



Mumshad Mannambeth  
Solutions Architect | Trainer | 55,000 Students | 10,227 Reviews | 4.5 Star Rated

**BEST SELLER** Docker for the Absolute Beginner - Hands On ...  
Mumshad Mannambeth  
★★★★★ 4.5 (2,380)

**HIGHEST RATED** Kubernetes for the Absolute Beginners -...  
Mumshad Mannambeth  
★★★★★ 4.5 (991)

**BEST SELLER** OpenShift for the Absolute Beginners -...  
Mumshad Mannambeth  
★★★★★ 4.4 (397)

**BEST SELLER** Puppet for the Absolute Beginners - Hands-on ...  
Mumshad Mannambeth, Yogesh...  
★★★★★ 4.4 (269)

**BEST SELLER** Ansible for the Absolute Beginner - Hands-On ...  
Mumshad Mannambeth  
★★★★★ 4.5 (2,583)

**BEST SELLER** Chef for the Absolute Beginners - DevOps  
Mumshad Mannambeth, Yogesh...  
★★★★★ 4.2 (52)

KodeKloud.com

Hello and welcome to this course on the Certified Kubernetes Applications Developer. My name is Mumshad Mannambeth and I will be your instructor for this course. So about me, I am a Solutions Architect specializing on Cloud, Automation and DevOps Technologies. I have authored several Best Seller and Top Rated courses Docker, Kubernetes and OpenShift as well as automation technologies like Ansible, Chef and Puppet. This course is the second installment in the series on Kubernetes and focuses on a certification.



Let's take a look at the structure of this course. We start with a series of lectures on various topics in Kubernetes, where we simplify complex concepts using illustration and animation.

We have optional quizzes that test your knowledge after each lecture.

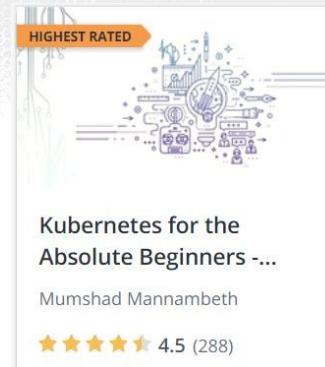
We then have coding quizzes that will help you practice what you learned on a real live environment right in your browser. The kubernetes certification is hands-on, so the coding exercises will give you enough experience and practice on getting ready for it. More on this in the upcoming lectures.

We will also discuss some tips & tricks to crack the certification exam.

And as always if you have any questions, you may reach out directly to us through our Q&A section.

## Pre-Requisites

- Lab Environment
- Kubernetes Architecture
- Master and Worker Nodes
- Pods, ReplicaSets, Deployments
- Command Line - kubectl
- Understanding YAML
- Services
- Namespaces



KodeKloud.com

Now, this is one of the course in the series on Kubernetes and focuses on getting the Kubernetes Applications Developer Certification. So a basic understanding is required. For example, you must know how to set up a lab environment to practice on. The certification curriculum does not include kubernetes setup or install, so you could setup a learning environment anyway. We discuss a lot of these in the beginners course. You also need a good understanding of YAML language for creating configuration files in kubernetes and a basic understanding of what master and worker nodes are, and what Pods, ReplicaSets and Deployments are. We do refresh some of these topics in this course, but if you are an absolute beginner I highly recommend taking my Kubernetes for the Absolute Beginners course.

## Course Objectives

- Core Concepts
- Configuration
- Multi-Container Pods
- Observability
- Pod Design
- Services & Networking
- State Persistence



KodeKloud.com

Let us now look at the Course Objectives. The objectives of this course are aligned to match the Certified Kubernetes Application Developer exam Curriculum. We will discuss about details around the Certification itself in one of the upcoming lectures before heading into any of these topics.

## Course Objectives

- Core Concepts
  - Kubernetes Architecture
  - Create and Configure Pods
- Configuration
- Multi-Container Pods
- Observability
- Pod Design
- Services & Networking
- State Persistence



KodeKloud.com

We start with the core concepts – we have covered a lot of the core concepts in the beginners course. We will however quickly recap some of these in this course to refresh our memory. Such as the kubernetes Architecture, what PODs are and how to create and configure PODs etc.

## Course Objectives

- Core Concepts
- Configuration
  - ConfigMaps
  - SecurityContexts
  - Resource Requirements
  - Secrets
  - ServiceAccounts
- Multi-Container Pods
- Observability
- Pod Design
- Services & Networking
- State Persistence



The next section is on Configuration and covers topics like ConfigMaps, SecurityContexts, Resource Requirements, secrets and service accounts.

## Course Objectives

- Core Concepts
- Configuration
- Multi-Container Pods
  - Ambassador
  - Adapter
  - Sidecar
- Observability
- Pod Design
- Services & Networking
- State Persistence



KodeKloud.com

We will then look deeper into Multi-Container Pods. The different patterns of multi-container pods such as Ambassador, Adapter and Sidecar. We will look at some examples and use cases around these.

## Course Objectives

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



KodeKloud.com

We then learn about Readiness and Liveness Probes and why you need them. We will also look at some of the Monitoring, Logging and Debugging options available with Kubernetes specifically around Pods, containers and applications.

## 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.

## Course Objectives

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



KodeKloud.com

We will then learn about Services and Network Policies.

## Course Objectives

- Core Concepts
- Configuration
- Multi-Container Pods
- Observability
- Pod Design
- Services & Networking
- State Persistence

 Persistent Volumes  
 Persistent Volume Claims



KodeKloud.com

And finally we look at Persistent Volumes and Claims. For all of these topics, we have lectures that makes these complex topics easy to understand. Followed by coding challenges where you will be practicing what you learned on a real environment. Let's take a look at that.

## Practical Exercises



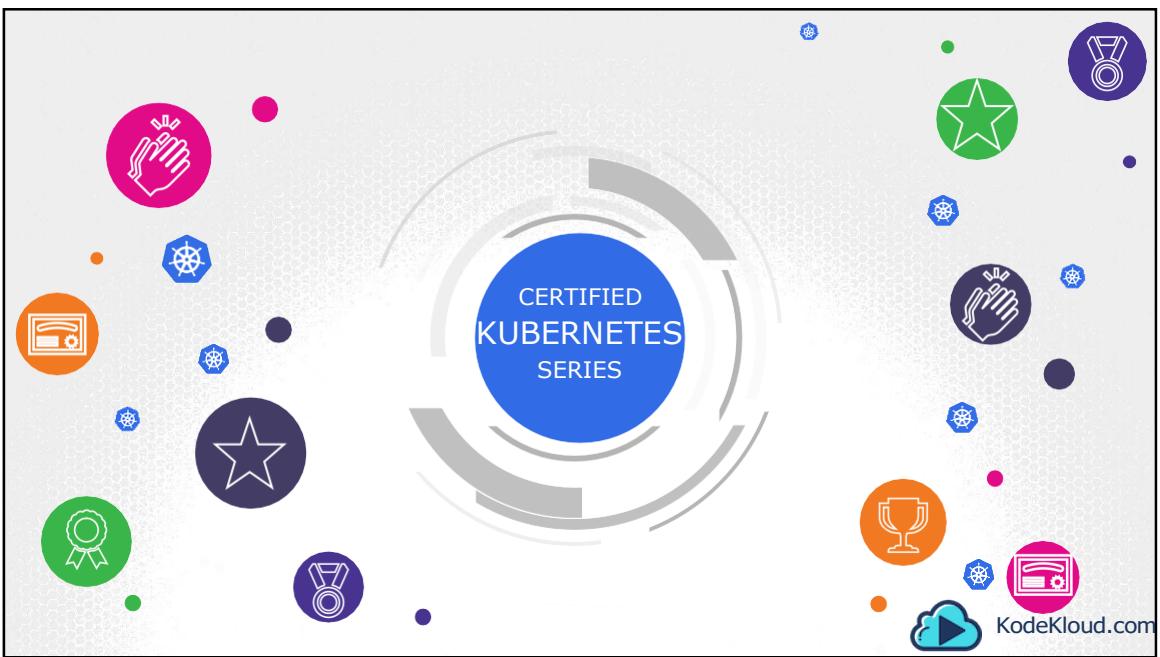
Check Links  
Below

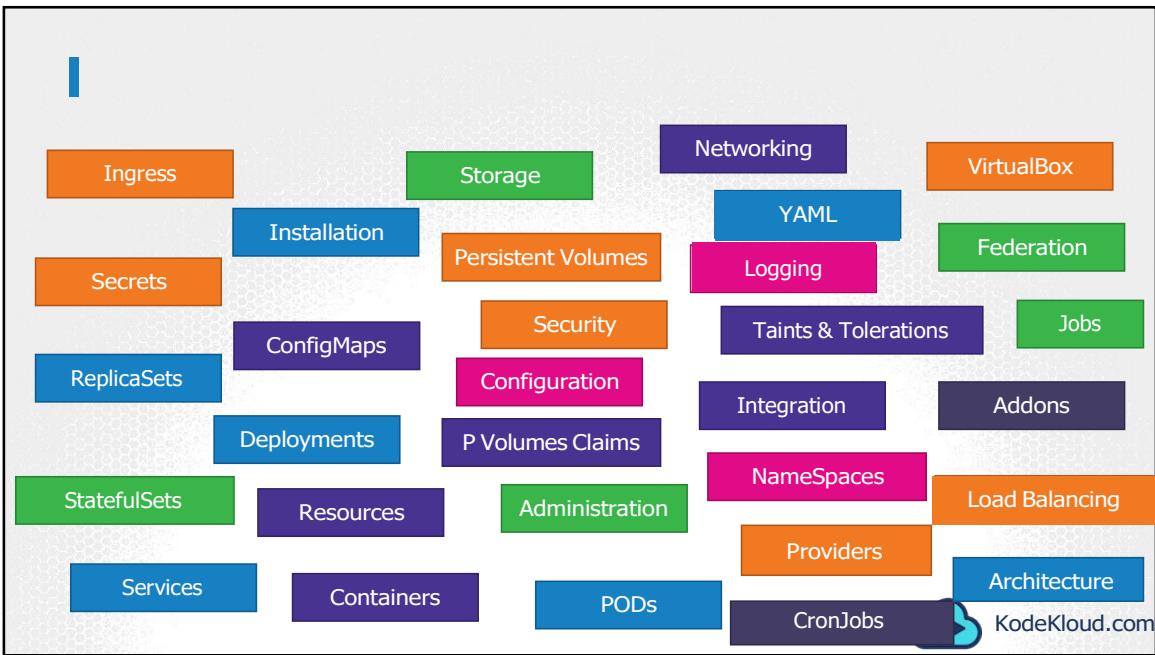
Try Here!

KodeKloud.com

Watch it here: <https://kodekloud.com/courses/kubernetes-certification-course/lectures/6731376>

Access Test Here: <https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743640>





Hello there. Before we begin, I want to spend a minute on the Kubernetes Series of courses. Kubernetes is one of the most trending technology in cloud computing as of today. It is supported on any cloud platform and supports hosting enhanced and complex applications on various kinds of architectures that makes it a vast and complex technology. There are a set of pre-requisite knowledge required such as containers, applications, YAML files etc. A lot of topics to discuss, a lot of concepts to cover such as the Architecture, Networking, Load Balancing, a variety of monitoring tools, Auto Scaling, Configuration, Security, Storage etc.

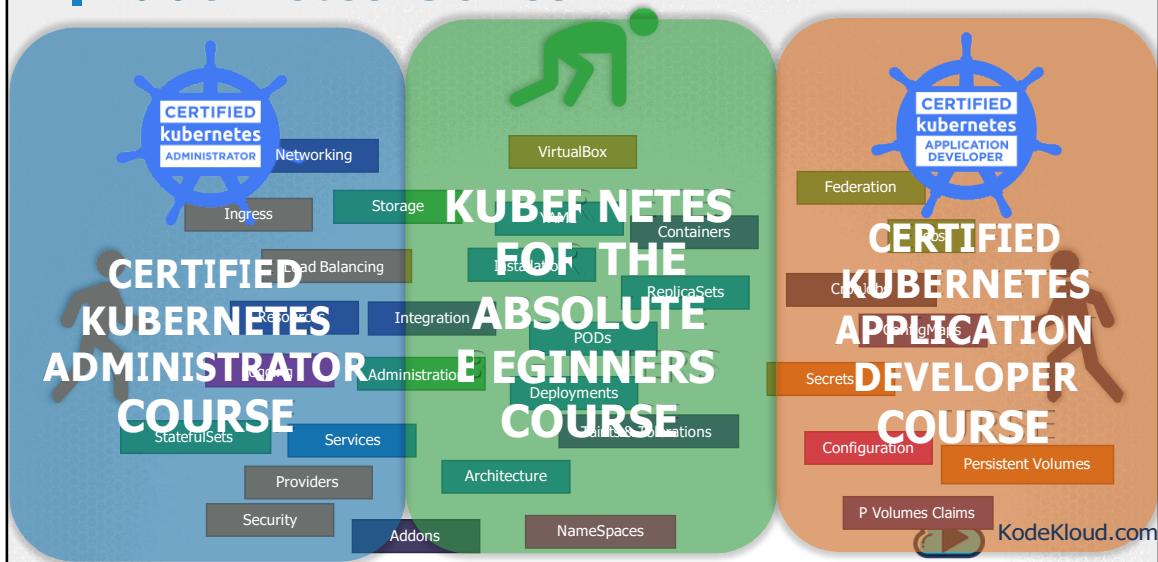


There are students from different backgrounds, such as the Absolute Beginners to Kubernetes, or those with some experience looking for specialized knowledge in Administration or those from a development background looking for concepts specific to Application Development on Kubernetes.



And there are two Certifications in the mix as well. One for Administrators and another for Application Developers. Covering all of these topics for all of these students in a single course is an impossible task.

## I Kubernetes Series



Which is why we created a 3 course series, so each course can target specific audience, topics and certifications. The Kubernetes for the Absolute Beginners course, the Certified Kubernetes Administrator's course and the certified kubernetes application developer's course.



KUBERNETES for the Absolute Beginners



KUBERNETES for Administrators



KUBERNETES for Developers

KodeKloud.com



Let's look at what we cover in each of these courses.



#### KUBERNETES for the Absolute Beginners

- |   |   |
|---|---|
| <input type="radio"/> Lab Environment   | <input type="radio"/> Pre-requisites - YAML       |
| <input type="radio"/> PODs, Deployments | <input type="radio"/> Networking Basics           |
| <input type="radio"/> Services          | <input type="radio"/> Micro-Services Architecture |
| <input type="radio"/> Demos             | <input type="radio"/> Coding Exercises            |



#### KUBERNETES for Administrators



#### KUBERNETES for Developers



KodeKloud.com

The Kubernetes for the Absolute Beginners course helps a beginner having no prior experience with containers or container orchestration get started with the concepts of Kubernetes. As this is a beginners course, we do not dive deep into technical details, instead we focus on a high level overview of Kubernetes, setting up a simple lab environment to play with Kubernetes, learning the pre-requisites required to understand and get started with Kubernetes, understanding the various concepts to deploy an application such as PODs, replica-sets, deployments and services. This course is also suitable for a non-technical person trying to understand the basic concepts of Kubernetes, just enough to get involved in discussions around the technology.



KUBERNETES for the Absolute Beginners

### KUBERNETES for Administrators

<input type="radio"/> HA Deployment	<input type="radio"/> Kubernetes Scheduler
<input type="radio"/> Logging/Monitoring	<input type="radio"/> Application Lifecycle
<input type="radio"/> Maintenance	<input type="radio"/> Security
<input type="radio"/> Troubleshooting	<input type="radio"/> Core Concepts
<input type="radio"/> Demos	<input type="radio"/> Coding Exercises

 Certified Kubernetes Administrator (CKA)



KUBERNETES for Developers



KodeKloud.com

The Kubernetes for Administrators course focuses on advanced topics on Kubernetes and in-depth discussions into the various concepts around Deploying a high-availability cluster for production use cases. Understanding more about scheduling, monitoring, maintenance, security, storage and troubleshooting. This course also helps you prepare for the Certified Kubernetes Administrator Exam and get yourself certified as a Kubernetes Administrator.

KUBERNETES for the Absolute Beginners

KUBERNETES for Administrators

**KUBERNETES for Developers**

- Core Concepts
- ConfigMaps, Secrets & ServiceAccounts
- Multi-Container Pods
- Readiness & Liveness Probes
- Logging & Monitoring
- Pod Design
- Jobs
- Services & Networking
- Demos
- Coding Exercises

Certified Kubernetes Application Developer (CKAD)

KodeKloud.com

The Kubernetes for Developers course is for Application Developers who are looking to learn how to design, build and configure cloud native applications. Now you don't have to be an expert application developer for this course and there is no real coding or application development involved in either this course or the certification itself. You only need to know the real basics of development on a platform like python or NodeJs.

This course focuses on topics relevant for a developer such as ConfigMaps, Secrets & ServiceAccounts, Multi-container Pods, Readiness & Liveness Probes, Logging & Monitoring, Jobs, Services and Networking. This course will also help you prepare for the Certified Kubernetes Application Developer exam.

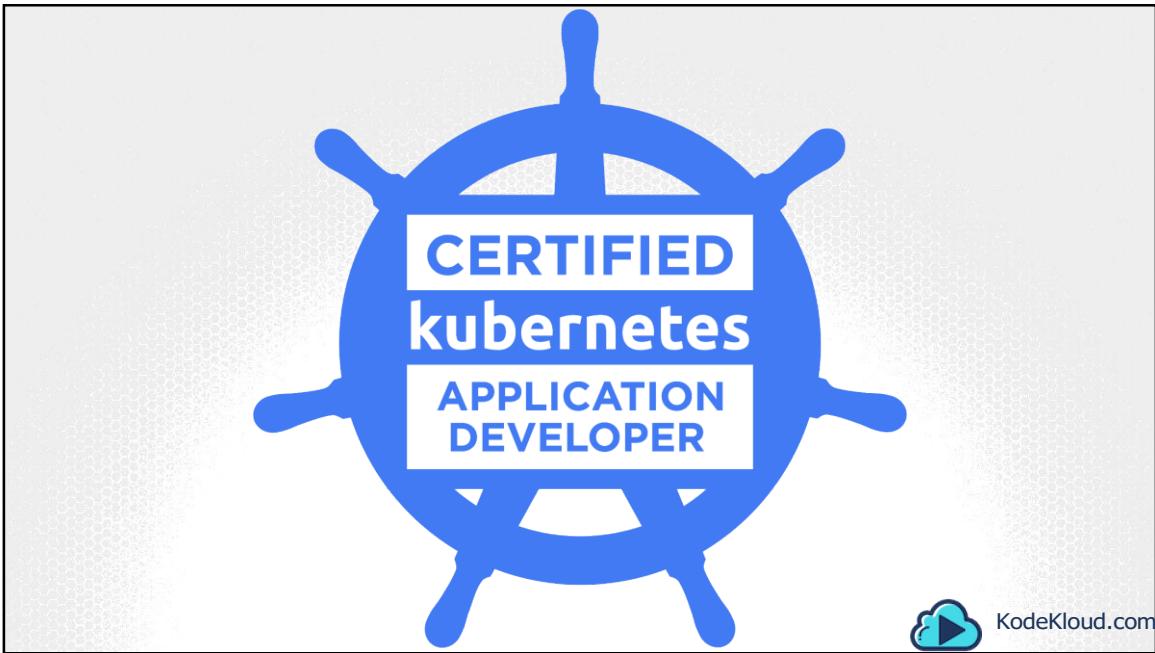
All of these courses are filled with Coding Exercises and quizzes that will help you PRACTICE developing and deploying applications on Kubernetes.

Now, remember that there are some topics that overlap between these courses. So we recap and discuss them as and when required.

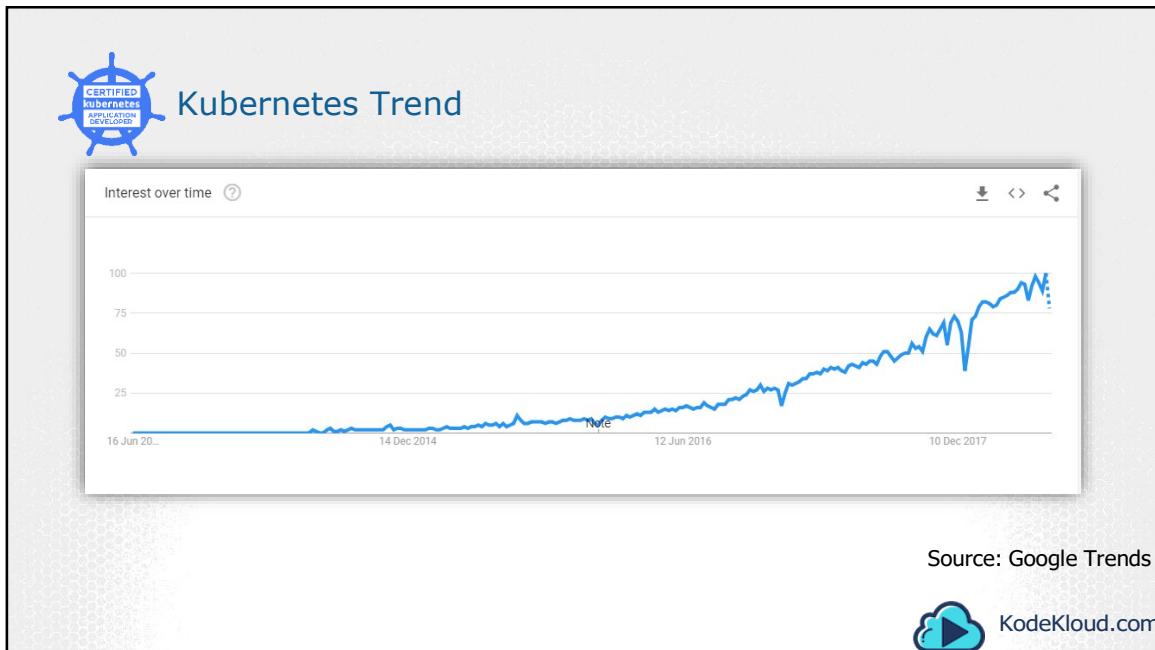


Now you don't have to take these courses in Order. If you are an administrator, you may chose to take the Beginners as well as the second course and get yourself certified as a Kubernetes Administrator. Or take the beginners course and the developers course to get yourself certified as a Kubernetes Application Developer. Which I'd say is the easier of the two if you were to ask me.

So if you are ready, let's get started.



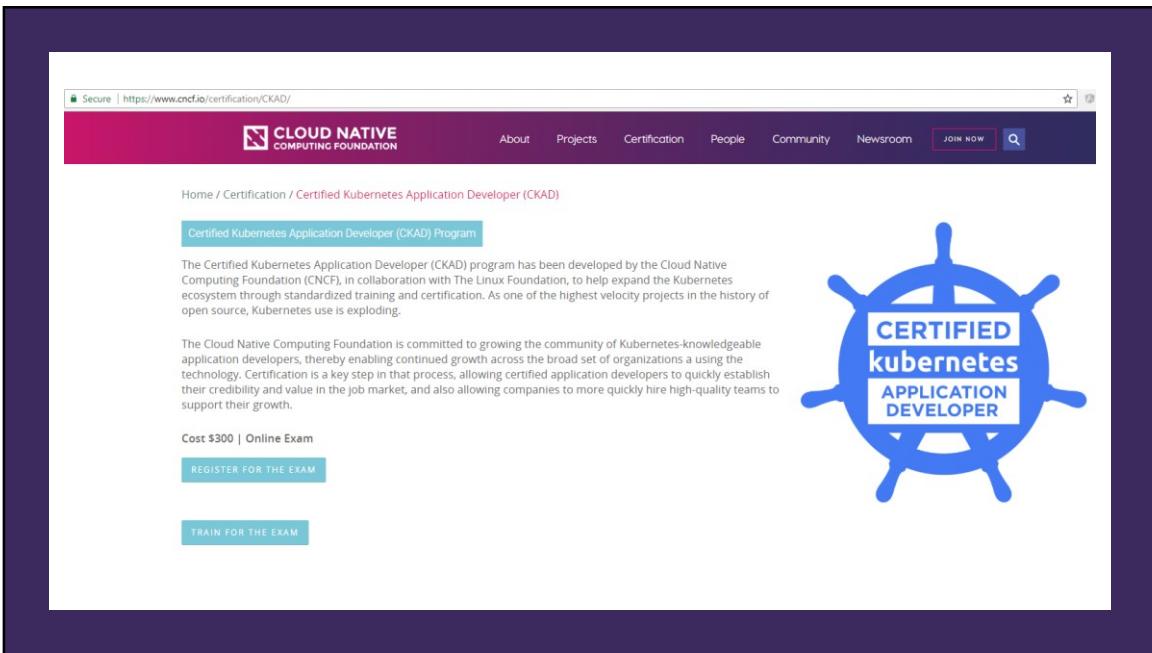
Hello and welcome to this lecture. In this lecture, we will look at some of the details around the Certified Kubernetes Application Developer program. What it is, why you need it and how to get started with it.



There is no doubt about the fact that the adoption of Kubernetes is expected to grow exponentially in the coming years, as seen in the graph from Google Trends. And so it is important for us to be prepared to establish credibility and value in the market.



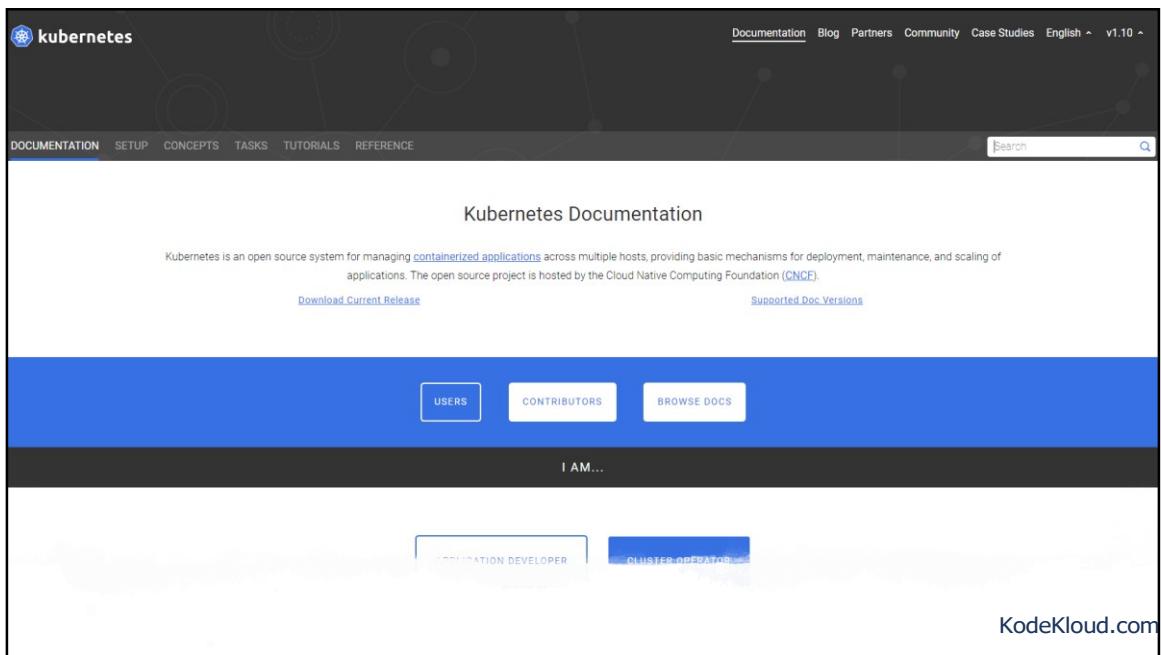
The Kubernetes Application Developers Certification, developed by the Cloud Native Computing Foundation in collaboration with The Linux Foundation, does just that. It helps you stand out in the crowd and allows companies to quickly hire high-quality engineers like you. On attaining the certification, you will be certified to design, and build cloud native applications for Kubernetes.



You can read more about the Certification at [cncf.io/certification/CKAD](https://cncf.io/certification/CKAD). As of today, the exam costs 300 USD, with one FREE retake. This means that in case you don't manage to pass on the initial attempt, which I am sure you will, you have one more attempt available for FREE within the next 12 months. The mode of delivery is Online. Which means you can deliver the exam, anytime anywhere at the comfort of your house. Since this is an online exam, an online proctor will be watching you at all times. There are a set of requirements that need to be met with respect to the environment, the room you are attending the exam from, the system you are using to give the test, your network connectivity etc. All of these are described in detail in the Candidate Handbook available on the certification web site.



Unlike most of the Certification exams out there, the Kubernetes Certification is not a multiple-choice exam. It is an online, performance-based exam that tests your hands-on skills with the technology. This would mean that you don't have to worry about memorizing lots of different numbers in preparation for the exam. However, you need to know how the technology works and how YOU can get it to work. You will be given different tasks to complete in a set amount of time – which happens to be 2 hours for this exam as per the exam guidelines. As far as I am concerned this is the BEST way to test a person's skills on a particular technology. I am not a big fan of multiple-choice exams.

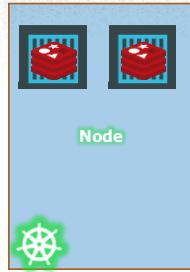


You will, however, be able to refer to the Kubernetes official documentation pages at all times during the exam. And in this course, we will walk through how to make best use of the documentation site, so that you can easily locate the right information. Well, I wish you good luck in preparing for and delivering the Kubernetes Certification exam and I am sure with enough practice, you will pass with flying colors. So let us begin.



In this lecture we start with the Core Concepts in Kubernetes - The Kubernetes Architecture. The Kubernetes Application Developers certification exam does not focus on setting up a Kubernetes cluster, it falls more under the Administrators certification. So as long as you have a working cluster ready, you are good to proceed with Application configuration. We have already gone through a high level overview of Kubernetes Architecture in the Beginner's course. And that is sufficient for this certification. You simply need to know what the various components are and what their responsibility is. We discuss these concepts in depth in the Kubernetes Administrators course. If you are familiar with the Architecture already, feel free to skip this lecture and head over to the next.

## Nodes (Minions)

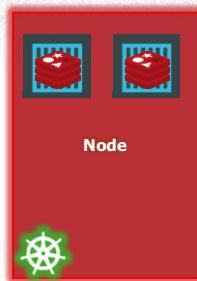


Kubernetes is configured on one or more Nodes. A node is a machine – physical or virtual – on which kubernetes is installed. A node is a worker machine and this is where containers are hosted.

It was also known as Minions in the past. So you might here these terms used interchangeably.

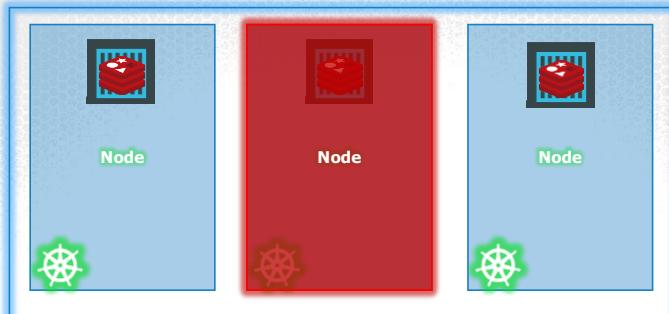
But what if the node on which our application is running fails? Well, obviously our application goes down. So you need to have more than one nodes for high availability and scaling.

## Nodes (Minions)



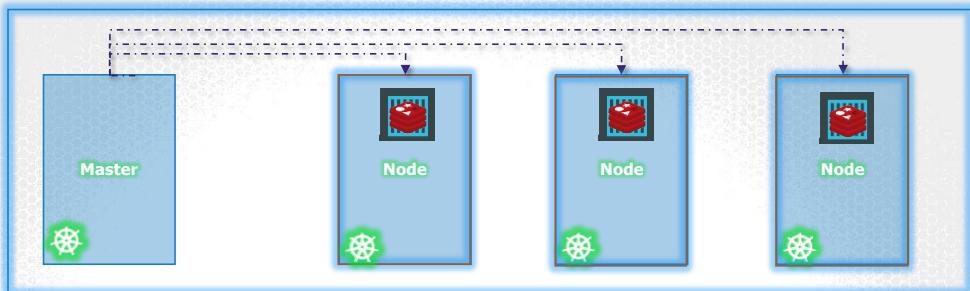
But what if the node on which our application is running fails? Well, obviously our application goes down. So you need to have more than one nodes for high availability and scaling.

# Cluster



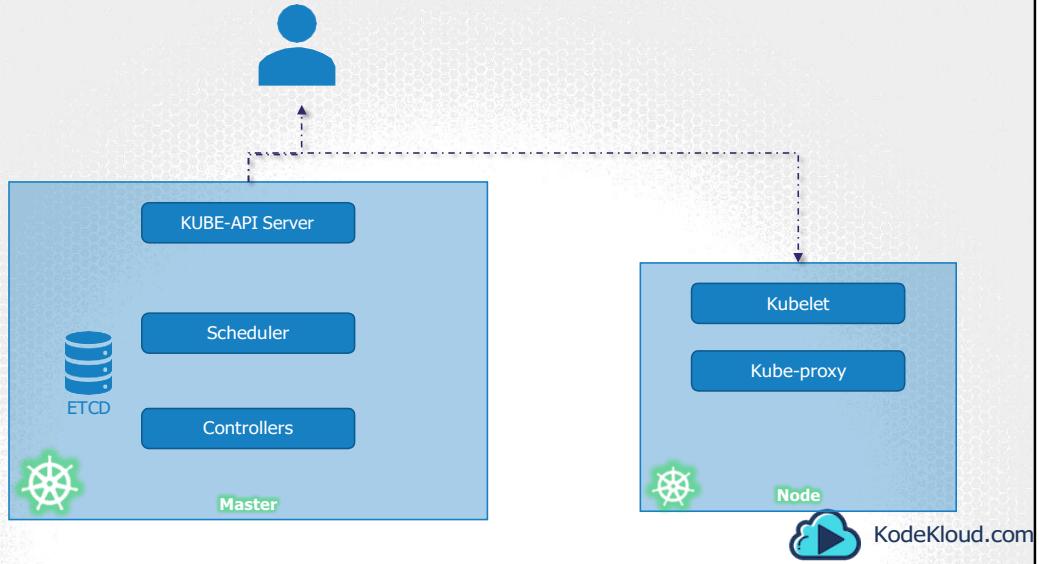
A cluster is a set of nodes grouped together. This way even if one node fails you have your application still accessible from the other nodes. Moreover having multiple nodes helps in sharing load as well.

## Master



Now we have a cluster, but who is responsible for managing the cluster? Where is the information about the members of the cluster stored? How are the nodes monitored? When a node fails how do you move the workload of the failed node to another worker node? That's where the Master comes in. The master is another node with Kubernetes installed in it, and is configured as a Master. The master watches over the nodes in the cluster and is responsible for the actual orchestration of containers on the worker nodes.

# Architecture



When you install Kubernetes on a System, you are actually installing the following components. An API Server. An ETCD service. A kubelet service. A Container Runtime, Controllers and Schedulers.

The API server acts as the front-end for kubernetes. The users, management devices, Command line interfaces all talk to the API server to interact with the kubernetes cluster.

Next is the ETCD key store. ETCD is a distributed reliable key-value store used by kubernetes to store all data used to manage the cluster. Think of it this way, when you have multiple nodes and multiple masters in your cluster, etcd stores all that information on all the nodes in the cluster in a distributed manner. ETCD is responsible for implementing locks within the cluster to ensure there are no conflicts between the Masters.

The scheduler is responsible for distributing work or containers across multiple nodes. It looks for newly created containers and assigns them to Nodes.

The controllers are the brain behind orchestration. They are responsible for noticing and responding when nodes, containers or endpoints goes down. The controllers makes decisions to bring up new containers in such cases.

The container runtime is the underlying software that is used to run containers. In our case it happens to be Docker.

And finally kubelet is the agent that runs on each node in the cluster. The agent is responsible for making sure that the containers are running on the nodes as expected.

An additional component on the Node is the kube-proxy. It takes care of networking within Kubernetes.

Well, that is all that you really need to know about the architecture in the scope of this certification.



... In this lecture we will discuss about PODs. We will first understand what PODs are and then practice developing POD definition files. You will work with advanced POD definition files and also troubleshoot issues with existing ones. This way you will get enough hands-on practice.

# Assumptions

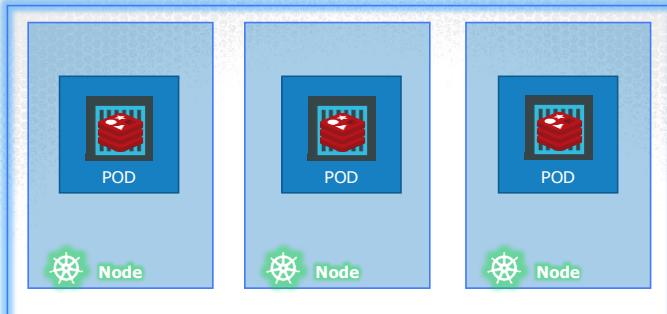
Docker Image

Kubernetes Cluster



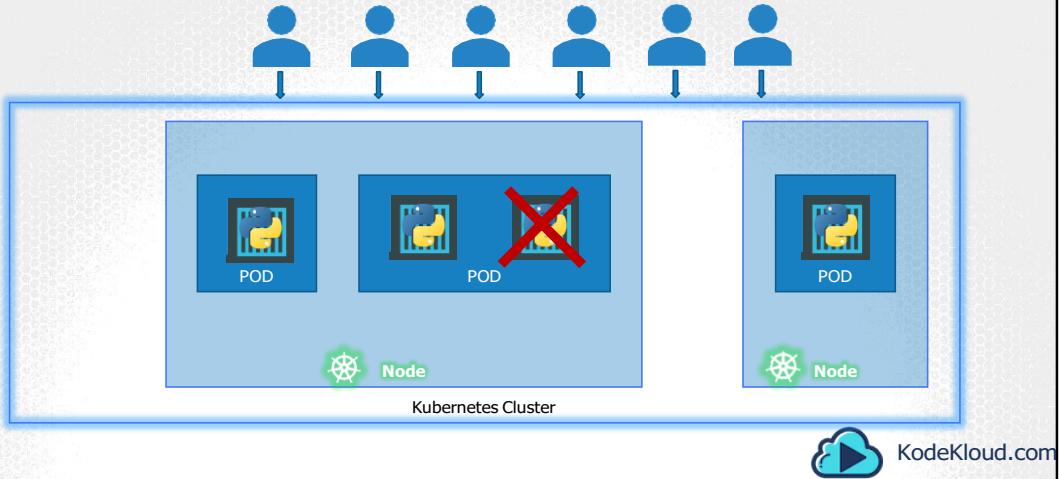
Before we head into understanding PODs, we would like to assume that the following have been setup already. At this point, we assume that the application is already developed and built into Docker Images and it is available on a Docker repository like Docker hub or any other internal registry, so kubernetes can pull it down. We also assume that the Kubernetes cluster has already been setup and is working. This could be a single-node setup or a multi-node setup, doesn't matter. All the services need to be in a running state.

## POD



As we discussed before, with kubernetes our ultimate aim is to deploy our application in the form of containers on a set of machines that are configured as worker nodes in a cluster. However, kubernetes does not deploy containers directly on the worker nodes. The containers are encapsulated into a Kubernetes object known as PODs. A POD is a single instance of an application. A POD is the smallest object, that you can create in kubernetes.

## POD

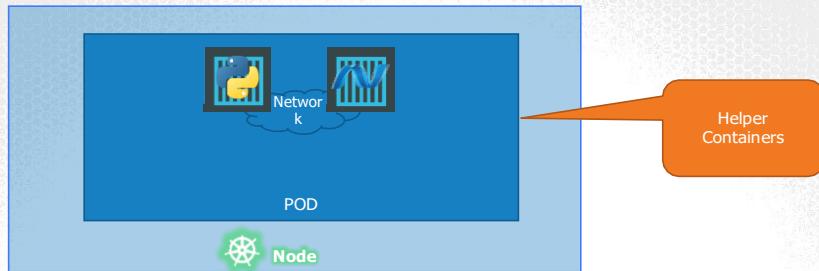


Here we see the simplest of simplest cases where you have a single node kubernetes cluster with a single instance of your application running in a single docker container encapsulated in a POD. What if the number of users accessing your application increase and you need to scale your application? You need to add additional instances of your web application to share the load. Now, where would you spin up additional instances? Do we bring up a new container instance within the same POD? No! We create a new POD altogether with a new instance of the same application. As you can see we now have two instances of our web application running on two separate PODs on the same kubernetes system or node.

What if the user base FURTHER increases and your current node has no sufficient capacity? Well THEN you can always deploy additional PODs on a new node in the cluster. You will have a new node added to the cluster to expand the cluster's physical capacity. <pause> SO, what I am trying to illustrate in this slide is that, PODs usually have a one-to-one relationship with containers running your application. To scale UP you create new PODs and to scale down you delete PODs. You do not add additional containers to an existing POD to scale your application. <pause> Also, if you are wondering how we implement all of this and how we achieve load balancing between containers etc, we will get into all of that in a later lecture. For now we are ONLY

trying to understand the basic concepts.

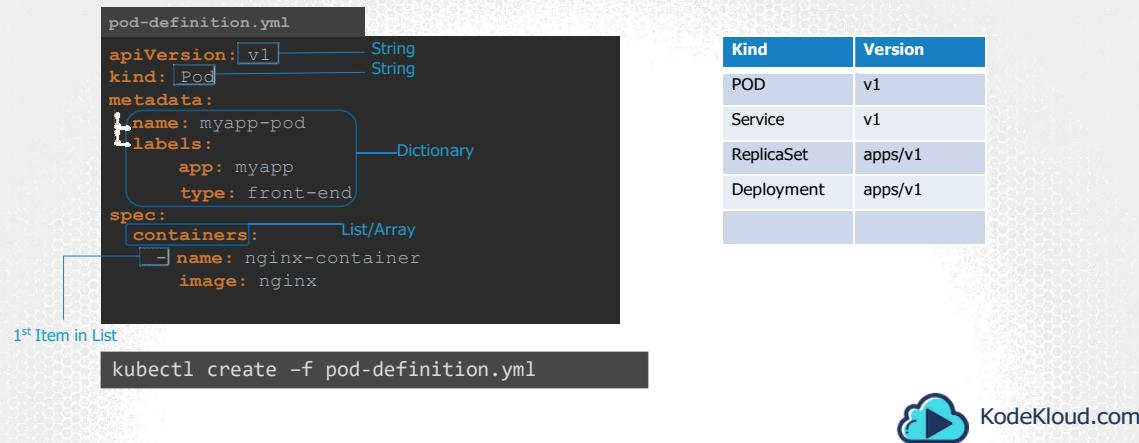
## Multi-Container PODs



Now we just said that PODs usually have a one-to-one relationship with the containers, but, are we restricted to having a single container in a single POD? No! A single POD CAN have multiple containers, except for the fact that they are usually not multiple containers of the same kind. As we discussed in the previous slide, if our intention was to scale our application, then we would need to create additional PODs. But sometimes you might have a scenario where you have a helper container, that might be doing some kind of supporting task for our web application such as processing a user entered data, processing a file uploaded by the user etc. and you want these helper containers to live along side your application container. In that case, you CAN have both of these containers part of the same POD, so that when a new application container is created, the helper is also created and when it dies the helper also dies since they are part of the same POD. The two containers can also communicate with each other directly by referring to each other as 'localhost' since they share the same network namespace. Plus they can easily share the same storage space as well.

This is only an introduction to multi-container PODs. We have a more detailed section coming up on the different types of multi-container PODs later in this course.

# YAML in Kubernetes



Kubernetes uses YAML files as input for the creation of objects such as PODs, Replicas, Deployments, Services etc. All of these follow similar structure. If you are not familiar with YAML language, refer to the beginners course where we learn YAML language through some fun coding exercises section.

A Kubernetes definition file always contains 4 top level fields. The `apiVersion`, `kind`, `metadata` and `spec`. These are top level or root level properties. Think of them as siblings, children of the same parent. These are all REQUIRED fields, so you MUST have them in your configuration file.

Let us look at each one of them. The first one is the `apiVersion`. This is the version of the Kubernetes API we're using to create the object. Depending on what we are trying to create we must use the RIGHT `apiVersion`. For now since we are working on PODs, we will set the `apiVersion` as `v1`. Few other possible values for this field are `apps/v1beta1`, `extensions/v1beta1` etc. We will see what these are for later in this course.

Next is the `kind`. The `kind` refers to the type of object we are trying to create, which in this case happens to be a POD. So we will set it as `Pod`. Some other possible values

here could be ReplicaSet or Deployment or Service, which is what you see in the kind field in the table on the right.

The next is metadata. The metadata is data about the object like its name, labels etc. As you can see unlike the first two were you specified a string value, this, is in the form of a dictionary. So everything under metadata is intended to the right a little bit and so names and labels are children of metadata. Under metadata, the name is a string value – so you can name your POD myapp-pod - and the labels is a dictionary. So labels is a dictionary within the metadata dictionary. And it can have any key and value pairs as you wish. For now I have added a label app with the value myapp. Similarly you could add other labels as you see fit which will help you identify these objects at a later point in time. Say for example there are 100s of PODs running a front-end application, and 100's of them running a backend application or a database, it will be DIFFICULT for you to group these PODs once they are deployed. If you label them now as front-end, back-end or database, you will be able to filter the PODs based on this label at a later point in time.

It's IMPORTANT to note that under metadata, you can only specify name or labels or anything else that kubernetes expects to be under metadata. You CANNOT add any other property as you wish under this. However, under labels you CAN have any kind of key or value pairs as you see fit. So its IMPORTANT to understand what each of these parameters expect.

So far we have only mentioned the type and name of the object we need to create which happens to be a POD with the name myapp-pod, but we haven't really specified the container or image we need in the pod. The last section in the configuration file is the specification which is written as spec. Depending on the object we are going to create, this is where we provide additional information to kubernetes pertaining to that object. This is going to be different for different objects, so its important to understand or refer to the documentation section to get the right format for each. Since we are only creating a pod with a single container in it, it is easy. Spec is a dictionary so add a property under it called containers, which is a list or an array. The reason this property is a list is because the PODs can have multiple containers within them as we learned in the lecture earlier. In this case though, we will only add a single item in the list, since we plan to have only a single container in the POD. The item in the list is a dictionary, so add a name and image property. The value for image is nginx.

Once the file is created, run the command `kubectl create -f` followed by the file name which is `pod-definition.yml` and kubernetes creates the pod.

So to summarize remember the 4 top level properties. apiVersion, kind, metadata and spec. Then start by adding values to those depending on the object you are creating.

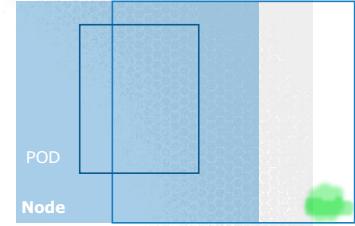
# kubectl

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

```
kubectl get pods
```

```
C:\Kubernetes>kubectl get pods
NAME          READY   STATUS        RESTARTS   AGE
nginx-8586cf59-whssr  0/1    ContainerCreating   0          3s
```

```
C:\Kubernetes>kubectl get pods
NAME          READY   STATUS        RESTARTS   AGE
nginx-8586cf59-whssr  1/1    Running     0          8s
```



KodeKloud.com

Let us now look at how to deploy PODs. Earlier we learned about the kubectl run command. What this command really does is it deploys a docker container by creating a POD. So it first creates a POD automatically and deploys an instance of the nginx docker image. But where does it get the application image from? For that you need to specify the image name using the --image parameter. The application image, in this case the nginx image, is downloaded from the docker hub repository. Docker hub as we discussed is a public repository where latest docker images of various applications are stored. You could configure kubernetes to pull the image from the public docker hub or a private repository within the organization.

Now that we have a POD created, how do we see the list of PODs available? The kubectl get PODs command helps us see the list of pods in our cluster. In this case we see the pod is in a ContainerCreating state and soon changes to a Running state when it is actually running.

Also remember that we haven't really talked about the concepts on how a user can access the nginx web server. And so in the current state we haven't made the web server accessible to external users. You can access it internally from the Node though. For now we will just see how to deploy a POD and in a later lecture once we learn

about networking and services we will get to know how to make this service accessible to end users.

# Commands

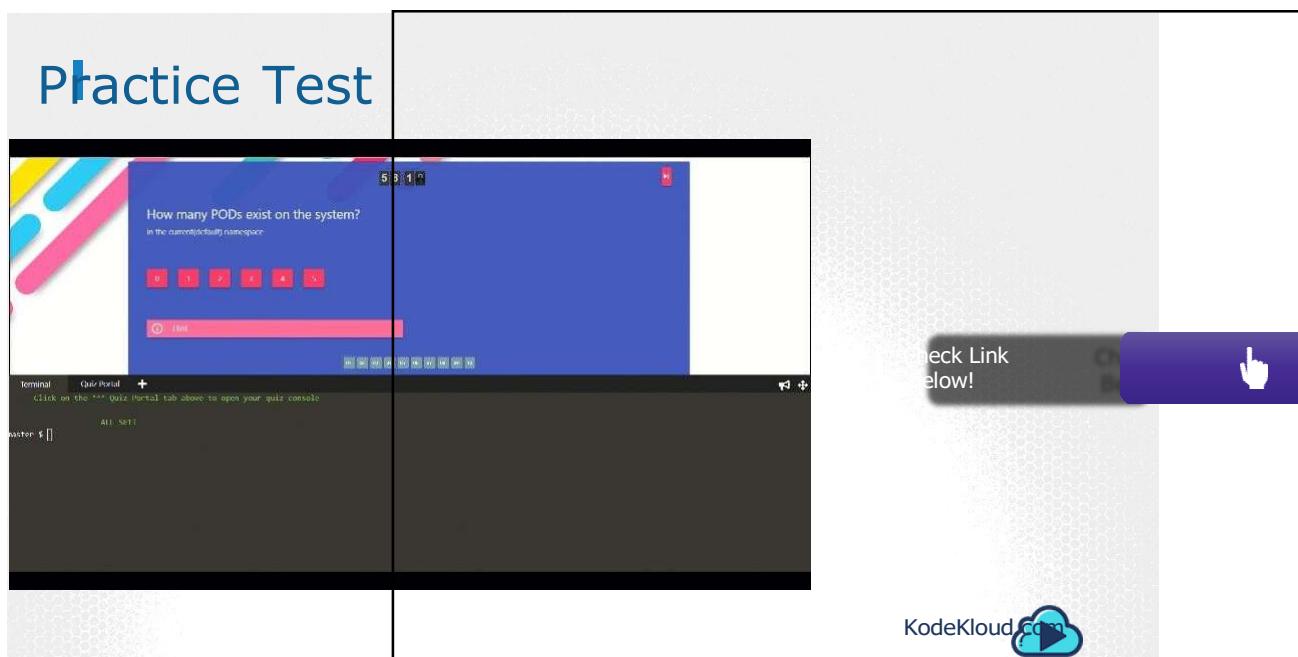
```
> kubectl get pods
NAME      READY   STATUS    RESTARTS   AGE
myapp-pod  1/1     Running   0          20s

> kubectl describe pod myapp-pod
Name:           myapp-pod
Namespace:      default
Node:          minikube/192.168.99.100
Start Time:    Sat, 03 Mar 2018 14:26:14 +0800
Labels:        app=nginx
               name=myapp-pod
Annotations:   <none>
Status:        Running
IP:           10.244.8.24
Containers:   nginx
              Container ID: docker://8390bb5ac8c42a80b4bb70e0c1488fa1bc38063e49180bc2f5a/83e7688b8c9d
              Image:          nginx
              Image ID:       docker-pullable://nginx@sha256:4771d09578c7c6a65299e110b3ee1c0a2502f5ea2618d23e4ffe7a4cabice5de
              Port:          <none>
              State:         Running
                Started:   Sat, 03 Mar 2018 14:26:21 +0800
              Ready:         True
              Restart Count: 0
              Environment:  <none>
              Mounts:        /var/run/secrets/kubernetes.io/serviceaccount from default-token-x95w7 (ro)
Conditions:   IP:           Status
              Initialized:  True
              Ready:        True
              PodScheduled: True
Events:        Type  Reason  Age   From            Message
              Normal  Scheduled  34s  default-scheduler  Successfully assigned myapp-pod to minikube
              Normal  SuccessfulMountVolume  33s  kubelet, minikube  MountVolume.SetUp succeeded for volume "default-token-x95w7"
              Normal  Pulling   33s  kubelet, minikube  pulling image "nginx"
              Normal  Pulled   27s  kubelet, minikube  Successfully pulled image "nginx"
              Normal  Created   27s  kubelet, minikube  Created container
              Normal  Started   27s  kubelet, minikube  Started container
```



KodeKloud.com

Once we create the pod, how do you see it? Use the `kubectl get pods` command to see a list of pods available. In this case its just one. To see detailed information about the pod run the `kubectl describe pod` command. This will tell you information about the POD, when it was created, what labels are assigned to it, what docker containers are part of it and the events associated with that POD.



Access Test Here: <https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743640>

## Course Objectives

- ✓ Core Concepts
  - ✓ Kubernetes Architecture
  - ✓ Create and Configure Pods
- Configuration
- Multi-Container Pods
- Observability
- Pod Design
- Services & Networking
- State Persistence



KodeKloud.com

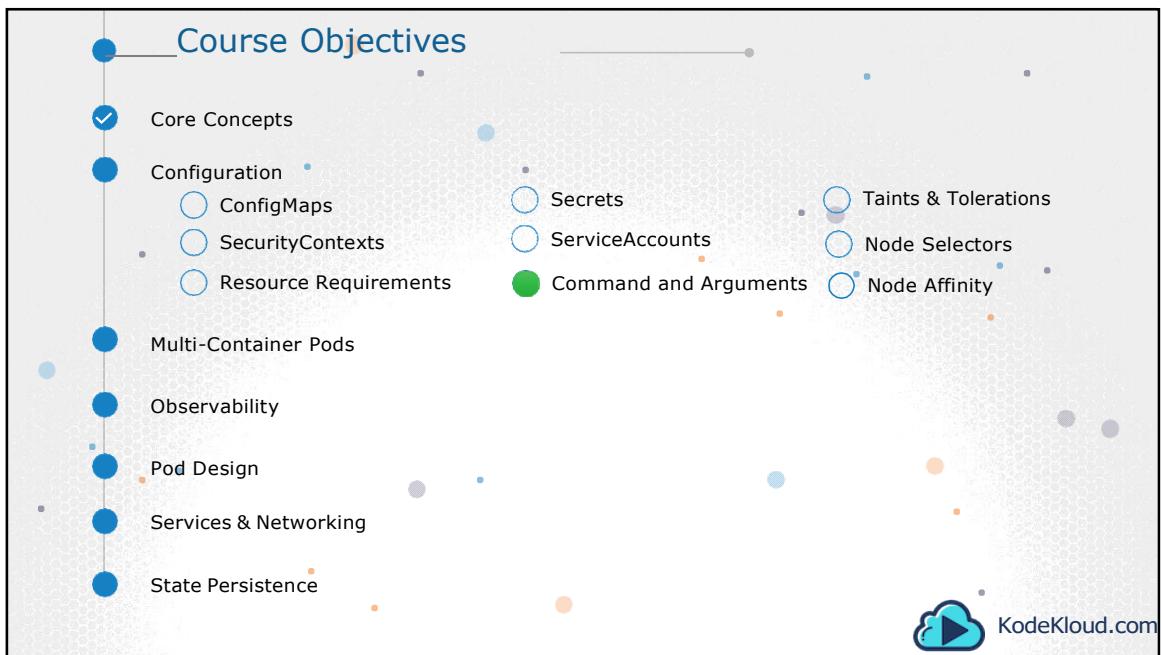


46  
r

# COMMANDS & ARGUMENTS



KodeKloud.com



In this section we will talk about Command and Arguments in a Pod Definition. This is not listed as a required topic in the certification curriculum, but I think its important to explain it as it is a topic that is usually overlooked.

Docker Container Creation and Status					
<code>▶ docker run ubuntu</code>					
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
<code>▶ docker ps -a</code>					
45aacca36850	ubuntu	"/bin/bash"	43 seconds ago	Exited (0) 41 seconds ago	



Let's first refresh our memory on commands in containers and docker. We will then translate this into PODs in the next lecture. Let's start with a simple scenario. Say you were to run a docker container from an Ubuntu image. When you run the “docker run ubuntu” command, it runs an instance of Ubuntu image and exits immediately. If you were to list the running containers you wouldn't see the container running. If you list all containers including those that are stopped, you will see that the new container you ran is in an Exited state. Now why is that?



KodeKloud.com

Unlike Virtual Machines, containers are not meant to host an Operating System. Containers are meant to run a specific task or process. Such as to host an instance of a Web Server, or Application Server or a database or simply to carry out some kind of computation or analysis. Once the task is complete the container exits. A container only lives as long as the process inside it is alive. If the web service inside the container is stopped or crashes the container exits.

```

# Install Nginx.
RUN \
    add-apt-repository -y ppa:nginx/stable && \
    apt-get update && \
    apt-get install -y nginx && \
    rm -rf /var/lib/apt/lists/* && \
    echo "\ndaemon off;" >> /etc/nginx/nginx.conf && \
    chown -R www-data:www-data /var/lib/nginx

# Define mountable directories.
VOLUME ["/etc/nginx/sites-enabled", "/etc/nginx/certs", "/etc/nginx/conf.d"]

# Define working directory.
WORKDIR /etc/nginx

# Define default command.
CMD ["nginx"]

```

```

ARG MYSQL_SERVER_PACKAGE_URL=https://repo.mysql.com/yum/mysql-8.0-community/docker/x86_64
ARG MYSQL_SHELL_PACKAGE_URL=https://repo.mysql.com/yum/mysql-tools-community/el/7/x86_64

# Install server
RUN rpmkeys --import https://repo.mysql.com/RPM-GPG-KEY-mysql \
    && yum install -y $MYSQL_SERVER_PACKAGE_URL $MYSQL_SHELL_PACKAGE_URL libpwquality \
    && yum clean all \
    && mkdir /docker-entrypoint-initdb.d

VOLUME /var/lib/mysql

COPY docker-entrypoint.sh /entrypoint.sh
COPY healthcheck.sh /healthcheck.sh
ENTRYPOINT ["/entrypoint.sh"]
HEALTHCHECK CMD /healthcheck.sh
EXPOSE 3306 33060
CMD ["mysqld"]

```



KodeKloud.com

So who defines what process is run within the container? If you look at the Dockerfile for the NGINX image, you will see an Instruction called CMD which stands for command that defines the program that will be run within the container when it starts. For the NGINX image it is the nginx command, for the mysql image it is the mysqld command.

```
# Pull base image.  
FROM ubuntu:14.04  
  
# Install.  
RUN \  
    sed -i 's/# \(\.*multiverse$\)/\1/g' /etc/apt/sources.list && \  
    apt-get update && \  
    apt-get -y upgrade && \  
    apt-get install -y build-essential && \  
    apt-get install -y software-properties-common && \  
    apt-get install -y byobu curl git htop man unzip vim wget && \  
    rm -rf /var/lib/apt/lists/*  
  
# Add files.  
ADD root/.bashrc /root/.bashrc  
ADD root/.gitconfig /root/.gitconfig  
ADD root/.scripts /root/.scripts  
  
# Set environment variables.  
ENV HOME /root  
  
# Define working directory.  
WORKDIR /root  
  
...# Define default command.  
CMD ["bash"]
```



KodeKloud.com

What we tried to do earlier was to run a container with a plain Ubuntu Operating System. Let us look at the Dockerfile for this image. You will see that it uses “bash” as the default command. Now, bash is not really a process like a web server or database. It is a shell that listens for inputs from a terminal. If it cannot find a terminal it exits.



??



KodeKloud.com

When we ran the Ubuntu container earlier, Docker created a container from the Ubuntu image, and launched the bash program. By default Docker does not attach a terminal to a container when it is run. And so the bash program does not find a terminal and so it exits. Since the process, that was started when the container was created, finished, the container exits as well.

```
▶ docker run ubuntu [COMMAND]
```

```
▶ docker run ubuntu sleep 5
```



5



KodeKloud.com

So how do you specify a different command to start the container? One option is to append a command to the docker run command and that way it overrides the default command specified within the image. In this case I run the docker run ubuntu command with the “sleep 5” command as the added option. This way when the container starts it runs the sleep program, waits for 5 seconds and then exits.

FROM Ubuntu

CMD sleep 5

CMD command param1      CMD sleep 5

CMD ["command", "param1"]      CMD ["sleep", "5"]      CMD ["sleep 5"]

✓      ✗

docker build -t ubuntu-sleeper .

docker run ubuntu-sleeper

5

KodeKloud.com

But how do you make that change permanent? Say you want the container to always run the sleep command when it starts.

You would then create your own image from the base Ubuntu image and specify a new command.

There are different ways of specifying the command. Either the command simply as is in a shell form. Or in a JSON array format like this. But remember, when you specify in a JSON array format, the first element in the array should be the executable. In this case the sleep program. Do not specify the command and parameters together like this. The first element should always be an executable.

So I now build by new image using the docker build command, and name it as ubuntu-sleeper. I could now simply run the docker ubuntu sleeper command and get the same results. It always sleeps for 5 seconds and exits.

The screenshot shows a terminal window with two examples of Docker commands.

**Example 1 (CMD):**

```
FROM Ubuntu
CMD sleep 5
```

Output:

```
▶ docker run ubuntu-sleeper sleep 10
```

Annotation: Command at Startup: sleep 10

**Example 2 (ENTRYPOINT):**

```
FROM Ubuntu
ENTRYPOINT ["sleep"]
```

Output:

```
▶ docker run ubuntu-sleeper 10
```

Annotation: Command at Startups: sleep 10

Output:

```
▶ docker run ubuntu-sleeper
sleep: missing operand
Try 'sleep --help' for more information.
```

Annotation: Command at Startup: sleep

In the bottom right corner, there is a KodeKloud.com logo.

But what if I wish to change the number of seconds it sleeps. Currently it is hardcoded to 5 seconds. As we learned before one option is to run the docker run command with the new command appended to it. In this case sleep 10. And so the command that will be run at startup will be sleep 10. But it doesn't look very good. The name of the image ubuntu-sleeper in itself implies that the container will sleep. So we shouldn't have to specify the sleep command again. Instead we would like it to be something like this. Docker run ubuntu-sleeper 10. We only want to pass in the number of seconds the container should sleep and the sleep command should be invoked automatically.

And that is where the entrypoint instruction comes into play. The entrypoint instruction is like the command instruction, as in you can specify the program that will be run when the container starts. And whatever you specify on the command line, in this case 10, will get appended to the entrypoint. So the command that will be run when the container starts is sleep 10.

So that's the difference between the two. In case of the CMD instruction the command line parameters passed will get replaced entirely, whereas in case of entrypoint the command line parameters will get appended.

Now, in the second case what if I run the ubuntu-sleeper without appending the number of seconds? Then the command at startup will be just sleep and you get the error that the operand is missing. So how do you add a default value as well?

The screenshot shows a terminal window with three distinct sections. The first section contains the Dockerfile code:

```
FROM Ubuntu
ENTRYPOINT ["sleep"]
CMD ["5"]
```

The second section shows the command: `docker run ubuntu-sleeper`, resulting in the output: `Command at Startup: sleep 5`.

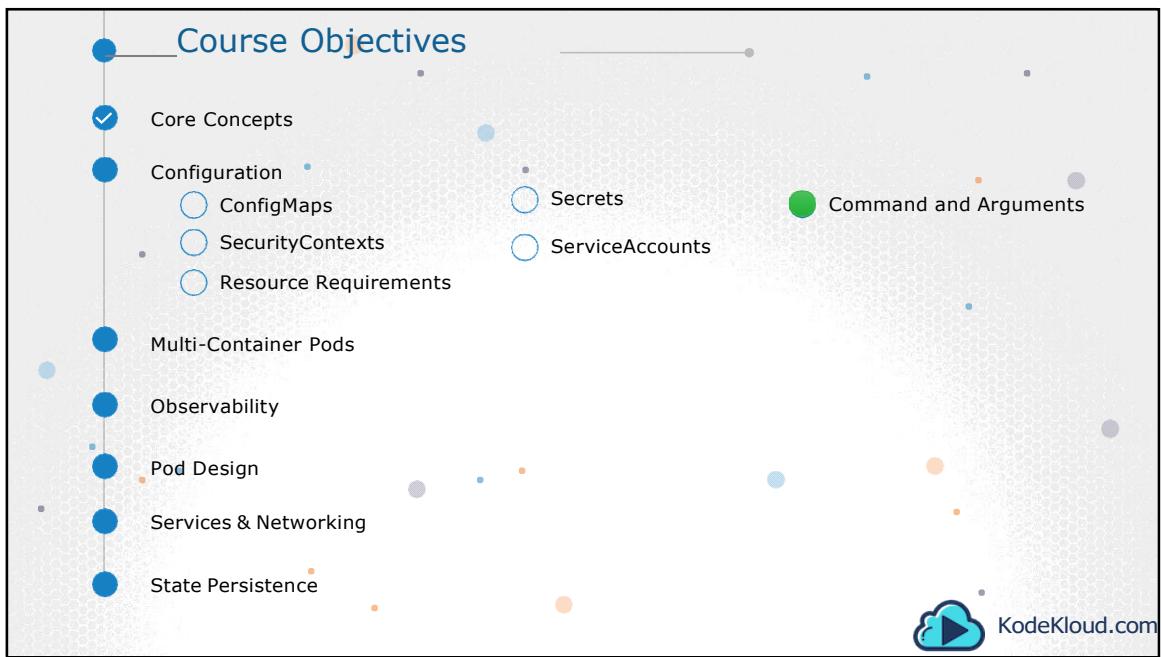
The third section shows the command: `docker run ubuntu-sleeper 10`, resulting in the output: `Command at Startup: sleep 10`.

The fourth section shows the command: `docker run --entrypoint sleep2.0 ubuntu-sleeper 10`, resulting in the output: `Command at Startup: sleep2.0 10`.

KodeKloud.com

That's where you would use both Entrypoint as well as the CMD instruction. In this case the command instruction will be appended to the entrypoint instruction. So at startup, the command would be sleep 5, if you didn't specify any parameters in the command line. If you did, then that will override the command instruction. And remember, for this to happen, you should always specify the entrypoint and command instructions in a JSON format.

Finally, what if you really want to modify the entrypoint during run time? Say from sleep to a hypothetical sleep2.0 command? Well in that case you can override it by using the --entrypoint option in the docker run command. The final command at startup would then be sleep2.0 10



We will now look at Command and Arguments in a Kubernetes POD.

```
▶ docker run --name ubuntu-sleeper ubuntu-sleeper  
▶ docker run --name ubuntu-sleeper ubuntu-sleeper 10
```

```
pod-definition.yml  
apiVersion: v1  
kind: Pod  
metadata:  
  name: ubuntu-sleeper-pod  
spec:  
  containers:  
    - name: ubuntu-sleeper  
      image: ubuntu-sleeper  
      args: ["10"]
```

```
▶ kubectl create -f pod-definition.yml
```



KodeKloud.com

In the previous lecture we created a simple docker image that sleeps for a given number of seconds. We named it ubuntu-sleeper and we ran it using the docker command docker run ubuntu-sleeper. By default it sleeps for 5 seconds, but you can override it by passing a command line argument. We will now create a pod using this image. We start with a blank pod definition template, input the name of the pod and specify the image name. When the pod is created, it creates a container from the specified image, and the container sleeps for 5 seconds before exiting.

Now, if you need the container to sleep for 10 seconds as in the second command, how do you specify the additional argument in the pod-definition file? Anything that is appended to the docker run command will go into the “args” property of the pod definition file, in the form of an array like this.

```
FROM Ubuntu
```

```
ENTRYPOINT ["sleep"]
```

```
CMD ["5"]
```

```
▶ docker run --name ubuntu-sleeper \
--entrypoint sleep2.0
ubuntu-sleeper 10
```

```
pod-definition.yml
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper-pod
spec:
  containers:
    - name: ubuntu-sleeper
      image: ubuntu-sleeper
      command: ["sleep2.0"]
      args: ["10"]
```

```
▶ kubectl create -f pod-definition.yml
```



KodeKloud.com

Let us try to relate that to the Dockerfile we created earlier. The Dockerfile has an Entrypoint as well as a CMD instruction specified. The entrypoint is the command that is run at startup, and the CMD is the default parameter passed to the command. With the args option in the pod-definition file we override the CMD instruction in the Dockerfile. But what if you need to override the entrypoint? Say from sleep to a hypothetical sleep2.0 command? In the docker world, we would run the docker run command with the --entrypoint option set to the new command. The corresponding entry in the pod definition file would be using a command field. The command field corresponds to entrypoint instruction in the Dockerfile.

FROM Ubuntu

**ENTRYPOINT** ["sleep"]

**CMD** ["5"]

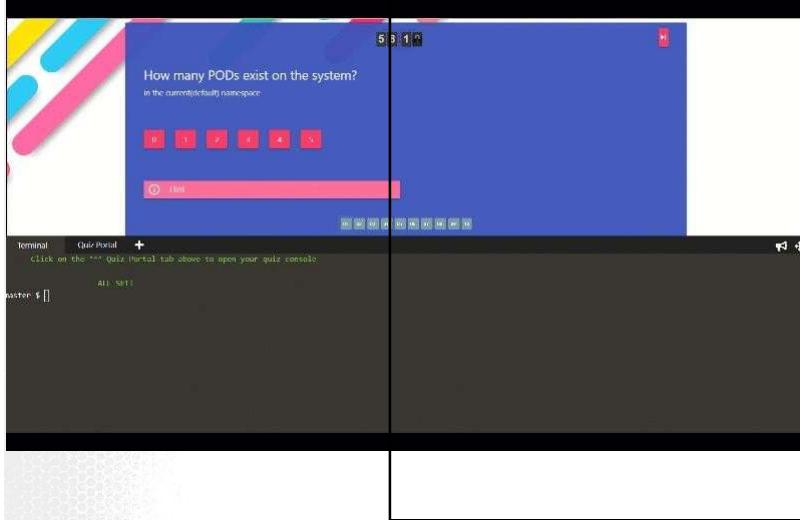
```
pod-definition.yml
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper-pod
spec:
  containers:
    - name: ubuntu-sleeper
      image: ubuntu-sleeper
      command: ["sleep2.0"]
      args: ["10"]
```



KodeKloud.com

So to summarize, there are two fields that correspond to two instructions in the Dockerfile. The command overrides the entrypoint instruction and the args field overrides the command instruction in the Dockerfile. Remember the command field does not override the CMD instruction in the Dockerfile.

## Practice Test



Check Link  
below!

KodeKloud.com

Access Test Here: <https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743655>

## I References

<https://kubernetes.io/docs/tasks/inject-data-application/define-command-argument-container/>

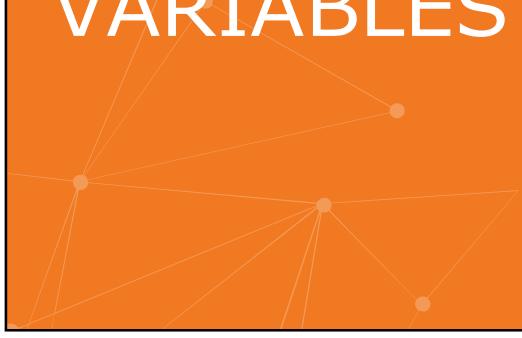




In this section we will talk about concepts around Configuration in Kubernetes. We will start with ConfigMaps. First we will see what a configuration item is by using a simple example of a web application and how it is set in Docker. And then we will see how it is configured in Kubernetes. If you know about environment variables and how they are set in Docker already, please skip the next lecture.



# ENVIRONMENT VARIABLES

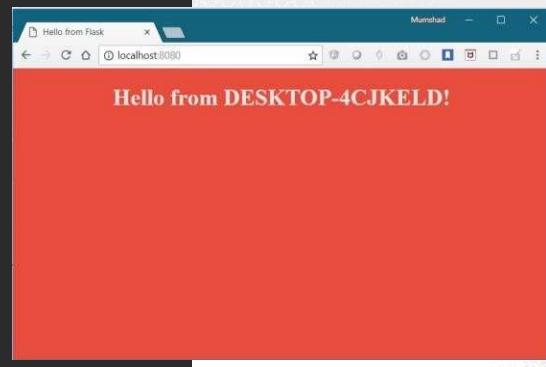


KodeKloud.com

# Environment Variables

```
import os  
from flask import Flask  
  
app = Flask( name )  
...  
...  
color = "red"  
  
@app.route("/")  
def main():  
    print(color)  
    return render_template('hello.html', color=color)  
  
if __name__ == "__main__":  
    app.run(host="0.0.0.0", port=8080)
```

python app.py



KodeKloud

Let us start with a simple web application written in Python. This piece of code is used to create a web application that displays a webpage with a background color.

# Environment Variables

app.py

```
import os
from flask import Flask

app = Flask(__name__)

...
...

color = "red"

@app.route("/")
def main():
    print(color)
    return render_template('hello.html', color=color)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port="8080")
```



If you look closely into the application code, you will see a line that sets the background color to red. Now, that works just fine. However, if you decide to change the color in the future, you will have to change the application code. It is a best practice to move such information out of the application code.

# Environment Variables

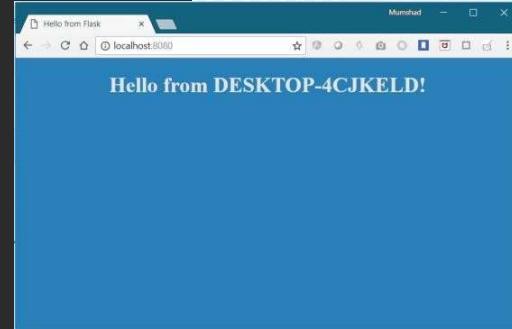
```
import os
from flask import Flask

app = Flask( name )
...
...
color = os.environ.get('APP_COLOR')

@app.route("/")
def main():
    print(color)
    return render_template('hello.html', color=color)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port="8080")
```

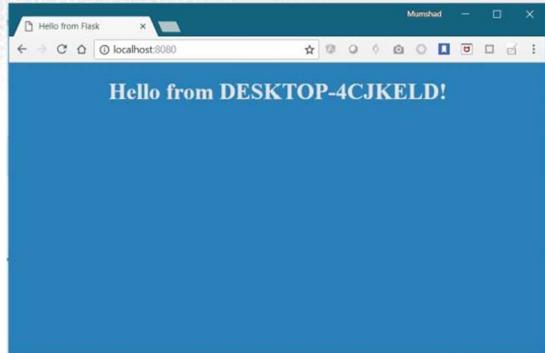
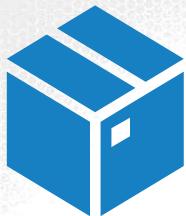
```
export APP_COLOR=blue; python app.py
```



KodeKloud

And into, say an environment variable called APP\_COLOR. The next time you run the application, set an environment variable called APP\_COLOR to a desired value, and the application now has a new color.

# ENV Variables in Docker



```
▶ docker run -e APP_COLOR=blue simple-webapp-color
```



KodeKloud.com

Once your application gets packaged into a Docker image, you would then run it with the docker run command followed by the name of the image. However, if you wish to pass the environment variable as we did before, you would now use the docker run command's -e option to set an environment variable within the container.

# ENV Variables in Docker

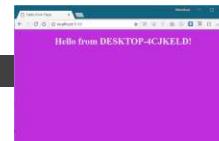
```
docker run -e APP_COLOR=blue simple-webapp-color
```



```
docker run -e APP_COLOR=green simple-webapp-color
```



```
docker run -e APP_COLOR=pink simple-webapp-color
```



KodeKloud

To deploy multiple containers with different colors, you would run the docker command multiple times and set a different value for the environment variable each time.

# ENV Variables in Kubernetes

```
▶ docker run -e APP_COLOR=pink simple-webapp-color
```

pod-definition.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
spec:
  containers:
    - name: simple-webapp-color
      image: simple-webapp-color
      ports:
        - containerPort: 8080
      env:
        - name: APP_COLOR
          value: pink
```



KodeKloud.com

Let us now see how to pass in an environment variable in Kubernetes. Given a pod definition file, which uses the same image as the docker command. To set an environment variable, use the ENV property. ENV is an array. So every item under the env property starts with a dash, indicating an item in the array. Each item has a name and a value property. The name is the name of the environment variable made available within the container and the value is its value.

# ENV Value Types

```
env:  
  - name: APP_COLOR  
    value: pink
```



```
env:  
  - name: APP_COLOR  
    valueFrom:  
      configMapKeyRef:
```



```
env:  
  - name: APP_COLOR  
    valueFrom:  
      secretKeyRef:
```



What we just saw was a direct way of specifying the environment variables using a plain key value pair format. However there are other ways of setting the environment variables – such as using ConfigMaps and Secrets. The difference in this case is that instead of specifying value, we say valueFrom. And then a specification of configMap or secret. We will discuss about configMaps and secretKeys in the upcoming lectures.

## Course Objectives

- ✓ Core Concepts
- Configuration
  - ConfigMaps
  - SecurityContexts
  - Resource Requirements
- Multi-Container Pods
- Observability
- Pod Design
- Services & Networking
- State Persistence

- Secrets
- ServiceAccounts



Hello, In this lecture we discuss how to work with configuration data in Kubernetes.

# I ConfigMaps

ConfigMap

pod-definition.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
    - containerPort: 8080
  env:
  - name: APP_COLOR
    value: blue
  - name: APP_MODE
    value: prod
```



KodeKloud.com

In the previous lecture we saw how to define environment variables in a pod definition file. When you have a lot of pod definition files, it will become difficult to manage the environment data stored within various files. We can take this information out of the pod definition file and manage it centrally using Configuration Maps.

ConfigMaps are used to pass configuration data in the form of key value pairs in Kubernetes.

When a POD is created, inject the ConfigMap into the POD, so the key value pairs are available as environment variables for the application hosted inside the container in the POD.

So there are two phases involved in configuring ConfigMaps. First create the ConfigMaps and second Inject them into the POD.

# I ConfigMaps

The diagram illustrates the two-step process of creating and injecting a ConfigMap into a Pod:

- Create ConfigMap:** A dark gray box labeled "ConfigMap" contains environment variables: APP\_COLOR : blue and APP\_MODE: prod.
- Inject into Pod:** A green circle with the number 2 contains the text "Inject into Pod".

**Code Examples:**

**ConfigMap:**

```
APP_COLOR : blue
APP_MODE: prod
```

**pod-definition.yaml:**

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
    - containerPort: 8080
  envFrom:
  - configMapRef:
      name: app-config
```

**KodeKloud.com**

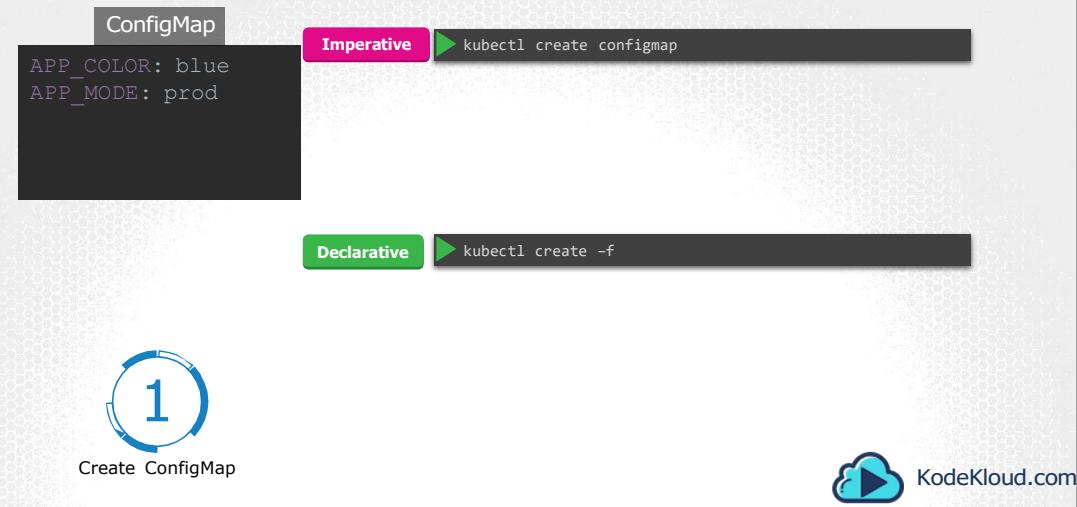
In the previous lecture we saw how to define environment variables in a pod definition file. When you have a lot of pod definition files, it will become difficult to manage the environment data stored within various files. We can take this information out of the pod definition file and manage it centrally using Configuration Maps.

ConfigMaps are used to pass configuration data in the form of key value pairs in Kubernetes.

When a POD is created, inject the ConfigMap into the POD, so the key value pairs are available as environment variables for the application hosted inside the container in the POD.

So there are two phases involved in configuring ConfigMaps. First create the ConfigMaps and second Inject them into the POD.

# Create ConfigMaps



Just like any other Kubernetes objects, there are two ways of creating a ConfigMap. The imperative way - without using a ConfigMap definition file and the Declarative way by using a ConfigMap Definition file.

If you do not wish to create a configmap definition, you could simply use the `kubectl create configmap` command and specify the required arguments. Let's take a look at that first.

With this method you can directly specify the key value pairs in the command line. To create a configMap of the given values, run the `kubectl create configmap` command.

# Create ConfigMaps

The diagram illustrates the creation of a ConfigMap using an imperative command. On the left, there is a terminal window showing the command:

```
kubectl create configmap <config-name> --from-literal=<key>=<value>
```

Below this, another terminal window shows a more complex example:

```
kubectl create configmap \ app-config --from-literal=APP_COLOR=blue \ --from-literal=APP_MODE=prod
```

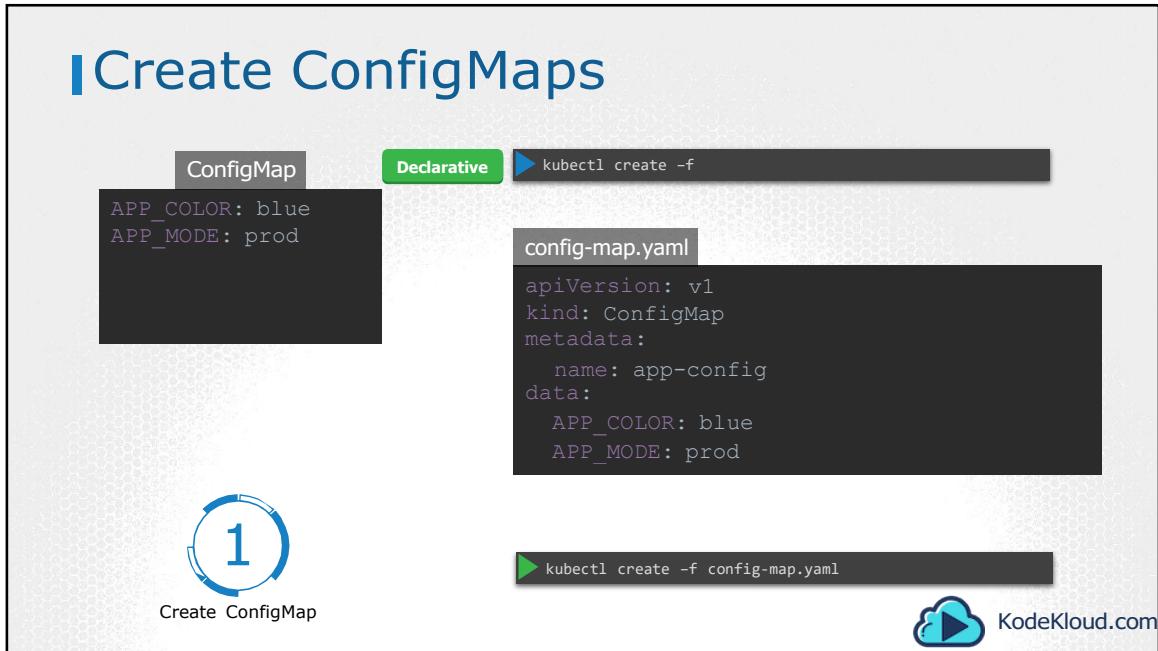
To the left of the terminal windows, there is a large blue circle containing the number "1". Below this circle, the text "Create ConfigMap" is displayed.

In the bottom right corner of the slide, there is a small KodeKloud logo.

The command is followed by the config name and the option –from-literal. The from literal option is used to specify the key value pairs in the command itself. In this example, we are creating a configmap by the name app-config, with a key value pair APP\_COLOR=blue. If you wish to add additional key value pairs, simply specify the from literal options multiple times. However, this will get complicated when you have too many configuration items.

Another way to input the configuration data is through a file. Use the –from-file option to specify a path to the file that contains the required data. The data from this file is read and stored under the name of the file

# Create ConfigMaps



Let us now look at the declarative approach. For this we create a definition file, just like how we did for the pod. The file has `apiVersion`, `kind`, `metadata` and instead of `spec`, here we have “`data`”. The `apiVersion` is `v1`, `kind` is `ConfigMap`. Under `metadata` specify the name of the configmap. We will call it `app-config`. Under `data` add the configuration data in a key-value format.

Run the `kubectl create` command and specify the configuration file name.

## Create ConfigMaps

app-config

```
APP_COLOR: blue  
APP_MODE: prod
```

mysql-config

```
port: 3306  
max_allowed_packet: 128M
```

redis-config

```
port: 6379  
rdb-compression: yes
```



Create ConfigMap



KodeKloud.com

So that creates the app-config config map with the values we specified. You can create as many configmaps as you need in the same way for various different purposes. Here I have one for my application, other for mysql and another one for redis. So it is important to name the config maps appropriately as you will be using these names later while associating it with PODs.

## I View ConfigMaps

```
▶ kubectl get configmaps
```

NAME	DATA	AGE
app-config	2	3s

```
▶ kubectl describe configmaps
```

Name:	app-config
Namespace:	default
Labels:	<none>
Annotations:	<none>
Data	====
APP_COLOR:	.....
blue	
APP_MODE:	.....
prod	
Events:	<none>



KodeKloud.com

To view the configmaps, run the `kubectl get configmaps` command. This lists the newly created configmap named `app-config`. The `describe configmaps` command lists the configuration data as well under the Data section.

## IConfigMap in Pods

pod-definition.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
  labels:
    name: simple-webapp-color
spec:
  containers:
    - name: simple-webapp-color
      image: simple-webapp-color
      ports:
        - containerPort: 8080
  envFrom:
    - configMapRef:
        name: app-config
```

▶ kubectl create -f pod-definition.yaml

config-map.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_COLOR: blue
  APP_MODE: prod
```

Hello from DESKTOP-4CJKELD!

com

Now that we have the configmap created let us proceed with Step 2 – Configuring it with a POD. Here I have a simple pod definition file that runs my application simple web application.

To inject an environment variable, add a new property to the container called envFrom. The envFrom property is a list, so we can pass as many environment variables as required. Each item in the list corresponds to a configMap item. Specify the name of the configmap we created earlier. This is how we inject a specific configmap from the ones we created before. Creating the pod definition file now creates a web application with a blue background.

# ConfigMap in Pods

```
envFrom:  
- configMapRef:  
  name: app-config
```

ENV

SINGLE ENV

```
env:  
- name: APP_COLOR  
  valueFrom:  
    configMapKeyRef:  
      name: app-config  
      key: APP_COLOR
```

```
volumes:  
- name: app-config-volume  
configMap:  
  name: app-config
```

VOLUME



KodeKloud.com

What we just saw was using configMaps to inject environment variables. There are other ways to inject configuration data into PODs. You can inject a single environment variable or inject the whole configuration data as files in a volume.

## Practice Test



Check Link  
below!

KodeKloud.com

Access Test Here: <https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743656>

## I References

<https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/>

<https://kubernetes.io/docs/tutorials/configuration/>



## Course Objectives

### Core Concepts

#### Configuration

##### ConfigMaps

##### SecurityContexts

##### Resource Requirements

##### Secrets

##### ServiceAccounts

### Multi-Container Pods

### Observability

### Pod Design

### Services & Networking

### State Persistence



KodeKloud.com

The next section on Configuration covers topics like ConfigMaps, SecurityContexts, Resource Requirements, secrets and service accounts.

85

# Kubernetes Secrets



KodeKloud.com

# I Web-MySQL Application

```
app.py
import os
from flask import Flask

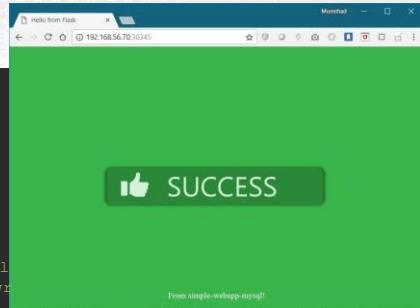
app = Flask(__name__)

@app.route("/")
def main():

    mysql.connector.connect(host='mysql', database='mysql',
                           user='root', password='password')

    return render_template('hello.html', color=fetchcolor())

if __name__ == "__main__":
    app.run(host="0.0.0.0", port="8080")
```



KodeKloud.com

Here we have a simple python web application that connects to a mysql database. On success the application displays a successful message.

# I Web-MySQL Application

app.py

```
import os
from flask import Flask

app = Flask(__name__)

@app.route("/")
def main():

    mysql.connector.connect(host='mysql', database='mysql',
                           user='root', password='paswrd')

    return render_template('hello.html', color=fetchcolor())

if __name__ == "__main__":
    app.run(host="0.0.0.0", port="8080")
```



KodeKloud.com

If you look closely into the code, you will see the hostname, username and password hardcoded. This is of-course not a good idea.

# Web-MySQL Application

app.py

```
import os
from flask import Flask

app = Flask(__name__)

@app.route("/")
def main():

    mysql.connector.connect(host='mysql', database='mysql',
                            user='root', password='paswrd')

    return render_template('hello.html', color=fetchcolor())

if __name__ == "__main__":
    app.run(host="0.0.0.0", port="8080")
```

config-map.yaml

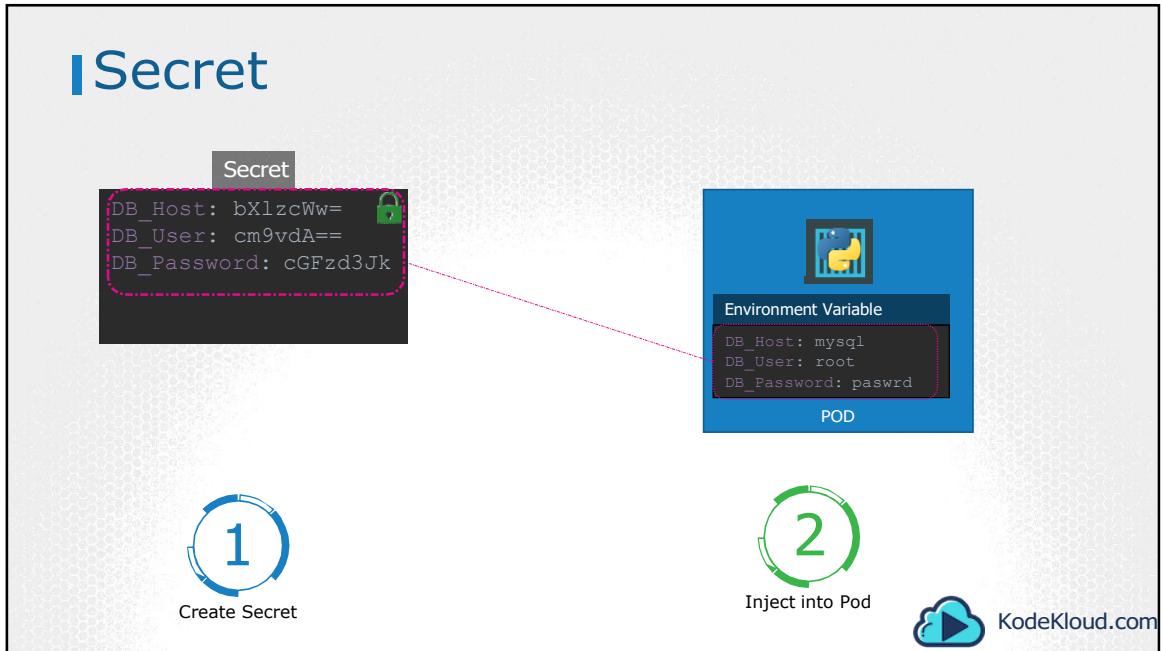
```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  DB_Host: mysql
  DB_User: root
  DB_Password: paswrd
```



KodeKloud.com

As we learned in the previous lecture, one option would be to move these values into a configMap. The configMap stores configuration data in plain text, so while it would be OK to move the hostname and username into a configMap, it is definitely not the right place to store a password.

# I Secret



This is where secrets come in. Secrets are used to store sensitive information, like passwords or keys. They are similar to configMaps, except that they are stored in an encoded or hashed format. As with configMaps, there are two steps involved in working with Secrets. First, create the secret and second inject it into Pod.

## Create Secrets

The diagram illustrates two methods for creating Kubernetes secrets:

- Imperative:** Represented by a black box containing command-line text: `kubectl create secret generic`. Above this box is a pink button labeled "Secret". Inside the box, there is sample configuration:

```
DB_Host: mysql
DB_User: root
DB_Password: paswrd
```
- Declarative:** Represented by a green box containing command-line text: `kubectl create -f`. Above this box is a green button labeled "Declarative".

At the bottom left, there is a blue circular icon with the number "1" and the text "Create Secret". At the bottom right, there is a cloud icon with the text "KodeKloud.com".

There are two ways of creating a secret. The imperative way - without using a Secret definition file and the Declarative way by using a Secret Definition file.

With the Imperative method you can directly specify the key value pairs in the command line itself. To create a secret of the given values, run the `kubectl create secret generic` command.

# Create Secrets

The diagram illustrates the creation of a Kubernetes secret named "app-secret" with three key-value pairs: DB\_Host=mysql, DB\_User=root, and DB\_Password=paswrd. It shows four command-line examples using the --from-literal option:

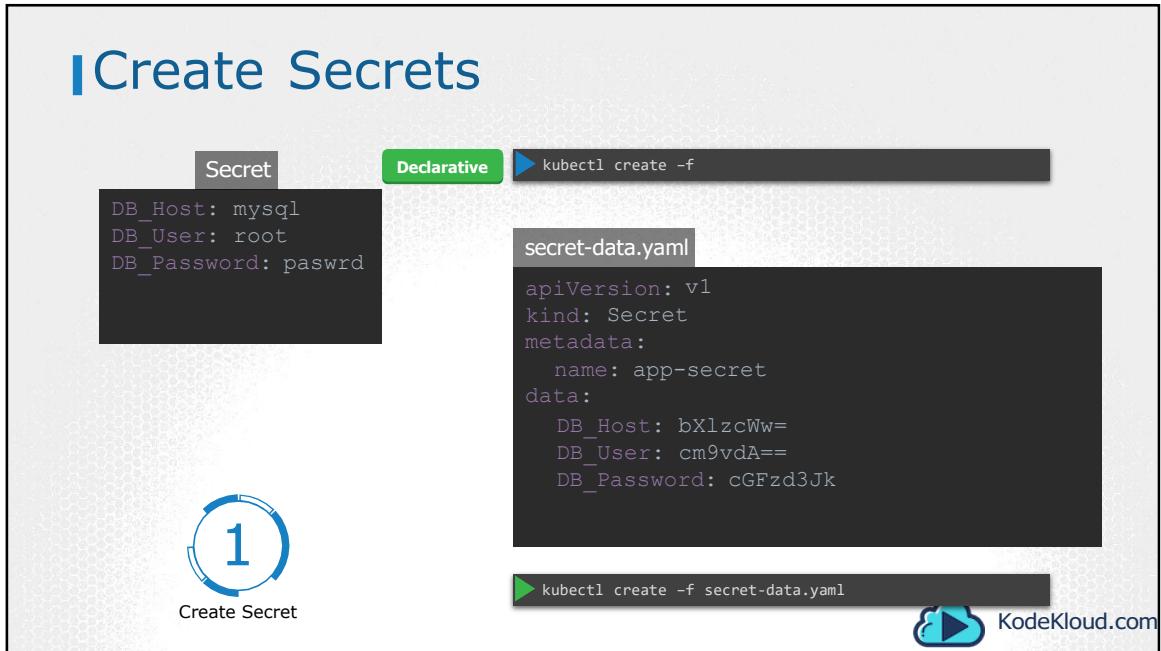
- Imperative:** `kubectl create secret generic <secret-name> --from-literal=<key>=<value>`
- Example 1:** `kubectl create secret generic app-secret --from-literal=DB_Host=mysql --from-literal=DB_User=root --from-literal=DB_Password=paswrd`
- Example 2:** `kubectl create secret generic <secret-name> --from-file=<path-to-file>`
- Example 3:** `kubectl create secret generic app-secret --from-file=app_secret.properties`

A large blue number "1" is centered on the left, with the text "Create Secret" below it. The KodeKloud.com logo is in the bottom right corner.

The command is followed by the secret name and the option –from-literal. The from literal option is used to specify the key value pairs in the command itself. In this example, we are creating a secret by the name app-secret, with a key value pair DB\_Host=mysql. If you wish to add additional key value pairs, simply specify the from literal options multiple times.

However, this could get complicated when you have too many secrets to pass in. Another way to input the secret data is through a file. Use the –from-file option to specify a path to the file that contains the required data. The data from this file is read and stored under the name of the file.

## Create Secrets



Let us now look at the declarative approach. For this we create a definition file, just like how we did for the ConfigMap. The file has apiVersion, kind, metadata and data. The apiVersion is v1, kind is Secret. Under metadata specify the name of the secret. We will call it app-secret. Under data add the secret data in a key-value format.

However, one thing we discussed about secrets was that they are used to store sensitive data and are stored in an encoded format. Here we have specified the data in plain text, which is not very safe. So, while creating a secret with the declarative approach, you must specify the secret values in a hashed format.

So you must specify the data in an encoded form like this. But how do you convert the data from plain text to an encoded format?

## |Encode Secrets

```
DB_Host: mysql  
DB_User: root  
DB_Password: paswrd
```



```
DB_Host: bXlzcWw=  
DB_User: cm9vdA==  
DB_Password: cGFzd3Jk
```

```
▶ echo -n 'mysql' | base64  
bXlzcWw=
```

```
▶ echo -n 'root' | base64  
cm9vdA==
```

```
▶ echo -n 'paswrd' | base64  
cGFzd3Jk
```

om

But how do you convert the data from plain text to an encoded format? On a linux host run the command echo –n followed by the text you are trying to convert, which is mysql in this case and pipe that to the base64 utility.

# I View Secrets

```
▶ kubectl get secrets
NAME          TYPE        DATA  AGE
app-secret    Opaque      3     10m
default-token-mvtkv  kubernetes.io/service-account-token 3     2h

▶ kubectl describe secrets
Name:         app-secret
Namespace:   default
Labels:      <none>
Annotations: <none>

Type:  Opaque

Data
=====
DB_Host:    10 bytes
DB_Password: 6 bytes
DB_User:    4 bytes

▶ kubectl get secret app-secret -o yaml
apiVersion: v1
data:
  DB_Host: bXlzcWw=
  DB_Password: cGFzd3Jk
  DB_User: cm9vdA==
kind: Secret
metadata:
  creationTimestamp: 2018-10-18T10:01:12Z
  labels:
    name: app-secret
    name: app-secret
    namespace: default
  uid: be96e989-d2bc-11e8-a545-080027931072
type: Opaque
```



KodeKloud.com

To view secrets run the `kubectl get secrets` command. This lists the newly created secret along with another secret previously created by kubernetes for its internal purposes.

To view more information on the newly created secret, run the `kubectl describe secret` command. This shows the attributes in the secret, but hides the value themselves.

To view the values as well, run the `kubectl get secret` command with the output displayed in a YAML format using the `-o` option. You can now see the hashed values as well.

## I Decode Secrets

```
DB_Host: mysql  
DB_User: root  
DB_Password: paswrd
```

```
DB_Host: bXlzcWw=  
DB_User: cm9vdA==  
DB_Password: cGFzd3Jk
```



```
▶ echo -n 'bXlzcWw=' | base64 --decode  
mysql
```

```
▶ echo -n 'cm9vdA==' | base64 --decode  
root
```

```
▶ echo -n 'cGFzd3Jk' | base64 --decode  
paswrd
```

om

How do you decode the hashed values? Use the same base64 command you used in linux to encode it, but this time add a decode option to it.

# Secrets in Pods

pod-definition.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
  labels:
    name: simple-webapp-color
spec:
  containers:
    - name: simple-webapp-color
      image: simple-webapp-color
      ports:
        - containerPort: 8080
      envFrom:
        - secretRef:
            name: app-secret
```

▶ kubectl create -f pod-definition.yaml

secret-data.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
data:
  DB_Host: bXlzcWw=
  DB_User: cm9vdA==
  DB_Password: cGFzd3Jk
```



Now that we have the secret created let us proceed with Step 2 – Configuring it with a POD. Here I have a simple pod definition file that runs my application.

To inject an environment variable, add a new property to the container called `envFrom`. The `envFrom` property is a list, so we can pass as many environment variables as required. Each item in the list corresponds to a `Secret` item. Specify the name of the secret we created earlier. Creating the POD definition file now makes the data in the secret available as environment variables for the application.

## I Secrets in Pods

```
envFrom:  
- secretRef:  
  name: app-config
```

ENV

SINGLE ENV

```
env:  
- name: DB_Password  
  valueFrom:  
    secretKeyRef:  
      name: app-secret  
      key: DB_Password
```

```
volumes:  
- name: app-secret-volume  
secret:  
  secretName: app-secret
```

VOLUME



KodeKloud.com

What we just saw was injecting secrets as environment variables into the PODs. There are other ways to inject secret into PODs. You can inject as single environment variables or inject the whole secret as files in a volume.

## I Secrets in Pods as Volumes

```
volumes:  
- name: app-secret-volume  
  secret:  
    secretName: app-secret
```

VOLUME

```
▶ ls /opt/app-secret-volumes  
DB_Host      DB_Password  DB_User
```

```
▶ cat /opt/app-secret-volumes/DB_Password  
paswrd
```

Inside the Container



If you were to mount the secret as a volume in the Pod, each attribute in the secret is created as a file with the value of the secret as its content. In this case, since we have 3 attributes in our secret, 3 files are created. And if we look at the contents of the DB\_password file, we see the password inside it. That's it for this lecture, head over to the coding exercises and practice working with secrets.

## Practice Test



Check Link  
below!

KodeKloud.com

Access Test Here: <https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743657>

## I References

<https://kubernetes.io/docs/concepts/configuration/secret/>

<https://kubernetes.io/docs/tasks/inject-data-application/distribute-credentials-secure/>





Hello and welcome to this lecture. In this lecture we will talk about Security Contexts in Kubernetes. But before we get into that, it is important to have some knowledge about Security in Docker. If you are familiar with Security in Docker, feel free to skip this lecture and head over to the next.

I

102

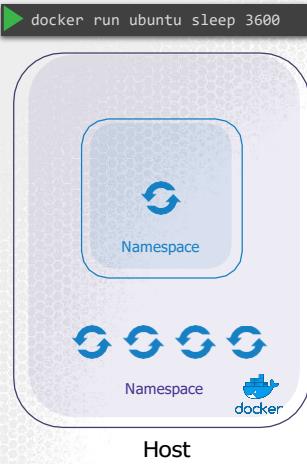
# Security in Docker



KodeKloud.com

Hello and welcome to this lecture. In this lecture we will talk about Security Contexts in Kubernetes. But before we get into that, it is important to have some knowledge about Security in Docker. If you are familiar with Security in Docker, feel free to skip this lecture and head over to the next.

# I Security



KodeKloud.com

In this lecture we will look at the various concepts related to security in Docker. Let us start with a host with Docker installed on it. This host has a set of its own processes running such as a number of operating system processes, the docker daemon itself, the SSH server etc.

We will now run an Ubuntu docker container that runs a process that sleeps for an hour.

We have learned that unlike virtual machines containers are not completely isolated from their host. Containers and the hosts share the same kernel. Containers are isolated using namespaces in Linux. The host has a namespace and the containers have their own namespace. All the processes run by the containers are in fact run on the host itself, but in their own namespaces.

# I Security

```
ps aux
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.0	4528	828	?	Ss	03:06	0:00	sleep 3600



Container



KodeKloud.com

As far as the docker container is concerned, it is in its own namespace and it can see its own processes only, cannot see anything outside of it or in any other namespace. So when you list the processes from within the docker container you see the sleep process with a process ID of 1.

# ISecurity

```
ps aux
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
project	3720	0.1	0.1	95500	4916	?	R	06:06	0:00	sshd: project@pts/0
project	3725	0.0	0.1	95196	4132	?	S	06:06	0:00	sshd: project@notty
project	3727	0.2	0.1	21352	5340	pts/0	Ss	06:06	0:00	-bash
root	3802	0.0	0.0	8924	3616	?	S1	06:06	0:00	docker-containerd-
shim	-namespace	m								
root	3816	1.0	0.0	4528	828	?	Ss	06:06	0:00	sleep 3600



For the docker host, all processes of its own as well as those in the child namespaces are visible as just another process in the system. So when you list the processes on the host you see a list of processes including the sleep command, but with a different process ID. This is because the processes can have different process IDs in different namespaces and that's how Docker isolates containers within a system. So that's process isolation.

## I Security - Users

The diagram illustrates the Docker container structure. On the left, a large rounded rectangle labeled 'Host' contains a smaller rounded rectangle labeled 'Namespace'. Inside the 'Namespace' are three user icons. Below the 'Namespace' is a 'docker' icon. To the right of the host structure are four terminal windows showing command outputs.

```
ps aux
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
project	3720	0.1	0.1	95500	4916	?	R	06:06	0:00	sshd: project@pts/0
project	3725	0.0	0.1	95196	4132	?	S	06:06	0:00	sshd: project@notty
project	3727	0.2	0.1	21352	5340	pts/0	Ss	06:06	0:00	-bash
root	3802	0.0	0.0	8924	3616	?	S1	06:06	0:00	docker-containe
shim	-namespace	m								
root	3816	1.0	0.0	4528	828	?	Ss	06:06	0:00	sleep 3600

```
ps aux
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.0	4528	828	?	Ss	03:06	0:00	sleep 3600

```
docker run --user=1001 ubuntu sleep 3600
```

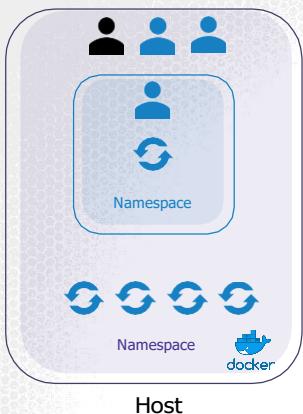
```
ps aux
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
1001	1	0.0	0.0	4528	828	?	Ss	03:06	0:00	sleep 3600

ud.com

Let us now look at users in context of security. The docker host has a set of users, a root user as well as a number of non-root users. By default docker runs processes within containers as the root user. This can be seen in the output of the commands we ran earlier. Both within the container and outside the container on the host, the process is run as the root user. Now if you do not want the process within the container to run as the root user, you may set the user using the user option with the docker run command and specify the new user ID. You will see that the process now runs with the new user id.

## I Security - Users



```
Dockerfile
FROM ubuntu

USER 1001

▶ docker build -t my-ubuntu-image .

▶ docker run my-ubuntu-image sleep 3600

▶ ps aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
USER 1001      1  0.0  0.0  4528  828 ?  Ss 03:06 0:00 sleep 3600
```



KodeKloud.com

Another way to enforce user security is to have this defined in the Docker image itself at the time of creation. For example, we will use the default ubuntu image and set the user ID to 1000 using the USER instruction. Then build the custom image. We can now run this image using without specifying the user ID and the process will be run with the user id 1000.

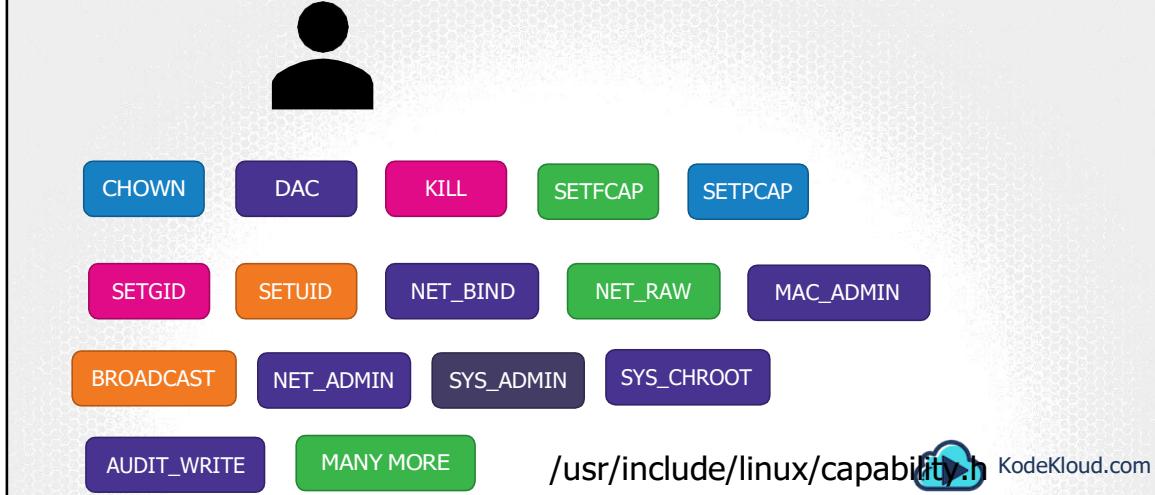
## I Security - Users



KodeKloud.com

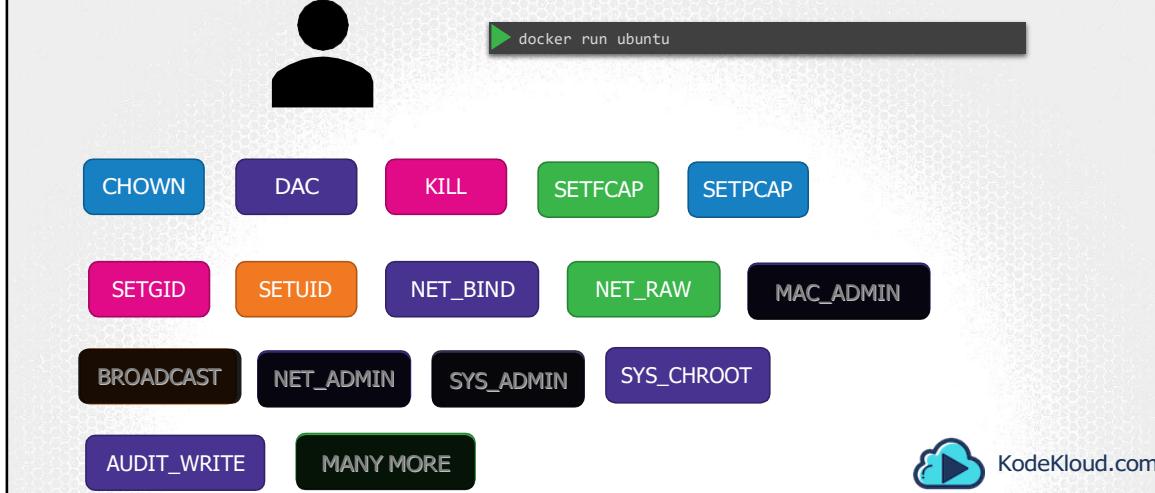
Let us take a step back. What happens when you run containers as the root user? Is the root user within the container the same as the root user on the host? Can the process inside the container do anything that the root user can do on the system? If so isn't that dangerous? Well, docker implements a set of security features that limits the abilities of the root user within the container. So the root user within the container isn't really like the root user on the host.

# Linux Capabilities



Docker uses Linux Capabilities to implement this. As we all know the root user is the most powerful user on a system. The root user can literally do anything. And so does a process run by the root user. It has unrestricted access to the system. From modifying files and permissions on files, Access Control, creating or killing processes, setting group id or user ID, performing network related operations such as binding to network ports, broadcasting on a network, controlling network ports; system related operations like rebooting the host, manipulating system clock and many more. All of these are the different capabilities on a Linux system and you can see a full list at this location. You can now control and limit what capabilities are made available to a process.

# Linux Capabilities



By default Docker runs a container with a limited set of capabilities. And so the processes running within the container do not have the privileges to say, reboot the host or perform operations that can disrupt the host or other containers running on the same host. In case you wish to override this behavior and enable all privileges to the container use the privileged flag.

# Linux Capabilities



```
▶ docker run --cap-add MAC_ADMIN ubuntu
```

CHOWN

DAC

KILL

SETFCAP

SETPCAP

SETGID

SETUID

NET\_BIND

NET\_RAW

MAC\_ADMIN

BROADCAST

NET\_ADMIN

SYS\_ADMIN

SYS\_CHROOT

AUDIT\_WRITE

MANY MORE



KodeKloud.com

If you wish to override this behavior and provide additional privileges than what is available use the cap-add option in the docker run command.

# Linux Capabilities



```
▶ docker run --cap-drop KILL      ubuntu
```

CHOWN

DAC

KILL

SETFCAP

SETPCAP

SETGID

SETUID

NET\_BIND

NET\_RAW

MAC\_ADMIN

BROADCAST

NET\_ADMIN

SYS\_ADMIN

SYS\_CHROOT

AUDIT\_WRITE

MANY MORE



KodeKloud.com

Similarly you can drop privileges as well using the cap drop option.

# Linux Capabilities



```
▶ docker run --privileged ubuntu
```

CHOWN

DAC

KILL

SETFCAP

SETPCAP

SETGID

SETUID

NET\_BIND

NET\_RAW

MAC\_ADMIN

BROADCAST

NET\_ADMIN

SYS\_ADMIN

SYS\_CHROOT

AUDIT\_WRITE

MANY MORE



KodeKloud.com

Or in case you wish to run the container with all privileges enabled, use the privileged flag. Well that's it on Docker Security for now. I will see you in the next lecture.



Hello and welcome to this lecture on Security Contexts in Kubernetes. My name is Mumshad Mannambeth and we are going through the Certified Kubernetes Applications Developer Course.

# Kubernetes Security Contexts



# I Container Security

```
▶ docker run --user=1001 ubuntu sleep 3600
```

```
▶ docker run --cap-add MAC_ADMIN ubuntu
```



MAC\_ADMIN



KodeKloud.com

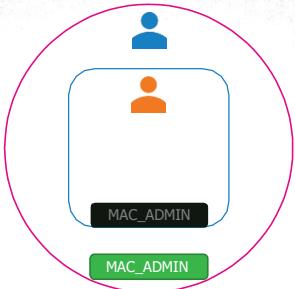
As we saw in the previous lecture, when you run a Docker Container you have the option to define a set of security standards, such as the ID of the user used to run the container, the Linux capabilities that can be added or removed from the container etc. These can be configured in Kubernetes as well.

# I Kubernetes Security



As you know already, in Kubernetes containers are encapsulated in PODs. You may chose to configure the security settings at a container level....

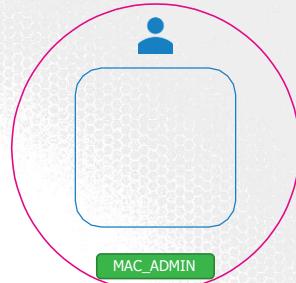
## I Kubernetes Security



... or at a POD level. If you configure it at a POD level, the settings will carry over to all the containers within the POD. If you configure it at both the POD and the Container, the settings on the container will override the settings on the POD.

# Security Context

```
apiVersion: v1
kind: Pod
metadata:
  name: web-pod
spec:
  securityContext:
    runAsUser: 1000
  containers:
    - name: ubuntu
      image: ubuntu
      command: ["sleep", "3600"]
```



KodeKloud.com

Let us start with a POD definition file. This pod runs an ubuntu image with the sleep command. To configure security context on the container, add a field called securityContext under the spec section of the pod. Use the runAsUser option to set the user ID for the POD.

# I Security Context

```
apiVersion: v1
kind: Pod
metadata:
  name: web-pod
spec:
  containers:
    - name: ubuntu
      image: ubuntu
      command: ["sleep", "3600"]
      securityContext:
        runAsUser: 1000
        capabilities:
          add: ["MAC_ADMIN"]
```

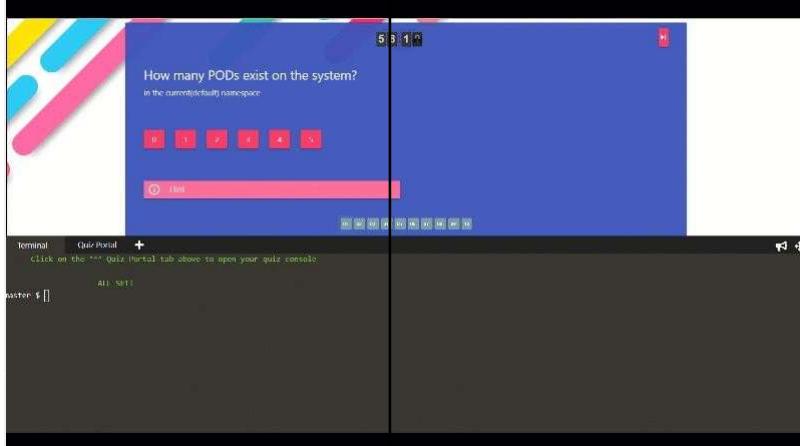


To set the same configuration on the container level, move the whole section under the container specification like this.

To add capabilities use the capabilities option and specify a list of capabilities to add to the POD.

Well that's all on Security Contexts. Head over to the coding exercises section and practice viewing, configuring and troubleshooting issues related to Security contexts in Kubernetes. That's it for now and I will see you in the next lecture.

## Practice Test



Check Link  
below!

KodeKloud.com

Access Test Here: <https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743658>

## I References

<https://kubernetes.io/docs/tasks/configure-pod-container/security-context/>

<https://kubernetes.io/docs/concepts/policy/pod-security-policy/>

<https://kubernetes.io/blog/2016/08/security-best-practices-kubernetes-deployment/>



## Course Objectives

- ✓ Core Concepts
- Configuration
  - ✓ ConfigMaps
  - ✓ SecurityContexts
  - Resource Requirements
- Multi-Container Pods
- Observability
- Pod Design
- Services & Networking
- State Persistence

✓ Secrets

● ServiceAccounts



KodeKloud.com

Hello and welcome to this lecture. In this lecture we will talk about Service Accounts in Kubernetes.

# SERVICE ACCOUNTS



The concept of service accounts is linked to other security related concepts in kubernetes such as Authentication, Authorization, Role based access controls etc.

**Certified Kubernetes  
Administrator  
(CKA)**

12% - Security

- Know how to configure authentication and authorization.
- Understand Kubernetes security primitives.
- Know to configure network policies.
- Create and manage TLS certificates for cluster components.
- Work with images securely.
- Define security contexts.
- Secure persistent key value store.

**Certified Kubernetes  
Application Developer  
(CKAD)**

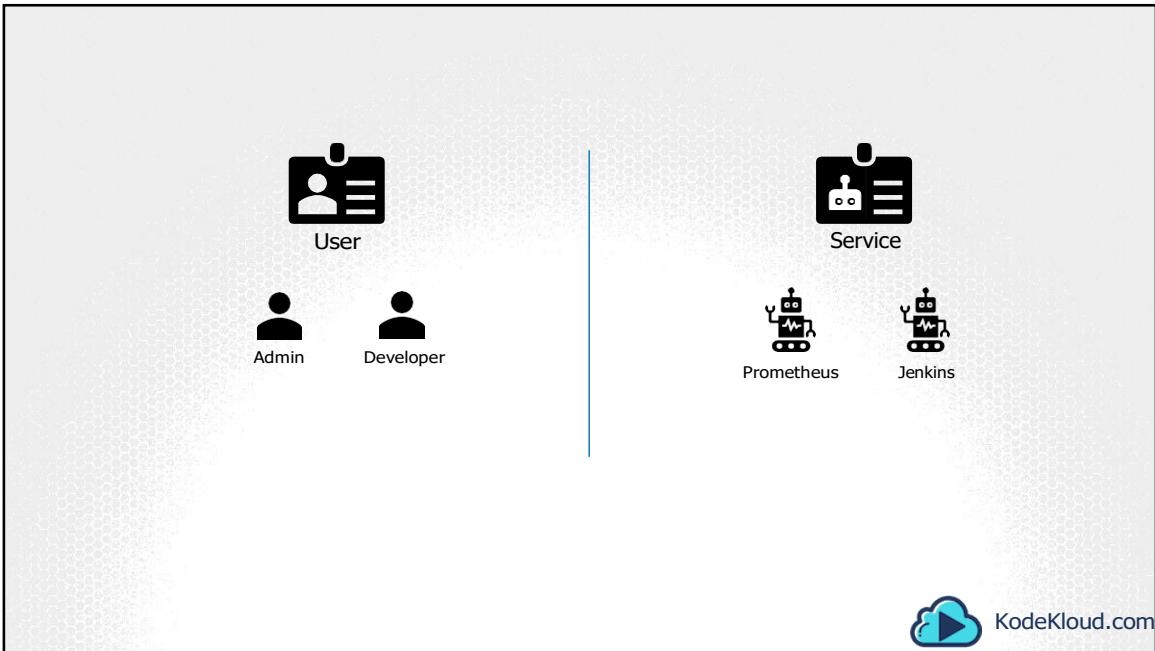
18% - Configuration

- Understand ConfigMaps
- Understand SecurityContexts
- Define an application's resource requirements
- Create & consume Secrets
- Understand ServiceAccounts



KodeKloud.com

However, as part of the Kubernetes for the Application Developers exam curriculum, you only need to know how to work with Service Accounts. We have detailed sections covering other concepts in security in the Kubernetes Administrators course.



So there are two types of accounts in Kubernetes. A user account and a service account. As you might already know, the user account is used by humans. And service accounts are used by machines. A user account could be for an administrator accessing the cluster to perform administrative tasks, a developer accessing the cluster to deploy applications etc. A service account, could be an account used by an application like Prometheus uses a service account to poll the kubernetes API for performance metrics. An automated build tool like Jenkins uses service accounts to deploy applications on the kubernetes cluster.

The image shows a screenshot of the 'My Kubernetes Dashboard' application. The dashboard has a dark blue header with the title 'My Kubernetes Dashboard' and a sub-header 'https://172.18.0.2:70443/api'. Below the header is a table titled 'POD STATUS LIST' with columns: Name, State, Containers, Service Account, and IP. The table lists five pods:

Name	State	Containers	Service Account	IP
dev	Running	1	default	10.244.1.219
blue-SVC-000001-dash	Running	1	default	10.244.1.221
blue-SVC-000001-Apache	Running	1	default	10.244.1.216
blue-SVC-000001-SaS	Running	1	default	10.244.1.222
blue-SVC-000001-Redis	Running	1	default	10.244.1.218

To the right of the dashboard is a diagram titled 'Kubernetes Cluster'. It features a central box labeled 'kube-api' containing three server icons. A dashed arrow points from the 'kube-api' box down to the 'Kubernetes Cluster' box, which contains the same three server icons. Above the 'kube-api' box is a small icon of a person with a key and a list.

KodeKloud.com

Let's take an example. I have built a simple kubernetes dashboard application named, my-kubernetes-dashboard. It's a simple application built in Python and all that it does when deployed is retrieve the list of PODs on a kubernetes cluster by sending a request to the kubernetes API and display it on a web page. In order for my application to query the kubernetes API, it has to be authenticated. For that we use a service account.

```
▶ kubectl create serviceaccount dashboard-sa
serviceaccount "dashboard-sa" created

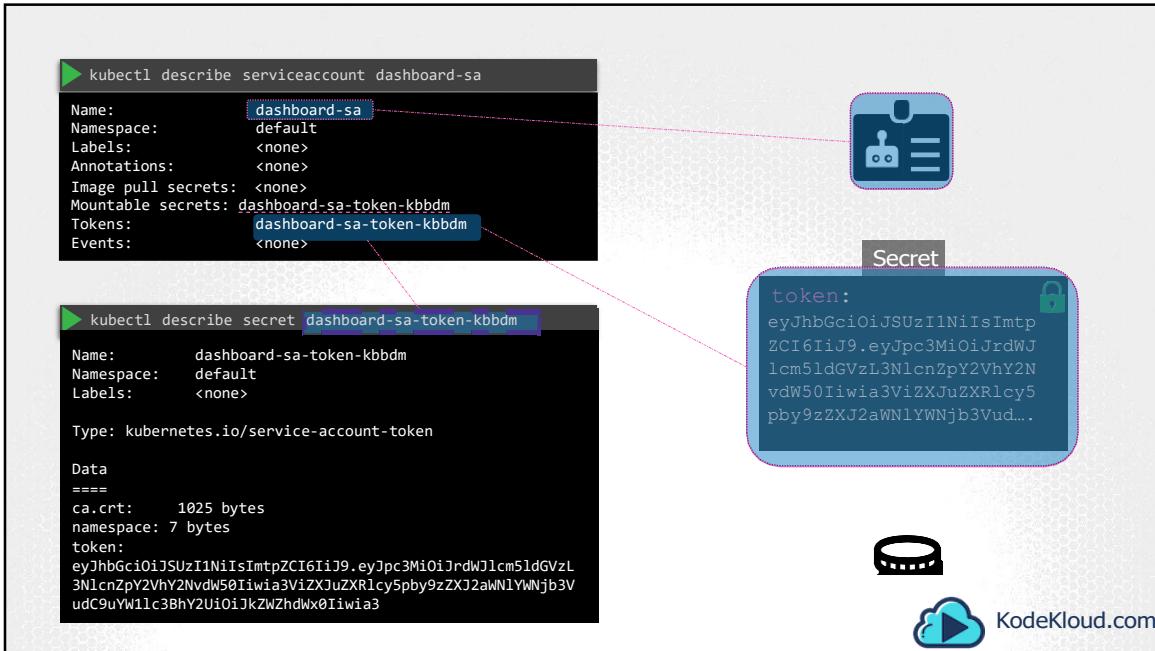
▶ kubectl get serviceaccount
NAME      SECRETS   AGE
default   1          218d
dashboard-sa   1          4d

▶ kubectl describe serviceaccount dashboard-sa
Name:           dashboard-sa
Namespace:      default
Labels:          <none>
Annotations:    <none>
Image pull secrets:  <none>
Mountable secrets:  dashboard-sa-token-kbbdm
Tokens:          dashboard-sa-token-kbbdm
Events:          <none>
```

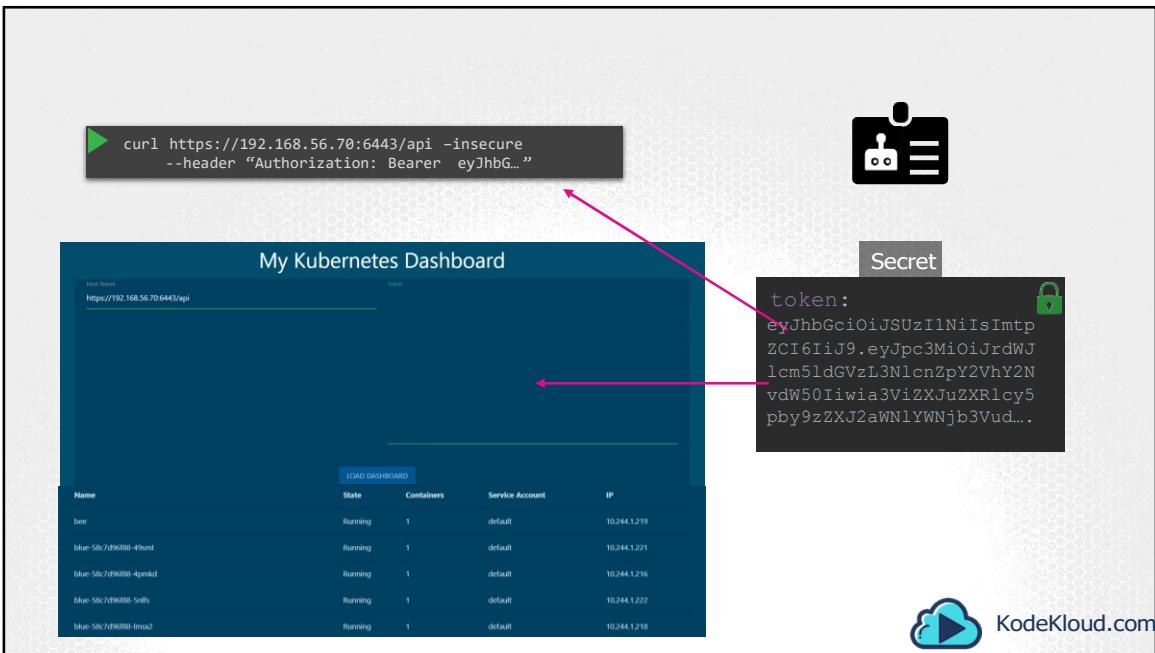


To create a service account run the command `kubectl create service account` followed by the account name, which is `dashboard-sa` in this case. To view the service accounts run the `kubectl get serviceaccount` command. This will list all the service accounts.

When the service account is created, it also creates a token automatically. The service account token is what must be used by the external application while authenticating to the Kubernetes API. The token, however, is stored as a secret object. In this case its named `dashboard-sa-token-kbbdm`.

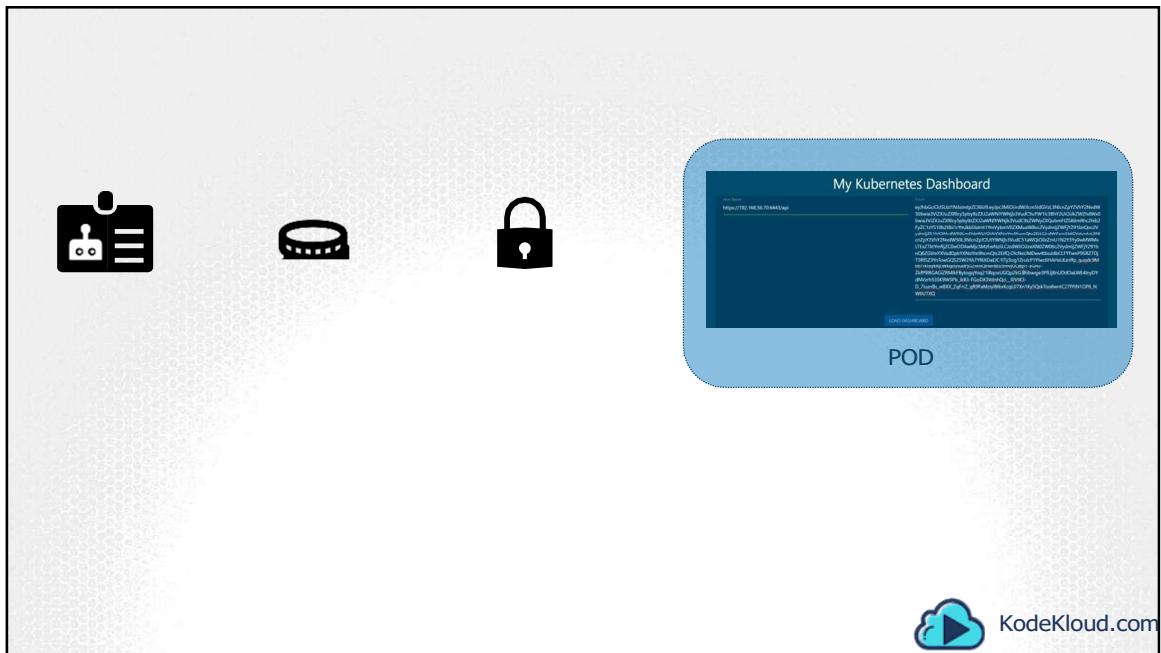


So when a service account is created, it first creates the service account object and then generates a token for the service account. It then creates a secret object and stores that token inside the secret object. The secret object is then linked to the service account. To view the token, view the secret object by running the command `kubectl describe secret`.

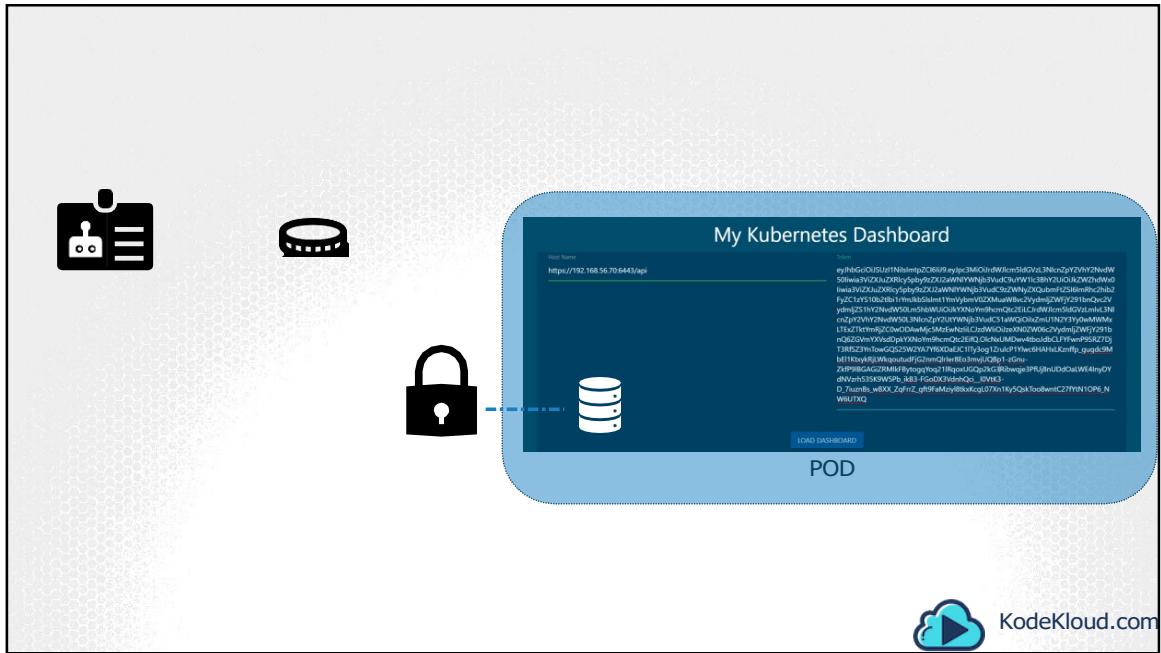


This token can then be used as an authentication bearer token while making a rest call to the kubernetes API. For example in this simple example using curl you could provide the bearer token as an Authorization header while making a rest call to the kubernetes API.

In case of my custom dashboard application, copy and paste the token into the tokens field to authenticate the dashboard application.



So, that's how you create a new service account and use it. You can create a service account, assign the right permissions using Role based access control mechanisms (which is out of scope for this course) and export your service account tokens and use it to configure your third party application to authenticate to the kubernetes API. But what if your third party application is hosted on the kubernetes cluster itself. For example, we can have our custom-kubernetes-dashboard or the Prometheus application used to monitor kubernetes, deployed on the kubernetes cluster itself.



In that case, this whole process of exporting the service account token and configuring the third party application to use it can be made simple by automatically mounting the service token secret as a volume inside the POD hosting the third party application. That way the token to access the kubernetes API is already placed inside the POD and can be easily read by the application.

```

▶ kubectl get serviceaccount
NAME      SECRETS   AGE
default   1          218d
dashboard-sa   1          4d

▶ kubectl get pod
  NAME           READY   STATUS    RESTARTS   AGE
  my-kubernetes-dashboard   1/1     Running   0          4d

▶ kubectl get pod my-kubernetes-dashboard
Name:         my-kubernetes-dashboard
Namespace:    default
Annotations: <none>
Status:       Running
IP:          10.244.0.15
Containers:
  nginx:
    Image:      my-kubernetes-dashboard
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-j4hkv (ro)
    Conditions:
      Type        Status
    Volumes:
      default-token-j4hkv:
        Type:      Secret (a volume populated by a Secret)
        SecretName: default-token-j4hkv
        Optional:   false

```

```

pod-definition.yml
apiVersion: v1
kind: Pod
metadata:
  name: my-kubernetes-dashboard
spec:
  containers:
    - name: my-kubernetes-dashboard
      image: my-kubernetes-dashboard

```


KodeKloud.com

If you go back and look at the list of service accounts, you will see that there is a default service account that exists already. For every namespace in kubernetes a service account named default is automatically created. Each namespace has its own default service account.

Whenever a POD is created the default service account and its token are automatically mounted to that POD as a volume mount. For example, we have a simple pod definition file that creates a POD using my `custom kubernetes dashboard` image. We haven't specified any secrets or volume mounts. However when the pod is created, if you look at the details of the pod, by running the `kubectl describe pod` command, you see that a volume is automatically created from the secret named `default-token-j4hkv`, which is in fact the secret containing the token for the default service account. The secret token is mounted at location `/var/run/secrets/kubernetes.io/serviceaccount` inside the pod. So from inside the pod if you run the `ls` command

```

▶ kubectl describe pod my-kubernetes-dashboard
Name:      my-kubernetes-dashboard
Namespace: default
Annotations: <none>
Status:    Running
IP:        10.244.0.15
Containers:
  nginx:
    Image:      my-kubernetes-dashboard
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-j4hkv (ro)
    Conditions:
      Type          Status
    Volumes:
      default-token-j4hkv:
        Type:       Secret (a volume populated by a Secret)
        SecretName: default-token-j4hkv
        Optional:   false

```

```

▶ kubectl exec -it my-kubernetes-dashboard ls /var/run/secrets/kubernetes.io/serviceaccount
ca.crt namespace token

```

```

▶ kubectl exec -it my-kubernetes-dashboard cat /var/run/secrets/kubernetes.io/serviceaccount/token
eyJhbGciOiJSUzIiNiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcmb5ldGVzL3NlcnZpY2VhY2Nvdw50Iiwiia3VizXJuZXRLcy5pb9zzXJ2awN1YWNb3V
udc9uYW11c3BhY2UiOiJKzWZhdWx0Iiwiia3VizXJuZXRLcy5pb9zzXJ2awN1YWNb3Vudc9zZWMyZXQubmfzSI6InrlZmF1bHQtdG9rZl4tajRoa3Y
iLCJrdWJlcmb5ldGVzLmlvL3NlcnZpY2VhY2Nvdw50L3NlcnZpY2utYWNb3Vudc5uYW11IjoizGvmyXVsdcIsImt1YmVybmv0ZXMuaw8vc2VydmljZW
jY291bnQvc2VydmljZS1hY2Nvdw50LnVpZCI6IjcxZGM4YWEltU2MGMtMTFl0C04YmI0LTA4MDAyNzkzMTA3MiIsInN1YiI6InN5c3R1bTpzZXJ2awN
com

```

If you list the contents of the directory inside the pod, you will see the secret mounted as 3 separate files. The one with the actual token is the file named token. If you list the contents of that file you will see the token to be used for accessing the kubernetes API. Now remember that the default service account is very much restricted. It only has permission to run basic kubernetes API queries.

```

▶ kubectl get serviceaccount
  NAME      SECRETS   AGE
default      1        218d
dashboard-sa 1        4d

▶ kubectl describe pod my-kubernetes-dashboard
Name:           my-kubernetes-dashboard
Namespace:      default
Annotations:    <none>
Status:         Running
IP:             10.244.0.15
Containers:
  nginx:
    Image:      my-kubernetes-dashboard
Mounts:
  /var/run/secrets/kubernetes.io/serviceaccount from dashboard-sa-token-kbbdm (ro)
Conditions:
  Type     Status
Volumes:
  dashboard-sa-token-kbbdm:
    Type:       Secret (a volume populated by a Secret)
    SecretName: dashboard-sa-token-kbbdm
    Optional:   false

```

```

pod-definition.yml
apiVersion: v1
kind: Pod
metadata:
  name: my-kubernetes-dashboard
spec:
  containers:
    - name: my-kubernetes-dashboard
      image: my-kubernetes-dashboard
  serviceAccount: dashboard-sa

```


[KodeKloud.com](https://www.kodekloud.com)

If you'd like to use a different serviceAccount, such as the ones we just created, modify the pod definition file to include a serviceAccount field and specify the name of the new service account. Remember, you cannot edit the service account of an existing pod, so you must delete and re-create the pod. However in case of a deployment, you will be able to edit the serviceAccount, as any changes to the pod definition will automatically trigger a new roll-out for the deployment. So the deployment will take care of deleting and re-creating new pods with the right service account. When you look at the pod details now, you see that the new service account is being used.

```
pod-definition.yml
apiVersion: v1
kind: Pod
metadata:
  name: my-kubernetes-dashboard
spec:
  containers:
    - name: my-kubernetes-dashboard
      image: my-kubernetes-dashboard
  automountServiceAccountToken: false
```



So remember, kubernetes automatically mounts the default service account if you haven't explicitly specified any. You may choose not to mount a service account automatically by setting the `automountServiceAccountToken` field to false in the PODspec section.

Well that's it for this lecture. Head over to the practice exercises section and practice working with service accounts. We will configure the custom kubernetes dashboard with the right service account.



# SERVICE ACCOUNTS

## 1.22/1.24 Update



So there were some changes made in release v1.22 and v1.24 of Kubernetes that changed the way service accounts/secrets and tokens worked.

```

▶ kubectl get serviceaccount
NAME      SECRETS   AGE
default      1        218d

▶ kubectl describe pod my-kubernetes-dashboard
Name:         my-kubernetes-dashboard
Namespace:    default
Annotations: <none>
Status:       Running
IP:          10.244.0.15
Containers:
  nginx:
    Image:      my-kubernetes-dashboard
Mounts:
  /var/run/secrets/kubernetes.io/serviceaccount from default-token-j4hk (ro)
Conditions:
  Type     Status
Volumes:
  default-token-j4hk:
    Type:      Secret (a volume populated by a Secret)
    SecretName: default-token-j4hk
    Optional:  false

```

```

pod-definition.yml
apiVersion: v1
kind: Pod
metadata:
  name: my-kubernetes-dashboard
spec:
  containers:
    - name: my-kubernetes-dashboard
      image: my-kubernetes-dashboard

```

As we discussed in the previous video every namespace has a default service account. And that service account has a secret object with a token associated with it. When a pod is created it automatically associates the service account to the pod and mounts the token to a well known location within the pod. This makes the token accessible to a process within the pod to query the kubernetes API.

```

▶ kubectl describe pod my-kubernetes-dashboard
Name:           my-kubernetes-dashboard
Namespace:      default
Annotations:    <none>
Status:         Running
IP:             10.244.0.15
Containers:
  nginx:
    Image:        my-kubernetes-dashboard
Mounts:
  /var/run/secrets/kubernetes.io/serviceaccount from default-token-j4hkv (ro)
Conditions:
  Type        Status
Volumes:
  default-token-j4hkv:
    Type:       Secret (a volume populated by a Secret)
    SecretName: default-token-j4hkv
    Optional:   false

```

```

▶ kubectl exec -it my-kubernetes-dashboard ls /var/run/secrets/kubernetes.io/serviceaccount
ca.crt  namespace  token

```

```

▶ kubectl exec -it my-kubernetes-dashboard cat /var/run/secrets/kubernetes.io/serviceaccount/token
eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcw5ldGVzL3NlcnZpY2VhY2NvdW50Iiwiia3ViZXJuZXRLcy5pbv9zZXJ2aWNlYWNNjb3VudC9uYW1lc3BhY2UiOjlkZWZhdWx0Iiwiia3ViZXJuZXRLcy5pbv9zZXJ2aWNlYWNNjb3VudC9zZWMyZXQubmFtZSI6ImRlZmF1bHQtdG9rZW4tajRoa3YiLCJrdWJlcw5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNNjb3VudC5uYW11IjoizGVmYXVsdcIsImt1YmVybmv0ZXMuaw8vc2VydmljZWfjY291bnQvc2VydmljZS1hY2NvdW50LnVpZCI6IjcxZGM4YWEExLTU2MGMtMTFlOC04YmI0LTA4MDAyNzkzMta3MiIsInN1YiI6InN5c3R1bTpzzXJ2aWN

```

If you list the <click> contents of the directory inside the pod, you will see the secret mounted as 3 separate files. The one with the actual token is the file named token.  
<click> If you list the contents of that file you will see the token to be used for accessing the kubernetes API.

```
jq -R 'split(".") | select(length > 0) | .[0],.[1] | @base64d | fromjson' <<< eyJhbGciOiJ...  
{  
    "iss": "kubernetes/serviceaccount",  
    "kubernetes.io/serviceaccount/namespace": "default",  
    "kubernetes.io/serviceaccount/secret.name": "default-token-ssdng",  
    "kubernetes.io/serviceaccount/service-account.name": "default",  
    "kubernetes.io/serviceaccount/service-account.uid": "47349a47-07c2-412a-bf0e-11dc0ad16508",  
    "sub": "system:serviceaccount:default:default"  
}
```

Decode a JWT via command line - dbubenheim  
<https://gist.github.com/angelo-v/e0208a18d455e2e6ea3c40ad637aac53>

Now, if you decode this token using this command or by copying pasting this token in the JWT website....

The screenshot shows the jwt.io website interface. At the top, there's a navigation bar with links for Debugger, Libraries, Introduction, Ask, and a Crafted by auth0 by Okta logo. Below the header, there are two main sections: 'Encoded' and 'Decoded'. The 'Encoded' section contains a long string of characters representing the JWT token. The 'Decoded' section is divided into two parts: 'HEADER: ALGORITHM & TOKEN TYPE' and 'PAYLOAD: DATA'. The 'HEADER' part shows a JSON object with 'alg': 'RS256' and 'kid': '28529-xfCnT8RpwPBiHbCcG\_wQoIb0g94cd-DPjZE'. The 'PAYLOAD' part shows a JSON object with various fields, including 'iss': 'kubernetes/serviceaccount', 'kubernetes.io/serviceaccount/namespace': 'default', 'kubernetes.io/serviceaccount/secret.name': 'default-token-ssndg', 'token.ssndg', 'kubernetes.io/serviceaccount/service-account.name': 'default', 'kubernetes.io/serviceaccount/service-account.uid': '47349a47-07c2-412a-bf0e-11dc0ad16508', and 'sub': 'system:serviceaccount:default'. The entire page is framed by a thick black border.

... at jwt.io you'll see that it has no expiry date defined in the payload here on the right.

# v1.22

## KEP 1205 - Bound Service Account Tokens

### Background

Kubernetes already provisions JWTs to workloads. This functionality is on by default and thus widely deployed. The current workload JWT system has serious issues:

1. Security: JWTs are not audience bound. Any recipient of a JWT can masquerade as the presenter to anyone else.
2. Security: The current model of storing the service account token in a Secret and delivering it to nodes results in a broad attack surface for the Kubernetes control plane when powerful components are run - giving a service account a permission means that any component that can see that service account's secrets is at least as powerful as the component.
3. Security: JWTs are not time bound. A JWT compromised via 1 or 2, is valid for as long as the service account exists. This may be mitigated with service account signing key rotation but is not supported by client-go and not automated by the control plane and thus is not widely deployed.
4. Scalability: JWTs require a Kubernetes secret per service account.

<https://github.com/kubernetes/enhancements/tree/master/keps/sig-auth/1205-bound-service-account-tokens#background>

This excerpt from the Kubernetes Enhancements Proposal for creating Bound service accounts tokens describes this form of JWT to be having some security and scalability related issues.

The current implementation of JWT is not bound to any audience, and is not time bound as we just saw – there was no expiry date for the token. The JWT is valid aslong as the service account exists. Moreover each JWT requires a separate secret object per service account which results in scalability issues.

# v1.22

## KEP 1205 - Bound Service Account Tokens



TokenRequestAP  
I

- ✓ Audience Bound
- ✓ Time Bound
- ✓ Object Bound

<https://github.com/kubernetes/enhancements/tree/master/keps/sig-auth/1205-bound-service-account-tokens#background>

and as such in v1.22 the TokenRequest API was introduced as part of the Kubernetes Enhancement Proposal 1205 that aimed to introduce a mechanism for provisioning kubernetes service account tokens that are more secure and scalable via an API.

Tokens generated by the TokenRequest API are audience bound, time bound and object bound and hence are more secure.

```
▶ kubectl get pod my-kubernetes-dashboard -o yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  containers:
    - image: nginx
      name: nginx
      volumeMounts:
        - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
          name: kube-api-access-6mtg8
          readOnly: true
  volumes:
    - name: kube-api-access-6mtg8
      projected:
        defaultMode: 420
        sources:
          - serviceAccountToken:
              expirationSeconds: 3607
              path: token
          - configMap:
              items:
                - key: ca.crt
                  path: ca.crt
                  name: kube-root-ca.crt
          - downwardAPI:
              items:
                - fieldRef:
                    apiVersion: v1
```

Since v1.22 when a new pod is created it no longer relies on the service account secret token, instead a token with a defined lifetime is generated through the TokenRequest API by the Service Account Admission Controller. This token is then mounted as a projected volume into the pod.

## v1.24

### KEP-2799: Reduction of Secret-based Service Account Tokens

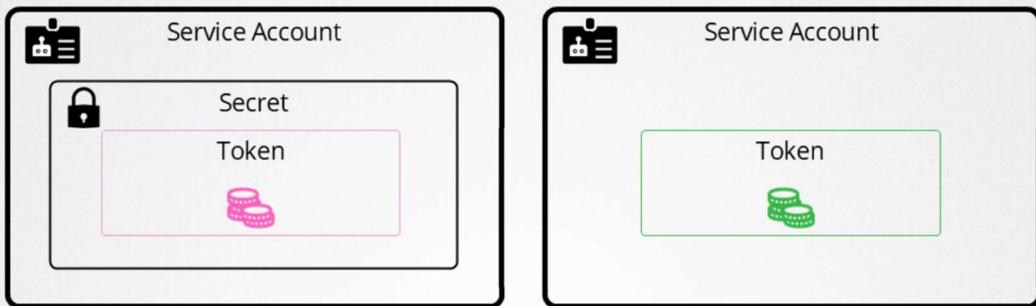
```
▶ kubectl create serviceaccount dashboard-sa
serviceaccount "dashboard-sa" created
```

```
▶ kubectl create serviceaccount dashboard-sa
serviceaccount "dashboard-sa" created
```

```
▶ kubectl create token dashboard-sa
eyJhbGciOiJSUzI1NiIsImtpZCI6I...
```



<https://github.com/kubernetes/enhancements/tree/master/keps/sig-auth/2799-reduction-of-secret-based-service-account-token>

With version 1.24 another enhancement was made as part of the KEP 2799 – Reduction of secret based service account tokens. In the past when a service account was created, it automatically created a secret with a token that has no expiry and is not bound to any audience. This was then automatically mounted as a volume to any pod that uses that service account.

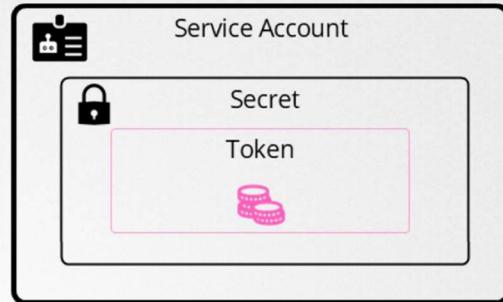
With v1.24 when a service account is created it does not automatically create a secret or a token. You must run the command `kubectl create token` followed by the name of the service account to generate a token for the service account. It then prints the token on screen.

```
▶ jq -R 'split(".") | select(length > 0) | .[0],.[1] | @base64d | fromjson' <<< eyJhbGciOiJSUz...
{
  "alg": "RS256",
  "kid": "vOp_YzJsnmxNV9uQ8zzs0LIHha0n3Dy800v1EjXnSs"
}
{
  "aud": [
    "https://kubernetes.default.svc.cluster.local"
  ],
  "exp": 1664037763,
  "iat": 1664034163,
  "iss": "https://kubernetes.default.svc.cluster.local",
  "kubernetes.io": {
    "namespace": "default",
    "serviceaccount": {
      "name": "dashboard-sa",
      "uid": "7d0fdfe8-fbf3-4ff9-b004297a73f6"
    }
  },
  "nbf": 1664034163,
  "sub": "system:serviceaccount:default:dashboard-sa"
}
```

If you decode this token, this time you'll see an expiry defined. If you haven't specified any time limit, then it's usually 1 hour from the time you ran the command. You can pass in additional options to the command to increase the expiry of the token.

## v1.24

```
secret-definition.yml
apiVersion: v1
kind: Secret
type: kubernetes.io/service-account-token
metadata:
  name: mysecretname
  annotations:
    kubernetes.io/service-account.name: dashboard-sa
```



Now post v1.24 if you would still like to create secrets the old way with non-expiringtoken, then you could create a secret object with the type set to kubernetes.io/service-account-token and the name of the service account specified within the annotations in metadata.

This will create a non-expiring token in a secret object and associate it with the service account. But for this make sure the service account already exists prior to creating the secret.

## Service account token Secrets

A `kubernetes.io/service-account-token` type of Secret is used to store a token credential that identifies a service account.

Since 1.22, this type of Secret is no longer used to mount credentials into Pods, and obtaining tokens via the `TokenRequest` API is recommended instead of using service account token Secret objects. Tokens obtained from the `TokenRequest` API are more secure than ones stored in Secret objects, because they have a bounded lifetime and are not readable by other API clients. You can use the `kubectl create token` command to obtain a token from the `TokenRequest` API.

You should only create a service account token Secret object if you can't use the `TokenRequest` API to obtain a token, and the security exposure of persisting a non-expiring token credential in a readable API object is acceptable to you.

<https://kubernetes.io/docs/concepts/configuration/secret/#service-account-token-secrets>

As per the kubernetes documentation pages on Service account token secrets, you should only create service account token secret if you can't use the `TokenRequest` API to obtain a token. That's either the `kubectl create token` command we just talked about. It talks to the token request API to generate that token. Or it's the automated token creation that happens on pods when they are created post v1.22. And also you should only create service account token secret if the security exposure of persisting a non-expiring token credential is acceptable to you.

The `TokenRequest` API is recommended instead of using service account token secret objects, as they are more secure and have a bounded lifetime unlike the service account token secrets that have no expiry.

## References

### KEP 1205 - Bound Service Account Tokens

<https://github.com/kubernetes/enhancements/tree/master/keps/sig-auth/1205-bound-service-account-tokens>

### KEP-2799: Reduction of Secret-based Service Account Tokens

<https://github.com/kubernetes/enhancements/tree/master/keps/sig-auth/2799-reduction-of-secret-based-service-account-token>

<https://kubernetes.io/docs/concepts/configuration/secret/#service-account-token-secrets>  
<https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/>

To read more about these changes refer to the Kubernetes Enhancement proposals and the documentation pages listed here.

```
kubectl create serviceaccount dashboard-sa
serviceaccount "dashboard-sa" created
```

```
kubectl get serviceaccount
NAME      SECRETS   AGE
default   1         218d
dashboard-sa   1         4d
```

```
kubectl describe serviceaccount dashboard-sa
Name:           dashboard-sa
Namespace:      default
Labels:          <none>
Annotations:    <none>
Image pull secrets: <none>
Mountable secrets: dashboard-sa-token-kbbdm
Tokens:          dashboard-sa-token-kbbdm
Events:          <none>
```

```
pod-definition.yml
apiVersion: v1
kind: Pod
metadata:
  name: my-kubernetes-dashboard
spec:
  containers:
    - name: my-kubernetes-dashboard
      image: my-kubernetes-dashboard
  serviceAccountName: dashboard-sa
```

As I mentioned in the previous video when a service account is created by running the command `kubectl create service account` command [click](#) it creates a `serviceaccount` by that name and also automatically creates a `secret` object by the same name with a token in it. This service account is then automatically mounted to a pod, and thus the token is available within the pod automatically.

```

▶ kubectl describe pod my-kubernetes-dashboard
Name:           my-kubernetes-dashboard
Namespace:      default
Annotations:    <none>
Status:         Running
IP:             10.244.0.15
Containers:
  nginx:
    Image:        my-kubernetes-dashboard
Mounts:
  /var/run/secrets/kubernetes.io/serviceaccount from default-token-j4hkv (ro)
Conditions:
  Type          Status
Volumes:
  default-token-j4hkv:
    Type:       Secret (a volume populated by a Secret)
    SecretName: default-token-j4hkv
    Optional:   false

```

```

▶ kubectl exec -it my-kubernetes-dashboard ls /var/run/secrets/kubernetes.io/serviceaccount
ca.crt  namespace  token

```

```

▶ kubectl exec -it my-kubernetes-dashboard cat /var/run/secrets/kubernetes.io/serviceaccount/token
eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcw5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRLcy5pbv9zZXJ2aWNlYWNNjb3VudC9uYW1lc3BhY2UiOjlkZWZhdWx0Iiwia3ViZXJuZXRLcy5pbv9zZXJ2aWNlYWNNjb3VudC9zZWMyZXQubmFtZSI6ImRlZmF1bHQtdG9rZW4tajRoa3YiLCJrdWJlcw5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNNjb3VudC5uYW11IjoicZGVmYXVsdsCIsImt1YmVybmv0ZXMuaw8vc2VydmljZWFjY291bnQvc2VydmljZS1hY2NvdW50LnVpZCI6IjcxZGM4YWExLTU2MGMtMTFlOC04YmI0LTA4MDAyNzkzMta3MiIsInN1YiI6InN5c3R1bTpzzXJ2aWN

```

If you list the <click> contents of the directory inside the pod, you will see the secret mounted as 3 separate files. The one with the actual token is the file named token.

<click> If you list the contents of that file you will see the token to be used for accessing the kubernetes API. Now remember that the default service account is verymuch restricted. It only has permission to run basic kubernetes API queries.

**Note:**

Versions of Kubernetes before v1.22 automatically created credentials for accessing the Kubernetes API. This older mechanism was based on creating **token** Secrets that could then be mounted into running Pods. In more recent versions, including Kubernetes v1.25, API credentials are obtained directly by using the **TokenRequest** API, and are mounted into Pods using a **projected volume**. The **tokens** obtained using this method have bounded lifetimes, and are automatically invalidated when the Pod they are mounted into is deleted.

You can still **manually create** a service account **token** Secret; for example, if you need a **token** that never expires. However, using the **TokenRequest** subresource to obtain a **token** to access the API is recommended instead. You can use the `kubectl create token` command to obtain a **token** from the **TokenRequest** API.

## Service account `token` Secrets

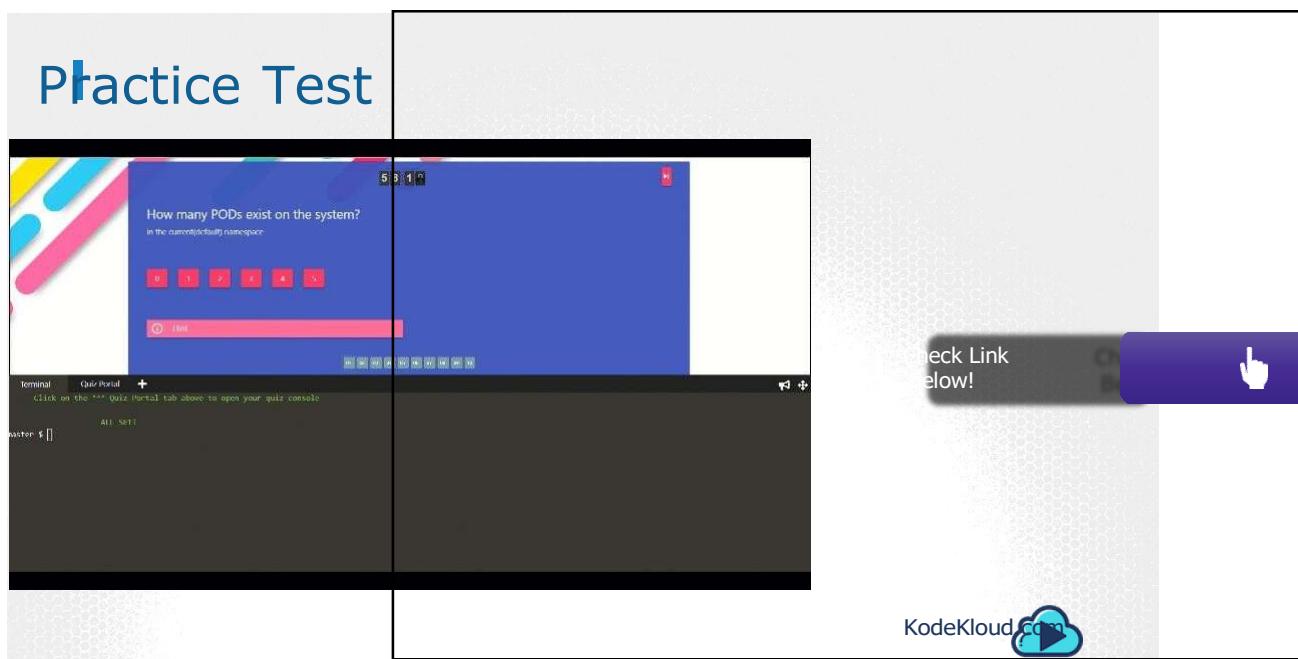
A `kubernetes.io/service-account-token` type of Secret is used to store a `token` credential that identifies a service account.

Since 1.22, this type of Secret is no longer used to mount credentials into Pods, and obtaining `tokens` via the `TokenRequest` API is recommended instead of using service account `token` Secret objects. `Tokens` obtained from the `TokenRequest` API are more secure than ones stored in Secret objects, because they have a bounded lifetime and are not readable by other API clients. You can use the `kubectl create token` command to obtain a `token` from the `TokenRequest` API.

You should only create a service account `token` Secret object if you can't use the `TokenRequest` API to obtain a `token`, and the security exposure of persisting a non-expiring `token` credential in a readable API object is acceptable to you.

When using this Secret type, you need to ensure that the `kubernetes.io/service-account.name` annotation is set to an existing service account name. If you are creating both the ServiceAccount and the Secret objects, you should create the ServiceAccount object first.

After the Secret is created, a Kubernetes controller fills in some other fields such as the `kubernetes.io/service-account.uid` annotation, and the `token` key in the `data` field, which is populated with an authentication `token`.



Access Test Here: <https://kodekloud.com/courses/kubernetes-certification-course/lectures/8598237>

## I References

<https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/>

<https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/>



## Course Objectives

- ✓ Core Concepts
- ✓ Configuration
  - ✓ ConfigMaps
  - ✓ SecurityContexts
  - Resource Requirements
- Multi-Container Pods
- Observability
- Pod Design
- Services & Networking
- State Persistence

✓ Secrets

✓ ServiceAccounts



KodeKloud.com

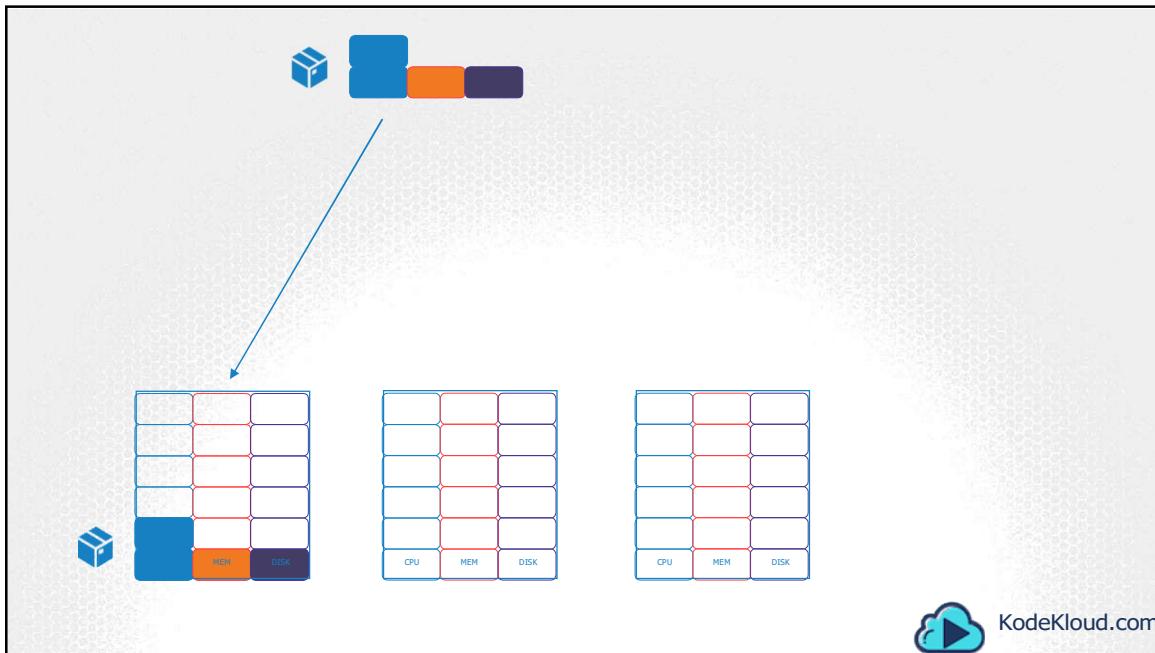
The next section on Configuration covers topics like ConfigMaps, SecurityContexts, Resource Requirements, secrets and service accounts.

140

# Resource Requirements

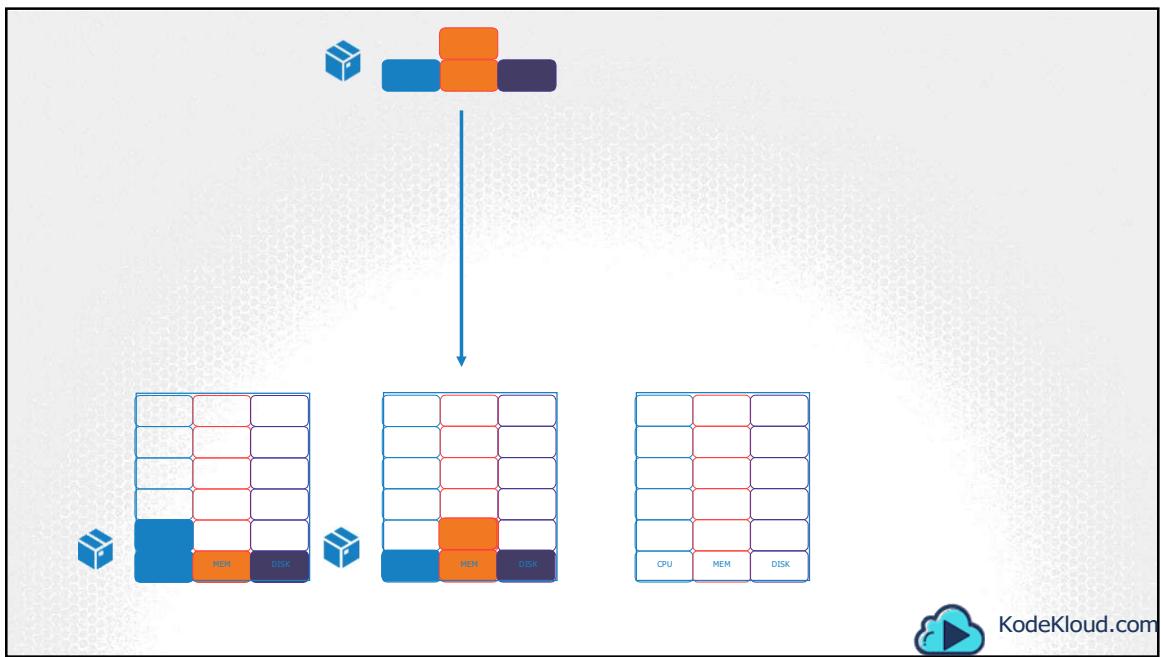


KodeKloud.com

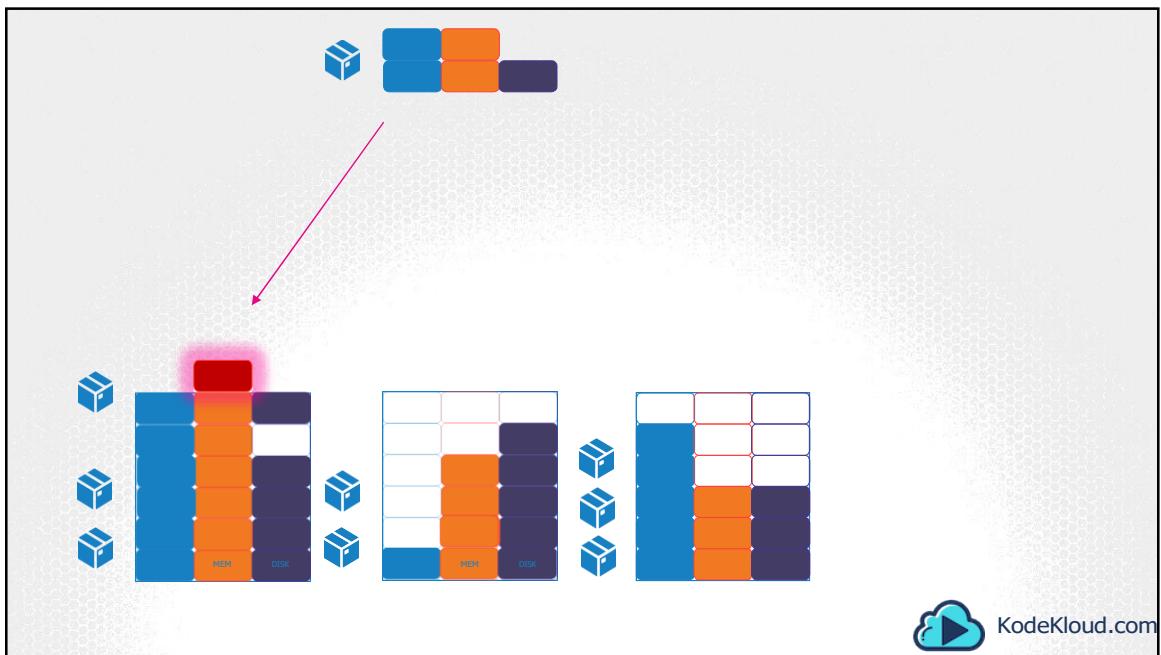


KodeKloud.com

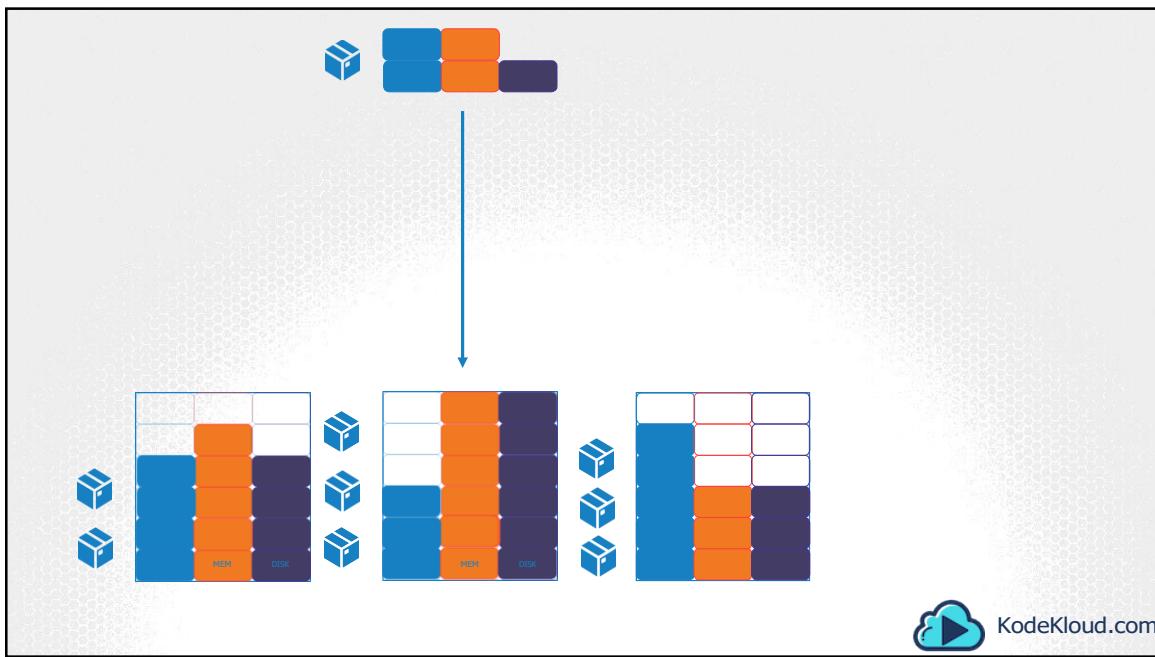
Let us look at a 3 Node Kubernetes cluster. Each node has a set of CPU, Memory and Disk resources available. Every POD consumes a set of resources. In this case 2 CPUs , one Memory and some disk space. Whenever a POD is placed on a Node, it consumes resources available to that node.



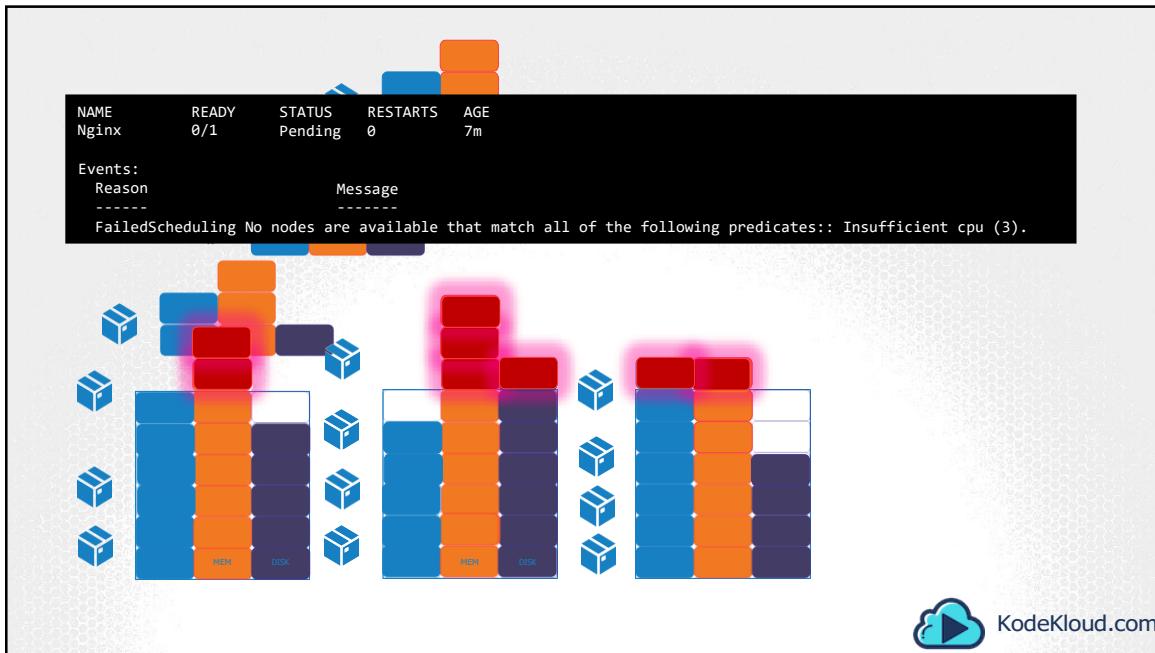
As we have discussed before, it is the kubernetes scheduler that decides which Node a POD goes to. The scheduler takes into consideration, the amount of resources required by a POD and those available on the Nodes. In this case, the scheduler schedules a new POD on Node 2.



If the node has no sufficient resources, the scheduler avoids placing the POD on that node...

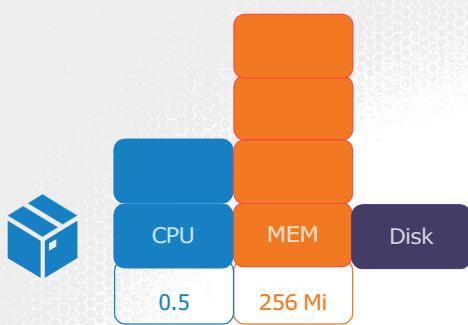


.. Instead places the POD on one were sufficient resources are available. Some of the related topics such as scaling and auto-scaling PODs and Nodes in the cluster and how the scheduler itself works are out of scope for this course and the Kubernetes Application Developer certification. These are discussed in much more detail in the Kubernetes Administrators course. In this course we focus on setting resource requirements for PODs from an application developer's viewpoint.



If there is no sufficient resources available on any of the nodes, Kubernetes holds back scheduling the POD, and you will see the POD in a pending state. If you look at the events, you will see the reason – insufficient cpu.

# Resource Requests



pod-definition.yaml

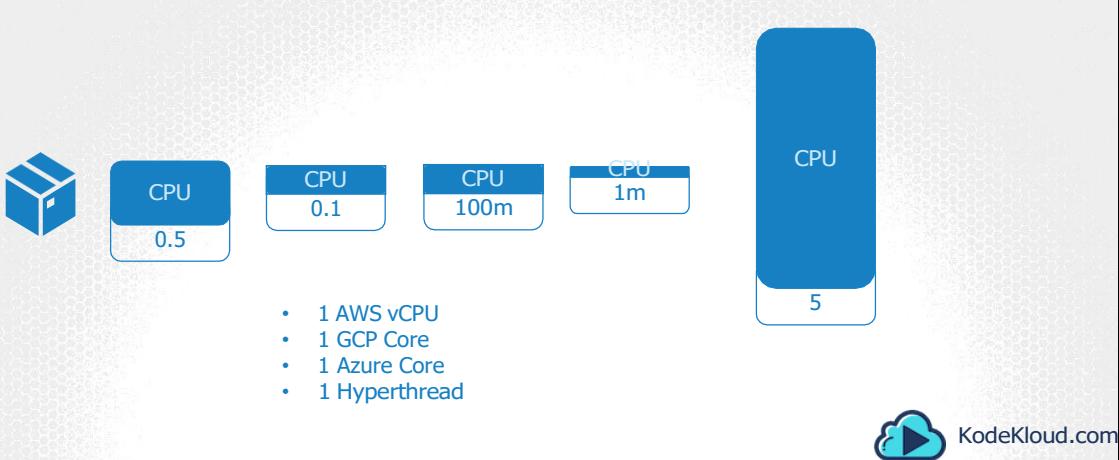
```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
  labels:
    name: simple-webapp-color
spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
    - containerPort: 8080
  resources:
    requests:
      memory: "1Gi"
      cpu: 1
```



KodeKloud.com

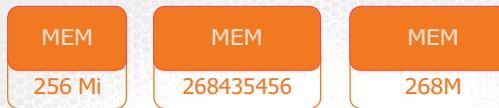
Let us now focus on the resource requirements for each POD. What are these blocks and what are their values? By default, kubernetes assumes that a POD or a container within a POD requires .5 CPU & 256 Mebibyte of memory. This is known as the resource request for a container. The minimum amount of CPU or Memory requested by the container. When the scheduler tries to place the POD on a Node, it uses these numbers to identify a Node which has sufficient amount of resources available. Now, if you know that your application will need more than these, you can modify these values, by specifying them in your POD or deployment definition files. In this sample pod definition file, add a section called resources, under which add requests and specify the new values for memory and cpu usage. In this case I set it to 1GB of memory and 1 count of vCPU.

## I Resource - CPU



So what does 1 count of CPU really mean? Remember these blocks are used for illustration purpose only. It doesn't have to be in the increment of .5. You can specify any value as low as 0.1. 0.1 CPU can also be expressed as 100m were m stands for milli. You can go as low as 1m, but not lower than that. 1 count of CPU is equivalent to 1 vCPU. That's 1 vCPU in AWS, or 1 Core in GCP or Azure or 1 Hyperthread. You could request a higher number of CPUs for the container, provided your Nodes are sufficiently funded.

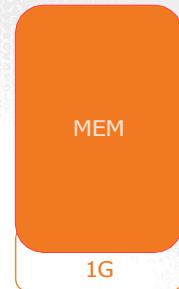
## I Resource - Memory



1 G (Gigabyte) = 1,000,000,000 bytes

1 M (Megabyte) = 1,000,000 bytes

1 K (Kilobyte) = 1,000 bytes



1 Gi (Gibibyte) = 1,073,741,824 bytes

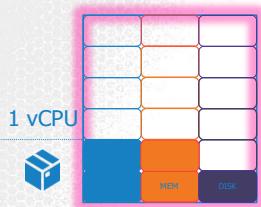
1 Mi (Mebibyte) = 1,048,576 bytes

1 Ki (Kibibyte) = 1,024 bytes



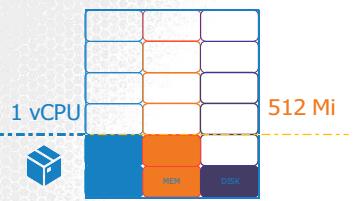
Similarly, with memory you could specify 256 Mebibyte using the Mi suffix. Or specify the same value in Memory like this. Or specify the same value in Memory like this. Or use the suffix G for Gigabyte. Note the difference between G and Gi. G is Gigabyte and it refers to a 1000 Megabytes, whereas Gi refers to Gibibyte and refers to 1024 Mebibyte. The same applies to Megabyte and Kilobyte

## I Resource Limits



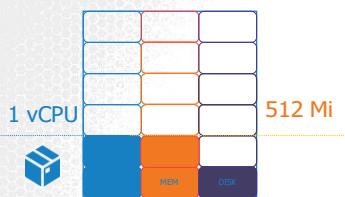
Let's now look at a container running on a Node. In the Docker world, a docker container has no limit to the resources it can consume on a Node. Say a container starts with 1 vCPU on a Node, it can go up and consume as much resource as it requires, suffocating the native processes on the node or other containers of resources. However, you can set a limit for the resource usage on these PODs. By default Kubernetes sets a limit of 1vCPU to containers. So if you do not specify explicitly, a container will be limited to consume only 1 vCPU from the Node.

## | Resource Limits



The same goes with memory. By default, kubernetes sets a limit of 512 Mebibyte on containers.

# Resource Limits



pod-definition.yaml

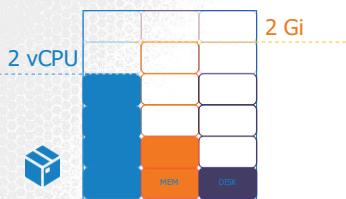
```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
  labels:
    name: simple-webapp-color
spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
      - containerPort: 8080
  resources:
    requests:
      memory: "1Gi"
      cpu: 1
    limits:
      memory: "2Gi"
      cpu: 2
```



KodeKloud.com

If you don't like the default limits, you can change them by adding a limits section under the resources section in your pod definition. Specify new limits for memory and cpu.

# Resource Limits



pod-definition.yaml

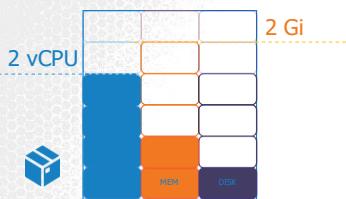
```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
  labels:
    name: simple-webapp-color
spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
      - containerPort: 8080
    resources:
      requests:
        memory: "1Gi"
        cpu: 1
      limits:
        memory: "2Gi"
        cpu: 2
```



KodeKloud.com

When the pod is created, kubernetes sets new limits for the container. Remember that the limits and requests are set for each container.

# Resource Limits



pod-definition.yaml

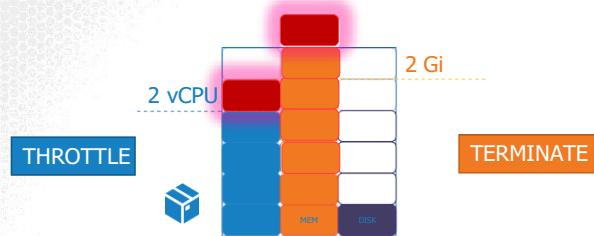
```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
  labels:
    name: simple-webapp-color
spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
      - containerPort: 8080
    resources:
      requests:
        memory: "1Gi"
        cpu: 1
      limits:
        memory: "2Gi"
        cpu: 2
```



KodeKloud.com

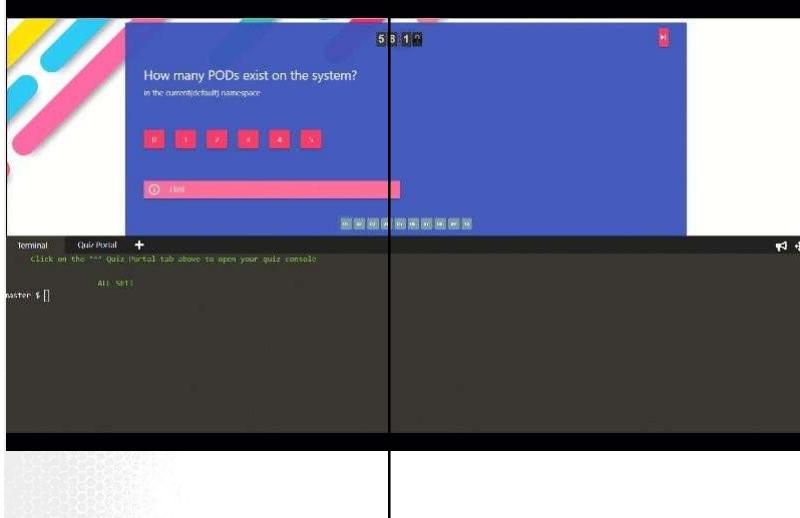
When the pod is created, kubernetes sets new limits for the container. Remember that the limits and requests are set for each container.

## I Exceed Limits



So what happens when a pod tries to exceed resources beyond its specified limit. In case of the CPU, kubernetes throttles the CPU so that it does not go beyond the specified limit. A container cannot use more CPU resources than its limit. However, this is not the case with memory. A container CAN use more memory resources than its limit. So if a pod tries to consume more memory than its limit constantly, the POD will be terminated.

## Practice Test



Check Link  
below!



KodeKloud.com

Access Test Here: <https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743659>

## I References

<https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/>

<https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource/>

<https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/memory-default-namespace/>

<https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/cpu-default-namespace/>

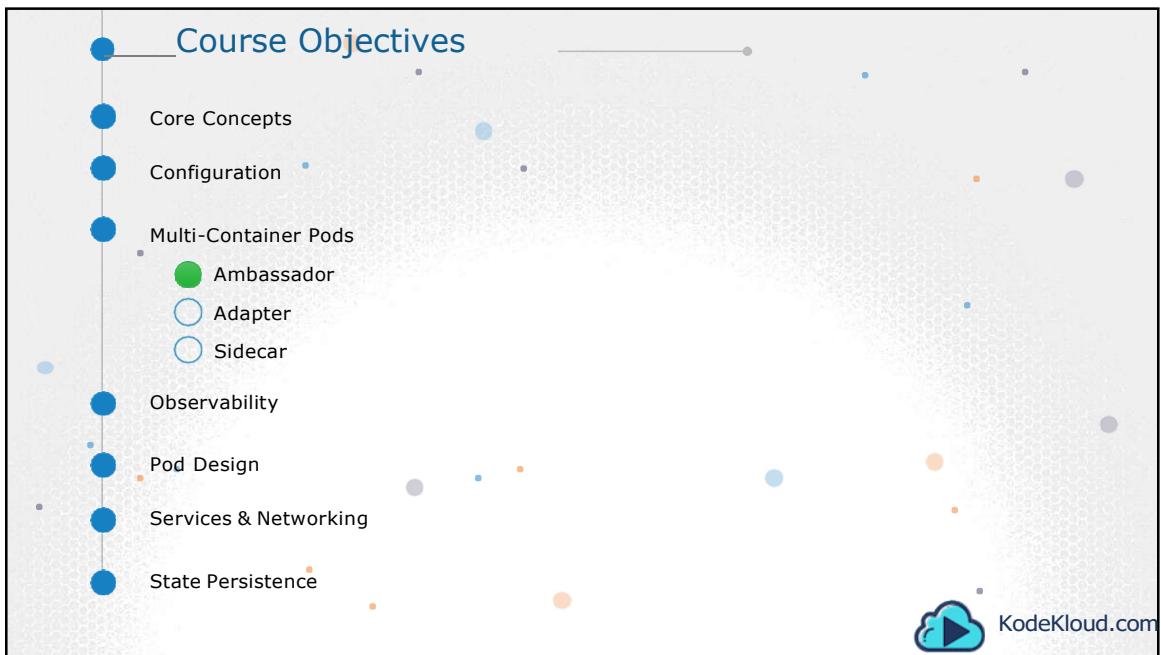
<https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/memory-constraint-namespace/>

<https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/>

<https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/quota-memory-cpu-namespace/>



KodeKloud.com



Hello and welcome to this section on Multi-Container Pods. My name is Mumshad Mannambeth and we are going through the Certified Kubernetes Applications Developer course.

There are different patterns of multi-container pods such as the Ambassador, Adapter and Sidecar. We will look at each of these in this section.

# Kubernetes Multi-Container PODs



Before we head into each of these, let us start with the basic type of POD.

I



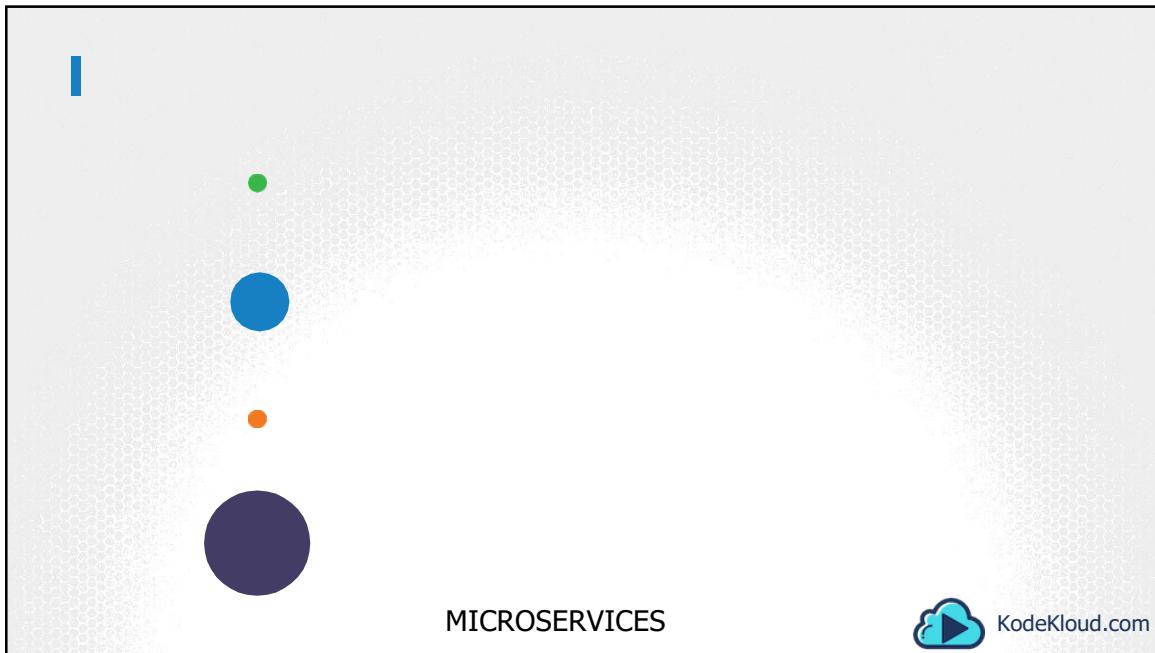
MONOLITH



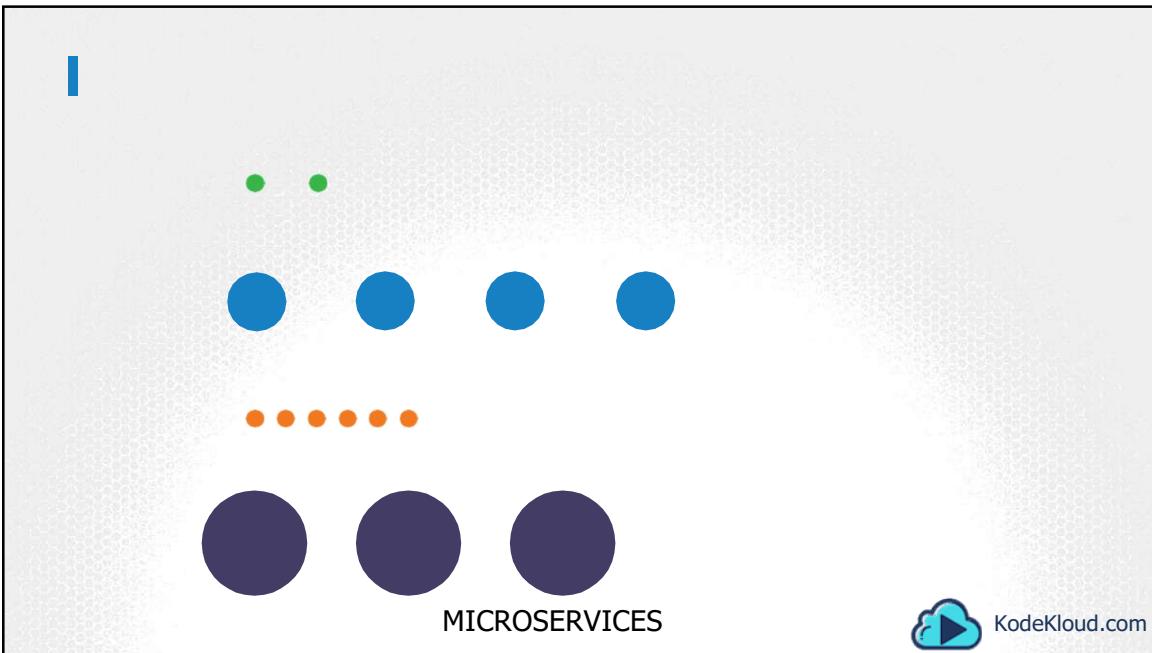
The idea of decoupling a large monolithic application into ...



.. subcomponents known as microservices enables us to develop and deploy a set of independent, small and reusable code.

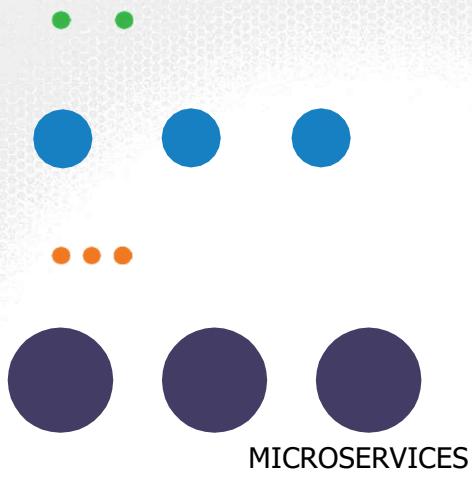


This, architecture then helps us



scale up..

I

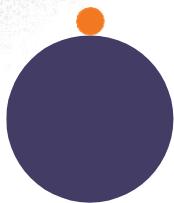


... , down as well as modify each service as required, as opposed to modifying the entire application.

I

LOG Agent

WEB Server

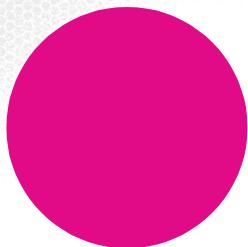


However, at times you may need two services to work together. Such as a web server and a logging service. You need one agent per web server instance paired together ...

I

LOG Agent

WEB Server



KodeKloud.com

You don't want to merge and bloat the code of the two services, as each of them target different features, and you'd still like them to be developed and deployed separately.

I

LOG Agent

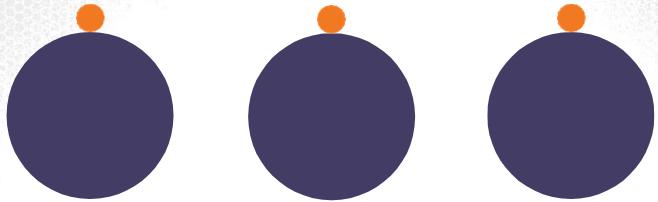
WEB Server



KodeKloud.com

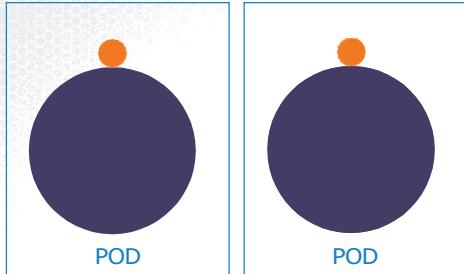
You only need the two functionality to work together. You need one agent per web server instance paired together ...

I



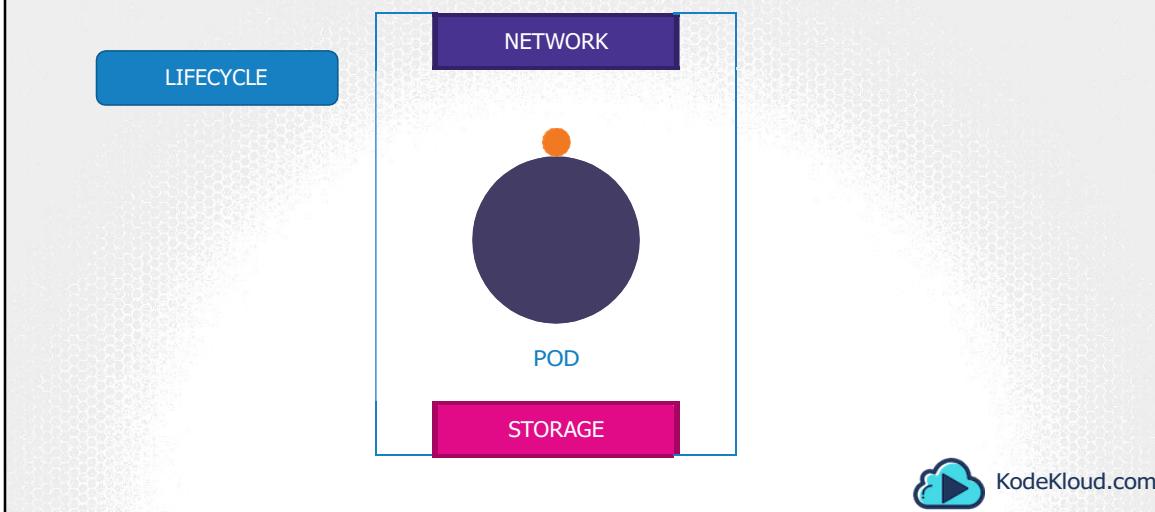
.. that can scale up ...

## | Multi-Container PODs



.. and down together. And that is why you have multi-container PODs....

## Multi-Container PODs

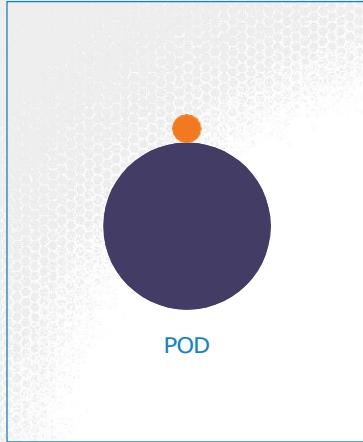


that share the same lifecycle – which means they are created together and destroyed together. They share the same network space, which means they can refer to each other as localhost. And they have access to the same storage volumes. This way, you do not have to establish, volume sharing or services between the PODs to enable communication between them.



KodeKloud.com

# Create



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
  - name: log-agent
    image: log-agent
```



To create a multi-container pod, add the new container information to the pod-definition file. Remember, the containers section under the spec section in a pod definition file is an array and the reason it is an array is to allow multiple containers in a single POD. In this case we add a new container named log-agent to our existing pod. We will look at more realistic examples later.

## I Design Patterns

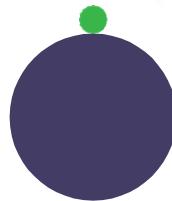


There are 3 common patterns, when it comes to designing multi-container PODs. The first and what we just saw with the logging service example is known as a side car pattern.

## I Design Patterns



SIDECAR



ADAPTER



AMBASSADOR

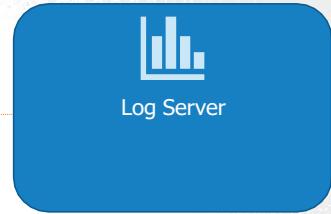


The others are the adapter and the ambassador pattern.

## I Design Patterns - Sidecar



SIDECAR



A good example of a side car pattern is deploying a logging agent along side a web server to collect logs and forward them to a central log server.

## I Design Patterns



● 12-JULY-2018 16:05:49 "GET /index1.html" 200



● 12/JUL/2018:16:05:49 -0800 "GET /index2.html" 200



● GET 1531411549 "/index3.html" 200



KodeKloud.com

Building on that example, say we have multiple applications generating logs in different formats. It would be hard to process the various formats on the central logging server.

## I Design Patterns - Adapter



12-JULY-2018 16:05:49 "GET /index1.html" 200

12-JULY-2018 16:05:49 "GET /index1.html" 200



12/JUL/2018:16:05:49 -0800 "GET /index2.html" 200

12-JULY-2018 16:05:49 "GET /index2.html" 200



GET 1531411549 "/index3.html" 200

12-JULY-2018 16:05:49 "GET /index3.html" 200

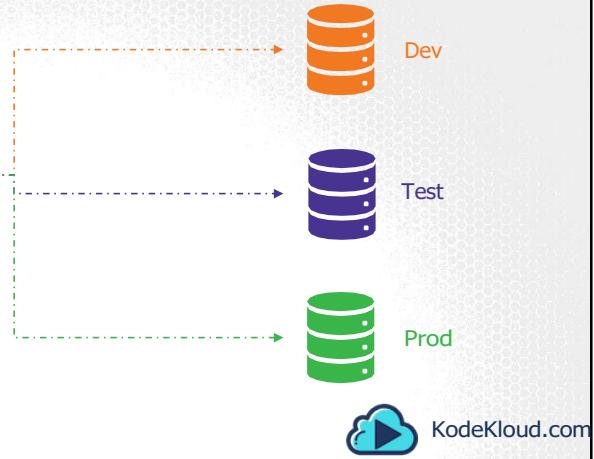
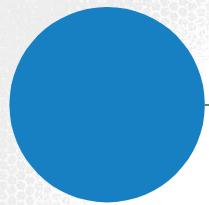
ADAPTER



KodeKloud.com

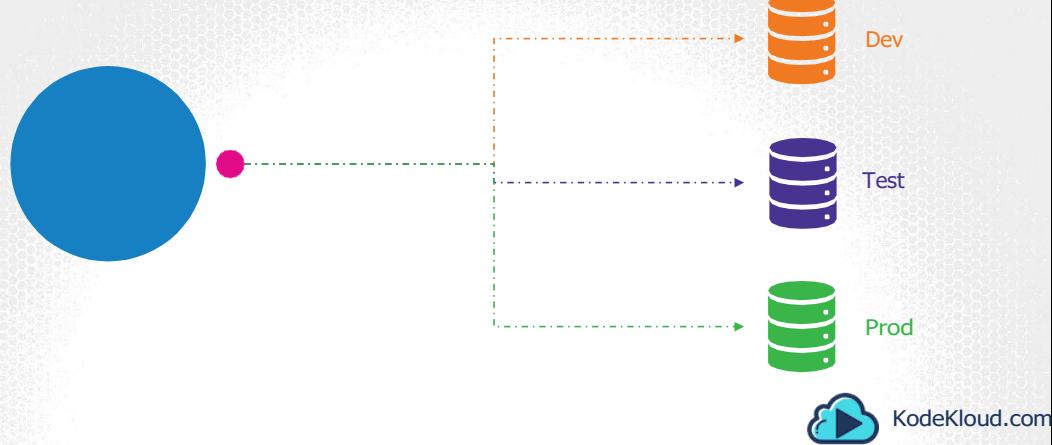
So, before sending the logs to the central server, we would like to convert the logs to a common format. For this we deploy an adapter container. The adapter container processes the logs, before sending it to the central server.

## I Design Patterns - Ambassador



So your application communicates to different database instances at different stages of development. A local database for development, one for testing and another for production. You must ensure to modify this connectivity depending on the environment you are deploying your application to.

## I Design Patterns - Ambassador



You may choose to outsource such logic to a separate container within your POD, so that your application can always refer to a database at localhost, and the new container, will proxy that request to the right database. This is known as an ambassador container.

## I Design Patterns - Ambassador

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
    - name: log-agent
      image: log-agent
```

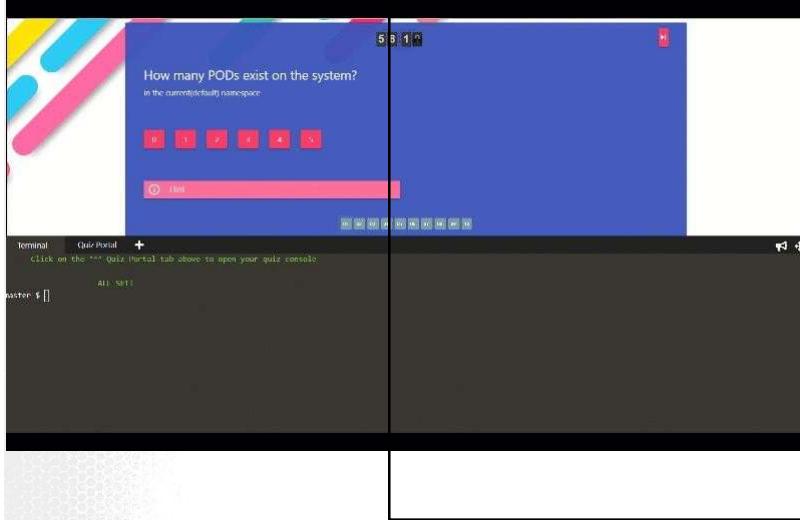


KodeKloud.com

Again, remember that these are different patterns in designing a multi-container pod. When it comes to implementing them using a pod-definition file, it is always the same. You simply have multiple containers within the pod definition file.

Well that's it for this lecture. Head over to the coding exercises section and practice configuring multi-container pods. See you in the next lecture.

## Practice Test



Check Link  
below!

KodeKloud.com

Access Test Here: <https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743661>

## I References

<https://kubernetes.io/docs/tasks/access-application-cluster/communicate-containers-same-pod-shared-volume/>

<https://kubernetes.io/docs/tasks/access-application-cluster/communicate-containers-same-pod-shared-volume/>





Hello and welcome to this section. In this section we learn about Observability in Kubernetes. We will discuss about Readiness and Liveness Probes, Logging and Monitoring concepts.

# Readiness Probes

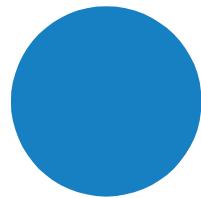


Let us start with Readiness Probes

I

POD Status

POD Conditions



We discuss about Pod Lifecycle in detail in another lecture. However, as a pre-requisite for this lecture, we will quickly recap few different stages in the lifecycle of a POD. A POD has a pod status and some conditions.

## POD Status



The POD status tells us where the POD is in its lifecycle. When a POD is first created, it is in a Pending state. This is when the Scheduler tries to figure out where to place the POD. If the scheduler cannot find a node to place the POD, it remains in a Pending state. To find out why it's stuck in a pending state, run the `kubectl describe pod` command, and it will tell you exactly why.

Once the POD is scheduled, it goes into a ContainerCreating status, where the images required for the application are pulled and the container starts. Once all the containers in a POD start, it goes into a running state, where it continues to be until the program completes successfully or is terminated.

You can see the pod status in the output of the `kubectl get pods` command. So remember, at any point in time the POD status can only be one of these values and only gives us a high level summary of a POD. However, at times you may want additional information.

## POD Conditions

```
osboxes@kubemaster:~$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
jenkins-566f687bf-c7nzf   1/1     Running   0          12m
nginx-65899c769f-9lwzh   1/1     Running   0          6h
redis-b48685f8b-fbnmx   1/1     Running   0          6h
```

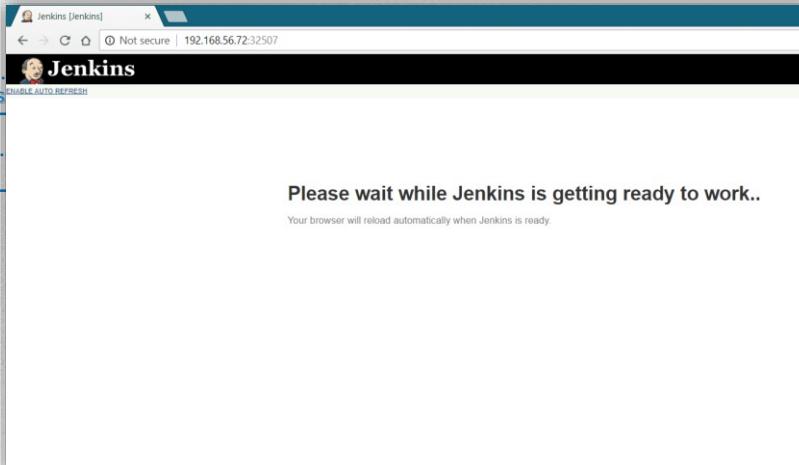
Conditions compliment POD status. It is an array of true or false values that tell us the state of a POD. When a POD is scheduled on a Node, the PodScheduled condition is set to True. When the POD is initialized, it's value is set to True. We know that a POD has multiple containers. When all the containers in the POD are ready, the Containers Ready condition is set to True and finally the POD itself is considered to be Ready.

To see the state of POD conditions run the kubectl describe POD command and look for the conditions section.

You can also see the Ready state of the POD, in the output of the kubectl get pods command.

And that is the condition we are interested in for this lecture.

## I POD Conditions



The screenshot shows a Jenkins pod status page. On the left, there's a sidebar with 'Containers' and 'Ready' buttons. The main area displays the message 'Please wait while Jenkins is getting ready to work..'. Below this, a note says 'Your browser will reload automatically when Jenkins is ready.' At the bottom, a terminal window shows the command 'kubectl get all' and its output:

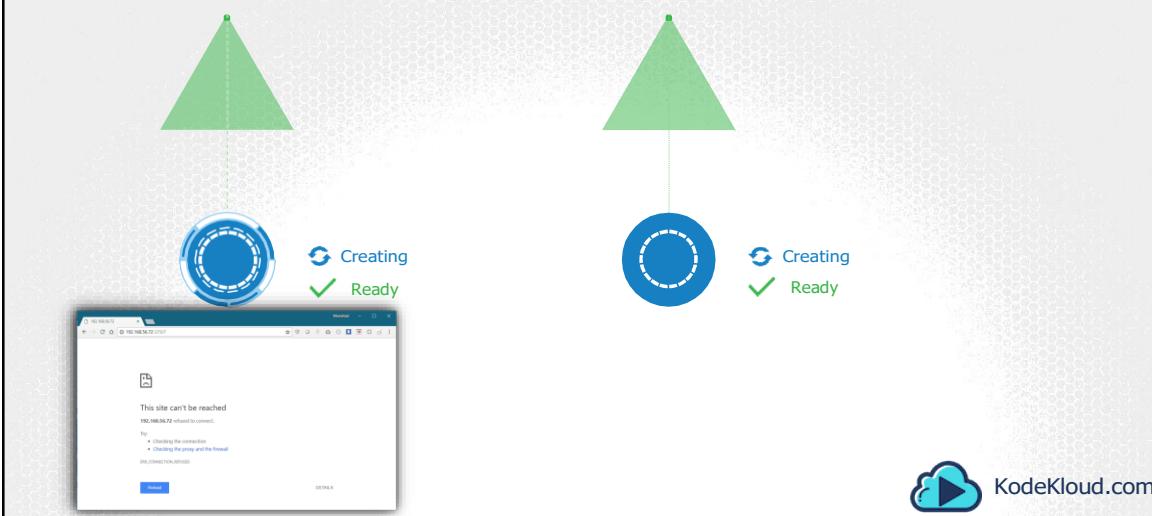
```
osboxes@kubemaster:~$ kubectl get all
NAME           READY   STATUS    RESTARTS   AGE
pod/jenkins    1/1     Running   0          11s
```

On the right side of the terminal window, there's a KodeKloud.com logo.

The ready conditions indicate that the application inside the POD is running and is ready to accept user traffic. What does that really mean? The containers could be running different kinds of applications in them. It could be a simple script that performs a job. It could be a database service. Or a large web server, serving front end users. The script may take a few milliseconds to get ready. The database service may take a few seconds to power up. Some web servers could take several minutes to warm up. If you try to run an instance of a Jenkins server, you will notice that it takes about 10-15 seconds for the server to initialize before a user can access the web UI. Even after the Web UI is initialized, it takes a few seconds for the server to warm up and be ready to serve users. During this wait period if you look at the state of the POD, it continues to indicate that the POD is ready, which is not very true.

So why is that happening and how does kubernetes know weather that the application inside the container is actually running or not? But before we get into that discussion, why does it matter if the state is reported incorrectly.

## I POD Conditions



Let us look at a simple scenario where you create a POD and expose it to external users using a service. The service will route traffic to the POD immediately. The service relies on the pod's READY condition to route traffic.

By default, Kubernetes assumes that as soon as the container is created, it is ready to serve user traffic. So it sets the value of the "Ready Condition" for each container to True. But if the application within the container took longer to get ready, the service is unaware of it and sends traffic through as the container is already in a ready state, causing users to hit a POD that isn't yet running a live application.

What we need here is a way to tie the ready condition to the actual state of the application inside the container. As a Developer of the application, YOU know better what it means for the application to be ready.

## I Readiness Probes



HTTP Test - /api/ready



TCP Test - 3306



Exec Command



KodeKloud.com

There are different ways that you can define if an application inside a container is actually ready. You can setup different kinds of tests or Probes, which is the appropriate term. In case of a web application it could be when the API server is up and running. So you could run a HTTP test to see if the API server responds. In case of database, you may test to see if a particular TCP socket is listening. Or You may simply execute a command within the container to run a custom script that would exit successfully if the application is ready.

# I Readiness Prob



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



KodeKloud.com

So how do you configure that test? In the pod definition file, add a new field called `readinessProbe` and use the `httpGet` option. Specify the port and the ready api. Now when the container is created, kubernetes does not immediately set the ready condition on the container to true, instead, it performs a test to see if the api responds positively. Until then the service does not forward any traffic to the pod, as it sees that the POD is not ready.

## I Readiness 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
```

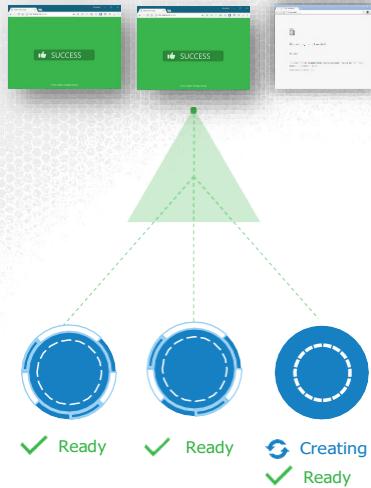
HTTP Test - /api/ready

TCP Test - 3306

Exec Command

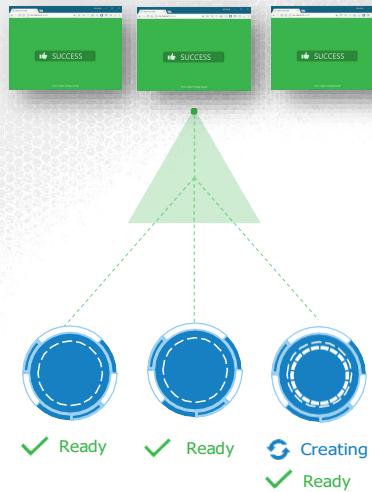
There are different ways a probe can be configured. For http, use the httpGet option with the path and port. For TCP use the tcpSocket option with port. And for executing a command specify the exec option with the command and options in an array format. There are some additional options as well. If you know that your application will always take a minimum of, say, 10 seconds to warm up, you can add an initial delay to the probe. If you'd like to specify how often to probe, you can do that using the periodSeconds option. By default if the application is not ready after 3 attempts, the probe will stop. If you'd like to make more attempts, use the failureThreshold option. We will look through more options in the Documentation Walkthrough.

## I POD Conditions



Finally, Let us look at how readinessProbes are useful in a multi-pod setup. Say you have a replica set or deployment with multiple pods. And a service serving traffic to all the pods. There are two PODs already serving users. Say you were to add an additional pod. And let's say the Pod takes a minute to warm up. Without the readinessProbe configured correctly, the service would immediately start routing traffic to the new pod. That will result in service disruption to atleast some of the users.

## I POD Conditions



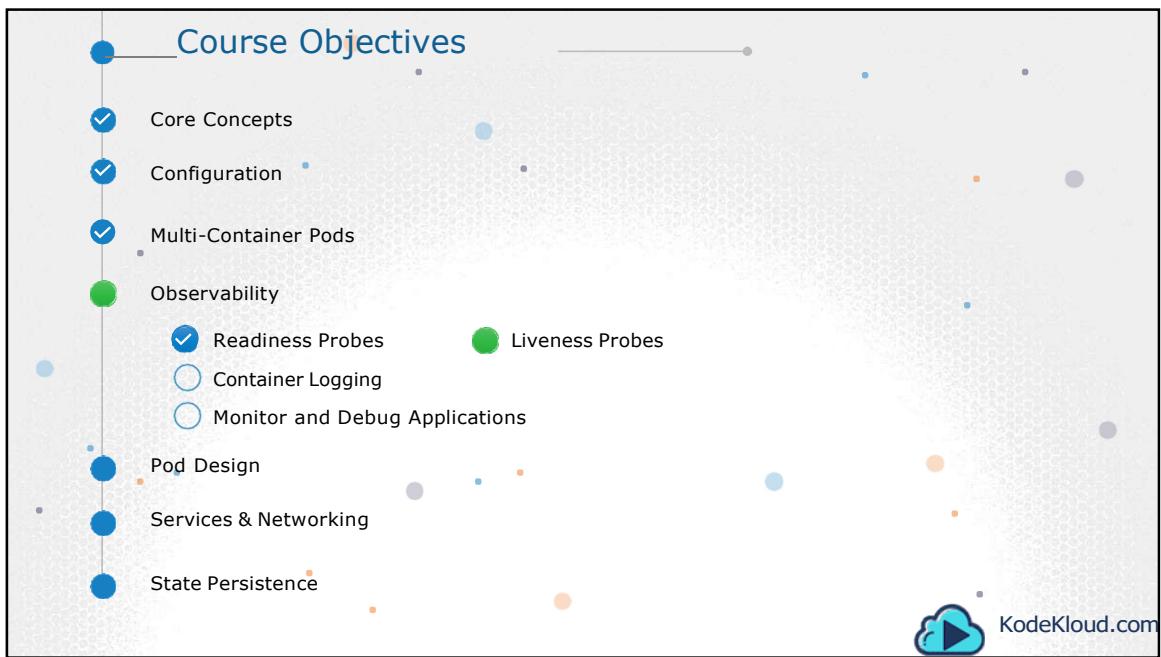
Instead if the pods were configured with the correct readinessProbe, the service will continue to serve traffic only to the older pods and wait until the new pod is ready. Once ready, traffic will be routed to the new pod as well, ensuring no users are affected.

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

## I References

<https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/>





Hello and welcome to this lecture. My name is Mumshad Mannambeth and we are learning the Certified Kubernetes Applications Developer's course. In this lecture we will talk about Liveness Probes.

195

# 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	



KodeKloud.com

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.

# I Kubernetes

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



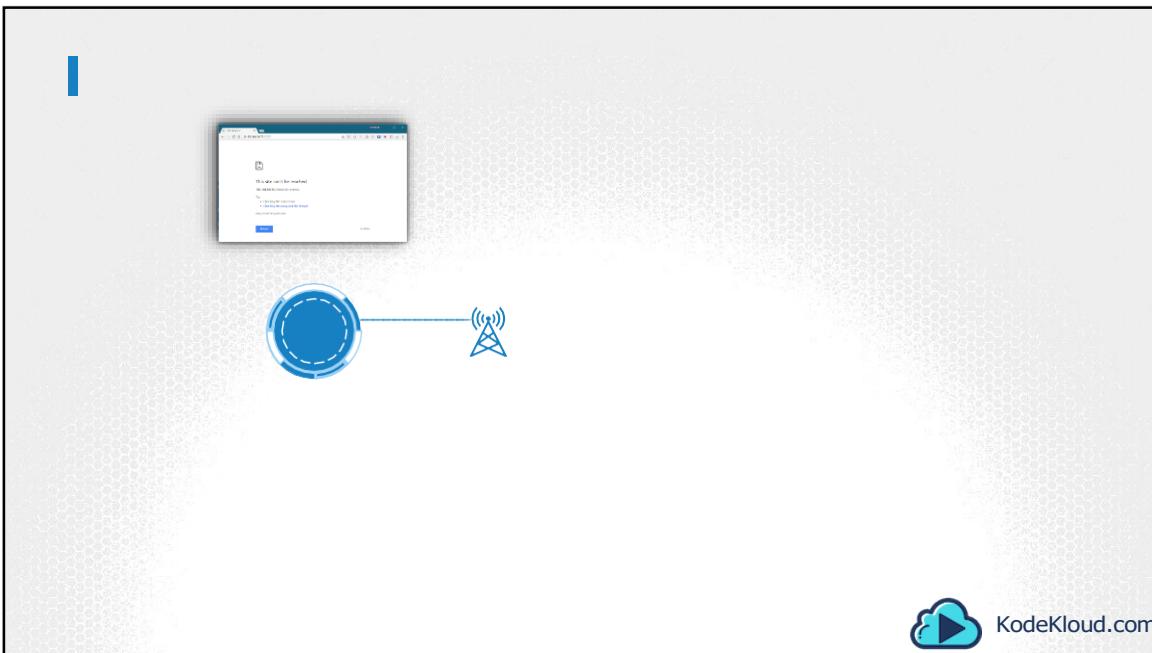
```
▶ kubectl get pods
```

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.



KodeKloud.com

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.

# I 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
```

HTTP Test - /api/ready

TCP Test - 3306

Exec Command

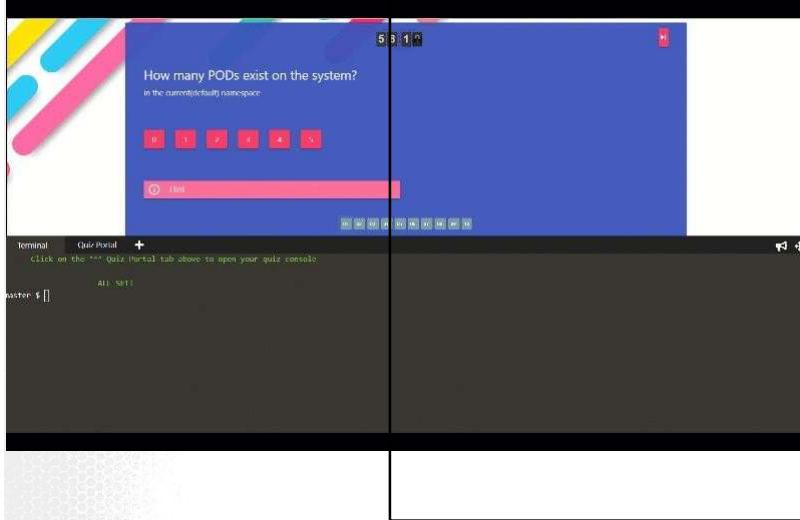
Similar to readiness probe you have httpGet option for apis, tcpSocket for ports and exec for commands. As well as additional options like initialDelay before the test is run, periodSeconds to define the frequency and success and failure thresholds.

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



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



## I 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
```



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.

## I 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.

## I 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.

## I 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

```
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
```



KodeKloud.com

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 3<sup>rd</sup> party support for logging in Kubernetes.

## Practice Test



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

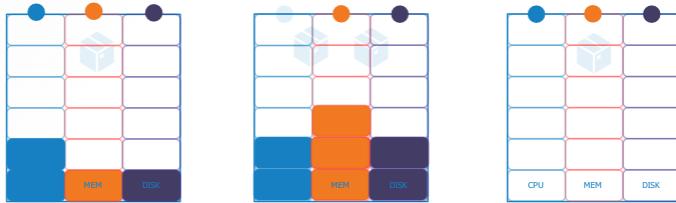
Hello and welcome to this lecture. In this lecture we will talk about the various monitoring and debugging options available.

# Monitoring Kubernetes



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

# | Monitor



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.

## I Monitor

METRICS  
SERVER



Prometheus



Monitoring Solution



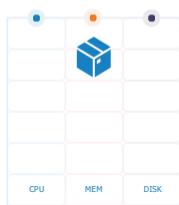
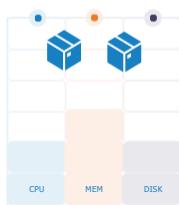
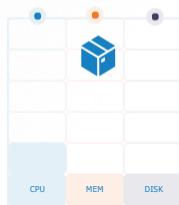
Elastic Stack



DATADOG



dynatrace



18 Oct 2018

RodekCloud.com

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.

# I 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.

# I Monitor

## METRICS SERVER



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

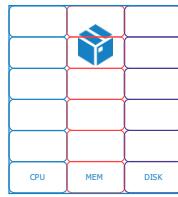
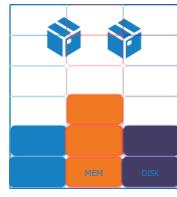
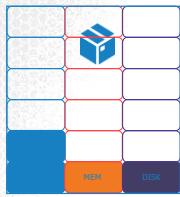


KodeKloud.com

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.

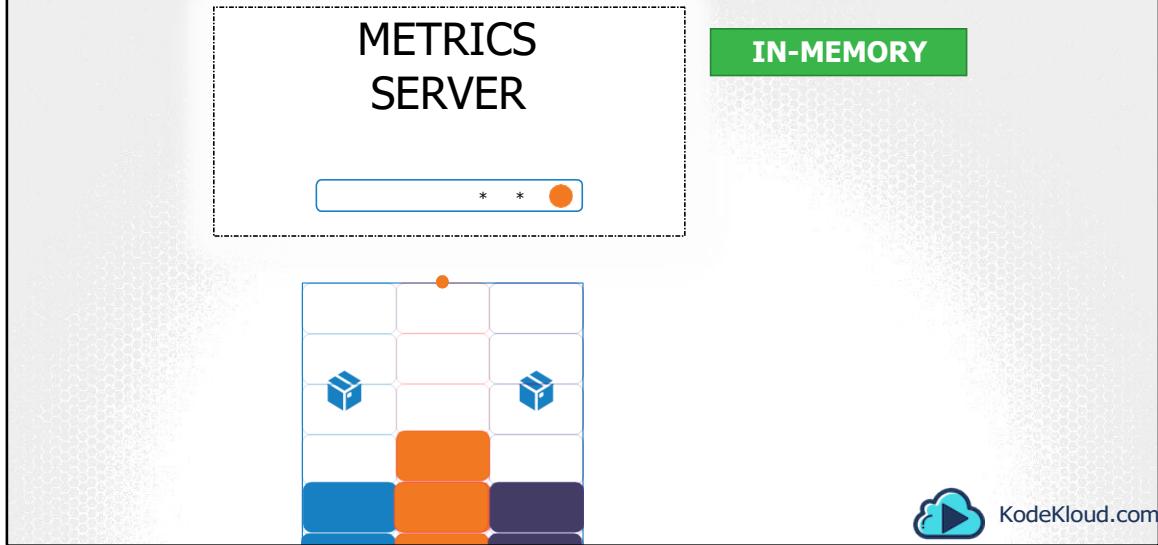
# Metrics Server

METRICS  
SERVER



You can have one metrics server per kubernetes cluster.

## I 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

### 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 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.

## I Metrics Server – Getting Started



```
▶ 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.

## IView

```
▶ 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%
```

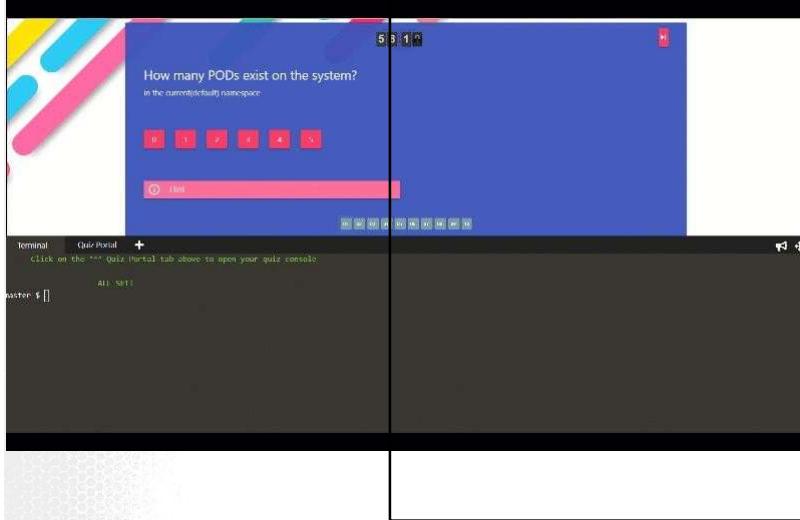
```
▶ kubectl top pod
NAME          CPU(cores)   CPU%     MEMORY(bytes)  MEMORY%
nginx         166m        8%       1337Mi        70%
redis         36m         1%       1046Mi        55%
```



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



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/>



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.

I



J 225

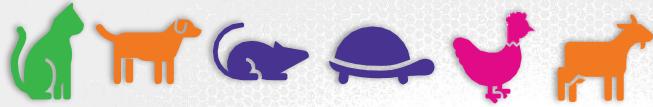
# Labels, Selectors & Annotations



KodeKloud.com

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

## I Animals



KodeKloud.com

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.

# I Class



Fish



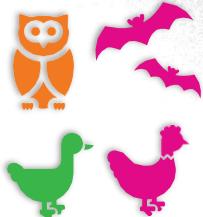
Mammals



Reptiles



Arthropods



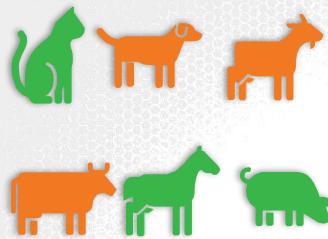
Birds



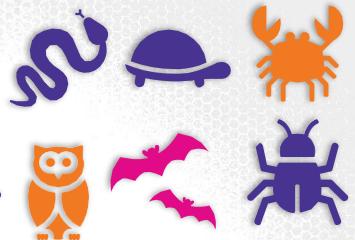
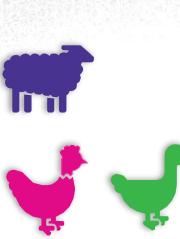
KodeKloud.com

Such as based on their class.

# I Kind



Domestic



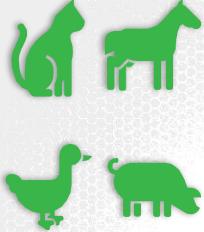
Wild



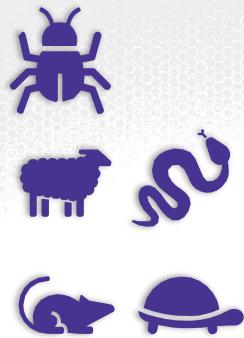
KodeKloud.com

Or based on their type – domestic or wild.

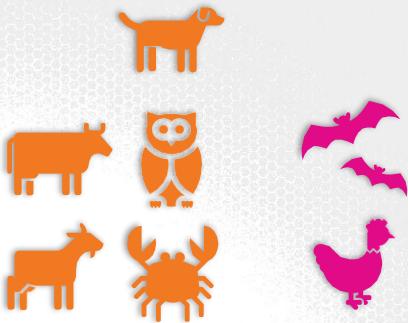
# | Color



Green



Blue



Orange



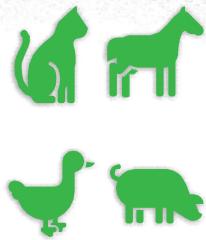
Pink



KodeKloud.com

Or say by color.

## I Color - Green

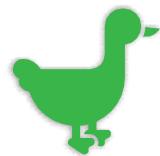


**Green**



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

## I Color – Green - Bird



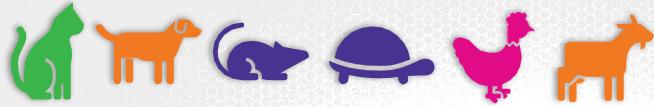
**Green - Bird**



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.

## I Labels



KodeKloud.com

Labels are properties attached to each item.

## I Labels



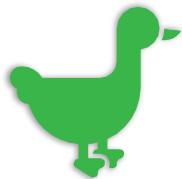
Class	Mammal
Kind	Domestic
Color	Green



Class	Reptile
Kind	Wild
Color	Purple



Class	Mammal
Kind	Domestic
Color	Orange



Class	Bird
Kind	Domestic
Color	Green



KodeKloud.com

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

# I Selectors



Class Mammal  
Kind Domestic  
Color Green



Class Reptile  
Kind Wild  
Color Purple



Class Mammal  
Kind Domestic  
Color Orange



Class Bird  
Kind Domestic  
Color Green

Class = Mammal

&

Color = Green



KodeKloud.com

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.

The screenshot shows the course structure for 'Kubernetes for the Absolute Beginners'. It includes sections for Lecture, Demo, Quiz, and Coding Exercises, along with a 'Tips & Tricks' icon. Below the course structure, there are tabs for Basic info, Translations, and Advanced settings, with 'Basic info' currently selected. The basic info section displays the course title 'Kubernetes for the Absolute Beginners' and its URL 'https://kodekcloud.com/p/kubernetes-for-the-absolute-beginners-hands-on'. It also shows a video thumbnail labeled 'Video from Course on Kubernetes for the Absolute Beginners'. At the bottom, there is a tag cloud with terms like kubernetes, docker, tutorial, training, udemy, beginners, and dummies. To the right, there is a sidebar with information about quizzes, basic information gathering, basic tasks, advanced tasks, and containerize applications, along with a link to a Docker lecture. The KodeKloud logo is at the bottom right.

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.

Refine by

Amazon Prime  prime

Global Store  global store

Price Under ₹10,000 ₹10,000 - ₹20,000 ₹20,000 - ₹30,000 ₹30,000 - ₹50,000 Over ₹50,000  ₹ Min  ₹ Max Go

Display Technology < Clear  LED  QLED  LCD  Mini LED

HD Format  4K Ultra HD  1080p Full HD  720p HD Ready

Screen Size  Up to 23"  24" - 31"  32" - 39"  40" - 47"  48" - 54"  55" & Above

TV Features  3D  Smart  Standard

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

**Sanyo 80 cm (32 inches) XT-32S7** by Sanyo ₹12,499 ₹39,999 You Save: ₹17,491 (57%) prime

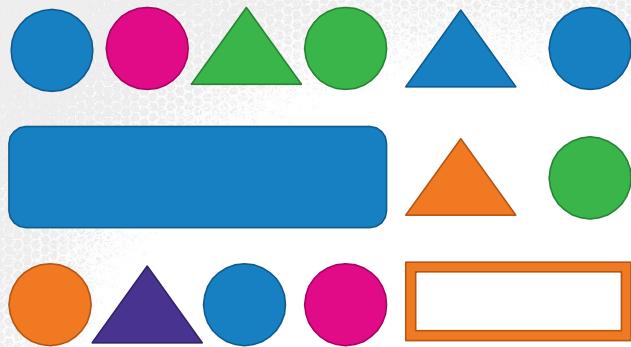
**BPL 80cm (32 inches) Vivid T32BI** by BPL ₹11,990 ₹49,999 You Save: ₹38,000 (40%) prime



KodeKloud.com

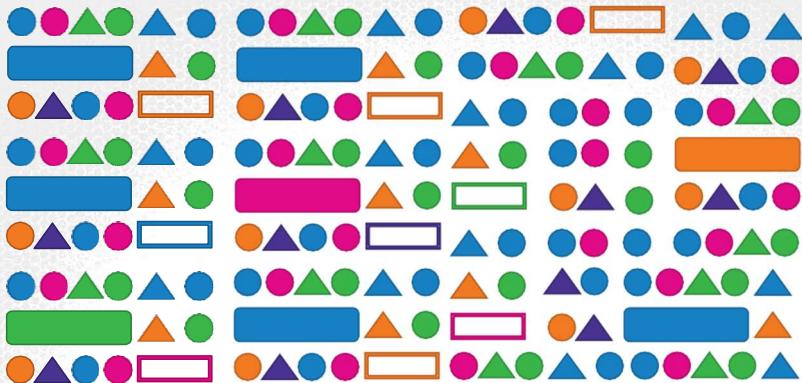
We see labels added to items in an online store that help you add different kinds of filters to view your products.

## I Labels & Selectors in Kubernetes



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

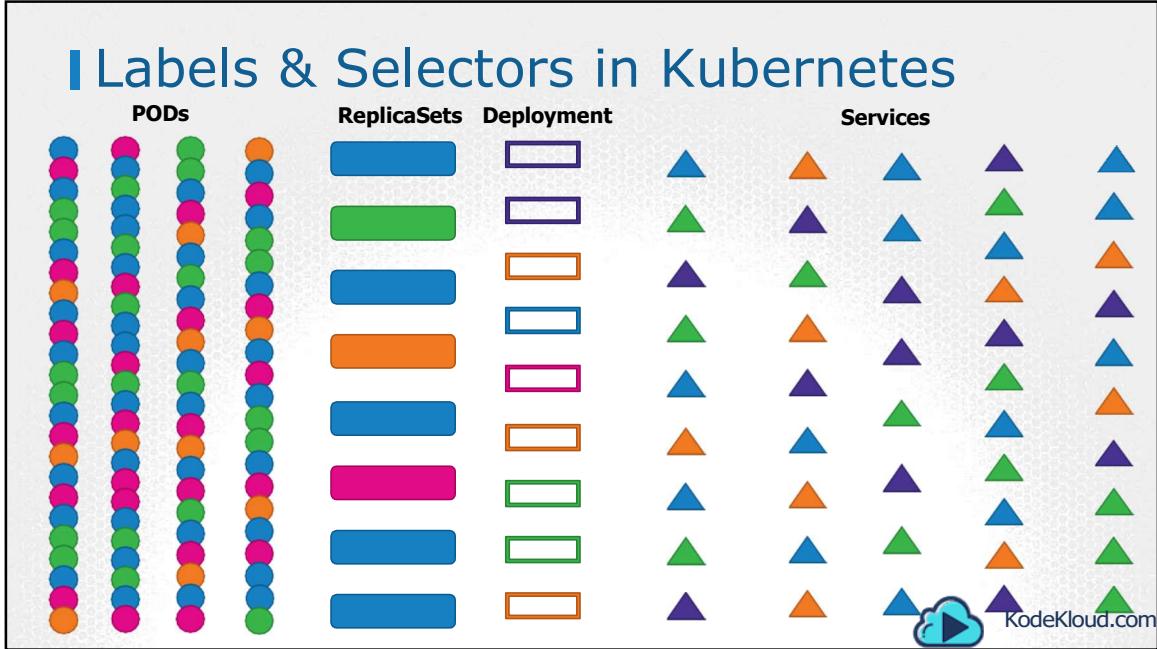
## I Labels & Selectors in Kubernetes



KodeKloud.com

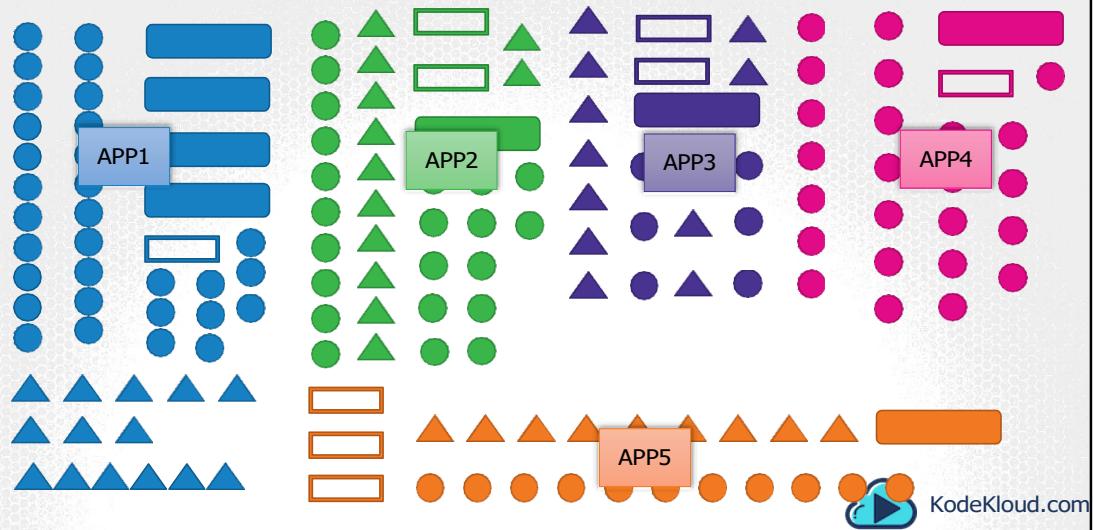
Over time you may end up having 100s and 1000s of these objects in your cluster. Then you will need a way to group, filter and view different objects by different categories.

## I Labels & Selectors in Kubernetes



Such as to group objects by their type.

## Labels & Selectors in Kubernetes

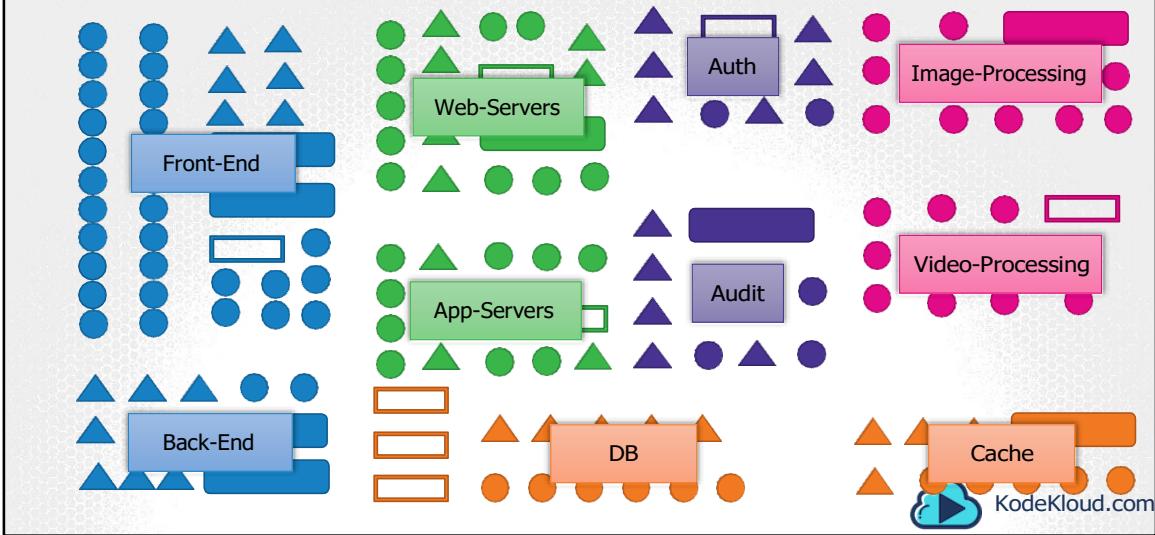


Or view objects by application.



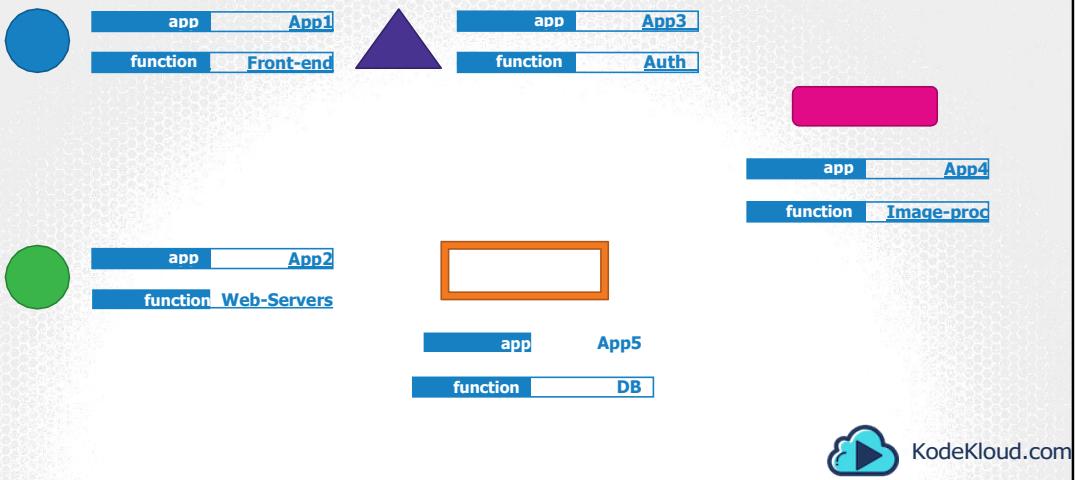
KodeKloud.com

## I 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.

## I Selectors



app = App1



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

# Labels



```
pod-definition.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
```



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.

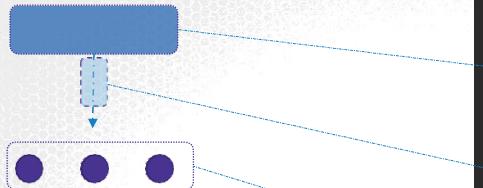
## I Select

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



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

```
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.

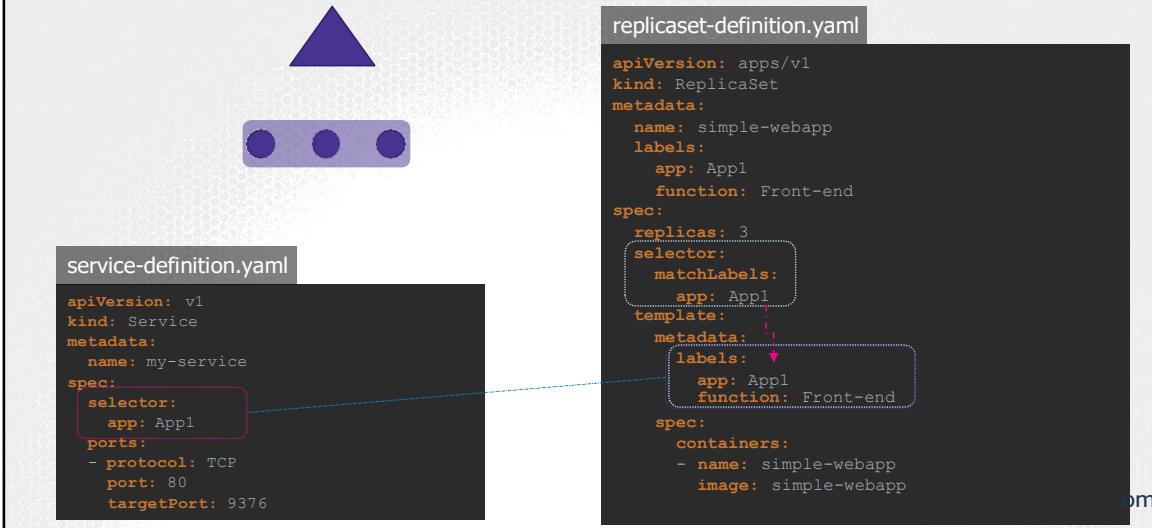
# ReplicaSet

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.

# I Service



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

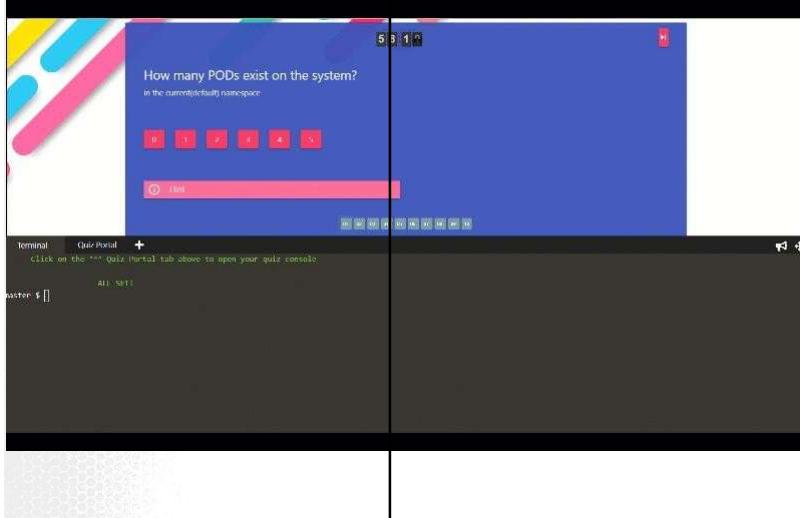
replicaset-definition.yaml

```
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 informative 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



Check Link  
below!

KodeKloud.com

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

# Rolling Updates & Rollbacks



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

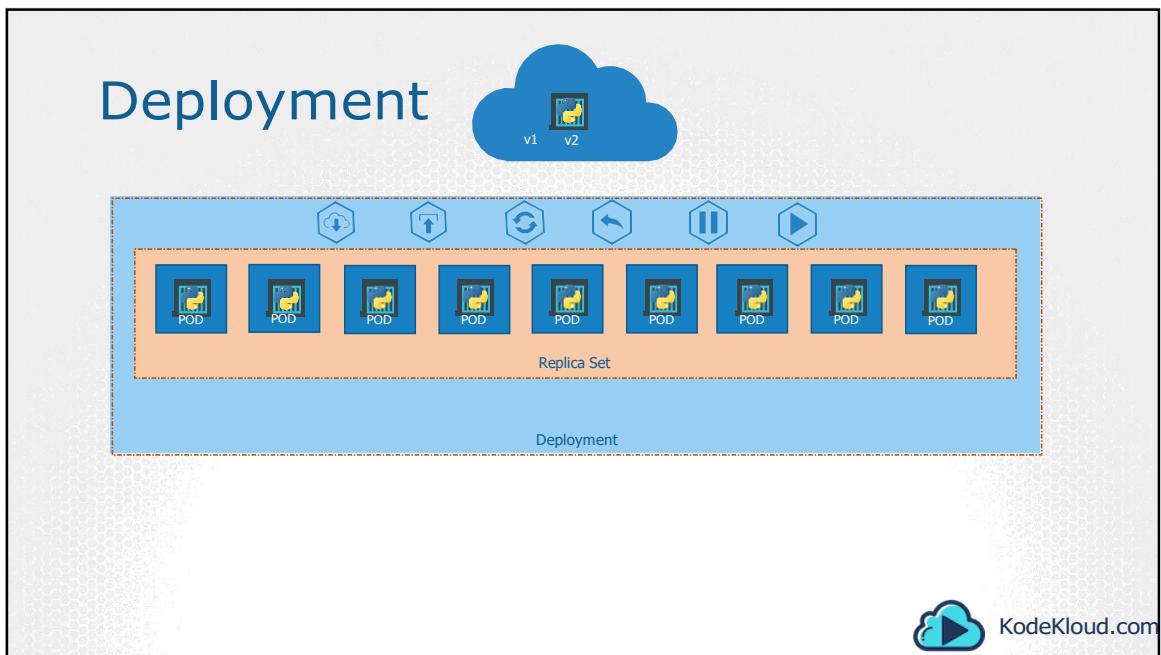
## Course Objectives

- ✓ 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.



For a minute, let us forget about PODs and replicaset 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-6795844b58-1fjhv   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 replicaset 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.

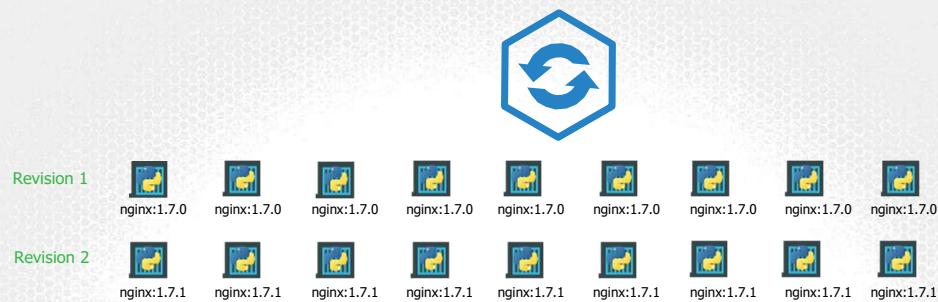
## commands

```
> 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
```



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

## Rollout and Versioning



KodeKloud.com

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

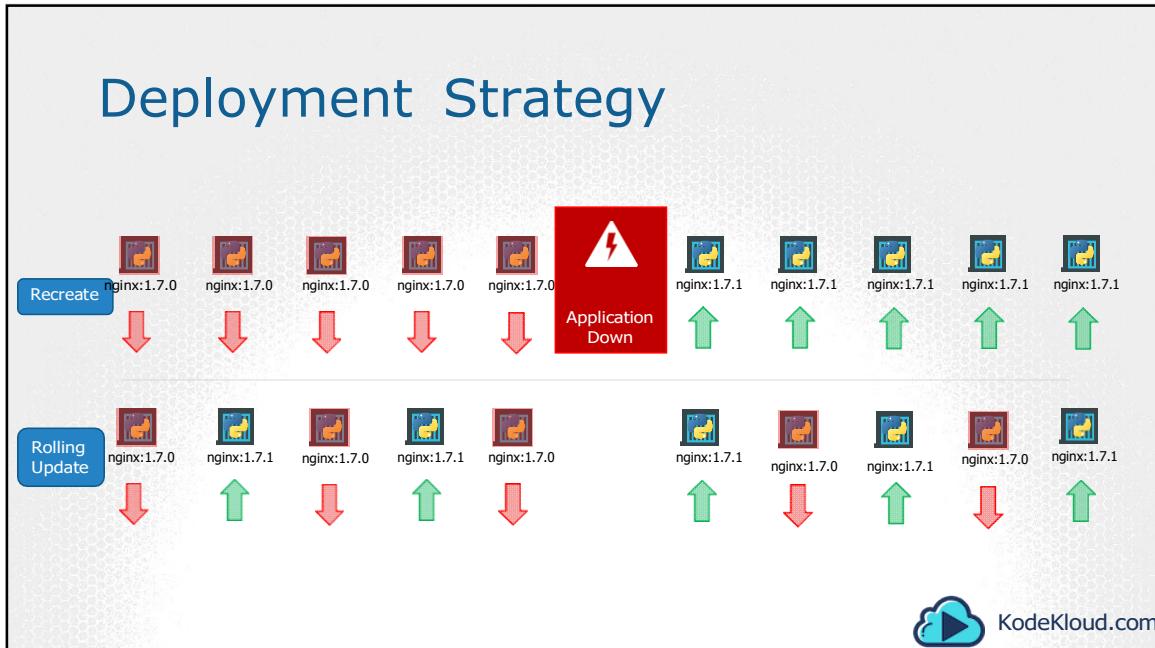
> kubectl rollout history deployment/myapp-deployment
deployments "myapp-deployment"
REVISION  CHANGE-CAUSE
1          <none>
2          kubectl apply --filename=deployment-definition.yml --record=true
```



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 where 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.

```

::\kubernetes>kubectl describe deployment myapp-deployment
Name:           myapp-deployment
Namespace:      default
CreationTimestamp:  Sat, 03 Mar 2018 17:01:55 +0000
Labels:          app=myapp
Annotations:    deployment.kubernetes.io/revision=2
                 kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"apps/v1","kind":"Deployment","metadata":{"name":"myapp-deployment","namespace":"default","labels":{"app":"myapp"},"spec":{"replicas":5,"selector":{"matchLabels":{"app":"myapp","type-front-end":true}}, "strategy":{"type":"Recreate"}, "template":{"spec":{"containers":[{"name":"nginx-container", "image": "nginx:1.7.1", "port": 80, "environment": {}, "mounts": {}, "volumes": {}}], "conditions": [{"type": "Available", "status": "True", "reason": "MinimumReplicasAvailable"}, {"type": "Progressing", "status": "True", "reason": "NewReplicaSetAvailable"}]}}, "oldReplicaSets": "none", "newReplicaSet": "myapp-deployment-54c7d6ccc (5/5 replicas created)"}
Events:          <none>

```

```

::\kubernetes>kubectl describe deployment myapp-deployment
Name:           myapp-deployment
Namespace:      default
CreationTimestamp:  Sat, 03 Mar 2018 17:16:53 +0000
Labels:          app=myapp
Annotations:    deployment.kubernetes.io/revision=2
                 kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"apps/v1","kind":"Deployment","metadata":{"name":"myapp-deployment","namespace":"default","labels":{"app":"myapp","type-front-end":true}}, "spec":{"replicas":5,"selector":{"matchLabels":{"app":"myapp","type-front-end":true}}, "strategy":{"type":"RollingUpdate"}, "template":{"spec":{"containers":[{"name":"nginx-container", "image": "nginx:1.7.1", "port": 80, "environment": {}, "mounts": {}, "volumes": {}}], "conditions": [{"type": "Available", "status": "True", "reason": "MinimumReplicasAvailable"}, {"type": "Progressing", "status": "True", "reason": "ReplicaSetUpdated"}]}}, "oldReplicaSets": "none", "newReplicaSet": "myapp-deployment-7057dbdd0 (0/5 replicas created)"}
Events:          <none>

```

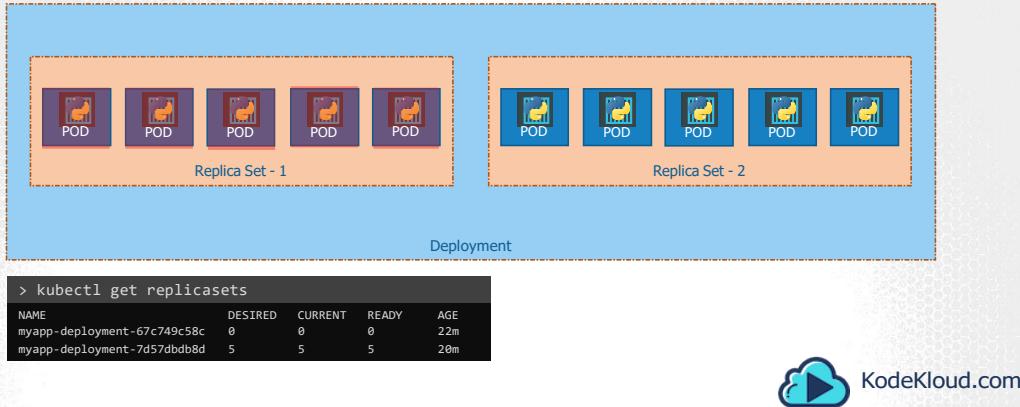
Recreate

RollingUpdate



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

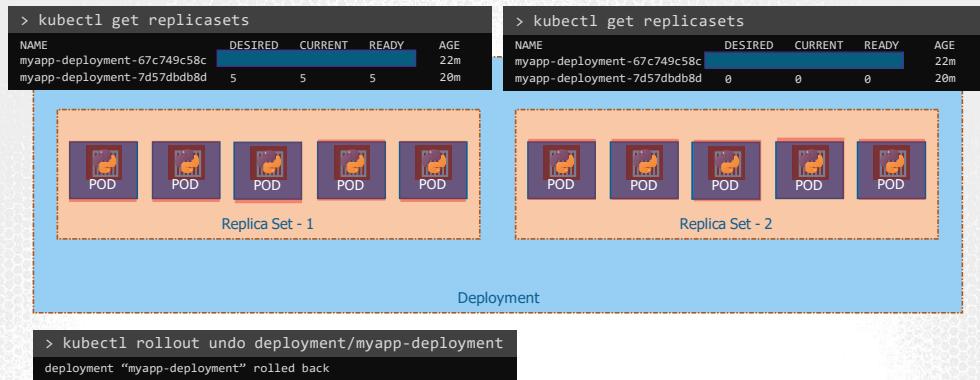


KodeKloud.com

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 replicasesets using the `kubectl get replicaset` command. Here we see the old replicaset with 0 PODs and the new replicaset with 5 PODs.

## Rollback



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 replicaset`s 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
```



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

```
> kubectl create -f deployment-definition.yml
```

Get

```
> kubectl get deployments
```

Update

```
> kubectl apply -f deployment-definition.yml
```

Status

```
> kubectl rollout status deployment/myapp-deployment
```

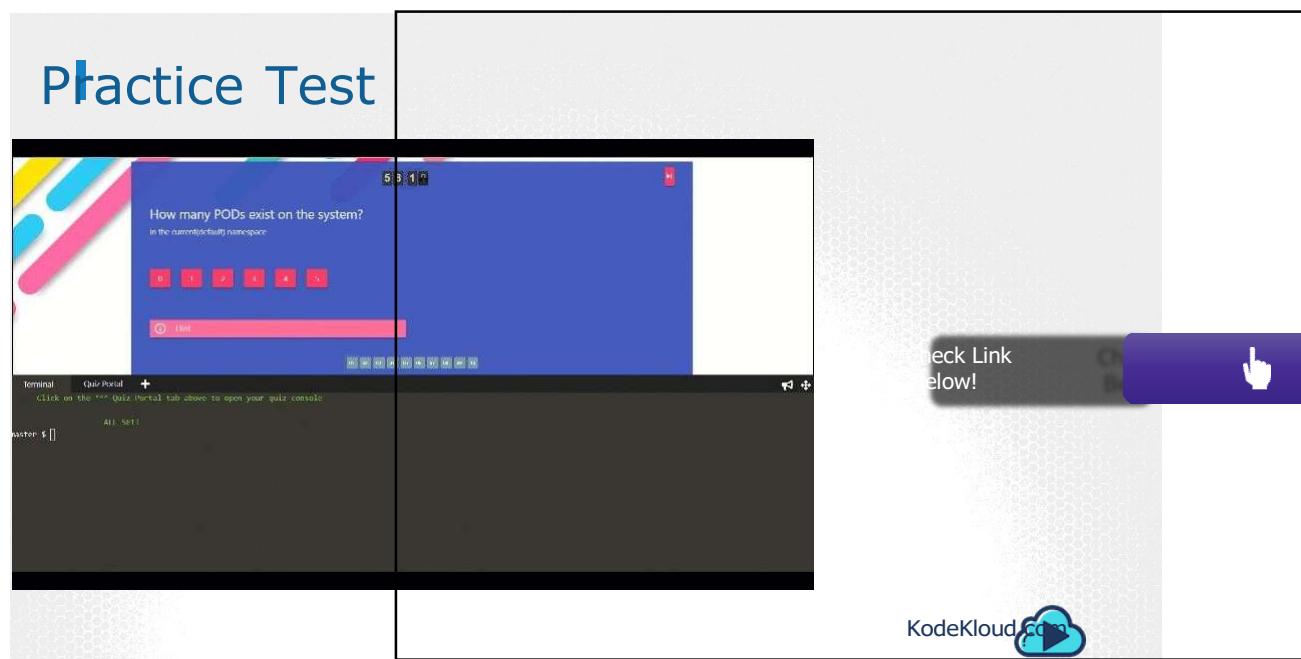
Rollback

```
> kubectl rollout history deployment/myapp-deployment
```



KodeKloud.com

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.



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



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

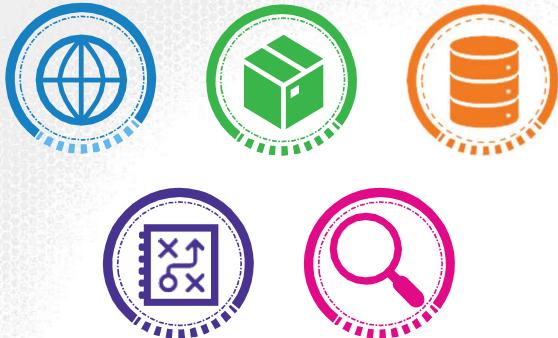
# Jobs



KodeKloud.com

Let us start with Jobs in Kubernetes.

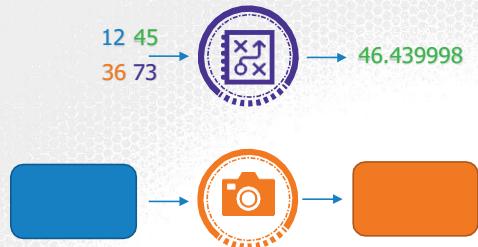
## Types of Workloads



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.

## I 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
```



```
▶ docker ps -a
```

CONTAINER ID	IMAGE	CREATED	STATUS	PORTS
45aacc36850	ubuntu	43 seconds ago	Exited (0) 41 seconds ago	



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
```

3 2  
3 2  
3 2



5  
5  
5

```
pod-definition.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



KodeKloud.com

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?

# I RestartPolicy

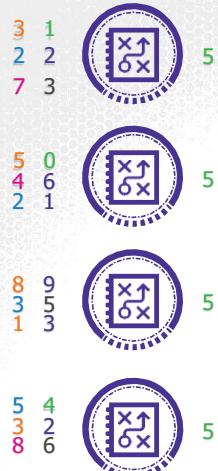
3 2      5

```
pod-definition.yaml
apiVersion: v1
kind: Pod
metadata:
  name: math-pod
spec:
  containers:
    - name: math-add
      image: ubuntu
      command: ['expr', '3', '+', '2']
  restartPolicy: Never
```



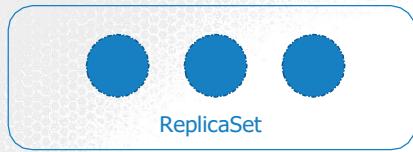
Kubernetes wants your applications to live forever. The default behavior of PODs is to attempt to restart the container in an effort to keep it running. This behavior is defined by the property `restartPolicy` set on the POD, which is by default set to `Always`. And that is why the POD `ALWAYS` recreates 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.

## I RestartPolicy



We have new use cases for batch processing. We have large data sets that require 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 gets done successfully.

## I Kubernetes Jobs



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 replicaset or deployments, we have template. And under template we move all of the content from pod definition specification.

## Create, View & Delete

```
job-definition.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-187pn
```

```
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, run the kubectl 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



KodeKloud.com

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-kmttt	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-nmgph	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-ktmmt	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-nmgph	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 complete 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.



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

# CronJobs



KodeKloud.com

Let us now look at CronJobs in Kubernetes.

## CronJob

```
job-definition.yaml
```

```
apiVersion: batch/v1
kind: Job
metadata:
  name: reporting-cron-job
spec:
  completions: 1
  parallelism: 1
  template:
    spec:
      containers:
        - name: reporting-tool
          image: reporting-tool
      restartPolicy: Never
```

```
cron-job-definition.yaml
```

```
apiVersion: batch/v1beta1
spec:
  schedule: "*/5 * * * *"
  jobTemplate:
    spec:
      containers:
        - name: reporting-tool
          image: reporting-tool
```

Wikipedia

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.

282

## Create CronJob

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

```
kubectl get cronjob
```

NAME	SCHEDULE	SUSPEND	ACTIVE
reporting-cron-job	*/1 * * * *	False	0

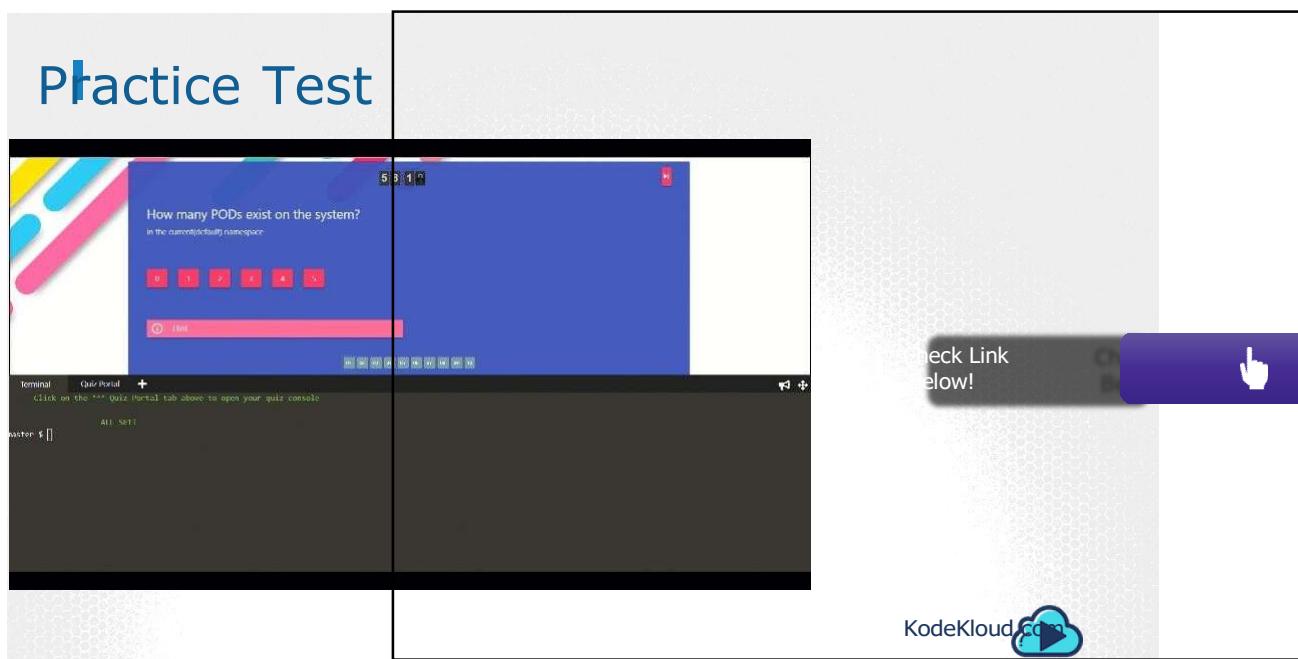
```
cron-job-definition.yaml
```

```
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
```

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.



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

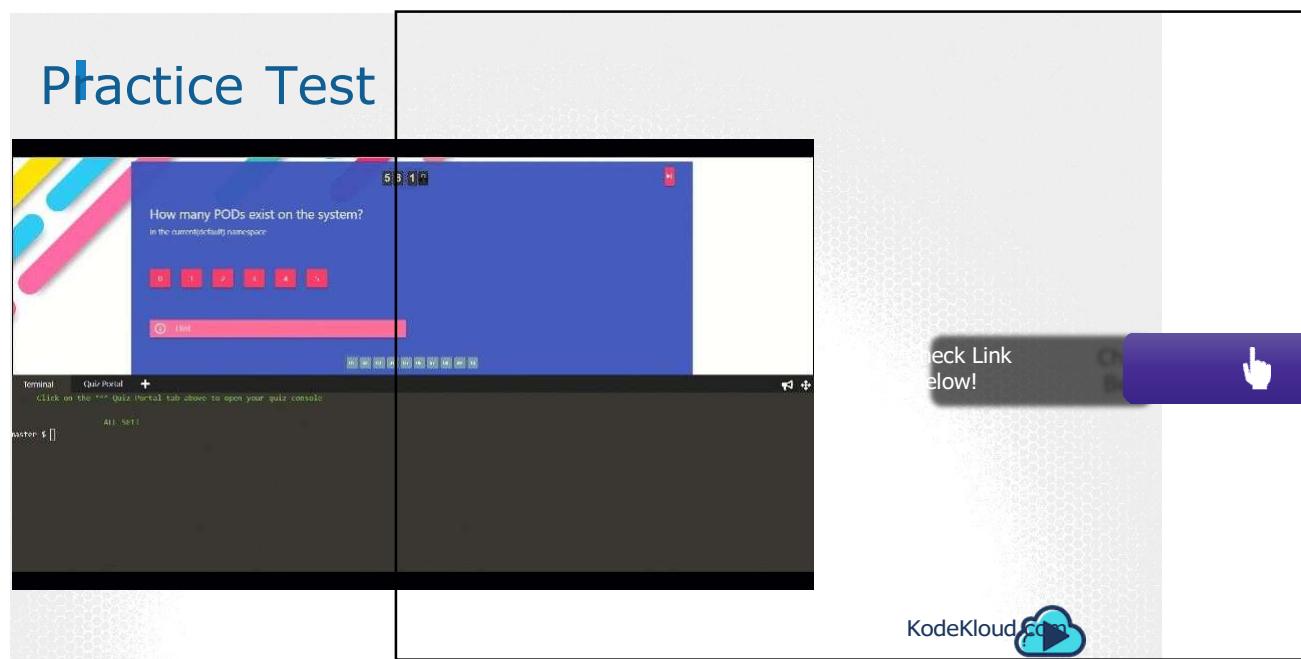


KodeKloud.com

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

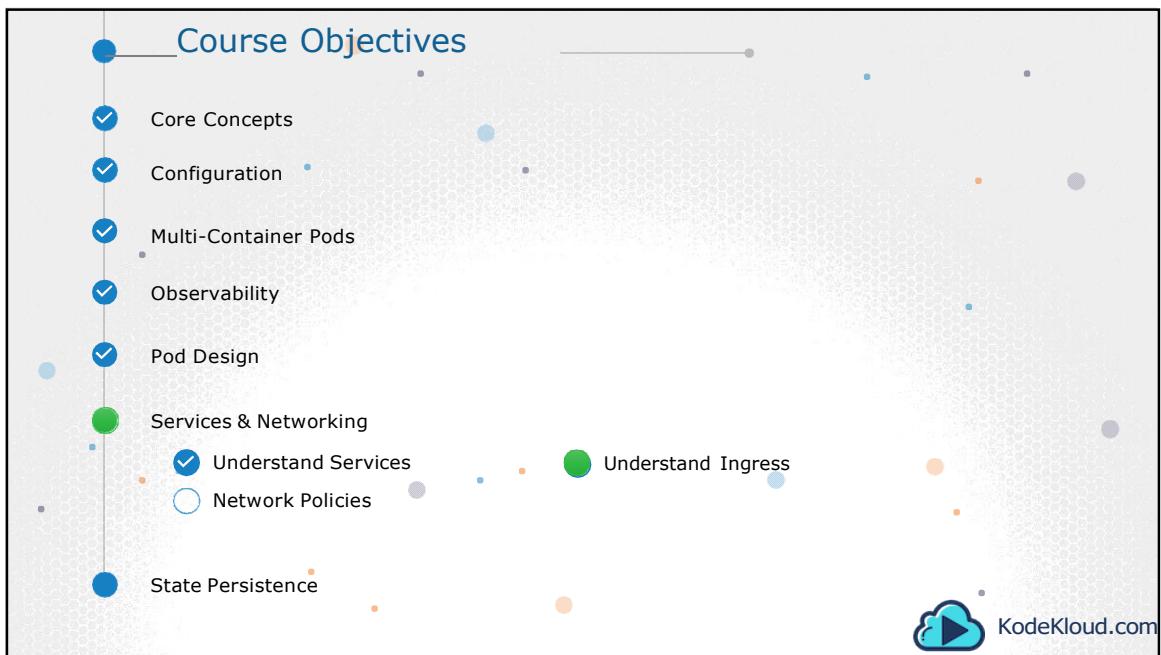


Hello and welcome to this lecture. We are going through the Certified Kubernetes Application Developers course. Services were discussed in the beginners course, so nothing more to add here. Checkout the practice tests.



KodeKloud.com

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



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

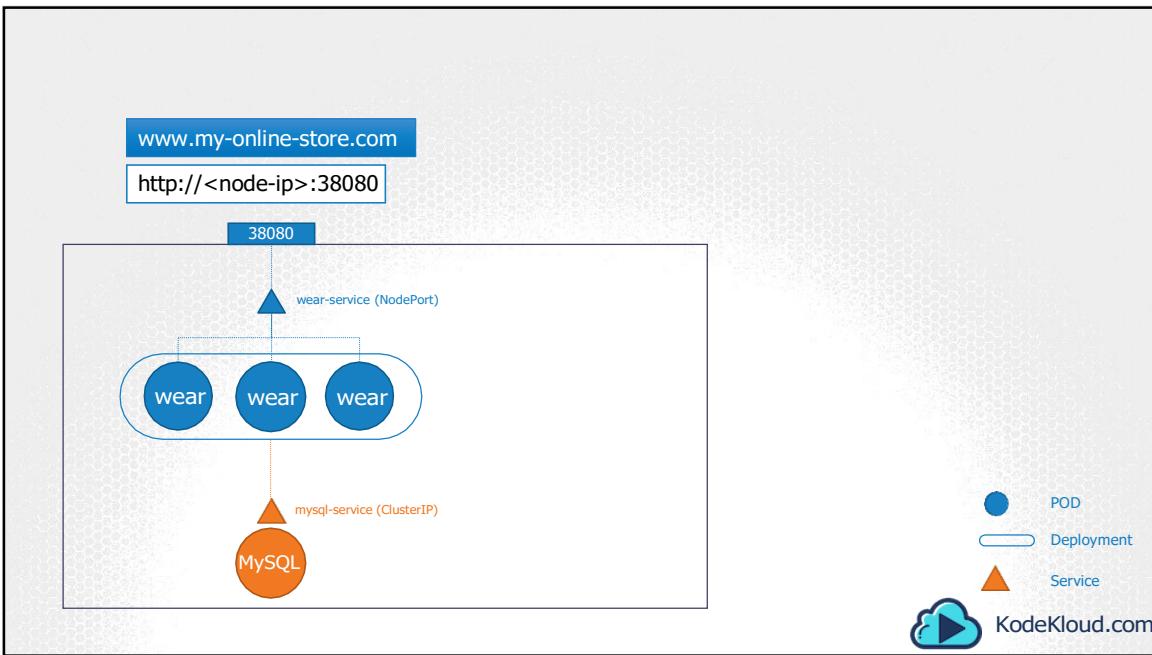


KodeKloud.com

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.

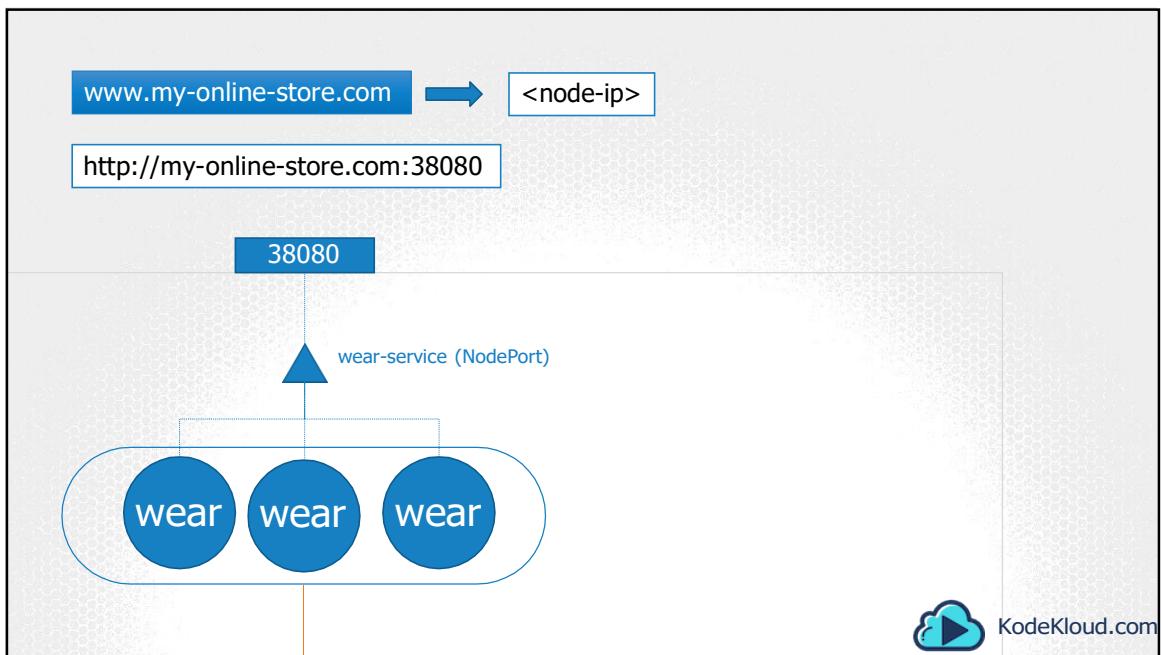


You build the application into a Docker Image and deploy it on the Kubernetes cluster as a POD in a Deployment. Your application needs a database so you deploy a MySQL database as a POD and create a service of type ClusterIP called mysql-service to make it accessible to your application. Your application 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.

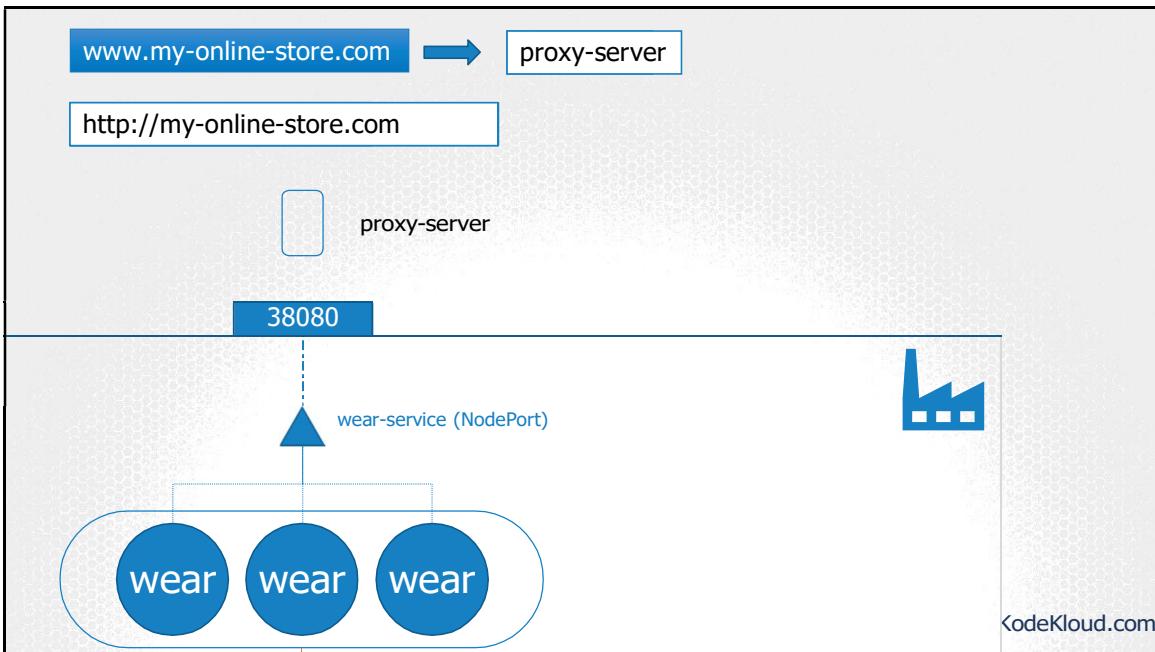
Whenever traffic increases, we increase the number of replicas of the POD to handle the additional traffic, and the service takes care of splitting traffic between the PODs.

<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.

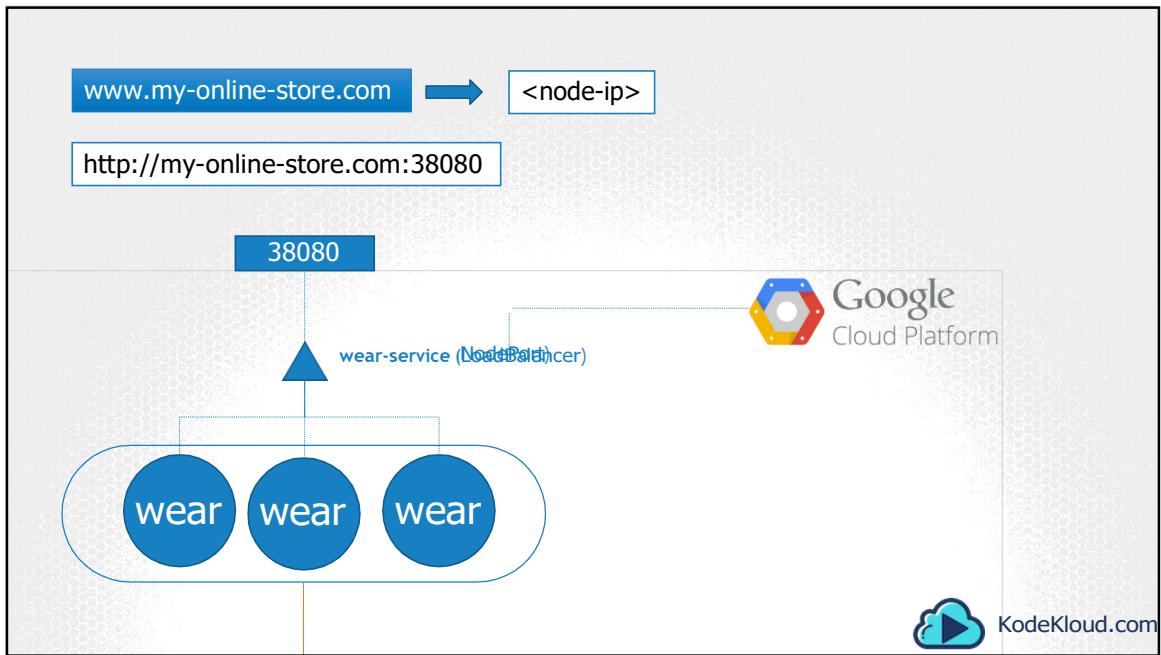


For example, we do not want the users to have to type in IP address everytime. So you configure your DNS server to point to the IP of the nodes. Your users can now access your application using the URL `http://my-online-store.com:38080`.



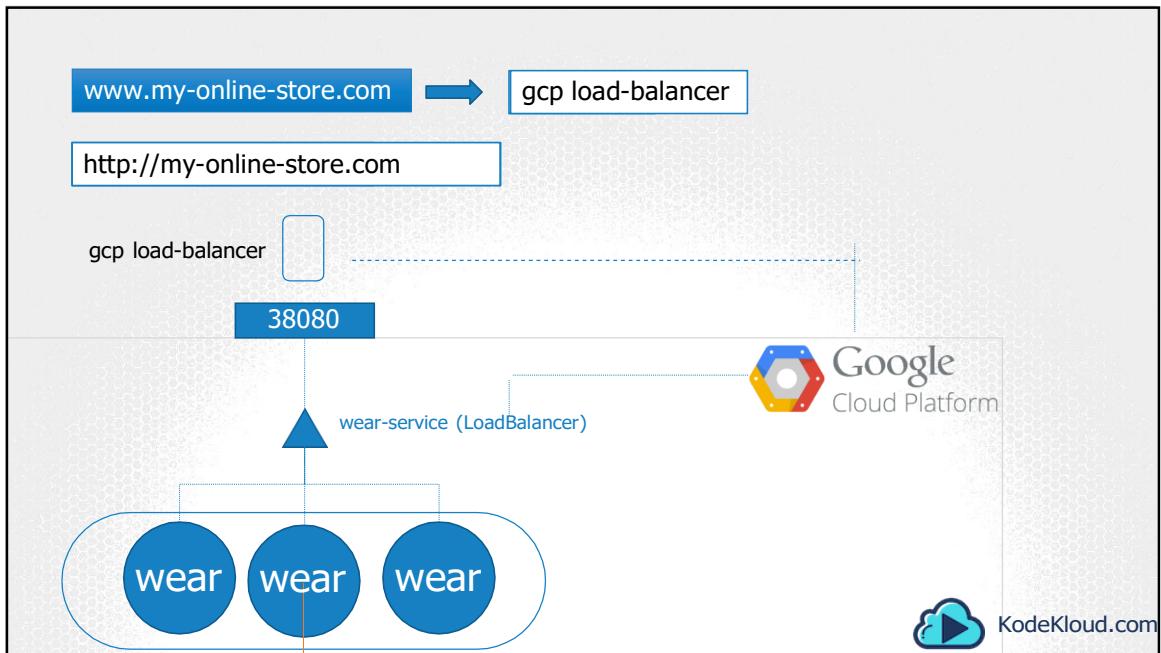
Now, you don't want your users to have to remember port number either. However, Service NodePorts can only allocate high numbered ports which are greater than 30,000. So you then bring in an additional layer between the DNS server and your cluster, like a proxy server, that proxies requests on port 80 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.

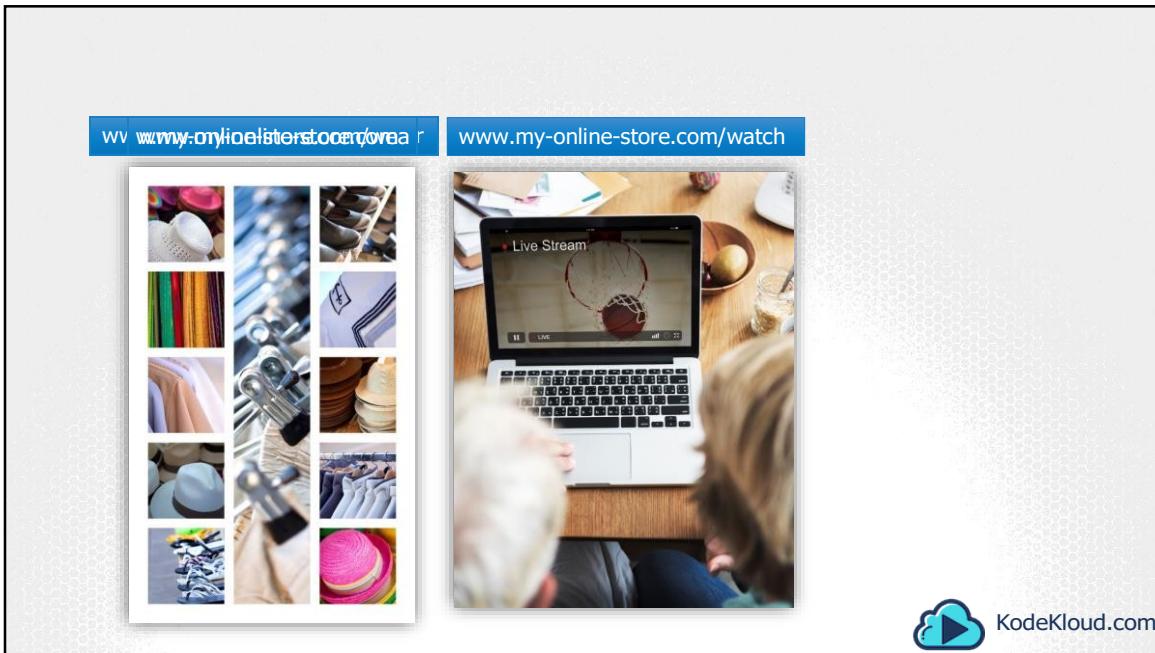


Let's take a step back and see what you could do if you were on a public cloud environment like Google Cloud Platform.

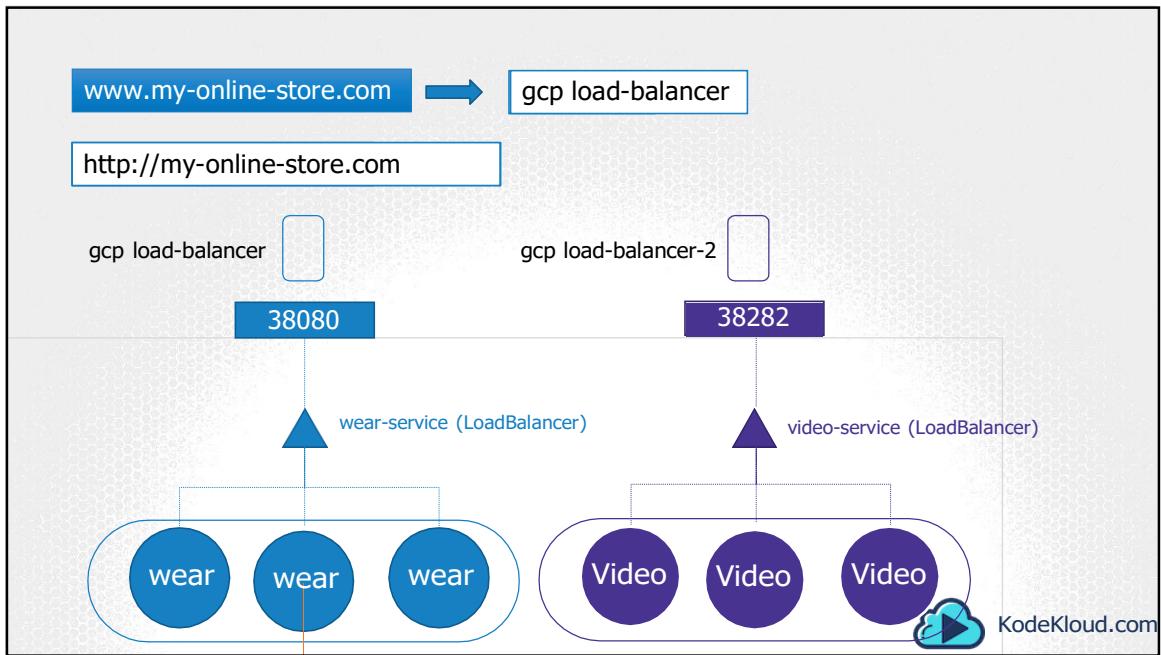
In that case, instead of creating a service of type NodePort for your `wear` application, you could set it to type LoadBalancer. When you do that Kubernetes would still do everything that it has to do for a NodePort, which is to provision a high port for the service, but in addition to that Kubernetes 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!!

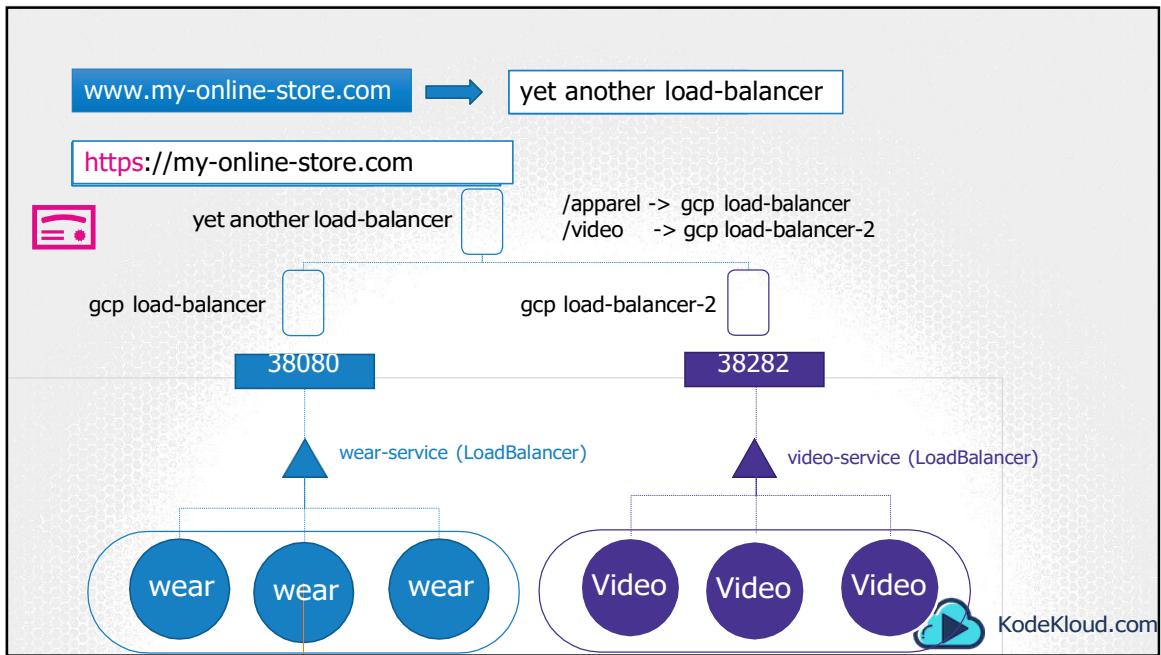


Your company's business grows and you now have new services for your customers. For example, a video streaming service. You want your users to be able to access your new video streaming service by going to [my-online-store.com/watch](http://my-online-store.com/watch). You'd like to make your old application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

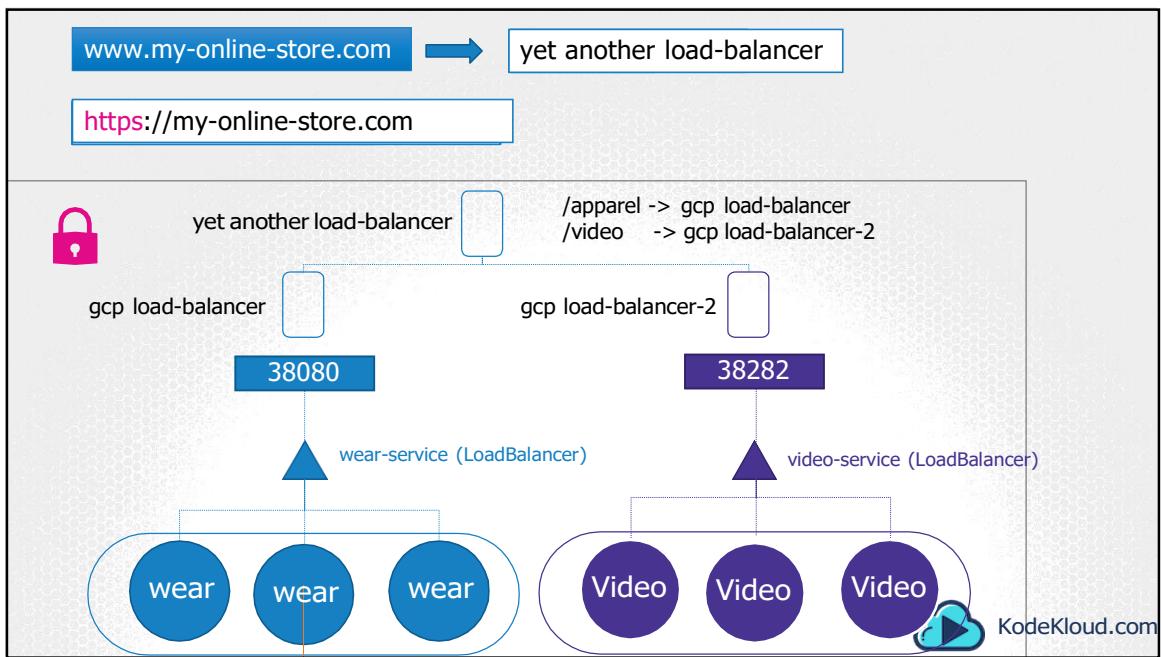
So how do you direct traffic between each of these load balancers based on URL?



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.

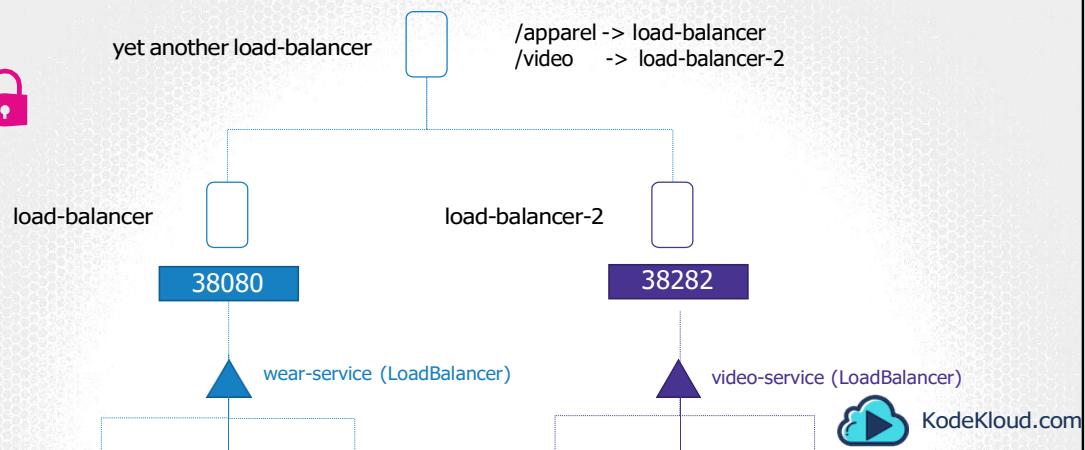
And finally you also need to enable SSL for your applications, so your users can access your application using https. Where do you configure that?

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.



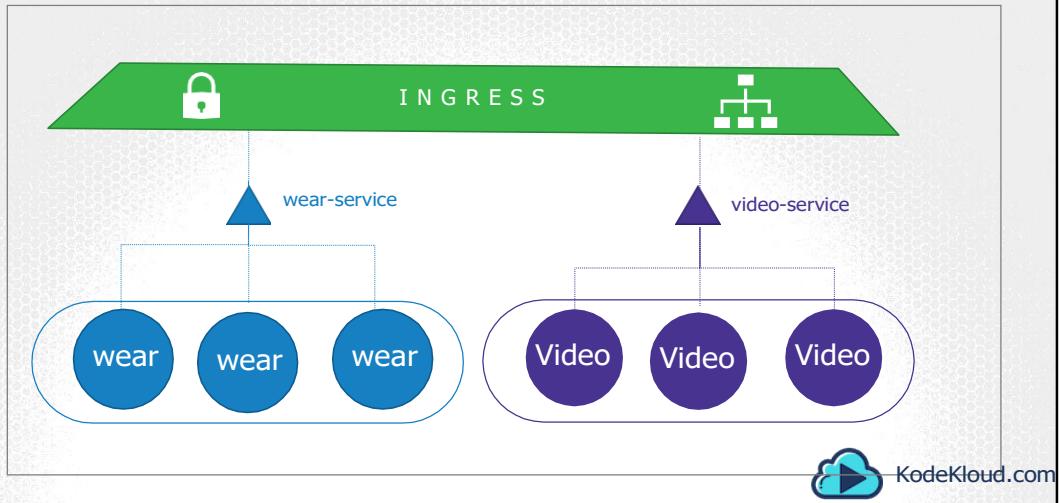
Wouldn't it be nice if you could manage all of that within the Kubernetes cluster, and have all that configuration as just another kubernetes definition file, that lives along with the rest of your application deployment files?

## Ingress



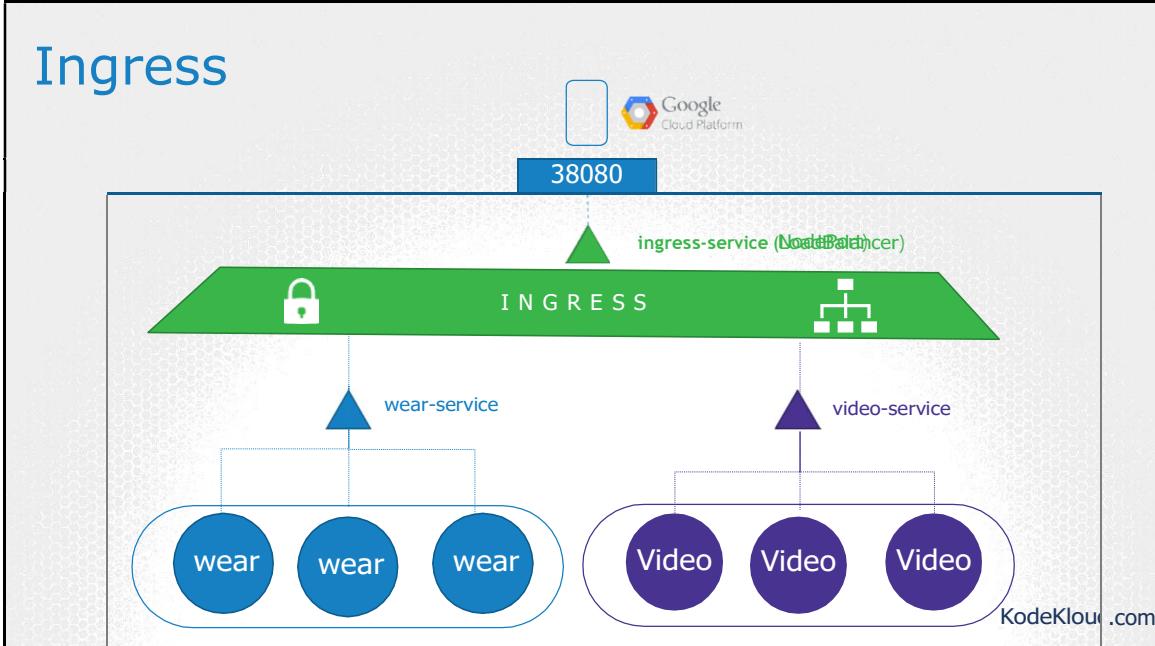
That's where Ingress comes into play. Ingress helps your users access your application using a single Externally accessible URL, that you can configure to route to different services within your cluster based on the URL path, at the same time terminate TLS.

## Ingress



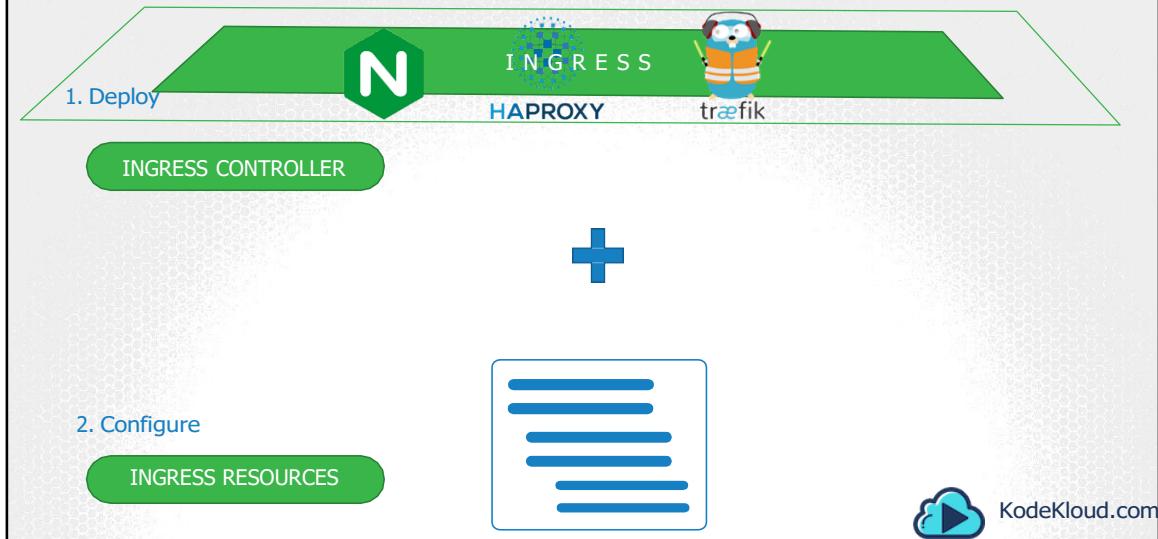
Simply put, think of ingress as a layer 7 load balancer built-in to the kubernetes cluster that can be configured using native kubernetes primitives just like any other object in kubernetes.

## 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.

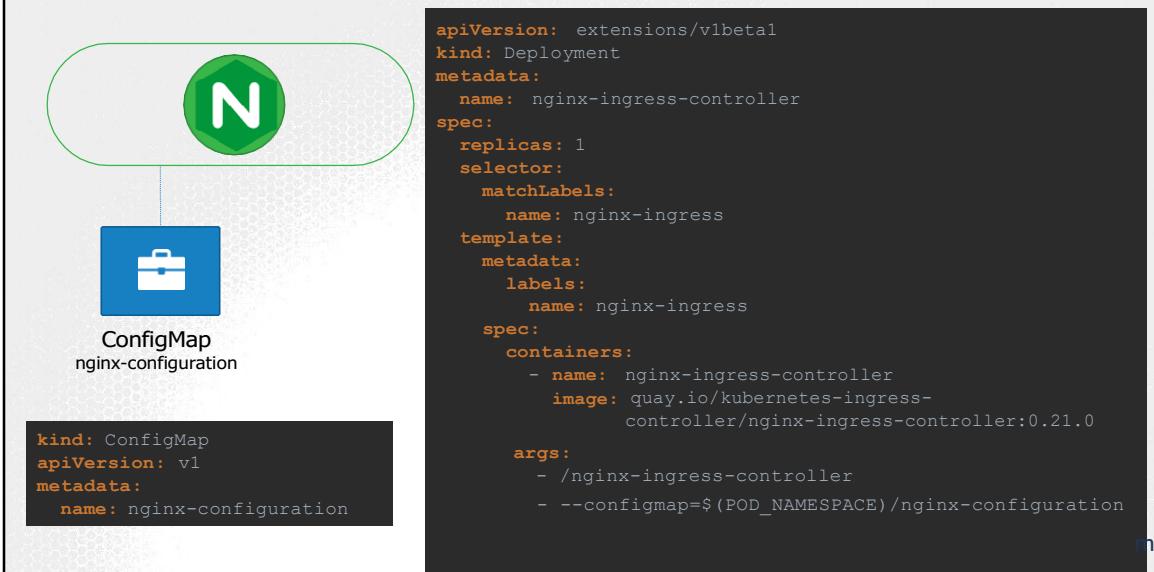
Let's look at each of these in a bit more detail now.

# INGRESS CONTROLLER



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



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`.

## INGRESS CO



ConfigMap  
nginx-configuration

```
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 CO



ConfigMap  
nginx-configuration

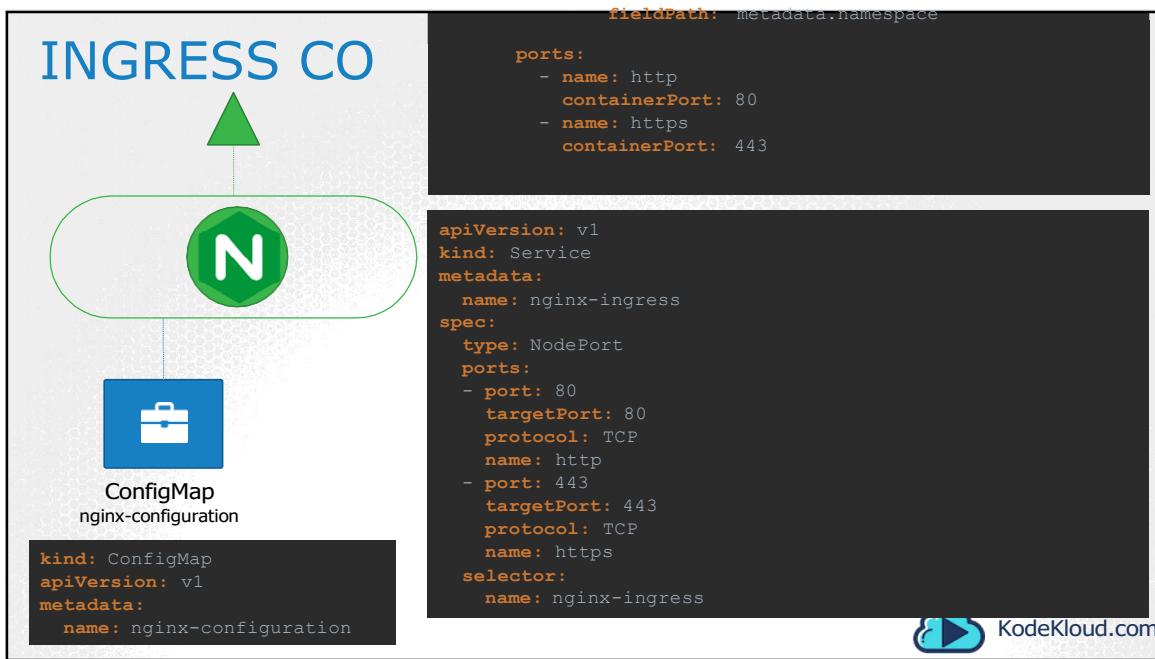
```
kind: ConfigMap
apiVersion: v1
metadata:
  name: nginx-configuration
```

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



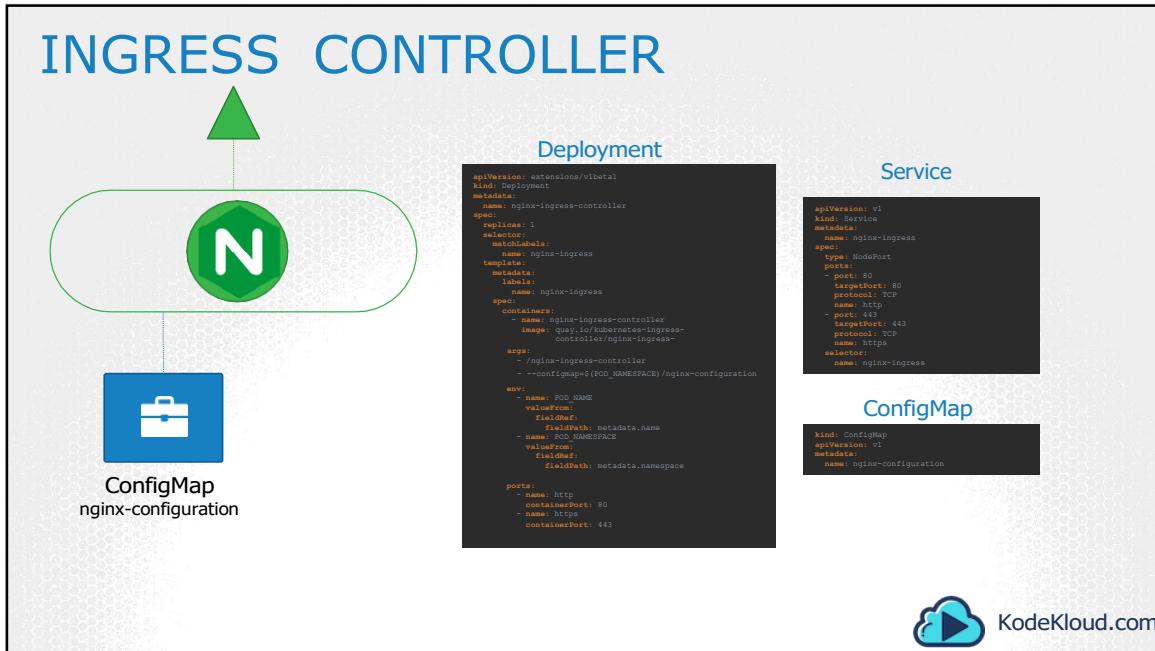
KodeKloud.com

And finally specify the ports used by the ingress controller.



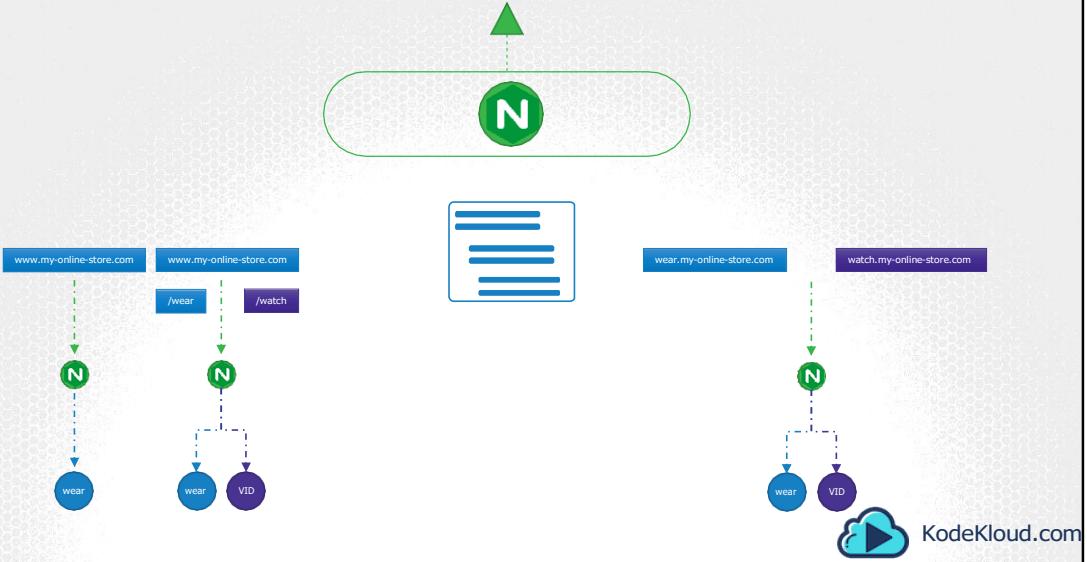
We then need a service to expose the ingress controller to the external world. So we create a service of type NodePort with the nginx-ingress label selector to link the service to the deployment. So with these three objects we should be ready with an ingress controller in its simplest form.

# INGRESS CONTROLLER



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

```
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



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 4<sup>th</sup> 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
<a href="#">www.my-online-store.com</a>	10.123.23.12 (INGRESS SERVICE)
<a href="#">www.wear.my-online-store.com</a>	10.123.23.12
<a href="#">www.watch.my-online.store.com</a>	10.123.23.12
<a href="#">www.my-wear-store.com</a>	10.123.23.12
<a href="#">www.my-watch-store.com</a>	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



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.

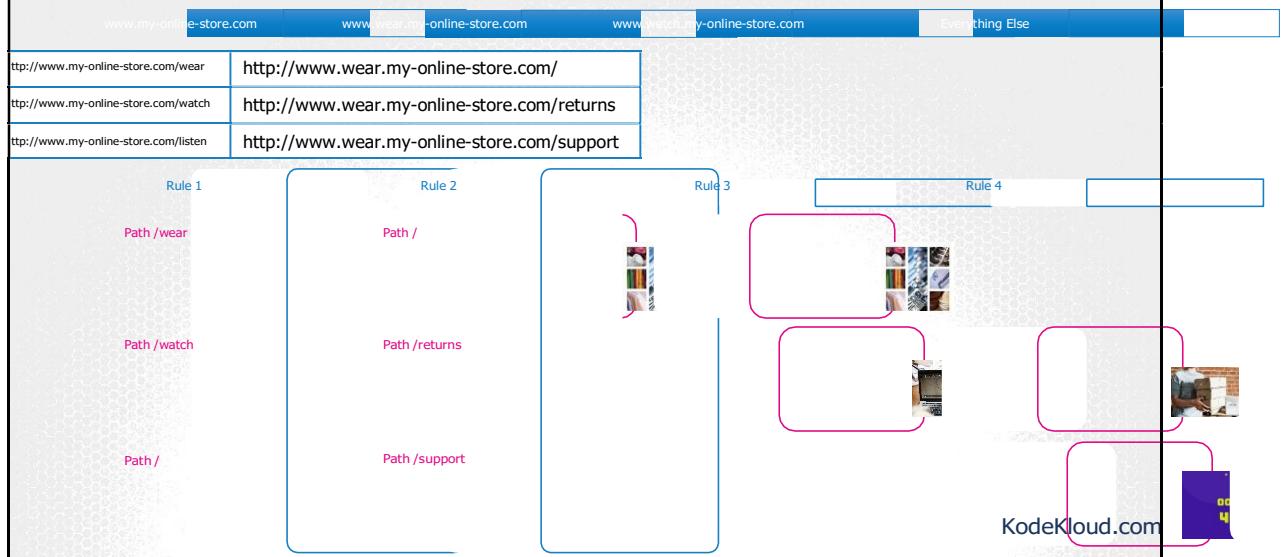
## INGRESS RESOURCE - RULES



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



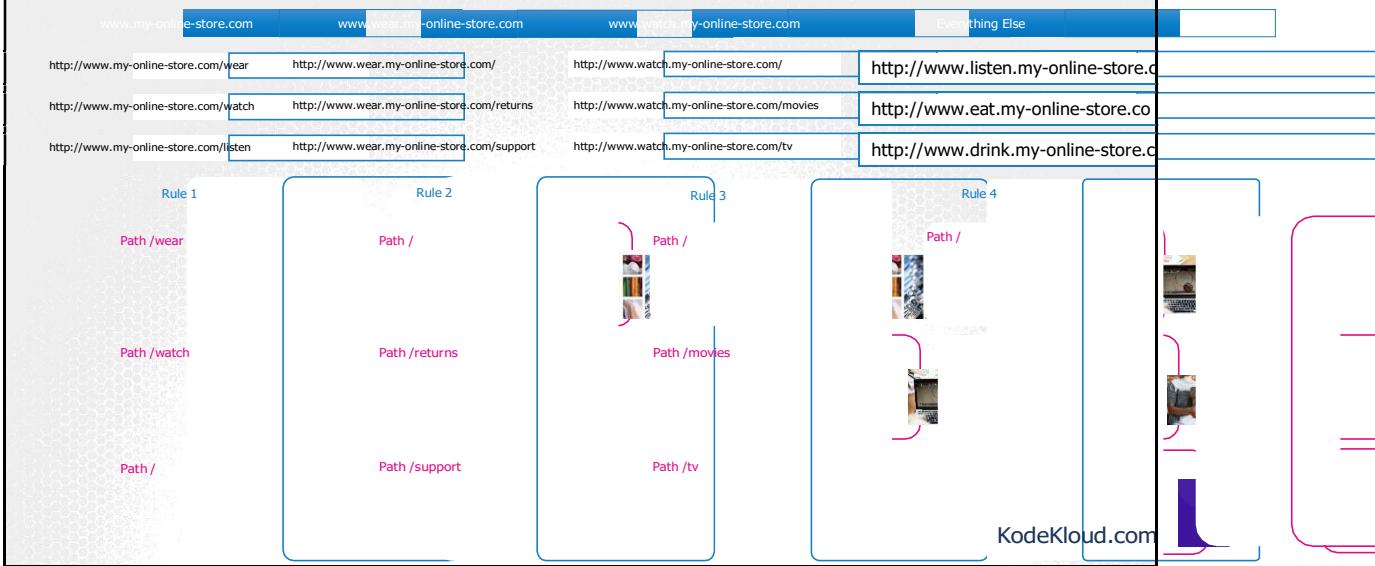
Similarly, the second rule handles all traffic from `wear.my-online-store.com`. You can have path definition within this rule, to route traffic based on different paths. For example, say you have different applications and services within the apparel section for shopping, or returns, or support, when 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



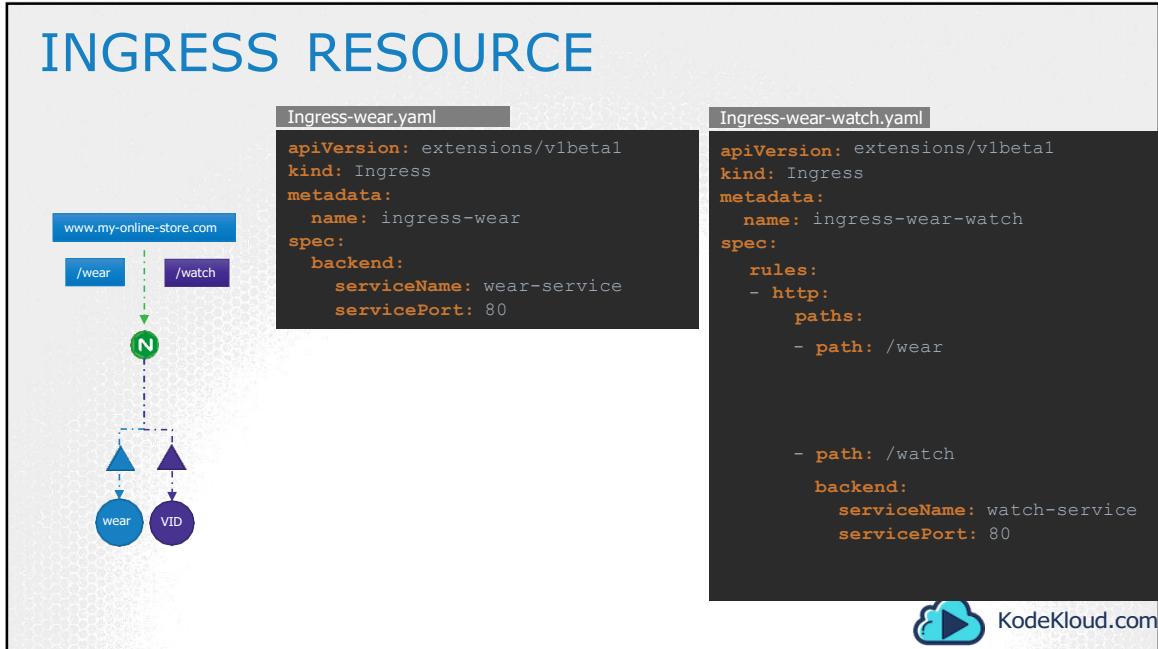
The same goes for Rule 3, where you route traffic to `watch.my-online-store.com` to the video streaming application. But you can have additional paths in it such as movies or tv.

## INGRESS RESOURCE - RULES



And finally anything other than the ones listed will go to the 4<sup>th</sup> 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



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
  ----     -----  --   --   Ingress default/ingress-wear-watch
  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. Hmm. 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](http://www.my-online-store.com/eat)

[www.my-online-store.com/watch](http://www.my-online-store.com/watch)



[www.my-online-store.com/eat](http://www.my-online-store.com/eat)

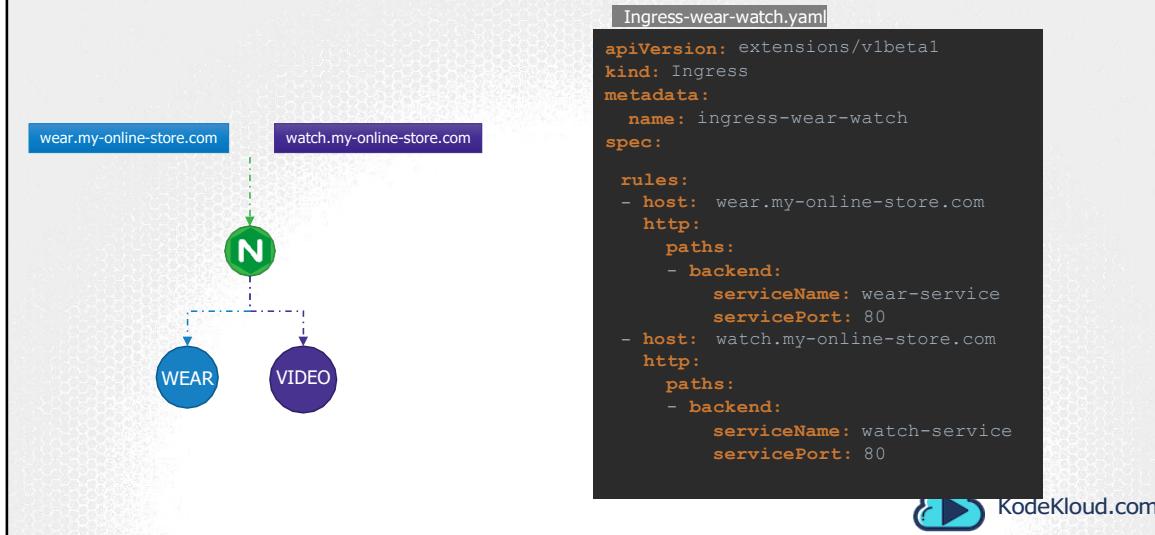
[www.my-online-store.com/listen](http://www.my-online-store.com/listen)



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



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



Hello and welcome to this lecture on Network Policies.

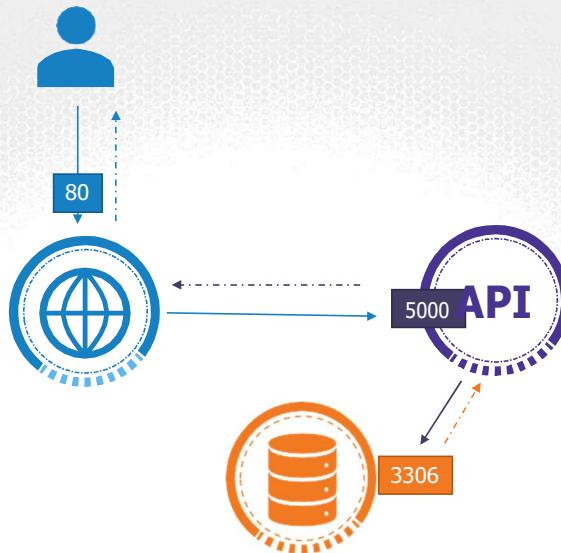
326

# Network Policies



KodeKloud.com

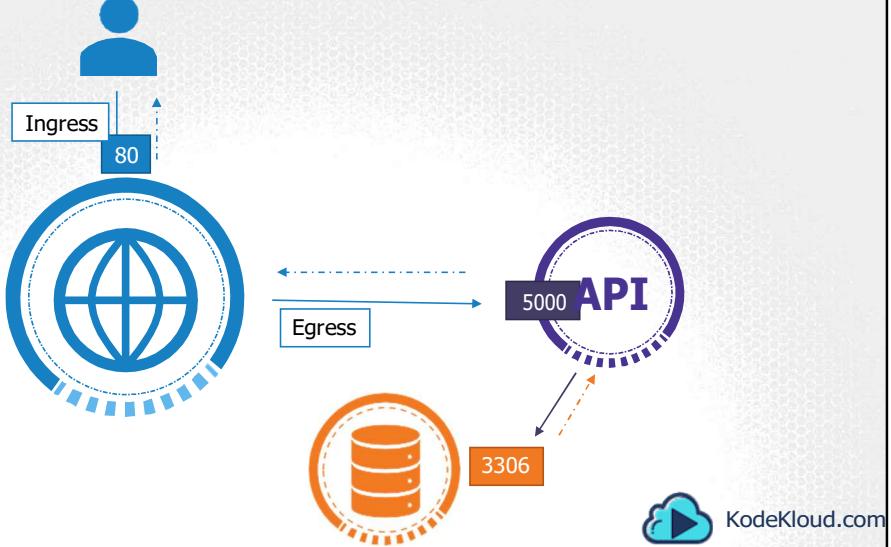
## ITraffic



KodeKloud.com

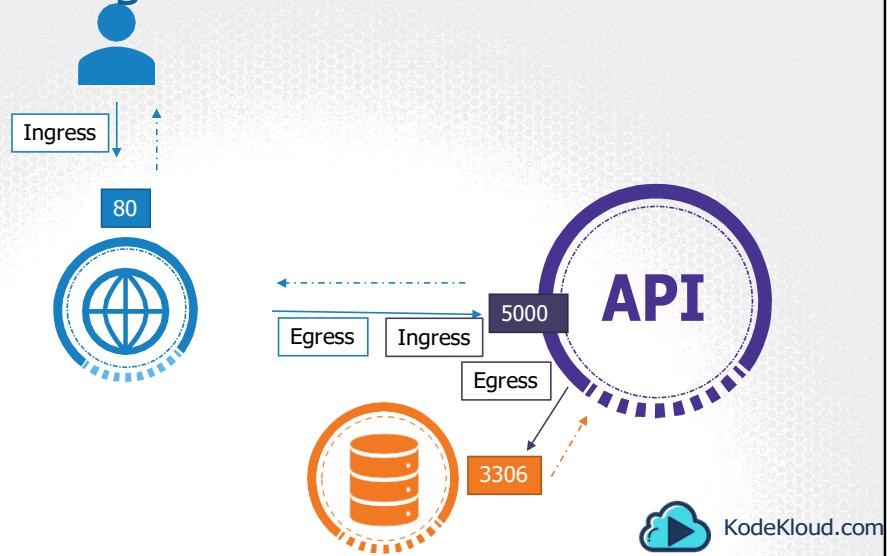
So let us first get our networking and security basics right. And I am sorry if this is too basic, but I just wanted to spend a minute on this to make sure we are all on the same page before we begin. We will start with a simple example of a traffic through a web, app and database server. So you have 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.

## I Ingress & Egress



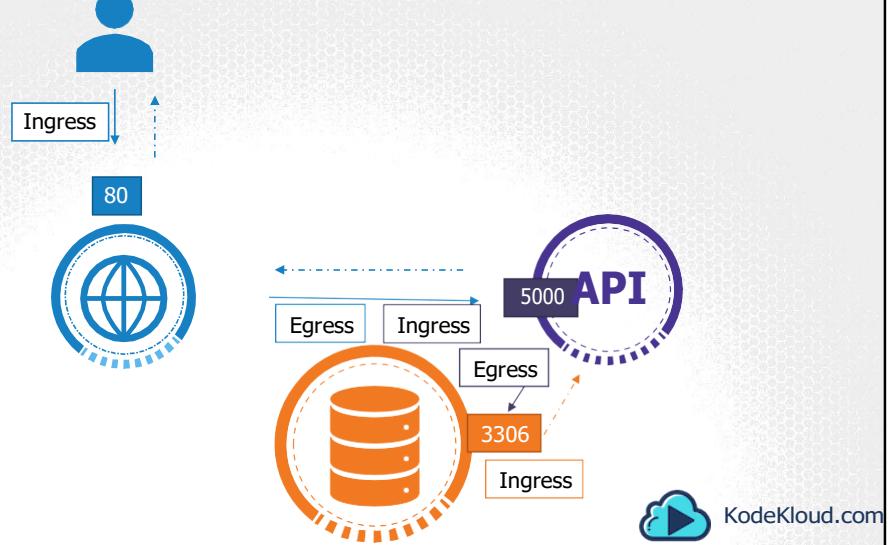
So there are two types of traffic here. Ingress and Egress. For example, for a web server, the incoming traffic from the users is an Ingress Traffic. And the outgoing requests to the app server is Egress traffic. And that is denoted by the straight arrow. When you define ingress and egress, remember 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



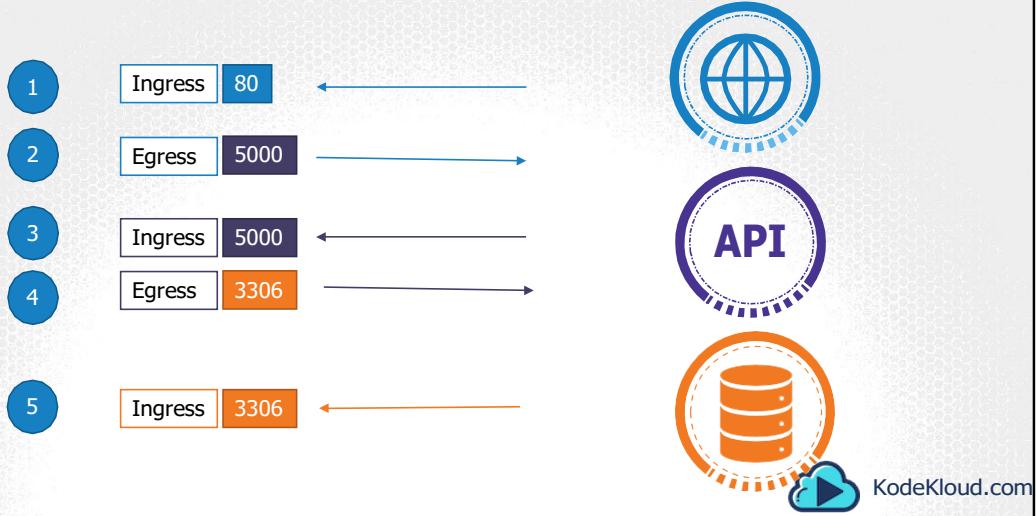
Similarly, in case of the backend API server, it receives ingress traffic from the web server on port 80 and has egress traffic to port 3306 to the database server.

## I Ingress & Egress



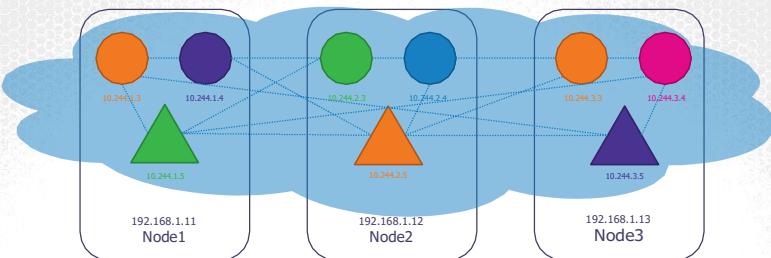
And from the database servers perspective, it receives Ingress traffic on port 3306 from the API server.

## ITraffic



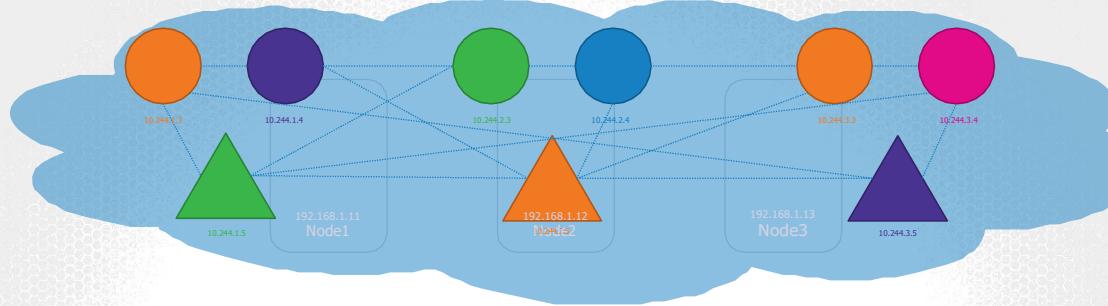
If we were to list the rules required to get this working, we would have an Ingress rule that is required to accept HTTP traffic on port 80 on the web server. An Egress rule to allow traffic from the web server to port 5000 on the API server. An ingress rule to accept traffic on port 5000 on the API 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



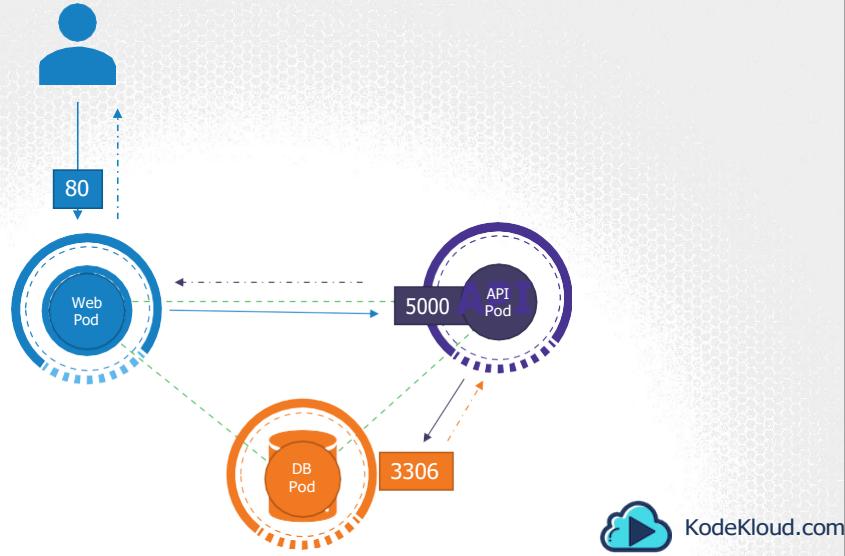
Let us now look at Network Security in Kubernetes. So we have a cluster with a set of nodes hosting a set of pods and services. Each node has an IP address and so does each pod as well as service. One of the pre-requisite for networking in kubernetes, is whatever solution you implement, the pods should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

## Network Security



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.

## ITraffic



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



80

Web Pod

5000

API Pod

3306

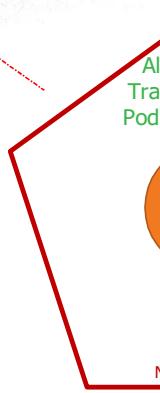
DB Pod



KodeKloud.com

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

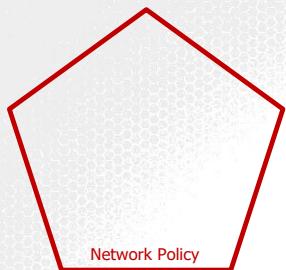


KodeKloud.com

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

Allow Ingress  
Traffic From API  
Pod on Port 3306



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



```
labels:  
  role: db
```



KodeKloud.com

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
```

Allow  
Ingress  
Traffic  
From  
API Pod  
on  
Port 3306



KodeKloud.com

And then we build our rule. Under policyTypes specify whether 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.

## I 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

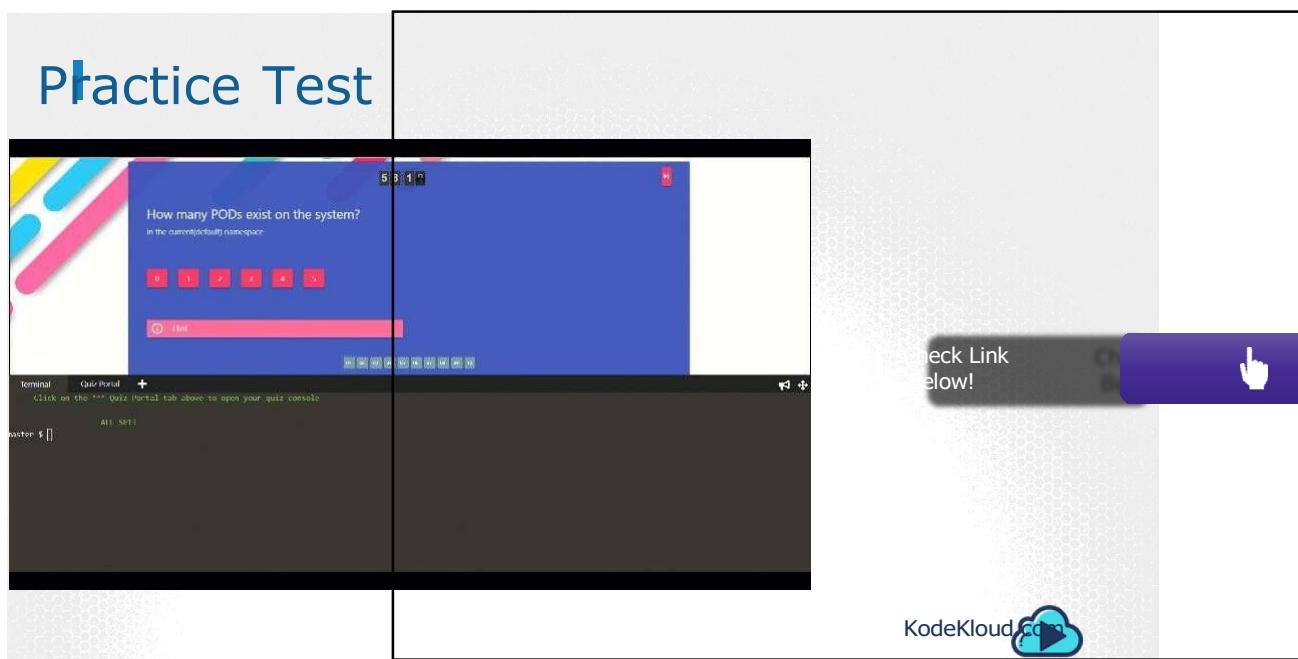
### Solutions that DO NOT Support Network Policies:

- Flannel



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.



KodeKloud.com

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



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



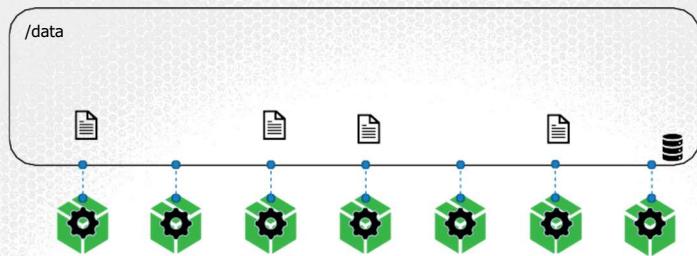
KodeKloud.com

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.

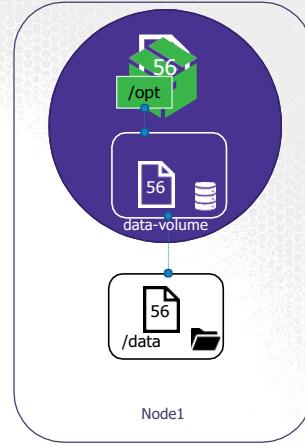
## I Volumes



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 deleted, the data remains.

# Volumes & Mounts

```
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
```



KodeKloud.com

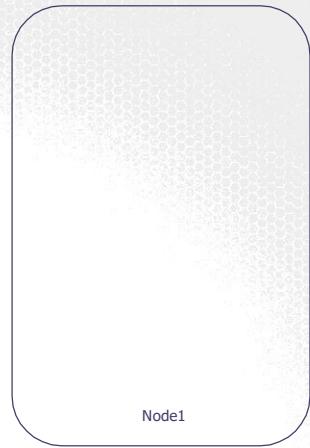
Let's look at a simple implementation of volumes. We have a single node kubernetes cluster. We create a simple POD that generates a random between 1 and 100 and writes that to a file at /data/number.out and then gets deleted along with the random number. To retain the number generated by the pod, we 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
```



data-volume



KodeKloud.com

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
```



data-volume



Node1



Node2



Node3



KodeKloud.com

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: Director
```



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
```



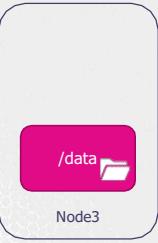
data-volume



Node1



Node2



Node3



KodeKloud.com

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.



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.

I



353

# Persistent Volumes



KodeKloud.com

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

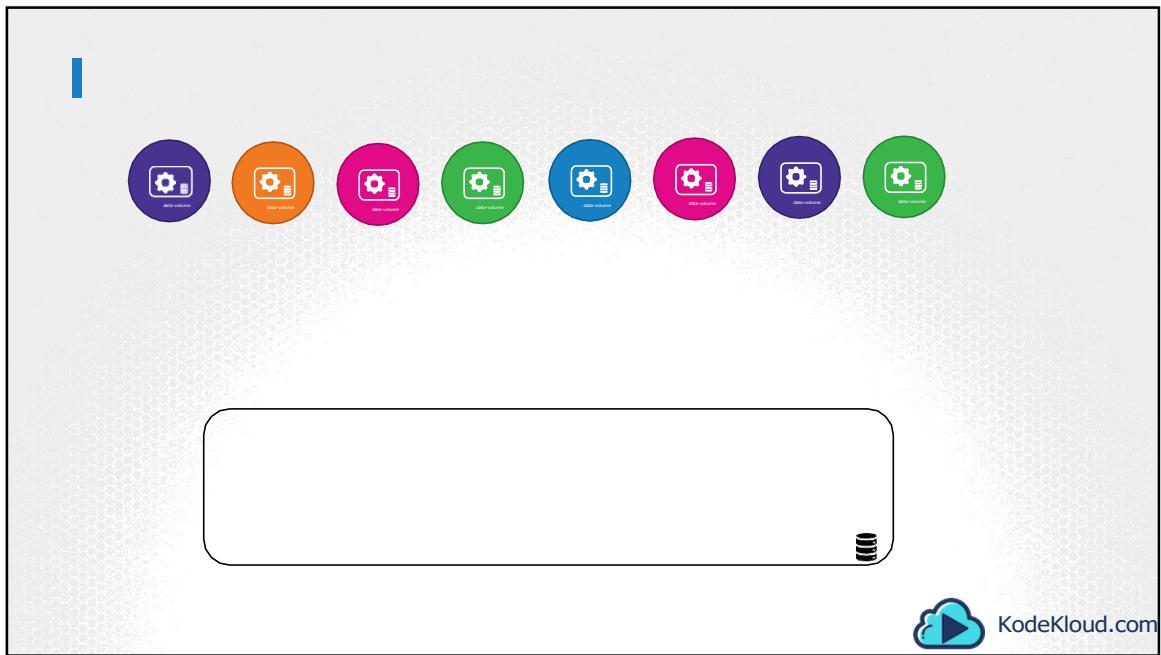
```
volumes:  
- name: data-volume  
  awsElasticBlockStore:  
    volumeID: <volume-id>  
    fsType: ext4
```



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.

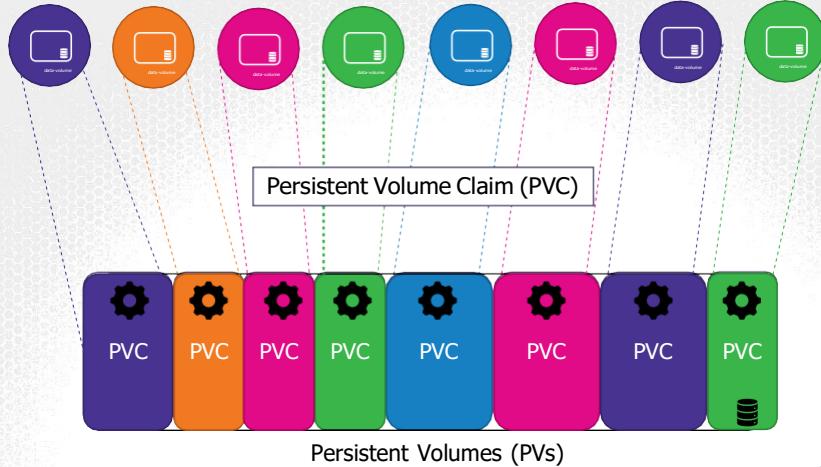


Now, when you have a large environment with a lot of users deploying a lot of PODs, the users would have to configure storage every time for each POD. Whatever storage solution is used, the user who deploys the PODs would have to configure that on all POD definition files in his environment. Every time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

# I Persistent Volume



KodeKloud.com

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

Persistent Volume (PV)

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.



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.



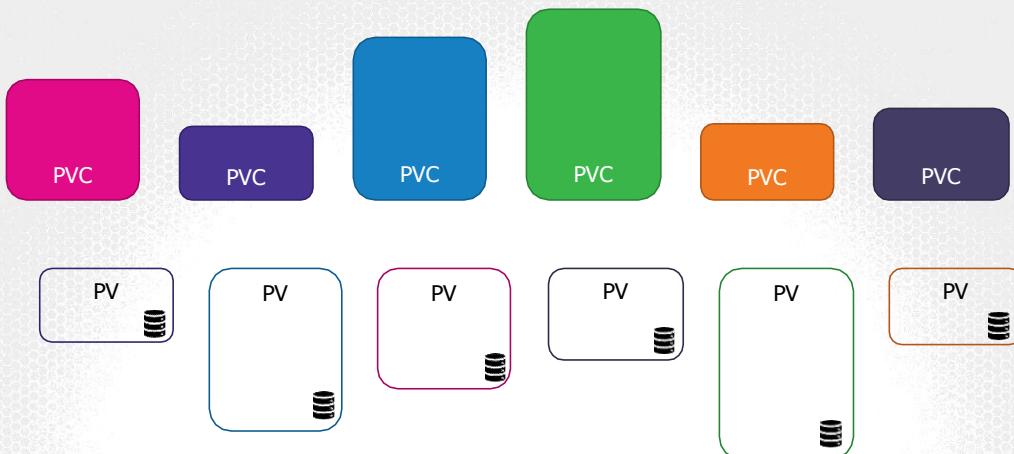
# Persistent Volume Claims



KodeKloud.com

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

## Persistent Volume Claim

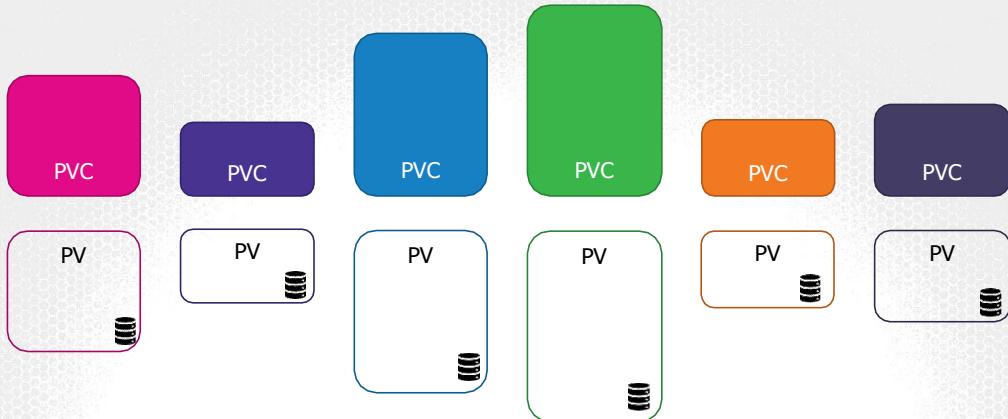


KodeKloud.com

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.

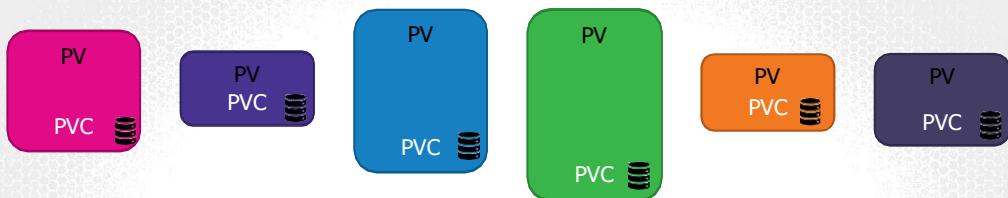
## | Binding



KodeKloud.com

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



Sufficient Capacity

Access Modes

Volume Modes

Storage Class



KodeKloud.com

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.

# I Binding

PVC

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

PV  


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

PV  


Sufficient Capacity

Access Modes

Volume Modes

Storage Class

Selector



KodeKloud.com

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



Pending

Sufficient Capacity

Access Modes

Volume Modes

Storage Class

Selector



KodeKloud.com

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

A terminal session illustrating the creation and status of a Persistent Volume Claim (PVC). The session starts with creating a YAML definition for the PVC, followed by running the kubectl get command to check its status.

```
pvc-definition.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 500Mi

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

kubectl create -f pvc-definition.yaml

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-voll
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.

## I View PVCs

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



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

## I Delete PVCs

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

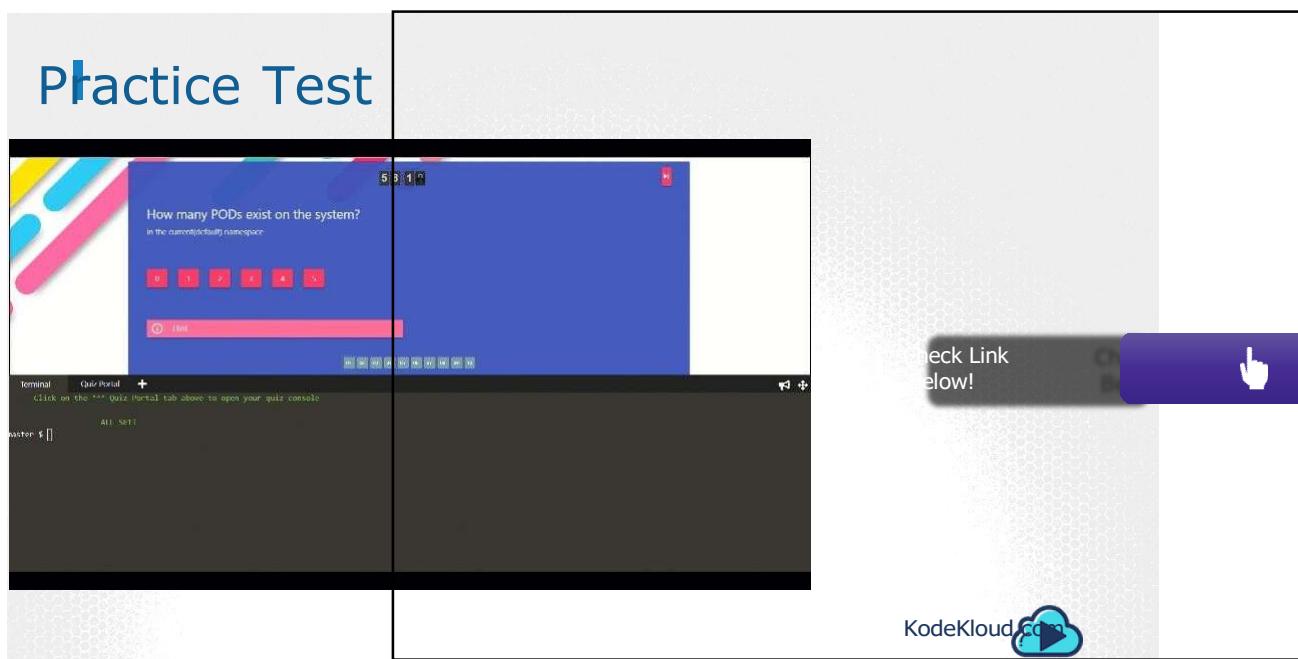


```
persistentVolumeReclaimPolicy: Recycle
```



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.



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

## Challenges



Check Link  
below!

KodeKloud.com

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



Hello there!.



# TIME MANAGEMENT



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.

I

Certified Kubernetes  
Administrator  
(CKA)



24

Certified Kubernetes  
Application Developer  
(CKAD)

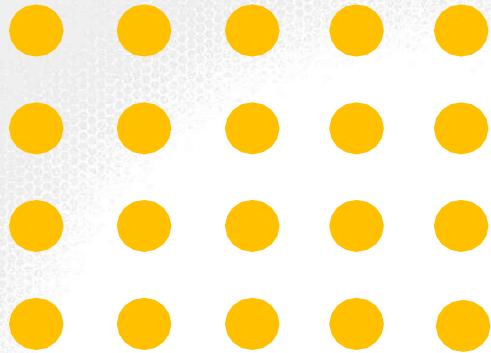


19



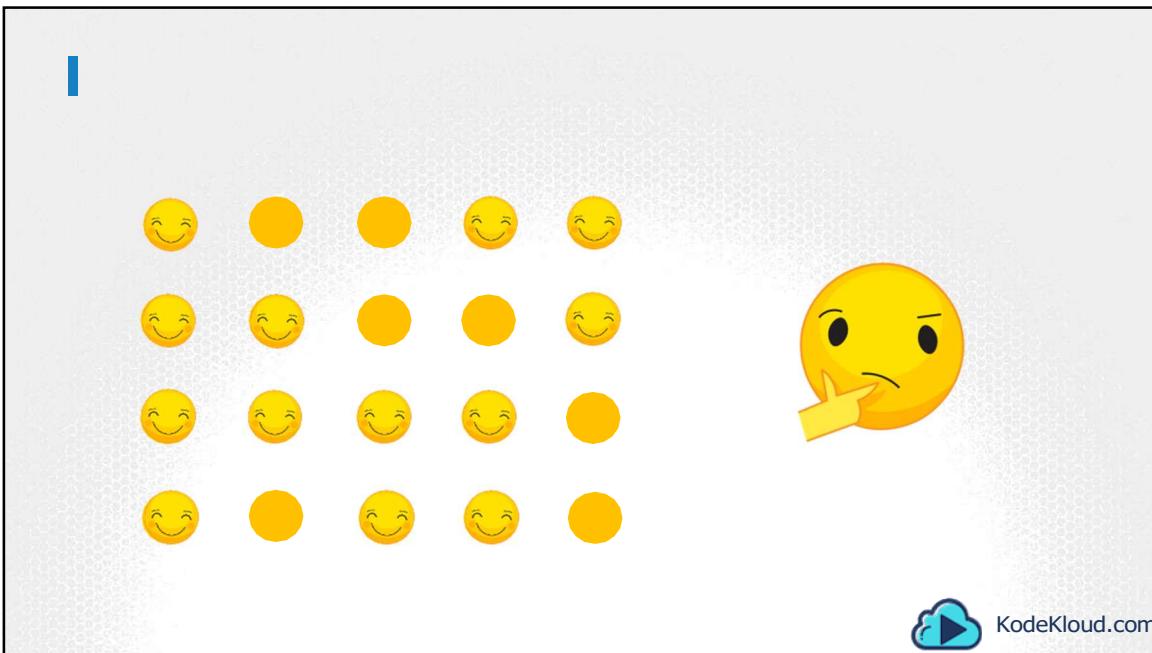
As of today, you get from 2 hours to 2.5 hours to complete the kubernetes certification exams. The duration of the administrators exam is 2.5 hours and the application developers exam is 2 hours. Now that is not sufficient to complete all the questions, so it is important to manage your time effectively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

I

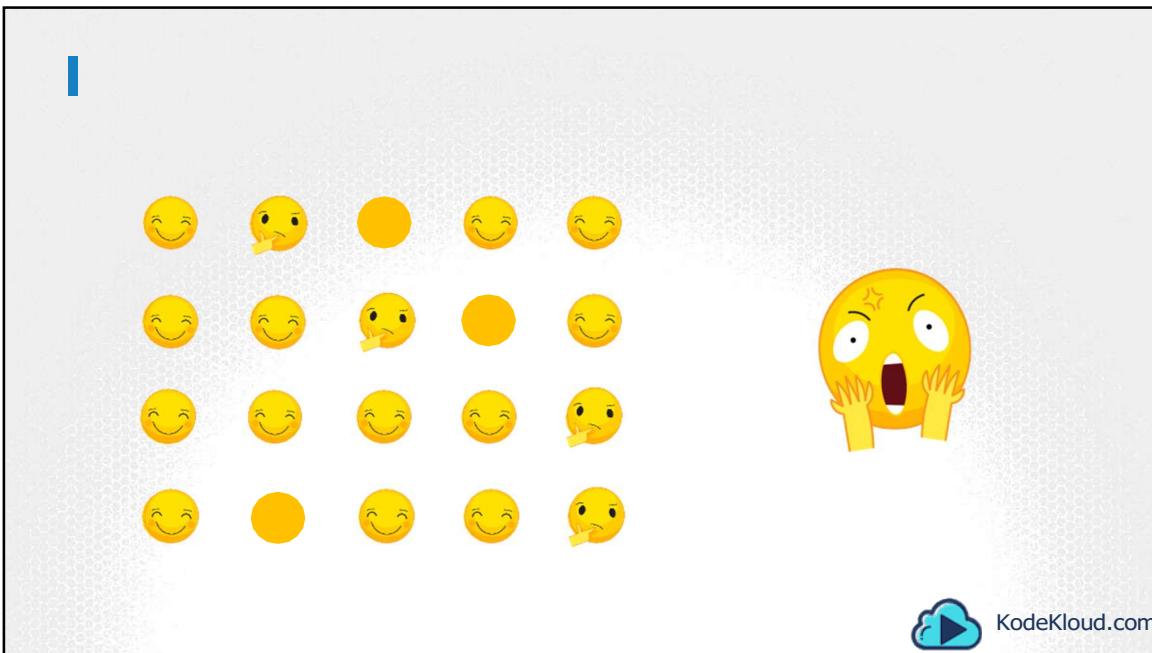


KodeKloud.com

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....



KodeKloud.com

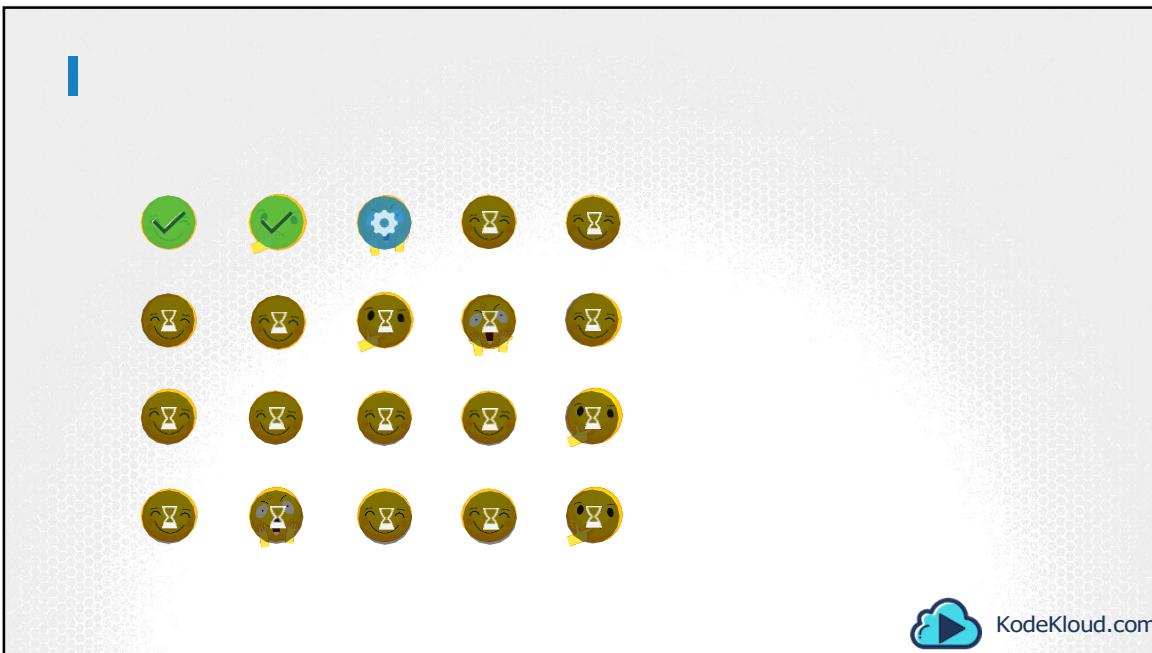
and some that you have no clue about, hopefully not too many of that. And they are not there in any particular order. You may have easy or tough questions in the beginning or towards the end.

I



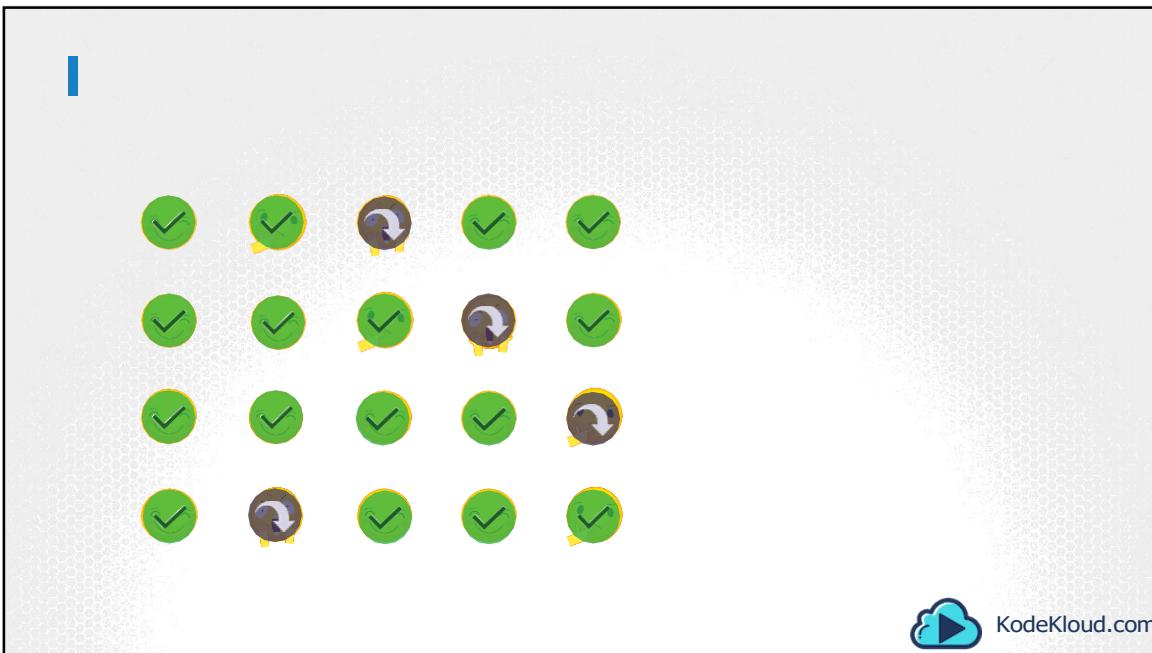
KodeKloud.com

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.



KodeKloud.com

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.



KodeKloud.com

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.

## I 1. Attempt all Questions



KodeKloud.com

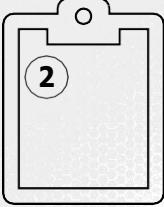
So that was the first and most important tip, attempt all the questions.

## 12. 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**

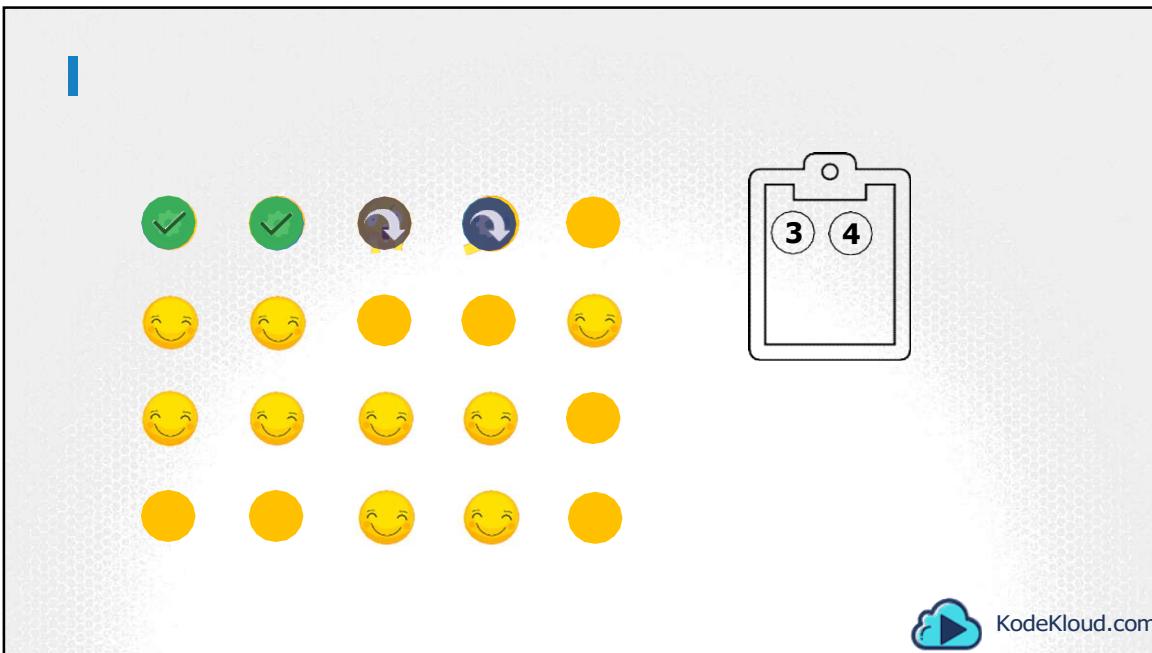
error: unable to recognize "deployment-definition.yaml": no matches for kind "Deployment" in version "apps/v1"

**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.



KodeKloud.com

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.

## I3. Get good with YAML

```
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:
```

```
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:
```

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.

## 14. 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



## ITime Management



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!

m m u m s h a d @ g m a i l . c o m