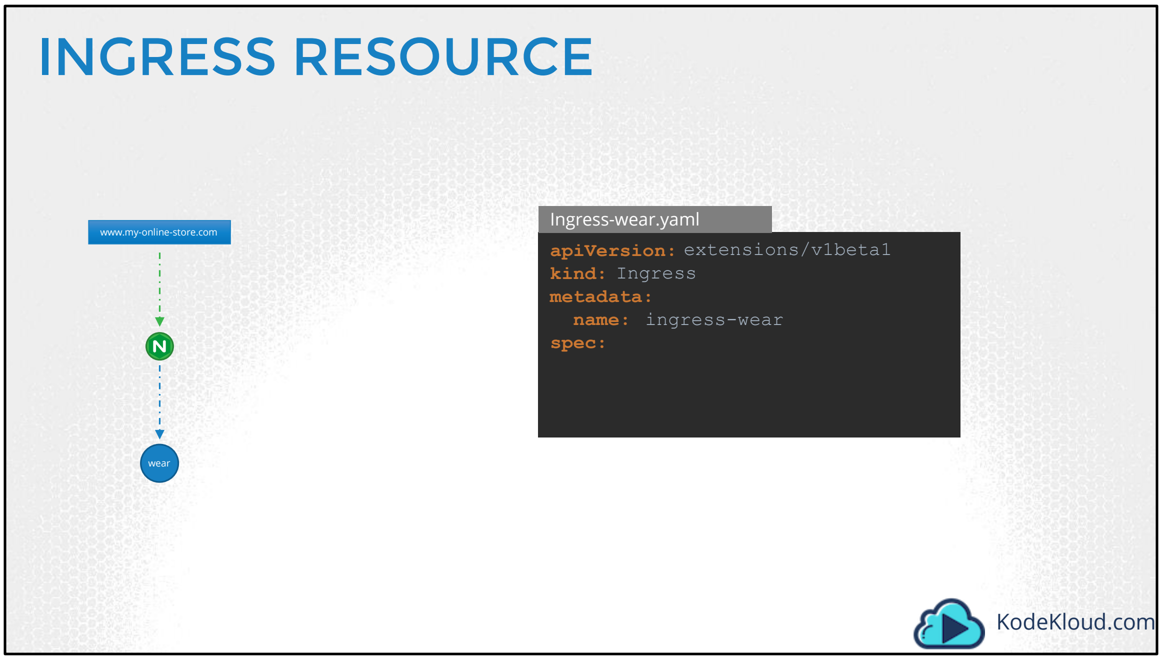
KodeKloud.com

So with these three objects we should be ready with an ingress controller in its simplest form.

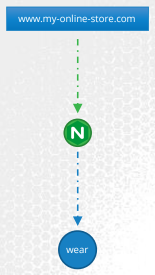
INGRESS RESOURCE



Now on to the next part, of creating Ingress Resources. An Ingress Resource is a set of rules and configurations applied on the ingress controller. You can configure rules to say, simply forward all incoming traffic to a single application, or route traffic to different applications based on the URL. So if user goes to my-online-store.com/wear, then route to one app, or if the user visits the /watch URL then route to the video app. Or you could route user based on the domain name itself.



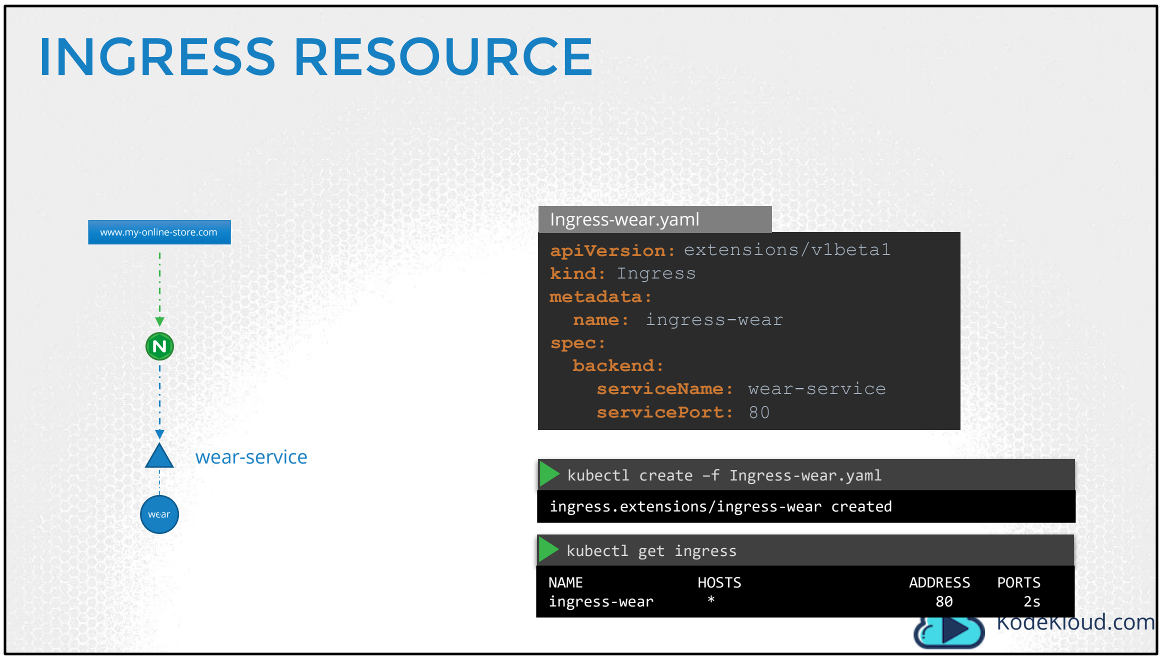
INGRESS RESOURCE



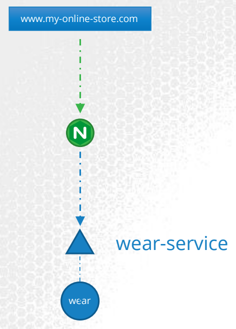
Ingress-wear.yaml

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear
spec:
```

Let us look at how to configure these in a bit more detail. The Ingress Resource is created with a Kubernetes Definition file. In this case, ingress-wear.yaml. As with any other object, we have apiVersion, kind, metadata and spec. The apiVersion is extensions/v1beta1, kind is Ingress, we will name it ingress-wear. And under spec we have backend.



INGRESS RESOURCE



Ingress-wear.yaml

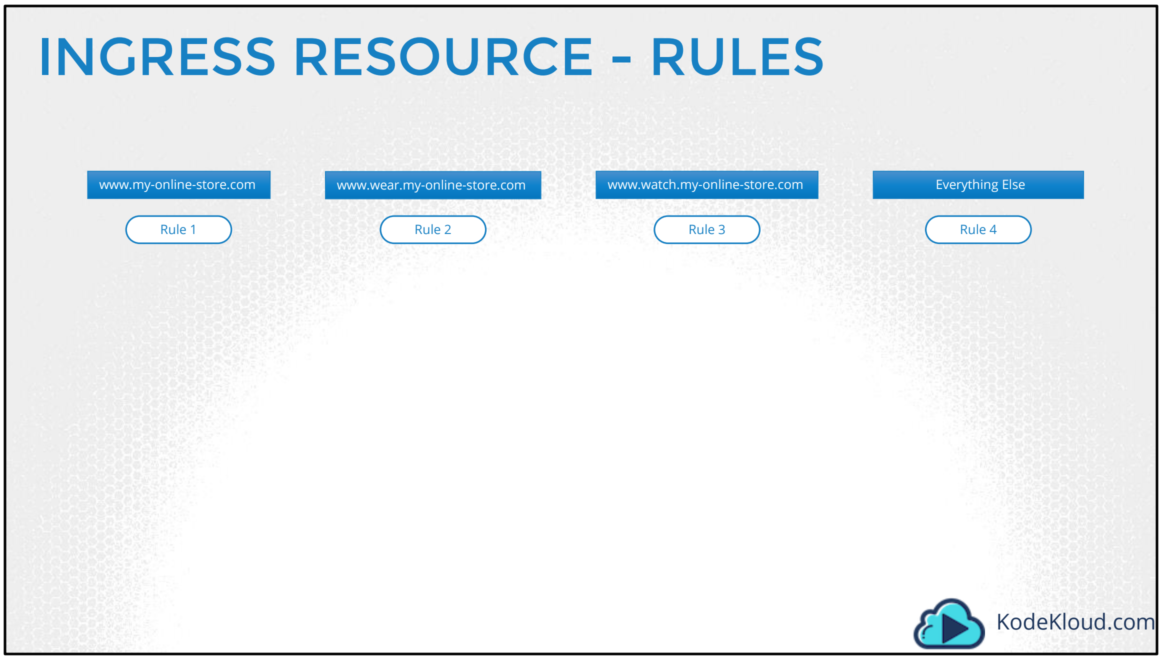
```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear
spec:
  backend:
     serviceName: wear-service
     servicePort: 80
```

```
▶ kubectl create –f Ingress-wear.yaml
ingress.extensions/ingress-wear created
```

```
▶ kubectl get ingress
NAME             HOSTS                    ADDRESS   PORTS
ingress-wear       *                        80        2s
```

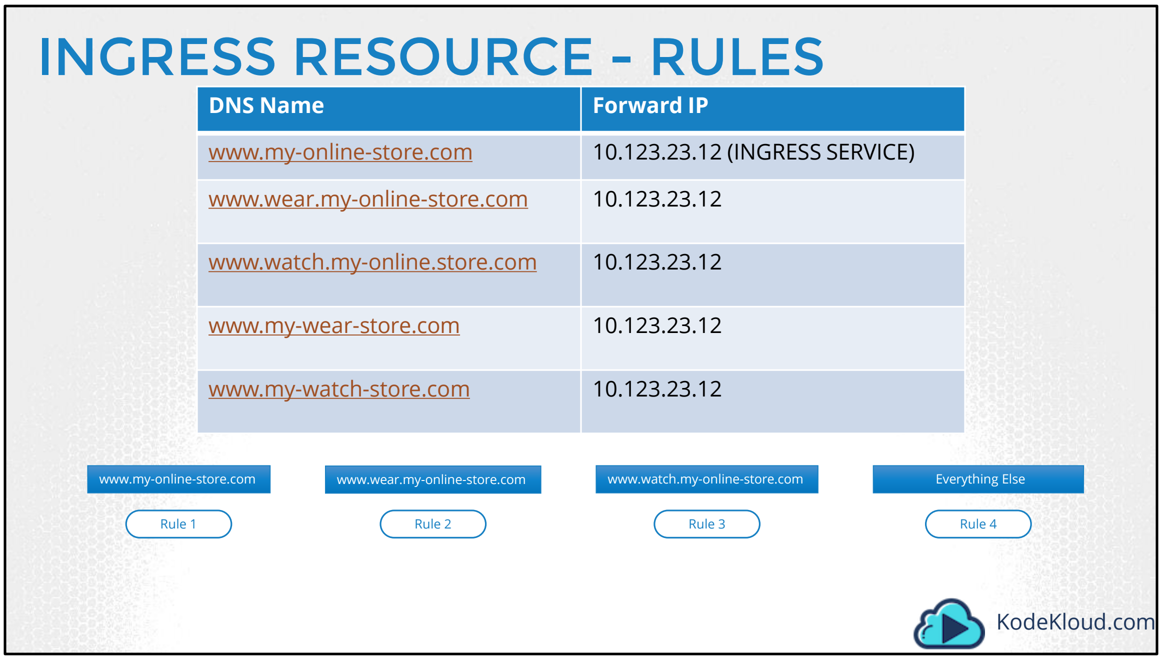
KodeKloud.com

As you might have guessed already, traffic is routed to the application services and not PODs directly. The Backend section defines where the traffic will be routed to. So if it's a single backend, then you don't really have any rules. You can simply specify the service name and port of the backend wear service. Create the ingress resource by running the kubectl create command. View the created ingress by running the kubectl get ingress command. The new ingress is now created and routes all incoming traffic directly to the wear-service.



INGRESS RESOURCE - RULES

You use rules, when you want to route traffic based on different conditions. For example, you create one rule for traffic originating from each domain or hostname. That means when users reach your cluster using the domain name, my-online-store.com, you can handle that traffic using rule1. When users reach your cluster using domain name wear.my-online-store.com, you can handle that traffic using a separate Rule2. Use Rule3 to handle traffic from watch.my-online-store.com. And say use a 4th rule to handle everything else. And you would achieve this, by adding multiple DNS entries, all of which would point to the same Ingress controller service on your kubernetes cluster.

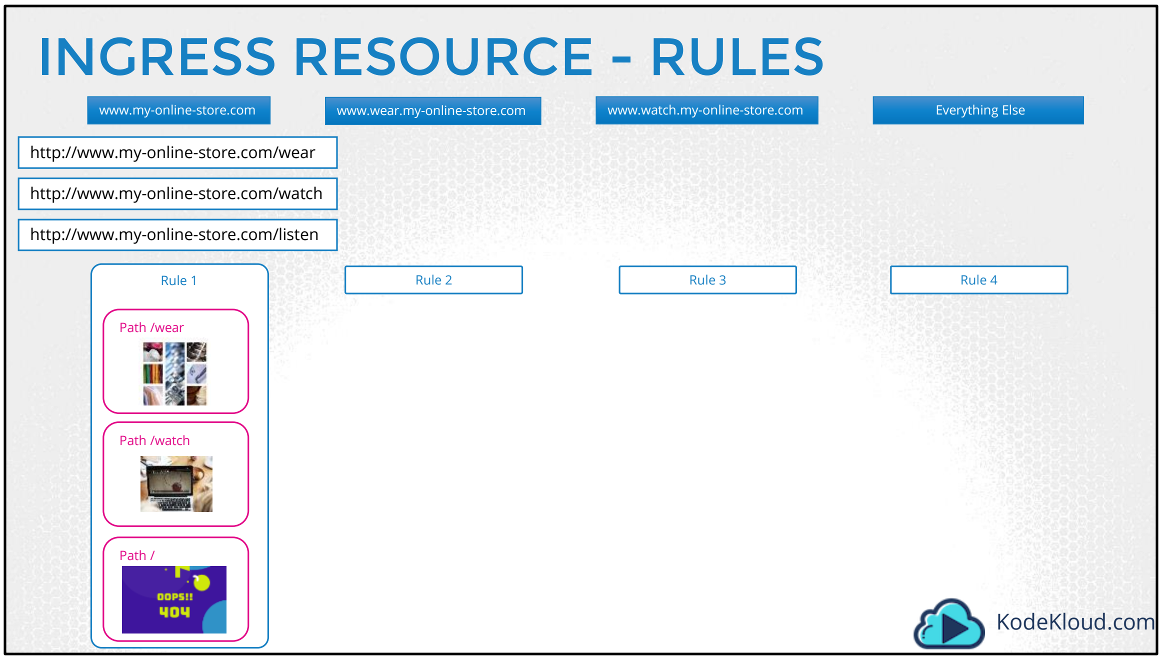


INGRESS RESOURCE - RULES

DNS Name	Forward IP
www.my-online-store.com	10.123.23.12 (INGRESS SERVICE)
www.wear.my-online-store.com	10.123.23.12
www.watch.my-online.store.com	10.123.23.12
www.my-wear-store.com	10.123.23.12
www.my-watch-store.com	10.123.23.12

www.my-online-store.com
Rule 1
www.wear.my-online-store.com
Rule 2
www.watch.my-online-store.com
Rule 3
Everything Else
Rule 4
KodeKloud.com

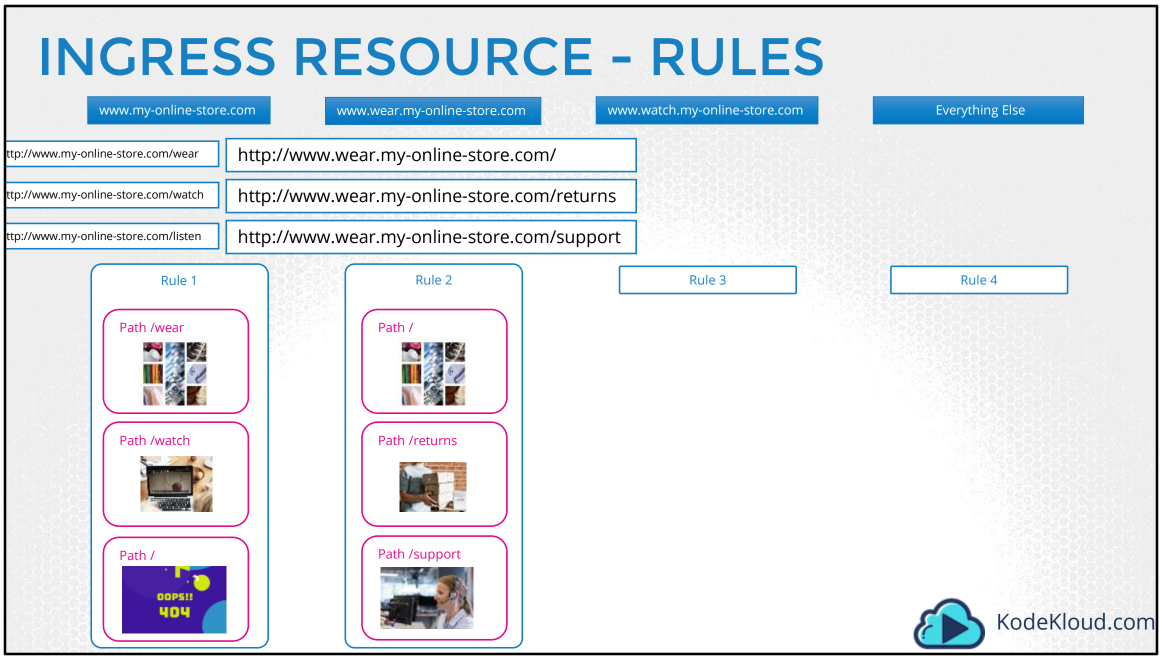
And just in case you didn't know, you would typically achieve this, by adding multiple DNS entries, all pointing to the same Ingress controller service on your kubernetes cluster.



INGRESS RESOURCE - RULES

www.my-online-store.com
www.wear.my-online-store.com
www.watch.my-online-store.com
Everything Else
http://www.my-online-store.com/wear
http://www.my-online-store.com/watch
http://www.my-online-store.com/listen
Rule 1
Rule 2
Rule 3
Rule 4
Path /wear
Path /watch
Path /
OOPS!!
404
KodeKloud.com

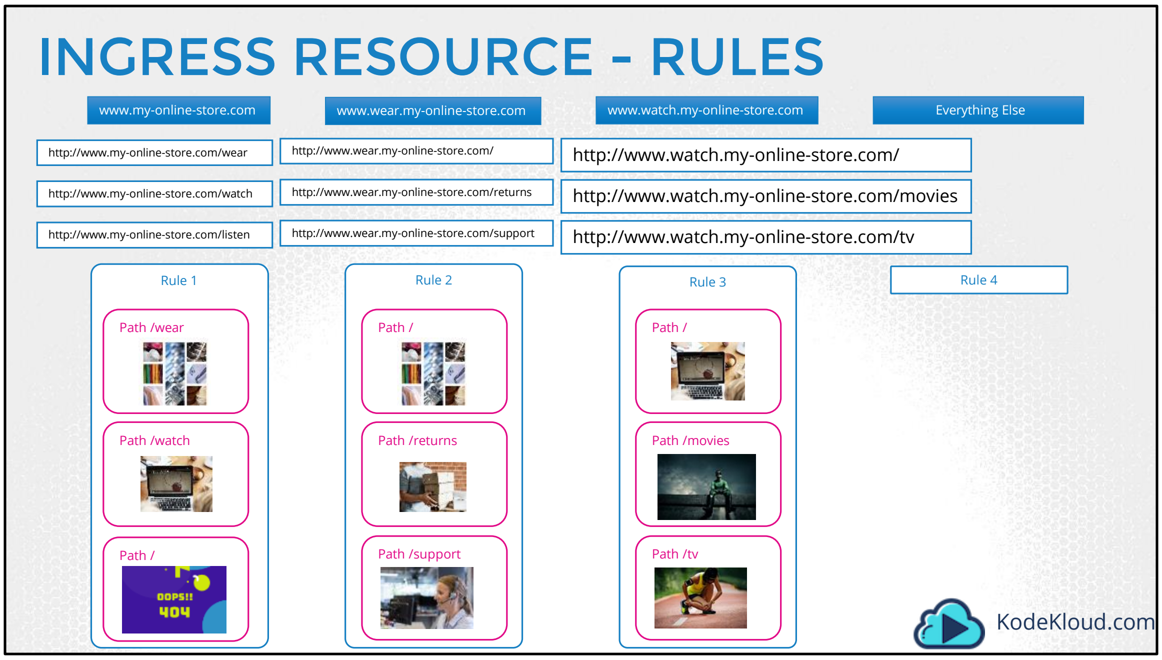
Now within each rule you can handle different paths. For example, within Rule 1 you can handle the wear path to route that traffic to the clothes application. And a watch path to route traffic to the video streaming application. And a third path that routes anything other than the first two to a 404 not found page.



INGRESS RESOURCE - RULES

www.my-online-store.com
www.wear.my-online-store.com
www.watch.my-online-store.com
Everything Else
ttp://www.my-online-store.com/wear
ttp://www.my-online-store.com/watch
ttp://www.my-online-store.com/listen
http://www.wear.my-online-store.com/
http://www.wear.my-online-store.com/returns
http://www.wear.my-online-store.com/support
Rule 1
Path /wear
Path /watch
Path /
OOPS!! 404
Rule 2
Path /
Path /returns
Path /support
Rule 3
Rule 4
KodeKloud.com

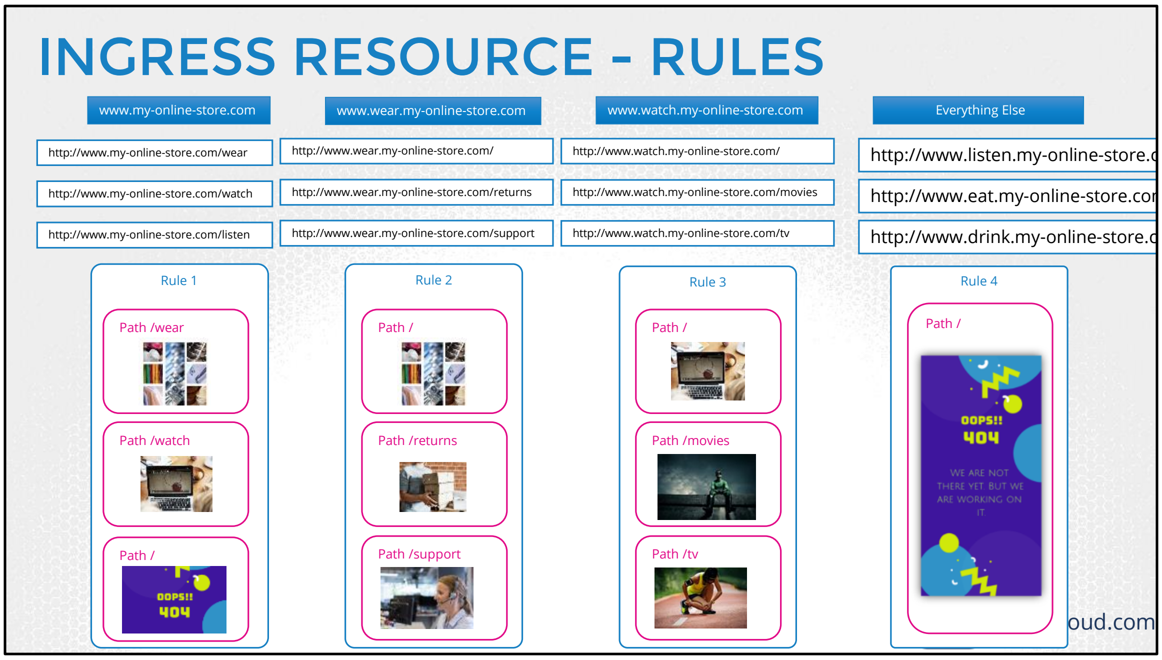
Similarly, the second rule handles all traffic from wear.my-online-store.com. You can have path definition within this rule, to route traffic based on different paths. For example, say you have different applications and services within the apparel section for shopping, or returns, or support, when a user goes to wear.my-online.store.com/, by default they reach the shopping page. But if they go to exchange or support, they reach different backend services.



INGRESS RESOURCE - RULES

www.my-online-store.com
www.wear.my-online-store.com
www.watch.my-online-store.com
Everything Else
http://www.my-online-store.com/wear
http://www.my-online-store.com/watch
http://www.my-online-store.com/listen
http://www.wear.my-online-store.com/
http://www.wear.my-online-store.com/returns
http://www.wear.my-online-store.com/support
http://www.watch.my-online-store.com/
http://www.watch.my-online-store.com/movies
http://www.watch.my-online-store.com/tv
Rule 1
Path /wear
Path /watch
Path /
OOPS!! 404
Rule 2
Path /
Path /returns
Path /support
Rule 3
Path /
Path /movies
Path /tv
Rule 4
KodeKloud.com

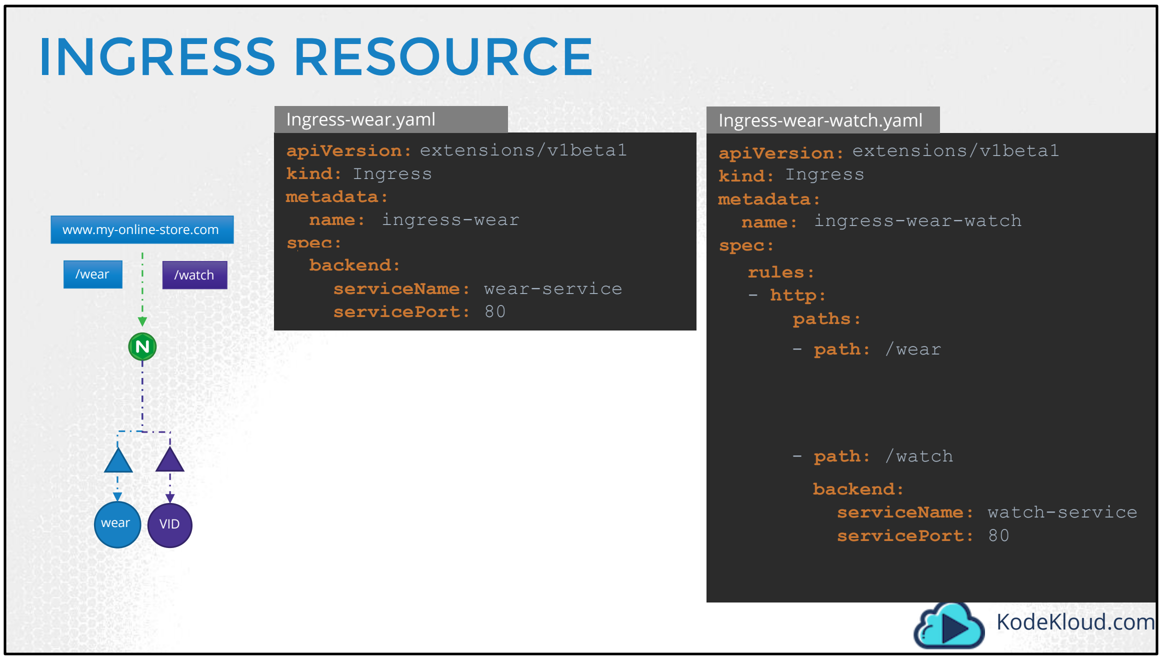
The same goes for Rule 3, where you route traffic to watch.my-online-store.com to the video streaming application. But you can have additional paths in it such as movies or tv.



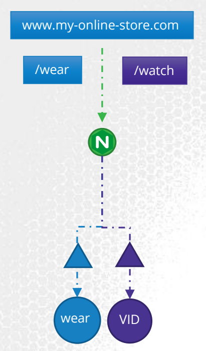
INGRESS RESOURCE - RULES

www.my-online-store.com	www.wear.my-online-store.com	www.watch.my-online-store.com	Everything Else
http://www.my-online-store.com/wear	http://www.wear.my-online-store.com/	http://www.watch.my-online-store.com/	http://www.listen.my-online-store.c
http://www.my-online-store.com/watch	http://www.wear.my-online-store.com/returns	http://www.watch.my-online-store.com/movies	http://www.eat.my-online-store.cor
http://www.my-online-store.com/listen	http://www.wear.my-online-store.com/support	http://www.watch.my-online-store.com/tv	http://www.drink.my-online-store.c

And finally anything other than the ones listed will got to the 4th Rule, that would simply show a 404 Not Found Error page. So remember, you have rules at the top for each host or domain name. And within each rule you have different paths to route traffic based on the URL.



INGRESS RESOURCE



Ingress-wear.yaml

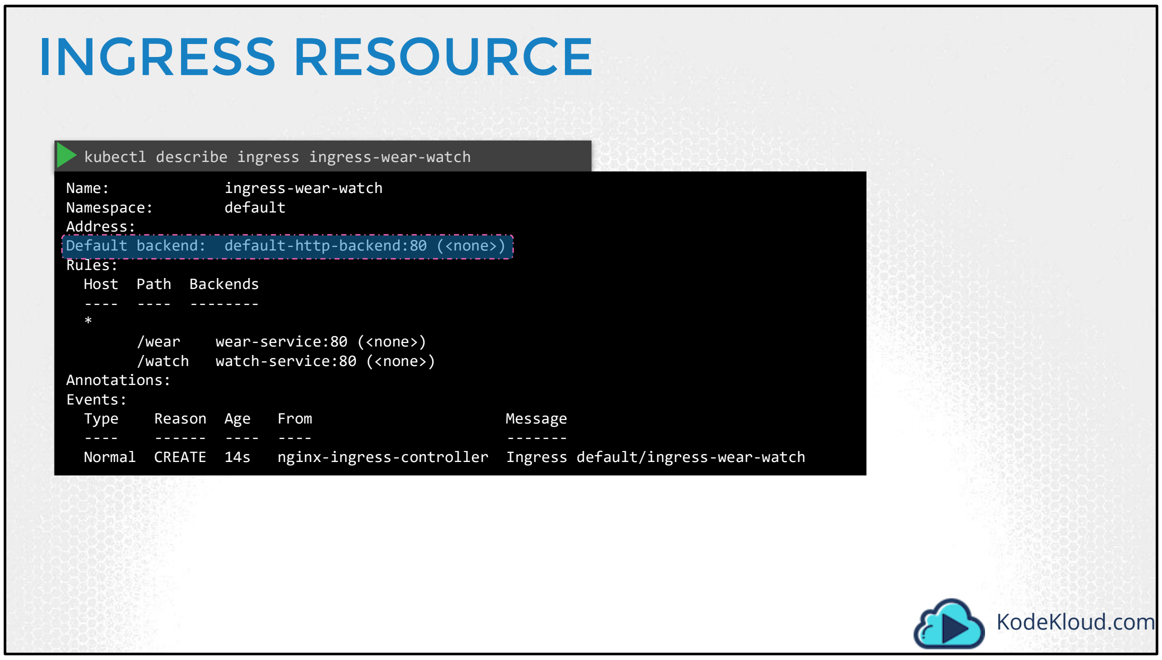
```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear
spec:
  backend:
    serviceName: wear-service
    servicePort: 80
```

Ingress-wear-watch.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:
   rules:
   - http:
       paths:
       - path: /wear



       - path: /watch
         backend:
           serviceName: watch-service
           servicePort: 80
```

KodeKloud.com

Now, let's look at how we configure ingress resources in Kubernetes. We will start where we left off. We start with a similar definition file. This time under spec, we start with a set of rules. Now our requirement here is to handle all traffic coming to my-online-store.com and route them based on the URL path. So we just need a single Rule for this, since we are only handling traffic to a single domain name, which is my-online-store.com in this case. Under rules we have one item, which is an http rule in which we specify different paths. So paths is an array of multiple items. One path for each url. Then we move the backend we used in the first example under the first path. The backend specification remains the same, it has a servicename and serviceport. Similarly we create a similar backend entry to the second URL path, for the watch-service to route all traffic to the /watch url to the watch-service. Create the ingress resource using the kubectl create command.



INGRESS RESOURCE

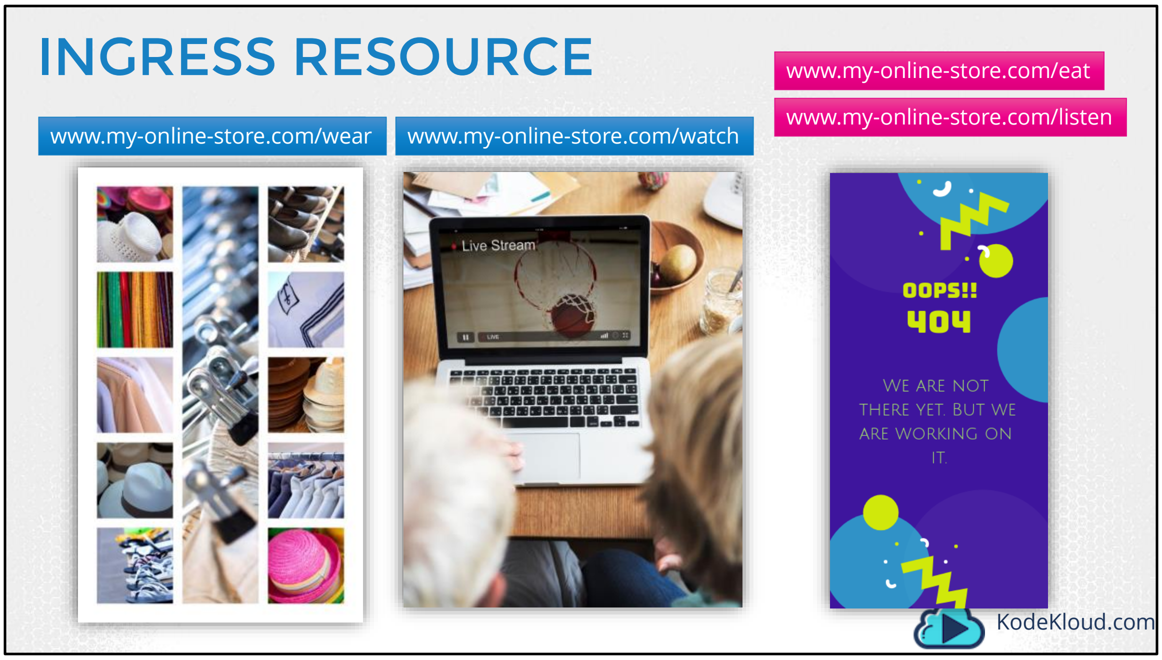
```
kubectl describe ingress ingress-wear-watch
```

```
Name:             ingress-wear-watch
Namespace:        default
Address:
Default backend:  default-http-backend:80 (<none>)
Rules:
  Host  Path  Backends
  ----  ----  --------
  *
        /wear    wear-service:80 (<none>)
        /watch   watch-service:80 (<none>)
Annotations:
Events:
  Type    Reason  Age   From                      Message
  ----    ------  ----  ----                      -------
  Normal  CREATE  14s   nginx-ingress-controller  Ingress default/ingress-wear-watch
```

Once created, view additional details about the ingress by running the kubectl describe ingress command. You now see two backend URLs under the rules, and the backend service they are pointing to. Just as we created it.

Now, If you look closely in the output of this command, you see that there is something about a Default-backend. Hmmm. What might that be?

If a user tries to access a URL that does not match any of these rules, then the user is directed to the service specified as the default backend. In this case it happens to be a service named default-http-backend. So you must remember to deploy such a service.



INGRESS RESOURCE

www.my-online-store.com/eat

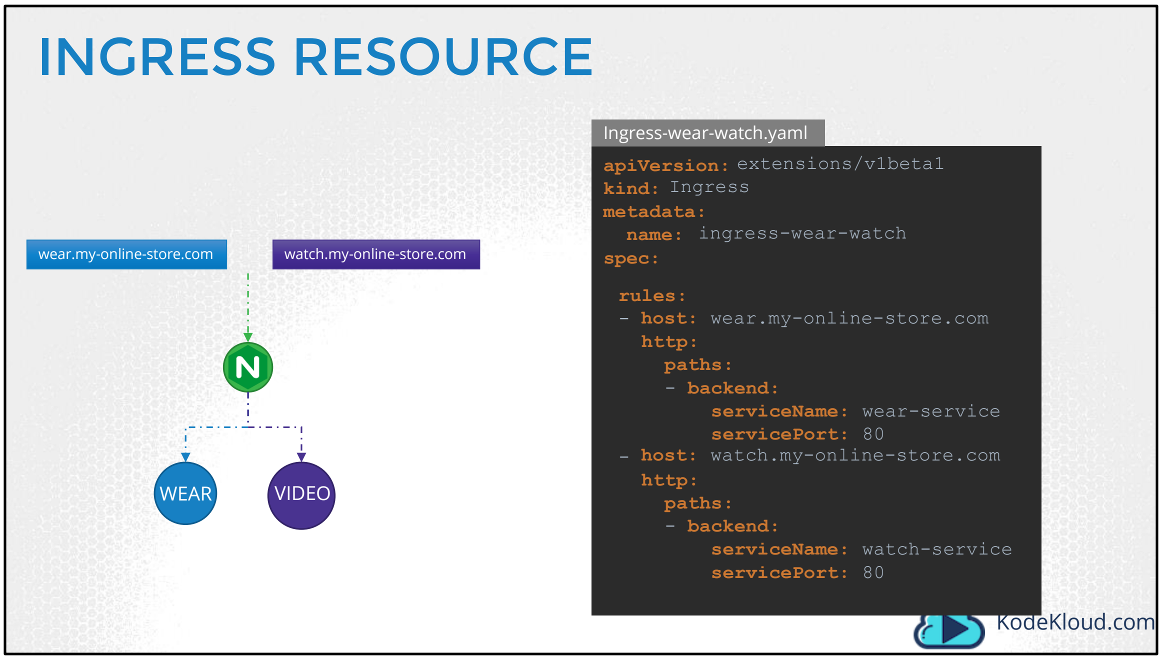
www.my-online-store.com/listen

www.my-online-store.com/wear

www.my-online-store.com/watch

Live Stream
OOPS!!
404
WE ARE NOT THERE YET. BUT WE ARE WORKING ON IT.
KodeKloud.com

Back in your application, say a user visits the URL my-online-store.com/listen or /eat and you don’t have an audio streaming or a food delivery service, you might want to show them a nice message. You can do this by configuring a default backend service to display this 404 Not Found error page.

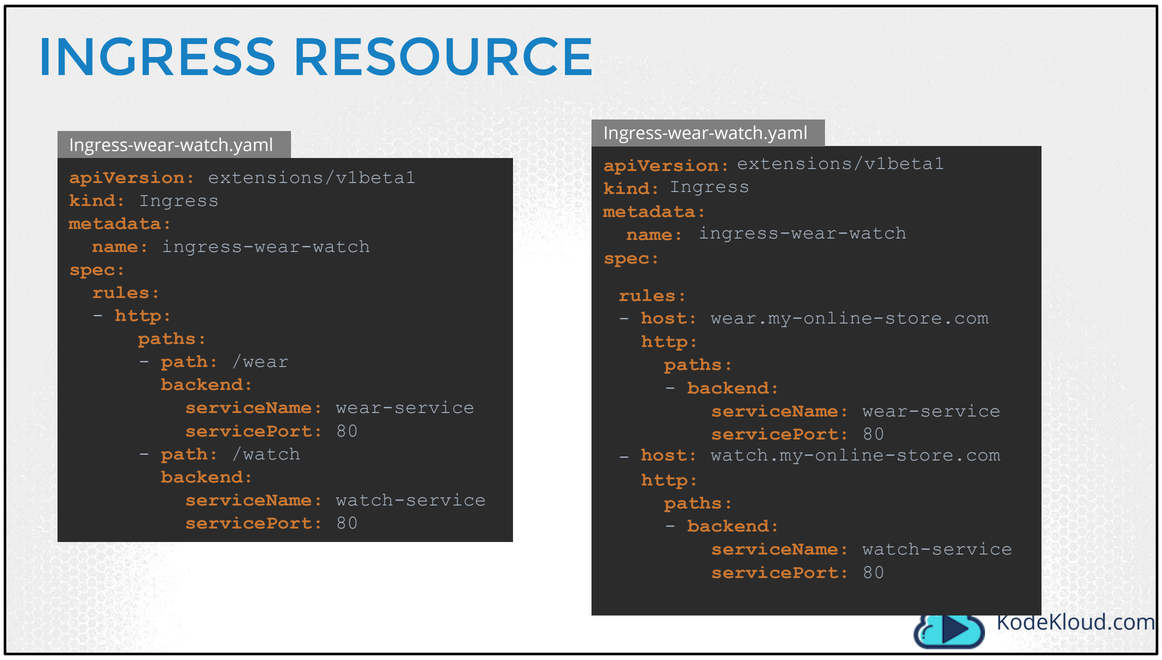


INGRESS RESOURCE

Ingress-wear-watch.yaml

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:

 rules:
 - host: wear.my-online-store.com
   http:
     paths:
     - backend:
         serviceName: wear-service
         servicePort: 80
 - host: watch.my-online-store.com
   http:
     paths:
     - backend:
         serviceName: watch-service
         servicePort: 80
```

The third type of configuration is using domain names or hostnames. We start by creating a similar definition file for Ingress. Now that we have two domain names, we create two rules. One for each domain. To split traffic by domain name, we use the host field. The host field in each rule matches the specified value with the domain name used in the request URL and routes traffic to the appropriate backend. In this case note that we only have a single backend path for each rule. Which is fine. All traffic from these domain names will be routed to the appropriate backend irrespective of the URL Path used. You can still have multiple path specifications in each of these to handle different URL paths.



INGRESS RESOURCE

Ingress-wear-watch.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:
  rules:
  - http:
      paths:
      - path: /wear
        backend:
          serviceName: wear-service
          servicePort: 80
      - path: /watch
        backend:
          serviceName: watch-service
          servicePort: 80
```

Ingress-wear-watch.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:

 rules:
 - host: wear.my-online-store.com
   http:
     paths:
     - backend:
         serviceName: wear-service
         servicePort: 80
 - host: watch.my-online-store.com
   http:
     paths:
     - backend:
         serviceName: watch-service
         servicePort: 80
```

KodeKloud.com

Let’s compare the two. Splitting traffic by URL had just one rule and we split the traffic with two paths. To split traffic by hostname, we used two rules and one path specification in each rule.

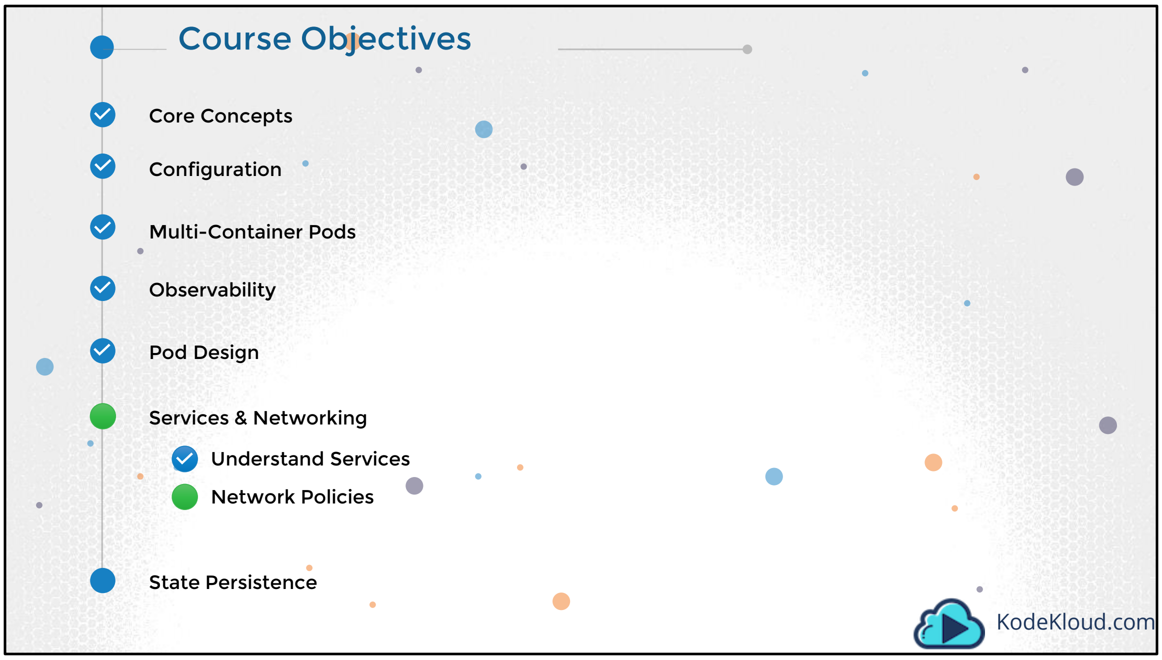
Bare-metal Considerations
https://kubernetes.github.io/ingress-nginx/deploy/baremetal/

https://github.com/kubernetes/ingress-gce/blob/master/README.md

https://kubernetes.github.io/ingress-nginx/deploy/

https://kubernetes.github.io/ingress-nginx/user-guide/nginx-configuration/

KodeKloud.com



Course Objectives

- ✓ Core Concepts
- ✓ Configuration
- ✓ Multi-Container Pods
- ✓ Observability
- ✓ Pod Design
- ● Services & Networking
  - ✓ Understand Services
  - ● Network Policies
- ● State Persistence

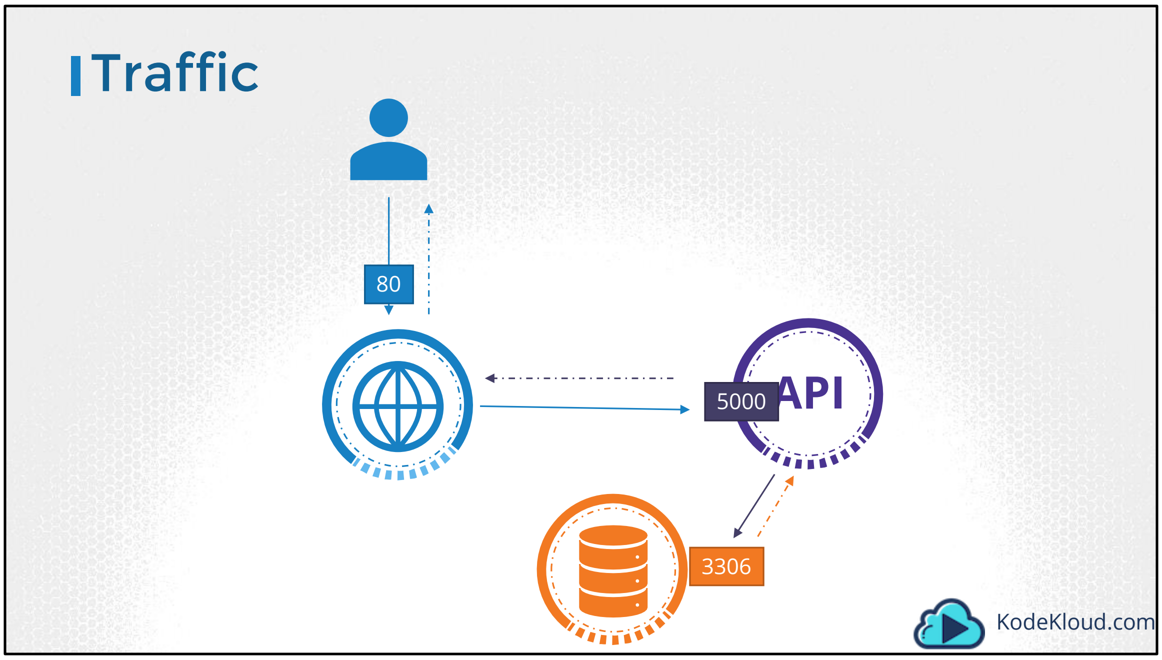
KodeKloud.com

Hello and welcome to this lecture on Network Policies.

326

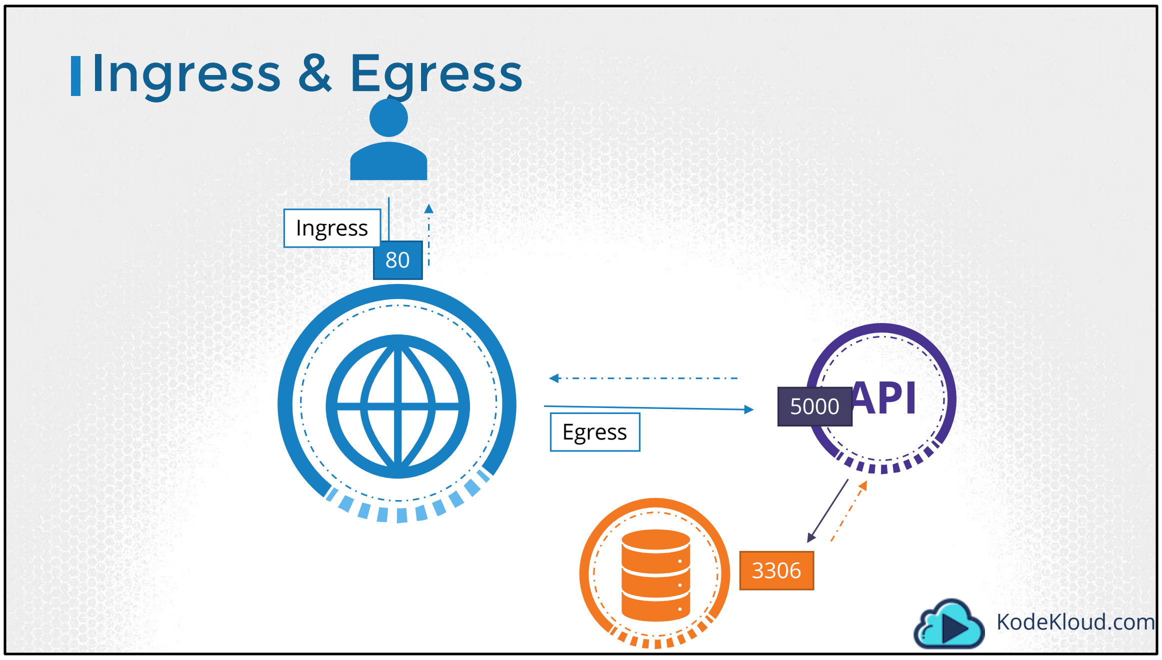
Network Policies

KodeKloud.com



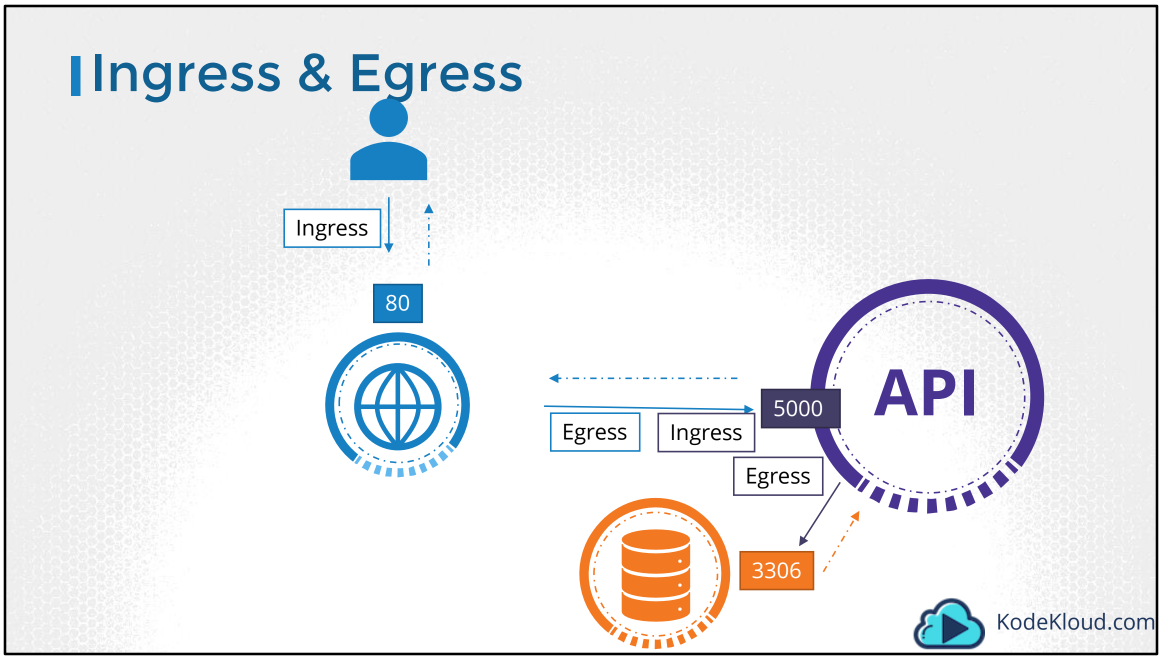
Traffic

So let us first get our networking and security basics right. And I am sorry if this is too basic, but I just wanted to spend a minute on this to make sure we are all on the same page before we begin. We will start with a simple example of a traffic through a web, app and database server. So you have a web server serving front-end to users, an app server serving backend API's and a database server. The user sends in a request to the web server at port 80. The web server then sends a request to the API server at port 5000 in the backend. The API server then fetches data from the database server at port 3306. And then sends the data back to the user. A very simple setup.



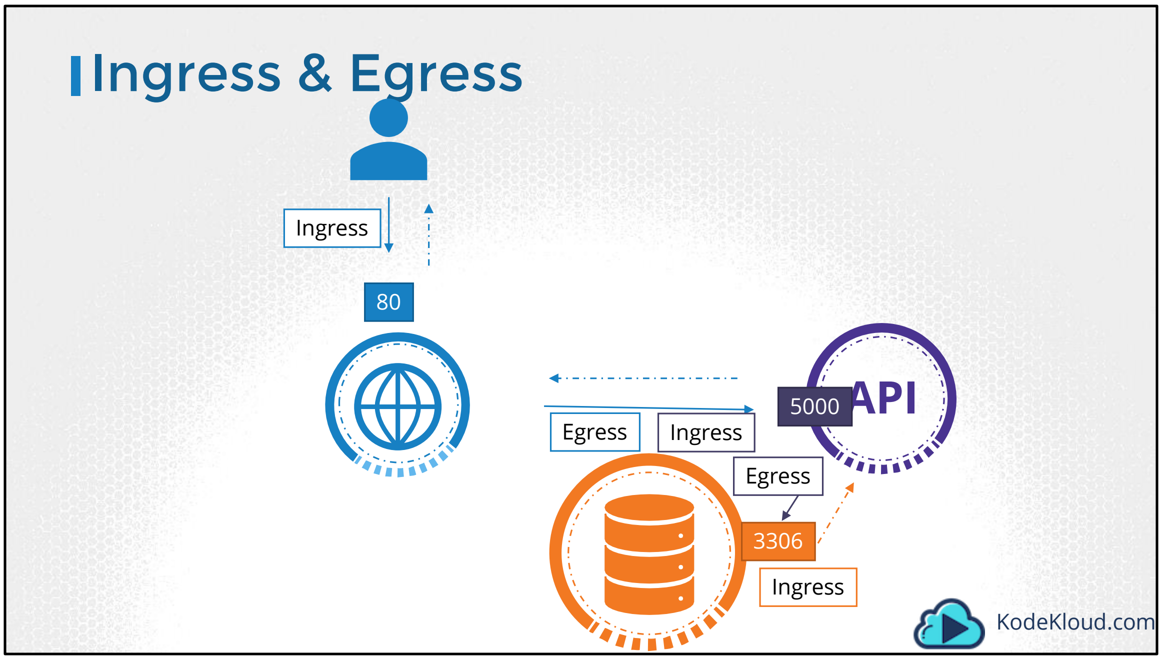
Ingress & Egress

So there are two types of traffic here. Ingress and Egress. For example, for a web server, the incoming traffic from the users is an Ingress Traffic. And the outgoing requests to the app server is Egress traffic. And that is denoted by the straight arrow. When you define ingress and egress, remember you are only looking at the direction in which the traffic originated. The response back to the user, denoted by the dotted lines do not really matter.



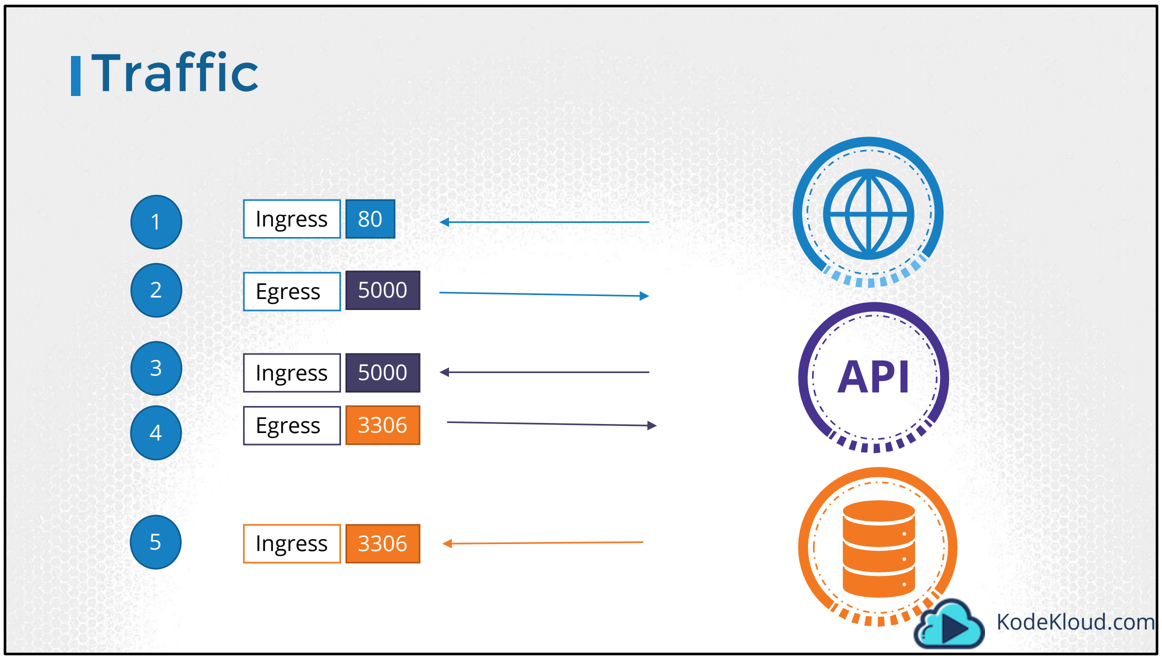
Ingress & Egress

Similarly, in case of the backend API server, it receives ingress traffic from the web server on port 80 and has egress traffic to port 3306 to the database server.



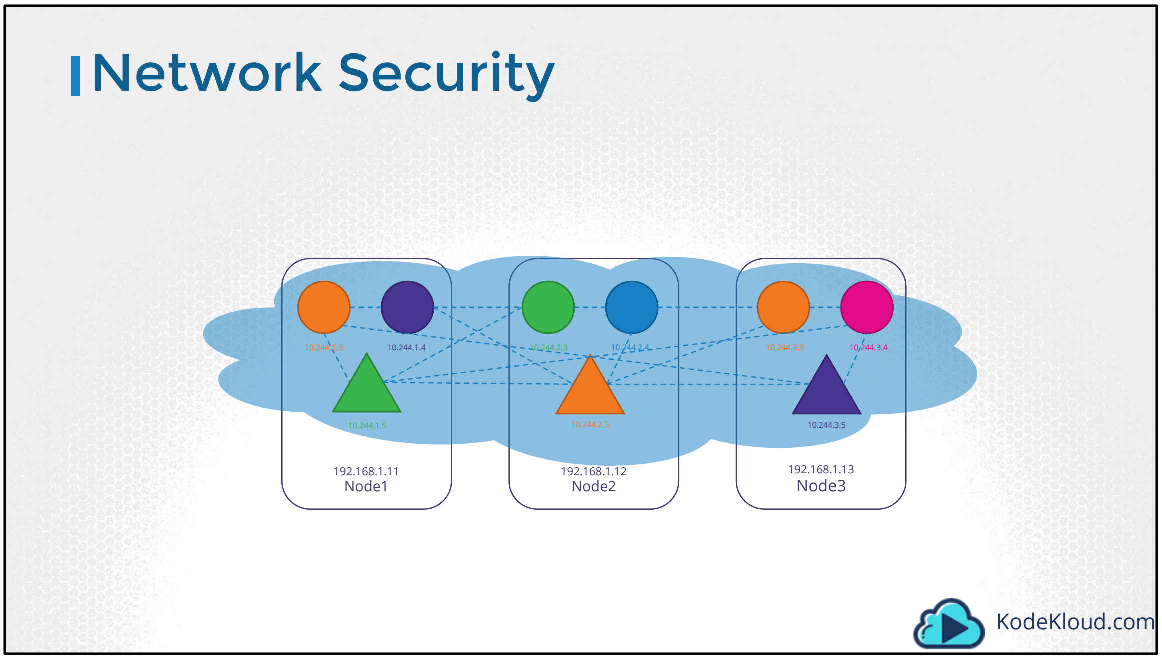
Ingress & Egress

And from the database servers perspective, it receives Ingress traffic on port 3306 from the API server.

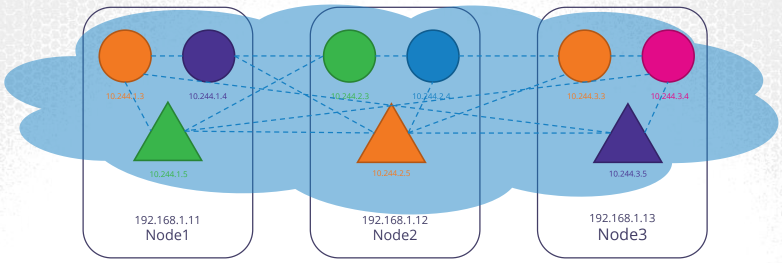


Traffic

If we were to list the rules required to get this working, we would have an Ingress rule that is required to accept HTTP traffic on port 80 on the web server. An Egress rule to allow traffic from the web server to port 5000 on the API server. An ingress rule to accept traffic on port 5000 on the API server and an egress rule to allow traffic to port 3306 on the database server. And finally an ingress rule on the database server to accept traffic on port 3306. So that’s traffic flow and rules basics.



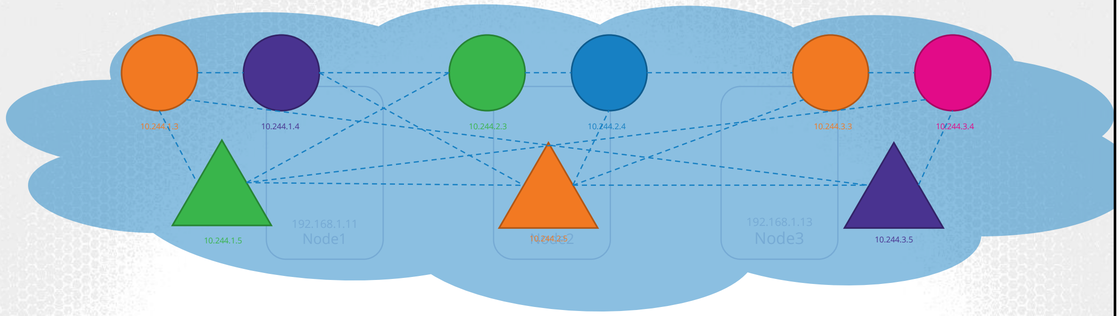
Network Security



KodeKloud.com

Let us now look at Network Security in Kubernetes. So we have a cluster with a set of nodes hosting a set of pods and services. Each node has an IP address and so does each pod as well as service. One of the pre-requisite for networking in kubernetes, is whatever solution you implement, the pods should be able to communicate with each other without having to configure any additional settings, like routes.

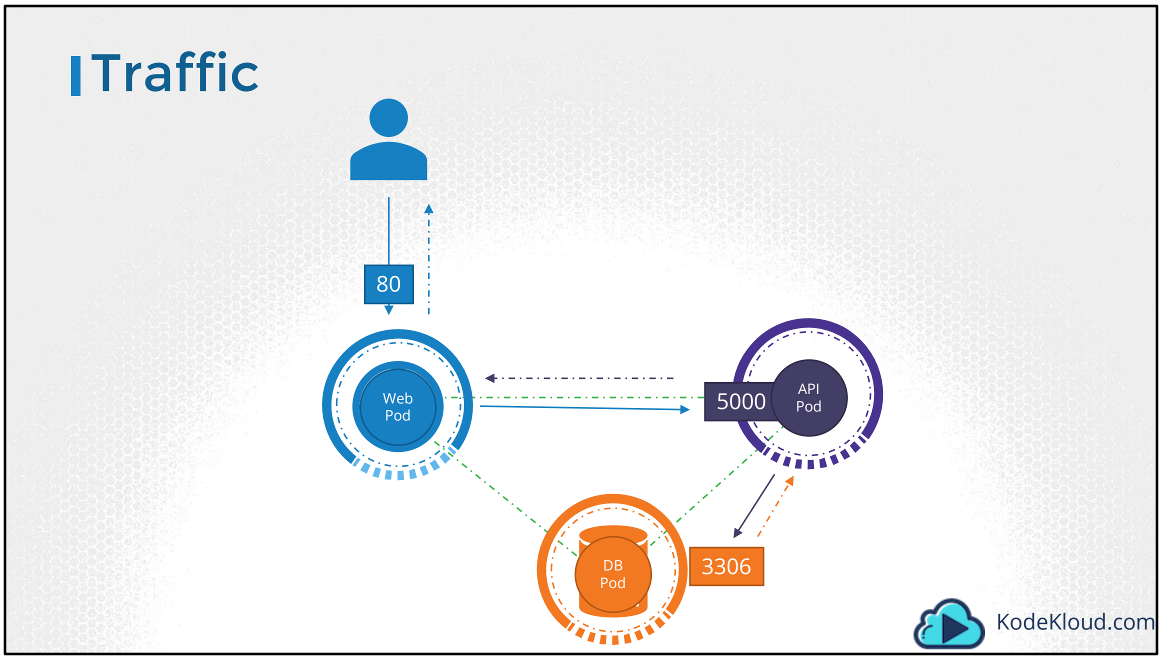
Network Security



"All Allow"

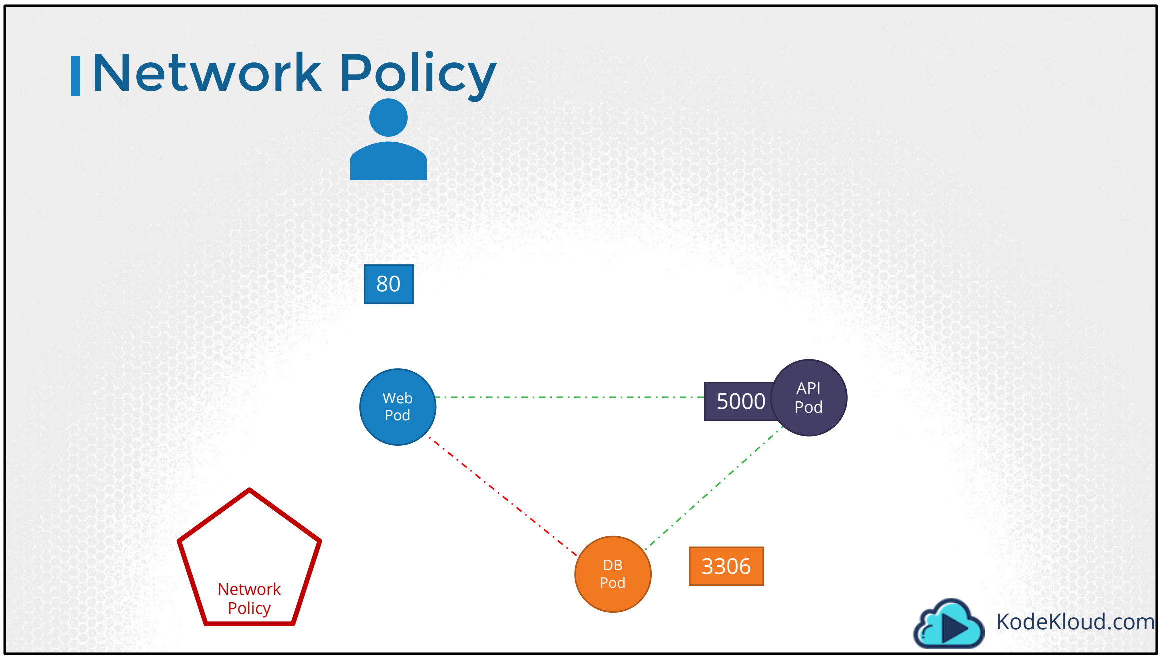
KodeKloud.com

For example, in this network solution, all pods are on a virtual private network that spans across the nodes in the kubernetes cluster. And they can all by default reach each other using the IPs or pod names or services configured for that purpose. Kubernetes is configured by default with an "All Allow" rule that allows traffic from any pod to any other pod or services.


Traffic

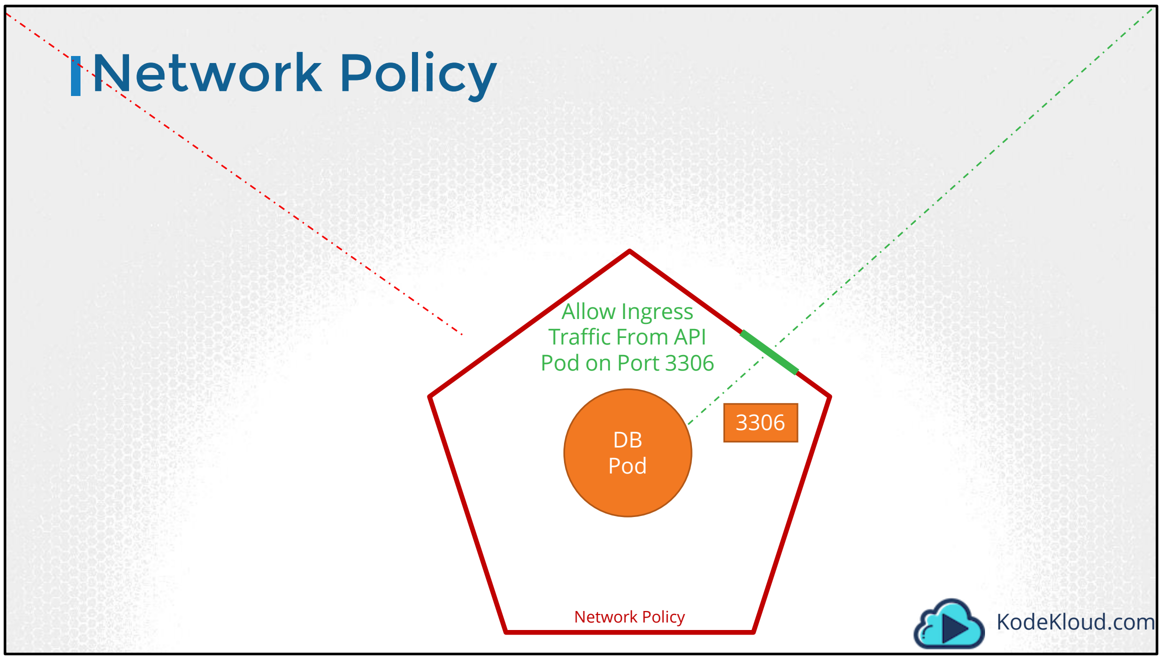
Let us now bring back our earlier discussion and see how it fits in to kubernetes. For each component in the application we deploy a POD. One for the front-end web server, for the API server and one for the database. We create services to enable communication between the PODs as well as to the end user. Based on what we discussed in the previous slide, by default all the three PODs can communicate with each other within the kubernetes cluster.

What if we do not want the front-end web server to be able to communicate with the database server directly? Say for example, the security teams and audits require you to prevent that from happening? That is where you would implement a Network Policy to allow traffic to the db server only from the api server. Let’s see how we do that.



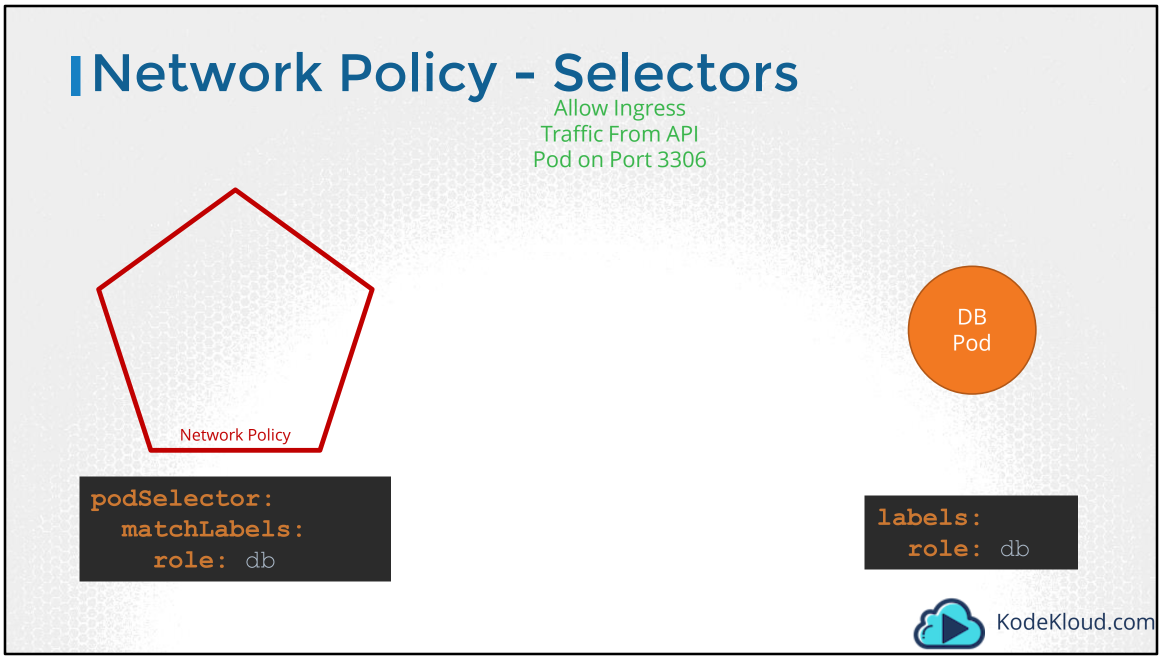
Network Policy

A Network policy is another object in the kubernetes namespace. Just like PODs, ReplicaSets or Services. You apply a network policy on selected pods.



Network Policy

A Network policy is another object in the kubernetes namespace. Just like PODs, ReplicaSets or Services. You link a network policy to one or more pods. You can define rules within the network policy. In this case I would say, only allow Ingress Traffic from the API Pod on Port 3306. Once this policy is created, it blocks all other traffic to the Pod and only allows traffic that matches the specified rule. Again, this is only applicable to the Pod on which the network policy is applied.

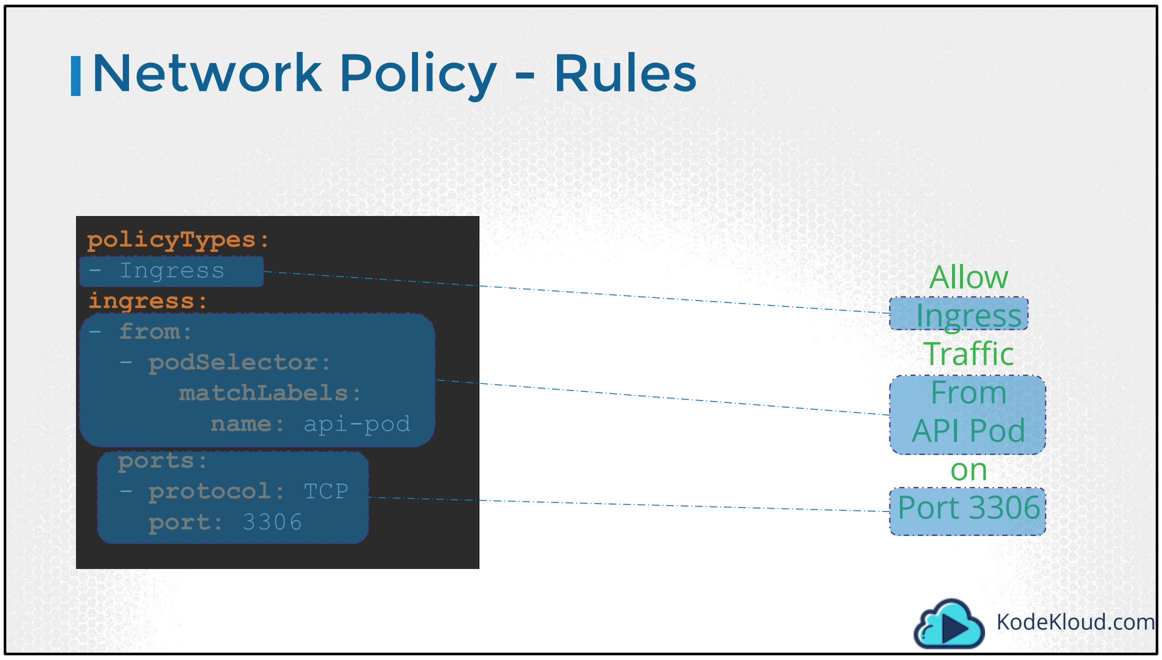


Network Policy - Selectors

```
podSelector:
  matchLabels:
    role: db
```

```
labels:
  role: db
```

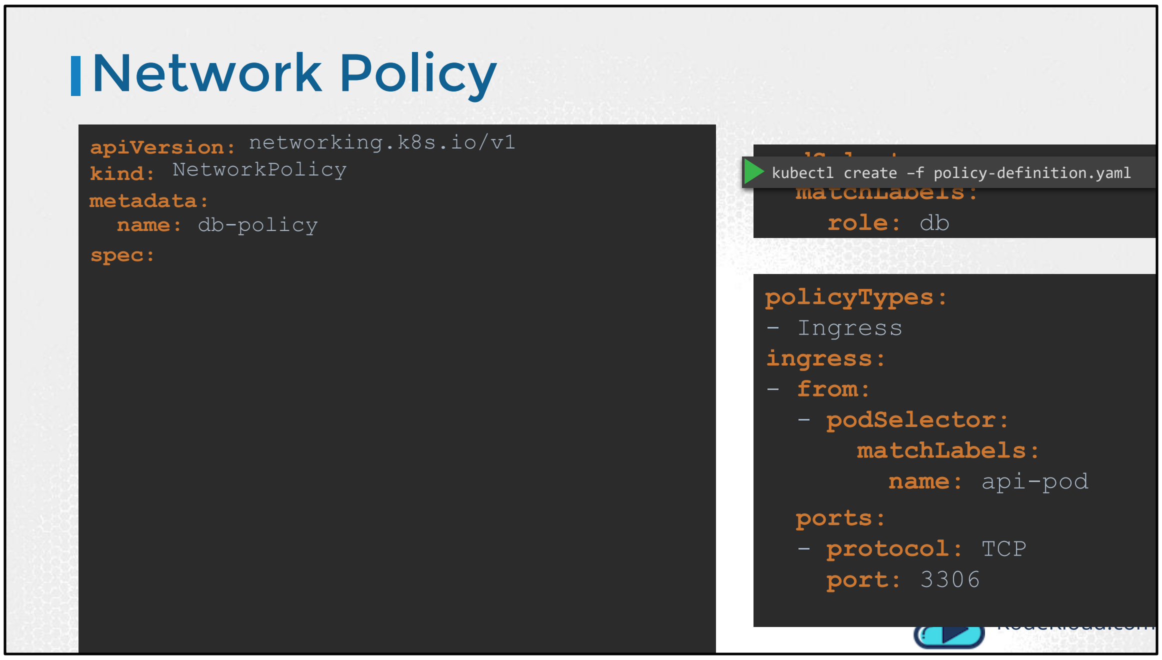
So how do you apply or link a network policy to a Pod? We use the same technique that was used before to link ReplicaSets or Services to Pods. Labels and Selectors. We label the Pod and use the same labels on the pod selector field in the network policy.



Network Policy - Rules

```
policyTypes:
- Ingress
ingress:
- from:
  - podSelector:
      matchLabels:
        name: api-pod
  ports:
  - protocol: TCP
    port: 3306
```

And then we build our rule. Under policyTypes specify weather the rule is to allow ingress or egress traffic or both. In our case we only want to allow ingress traffic to the db-pod. So we add Ingress. Next, we specify the ingress rule, that allows traffic from the API pod. And you specify the api pod, again using labels and selectors. And finally the port to allow traffic on, which is 3306.



Network Policy

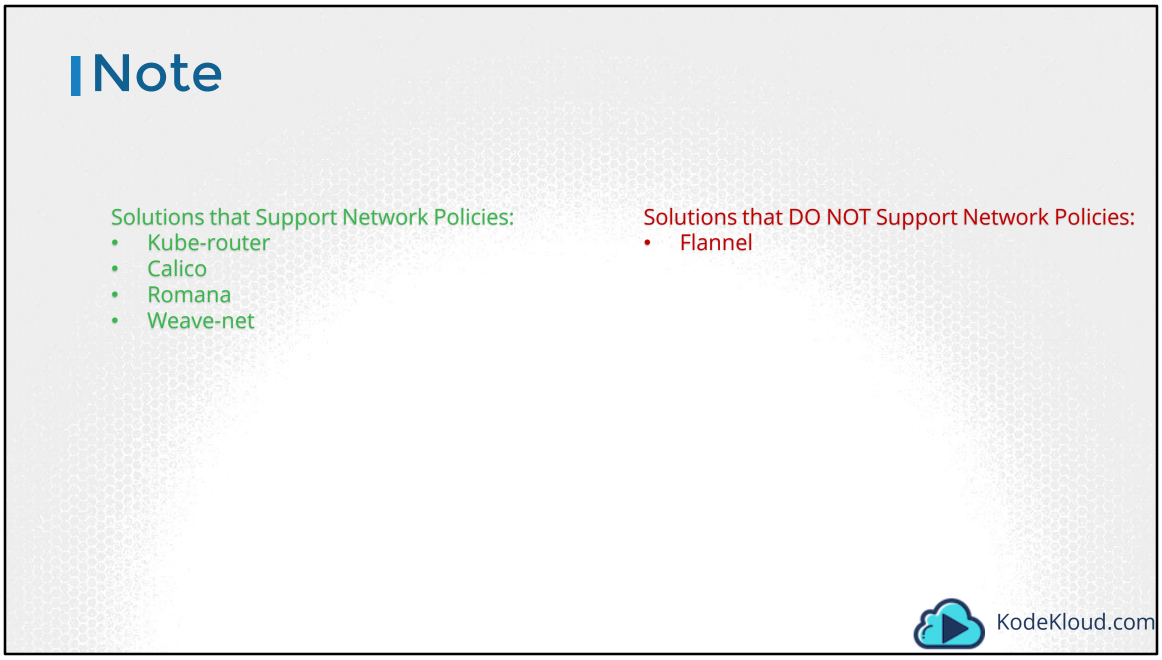
```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
```

```
kubectl create -f policy-definition.yaml
```

```
  matchLabels:
    role: db
```

```
policyTypes:
- Ingress
ingress:
- from:
  - podSelector:
      matchLabels:
        name: api-pod
  ports:
  - protocol: TCP
    port: 3306
```

Let us put all that together. We start with a blank object definition file and as usual we have apiVersion, kind, metadata and spec. The apiVersion is networking.k8s.io/v1, the kind is NetworkPolicy. We will name the policy db-policy. And then under the spec section, we will first move the pod selector to apply this policy to the db pod. Then we will move the rule we created in the previous slide under it. And that’s it. We have our first network policy ready. Run the kubectl create command to create the policy.



Note

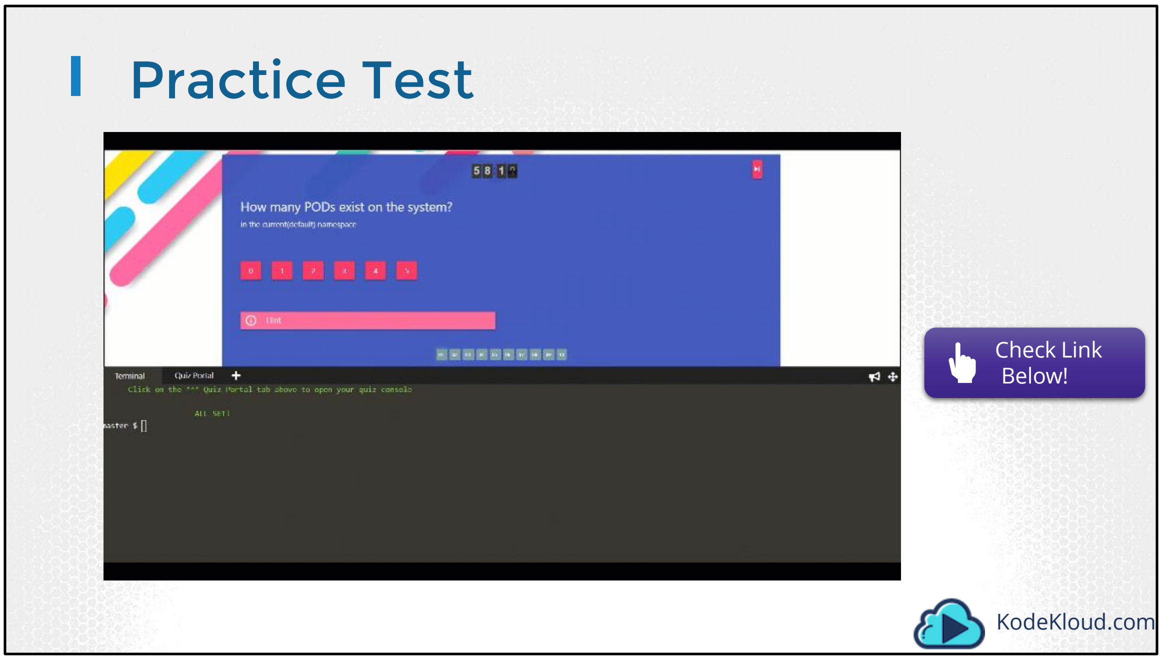
Solutions that Support Network Policies:
- Kube-router
- Calico
- Romana
- Weave-net

Solutions that DO NOT Support Network Policies:
- Flannel

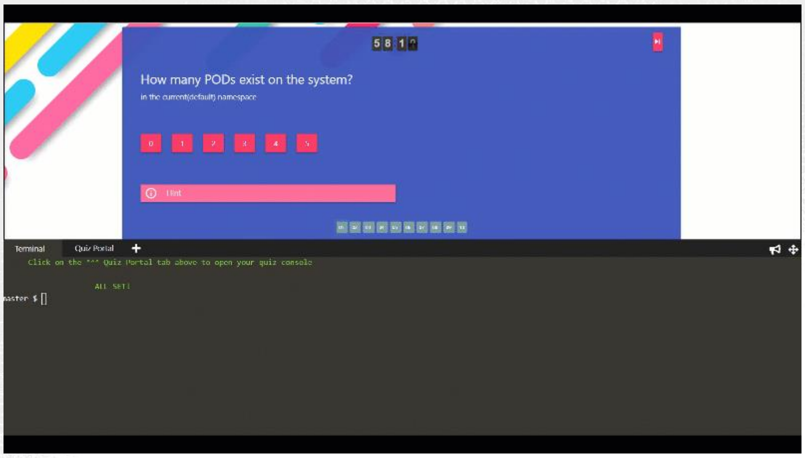
KodeKloud.com

Remember that Network Policies are enforced by the Network Solution implemented on the Kubernetes Cluster. And not all network solutions support network policies. A few of them that are supported are kube-router, Calico, Romana and Weave-net. If you used Flannel as the networking solution, it does not support network policies as of this recording. Always refer to the network solution's documentation to see support for network policies. Also remember, even in a cluster configured with a solution that does not support network policies, you can still create the policies, but they will just not be enforced. You will not get an error message saying the networking solution does not support network policies.

Well, that's it for this lecture. Walk through the documentation and head over to coding challenges to practice network policies.



Practice Test

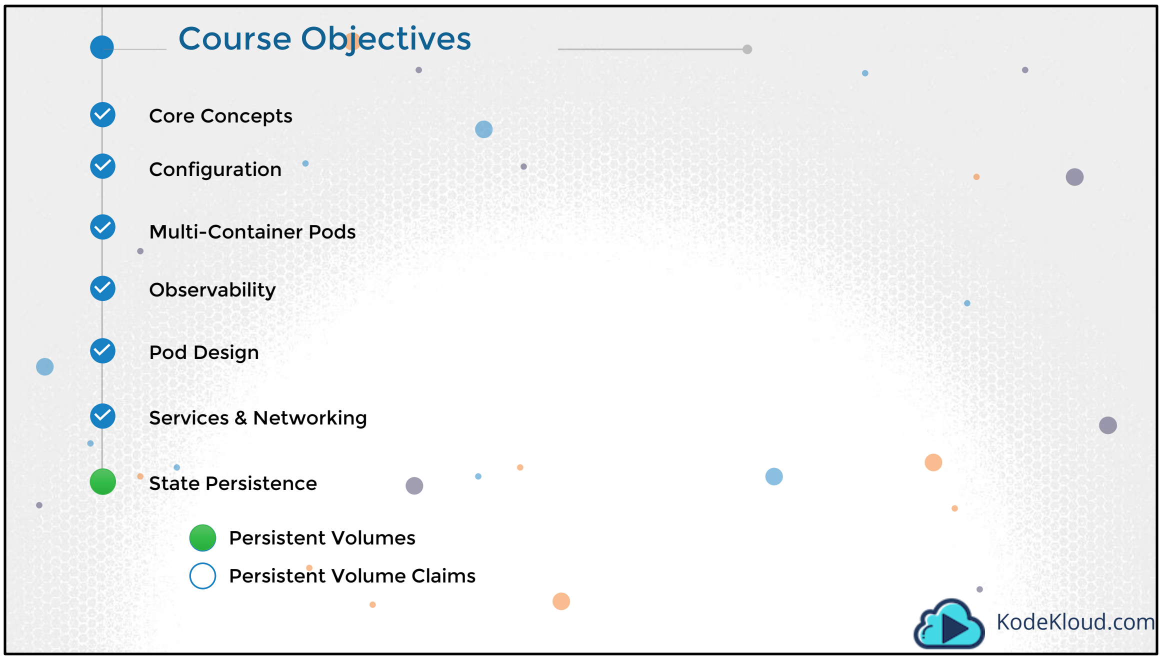


How many PODs exist on the system?

in the current(default) namespace

```
    Click on the ^^^ Quiz Portal tab above to open your quiz console

                ALL SET!
aster $ []
```

Access test here: https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743700



Course Objectives

- ✓ Core Concepts
- ✓ Configuration
- ✓ Multi-Container Pods
- ✓ Observability
- ✓ Pod Design
- ✓ Services & Networking
- ● State Persistence
  - ● Persistent Volumes
  - ○ Persistent Volume Claims

KodeKloud.com

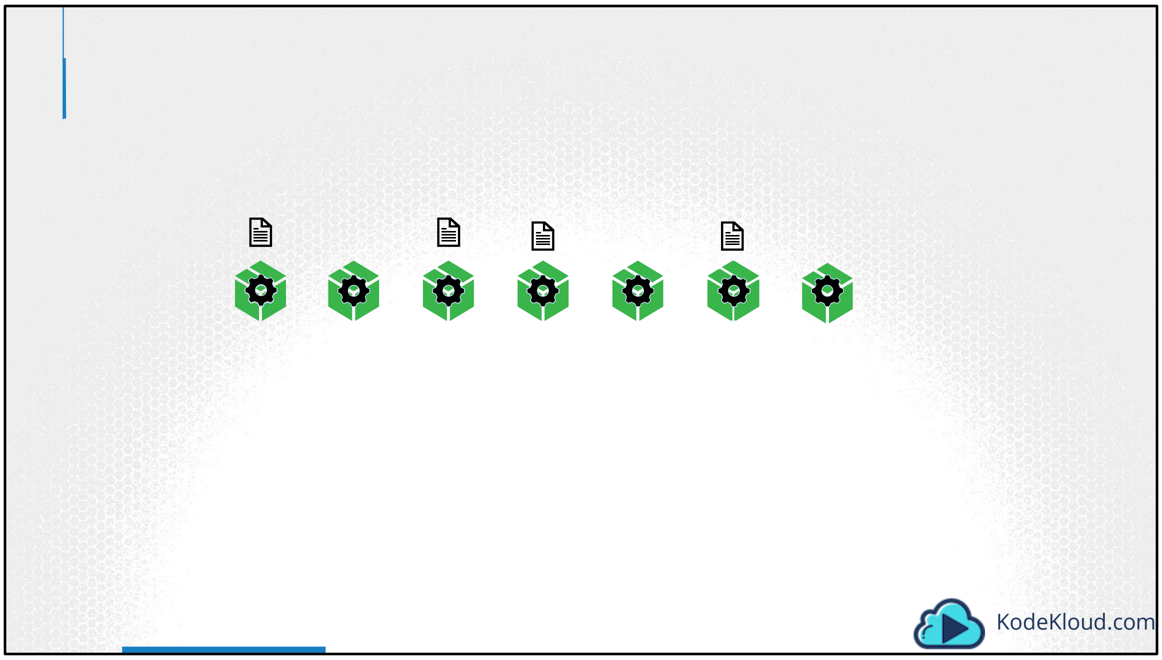
Hello and welcome to this lecture on Persistent Volumes in Kubernetes. My name is Mumshad Mannambeth and we are going through the Certified Kubernetes Application Developer's course.

Volumes

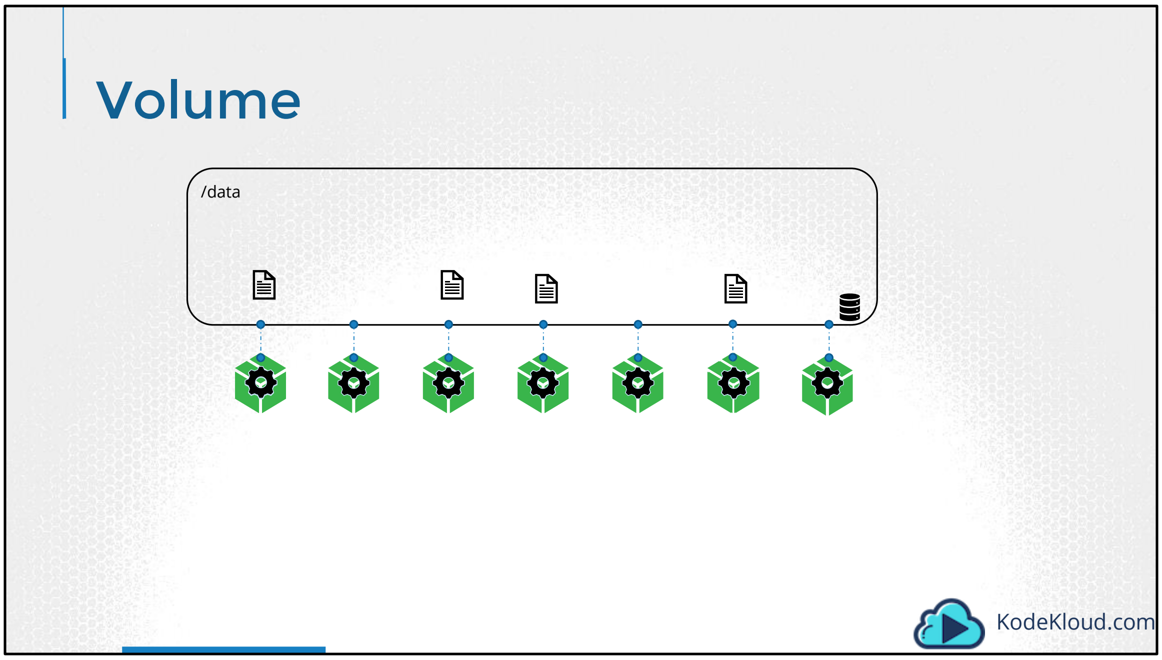
343

KodeKloud.com

Before we head into persistent volumes let us start with Volumes in Kubernetes.

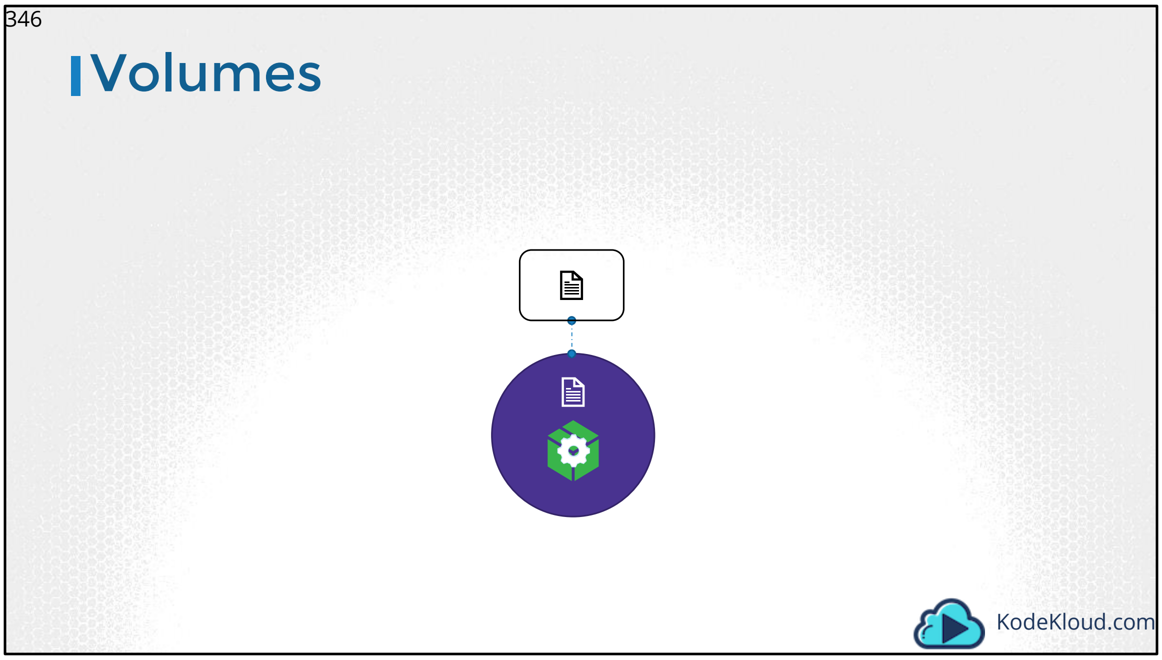


Let us look at volumes in Docker first. Docker containers are meant to be transient in nature. Which means they are meant to last only for a short period of time. They are called upon when required to process data and destroyed once finished. The same is true for the data within the container. The data is destroyed along with the container.



Volume

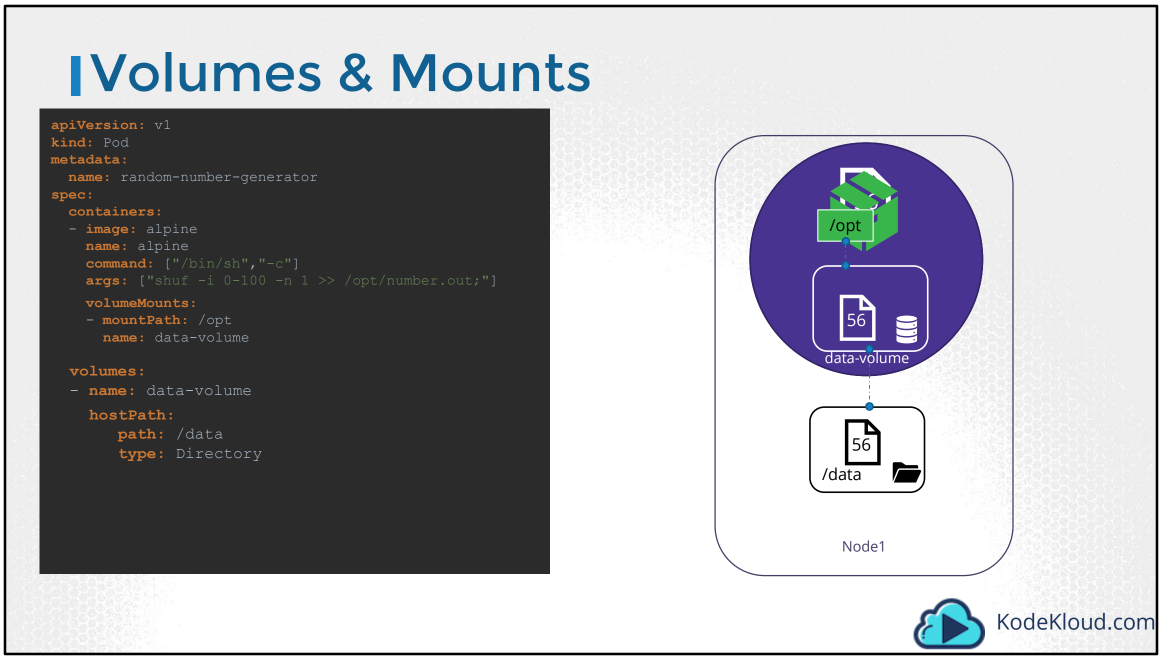
To persist data processed by the containers, we attach a volume to the containers when they are created. The data processed by the container is now placed in this volume, thereby retaining it permanently. Even if the container is deleted, the data generated or processed by it remains.



Volumes

KodeKloud.com

So how does that work in the Kubernetes world. Just as in Docker, the PODs created in Kubernetes are transient in nature. When a POD is created to process data and then deleted, the data processed by it gets deleted as well. For this we attach a volume to the POD. The data generated by the POD is now stored in the volume, and even after the POD is delete, the data remains.

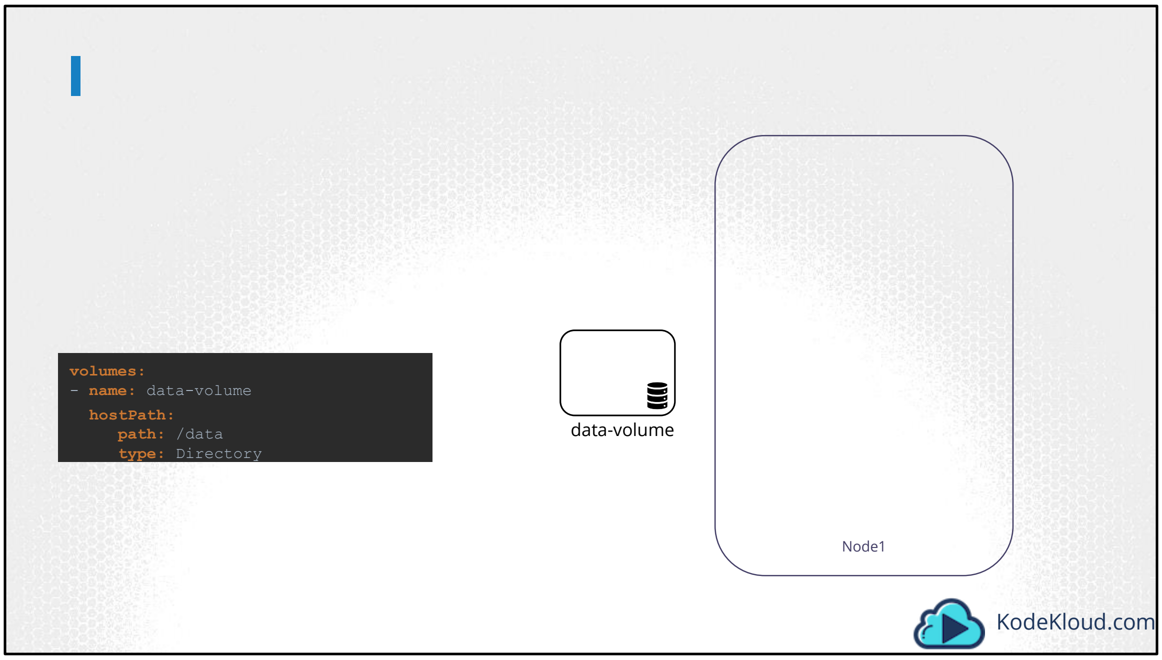


Volumes & Mounts

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: random-number-generator
spec:
  containers:
  - image: alpine
    name: alpine
    command: ["/bin/sh","-c"]
    args: ["shuf -i 0-100 -n 1 >> /opt/number.out;"]
    volumeMounts:
    - mountPath: /opt
      name: data-volume

  volumes:
  - name: data-volume
    hostPath:
        path: /data
        type: Directory
```

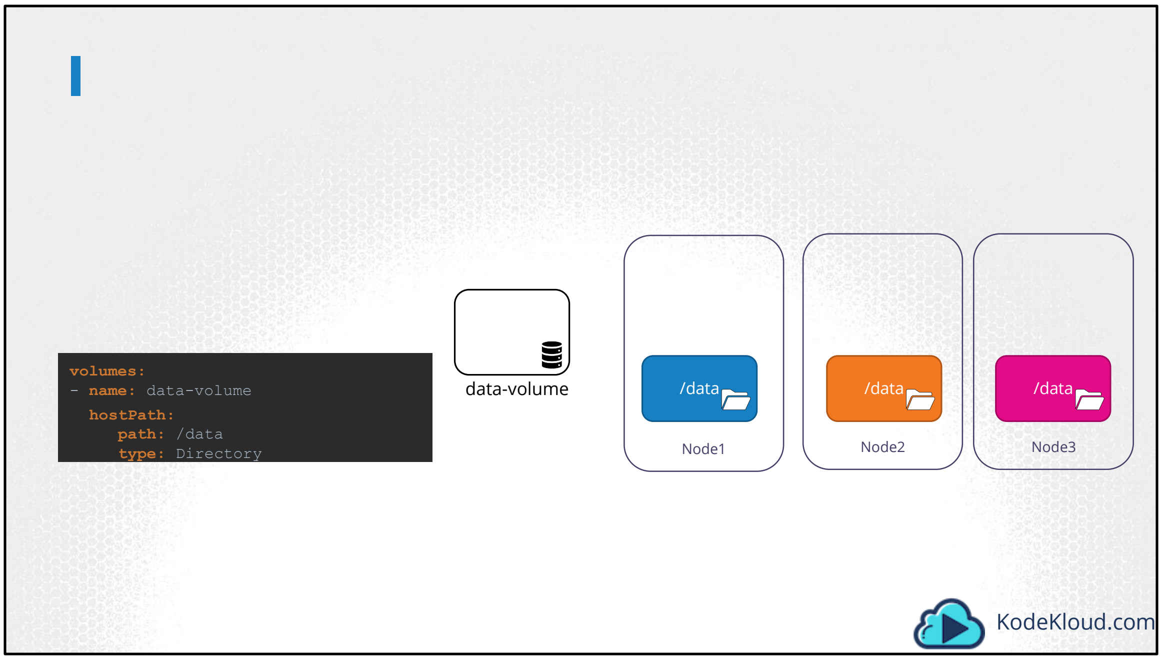
Let’s look at a simple implementation of volumes. We have a single node kubernetes cluster. We create a simple POD that generates a random between 1 and 100 and writes that to a file at /data/number.out and then gets deleted along with the random number. To retain the number generated by the pod, we create a volume. And a Volume needs a storage. When you create a volume you can chose to configure it’s storage in different ways. We will look at the various options in a bit, but for now we will simply configure it to use a directory on the host. In this case I specify a path /data on the host. This way any files created in the volume would be stored in the directory data on my node.

Once the volume is created, to access it from a container we mount the volume to a directory inside the container. We use the volumeMounts field in each container to mount the data-volume to the directory /opt within the container. The random number will now be written to /opt mount inside the container, which happens to be on the data-volume which is in fact /data directory on the host. When the pod gets deleted, the file with the random number still lives on the host.



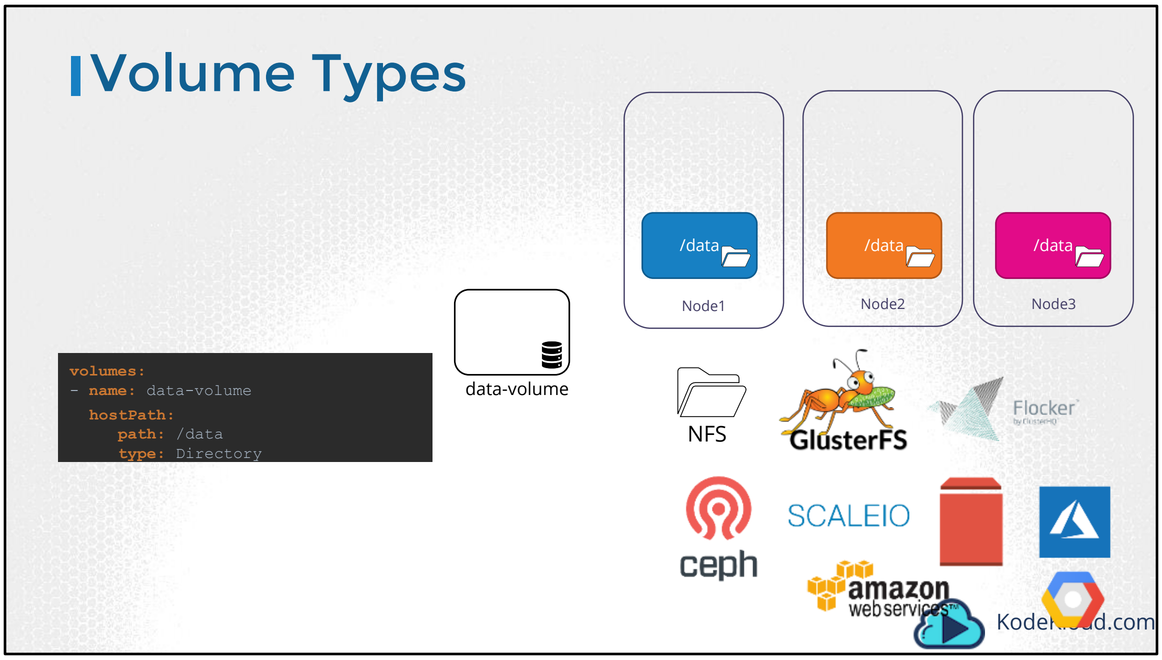
```
volumes:
- name: data-volume
  hostPath:
      path: /data
      type: Directory
```

Let’s take a step back and look at the Volume Storage option. We just used the hostPath option to configure a directory on the host as storage space for the volume. Now that works on a single node.



```
volumes:
- name: data-volume
  hostPath:
     path: /data
     type: Directory
```

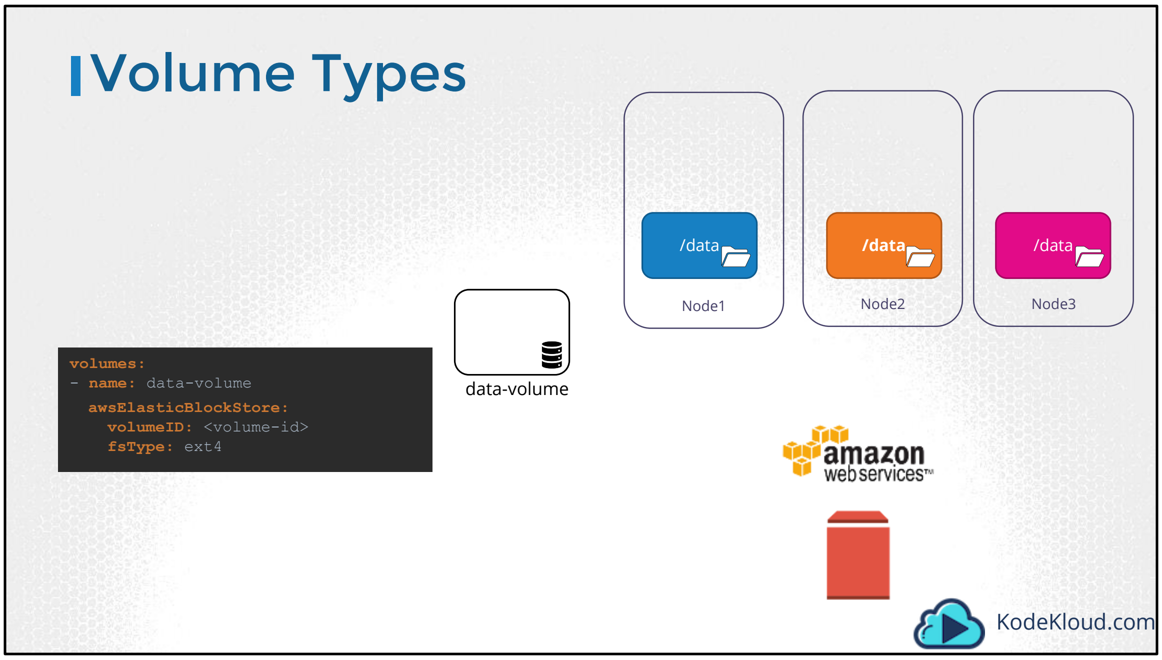
However it is not recommended for use in a multi-node cluster. This is because the PODs would use the /data directory on all the nodes, and expect all of them to be the same and have the same data. Since they are on different servers, they are in fact not the same, unless you configure some kind of external replicated clustered storage solution.



Volume Types

```
volumes:
- name: data-volume
  hostPath:
     path: /data
     type: Directory
```

Kubernetes supports several types of standard storage solutions such as NFS, glusterFS, Flocker, FibreChannel, CephFS, ScaleIO or public cloud solutions like AWS EBS, Azure Disk or File or Google’s Persistent Disk.

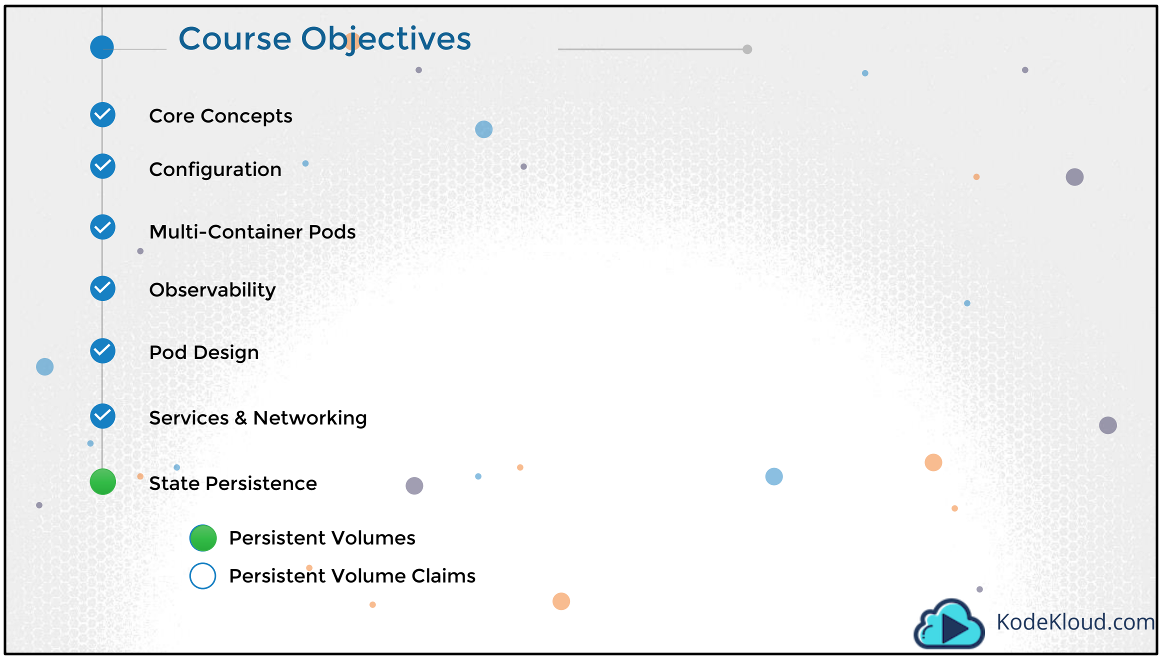


Volume Types

```
volumes:
- name: data-volume
  awsElasticBlockStore:
    volumeID: <volume-id>
    fsType: ext4
```

For example, to configure an AWS Elastic Block Store volume as the storage or the volume, we replace hostPath field of the volume with awsElasticBlockStore field along with the volumeID and filesystem type. The Volume storage will now be on AWS EBS.

Well, that’s it about Volumes in Kubernetes. We will now head over to discuss Persistent Volumes next.



Course Objectives

- ✓ Core Concepts
- ✓ Configuration
- ✓ Multi-Container Pods
- ✓ Observability
- ✓ Pod Design
- ✓ Services & Networking
- ● State Persistence
  - ● Persistent Volumes
  - ○ Persistent Volume Claims

KodeKloud.com

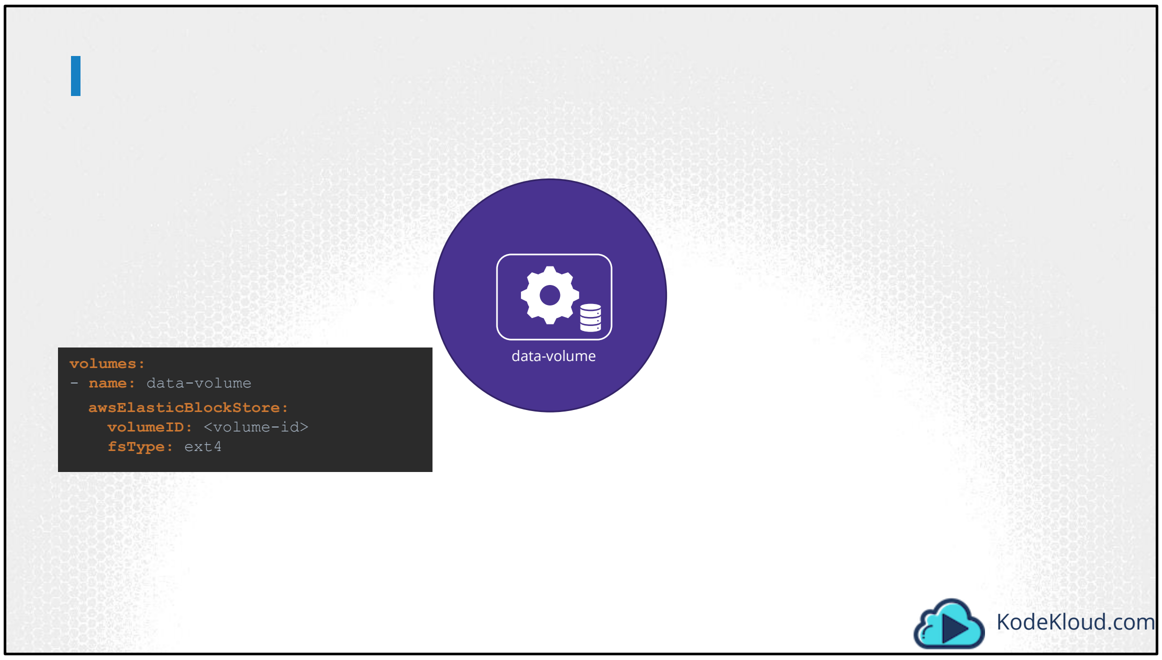
Hello and welcome to this lecture on Persistent Volumes in Kubernetes. My name is Mumshad Mannambeth and we are going through the Certified Kubernetes Application Developer's course.

Persistent Volumes

353

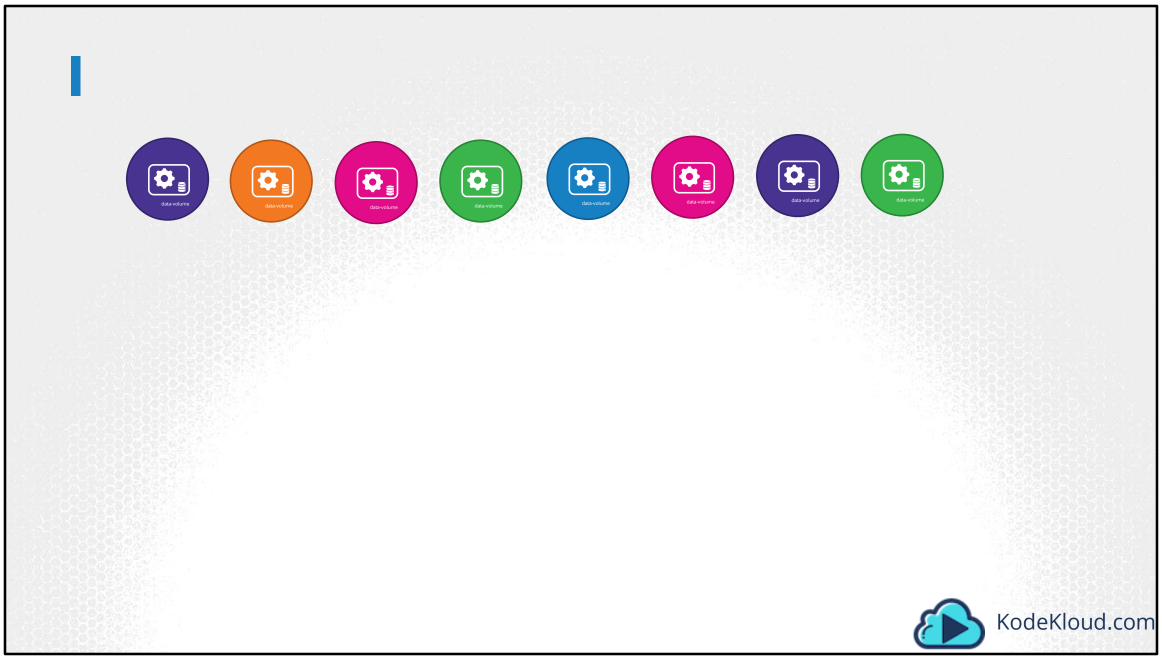
KodeKloud.com

In the last lecture we learned about Volumes. Now we will discuss Persistent Volumes in Kubernetes.

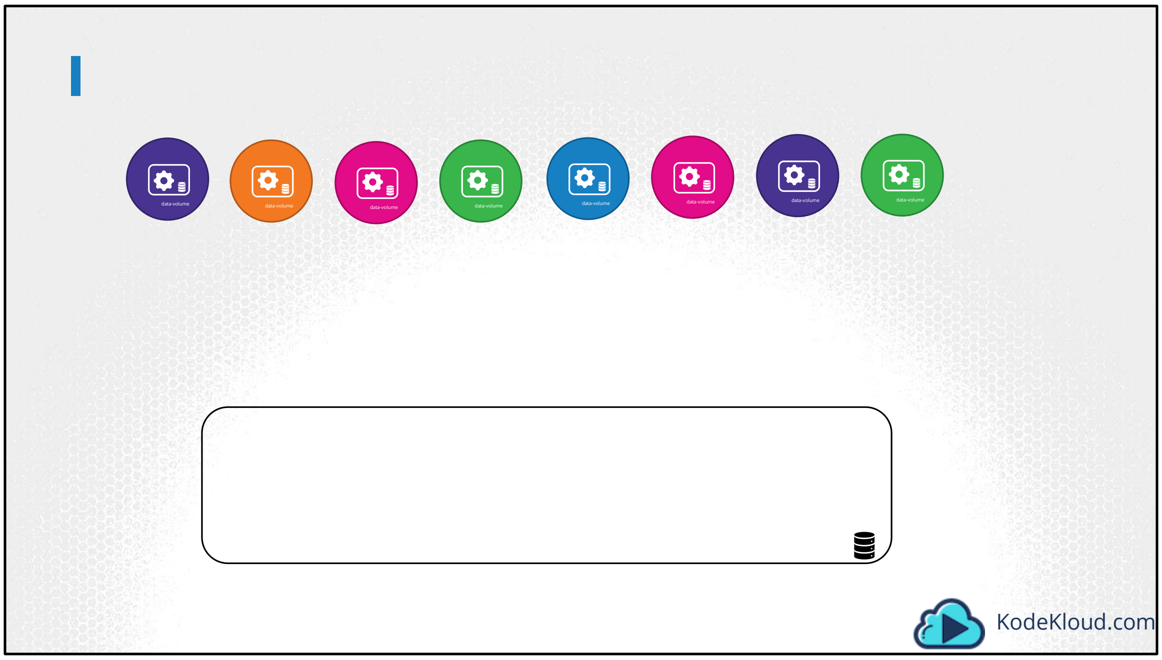


```
volumes:
- name: data-volume
  awsElasticBlockStore:
    volumeID: <volume-id>
    fsType: ext4
```

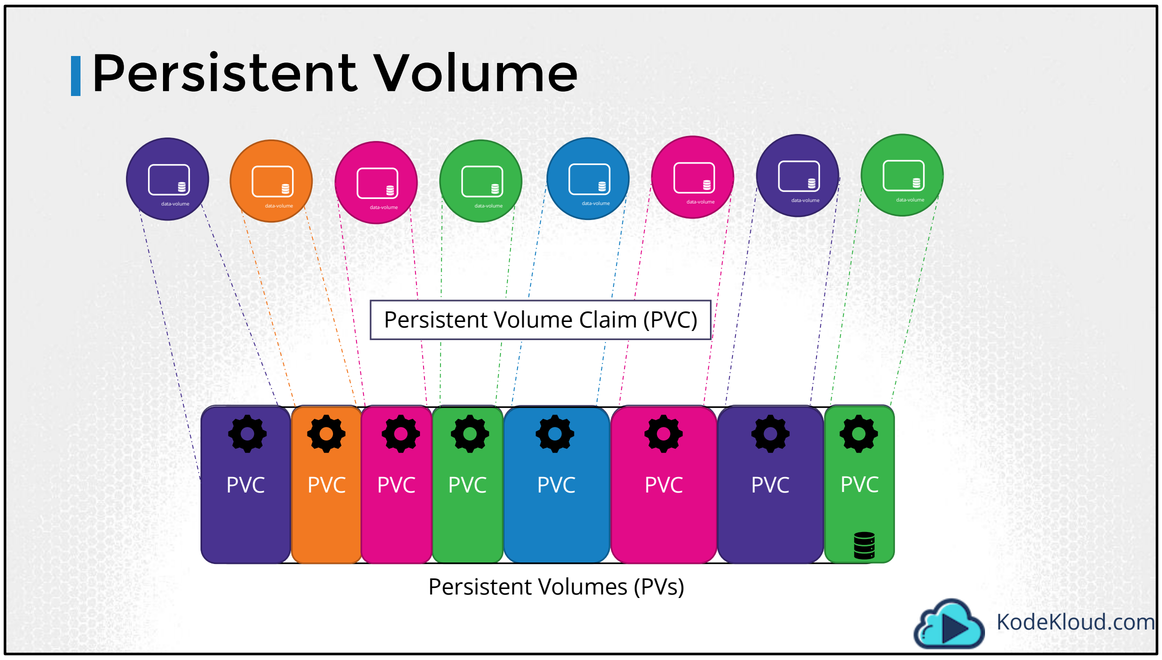
When we created volumes in the previous section we configured volumes within the POD definition file. So every configuration information required to configure storage for the volume goes within the pod definition file.



Now, when you have a large environment with a lot of users deploying a lot of PODs, the users would have to configure storage every time for each POD. Whatever storage solution is used, the user who deploys the PODs would have to configure that on all POD definition files in his environment. Every time a change is to be made, the user would have to make them on all of his PODs.

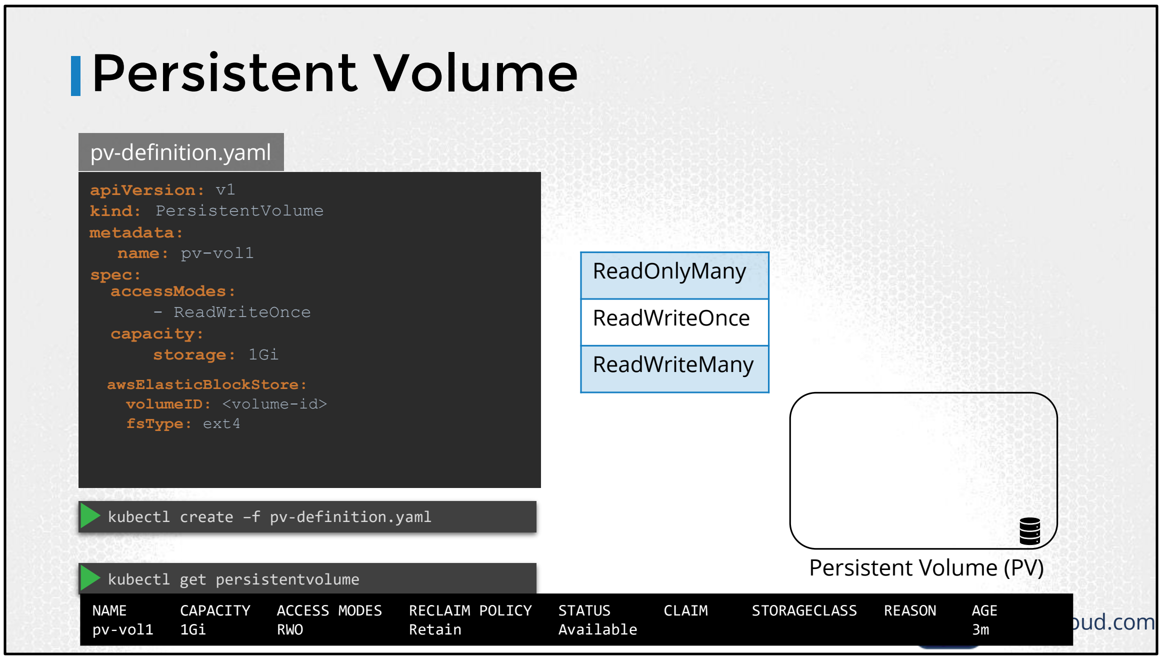


Instead, you would like to manage storage more centrally.



Persistent Volume

You would like it to be configured in a way that an administrator can create a large pool of storage, and then have users carve out pieces from it as required. That is where Persistent Volumes can help us. A Persistent Volume is a Cluster wide pool of storage volumes configured by an Administrator, to be used by users deploying applications on the cluster. The users can now select storage from this pool using Persistent Volume Claims.



Persistent Volume

pv-definition.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
   name: pv-vol1
spec:
  accessModes:
      - ReadWriteOnce
  capacity:
      storage: 1Gi

  awsElasticBlockStore:
    volumeID: <volume-id>
    fsType: ext4
```

```
kubectl create -f pv-definition.yaml
```

```
kubectl get persistentvolume
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM     STORAGECLASS   REASON    AGE
pv-vol1   1Gi        RWO            Retain           Available                                      3m
```

Let us now create a Persistent Volume. We start with the base template and update the apiVersion, set the Kind to PersistentVolume, and name it pv-vol1. Under the spec section specify the accessModes.

Access Mode defines how the Volume should be mounted on the hosts. Weather in a ReadOnly mode, or ReadWrite mode. The supported values are ReadOnlyMany, ReadWriteOnce or ReadWriteMany mode.

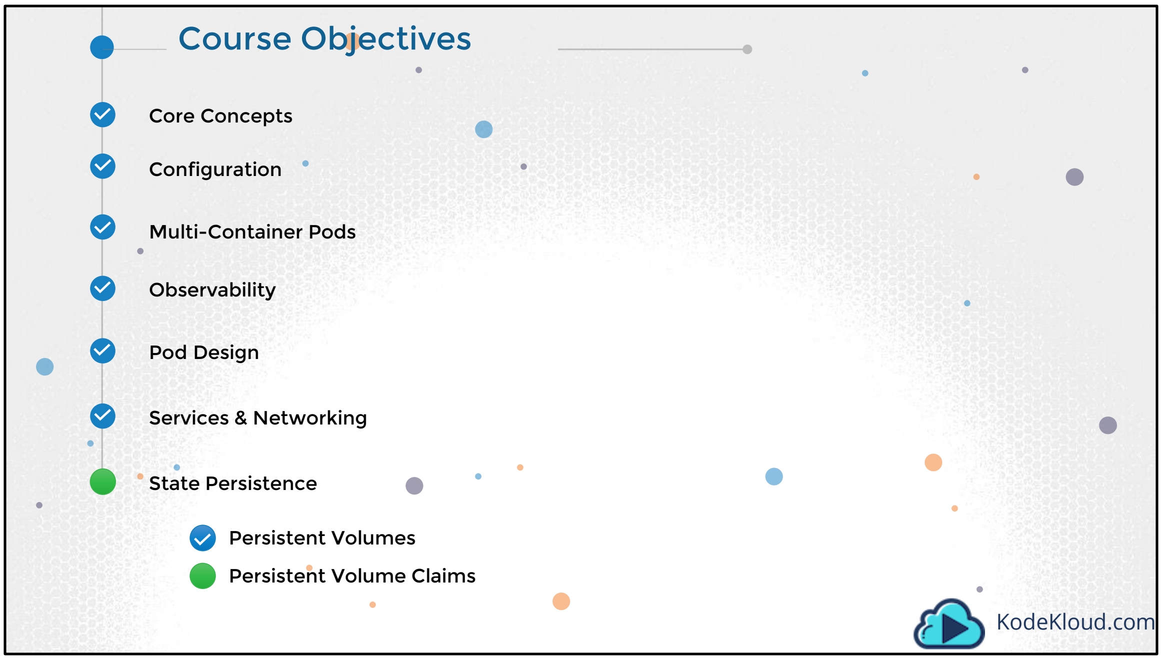
Next, is the capacity. Specify the amount of storage to be reserved for this Persistent Volume. Which is set to 1GB here.

Next comes the volume type. We will start with the hostPath option that uses storage from the node's local directory. Remember this option is not to be used in a production environment.

To create the volume run the kubectl create command and to list the created volume run the kubectl get persistentvolume command.

Replace the hostPath option with one of the supported storage solutions as we saw in the previous lecture like AWS Elastic Block Store.

Well that’s it on Persistent Volumes in this lecture. Head over to coding challenges and practice configuring Persistent Volumes.



Course Objectives

- ✓ Core Concepts
- ✓ Configuration
- ✓ Multi-Container Pods
- ✓ Observability
- ✓ Pod Design
- ✓ Services & Networking
- ● State Persistence
  - ✓ Persistent Volumes
  - ● Persistent Volume Claims

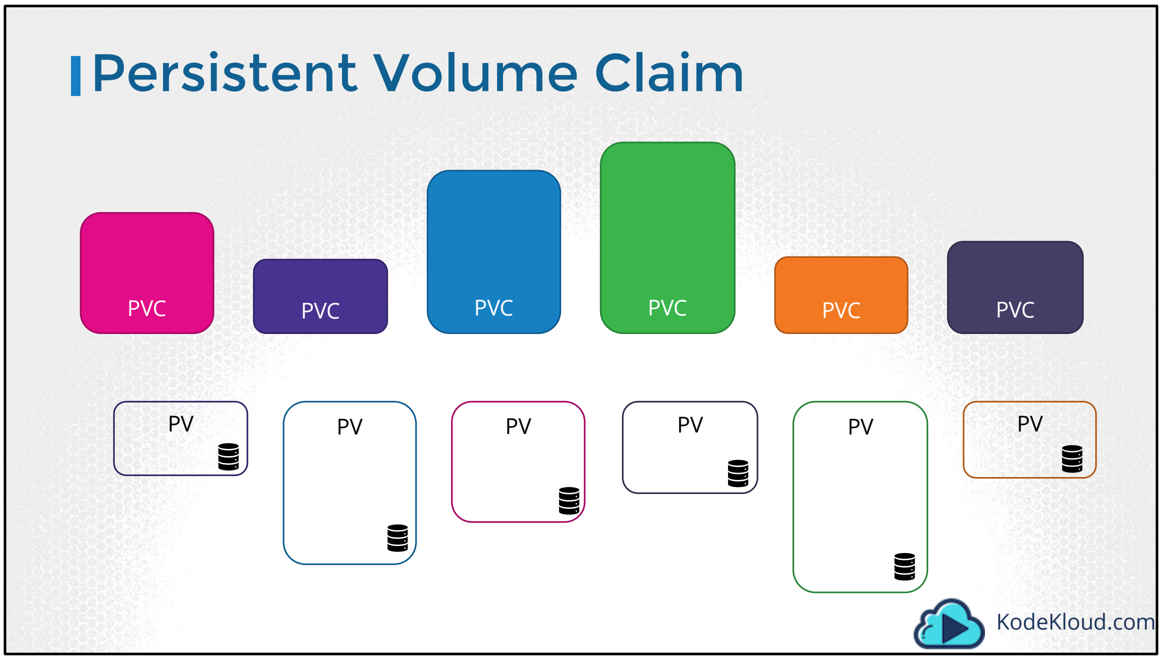
KodeKloud.com

Hello and welcome to this lecture on Persistent Volumes in Kubernetes. My name is Mumshad Mannambeth and we are going through the Certified Kubernetes Application Developer's course.

Persistent Volume Claims

KodeKloud.com

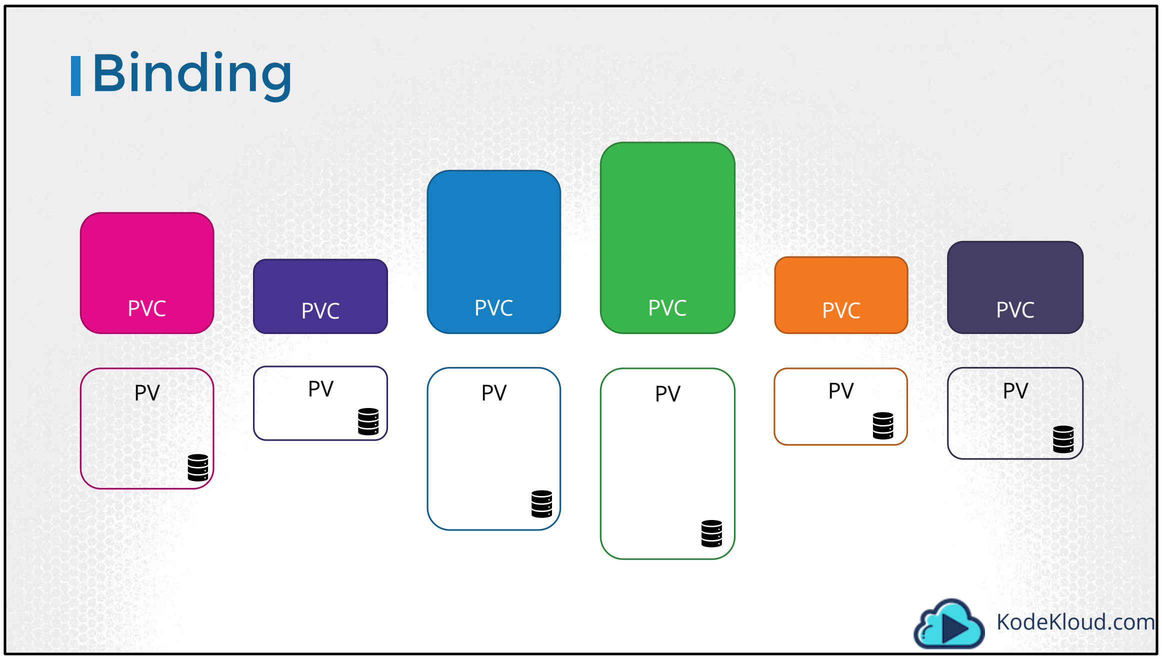
In the last lecture we learned about Volumes. Now we will discuss Persistent Volumes in Kubernetes.



Persistent Volume Claim

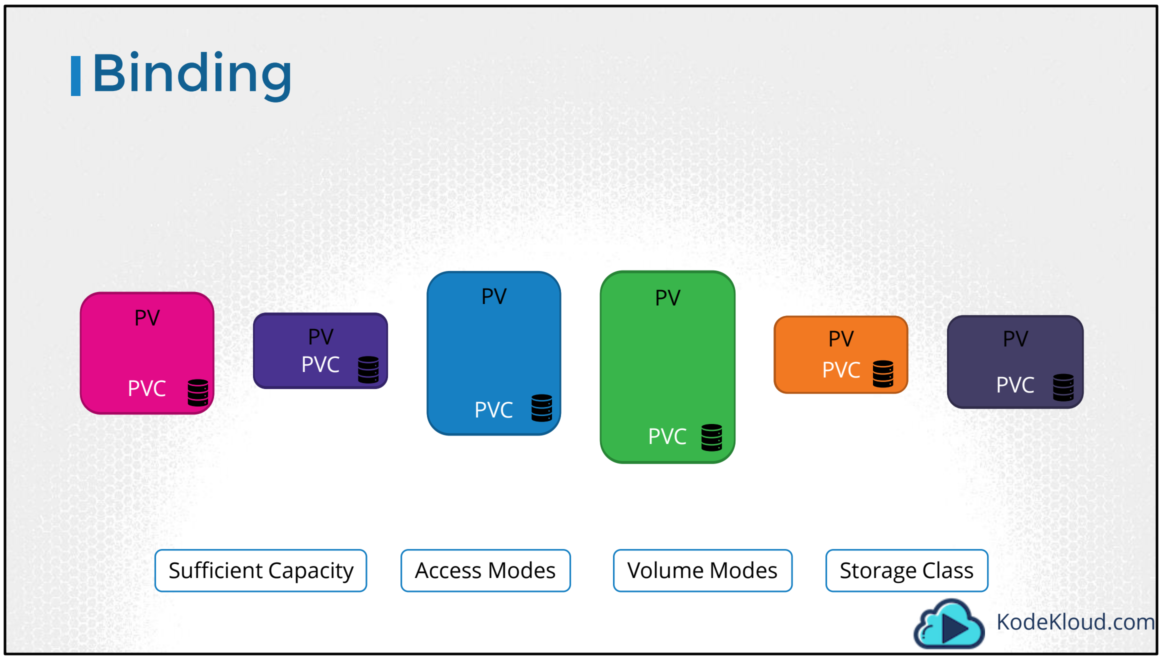
In the previous lecture we created a Persistent Volume. Now we will create a Persistent Volume Claim to make the storage available to a node.

Persistent Volumes and Persistent Volume Claims are two separate objects in the Kubernetes namespace. An Administrator creates a set of Persistent Volumes and a user creates Persistent Volume Claims to use the storage. Once the Persistent Volume Claims are created, Kubernetes binds the Persistent Volumes to Claims based on the request and properties set on the volume.



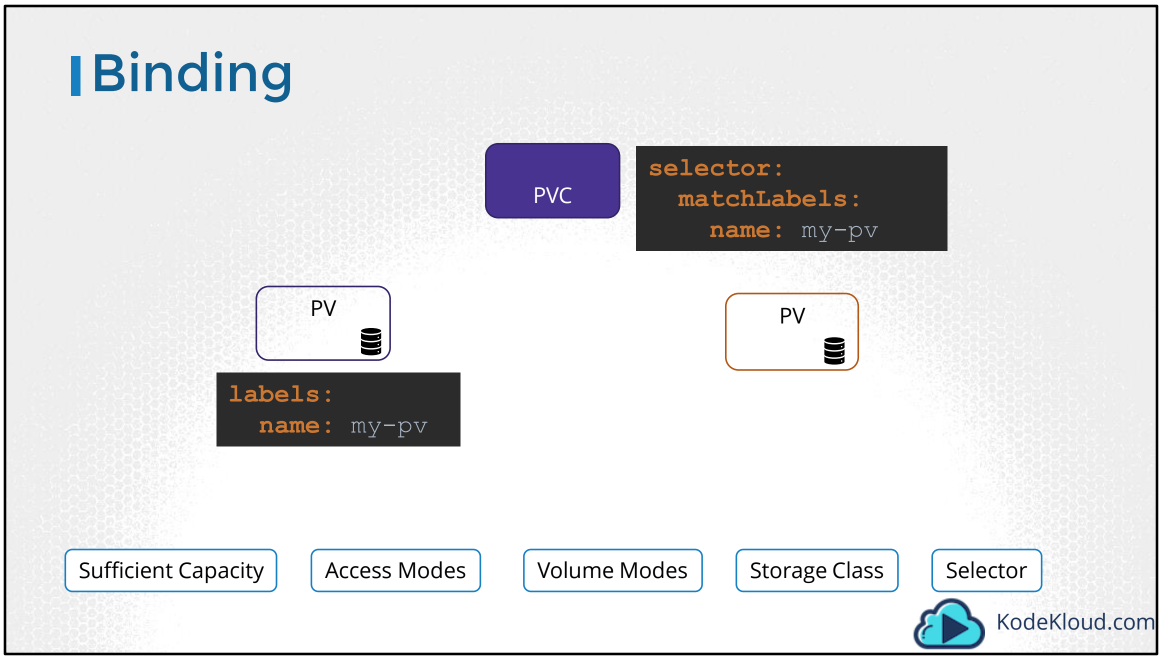
Binding

Once the Persistent Volume Claims are created, Kubernetes binds the Persistent Volumes to Claims based on the request and properties set on the volume.



Binding

Every Persistent Volume Claim is bound to a single Persistent volume. During the binding process, kubernetes tries to find a Persistent Volume that has sufficient Capacity as requested by the Claim, and any other requested properties such as Access Modes, Volume Modes, Storage Class etc.

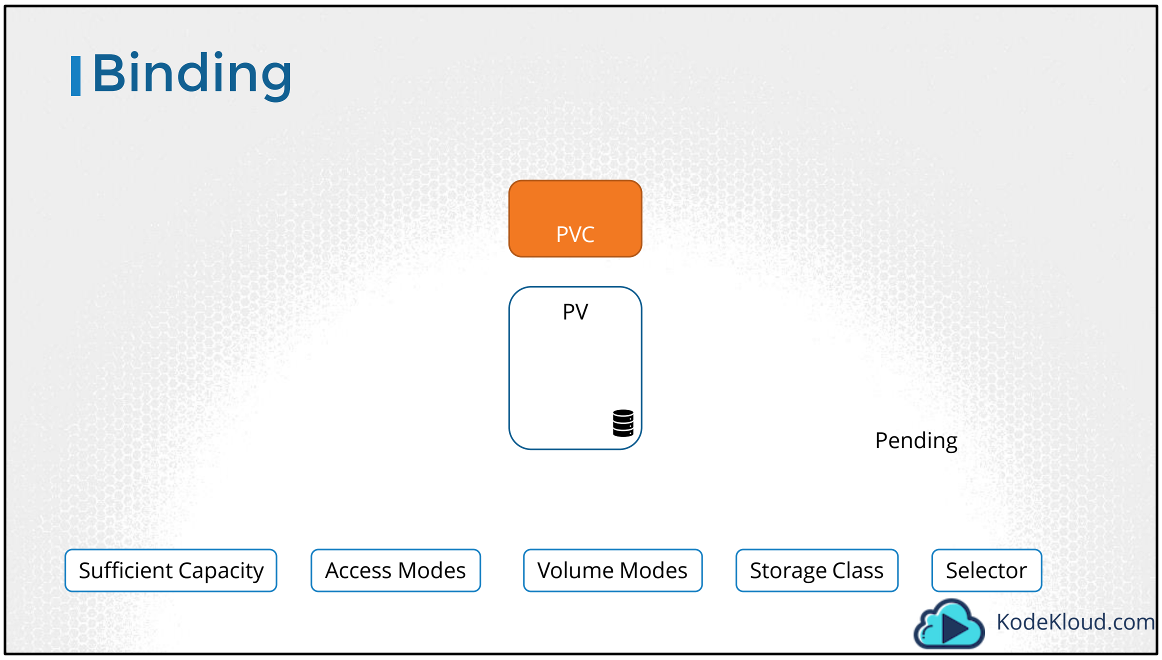


Binding

```
selector:
  matchLabels:
    name: my-pv
```

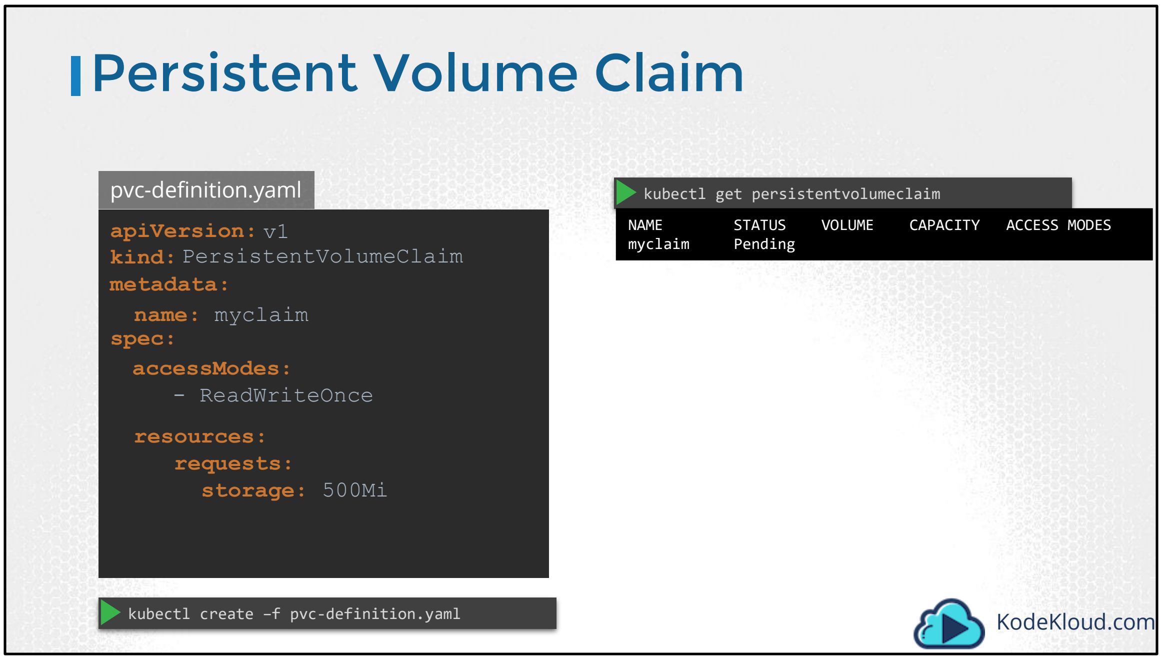
```
labels:
  name: my-pv
```

However, if there are multiple possible matches for a single claim, and you would like to specifically use a particular Volume, you could still use labels and selectors to bind to the right volumes.



Binding

Finally, note that a smaller Claim may get bound to a larger volume if all the other criteria matches and there are no better options. There is a one-to-one relationship between Claims and Volumes, so no other claim can utilize the remaining capacity in the volume. If there are no volumes available the Persistent Volume Claim will remain in a pending state, until newer volumes are made available to the cluster. Once newer volumes are available the claim would automatically be bound to the newly available volume.



Persistent Volume Claim

pvc-definition.yaml

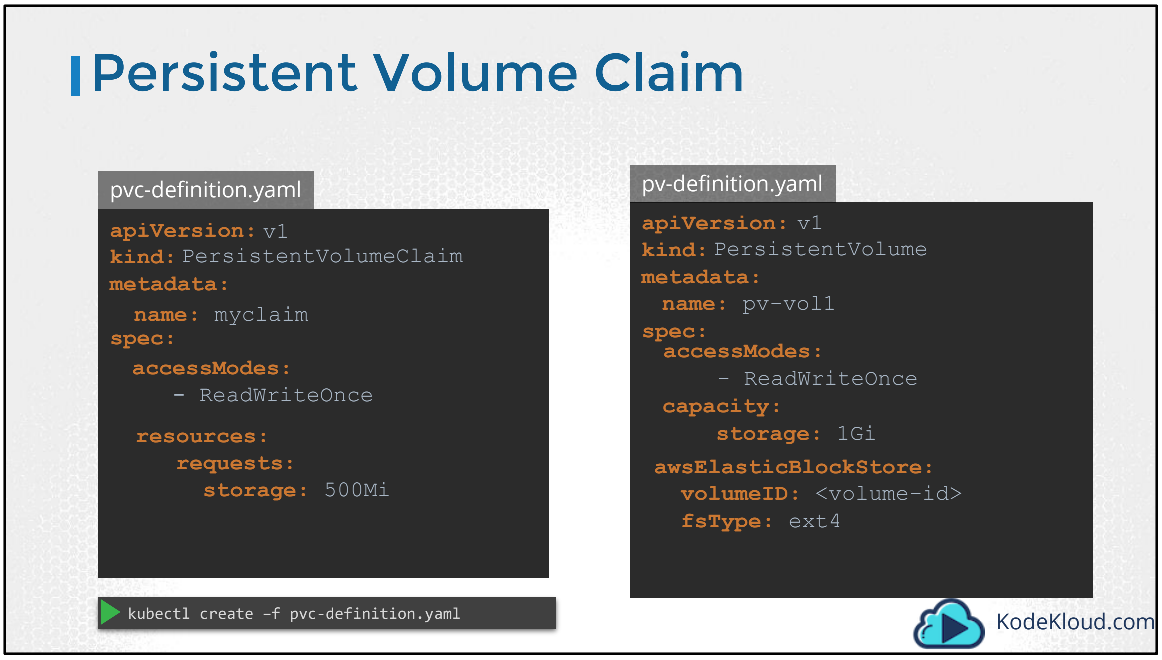
```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
     - ReadWriteOnce

  resources:
     requests:
       storage: 500Mi
```

```
▶ kubectl create –f pvc-definition.yaml
```

```
▶ kubectl get persistentvolumeclaim
NAME        STATUS     VOLUME      CAPACITY    ACCESS MODES
myclaim     Pending
```

KodeKloud.com

Let us now create a Persistent Volume Claim. We start with a blank template. Set the apiVersion to v1 and kind to PersistentVolumeClaim. We will name it myclaim. Under specification set the accessModes to ReadWriteOnce. And set resources to request a storage of 500 mega bytes. Create the claim using kubectl create command. To view the created claim run the kubectl get persistentvolumeclaim command. We see the claim in a pending state.



Persistent Volume Claim

pvc-definition.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
     - ReadWriteOnce

  resources:
     requests:
       storage: 500Mi
```

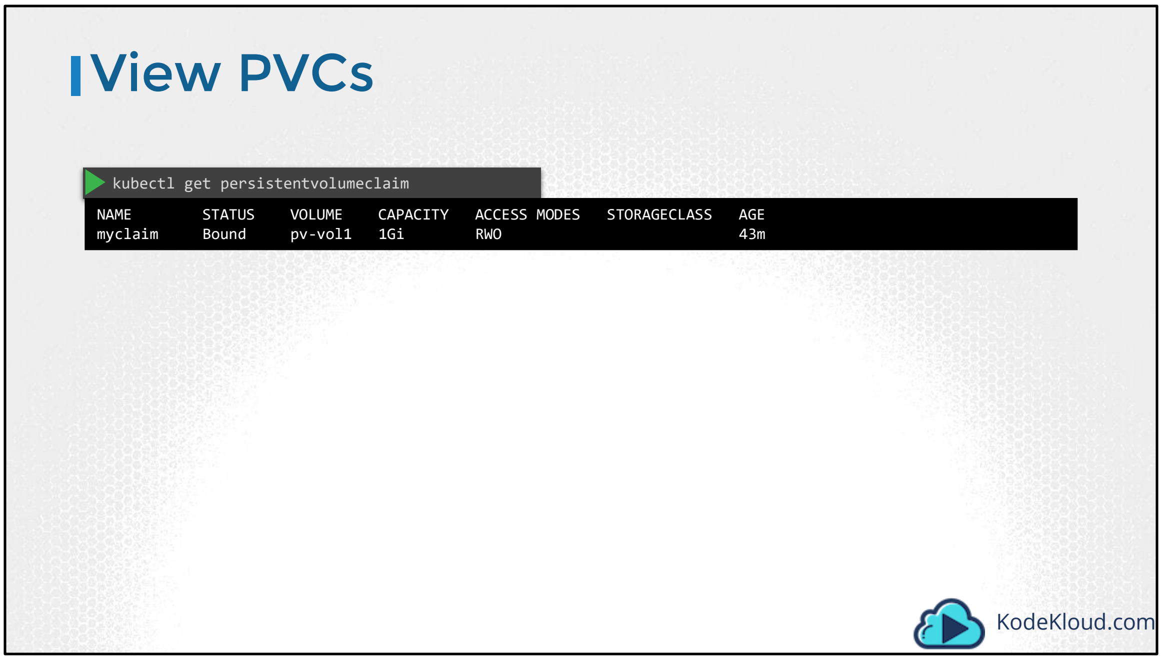
```
kubectl create -f pvc-definition.yaml
```

pv-definition.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-vol1
spec:
  accessModes:
       - ReadWriteOnce
  capacity:
       storage: 1Gi
  awsElasticBlockStore:
    volumeID: <volume-id>
    fsType: ext4
```

KodeKloud.com

When the claim is created, kubernetes looks at the volume created previously. The access Modes match. The capacity requested is 500 Megabytes but the volume is configured with 1 GB of storage. Since there are no other volumes available, the PVC is bound to the PV.

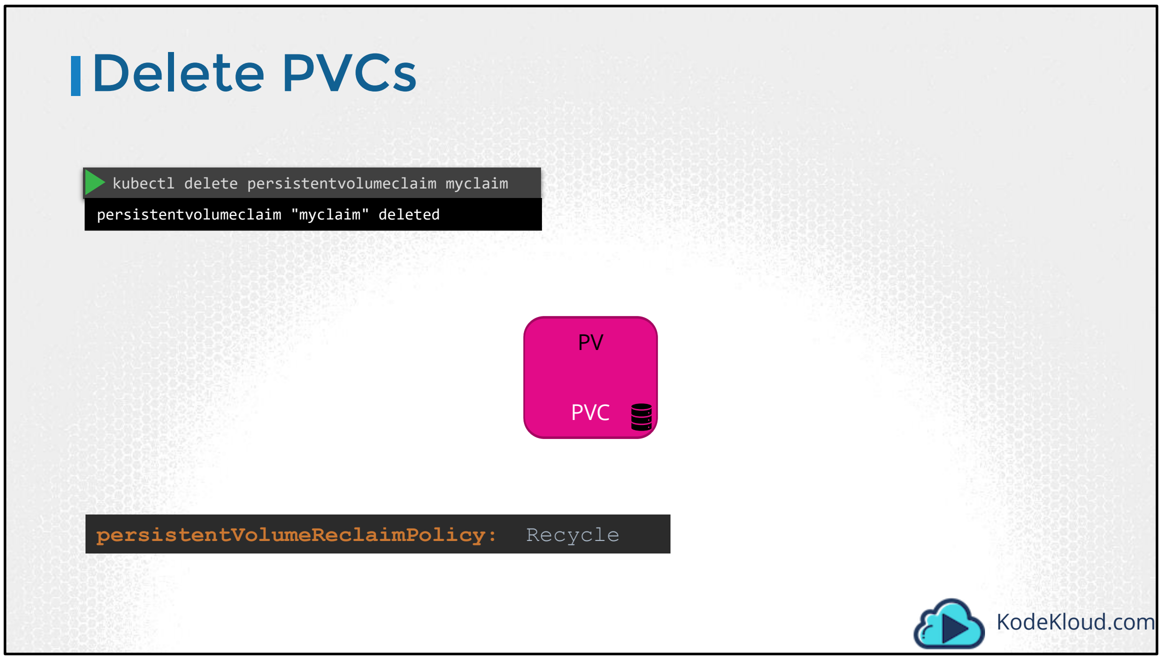


View PVCs

```
kubectl get persistentvolumeclaim
NAME        STATUS     VOLUME     CAPACITY    ACCESS MODES    STORAGECLASS    AGE
myclaim     Bound      pv-vol1    1Gi         RWO                             43m
```

KodeKloud.com

When we run the get volumes command again, we see the claim is bound to the persistent volume we created. Perfect!



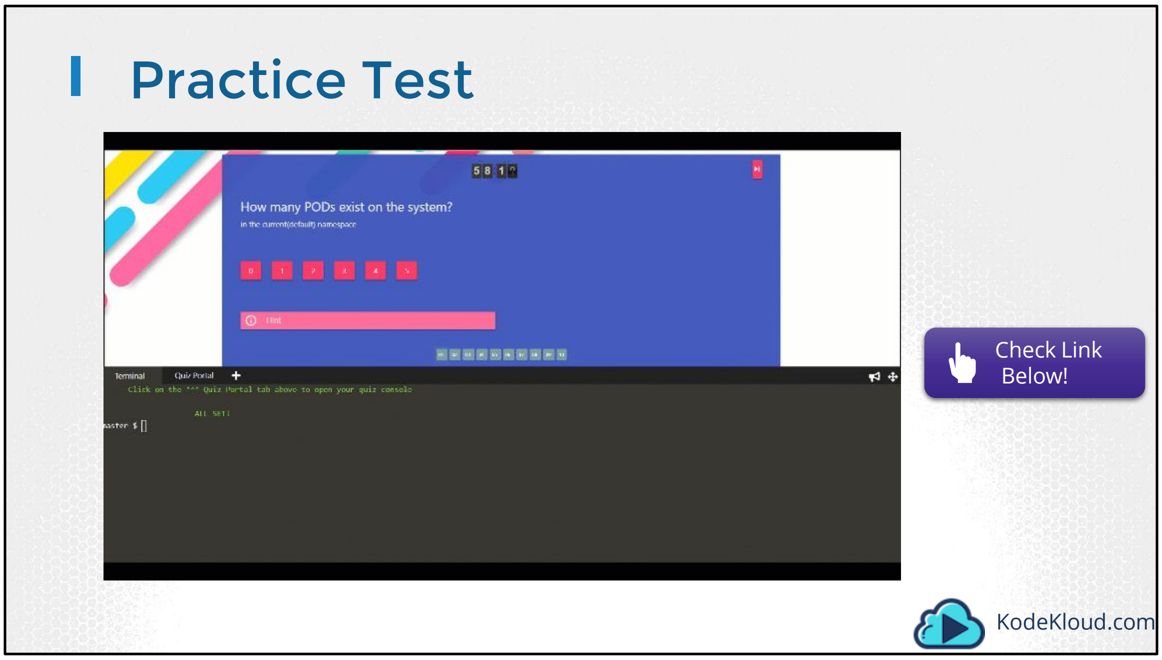
Delete PVCs

```
▶ kubectl delete persistentvolumeclaim myclaim
persistentvolumeclaim "myclaim" deleted
```

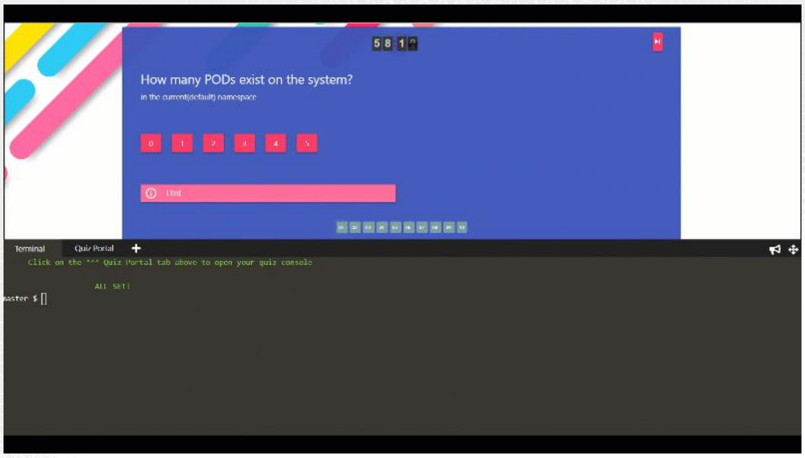
```
persistentVolumeReclaimPolicy:   Recycle
```

To delete a PVC run the kubectl delete persistentvolumeclaim command. But happens to the Underlying Persistent Volume when the claim is deleted? You can chose what is to happen to the volume. By default, it is set to Retain. Meaning the Persistent Volume will remain until it is manually deleted by the administrator. It is not available for re-use by any other claims. Or it can be Deleted automatically. This way as soon as the claim is deleted, the volume will be deleted as well. Or a third option is to recycle. In this case the data in the volume will be scrubbed before making it available to other claims.

Well that's it for this lecture. Head over to the coding exercises section and practice configuring and troubleshooting persistent volumes and volume claims in Kubernetes.



Practice Test



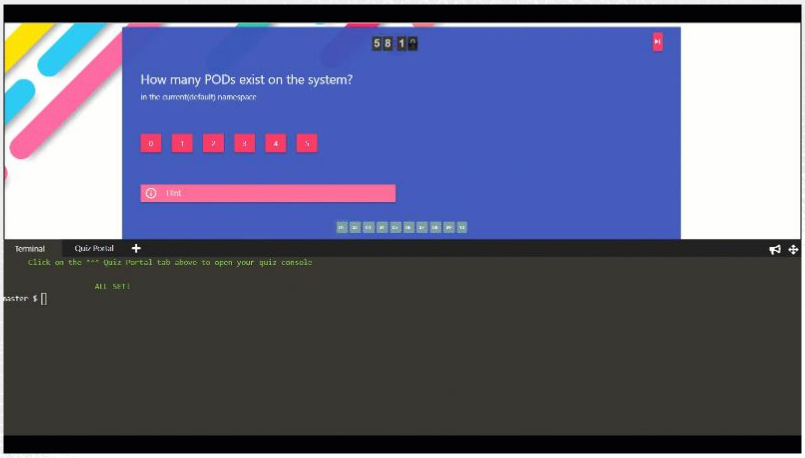
How many PODs exist on the system?

in the current(default) namespace

```
   Click on the ^^^ Quiz Portal tab above to open your quiz console

                ALL SET!
master $ 
```

Access Test Here: https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743707

Challenges



How many PODs exist on the system?

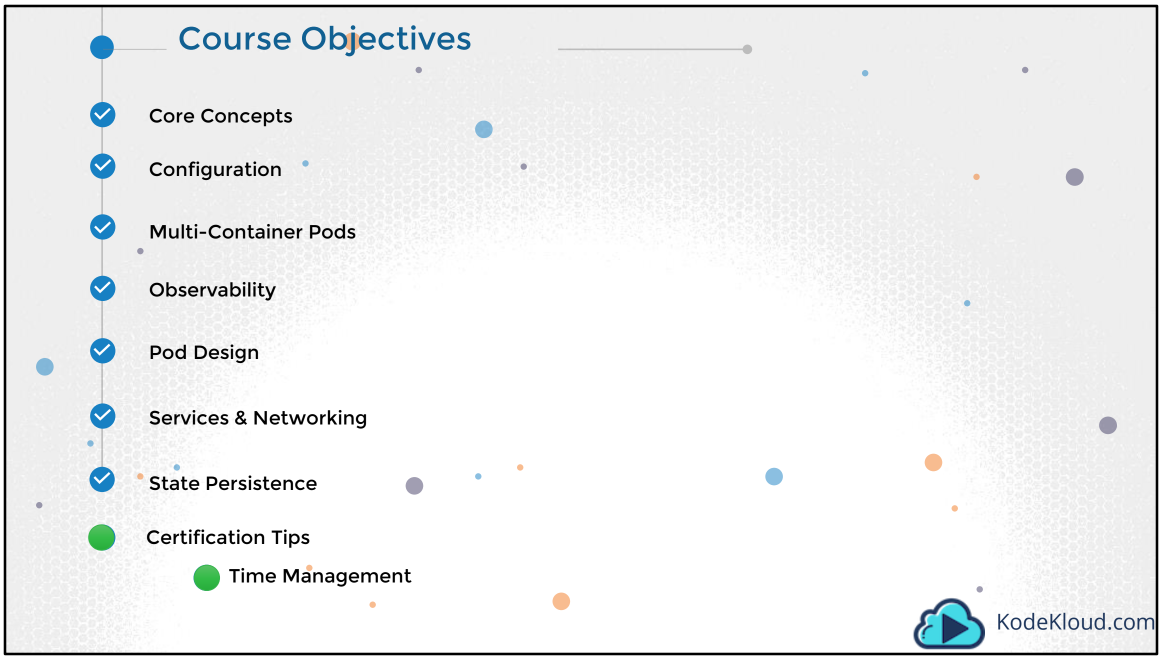
in the current(default) namespace

```
    Click on the ^^^ Quiz Portal tab above to open your quiz console

                   ALL SET!
master $ []
```

Check Link Below!

KodeKloud.com

Access the test here https://kodekloud.com/courses/kubernetes-certification-course/lectures/8414630



Course Objectives

- ✓ Core Concepts
- ✓ Configuration
- ✓ Multi-Container Pods
- ✓ Observability
- ✓ Pod Design
- ✓ Services & Networking
- ✓ State Persistence
- ● Certification Tips
  - ● Time Management

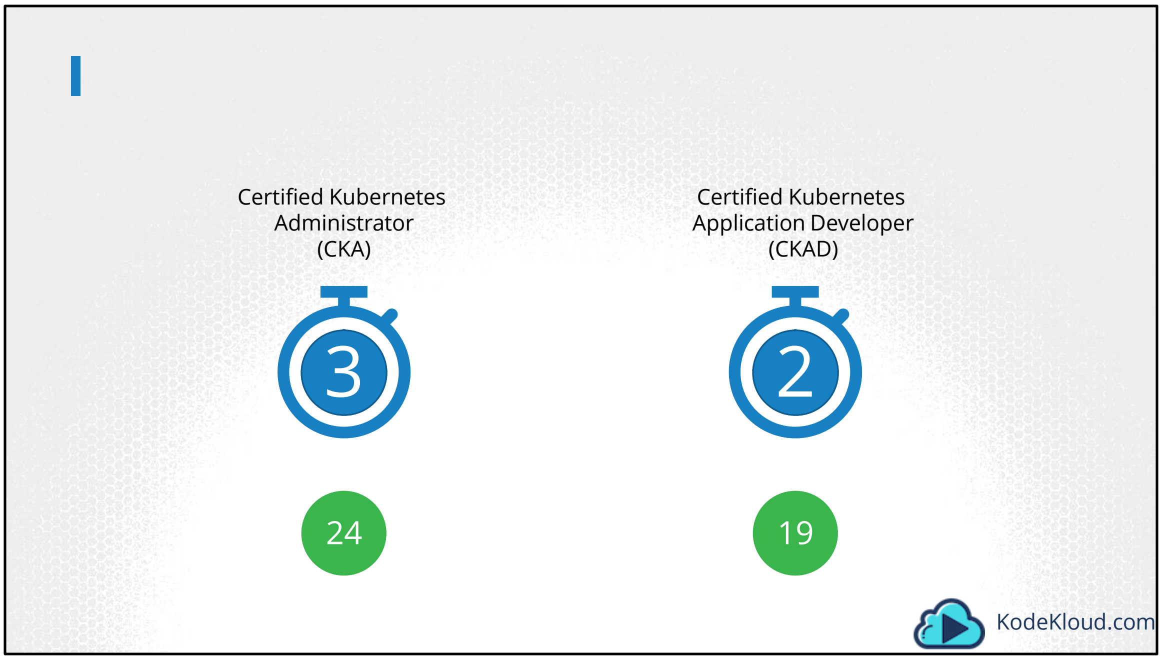
KodeKloud.com

Hello there!.

TIME MANAGEMENT

373
KodeKloud.com

In this lecture we will discuss , how to effectively manage your time during the certification exam. And this is applicable to all practical exams of this kind.



Certified Kubernetes Administrator (CKA)

3

24

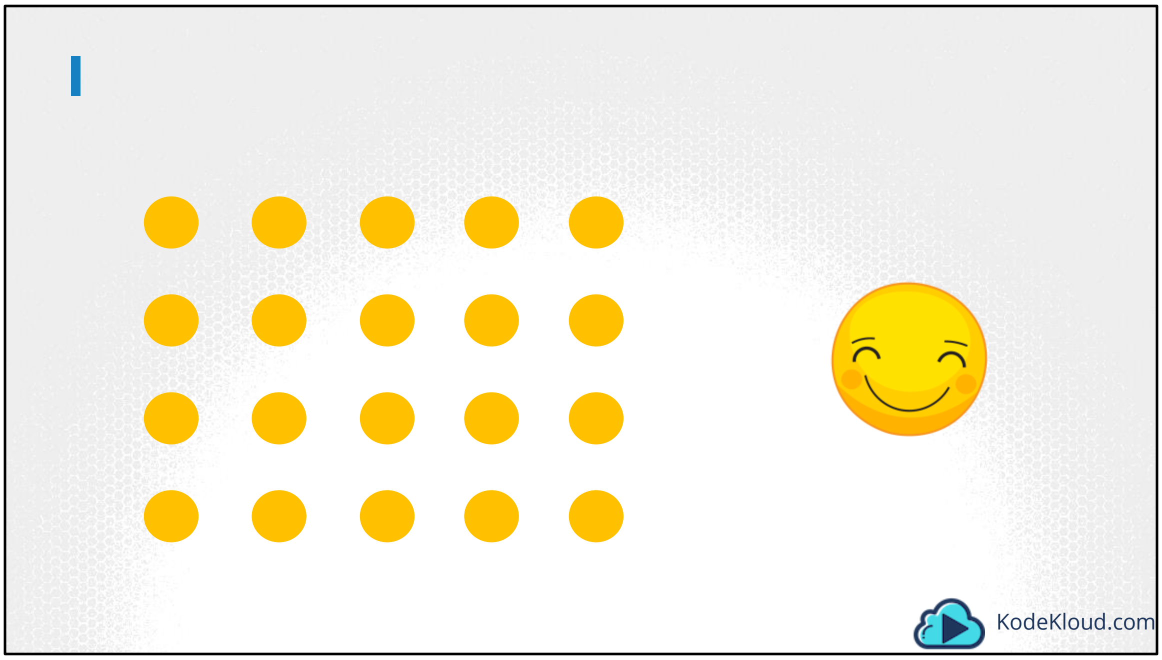
Certified Kubernetes Application Developer (CKAD)

2

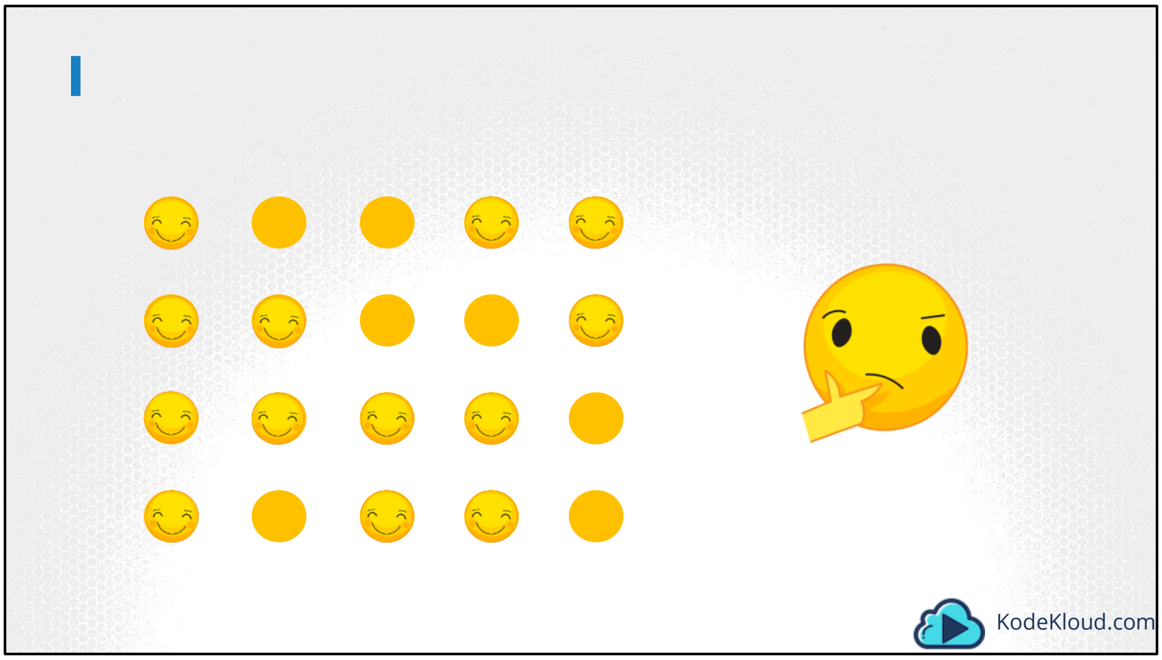
19

KodeKloud.com

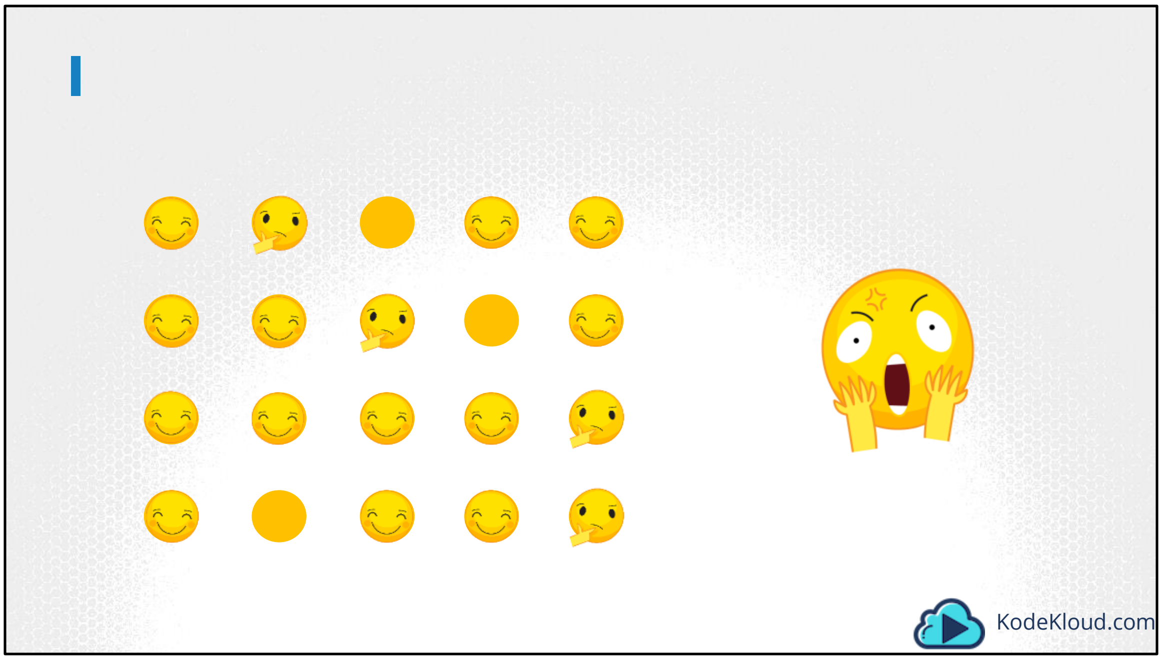
As of today, you get from 2 hours to 2.5 hours to complete the kubernetes certification exams. The duration of the administrators exam is 2.5 hours and the application developers exam is 2 hours. Now that is not sufficient to complete all the questions, so it is important to manage your time effectively to pass the exams.



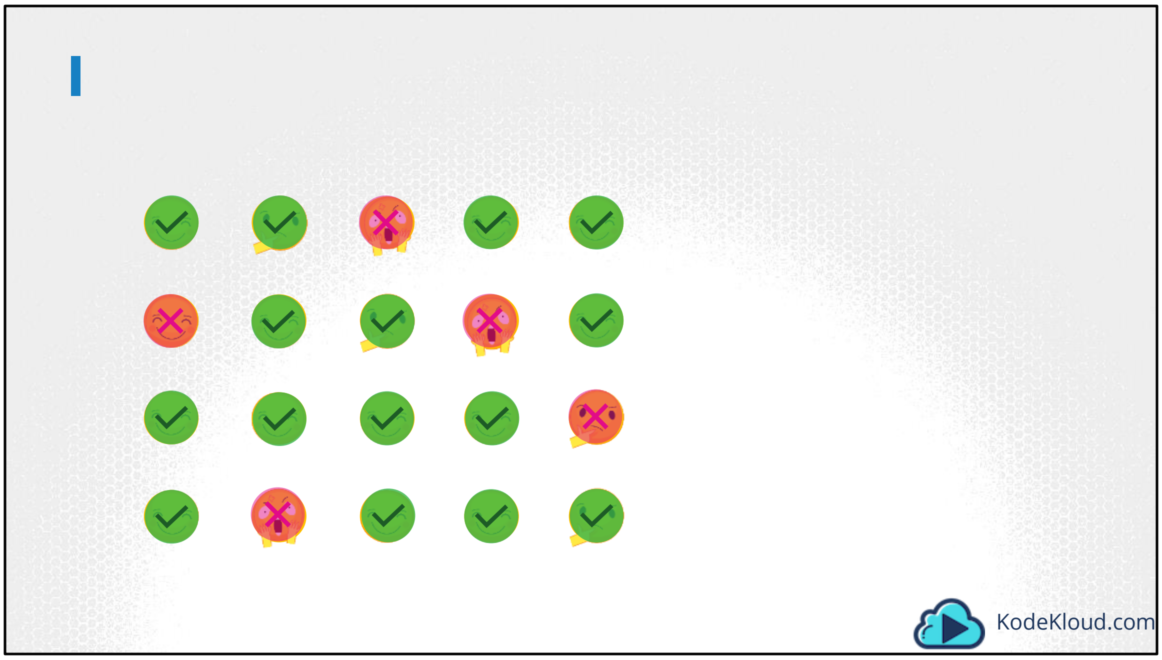
During the exam, you are presented with a set of questions. Some of which may be very easy…



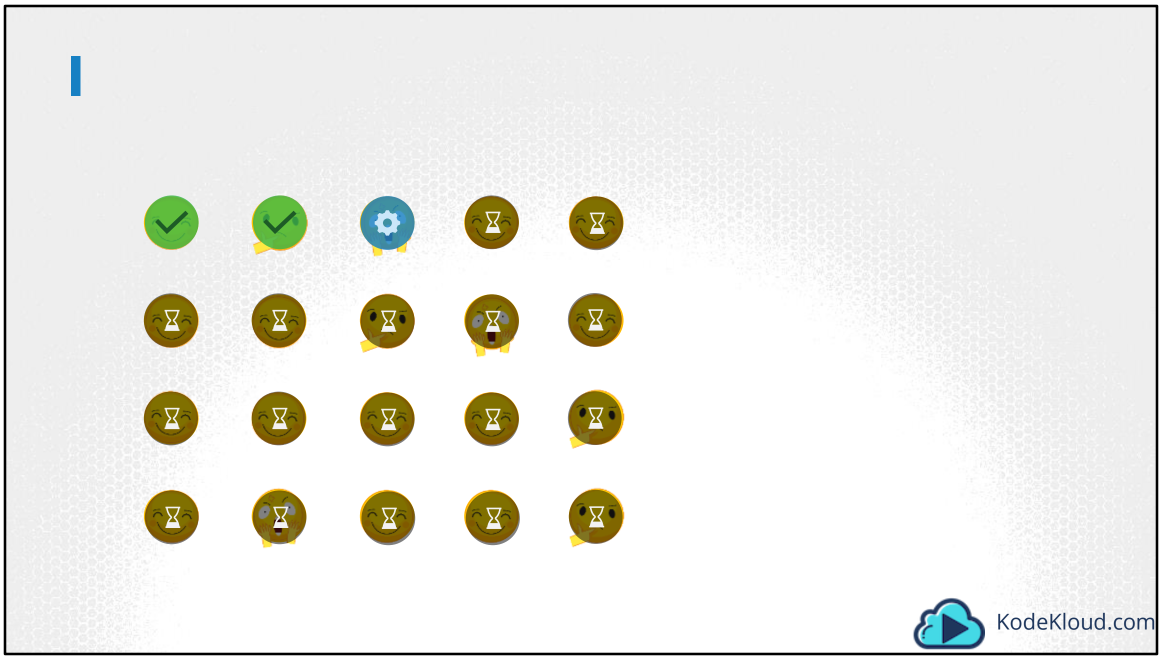
some that makes you think a little bit….



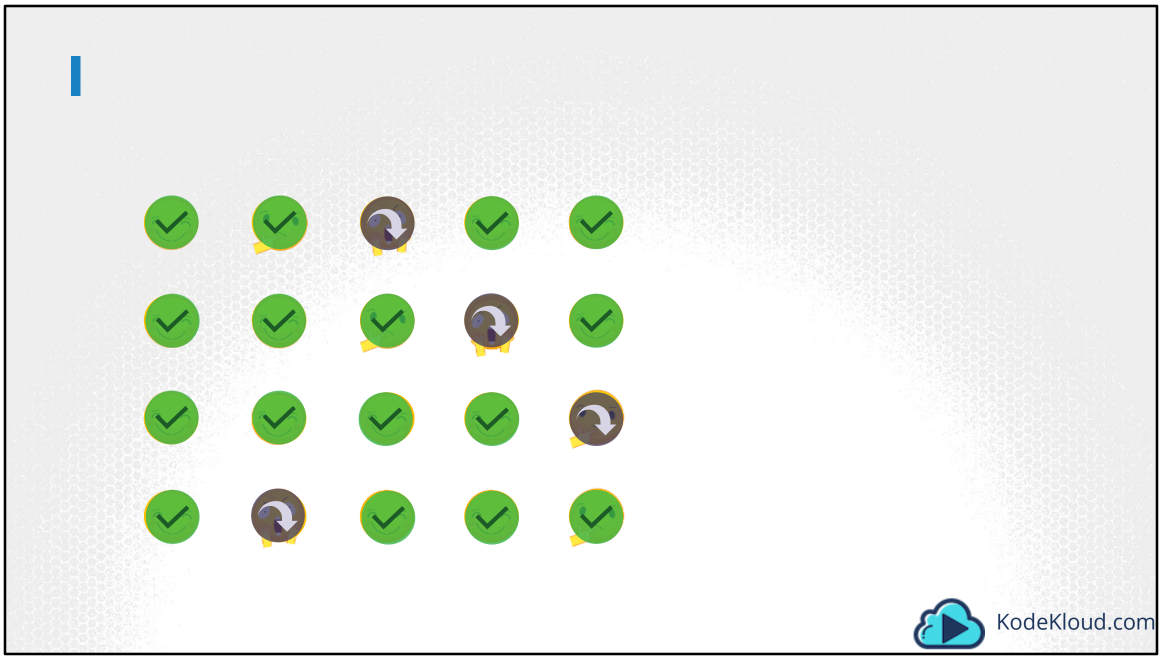
and some that you have no clue about, hopefully not too many of that. And they are not there in any particular order. You may have easy or tough questions in the beginning or towards the end.



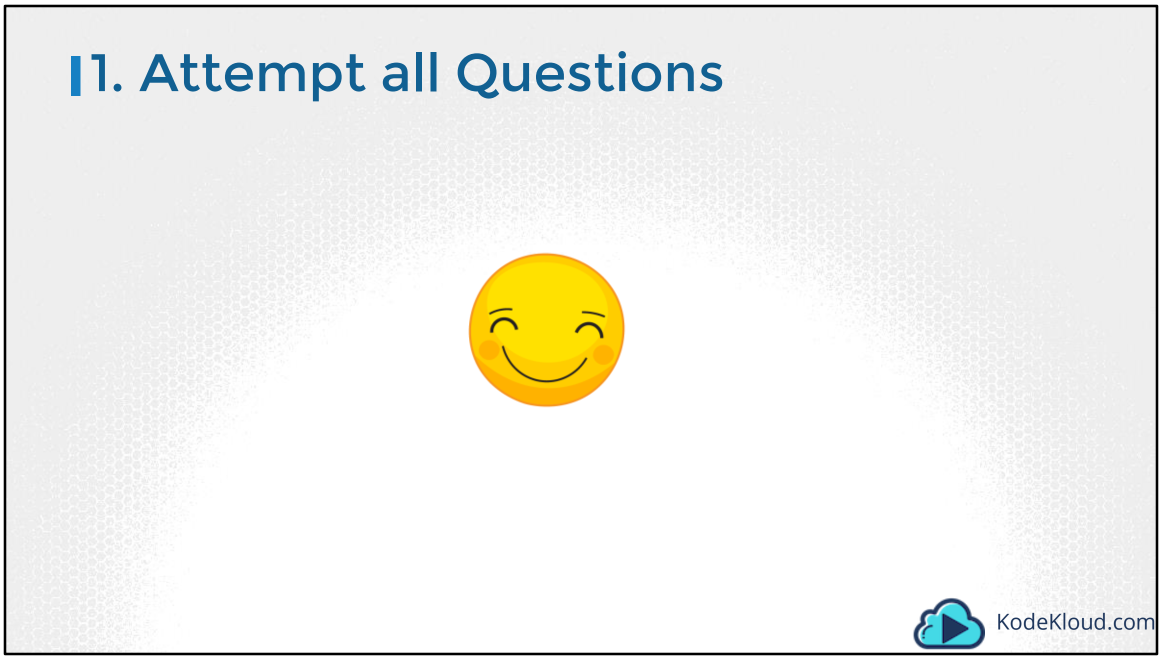
Now you don't have to get all of it right. You only need to solve enough to gain the minimum required percentage to pass the exam. So it is very important to attempt all of the questions.



You don’t want to get stuck in any of the early tough questions, and not have enough time to attempt the easy ones that come after.



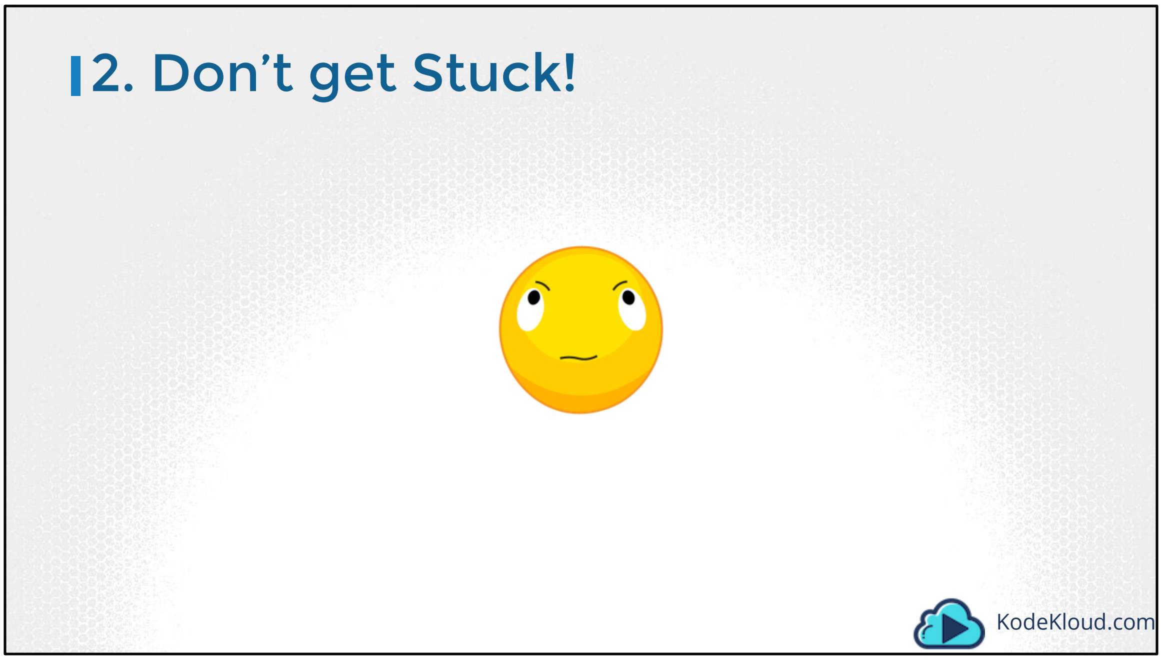
You have the option to attempt the questions in any order you like. So you could skip the tough ones and chose to attempt all the easy ones first. Once you are done, if you still have time, you can go back and attempt the ones you skipped.



1. Attempt all Questions

KodeKloud.com

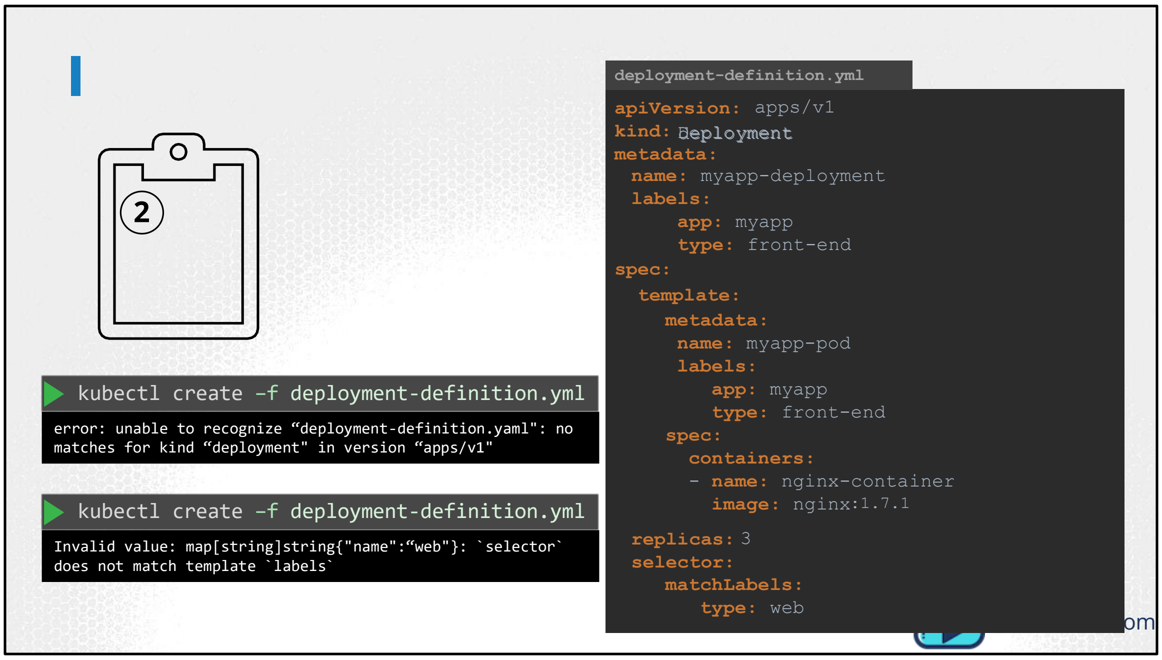
So that was the first and most important tip, attempt all the questions.



2. Don’t get Stuck!

KodeKloud.com

The second tip is not get stuck on any question. Even for a simple one.



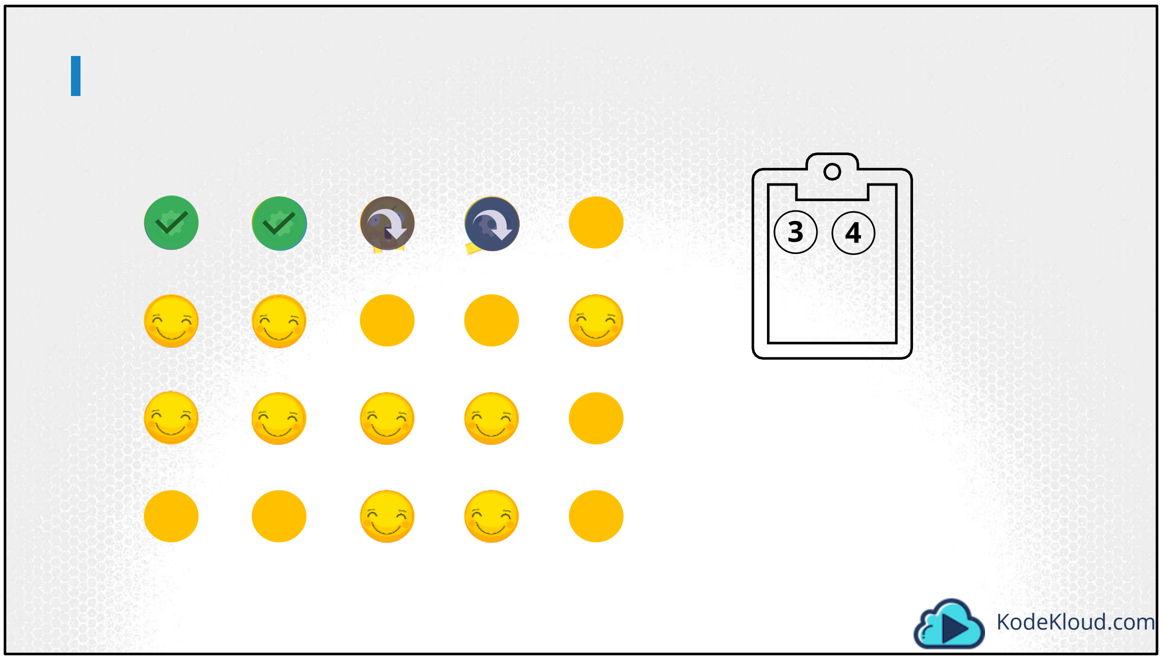
```
deployment-definition.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
     name: myapp-pod
     labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.7.1

  replicas: 3
  selector:
    matchLabels:
       type: web
```

```
kubectl create -f deployment-definition.yml
error: unable to recognize “deployment-definition.yaml": no
matches for kind “deployment" in version “apps/v1"
```

```
kubectl create -f deployment-definition.yml
Invalid value: map[string]string{"name":“web"}: `selector`
does not match template `labels`
```

For example you are attempting to solve a question that looks simple. You know what you are doing, so you make an attempt. The first time you try to execute your work, it fails. You read the error message and realize that you had made a mistake, like a typo. So you go back and fix it and run it again. This time you get an error message, but you are not able to make any sense out of it. Even though that was an easy question, and you knew you could do it, if you are not able to make any sense out of the error message, don’t spend any more time troubleshooting or debugging that error. Mark that question for review later, skip it and move on to the next.

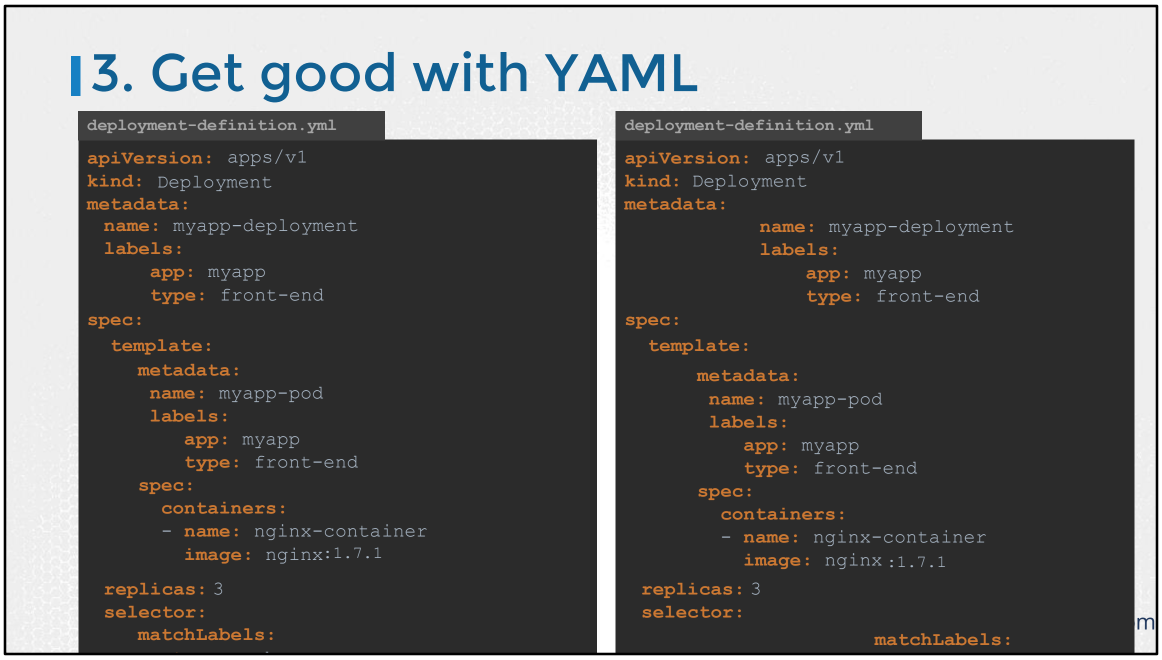
Now, I KNOW that urge to troubleshoot and fix issues. But this is not the time for it. Leave it to the end and do all the troubleshooting you want after you have attempted all the questions.



So here is how I would go about it.   Start with the first question. If it looks easy, attempt it. Once you solve it, move over to the next. If that looks easy, attempt it. Once that is finished, go over to the next. If that looks hard, and you think you will need to read up on it, mark it down and go over to the next.

The next one looks a bit difficult, but you think you can figure it out. So give it a try. First attempt it fails, you know what the issue is so you try to fix it. The second attempt, it fails again and you don't know what the issue is. Don't spend any more time on it, as there are many easy questions waiting ahead. Mark it down, for review later and go over to the next.

Follow the same technique to finish as many questions as possible.



3. Get good with YAML

deployment-definition.yml

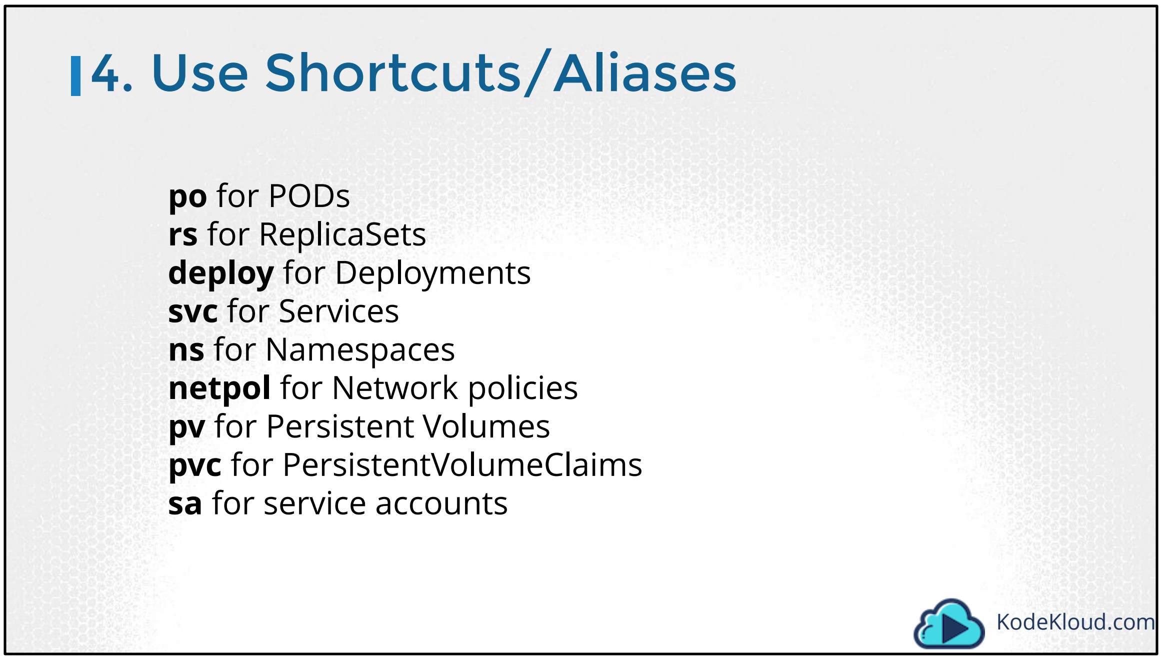
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
      app: myapp
      type: front-end
spec:
  template:
     metadata:
      name: myapp-pod
      labels:
         app: myapp
         type: front-end
     spec:
       containers:
       - name: nginx-container
         image: nginx:1.7.1

  replicas: 3
  selector:
     matchLabels:
```

deployment-definition.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
            name: myapp-deployment
            labels:
                app: myapp
                type: front-end
spec:
  template:

      metadata:
       name: myapp-pod
       labels:
          app: myapp
          type: front-end
      spec:
        containers:
        - name: nginx-container
          image: nginx :1.7.1
  replicas: 3
  selector:
                     matchLabels:
```

The third tip, is to be really good with YAML. You must spend enough time practicing your definition files in advance. If, for each question, you are having to go through each line of your YAML file and fix the indentation errors, you are not going to be able to make it through all questions. Your YAML files don't have to look pretty. Because nobody is going to look at them. I am guessing that the work is evaluated automatically, so only the end result is evaluated and not how pretty your YAML files are.

So even if your file looks like this one on the right, where as it should have looked like the one on the left, it's still fine as long as the structure of the file is correct. And that you have the right information in the file and are able to create the required kubernetes object using the file. For that you need to get your YAML basics right. If you are a beginner, check out the coding exercises at KodeKloud that helps you practice and get good with YAML.



4. Use Shortcuts/Aliases

po for PODs
rs for ReplicaSets
deploy for Deployments
svc for Services
ns for Namespaces
netpol for Network policies
pv for Persistent Volumes
pvc for PersistentVolumeClaims
sa for service accounts

KodeKloud.com

Time Management

KodeKloud.com

Well that’s it for this lecture. If you have any more tips, please feel free to leave a comment below. Thank you for listening and I wish you good luck with your exams.

THANK YOU!

mmumshad@gmail.com