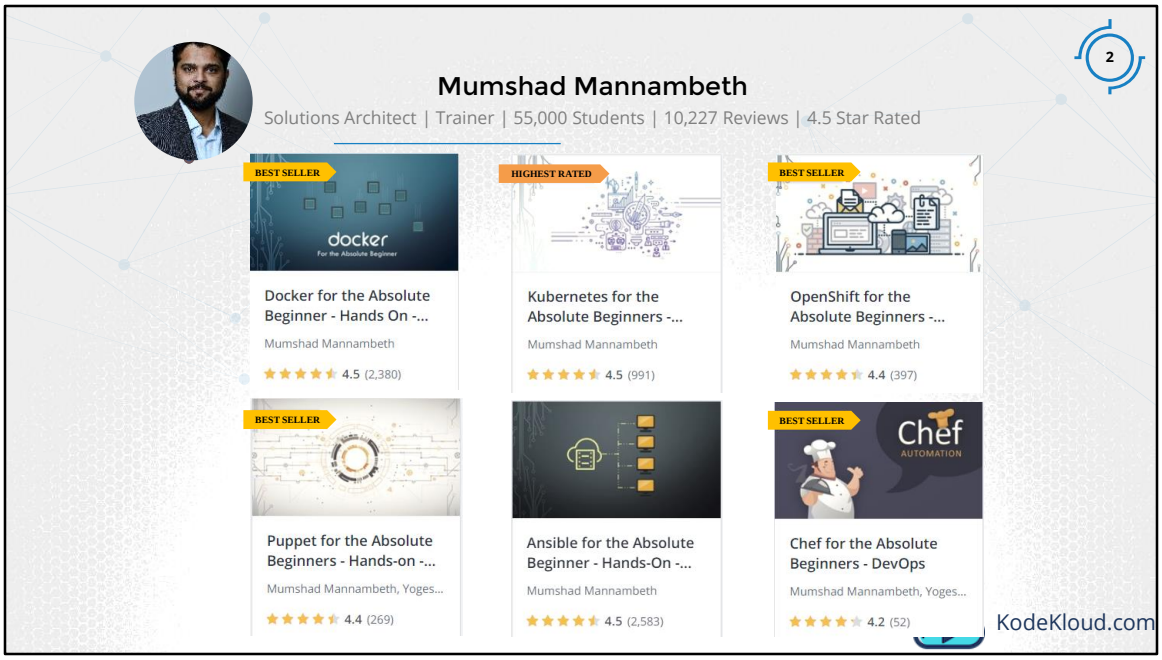
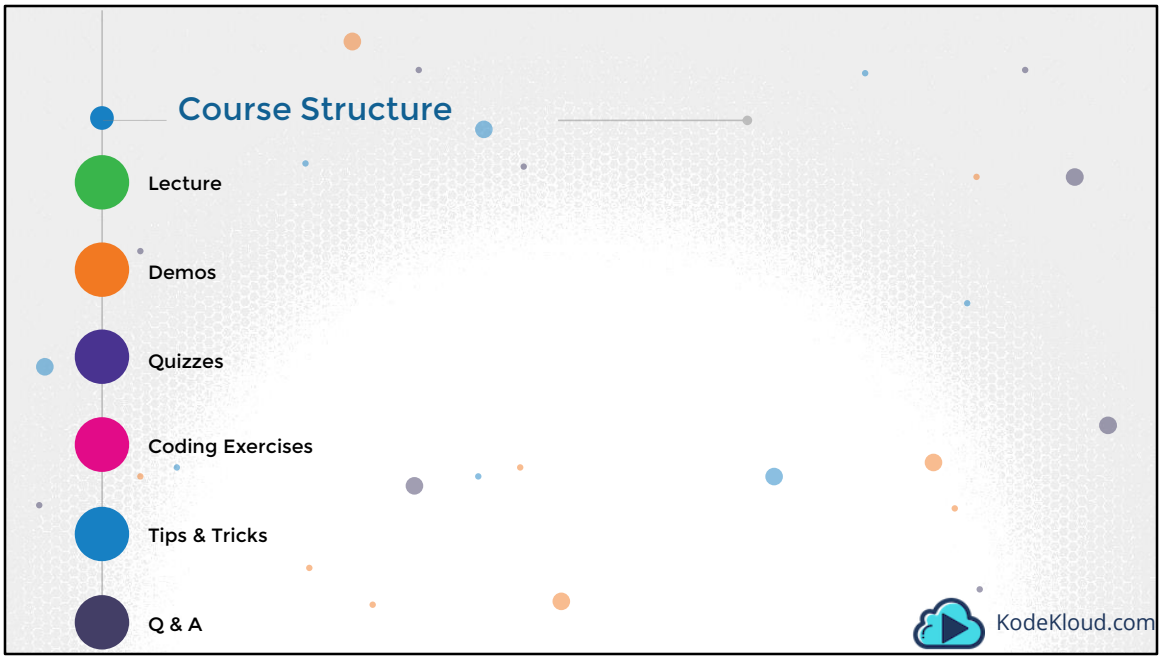

CERTIFIED
KUBERNETES
APPLICATION
DEVELOPER
Start Here
KodeKloud.com



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.

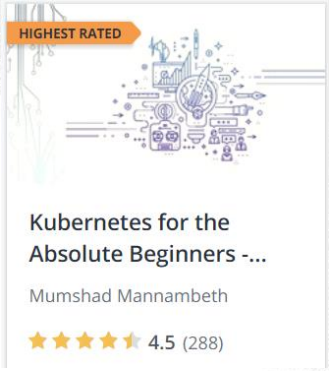
We then have coding quizzes that will help you practice what you learned on a real live environment right in your browser. The kubernetes certification is hands-on, so the coding exercises will give you enough experience and practice on getting ready for it. More on this in the upcoming lectures.

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.

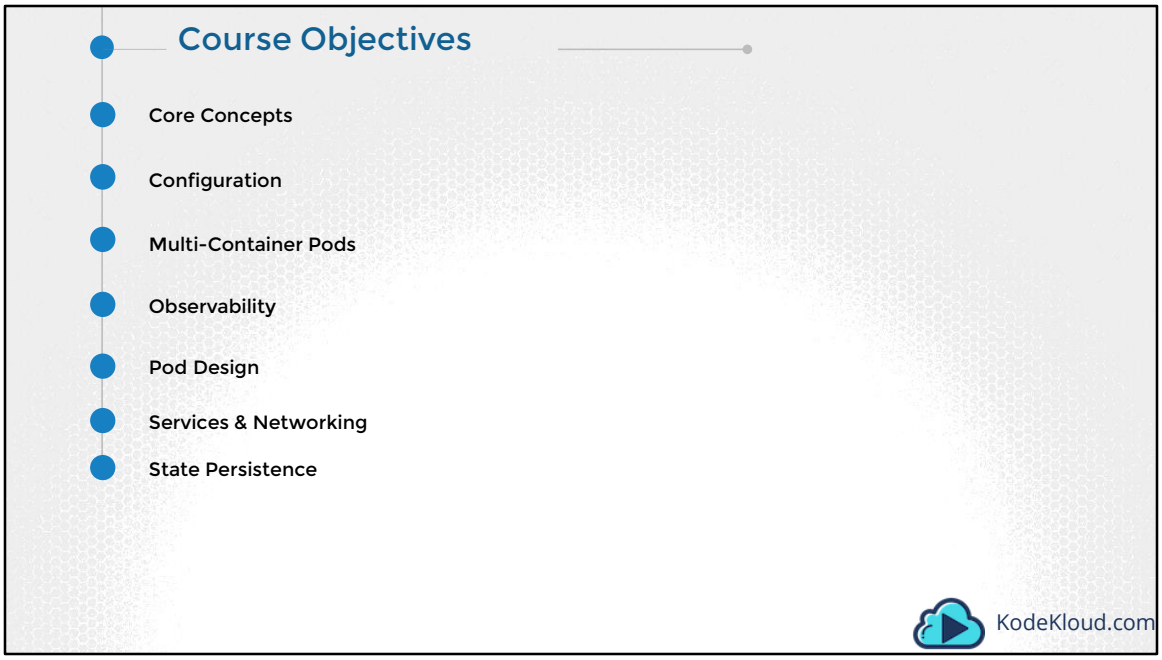
Pre-Requisites

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



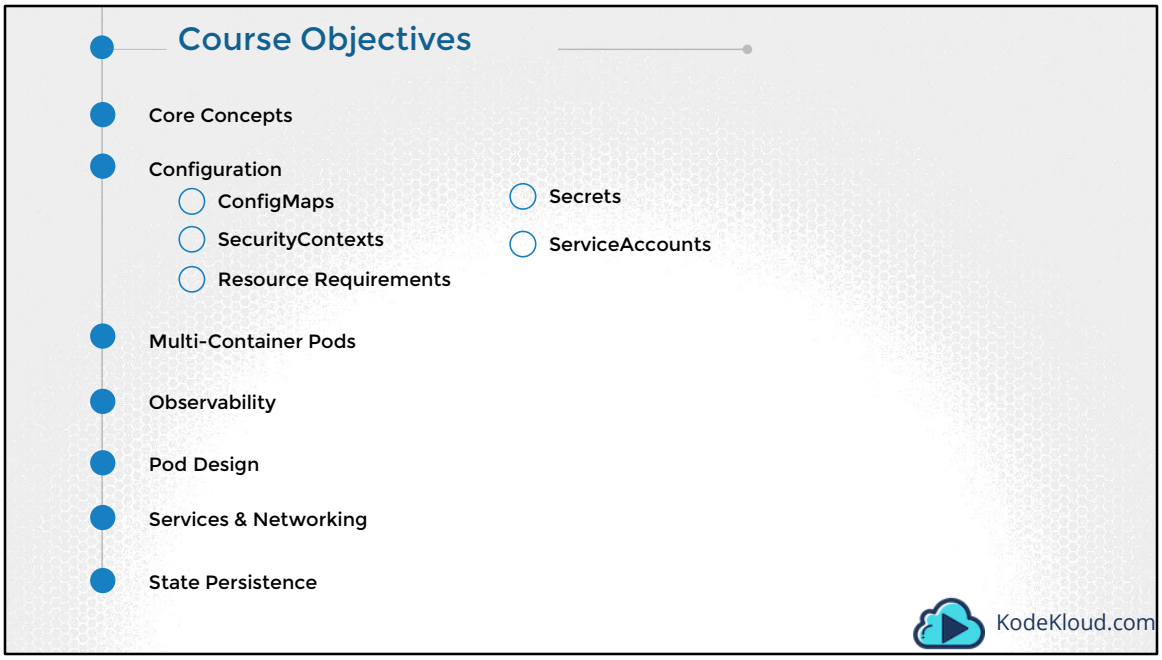
KodeKloud.com

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.

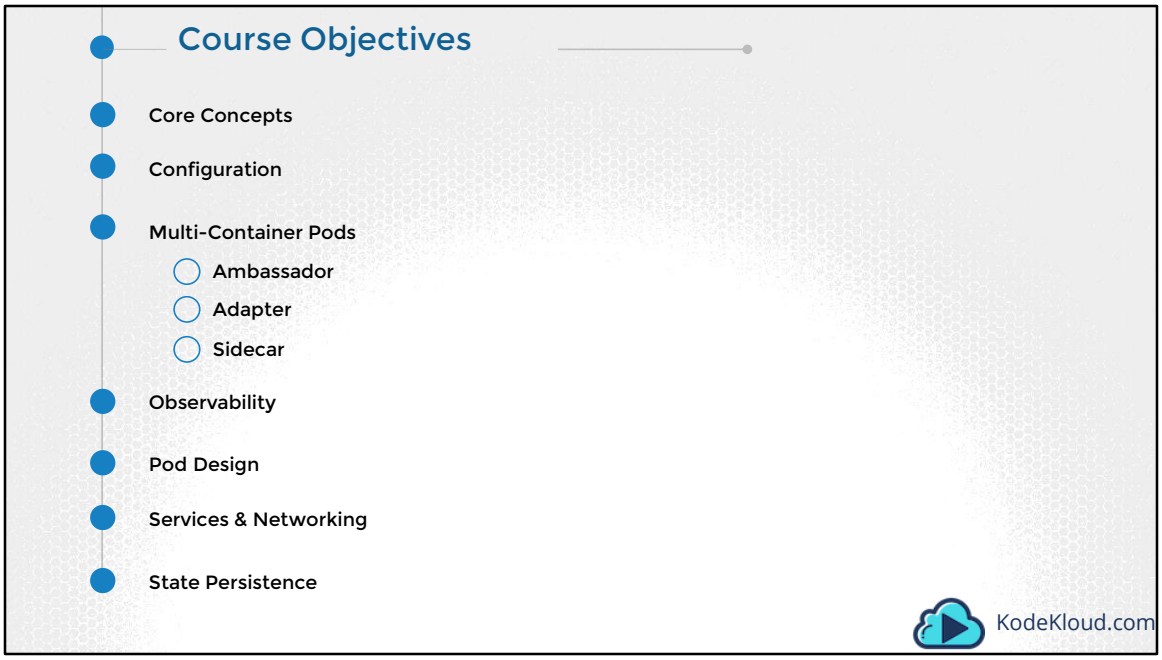


Let us now look at the Course Objectives. The objectives of this course are aligned to match the Certified Kubernetes Application Developer exam Curriculum. We will discuss about details around the Certification itself in one of the upcoming lectures before heading into any of these topics.

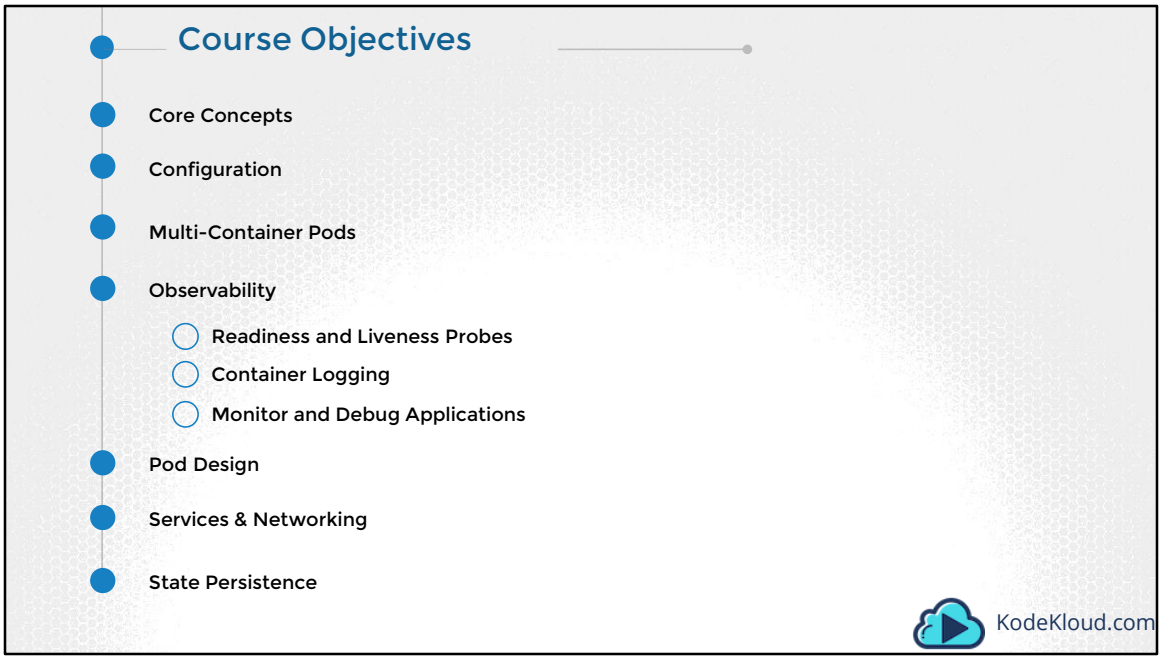
We start with the core concepts – we have covered a lot of the core concepts in the beginners course. We will however quickly recap some of these in this course to refresh our memory. Such as the kubernetes Architecture, what PODs are and how to create and configure PODs etc.



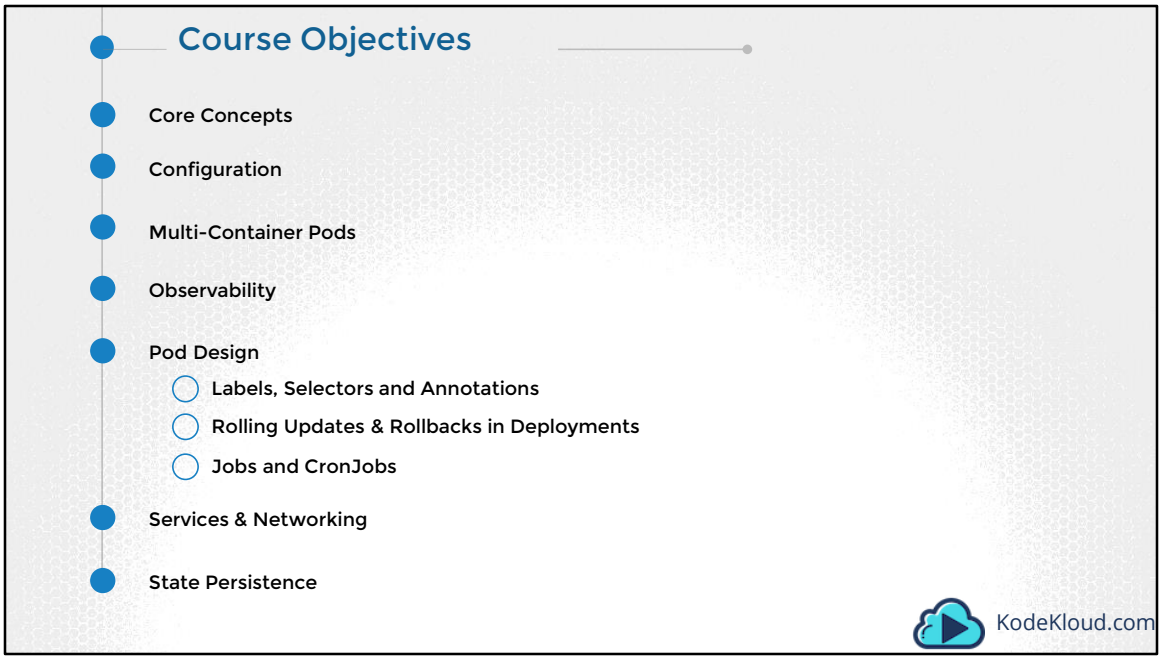
The next section is on Configuration and covers topics like ConfigMaps, SecurityContexts, Resource Requirements, secrets and service accounts.



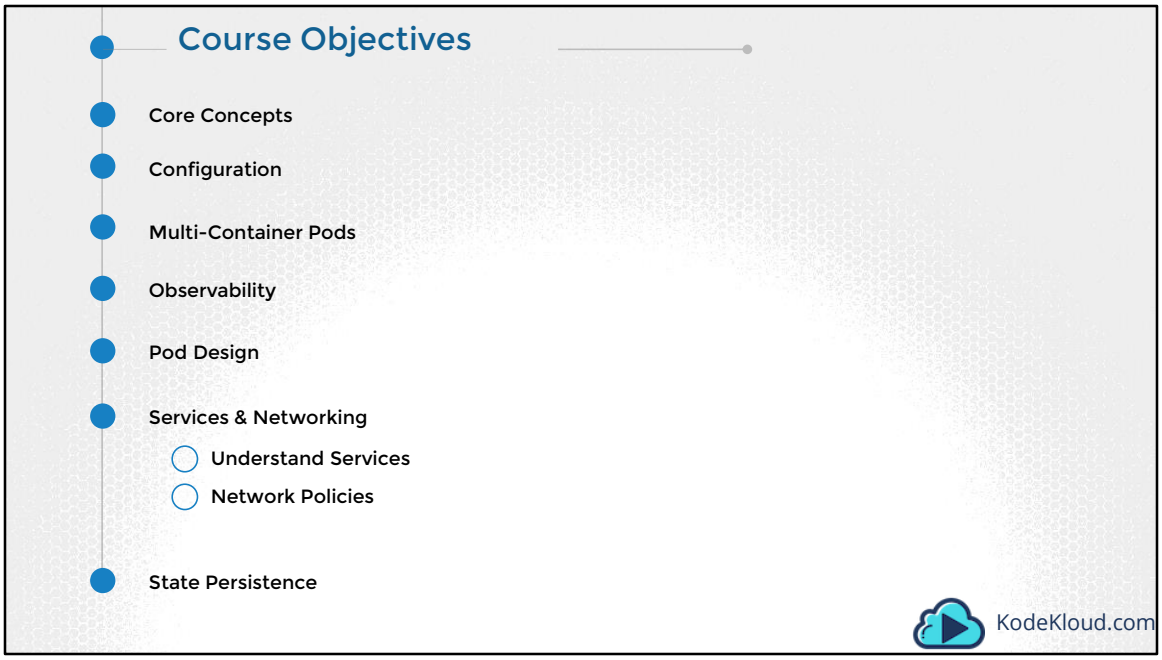
We will then look deeper into Multi-Container Pods. The different patterns of multi-container pods such as Ambassador, Adapter and Sidecar. We will look at some examples and use cases around these.



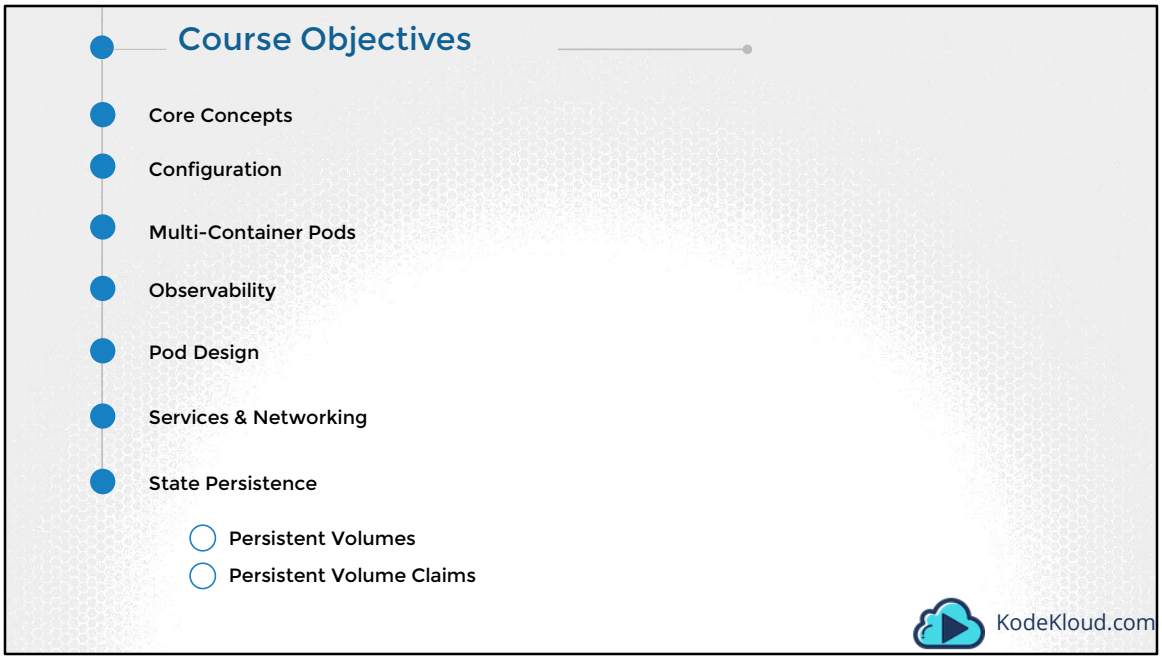
We then learn about Readiness and Liveness Probes and why you need them. We will also look at some of the Monitoring, Logging and Debugging options available with Kubernetes specifically around Pods, containers and applications.



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.

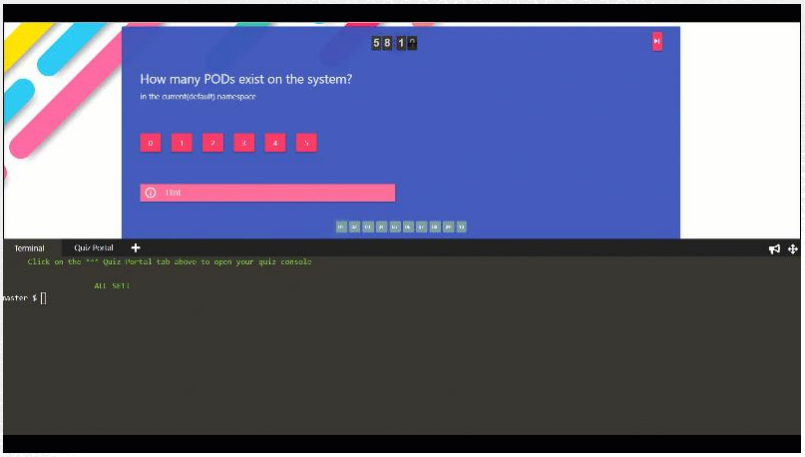


We will then learn about Services and Network Policies.



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

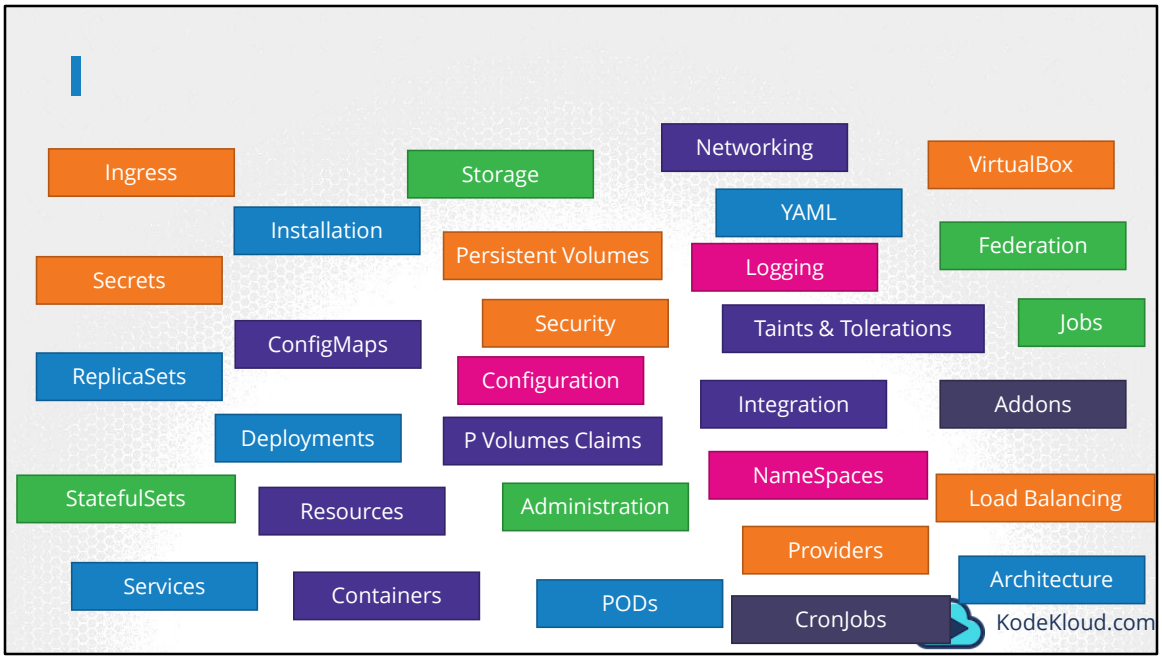
Try Here!

KodeKloud.com

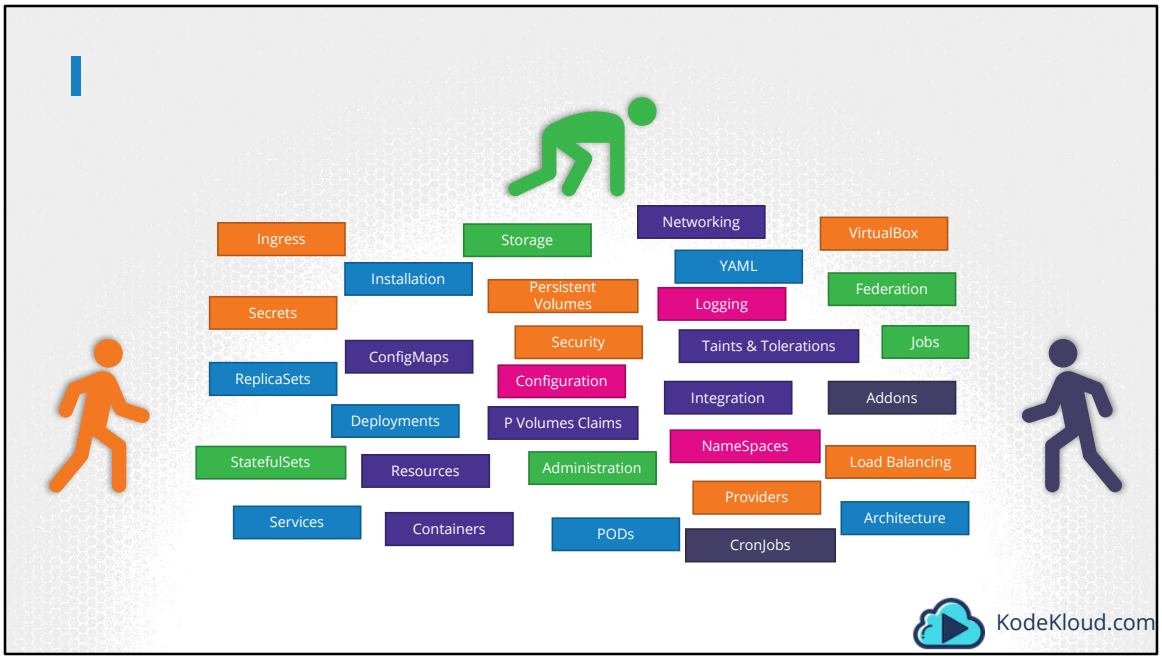
Watch it here: https://kodekloud.com/courses/kubernetes-certification-course/lectures/6731376

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

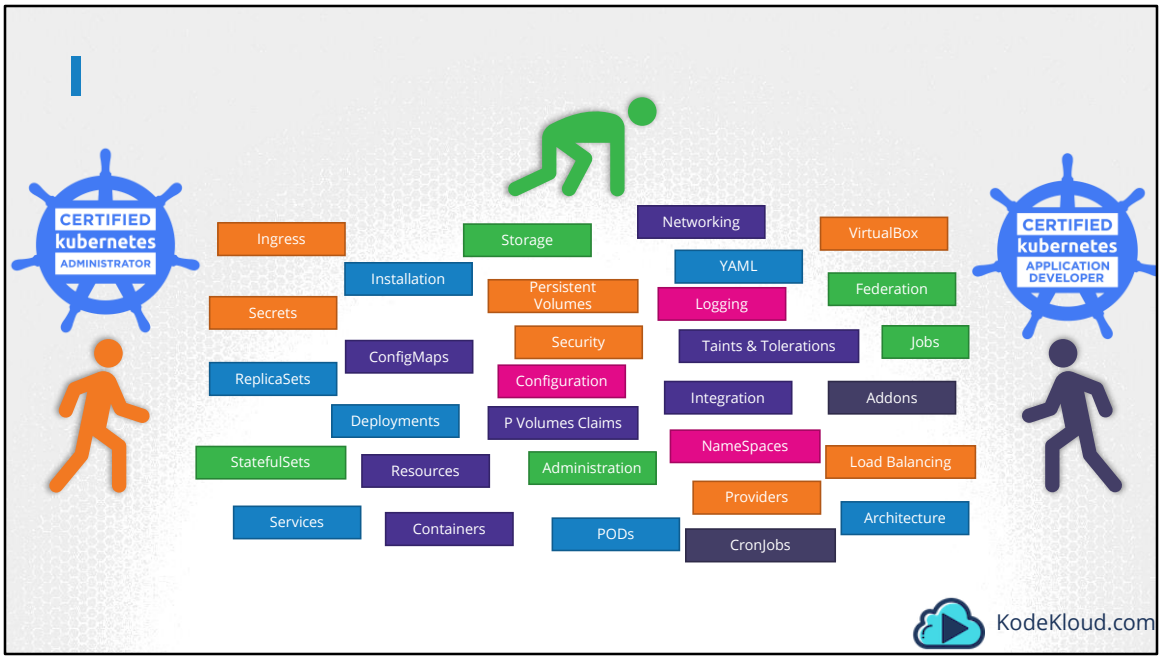

CERTIFIED
KUBERNETES
SERIES
KodeKloud.com



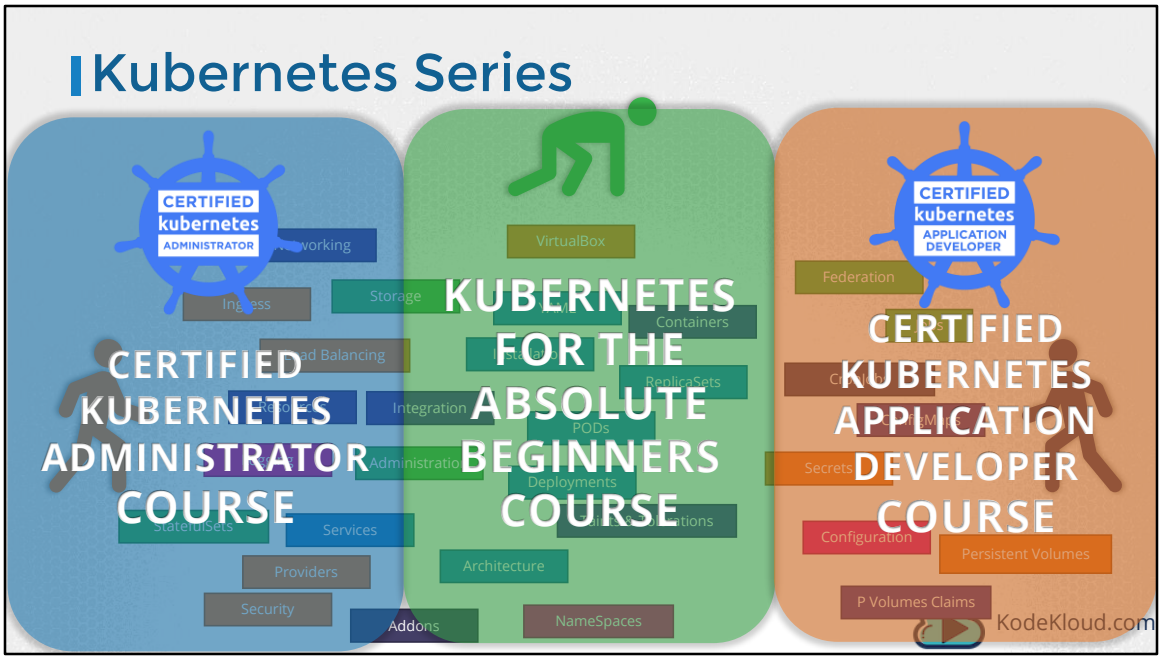
Hello there. Before we begin, I want to spend a minute on the Kubernetes Series of courses. Kubernetes is one of the most trending technology in cloud computing as of today. It is supported on any cloud platform and supports hosting enhanced and complex applications on various kinds of architectures 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.



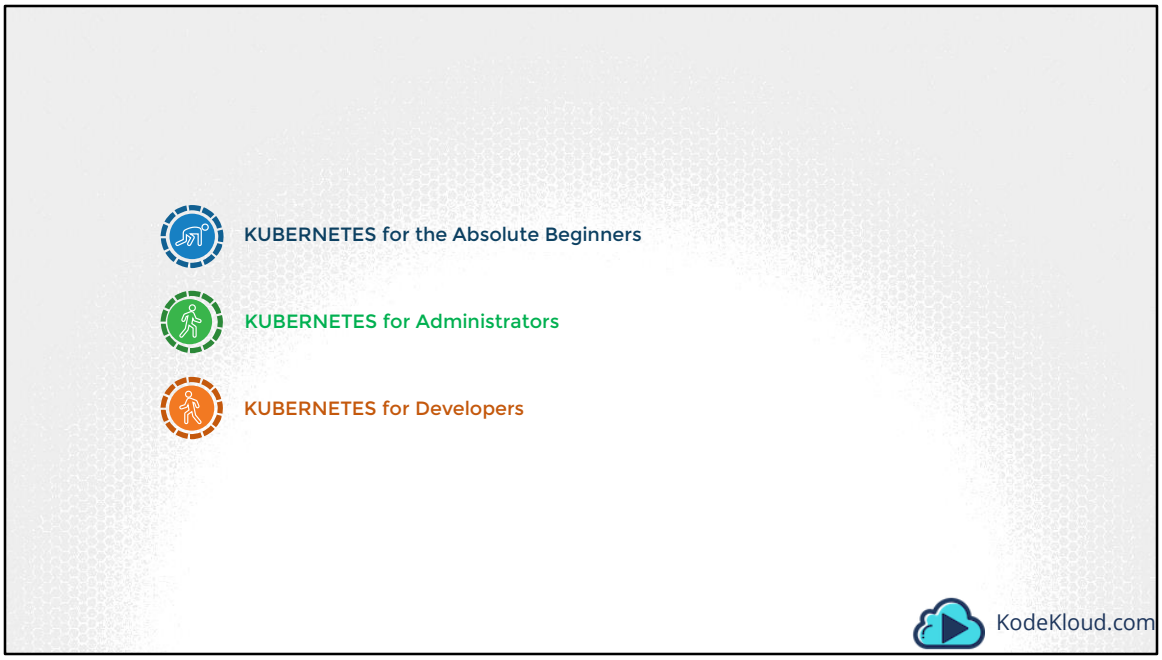
There are students from different backgrounds, such as the Absolute Beginners to Kubernetes, or those with some experience looking for specialized knowledge in Administration or those from a development background looking for concepts specific to Application Development on Kubernetes.



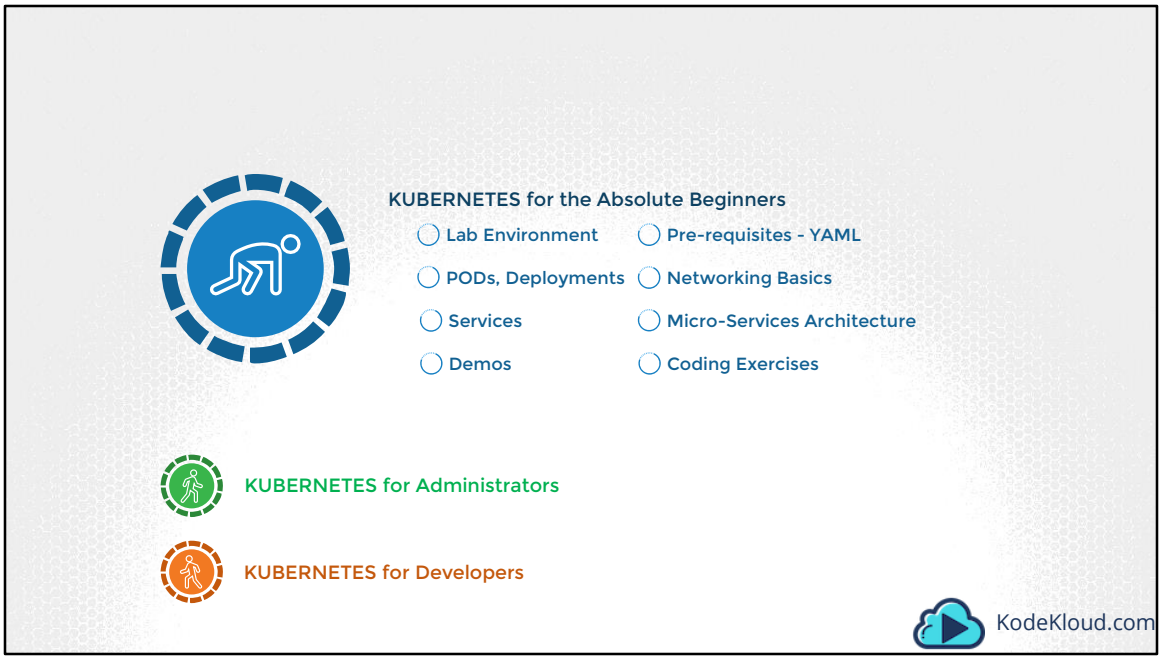
And there are two Certifications in the mix as well. One for Administrators and another for Application Developers. Covering all of these topics for all of these students in a single course is an impossible task.



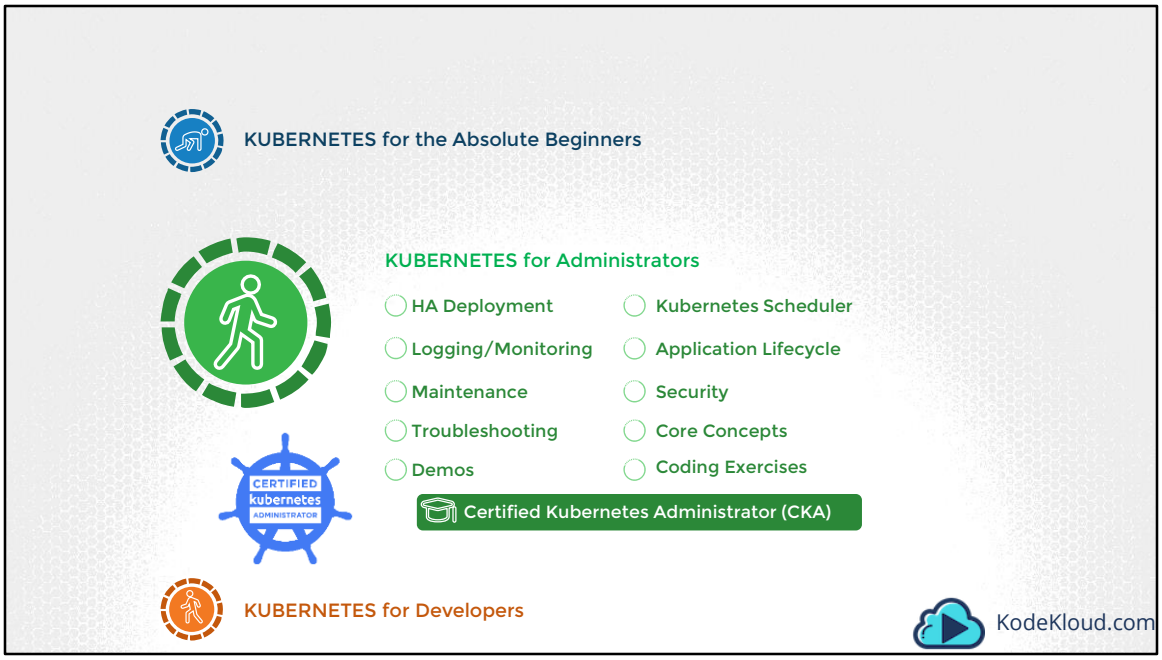
Which is why we created a 3 course series, so each course can target specific audience, topics and certifications. The Kubernetes for the Absolute Beginners course, the Certified Kubernetes Administrator's course and the certified kubernetes application developer's course.



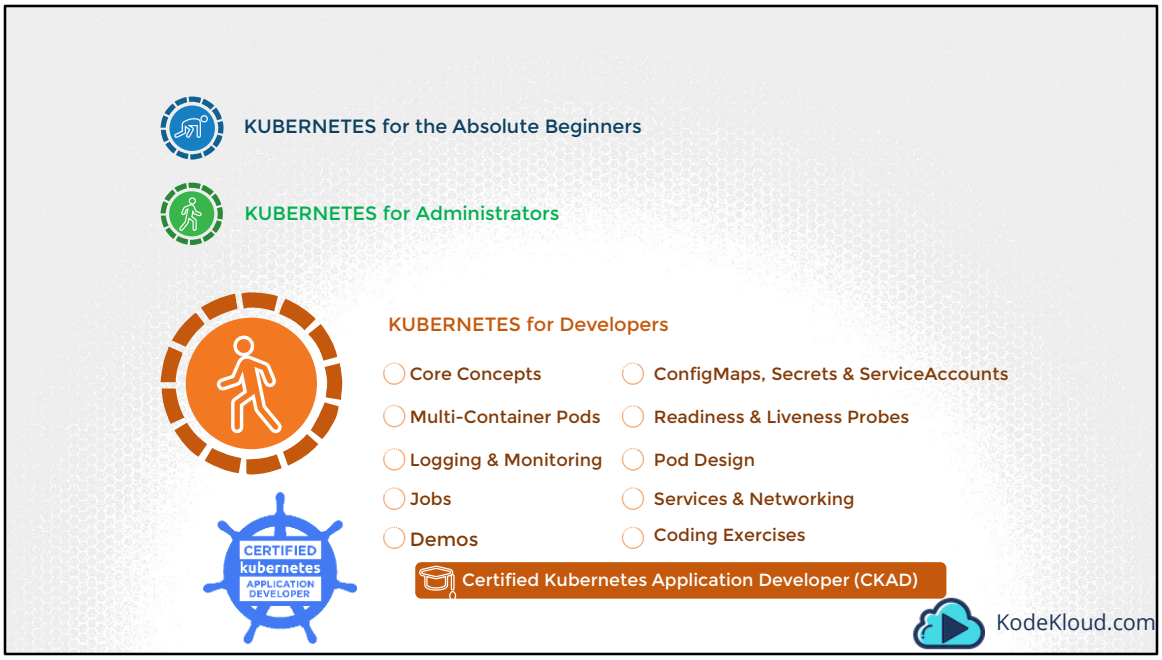
Let’s look at what we cover in each of these courses.



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.



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.

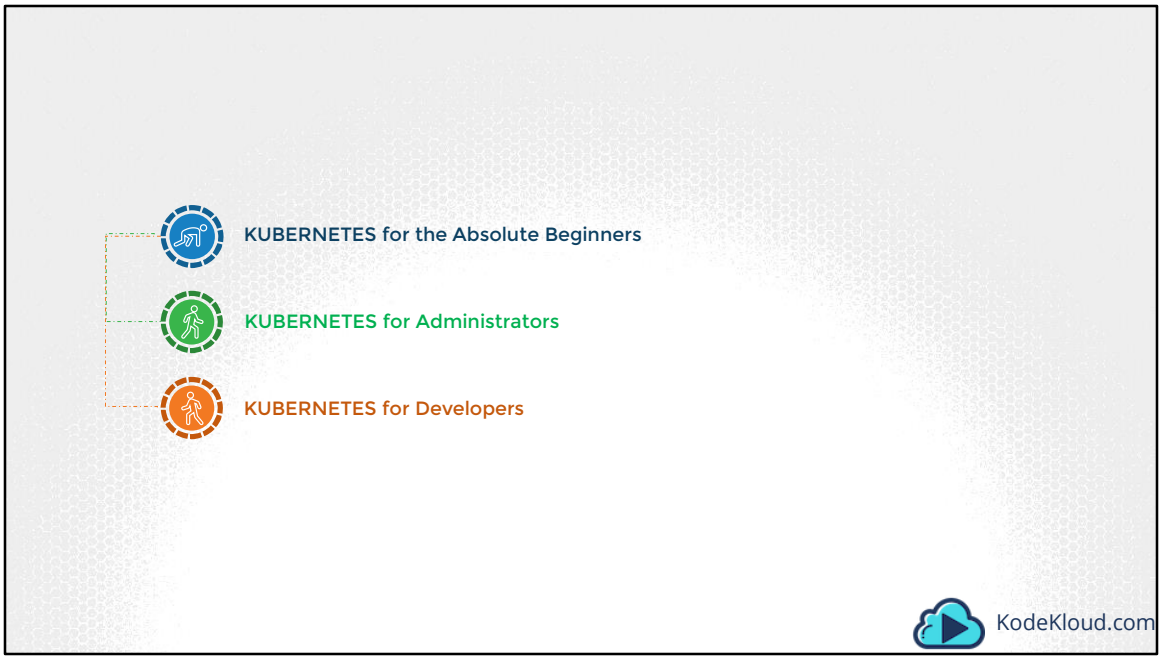


The Kubernetes for Developers course is for Application Developers who are looking to learn how to design, build and configure cloud native applications. Now you don't have to be an expert application developer for this course and there is no real coding or application development involved in either 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.

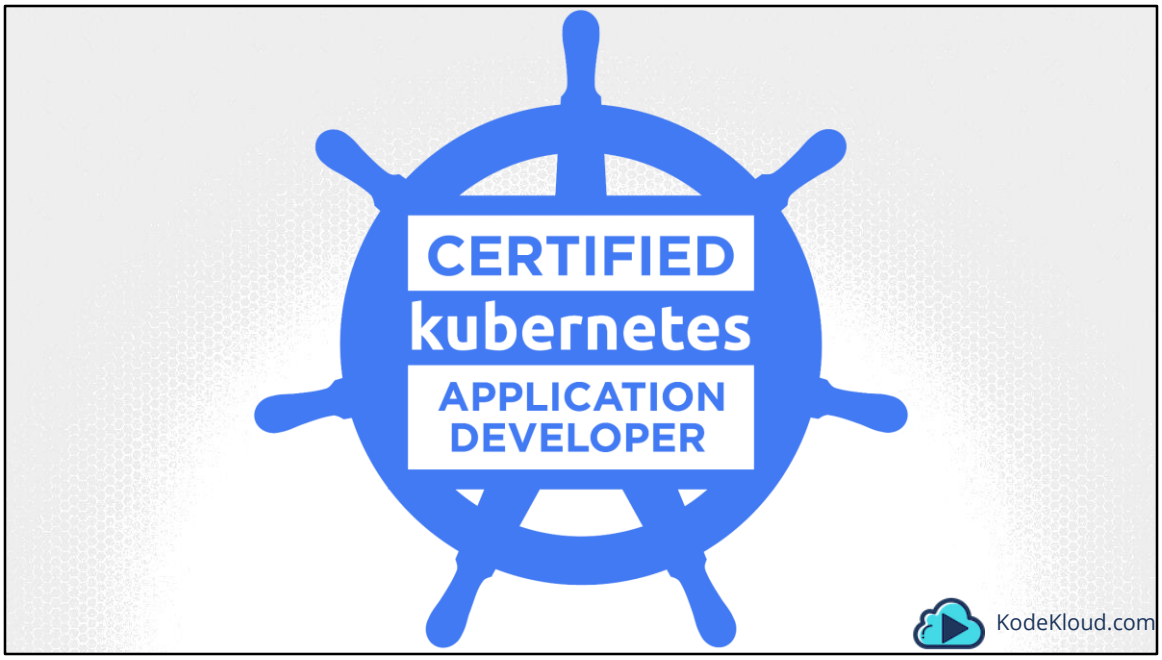
All of these courses are filled with Coding Exercises and quizzes that will help you PRACTICE developing and deploying applications on Kubernetes.

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

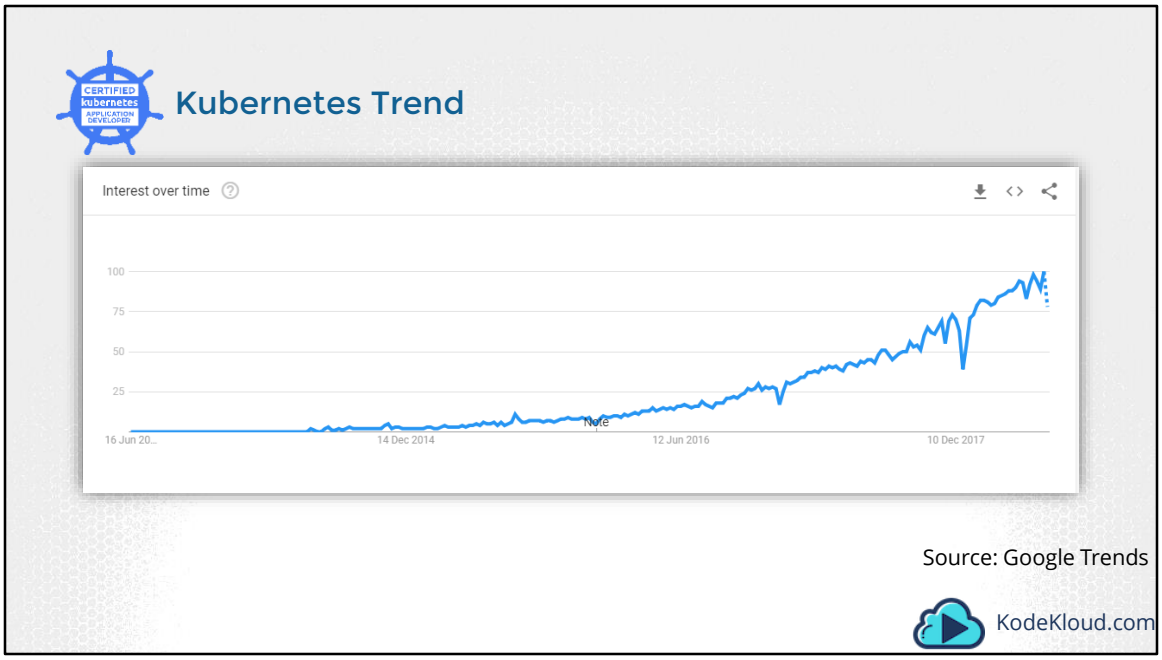


Now you don't have to take these courses in Order. If you are an administrator, you may chose to take the Beginners as well as the second course and get yourself certified as a Kubernetes Administrator. Or take the beginners course and the developers course to get yourself certified as a Kubernetes Application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

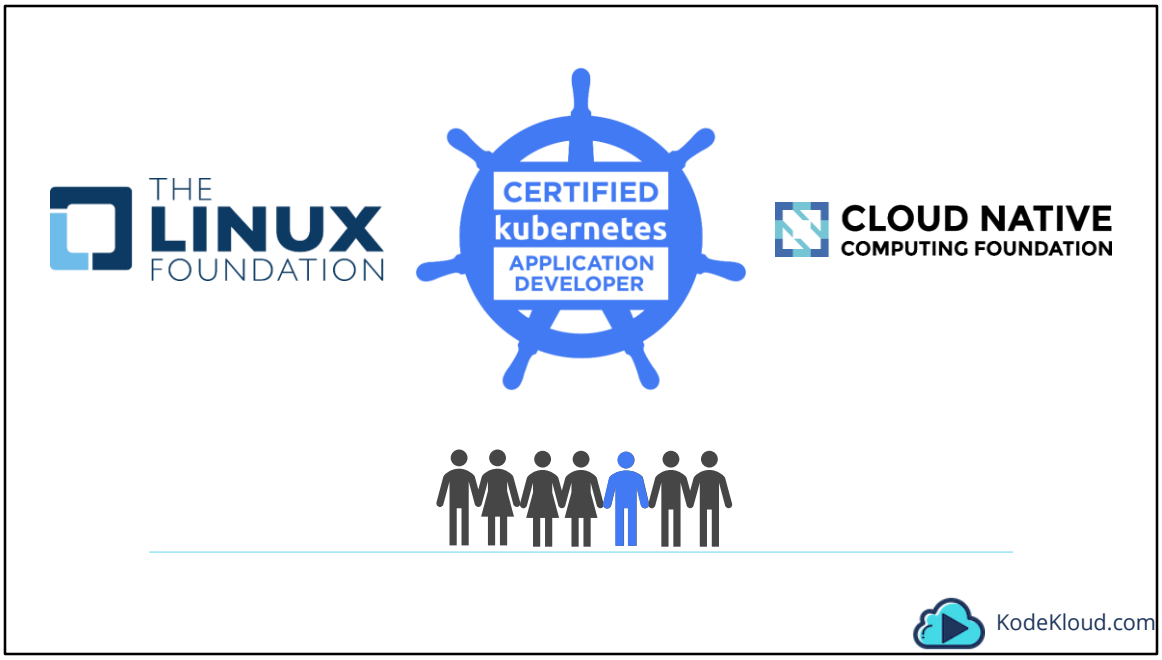
So if you are ready, let's get started.



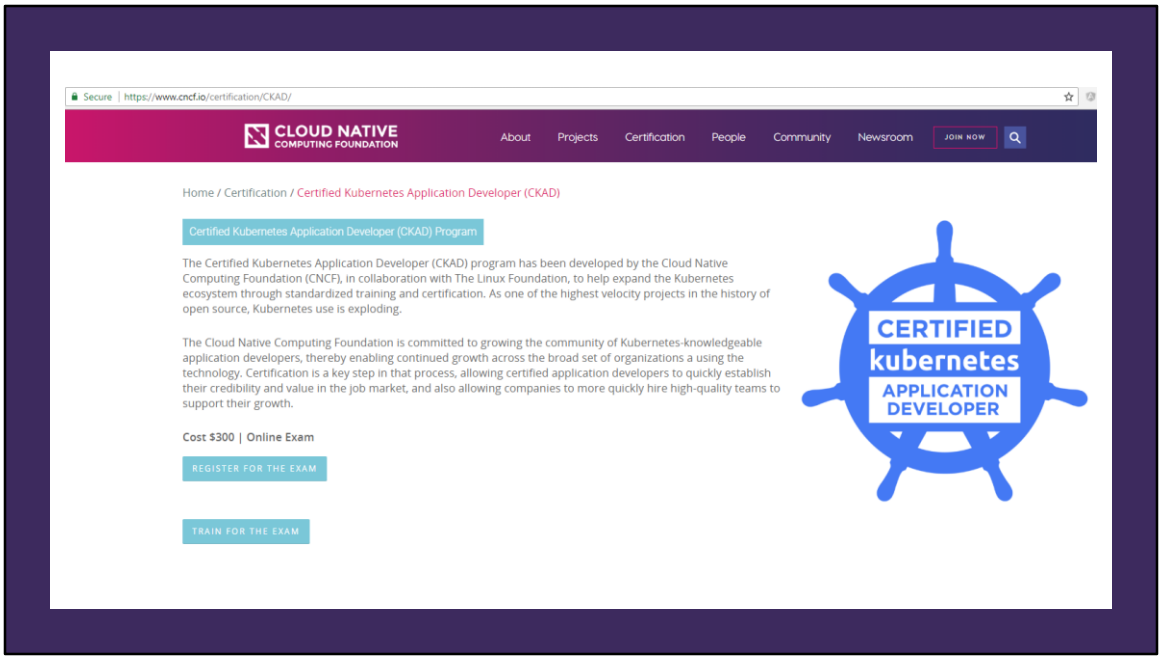
Hello and welcome to this lecture. In this lecture, we will look at some of the details around the Certified Kubernetes Application Developer program. What it is, why you need it and how to get started with it.



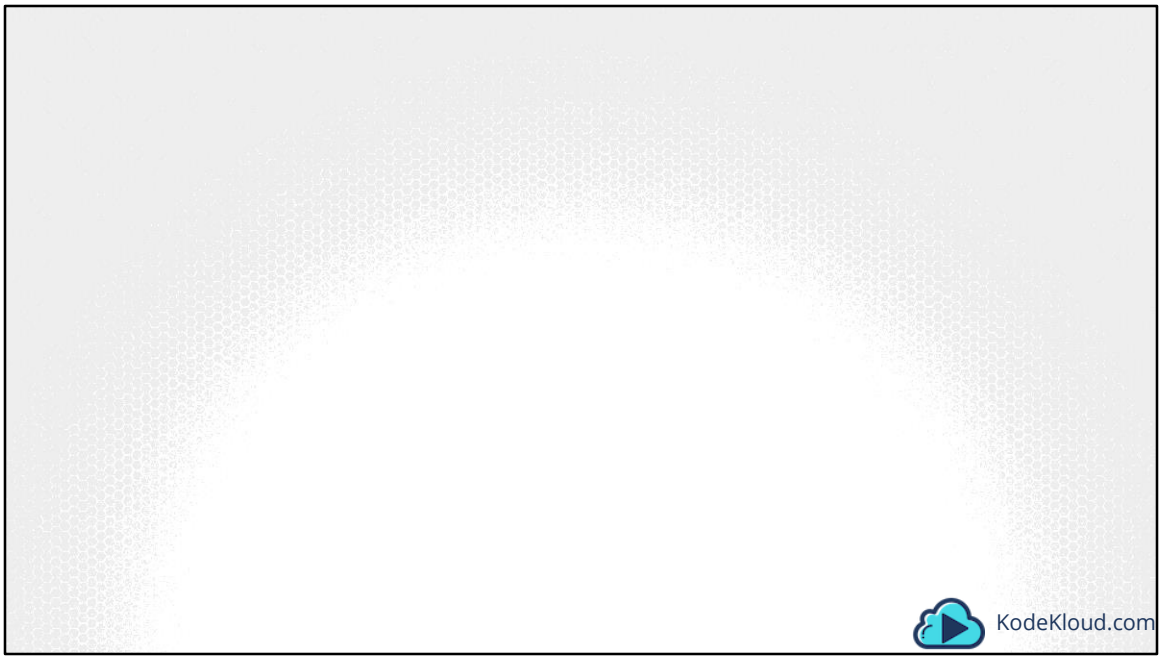
There is no doubt about the fact that the adoption of Kubernetes is expected to grow exponentially in the coming years, as seen in the graph from Google Trends. And so it is important for us to be prepared to establish credibility and value in the market.



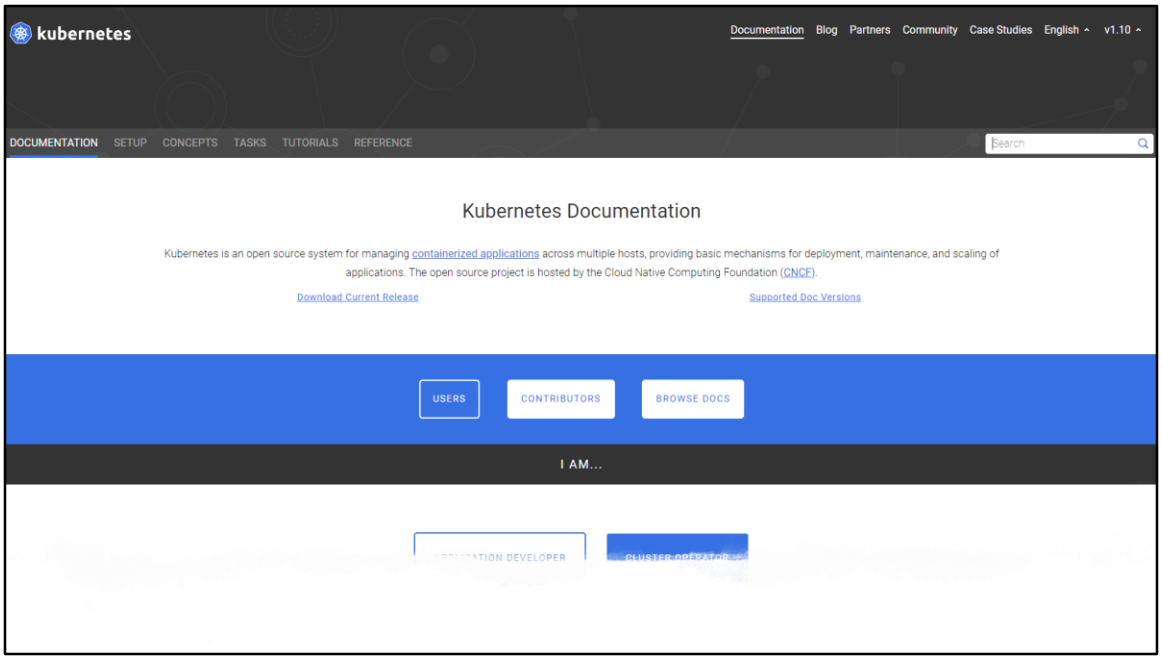
The Kubernetes Application Developers Certification, developed by the Cloud Native Computing Foundation in collaboration with The Linux Foundation, does just that. It helps you stand out in the crowd and allows companies to quickly hire high-quality engineers like you. On attaining the certification,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



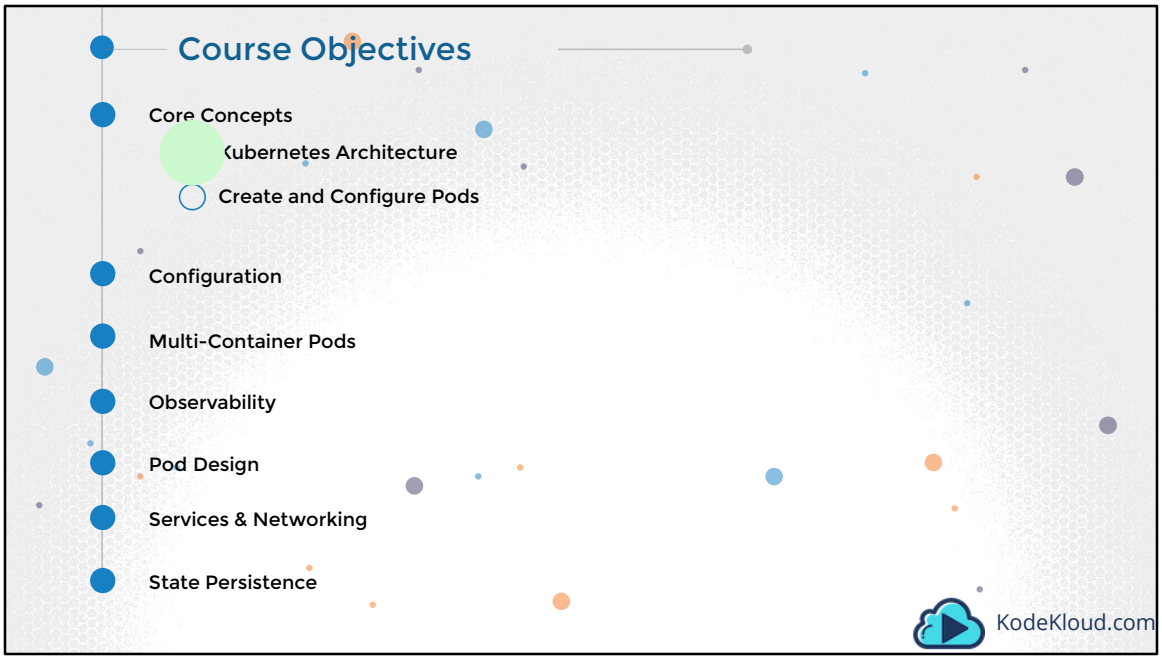
You can read more about the Certification at cncf.io/certification/CKAD. As of today, the exam costs 300 USD, with one FREE retake. This means that in case you don't manage to pass on the initial attempt, which I am sure you will, you have one more attempt available for FREE within the next 12 months. 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.



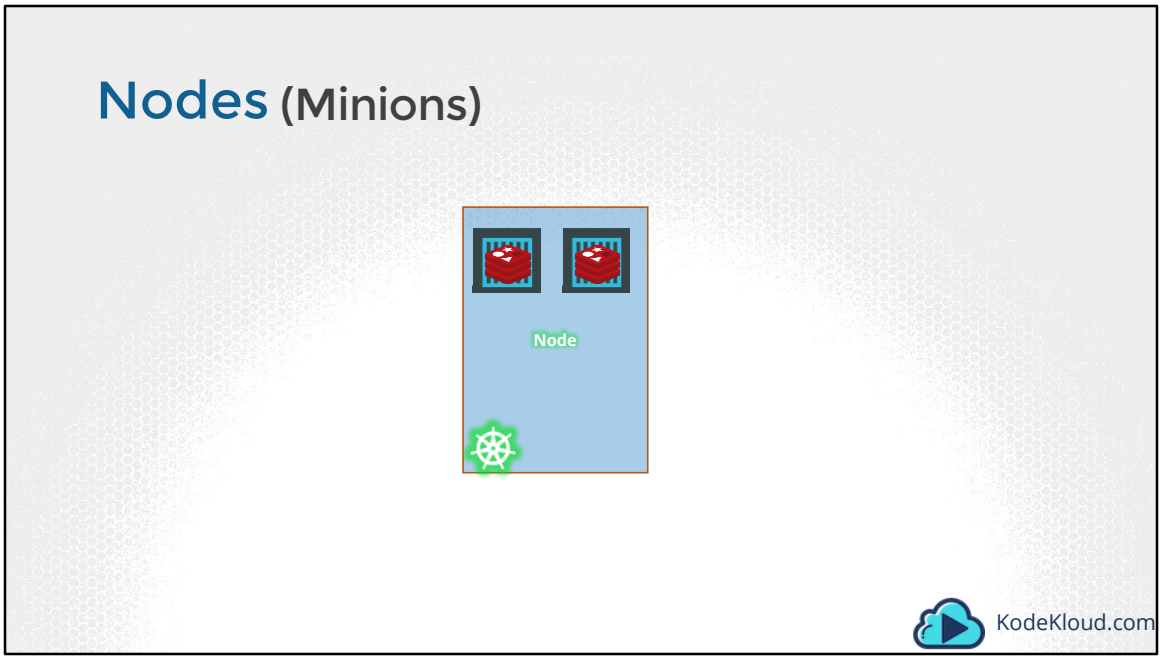
Unlike most of the Certification exams out there, the Kubernetes Certification is not a multiple-choice exam. It is an online, performance-based exam that tests your hands-on skills with the technology. This would mean that you don't have to worry about memorizing lots of different numbers in preparation 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.



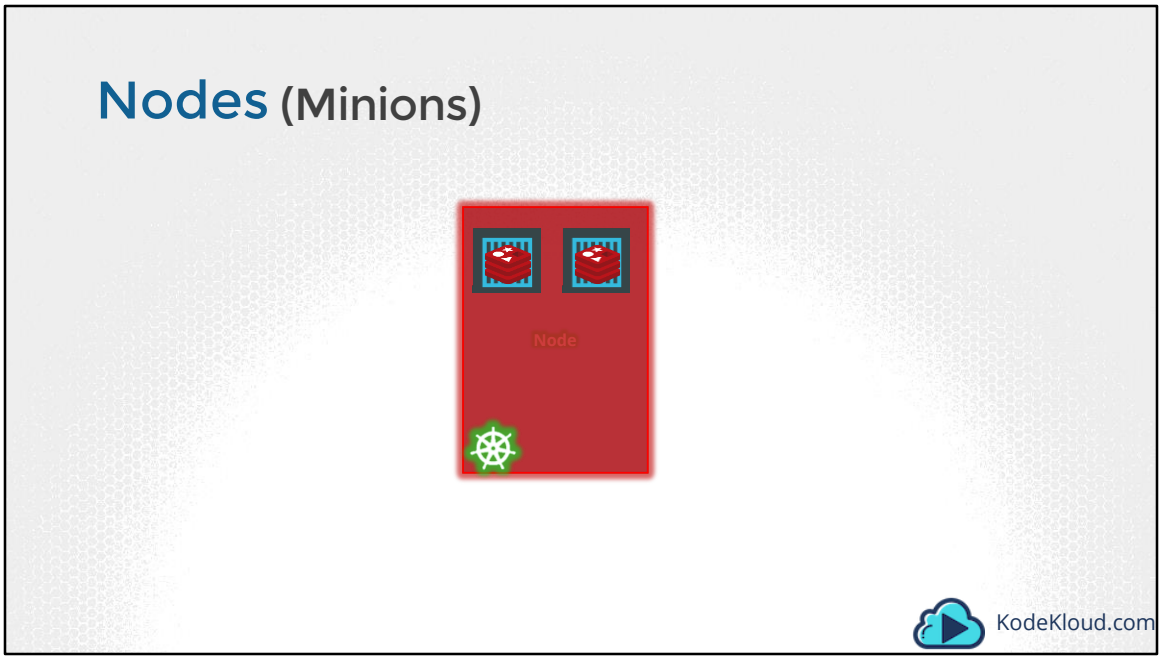
In this lecture we start with the Core Concepts in Kubernetes - The Kubernetes Architecture. The Kubernetes Application Developers certification exam does not focus on setting up a Kubernetes cluster, it falls more under the Administrators certification. So as long as you have a working cluster ready, 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.



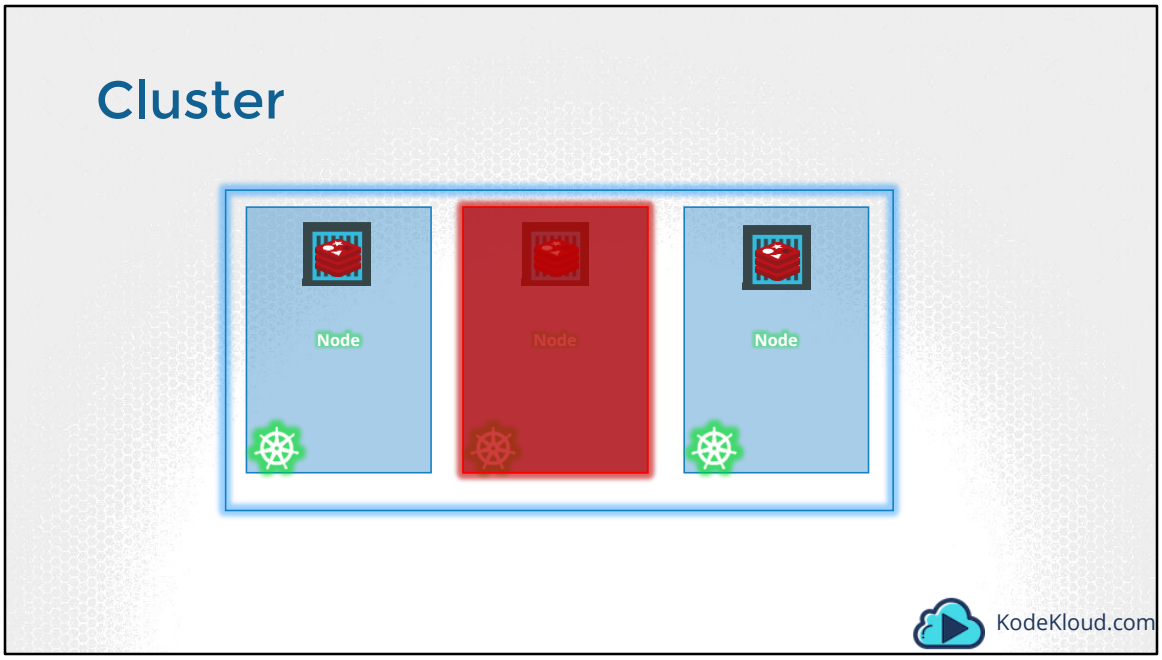
Kubernetes is configured on one or more Nodes. A node is a machine – physical or virtual – on which kubernetes is installed. A node is a worker machine and this is were containers are hosted.

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

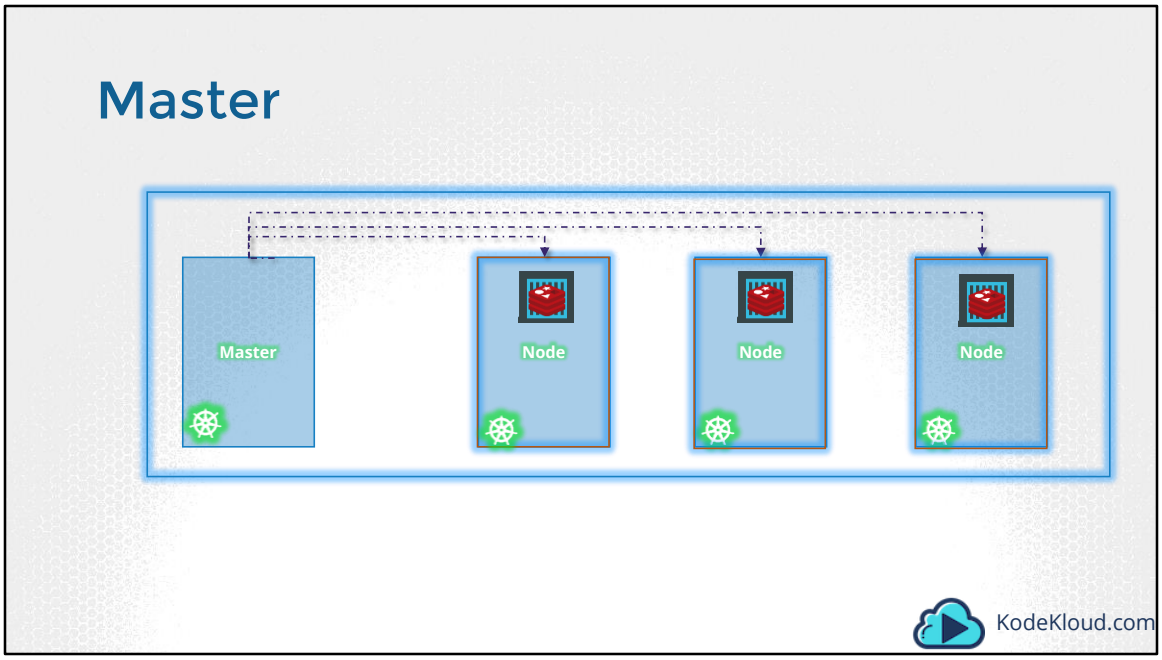
But what if the node on which our application is running fails? Well, obviously our application goes down. So you need to have more than one nodes for high availability and scaling.



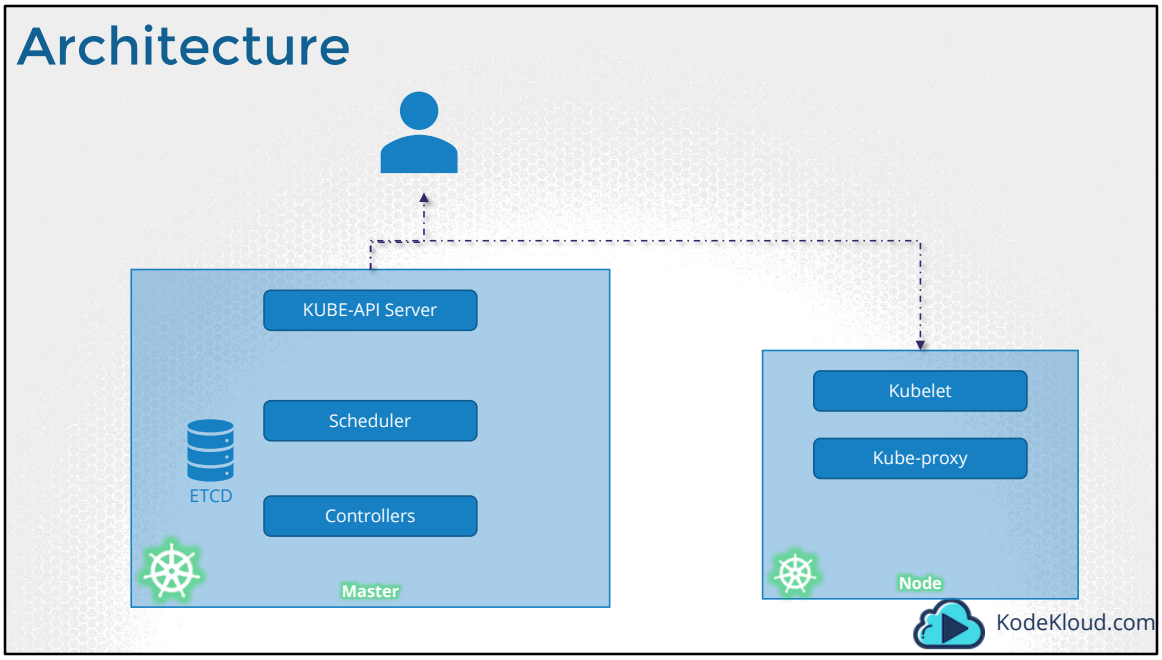
But what if the node on which our application is running fails? Well, obviously our application goes down. So you need to have more than one nodes for high availability and scaling.



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.



Now we have a cluster, but who is responsible for managing the cluster? Were 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 were 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.



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.

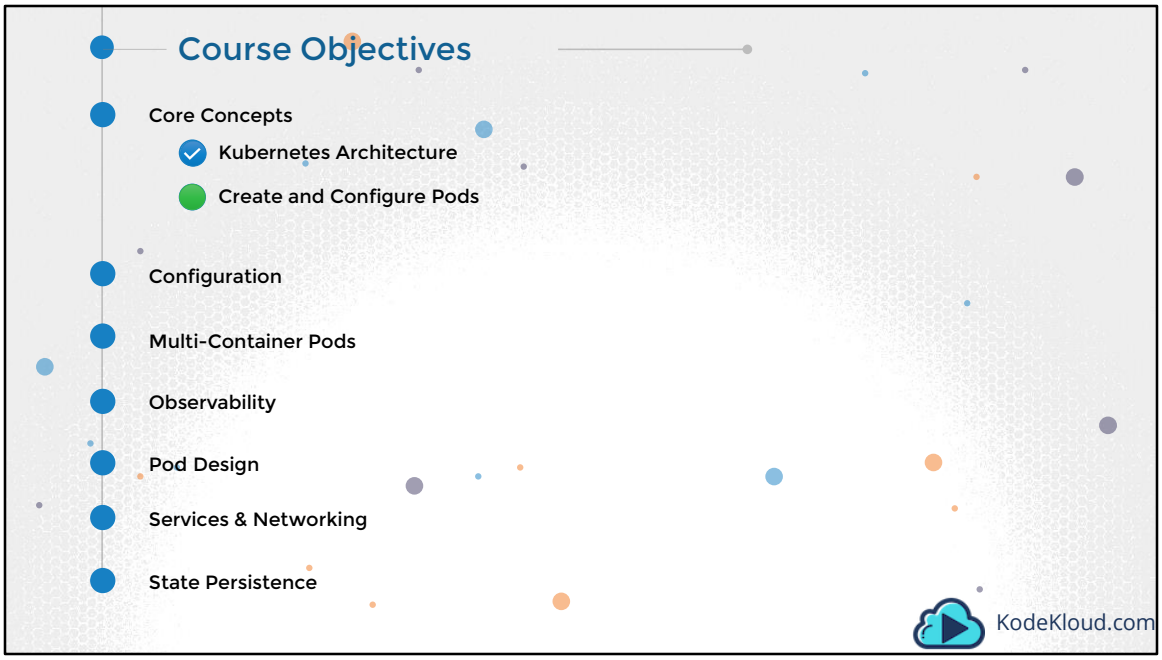
The controllers are the brain behind orchestration. They are responsible for noticing and responding when nodes, containers or endpoints goes down. The controllers makes decisions to bring up new containers in such cases.

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.

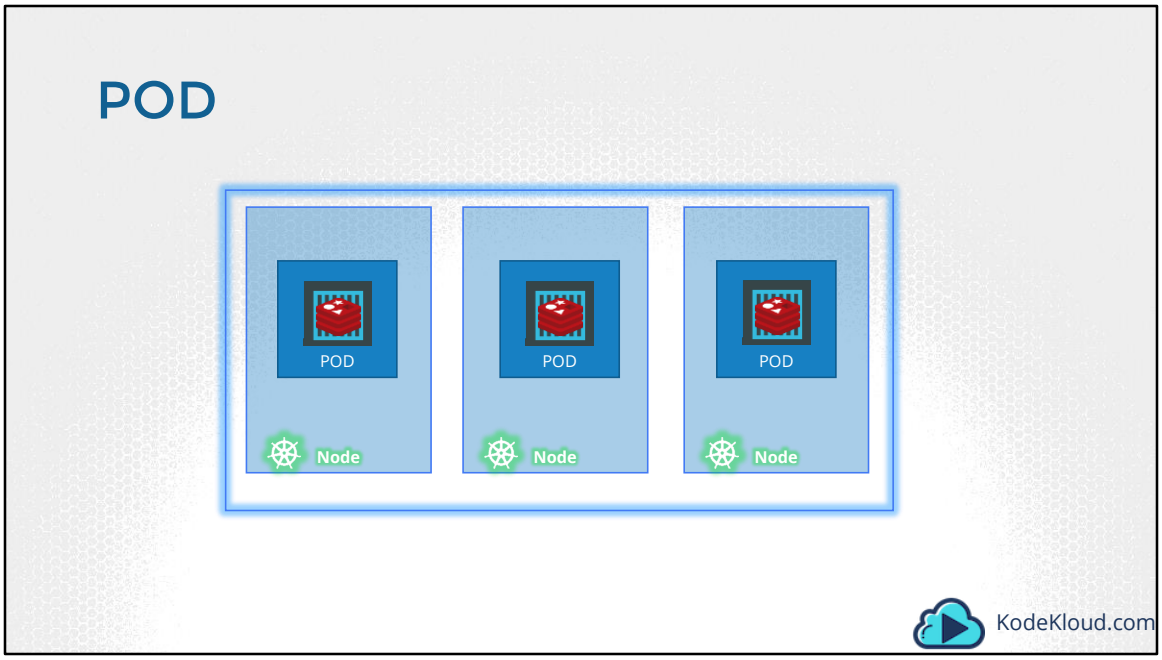
An additional component on the Node is the kube-proxy. It takes care of networking within Kubernetes.

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

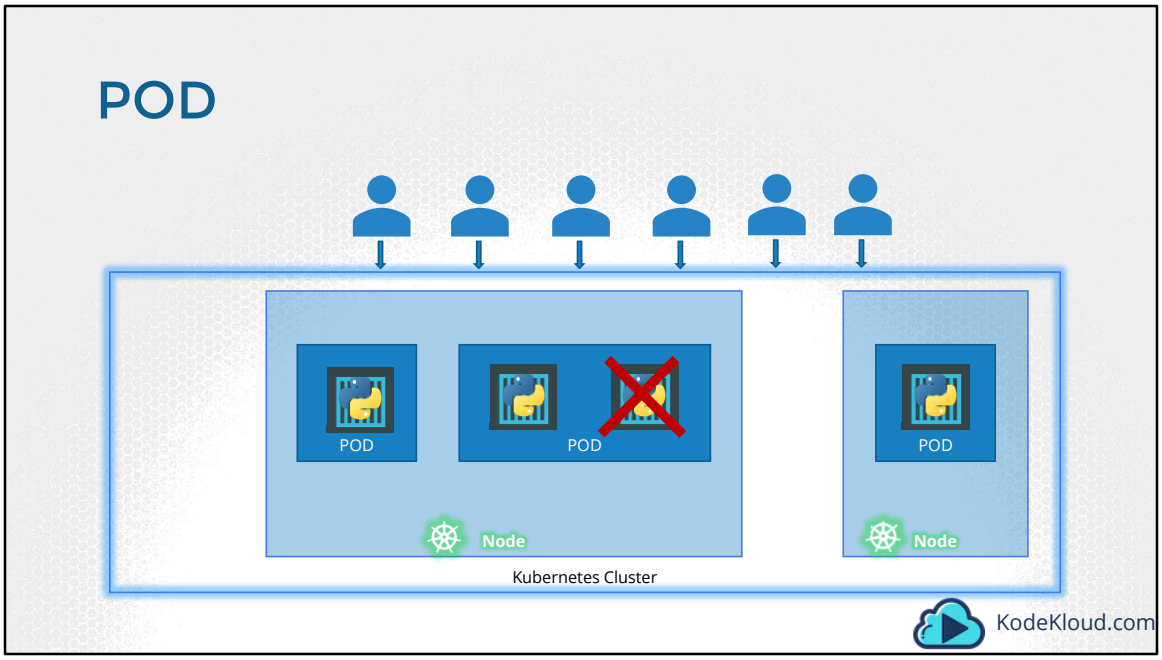


… In this lecture we will discuss about PODs. We will first understand what PODs are and then practice developing POD definition files. You will work with advanced POD definition files and also troubleshoot issues with existing ones. This way you will get enough hands-on practice.

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.



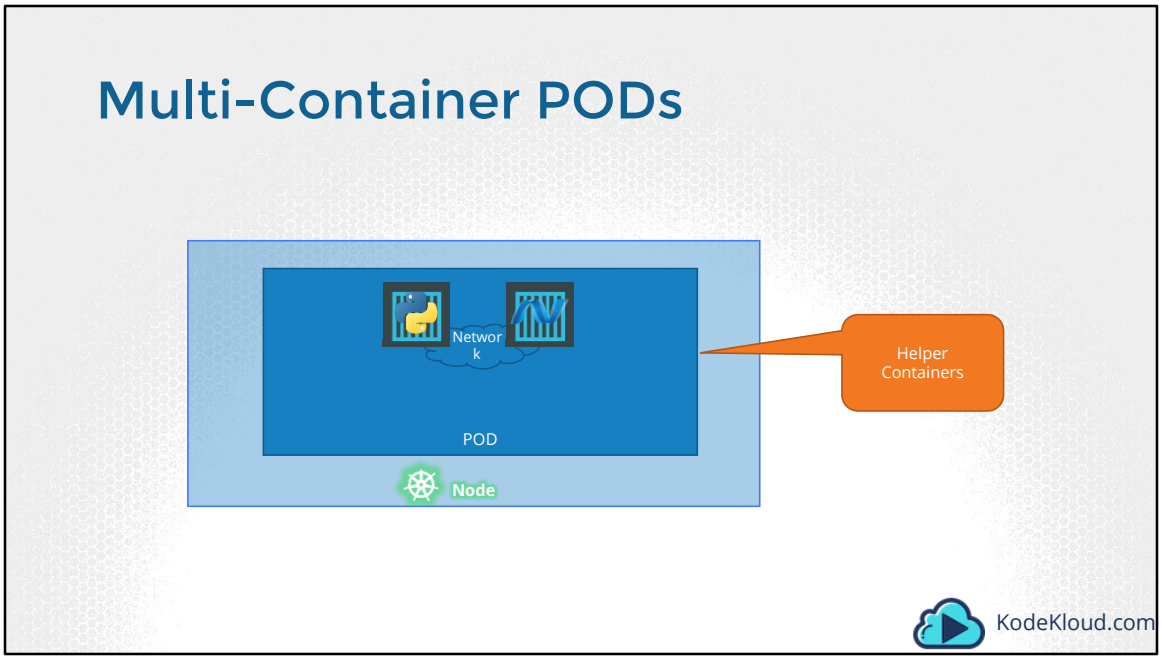
As we discussed before, with kubernetes our ultimate aim is to deploy our application in the form of containers on a set of machines that are configured as worker nodes in a cluster. However, kubernetes does not deploy containers directly on the worker nodes. The containers are encapsulated into a Kubernetes 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.



Here we see the simplest of simplest cases were 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, were 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.

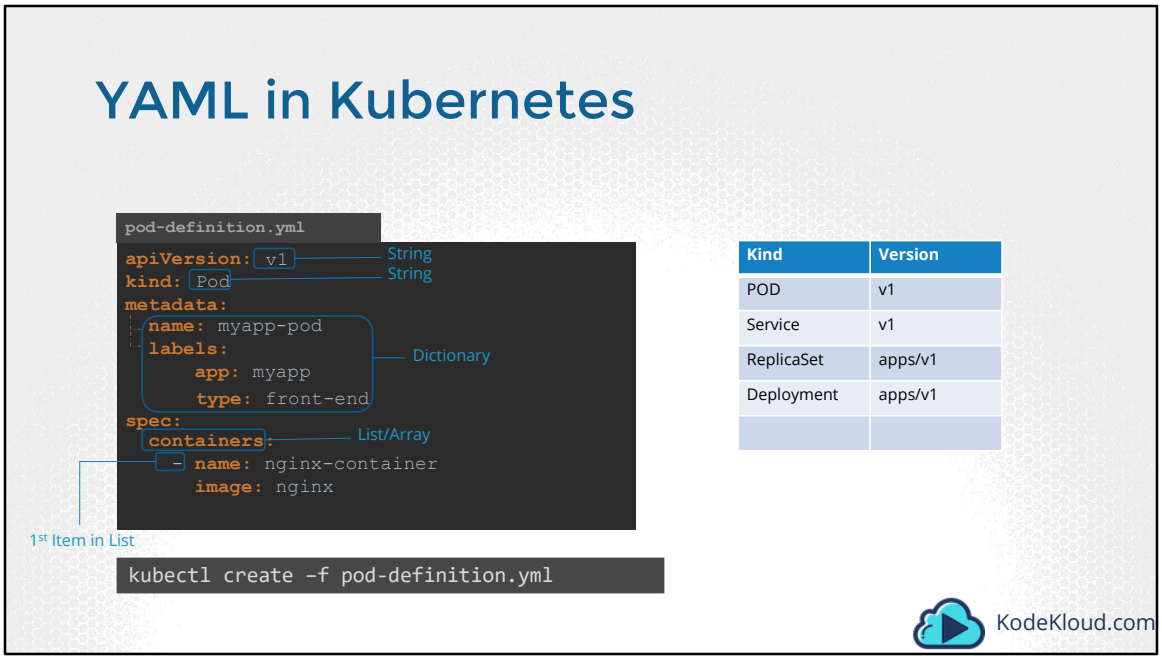
What if the user base FURTHER increases and your current node has no sufficient capacity? Well THEN you can always deploy additional PODs on a new node in the cluster. You will have a new node added to the cluster to expand the cluster's physical capacity. <pause> SO, what I am trying to illustrate in 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.



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 were 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.



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 were 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.

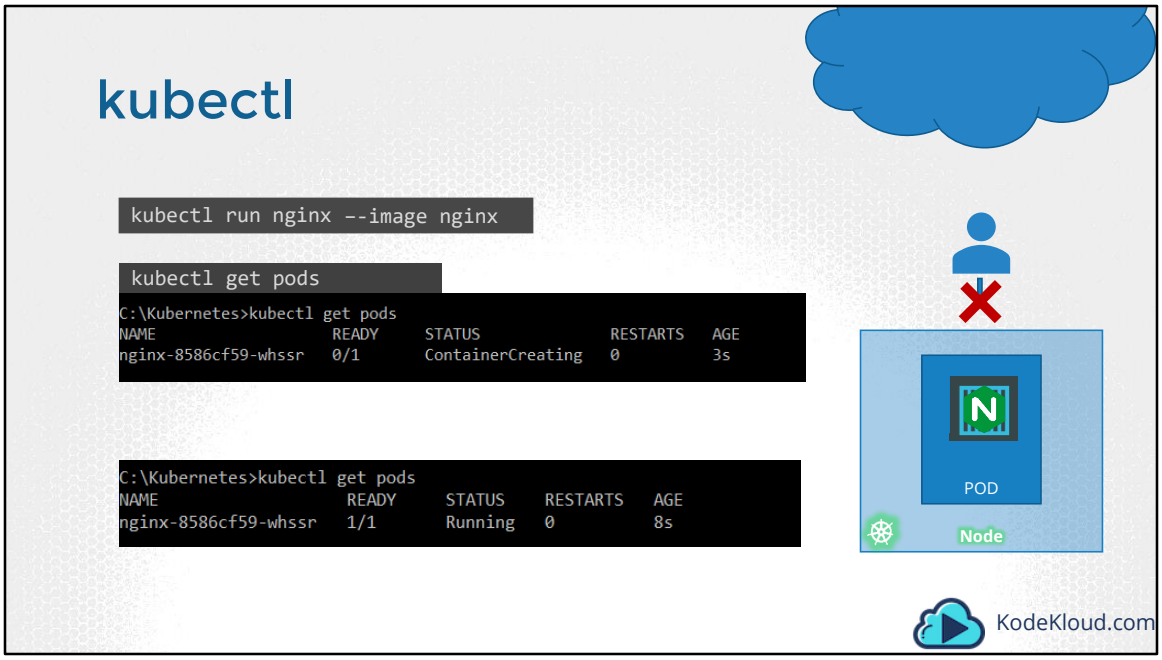
The next is metadata. The metadata is data about the object like its name, labels etc. As you can see unlike the first two were you specified a string value, this, is in the form of a dictionary. So everything under metadata is intended to the right a little bit and so names and labels are children of 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 were 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.

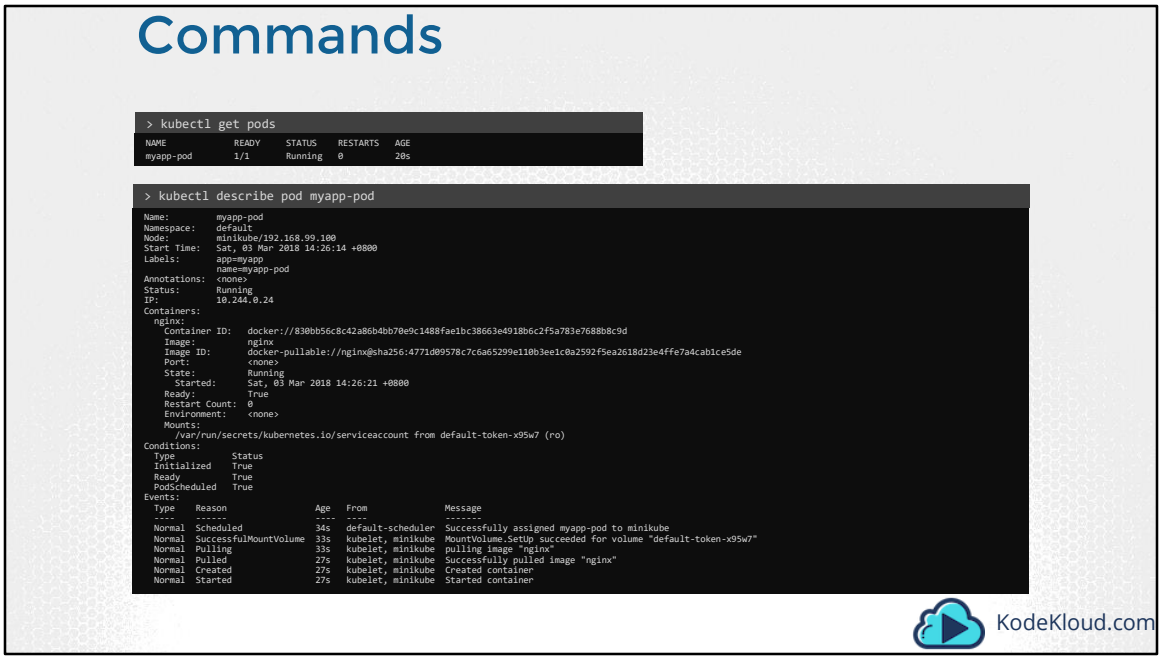


Let us now look at how to deploy PODs. Earlier we learned about the kubectl run command. What this command really does is it deploys a docker container by creating a POD. So it first creates a POD automatically and deploys an instance of the nginx docker image. But were does it get the application image 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 were 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.

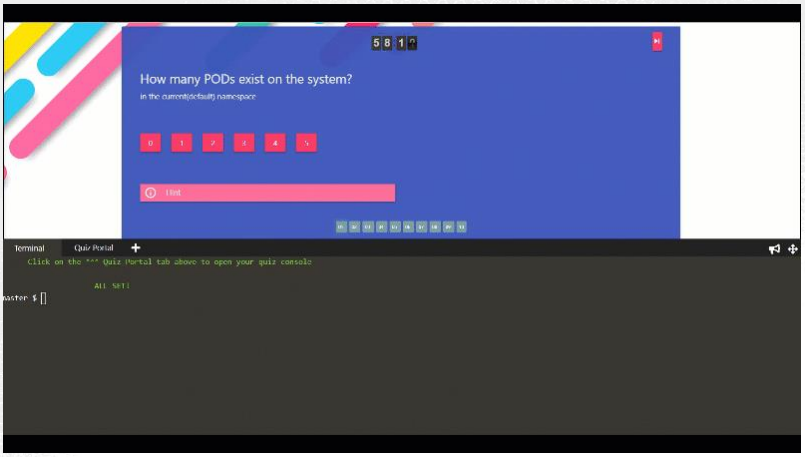
Also remember that we haven't really talked about the concepts on how a user can access the nginx web server. And so in the current state we haven't made the web server accessible to external users. You can access it internally from the Node though. For now we will just see how to deploy a POD and in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

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



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.

Practice Test



Check Link Below!

KodeKloud.com

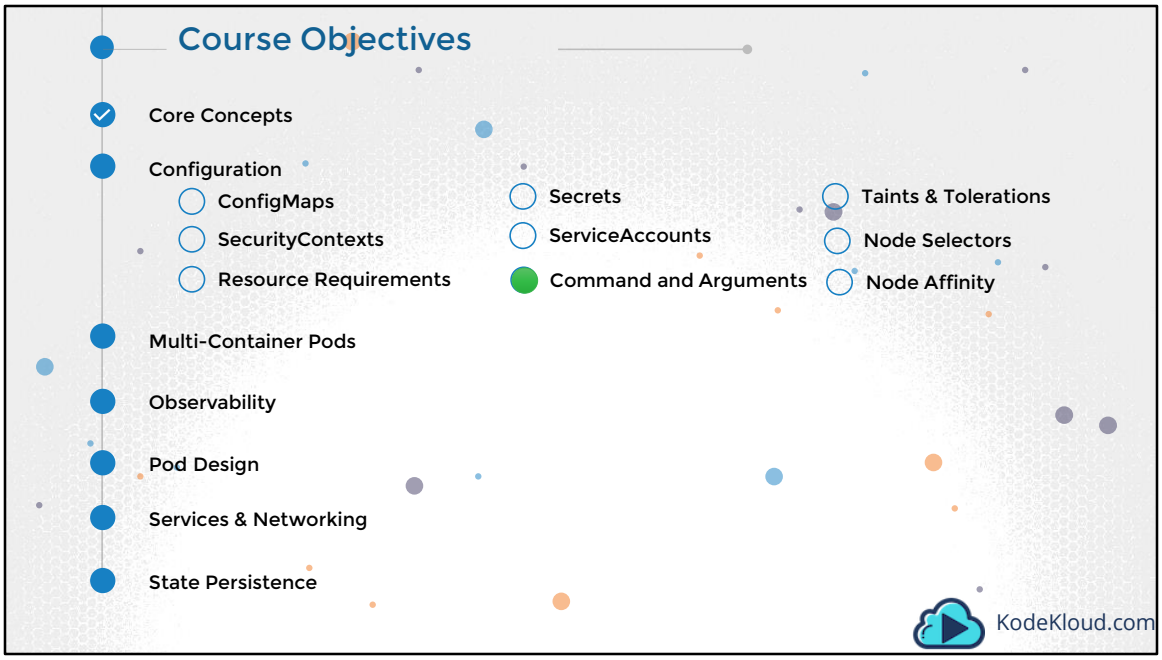
Access Test Here: https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743640

Course Objectives

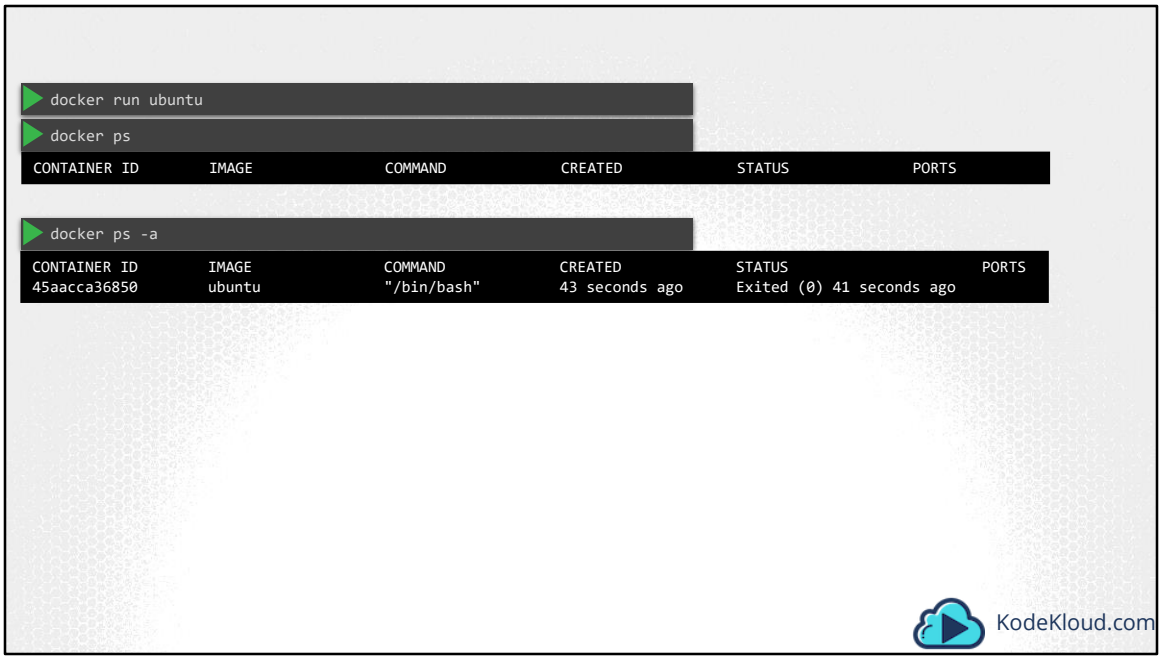
- Core Concepts
  - Kubernetes Architecture
  - Create and Configure Pods
- Configuration
- Multi-Container Pods
- Observability
- Pod Design
- Services & Networking
- State Persistence

KodeKloud.com


46
COMMANDS
&
ARGUMENTS
KodeKloud.com

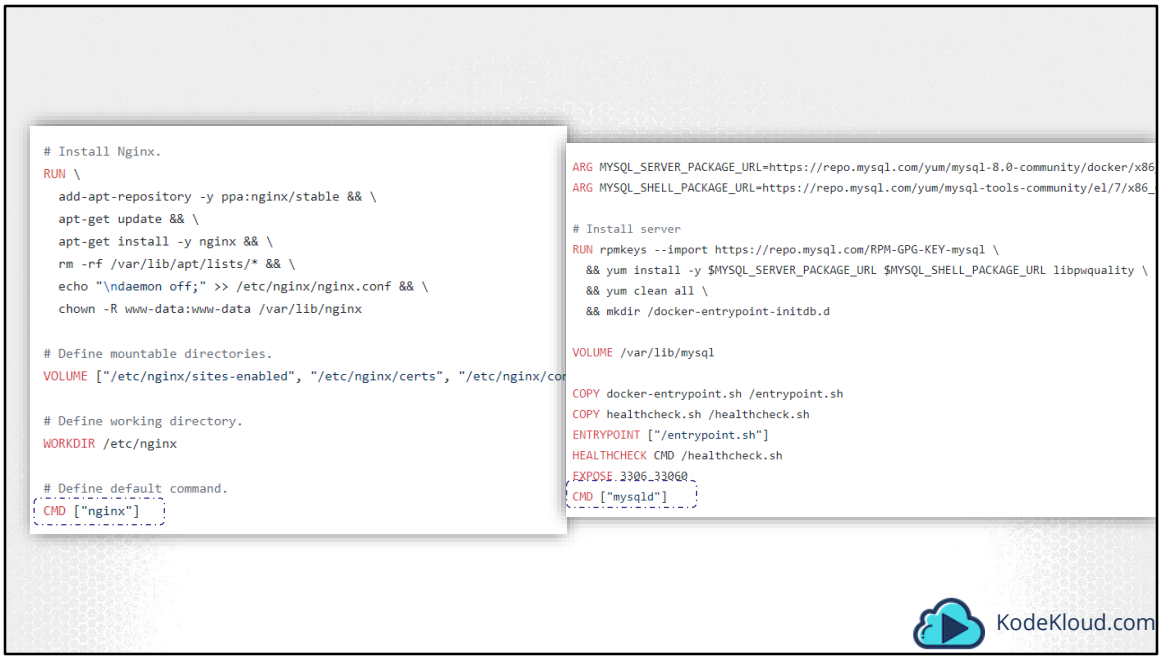


In this section we will talk about Command and Arguments in a Pod Definition. This is not listed as a required topic in the certification curriculum, but I think its important to explain it as it is a topic that is usually overlooked.



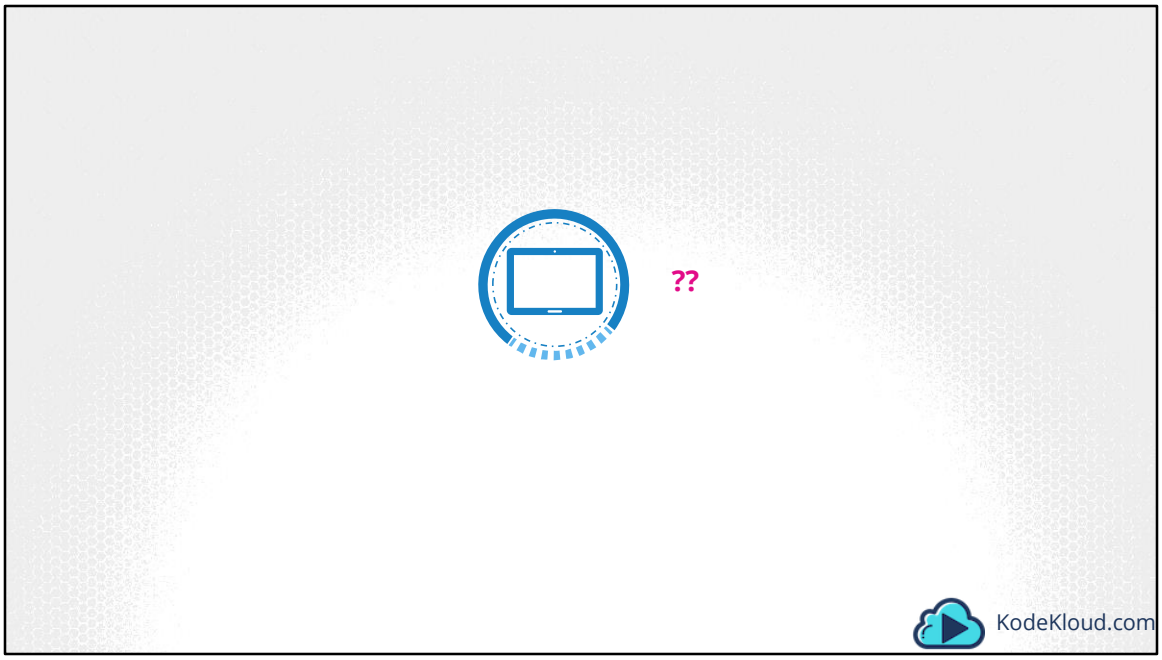
Let's first refresh our memory on commands in containers and docker. We will then translate this into PODs in the next lecture. Let's start with a simple scenario. Say you were to run a docker container from an Ubuntu image. When you run the "docker run ubuntu" command, it runs an instance of Ubuntu 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?

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.



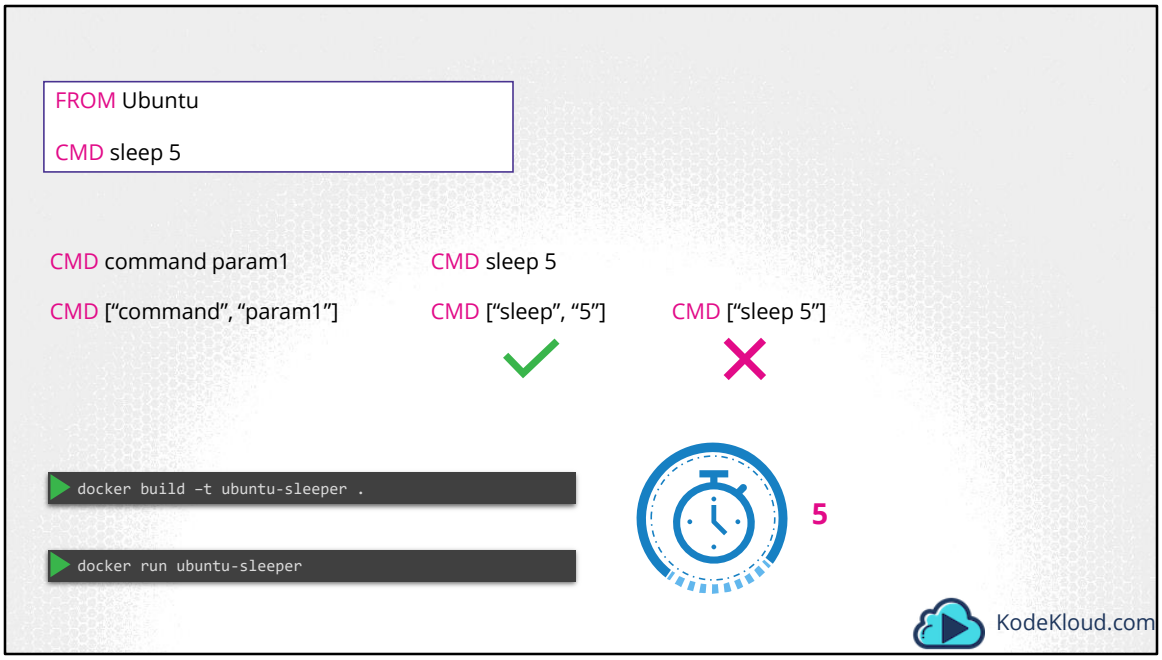
So who defines what process is run within the container? If you look at the Dockerfile for the NGINX image, you will see an Instruction called CMD which stands for command that defines the program that will be run within the container when it starts. For the NGINX image it is the nginx command, for the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

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.



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.

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.

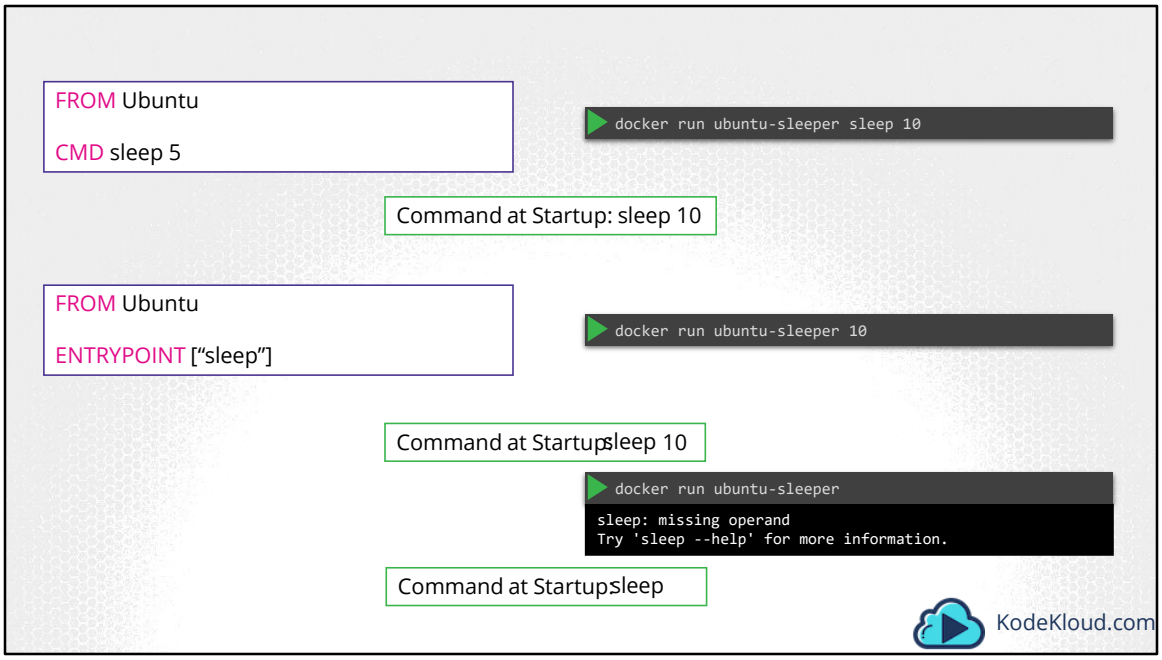


But how do you make that change permanent? Say you want the container to always run the sleep command when it starts.

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.

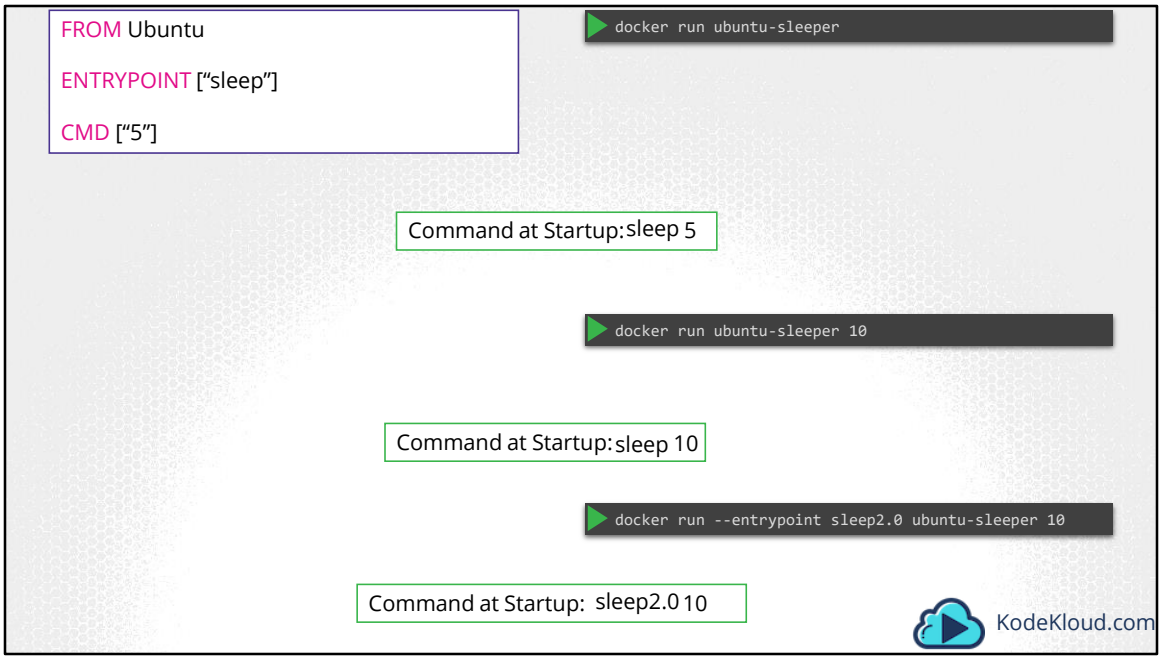


But what if I wish to change the number of seconds it sleeps. Currently it is hardcoded to 5 seconds. As we learned before one option is to run the docker run command with the new command appended to it. In this case sleep 10. And so the command that will be run at startup will be sleep 10. But it doesn't 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.

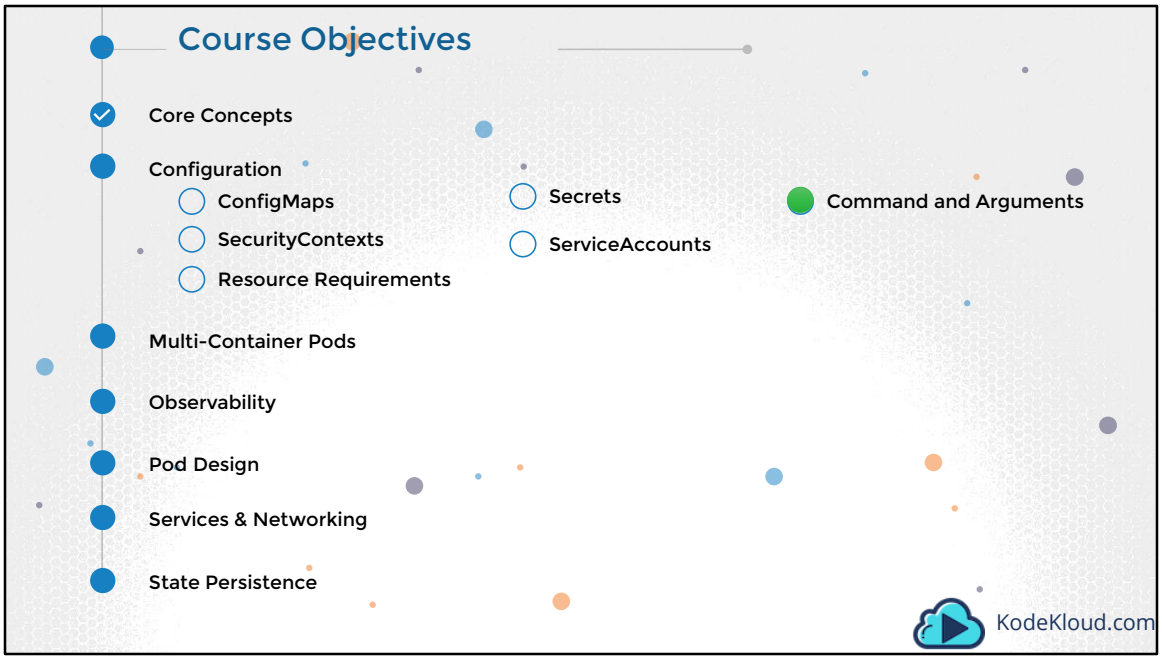
So that's the difference between the two. In case of the CMD instruction the command line parameters passed will get replaced entirely, whereas in case of entrypoint the command line parameters will get appended.

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?

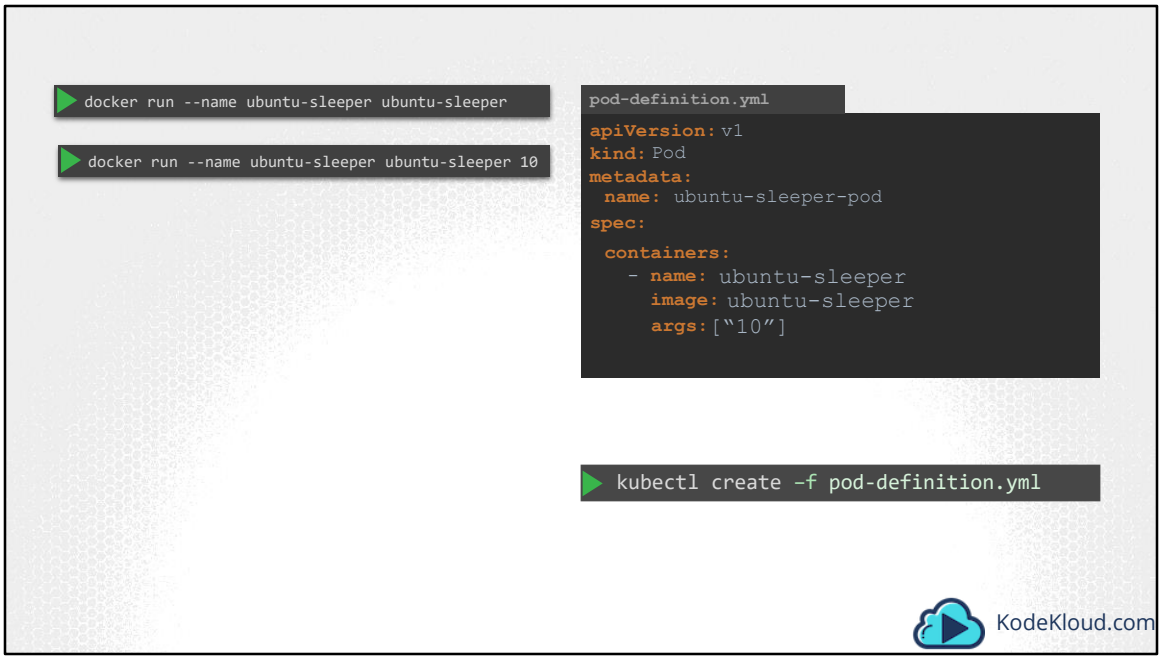


That's where you would use both Entrypoint as well as the CMD instruction. In this case the command instruction will be appended to the entrypoint instruction. So at startup, the command would be sleep 5, if you didn't specify any parameters in the command line. If you did, then that will override the 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

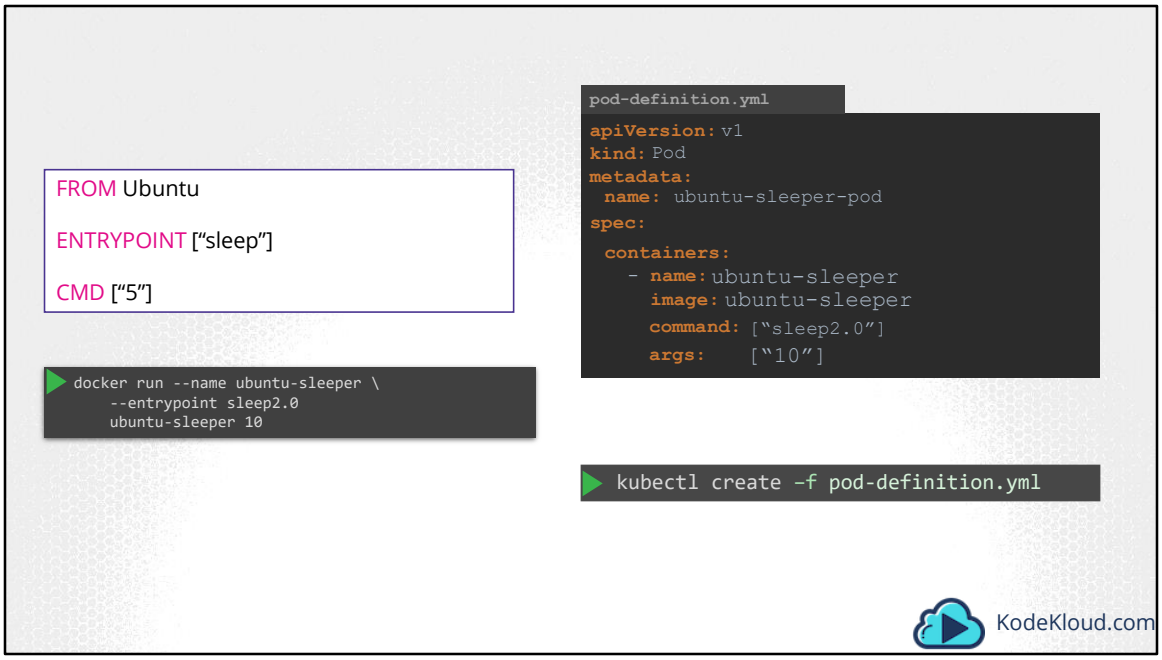


We will now look at Command and Arguments in a Kubernetes POD.

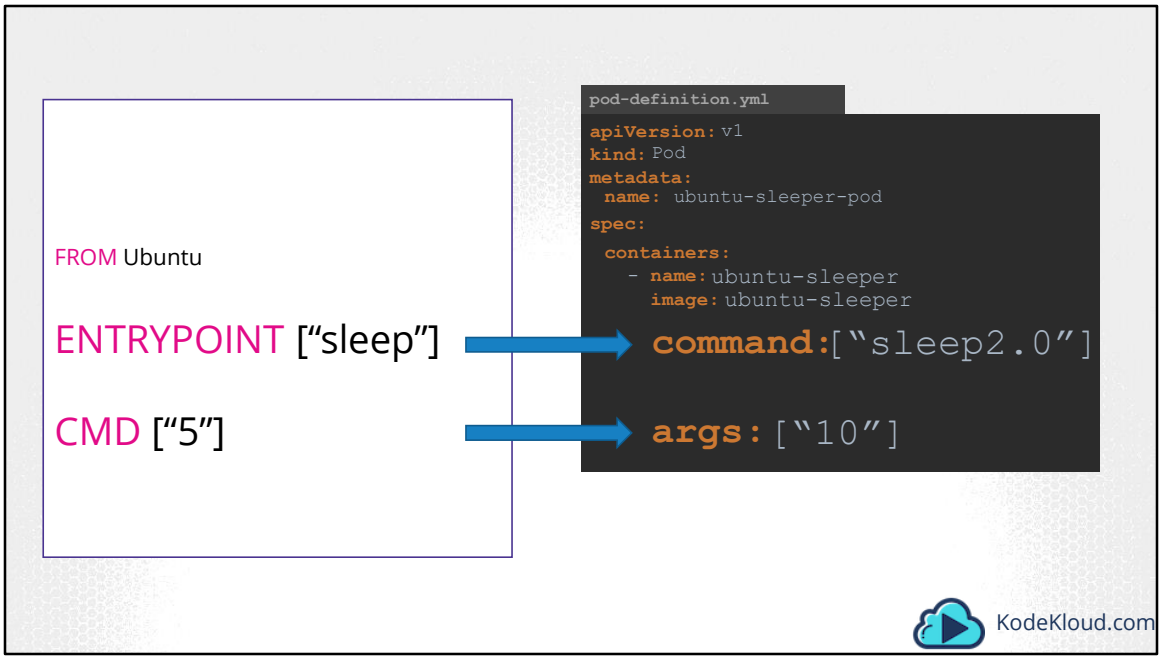


In the previous lecture we created a simple docker image that sleeps for a given number of seconds. We named it ubuntu-sleeper and we ran it using the docker command docker run ubuntu-sleeper. By default it sleeps for 5 seconds, but you can override it by passing a command line argument. We will now 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.

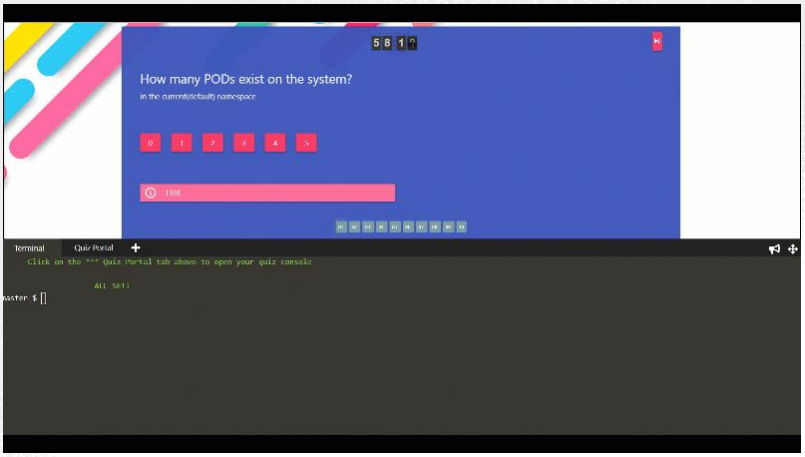


Let us try to relate that to the Dockerfile we created earlier. The Dockerfile has an Entrypoint as well as a CMD instruction specified. The entrypoint is the command that is run at startup, and the CMD is the default parameter passed to the command. With the args option in the pod-definition file we 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.



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!

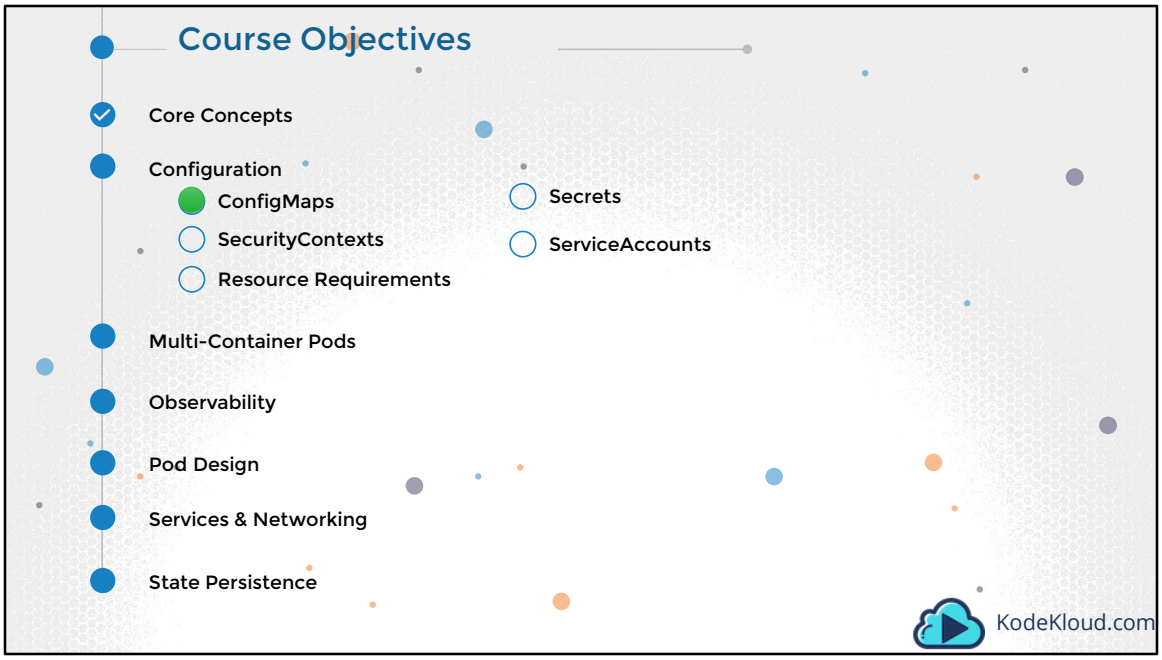
KodeKloud.com

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

References

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

KodeKloud.com



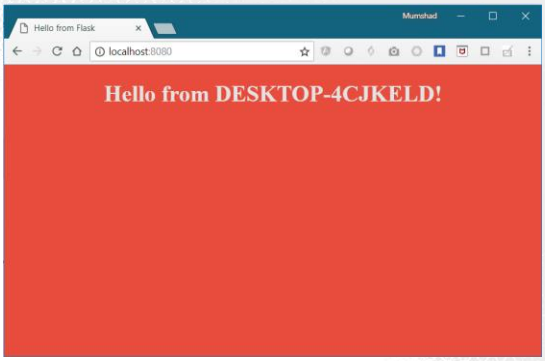
In this section we will talk about concepts around Configuration in Kubernetes. We will start with ConfigMaps. First we will see what a configuration item is by using a simple example of a web application and how it is set in Docker. And then we will see how it is configured in Kubernetes. If you know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


64
ENVIRONMENT VARIABLES
KodeKloud.com

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")
```




```
python app.py
```

KodeKloud.com

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")
```

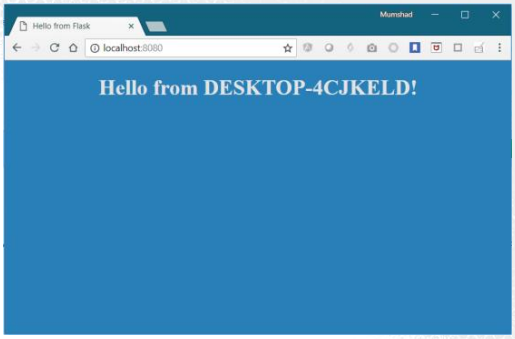
KodeKloud.com

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

app.py

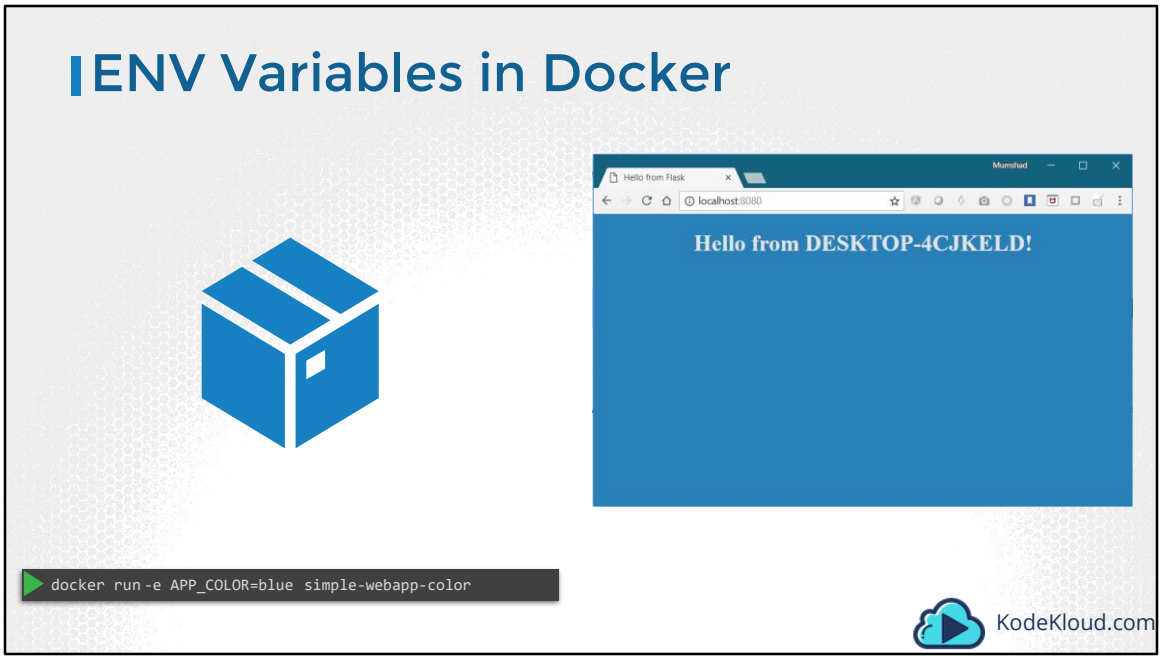
```
import os
from flask import Flask

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.com

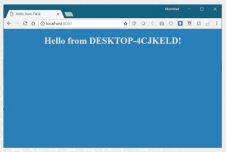
And into, say an environment variable called APP_COLOR. The next time you run the application, set an environment variable called APP_COLOR to a desired value, and the application now has a new color.



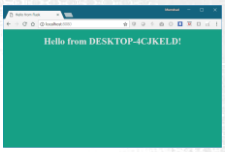
Once your application get's packaged into a Docker image, you would then run it with the docker run command followed by the name of the image. However, if you wish to pass the environment variable as we did before, you would now use the docker run command's –e option to set an environment variable within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

ENV Variables in Docker

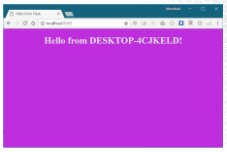
```
docker run  -e APP_COLOR=blue    simple-webapp-color
```




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

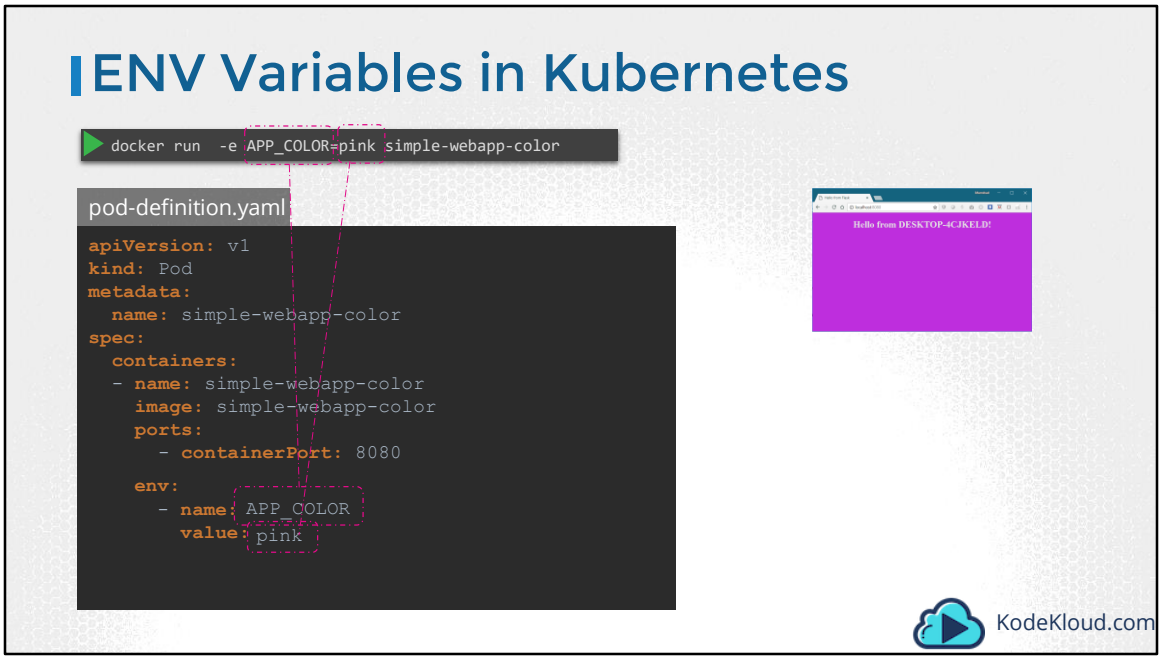



```
docker run  -e APP_COLOR=pink simple-webapp-color
```

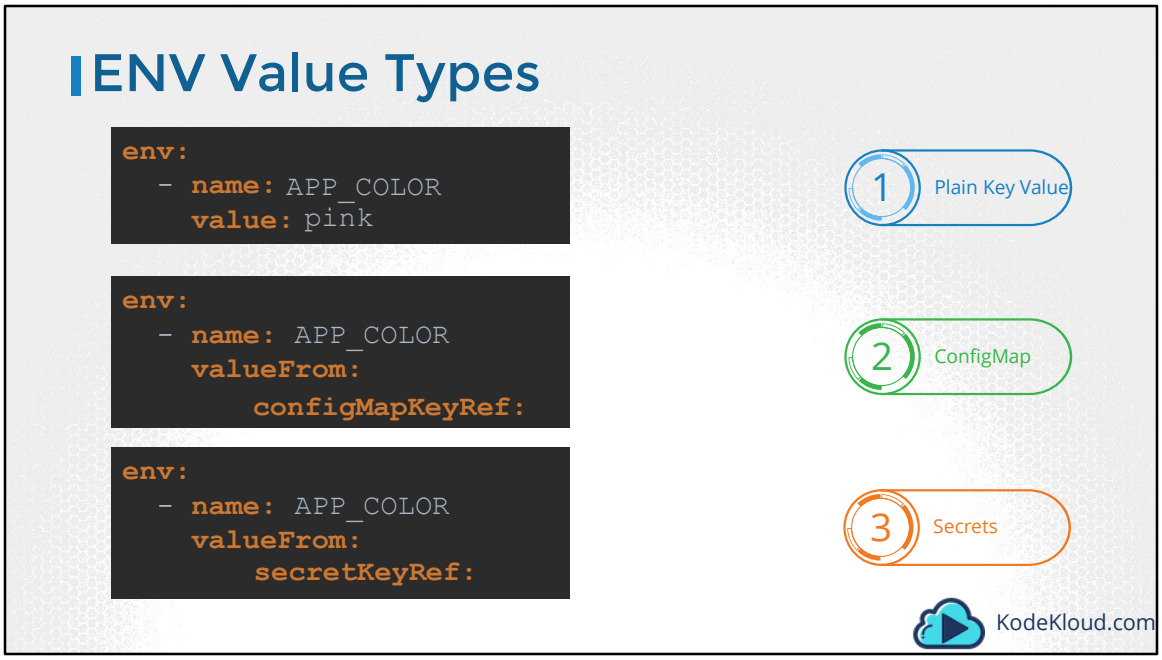


KodeKloud.com

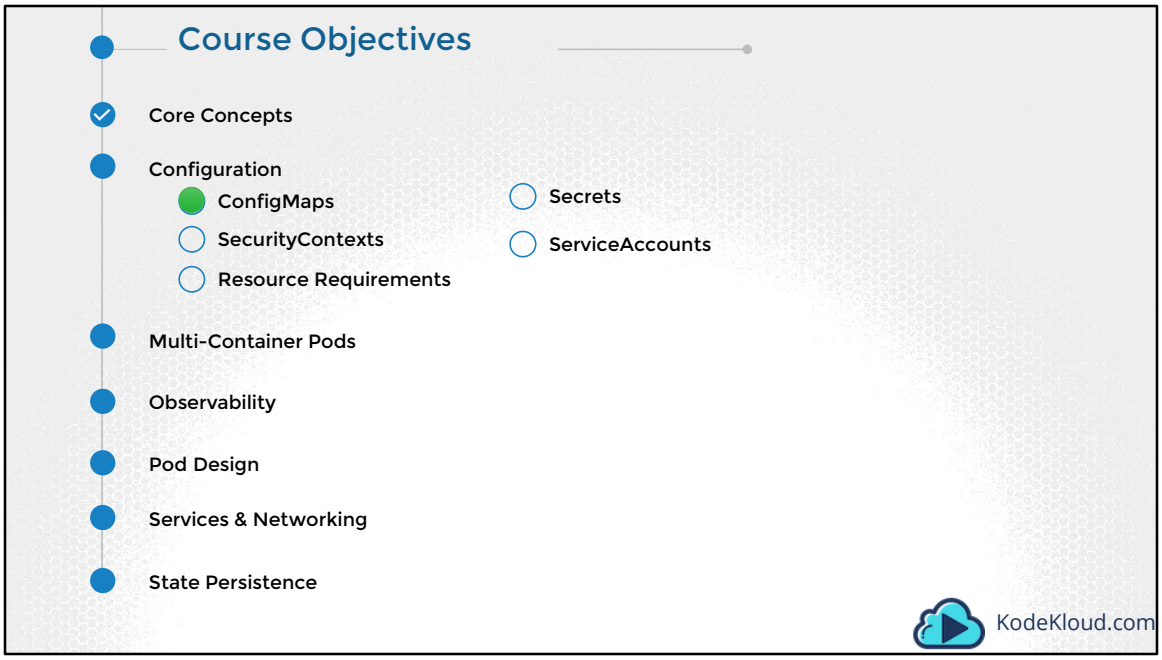
To deploy multiple containers with different colors, you would run the docker command multiple times and set a different value for the environment variable each time.



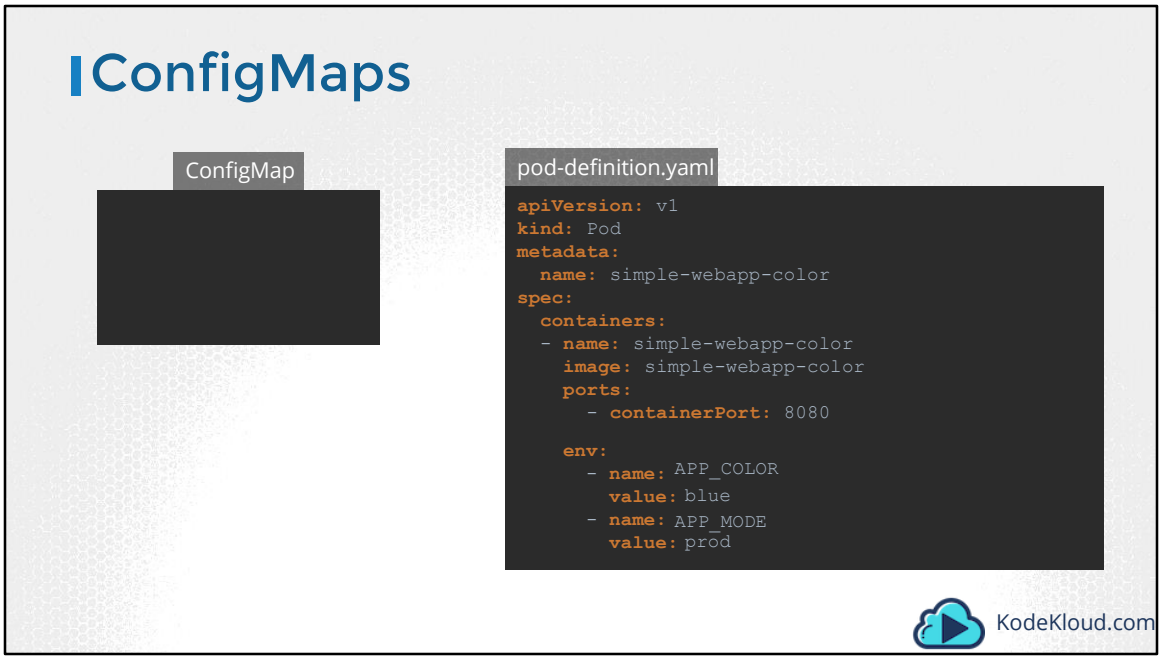
Let us now see how to pass in an environment variable in Kubernetes. Given a pod definition file, which uses the same image as the docker command. To set an environment variable, use the ENV property. ENV is an array. So every item under the env property starts with a dash, indicating an item in the 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.



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.



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

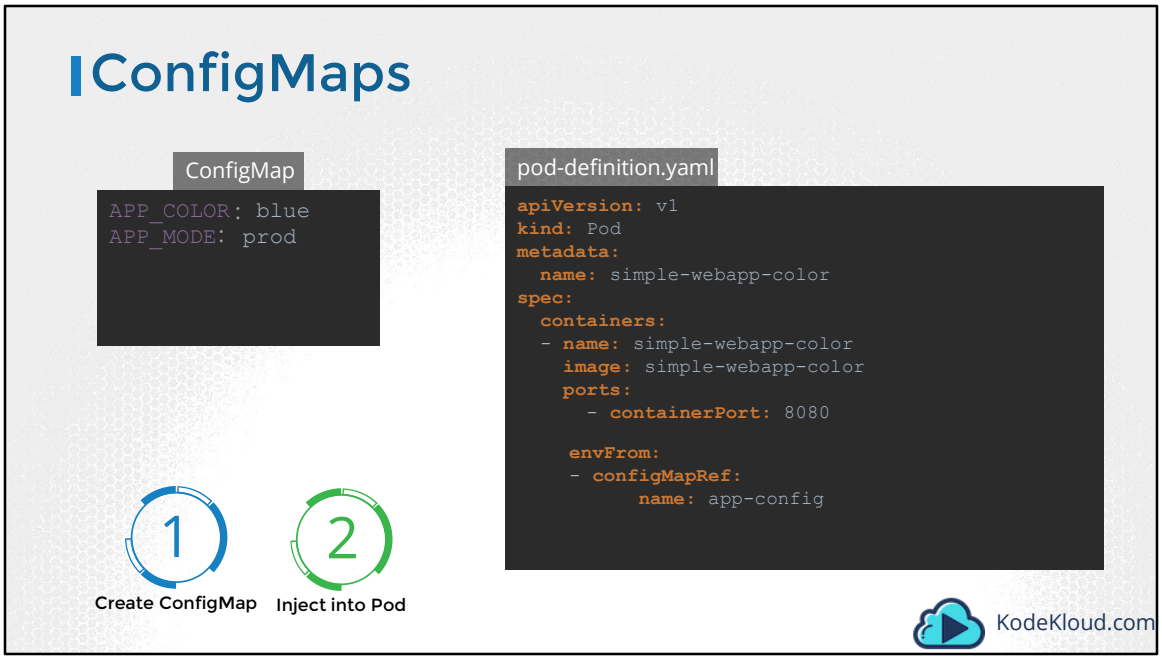


In the previous lecture we saw how to define environment variables in a pod definition file. When you have a lot of pod definition files, it will become difficult to manage the environment data stored within various files. We can take this information out of the pod definition file and manage it centrally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

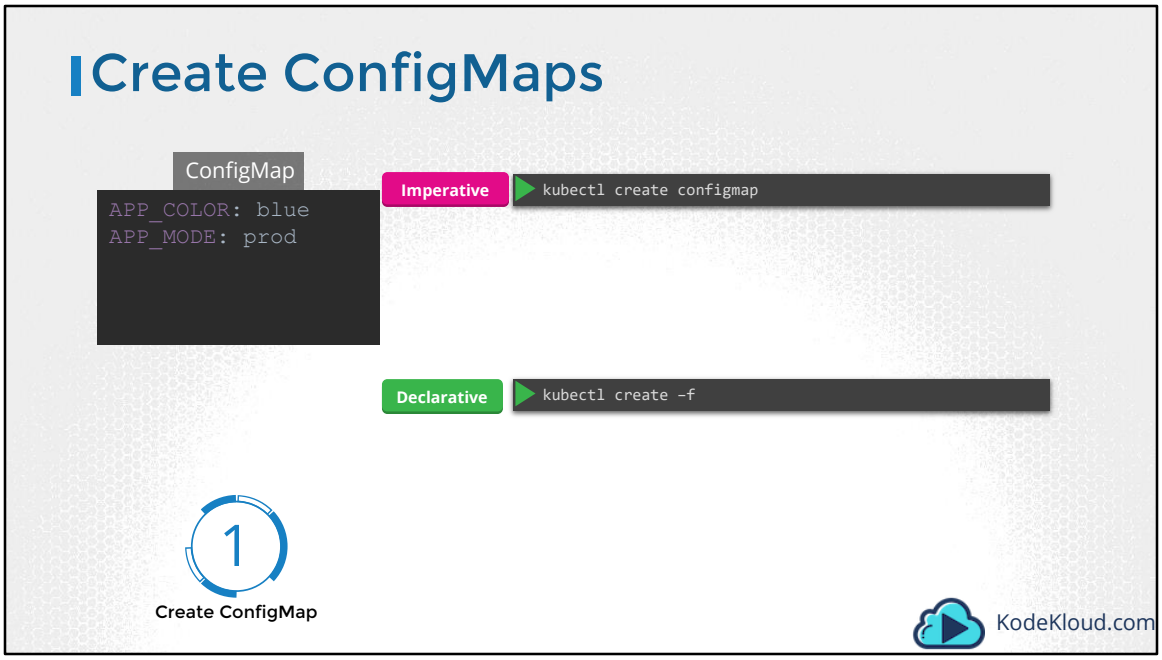


In the previous lecture we saw how to define environment variables in a pod definition file. When you have a lot of pod definition files, it will become difficult to manage the environment data stored within various files. We can take this information out of the pod definition file and manage it centrally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

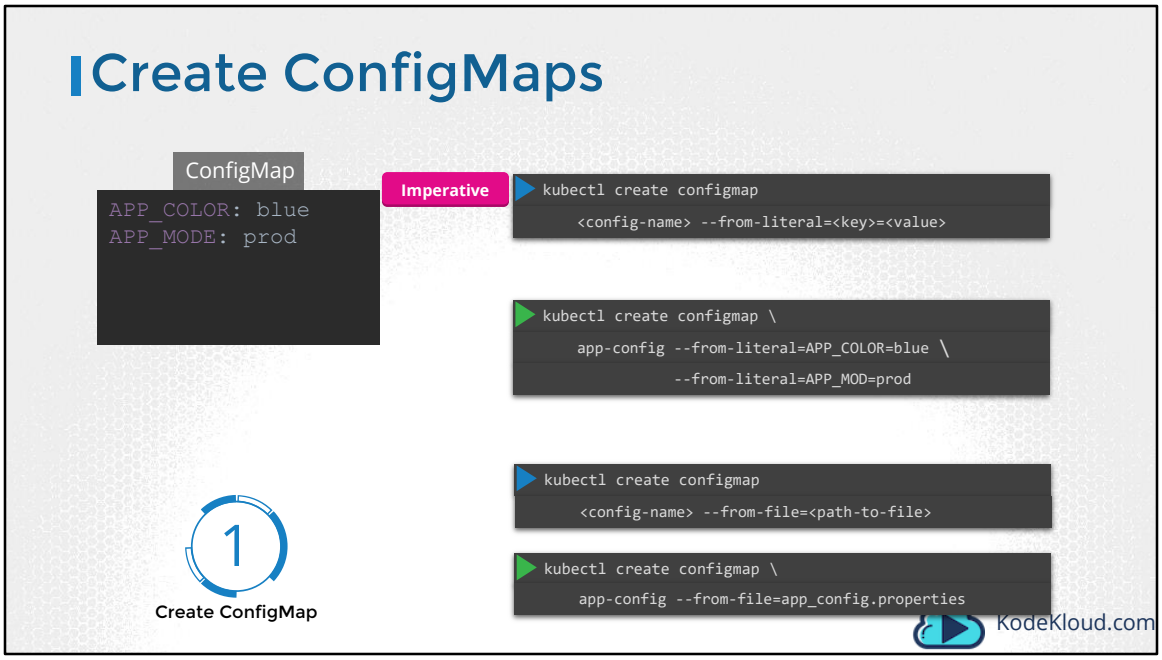
So there are two phases involved in configuring ConfigMaps. First create the ConfigMaps and second Inject them into the POD.



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.

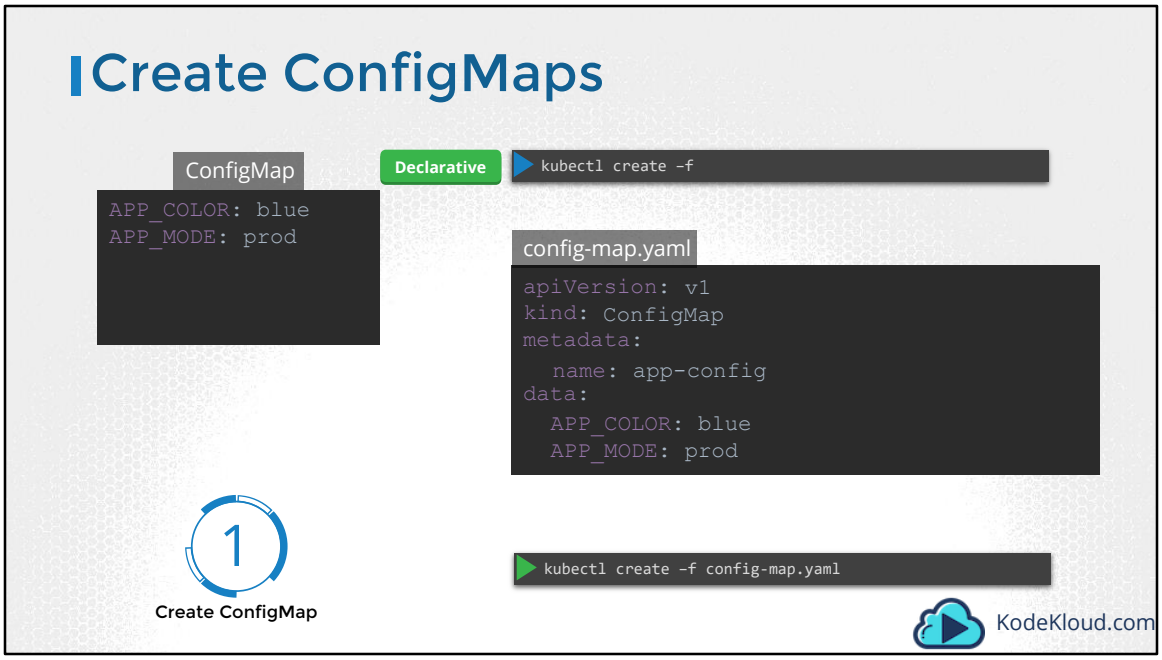
If you do not wish to create a configmap definition, you could simply use the kubectl create configmap command and specify the required arguments. Let's take a look at that first.

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.



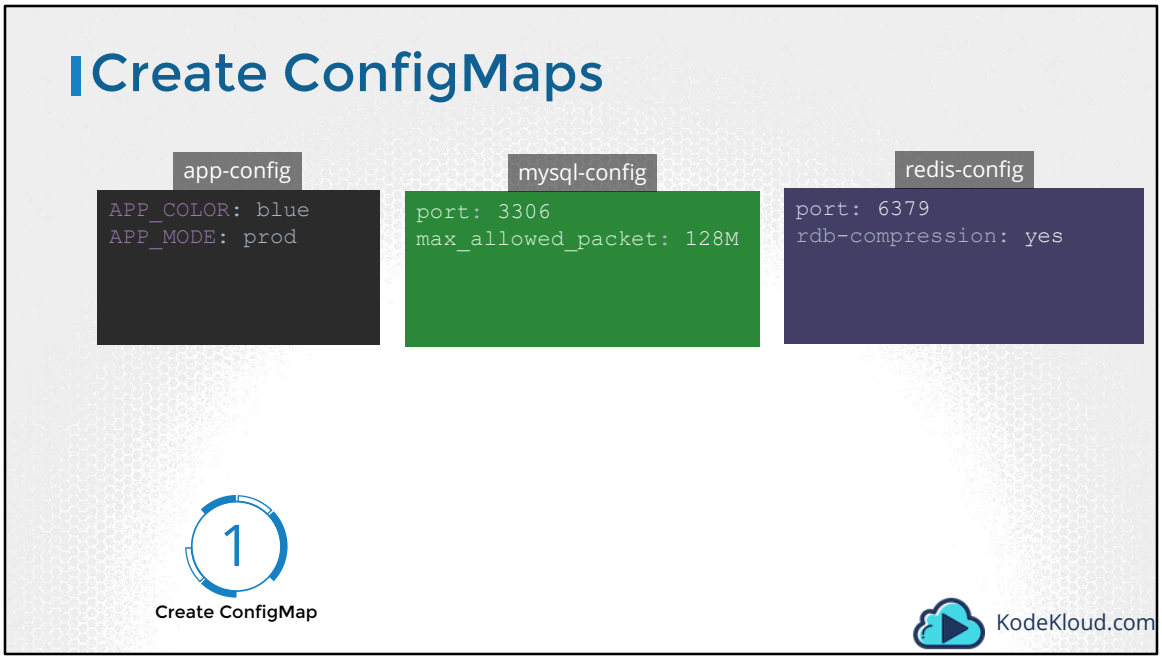
The command is followed by the config name and the option –from-literal. The from literal option is used to specify the key value pairs in the command itself. In this example, we are creating a configmap by the name app-config, with a key value pair APP_COLOR=blue. If you wish to add additional key value 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

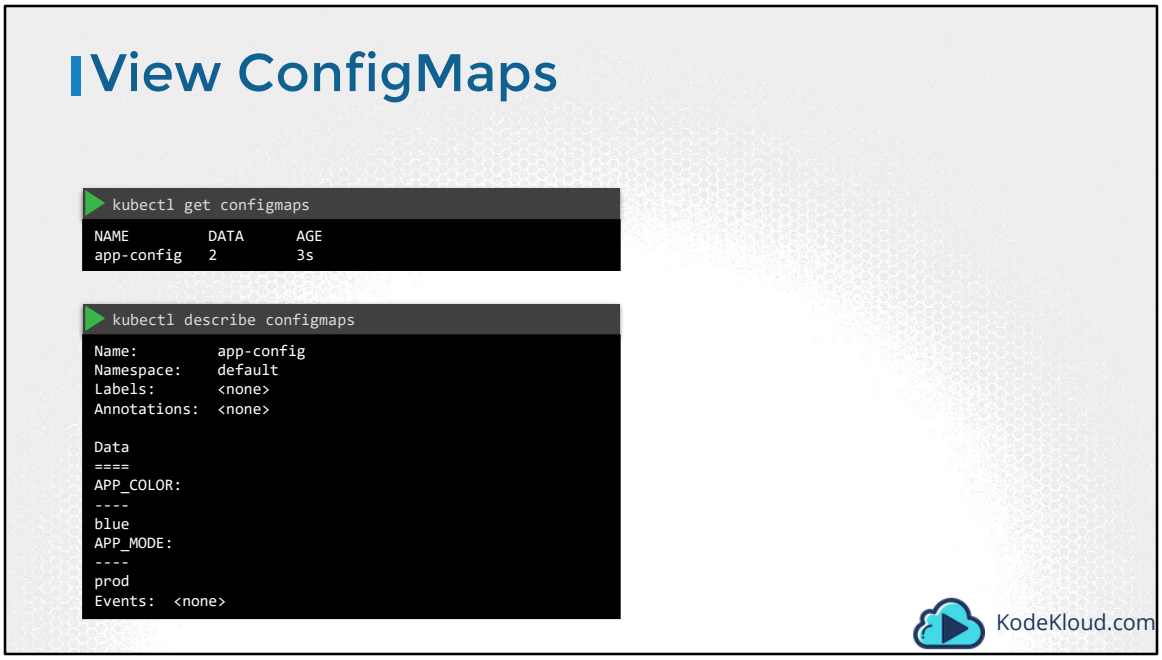


Let us now look at the declarative approach. For this we create a definition file, just like how we did for the pod. The file has apiVersion, kind, metadata and instead of spec, here we have “data”. The apiVersion is v1, kind is ConfigMap. Under metadata specify the name of the configmap. We will call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

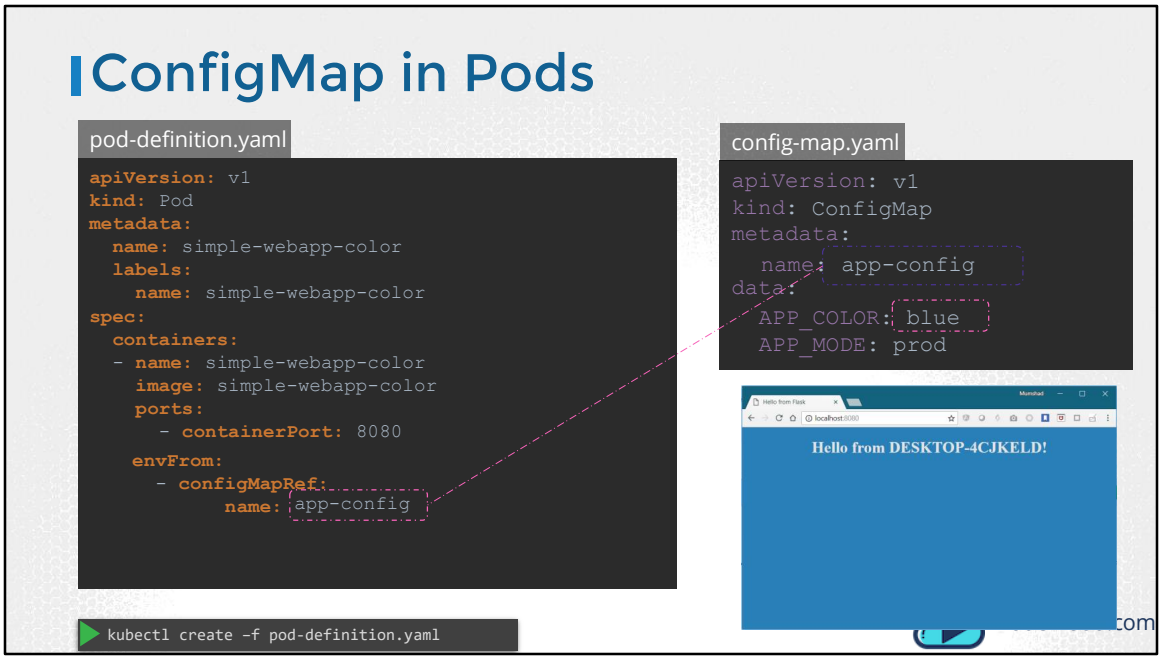
Run the kubectl create command and specify the configuration file name.



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.

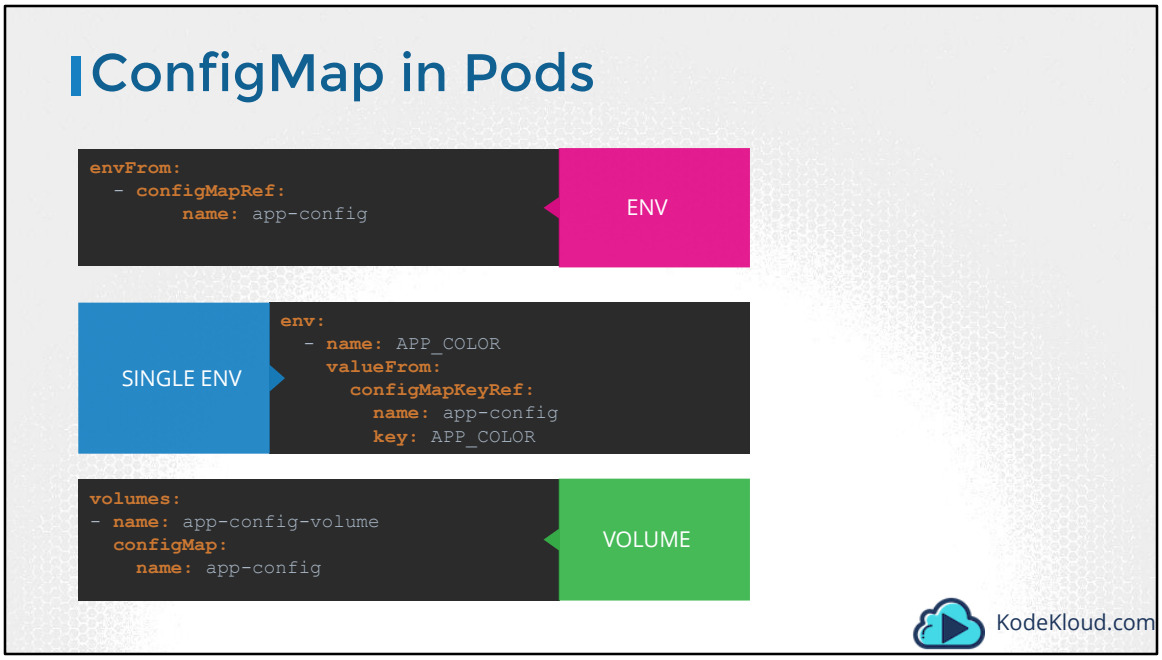


To view the configmaps, run the kubectl get configmaps command. This lists the newly created configmap named app-config. The describe configmaps command lists the configuration data as well under the Data section.



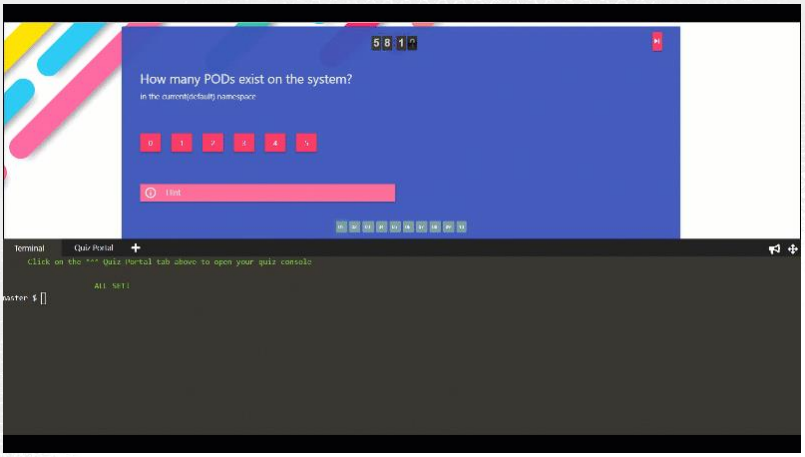
Now that we have the configmap created let us proceed with Step 2 – Configuring it with a POD. Here I have a simple pod definition file that runs my application simple web application.

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.



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

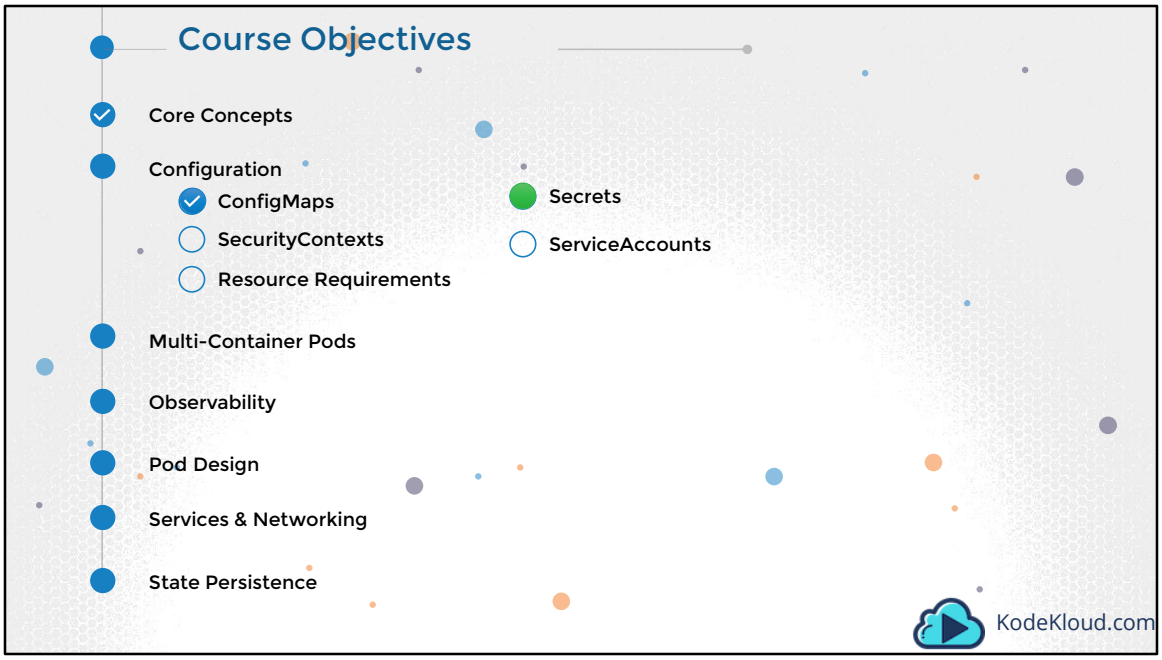
Access Test Here: https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743656

References

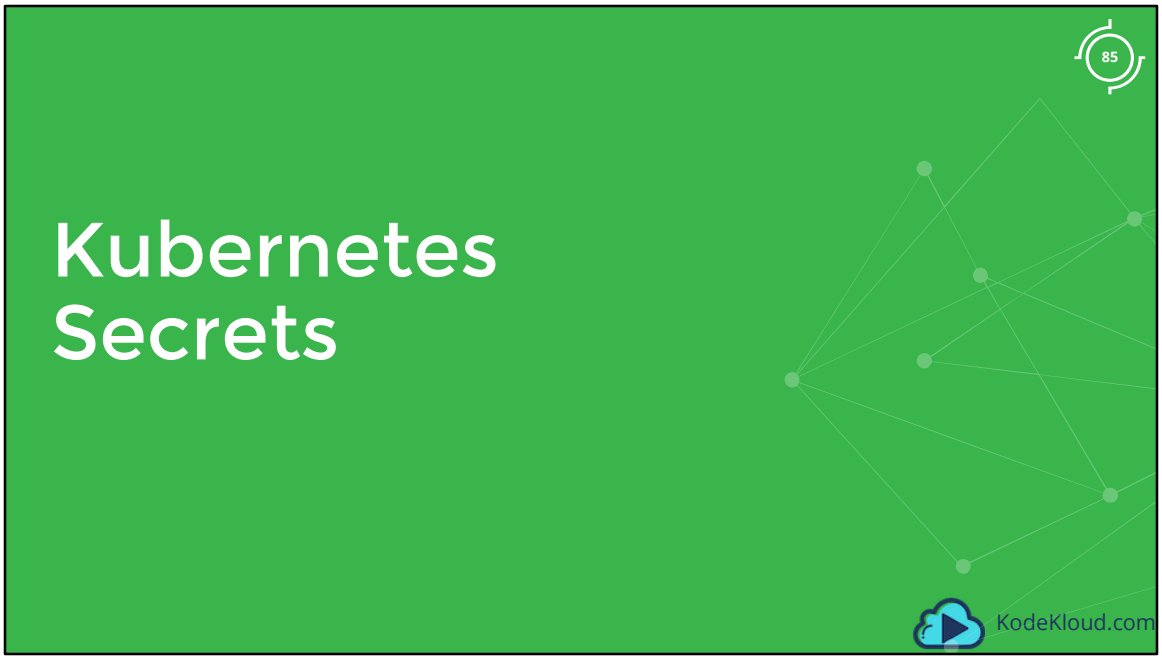
https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/

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

KodeKloud.com



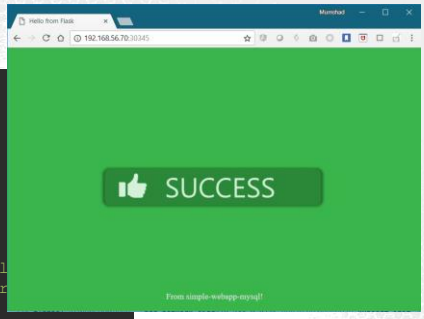
The next section on Configuration covers topics like ConfigMaps, SecurityContexts, Resource Requirements, secrets and service accounts.


85
Kubernetes
Secrets
KodeKloud.com

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='paswr

    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.

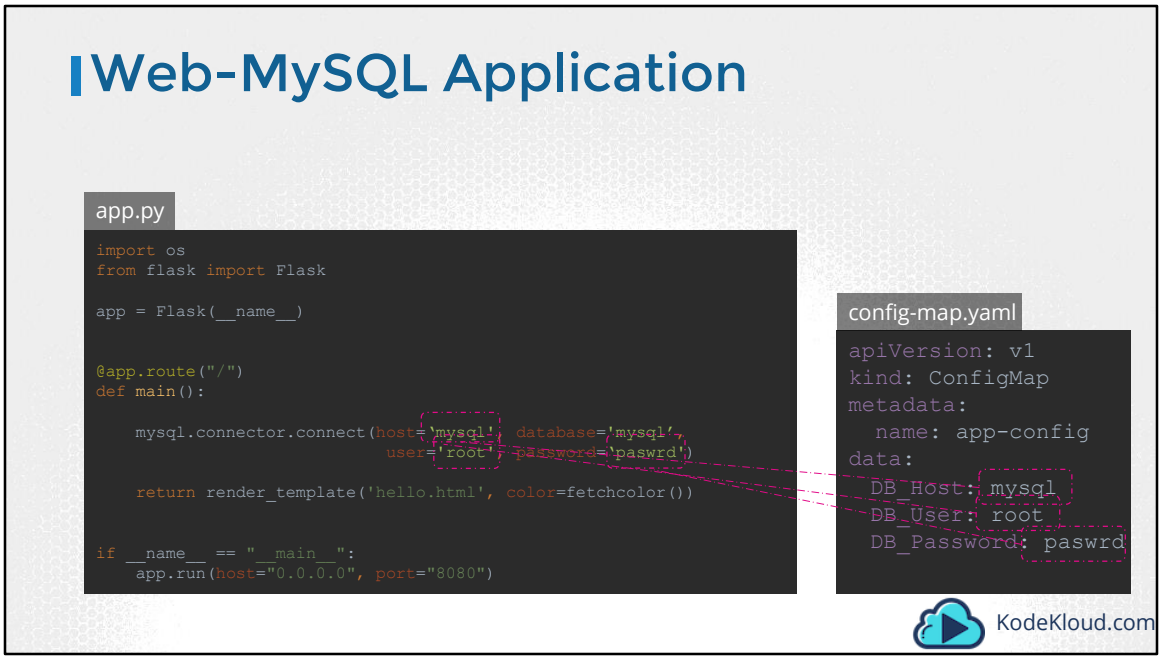
Web-MySQL Application

app.py

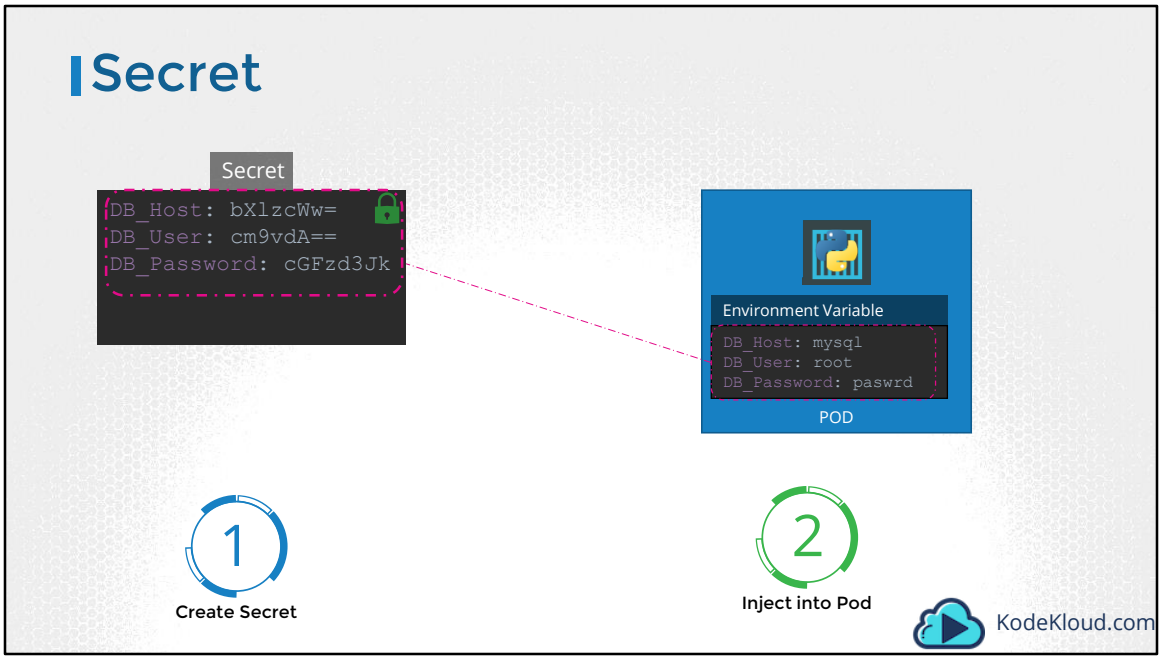
```
import os
from flask import Flask

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

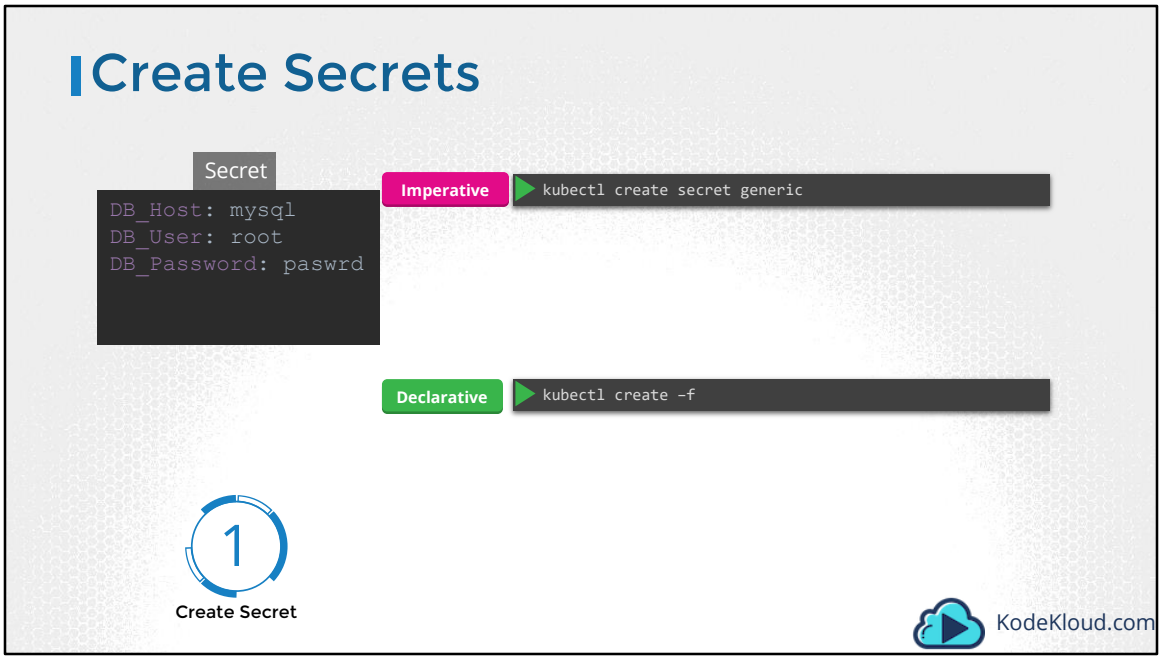
If you look closely into the code, you will see the hostname, username and password hardcoded. This is of-course not a good idea.



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.

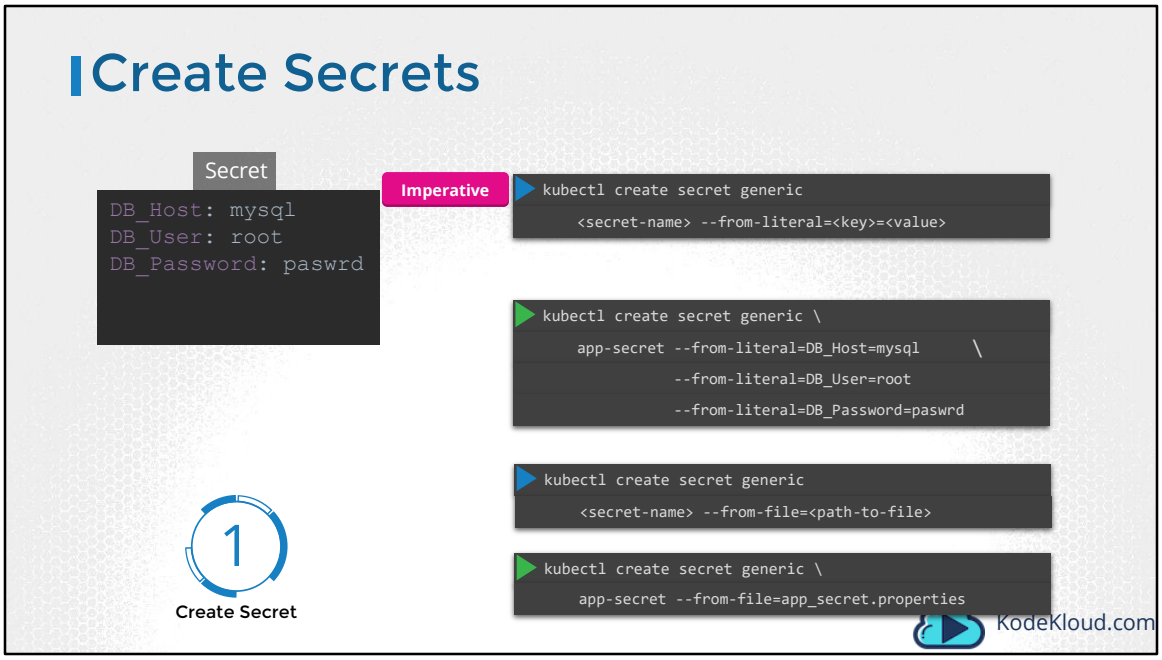


This is were secrets come in. Secrets are used to store sensitive information, like passwords or keys. They are similar to configMaps, except that they are stored in an encoded or hashed format. As with configMaps, there are two steps involved in working with Secrets. First, create the secret and second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



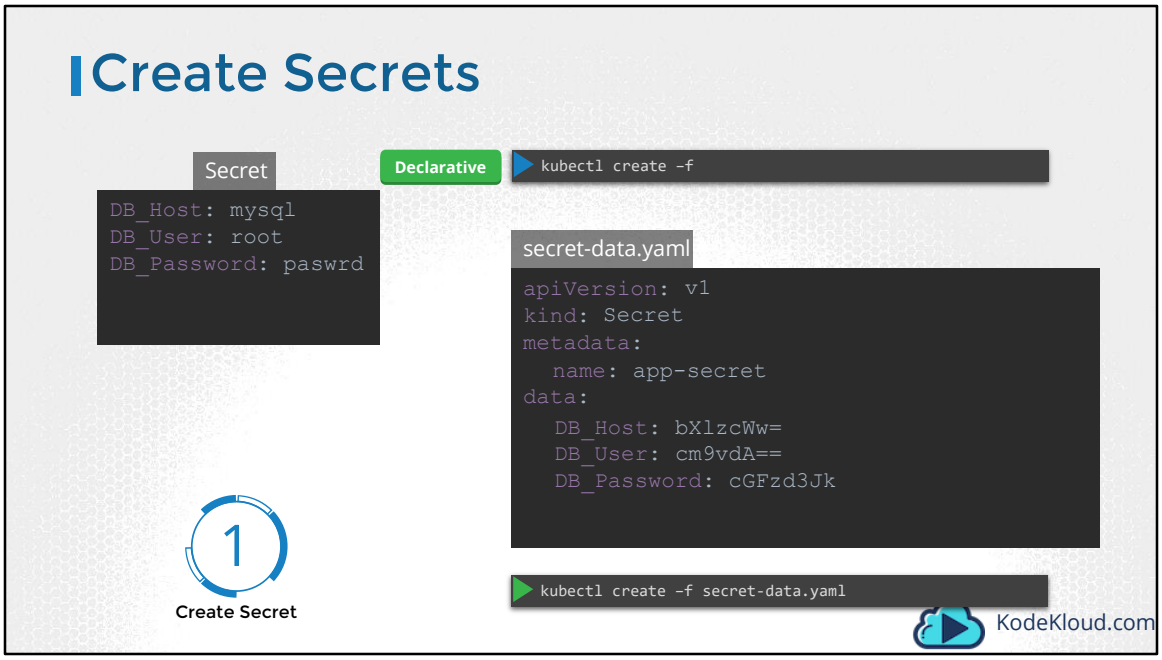
There are two ways of creating a secret. The imperative way - without using a Secret definition file and the Declarative way by using a Secret Definition file.

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.



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.

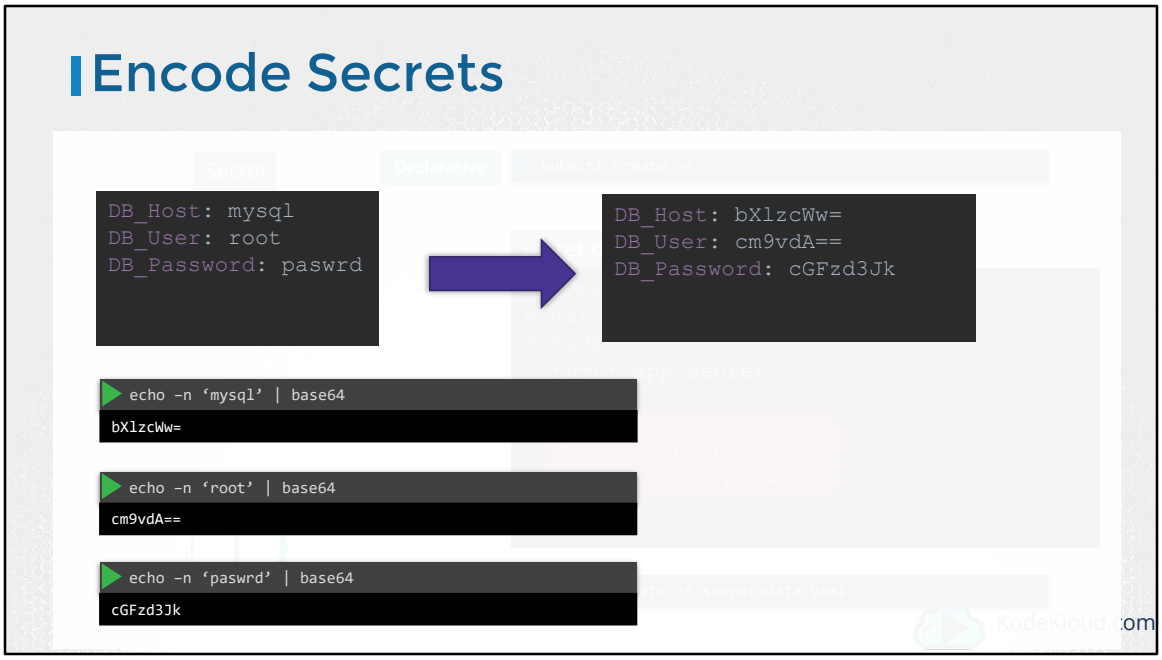
However, this could get complicated when you have too many secrets to pass in. Another way to input the secret data is through a file. Use the –from-file option to specify a path to the file that contains the required data. The data from this file is read and stored under the name of the file.



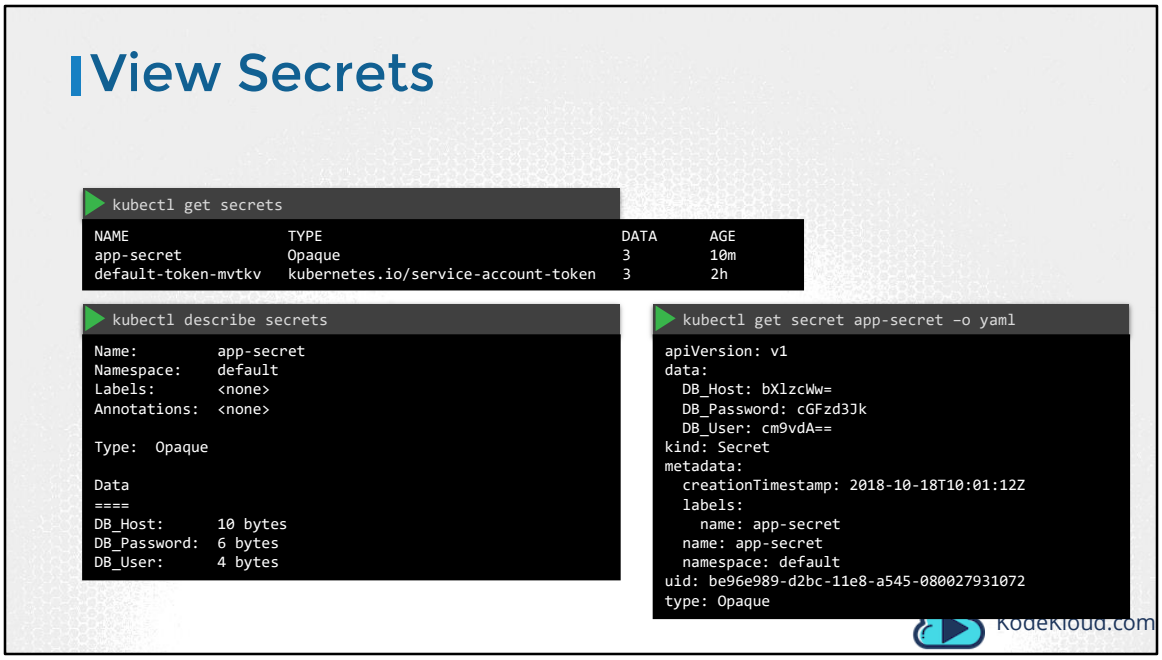
Let us now look at the declarative approach. For this we create a definition file, just like how we did for the ConfigMap. The file has apiVersion, kind, metadata and data. The apiVersion is v1, kind is Secret. Under metadata specify the name of the secret. We will call it app-secret. Under data add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



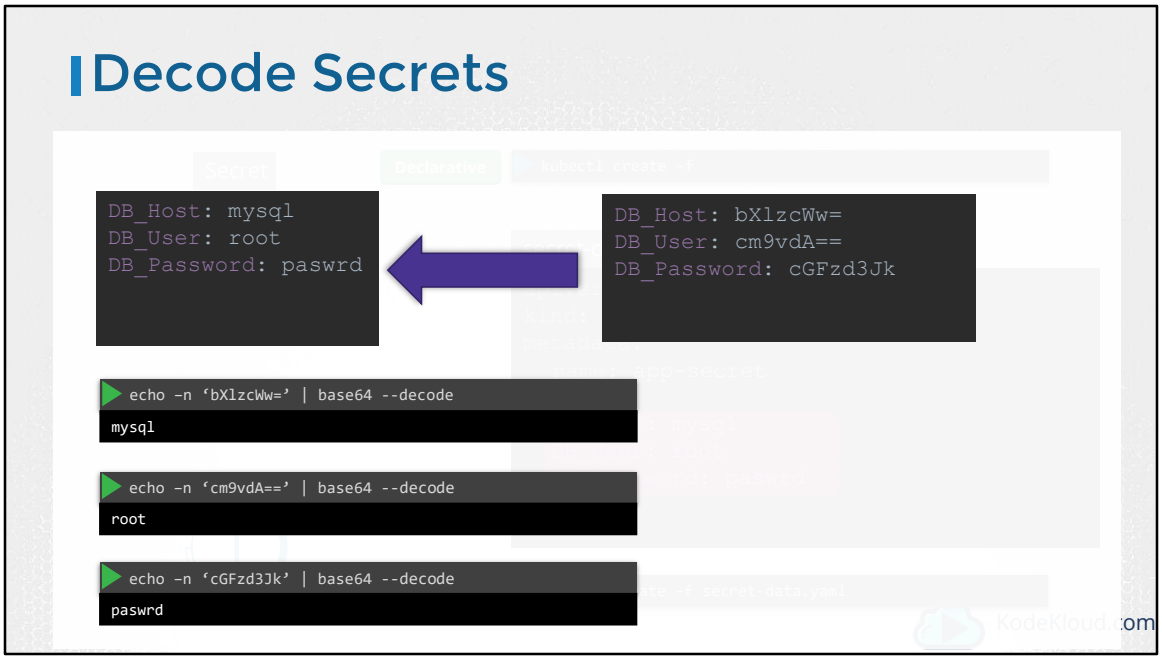
But how do you convert the data from plain text to an encoded format? On a linux host run the command echo –n followed by the text you are trying to convert, which is mysql in this case and pipe that to the base64 utility.



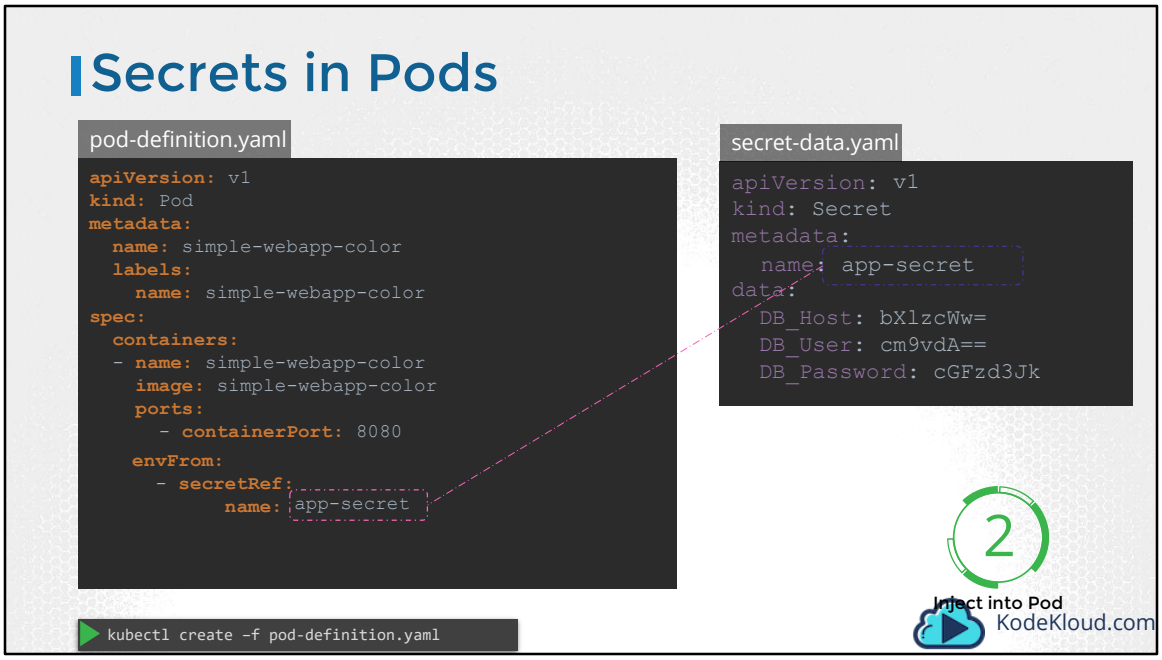
To view secrets run the kubectl get secrets command. This lists the newly created secret along with another secret previously created by kubernetes for its internal purposes.

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.

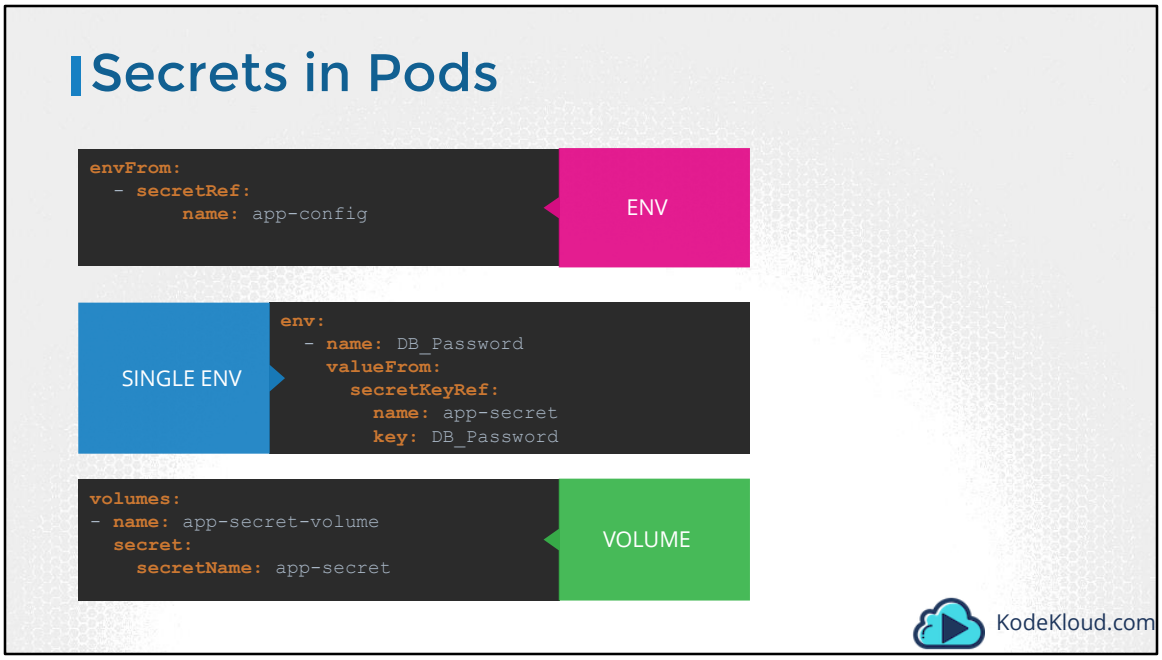


How do you decode the hashed values? Use the same base64 command you used in linux to encode it, but this time add a decode option to it.

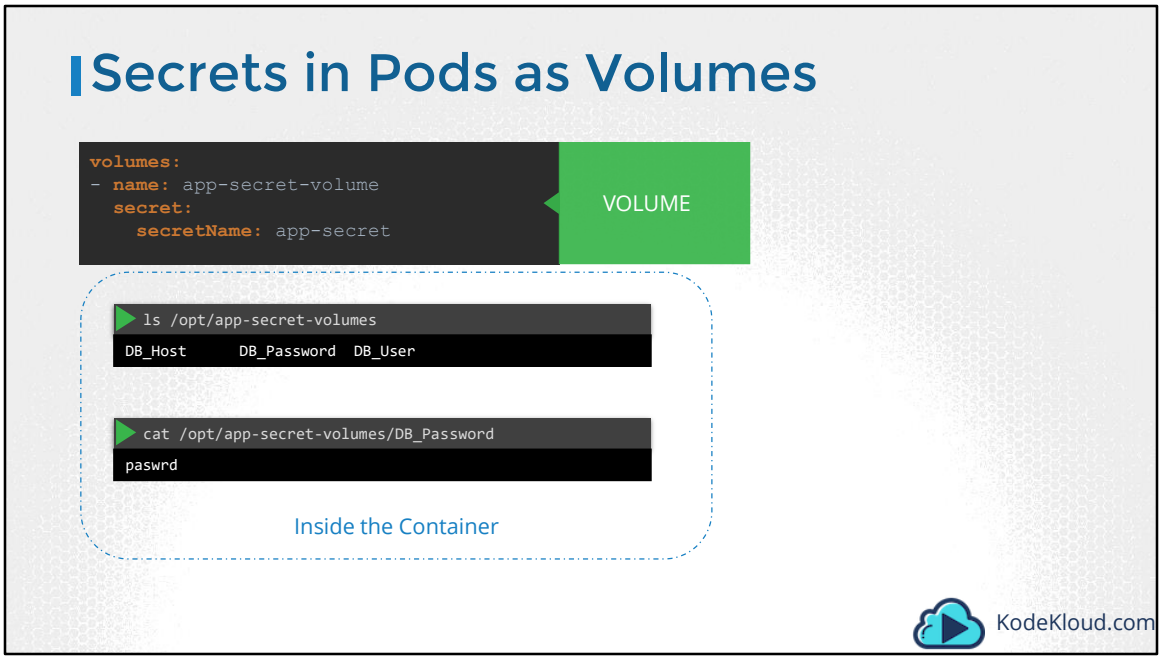


Now that we have the secret created let us proceed with Step 2 – Configuring it with a POD. Here I have a simple pod definition file that runs my application.

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.

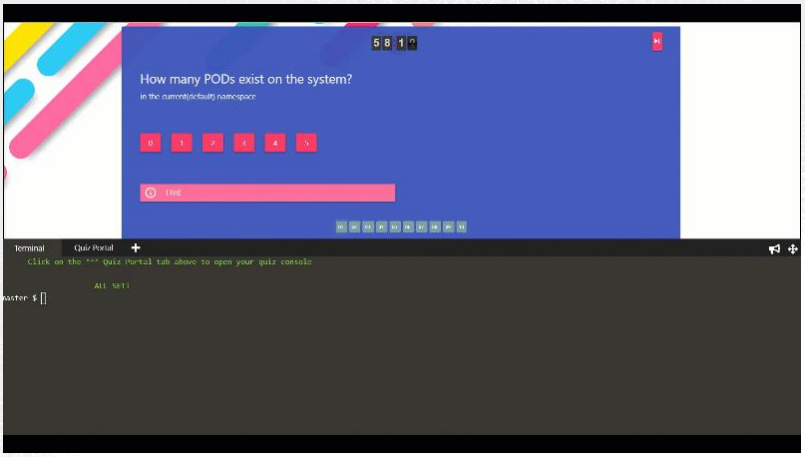


What we just saw was injecting secrets as environment variables into the PODs. There are other ways to inject secret into PODs. You can inject as single environment variables or inject the whole secret as files in a volume.



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

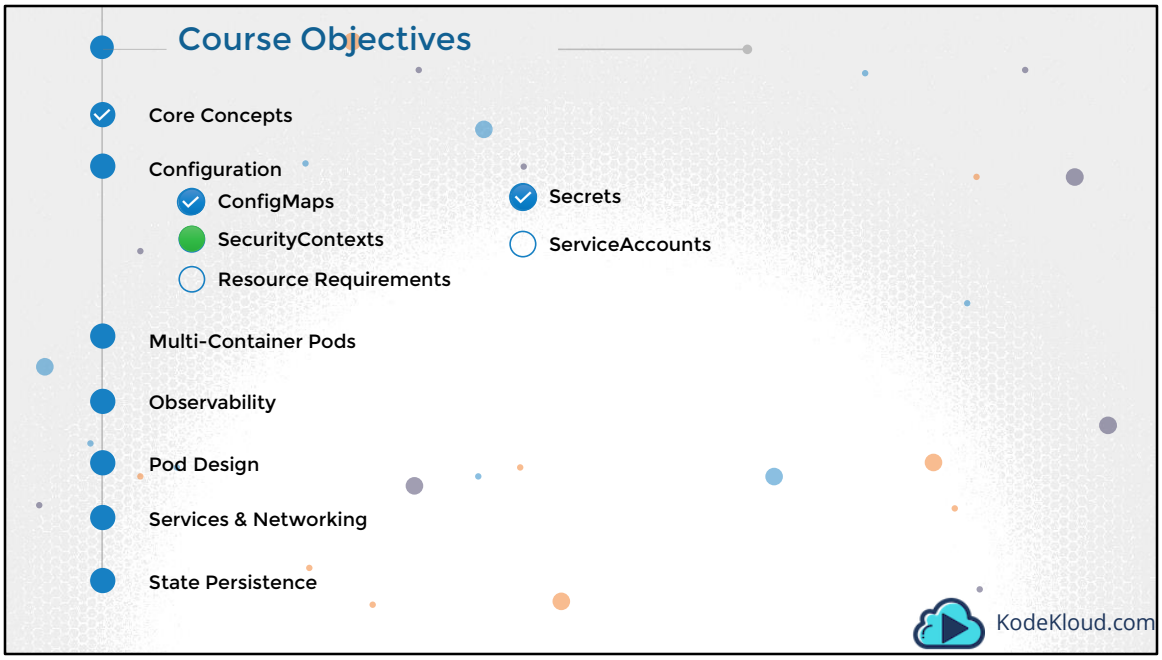
Access Test Here: https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743657

References

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

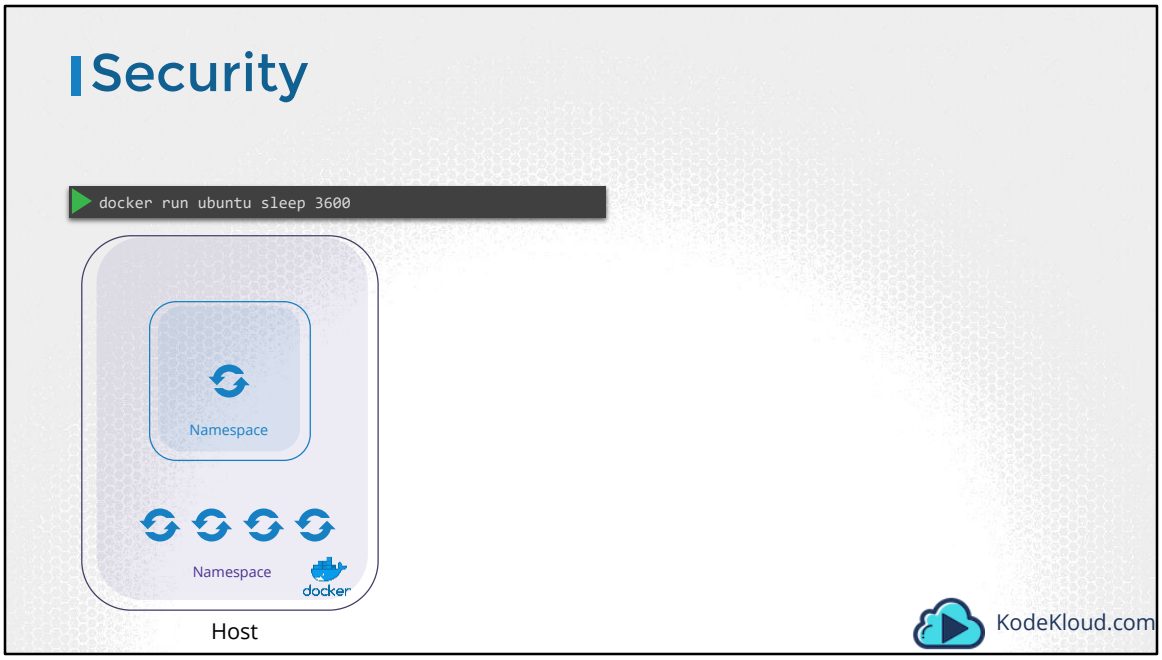
https://kubernetes.io/docs/tasks/inject-data-application/distribute-credentials-secure/

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.

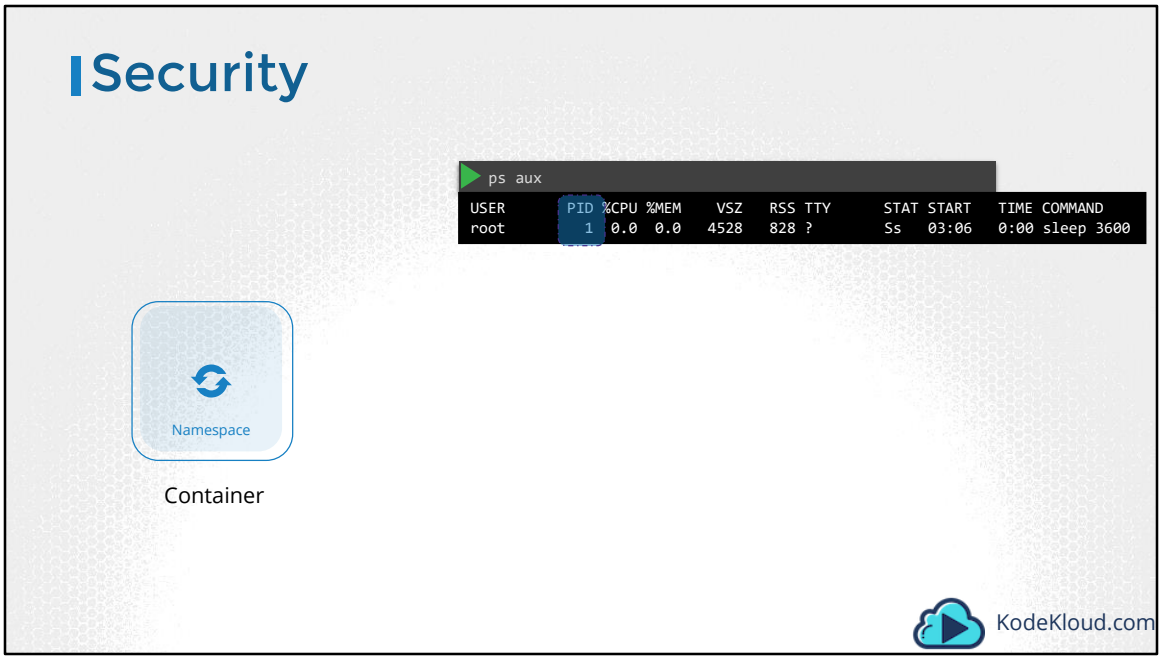
Hello and welcome to this lecture. In this lecture we will talk about Security Contexts in Kubernetes. But before we get into that, it is important to have some knowledge about Security in Docker. If you are familiar with Security in Docker, feel free to skip this lecture and head over to the next.



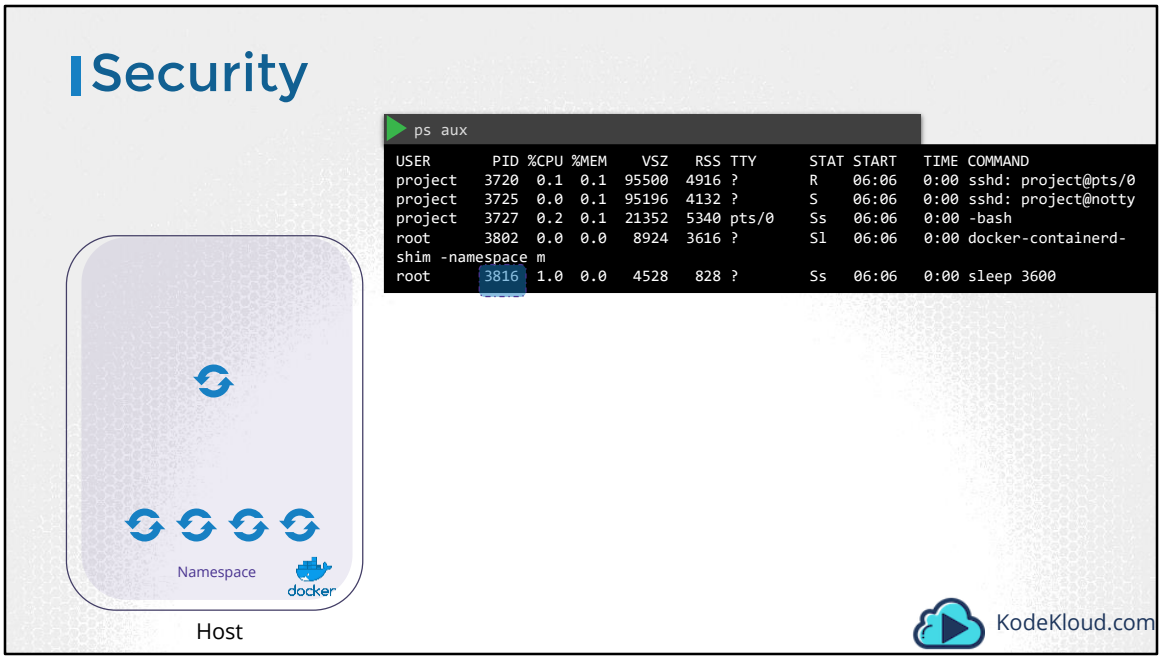
In this lecture we will look at the various concepts related to security in Docker. Let us start with a host with Docker installed on it. This host has a set of its own processes running such as a number of operating system processes, the docker daemon itself, the SSH server etc.

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

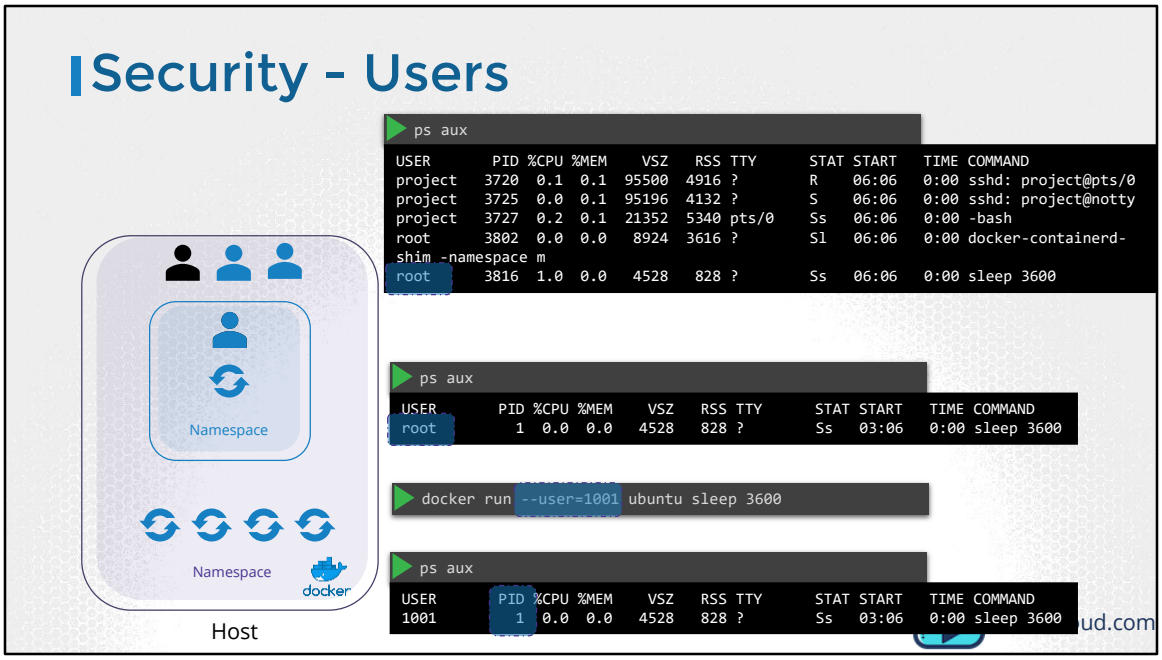
We have learned that unlike virtual machines containers are not completely isolated from their host. Containers and the hosts share the same kernel. Containers are isolated using namespaces in Linux. The host has a namespace and the containers have their own namespace. All the processes run by the containers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



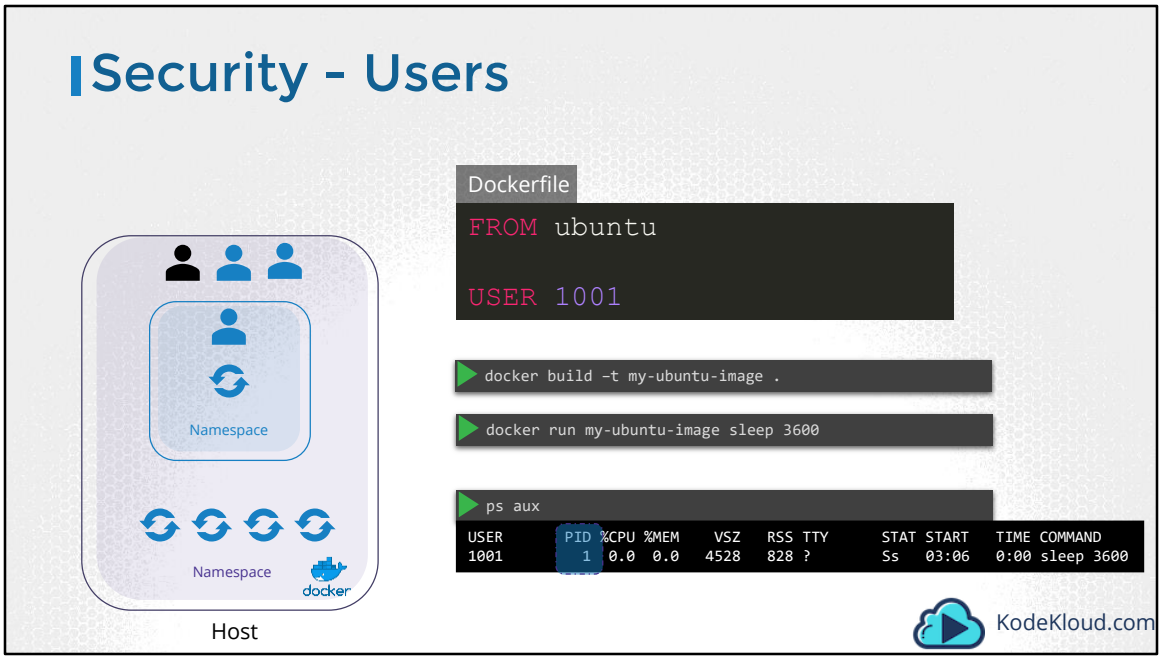
As far as the docker container is concerned, it is in its own namespace and it can see its own processes only, cannot see anything outside of it or in any other namespace. So when you list the processes from within the docker container you see the sleep process with a process ID of 1.



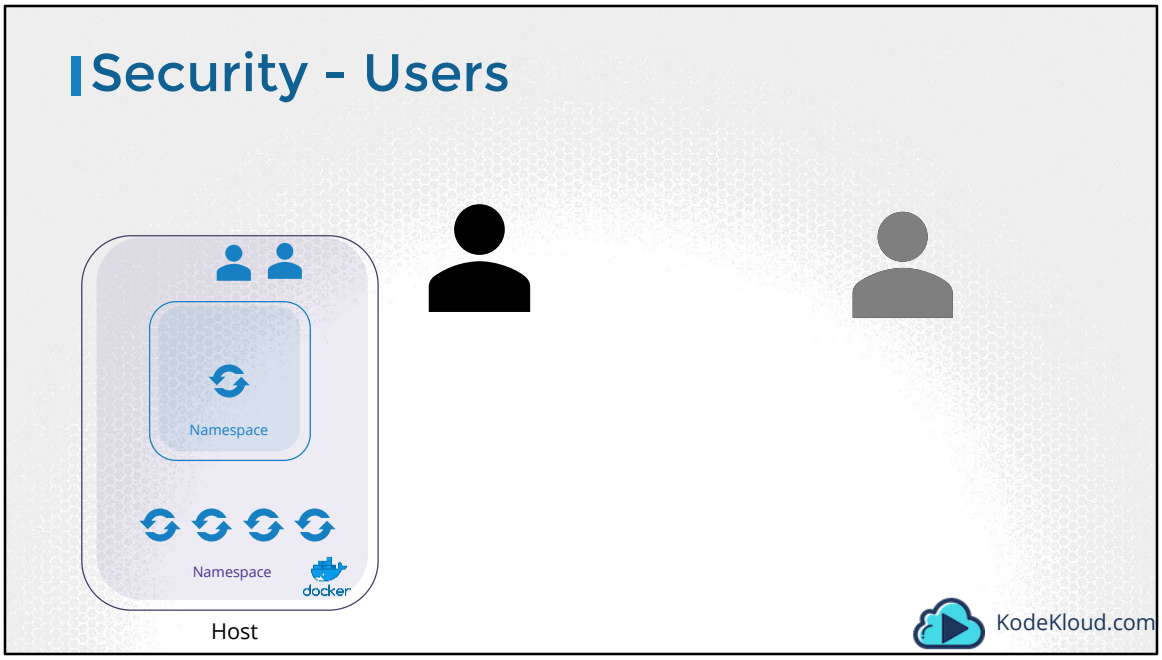
For the docker host, all processes of its own as well as those in the child namespaces are visible as just another process in the system. So wen you list the processes on the host you see a list of processes including the sleep command, but with a different process ID. This is because the processes can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



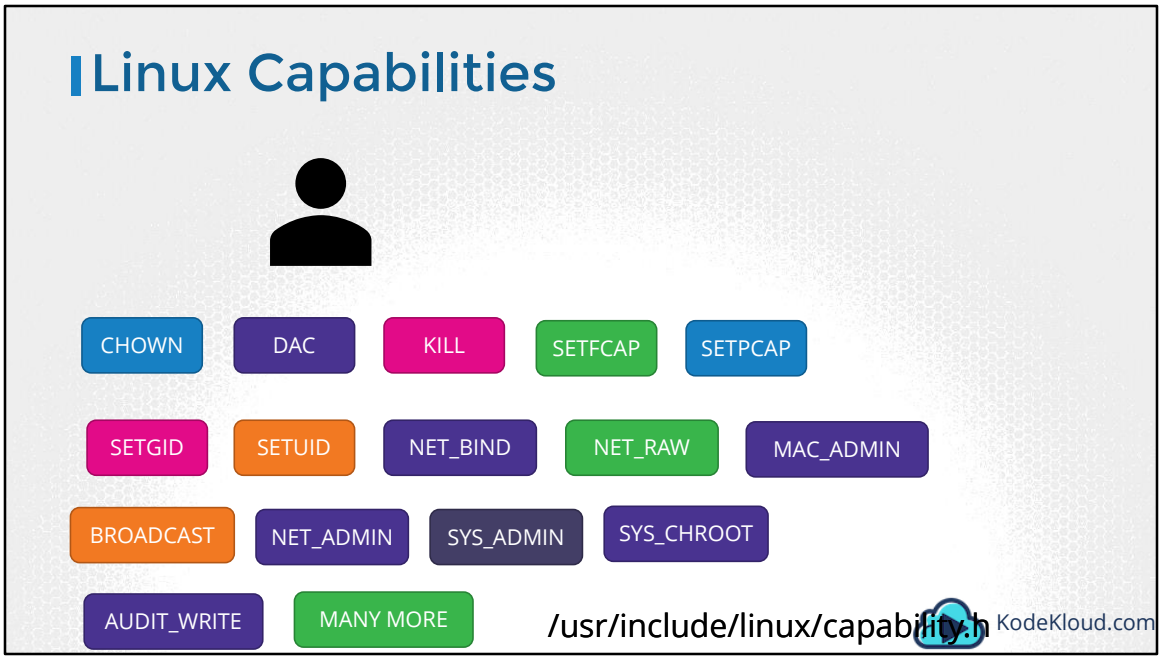
Let us now look at users in context of security. The docker host has a set of users, a root user as well as a number of non-root users. By default docker runs processes within containers as the root user. This can be seen in the output of the commands we ran earlier. Both within the container and outside 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.



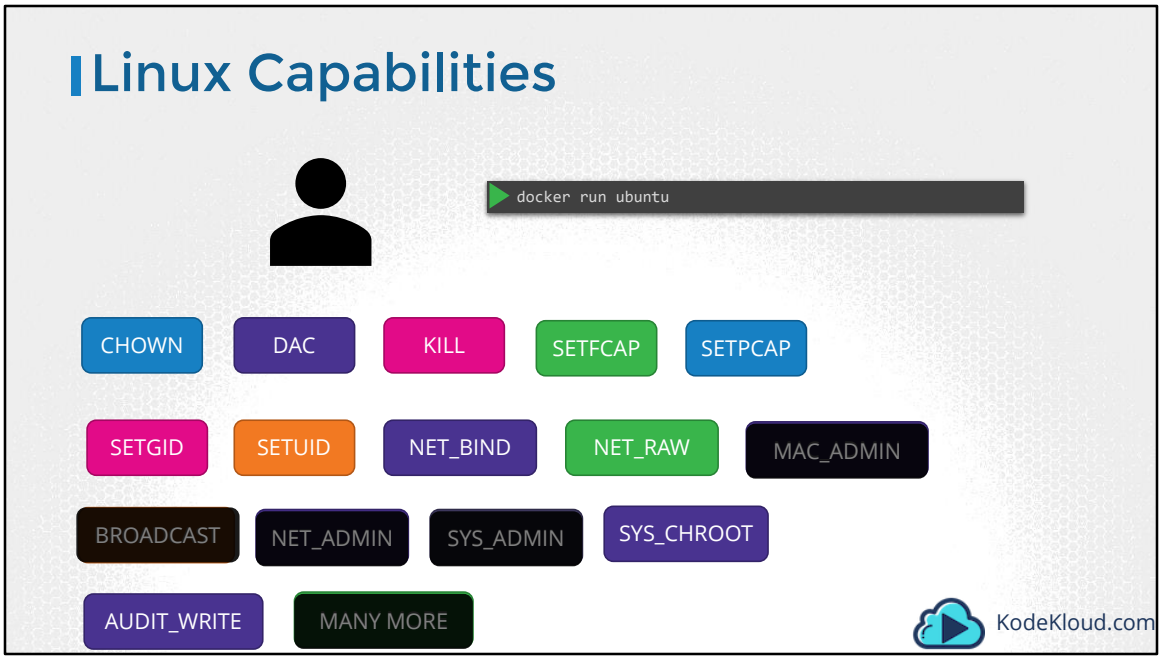
Another way to enforce user security is to have this defined in the Docker image itself at the time of creation. For example, we will use the default ubuntu image and set the user ID to 1000 using the USER instruction. Then build the custom image. We can now run this image using without specifying the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



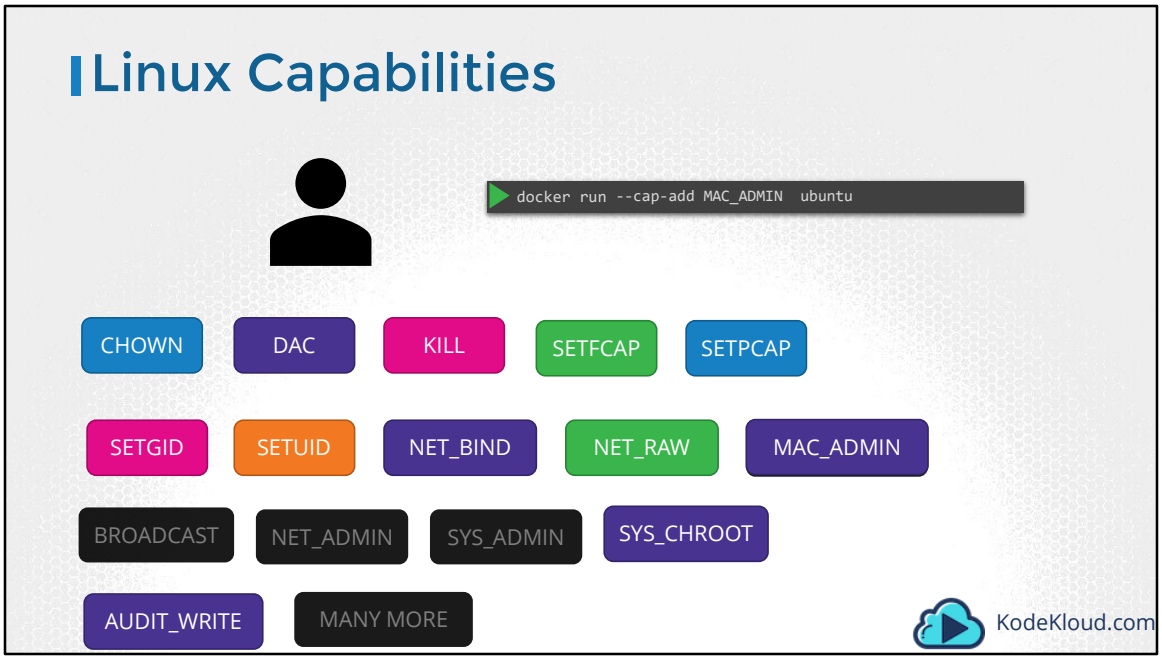
Let us take a step back. What happens when you run containers as the root user? Is the root user within the container the same as the root user on the host? Can the process inside the container do anything that the root user can do on the system? If so isn't that dangerous? Well, docker implements a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



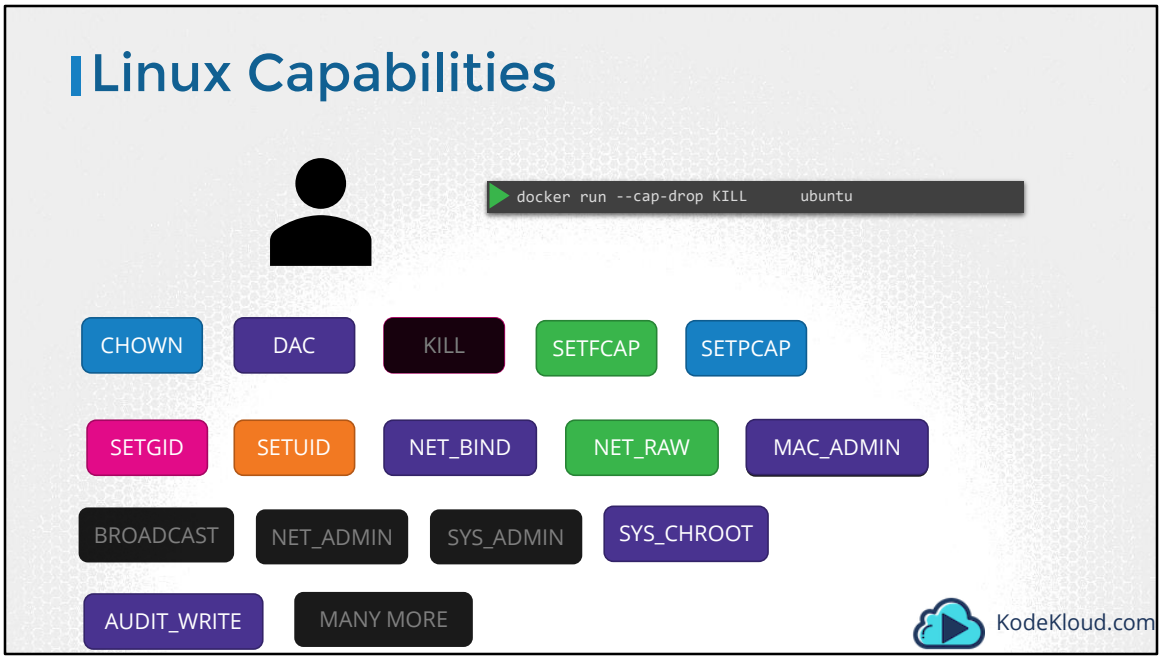
Docker uses Linux Capabilities to implement this. As we all know the root user is the most powerful user on a system. The root user can literally do anything. And so does a process run by the root user. It has unrestricted access to the system. From modifying files and permissions on files, Access Control, 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.



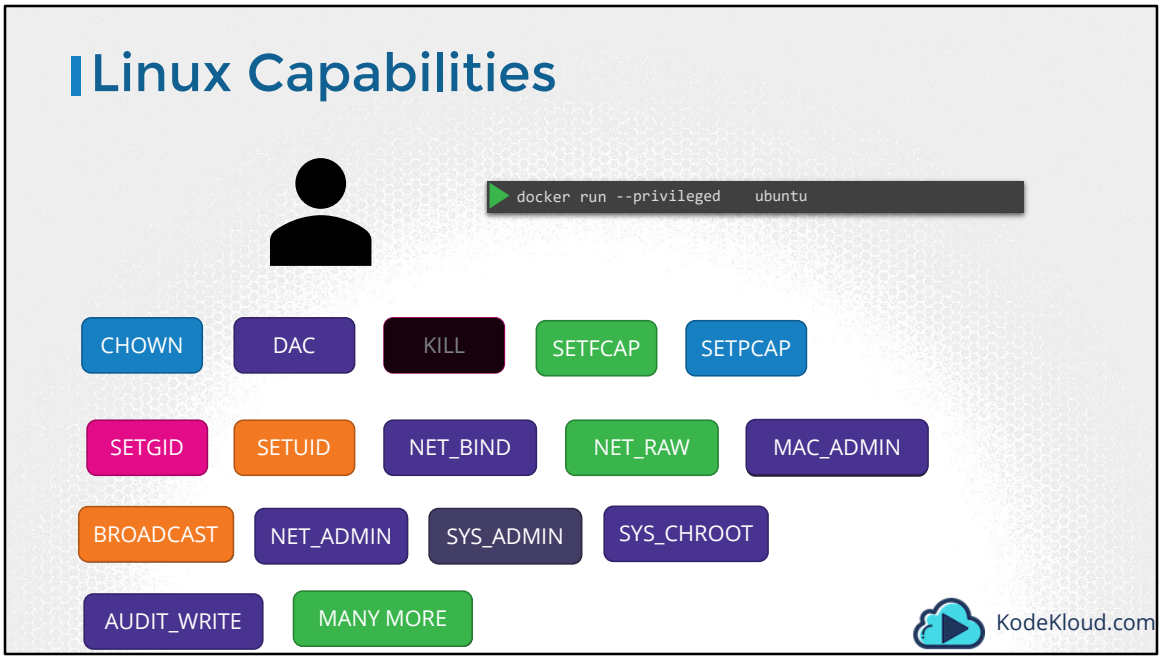
By default Docker runs a container with a limited set of capabilities. And so the processes running within the container do not have the privileges to say, reboot the host or perform operations that can disrupt the host or other containers running on the same host. In case you wish to override this behavior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



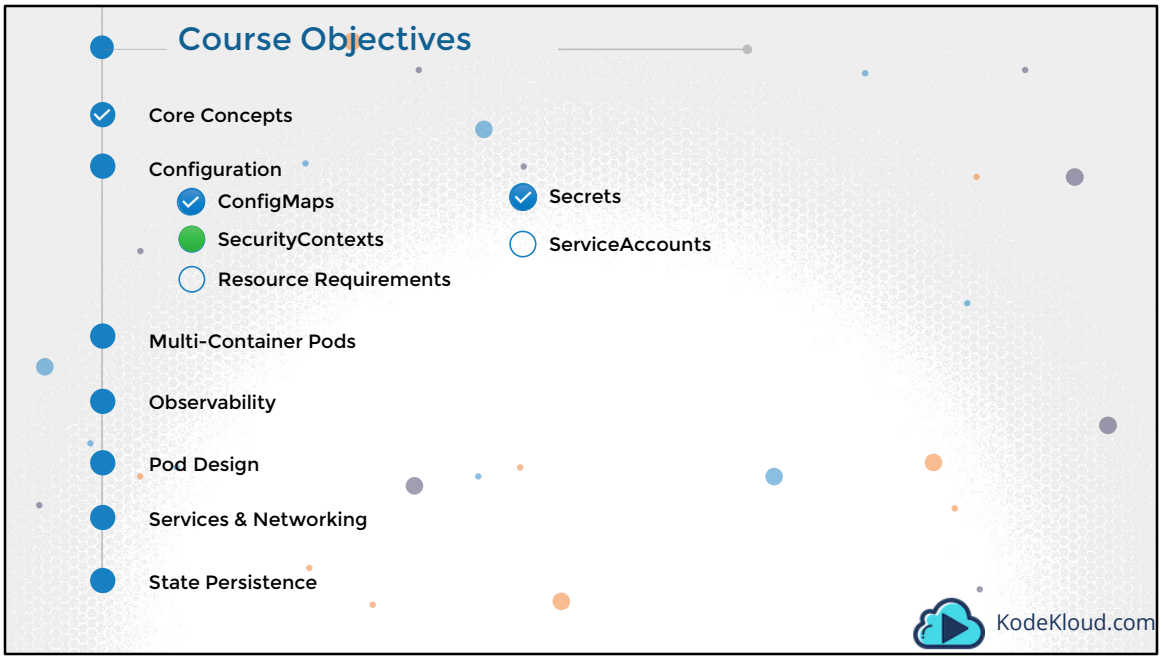
If you wish to override this behavior and provide additional privileges than what is available use the cap-add option in the docker run command.



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

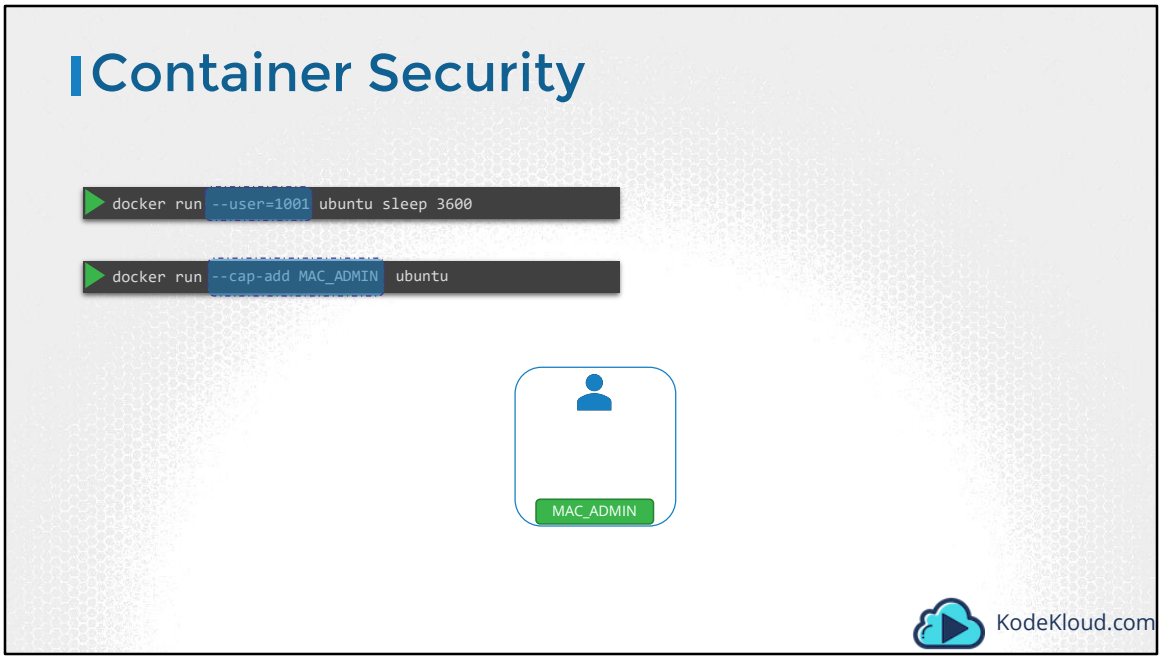


Or in case you wish to run the container with all privileges enabled, use the privileged flag. Well that's it on Docker Security for now. I will see you in the next lecture.

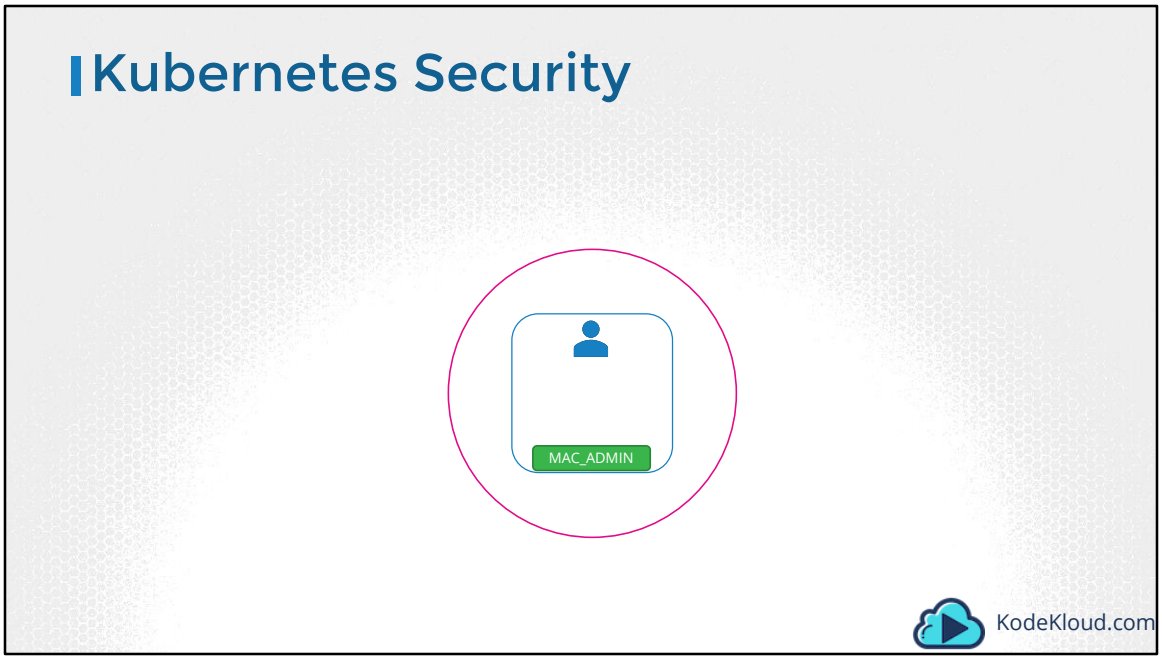


Hello and welcome to this lecture on Security Contexts in Kubernetes. My name is Mumshad Mannambeth and we are going through the Certified Kubernetes Applications Developer Course.

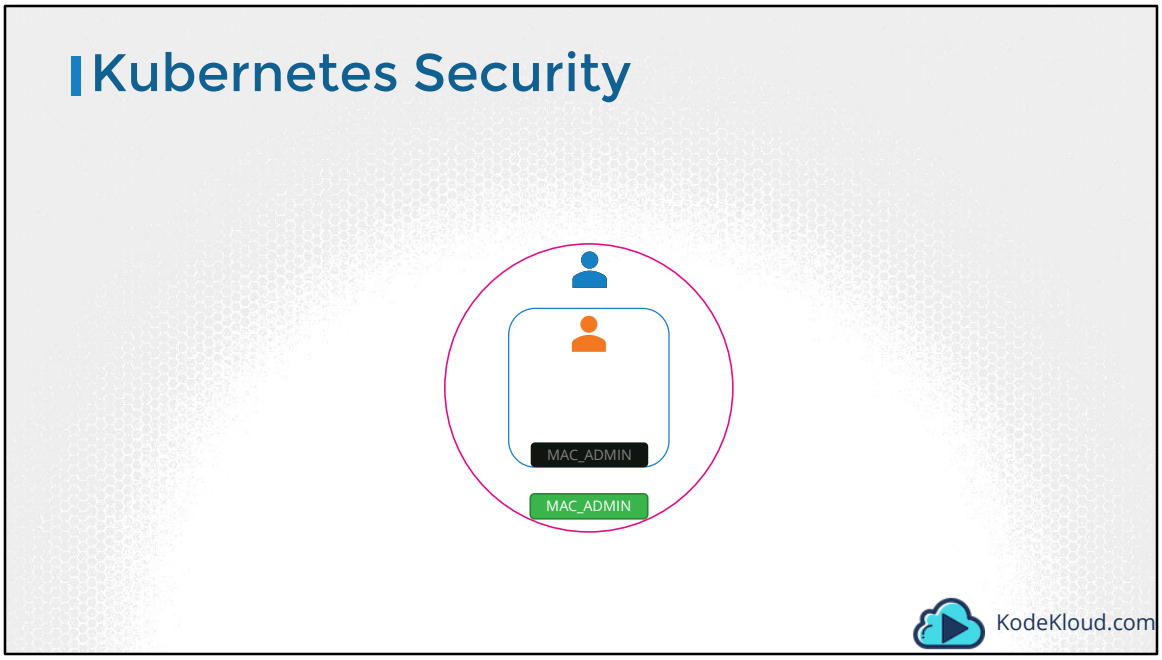

115
Kubernetes
Security
Contexts
KodeKloud.com



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.



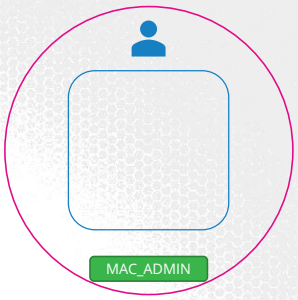
As you know already, in Kubernetes containers are encapsulated in PODs. You may chose to configure the security settings at a container level….



… 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.

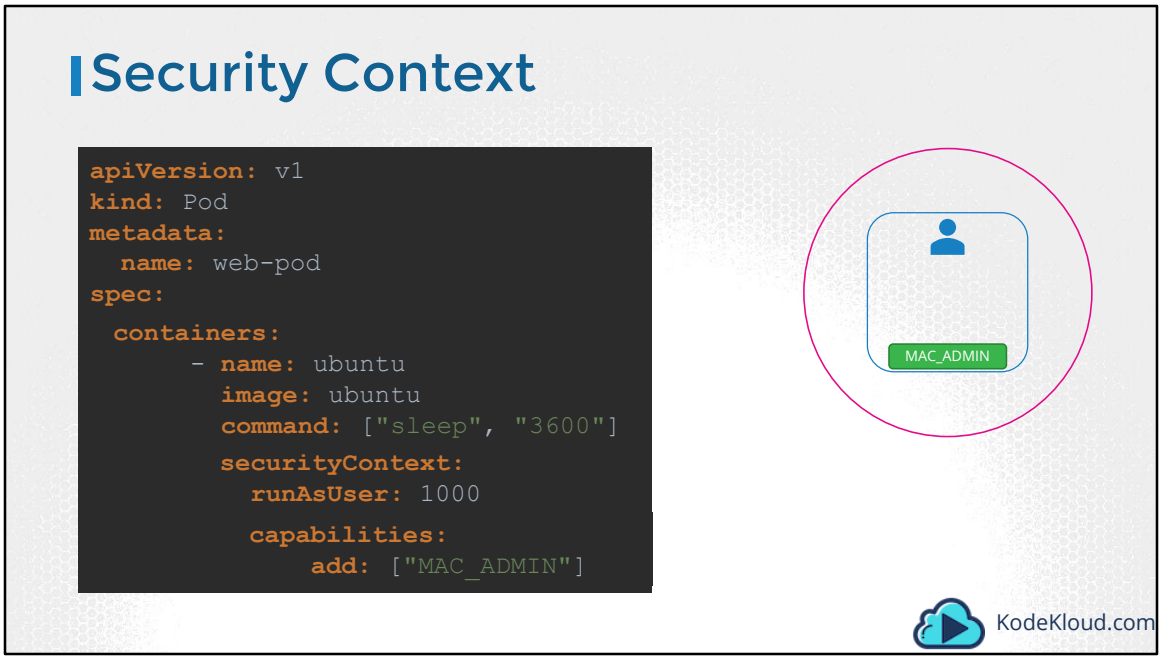
Security Context

```
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.

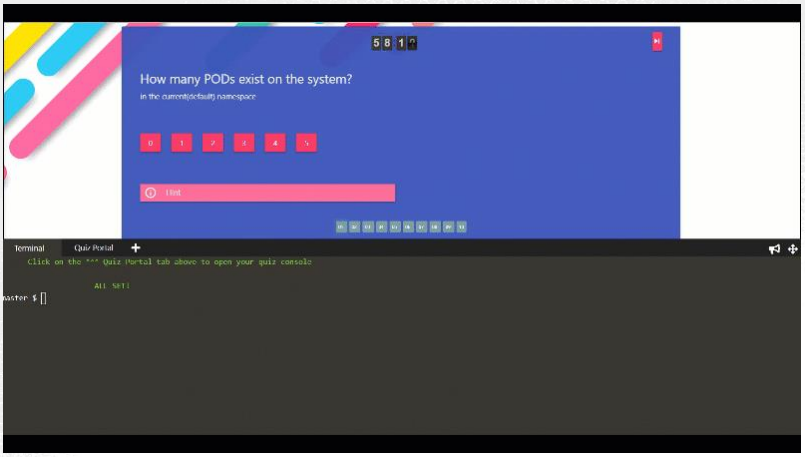


To set the same configuration on the container level, move the whole section under the container specification like this.

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

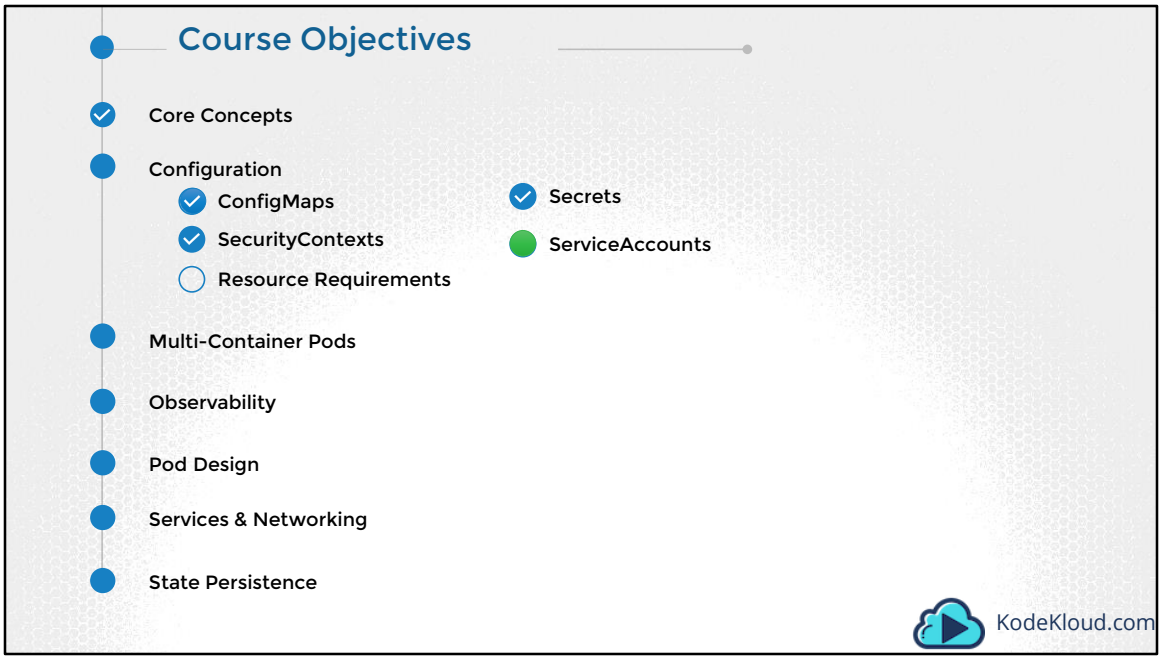
References

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

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

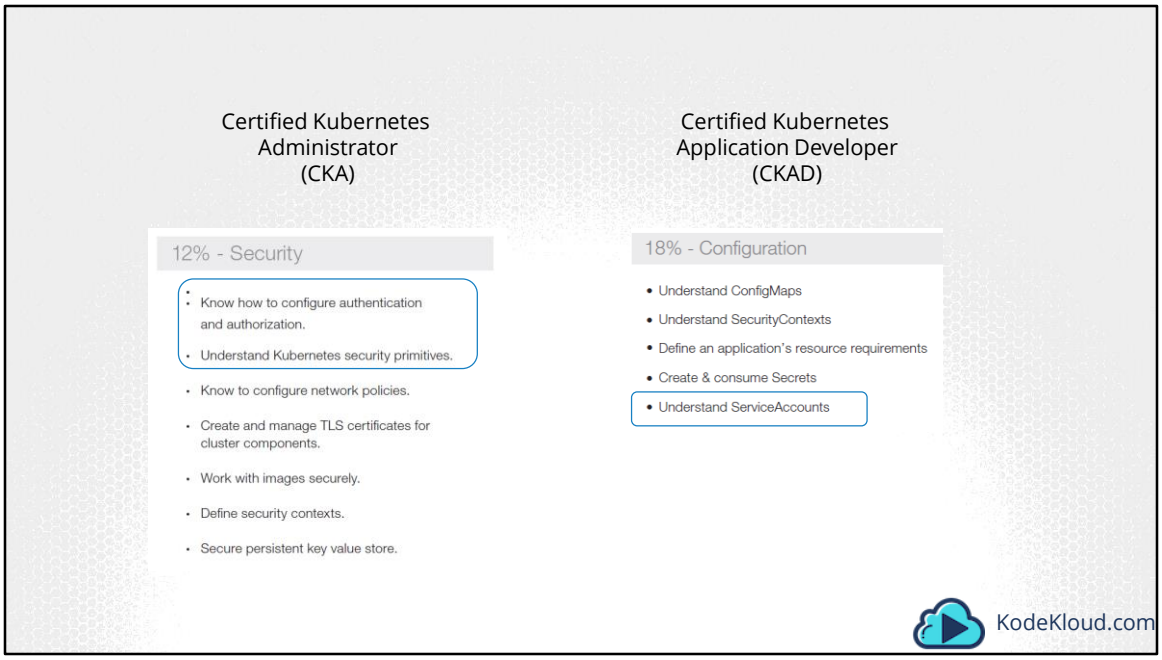
https://kubernetes.io/blog/2016/08/security-best-practices-kubernetes-deployment/

KodeKloud.com

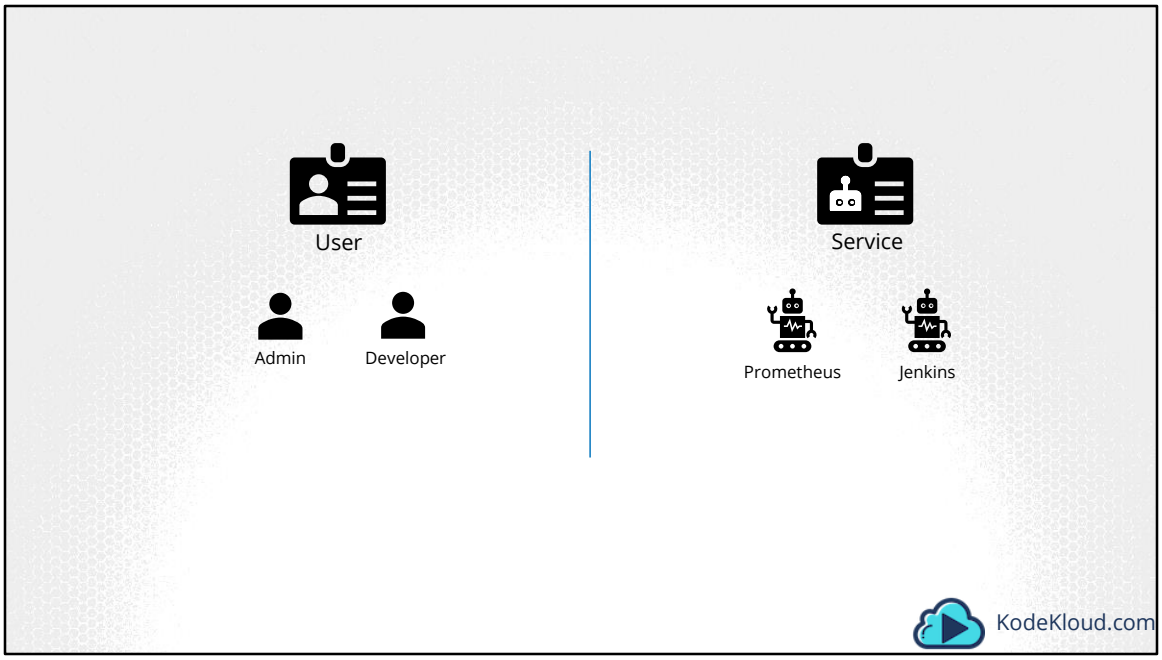


Hello and welcome to this lecture. In this lecture we will talk about Service Accounts in Kubernetes.

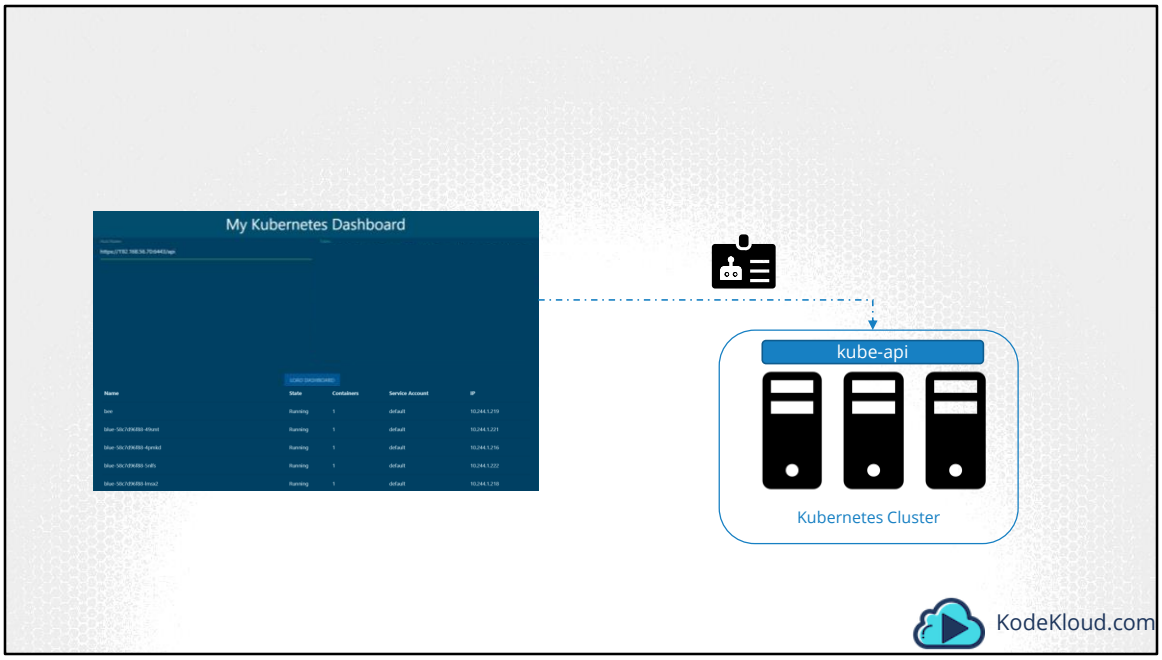
The concept of service accounts is linked to other security related concepts in kubernetes such as Authentication, Authorization, Role based access controls etc.



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.



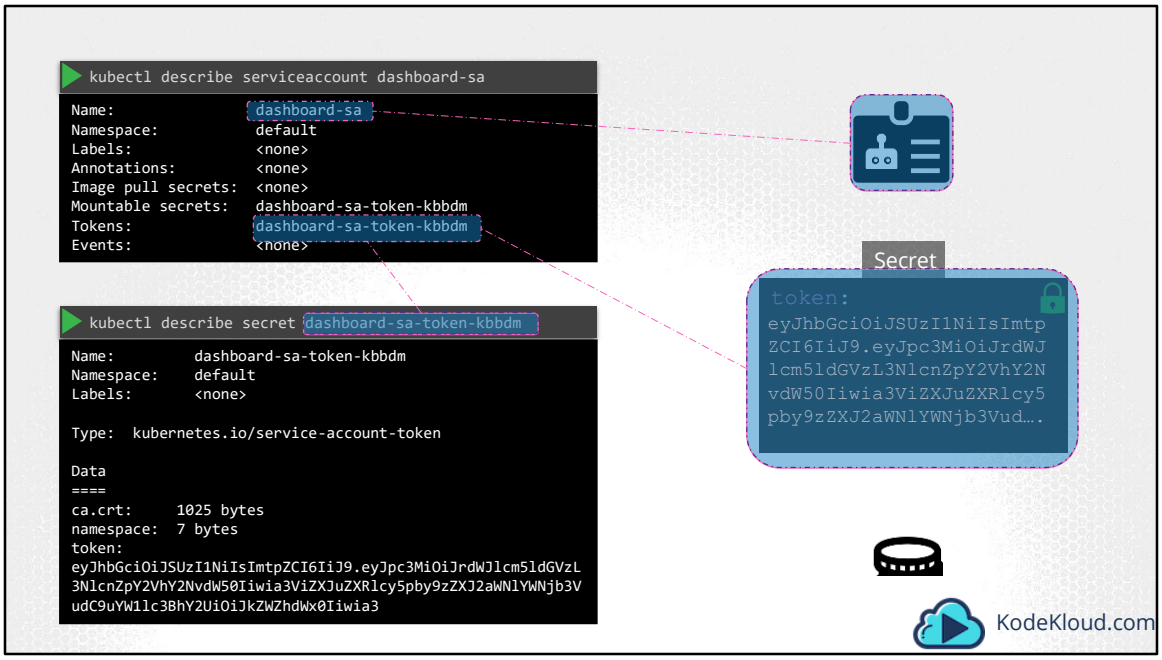
So there are two types of accounts in Kubernetes. A user account and a service account. As you might already know, the user account is used by humans. And service accounts are used by machines. A user account could be for an administrator accessing the cluster to perform administrative tasks, a developer accessing the cluster to deploy applications etc. A service account, could be an account used by an application to interact with the kubernetes cluster. For example a monitoring 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.



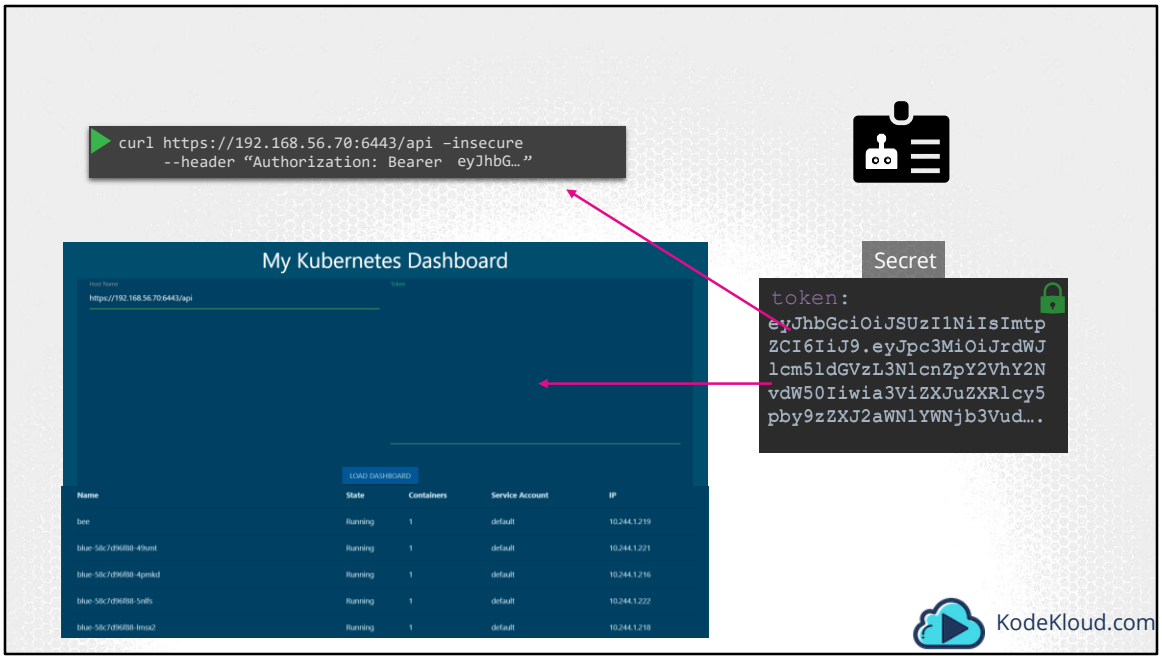
Let's take an example. I have built a simple kubernetes dashboard application named, my-kubernetes-dashboard. It's a simple application built in Python and all that it does when deployed is retrieve the list of PODs on a kubernetes cluster by sending a request to the kubernetes API and display it on 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.

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.

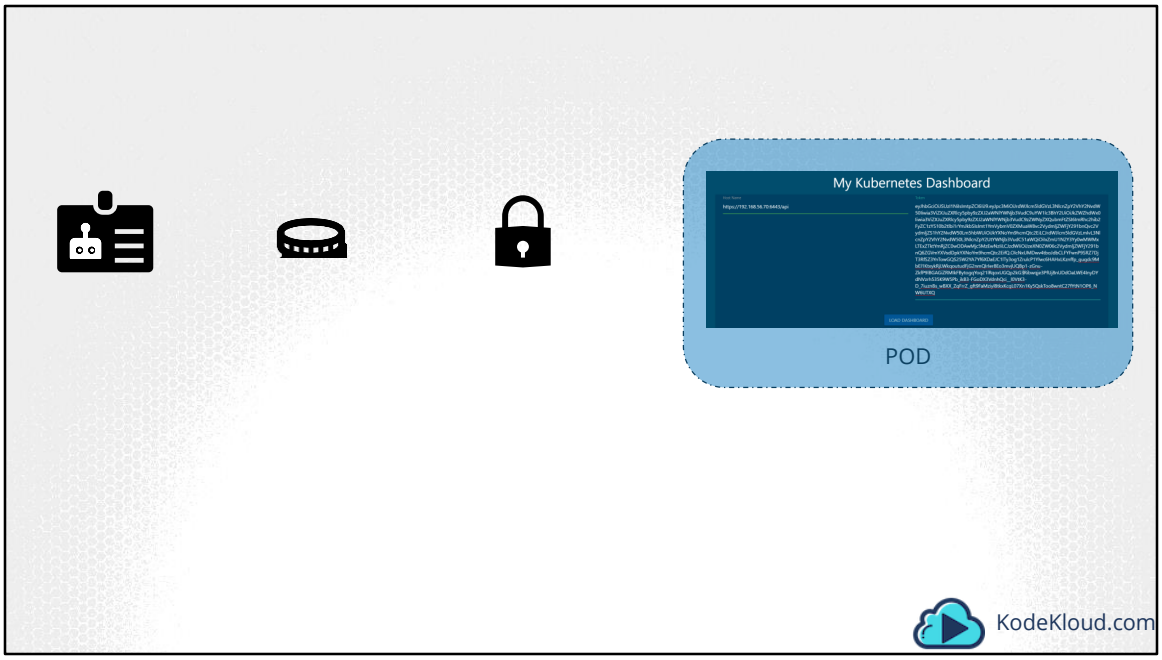


So when a service account is created, it first creates the service account object and then generates a token for the service account. It then creates a secret object and stores that token inside the secret object. The secret object is then linked to the service account. To view the token, view the secret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

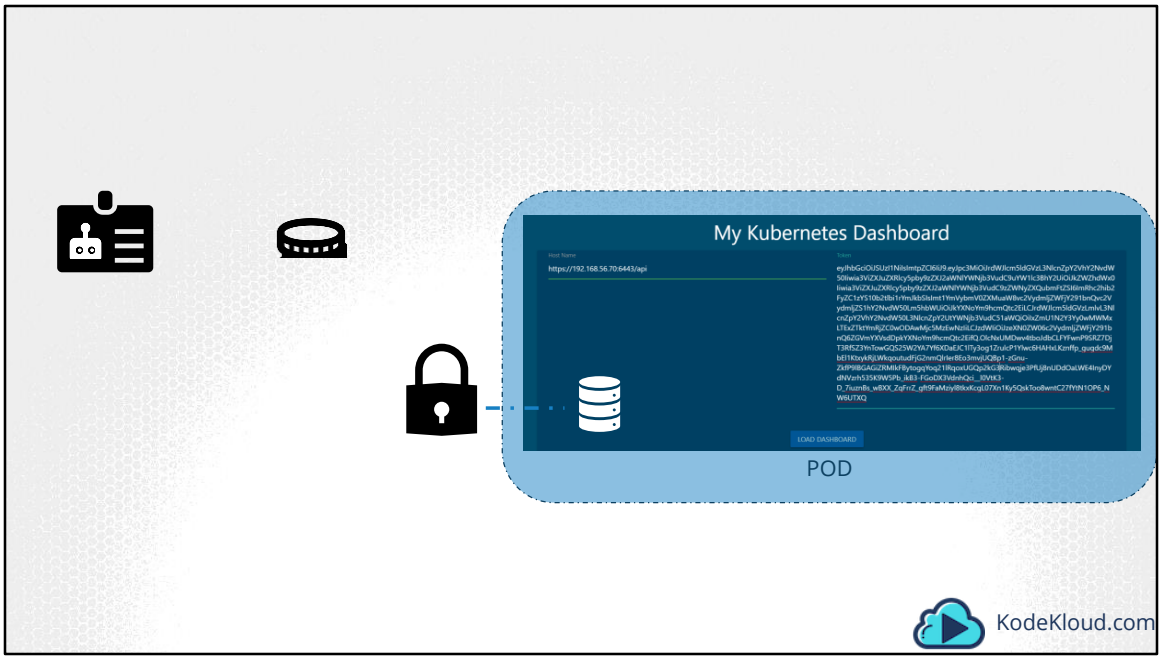


This token can then be used as an authentication bearer token while making a rest call to the kubernetes API. For example in this simple example using curl you could provide the bearer token as an Authorization header while making a rest call to the kubernetes API.

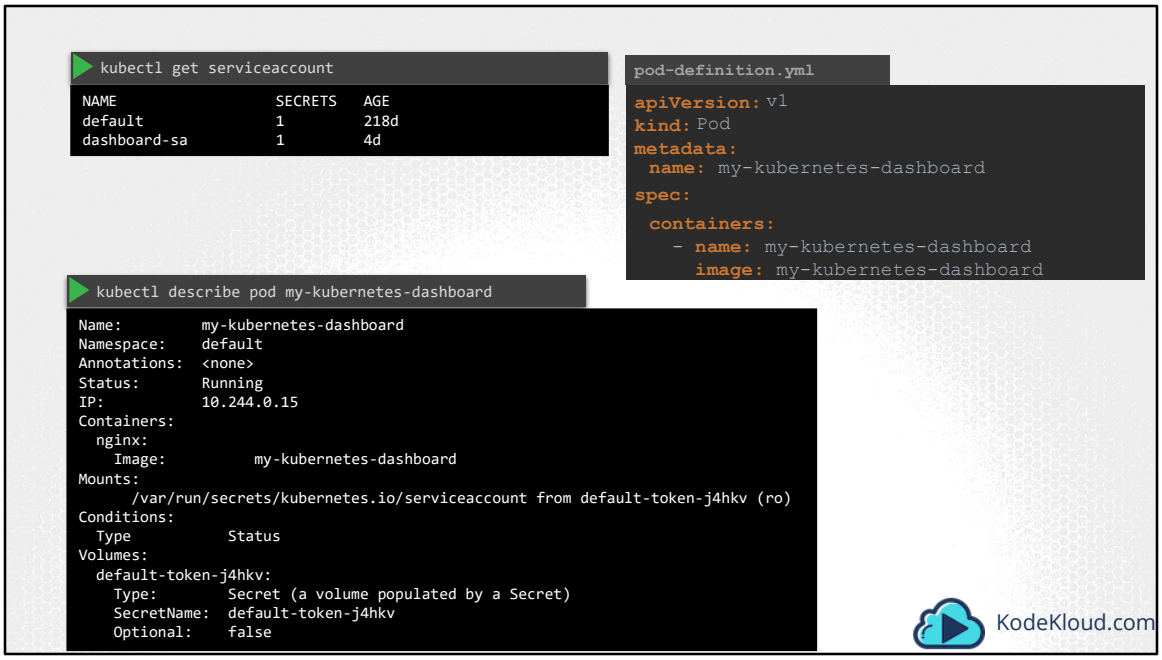
In case of my custom dashboard application, copy and paste the token into the tokens field to authenticate the dashboard application.



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.

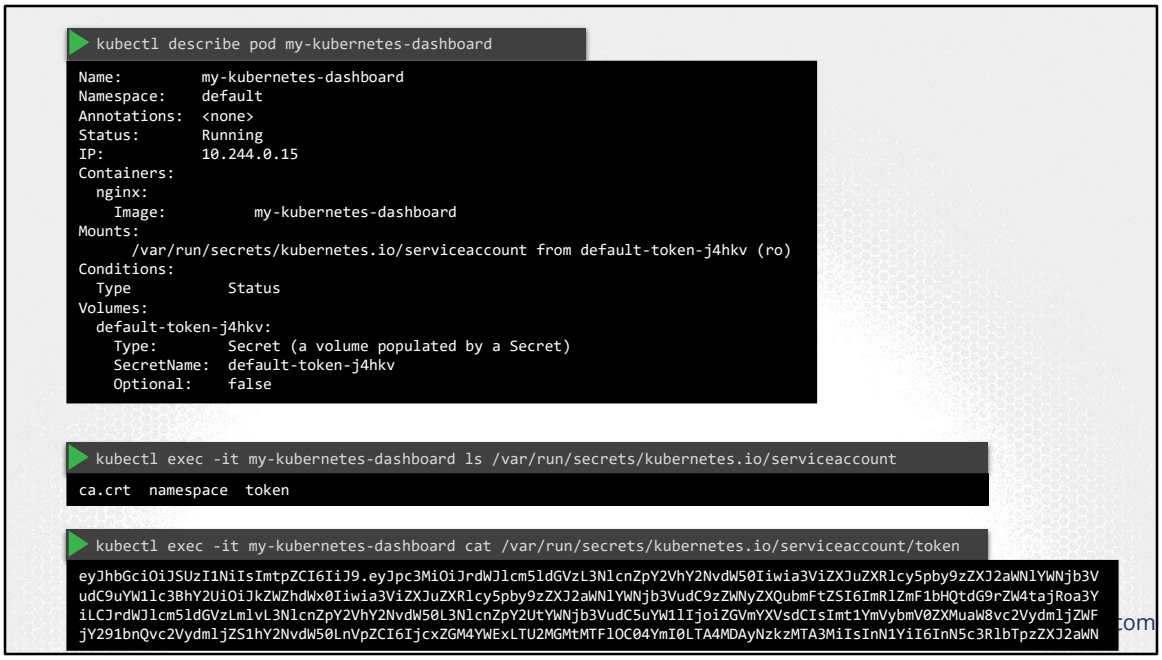


In that case, this whole process of exporting the service account token and configuring the third party application to use it can be made simple by automatically mounting the service token secret as a volume inside the POD hosting the third party application. That way the token to access the kubernetes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

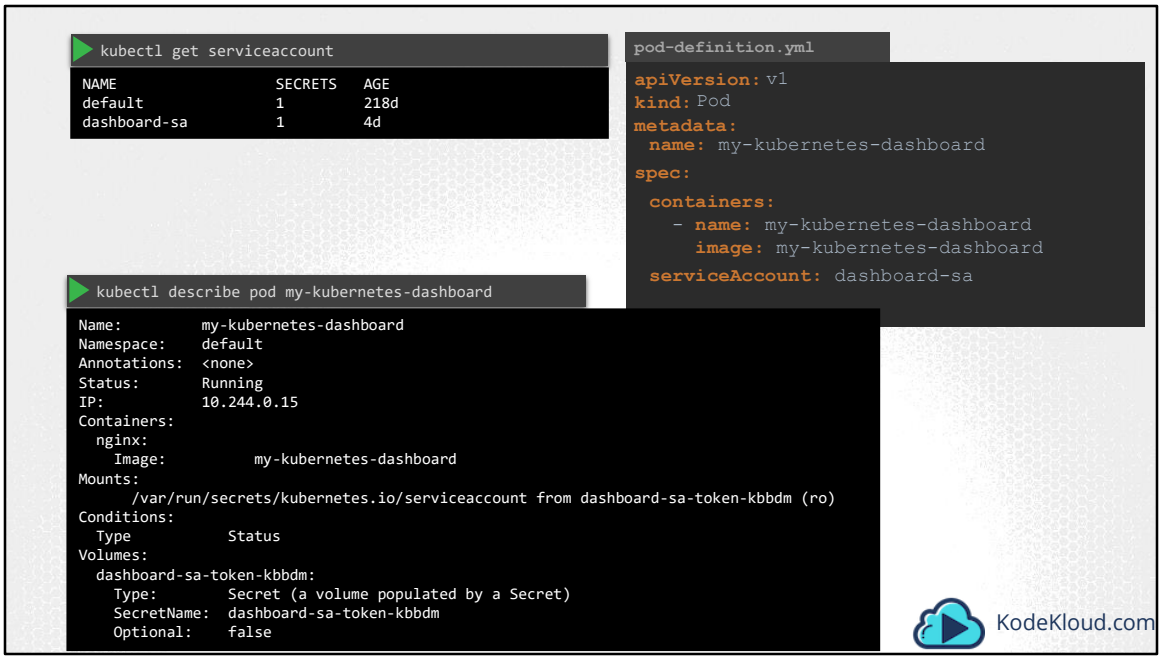


If you go back and look at the list of service accounts, you will see that there is a default service account that exists already. For every namespace in kubernetes a service account named default is automatically created. Each namespace has its own default service account.

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



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.



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

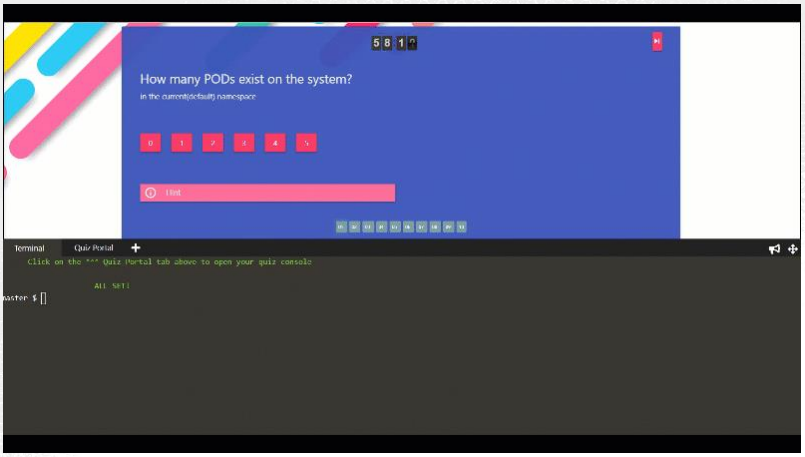
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-kubernetes-dashboard
spec:
  containers:
    - name: my-kubernetes-dashboard
      image: my-kubernetes-dashboard
  automountServiceAccountToken: false
```

KodeKloud.com

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 POD spec 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.

Practice Test



Check Link Below!

KodeKloud.com

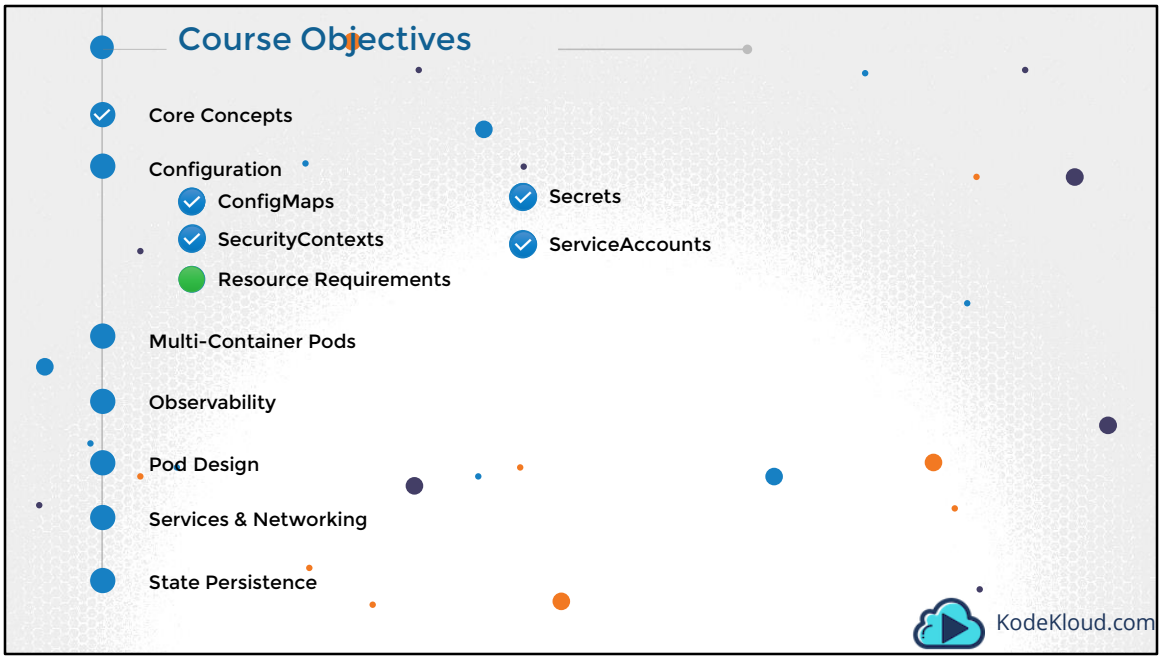
Access Test Here: https://kodekloud.com/courses/kubernetes-certification-course/lectures/8598237

References

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

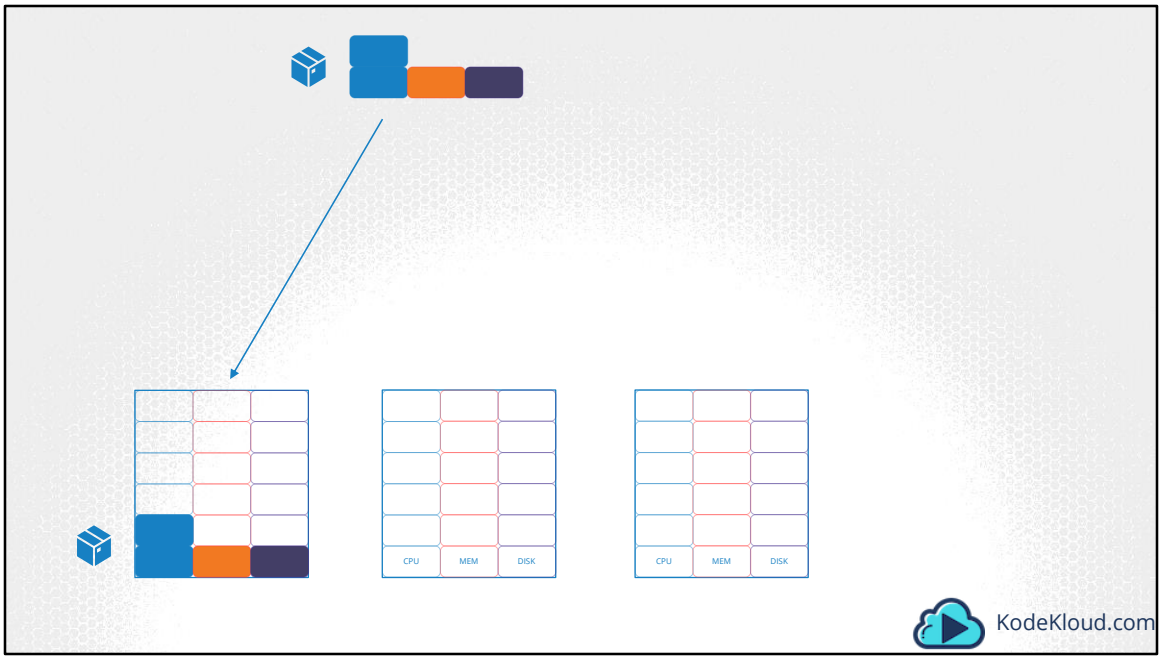
https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/

KodeKloud.com

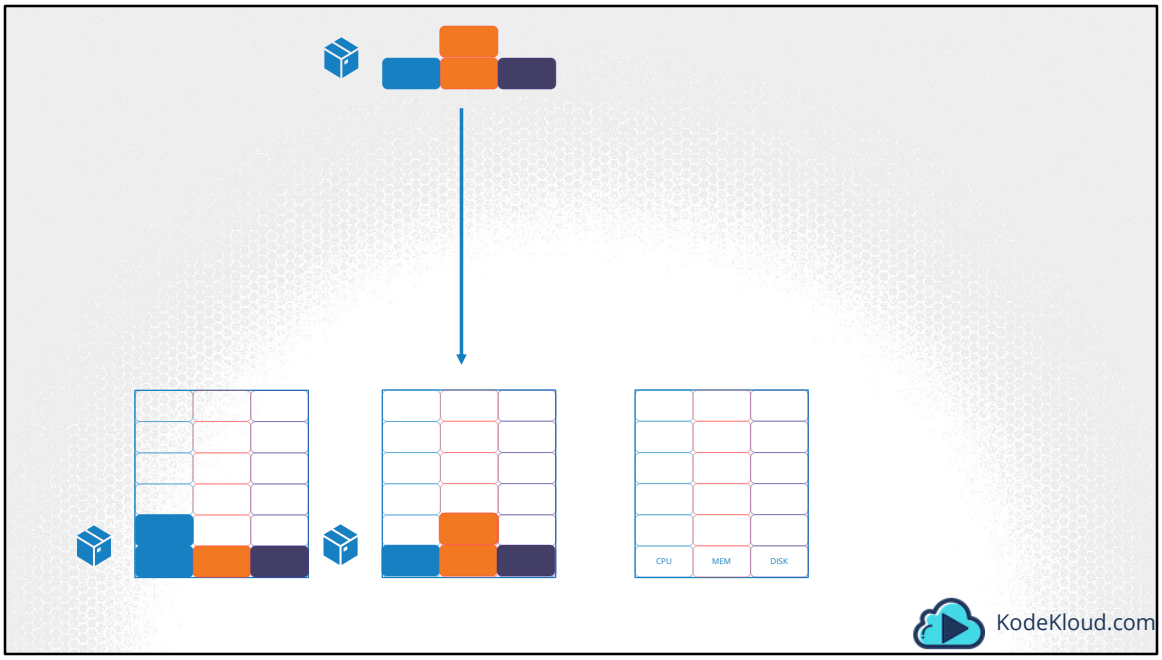


The next section on Configuration covers topics like ConfigMaps, SecurityContexts, Resource Requirements, secrets and service accounts.

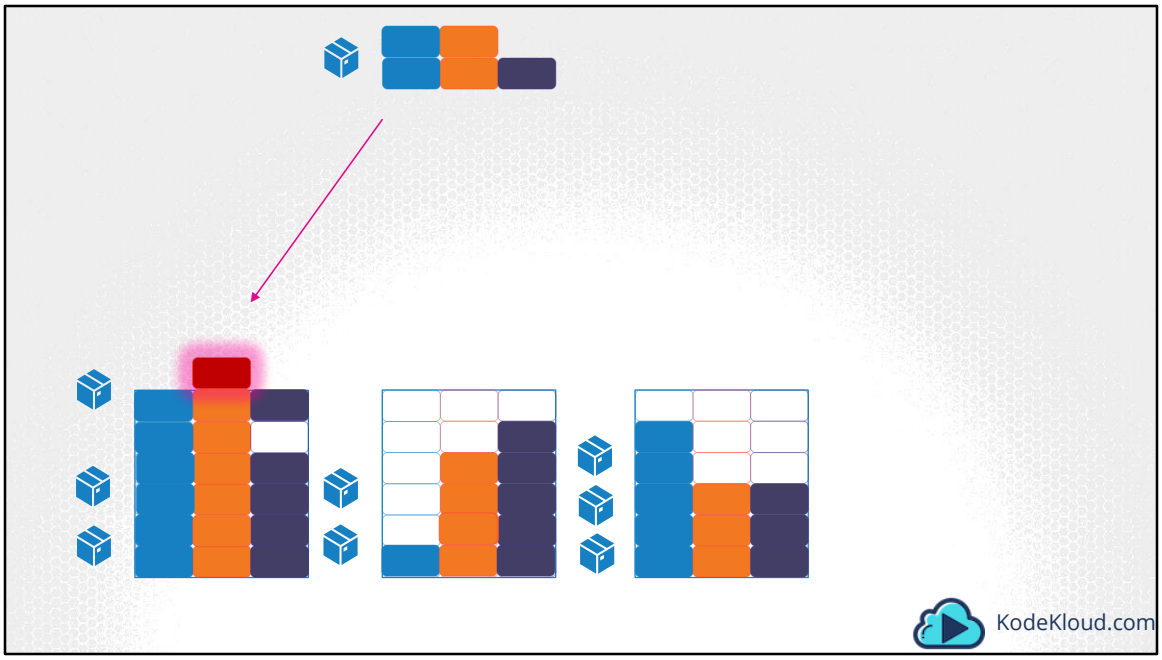

140
Resource Requirements
KodeKloud.com



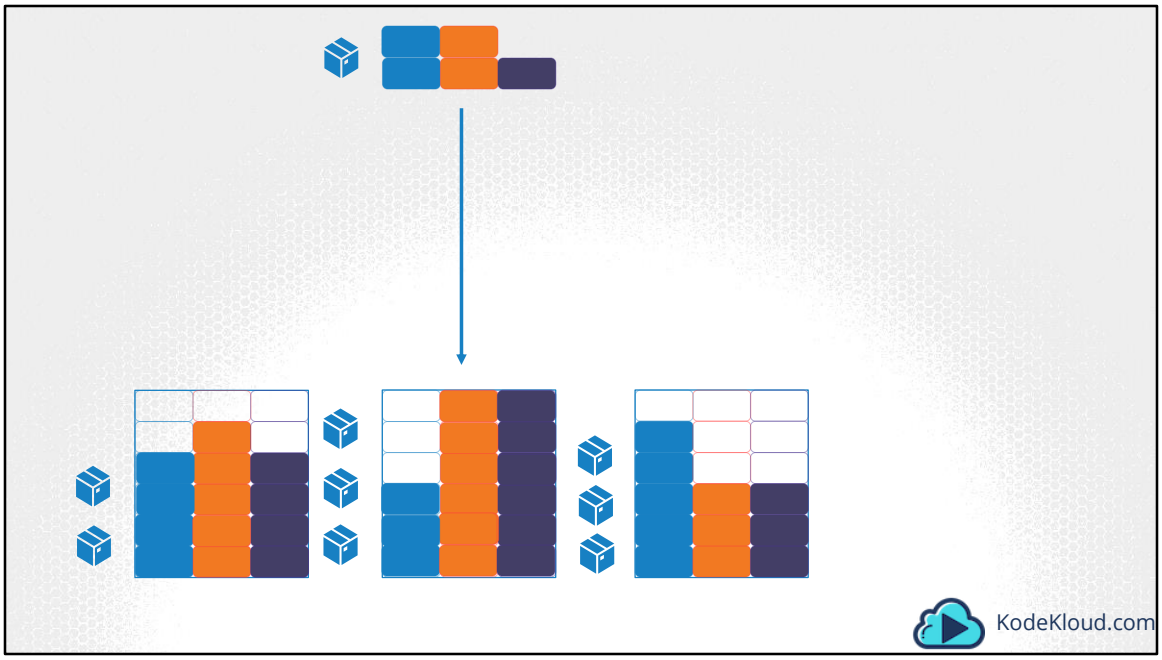
Let us look at a 3 Node Kubernetes cluster. Each node has a set of CPU, Memory and Disk resources available. Every POD consumes a set of resources. In this case 2 CPUs , one Memory and some disk space. Whenever a POD is placed on a Node, it consumes resources available to that node.



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.

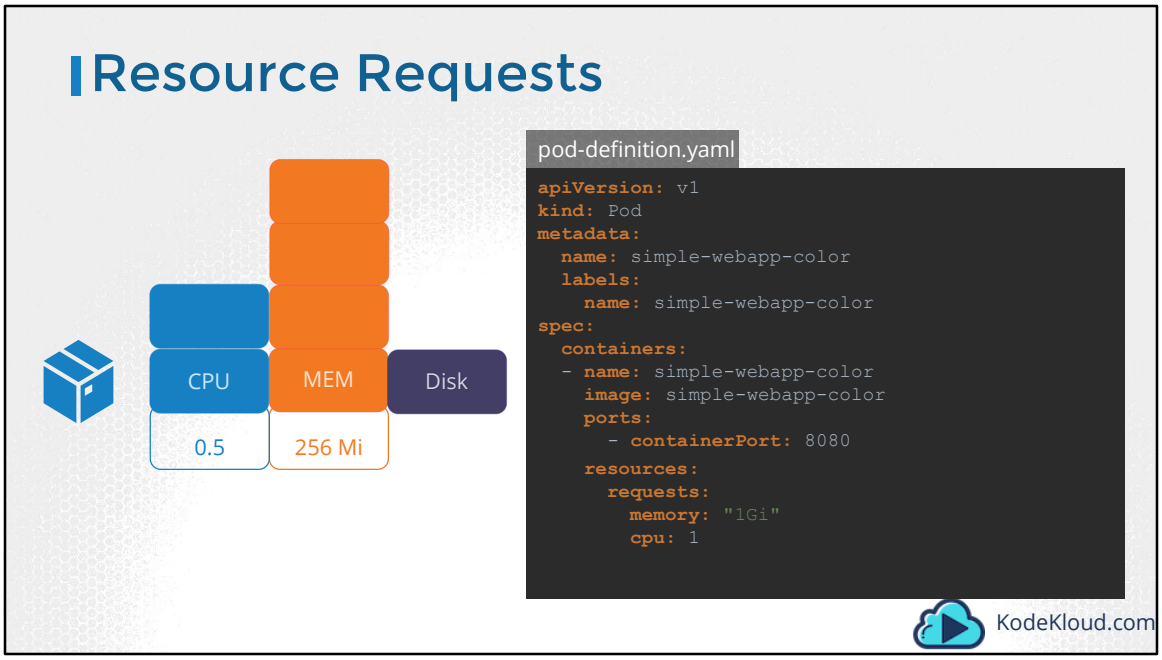


If the node has no sufficient resources, the scheduler avoids placing the POD on that node…

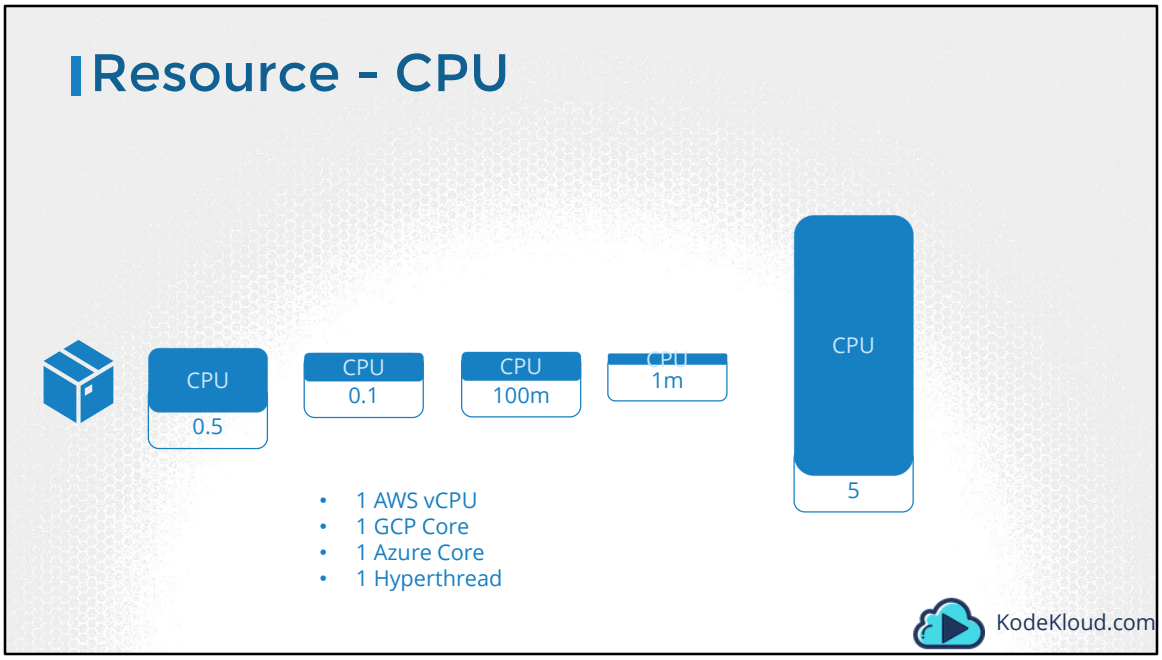


.. Instead places the POD on one were sufficient resources are available. Some of the related topics such as scaling and auto-scaling PODs and Nodes in the cluster and how the scheduler itself works are out of scope for this course and the Kubernetes Application Developer certification. These are discussed 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.

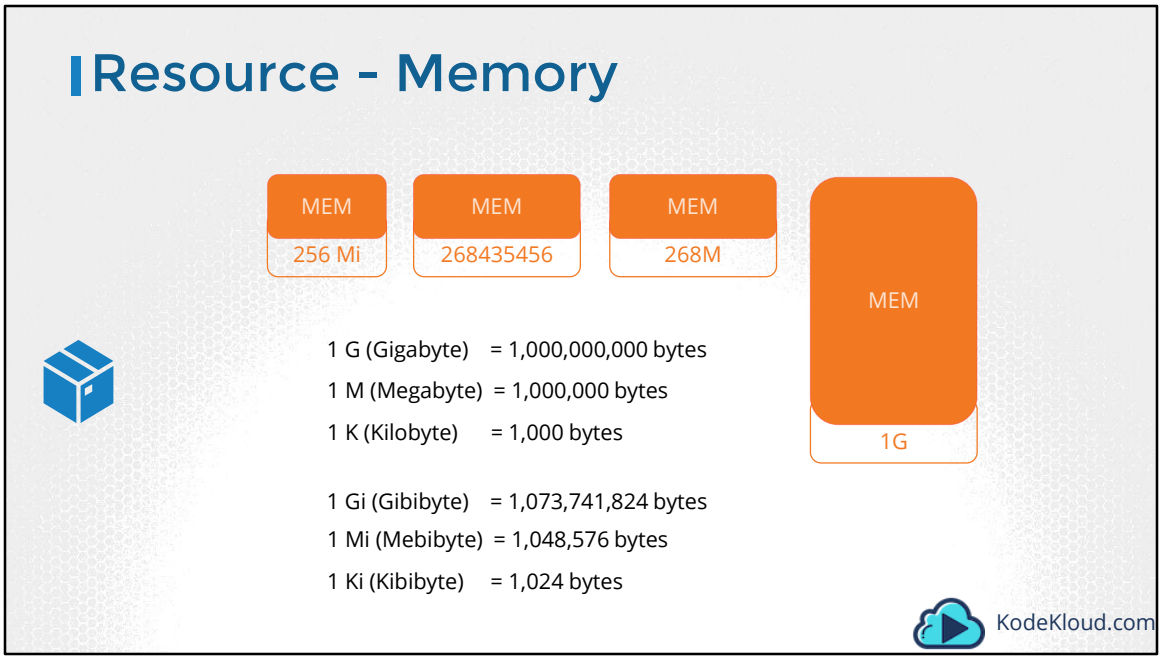
If there is no sufficient resources available on any of the nodes, Kubernetes holds back scheduling the POD, and you will see the POD in a pending state. If you look at the events, you will see the reason – insufficient cpu.



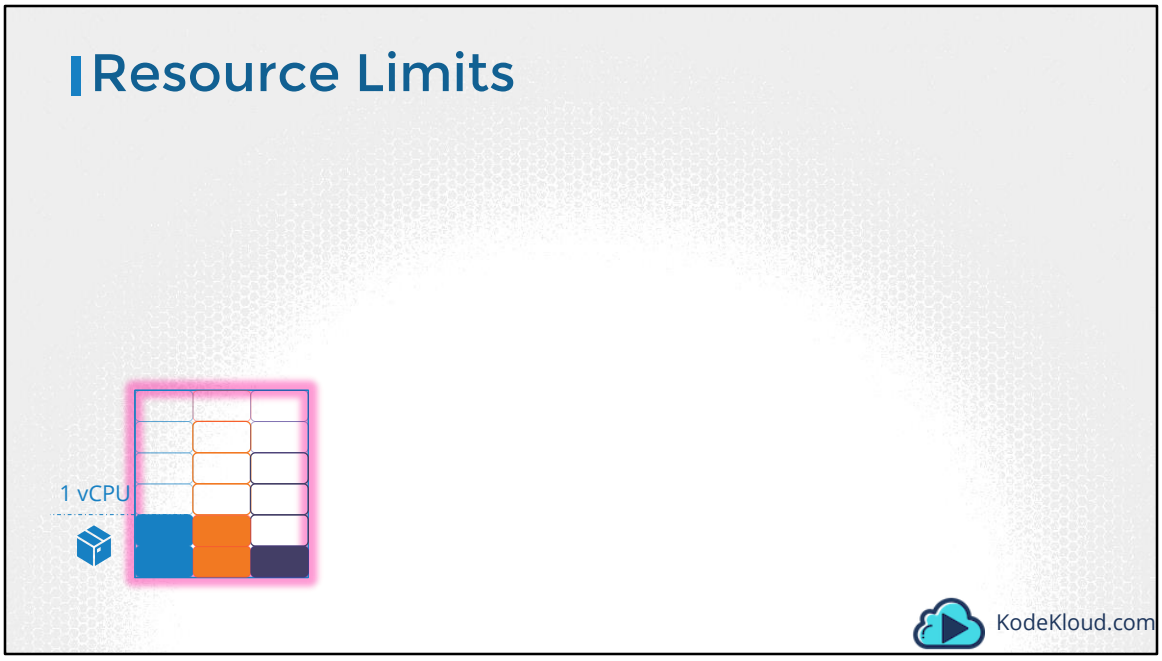
Let us now focus on the resource requirements for each POD. What are these blocks and what are their values? By default, kubernetes assumes that a POD or a container within a POD requires .5 CPU & 256 Mebibyte of memory. This is known as the resource request for a container. The minimum amount of CPU 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.



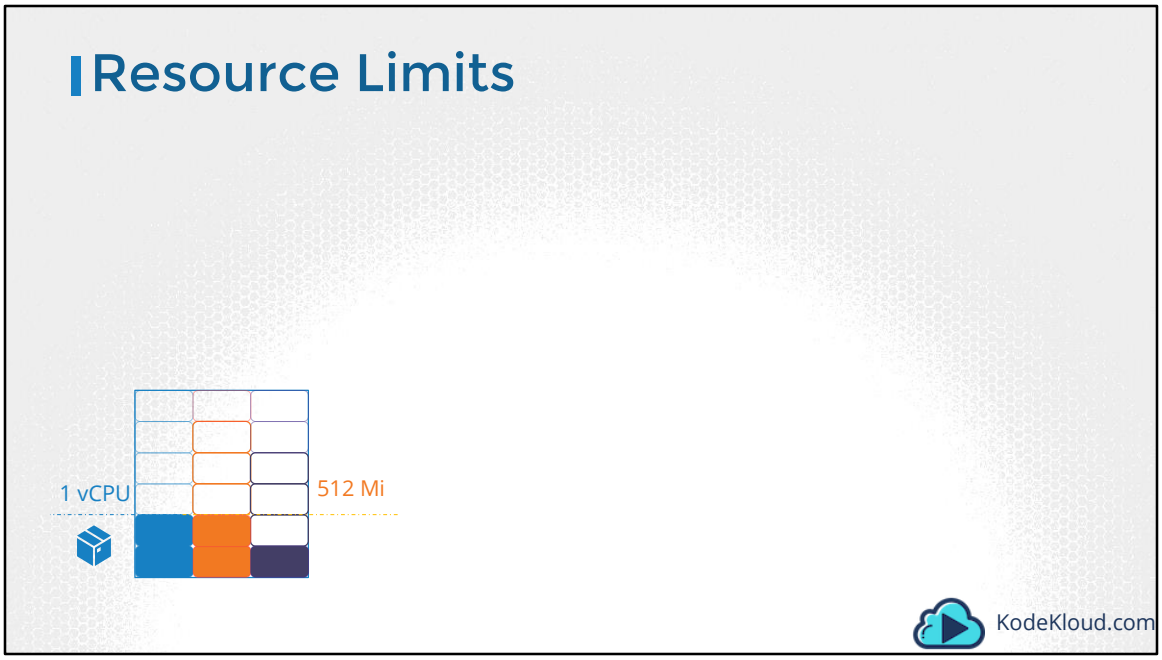
So what does 1 count of CPU really mean? Remember these blocks are used for illustration purpose only. It doesn't have to be in the increment of .5. You can specify any value as low as 0.1. 0.1 CPU can also be expressed as 100m were m stands for milli. You can go as low as 1m, but not lower than that. 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.



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



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.



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

Resource Limits

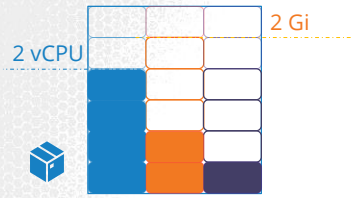
pod-definition.yaml

```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



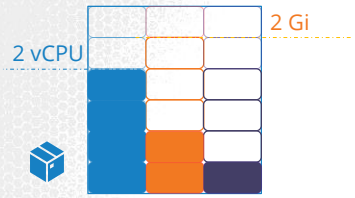
pod-definition.yaml

```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

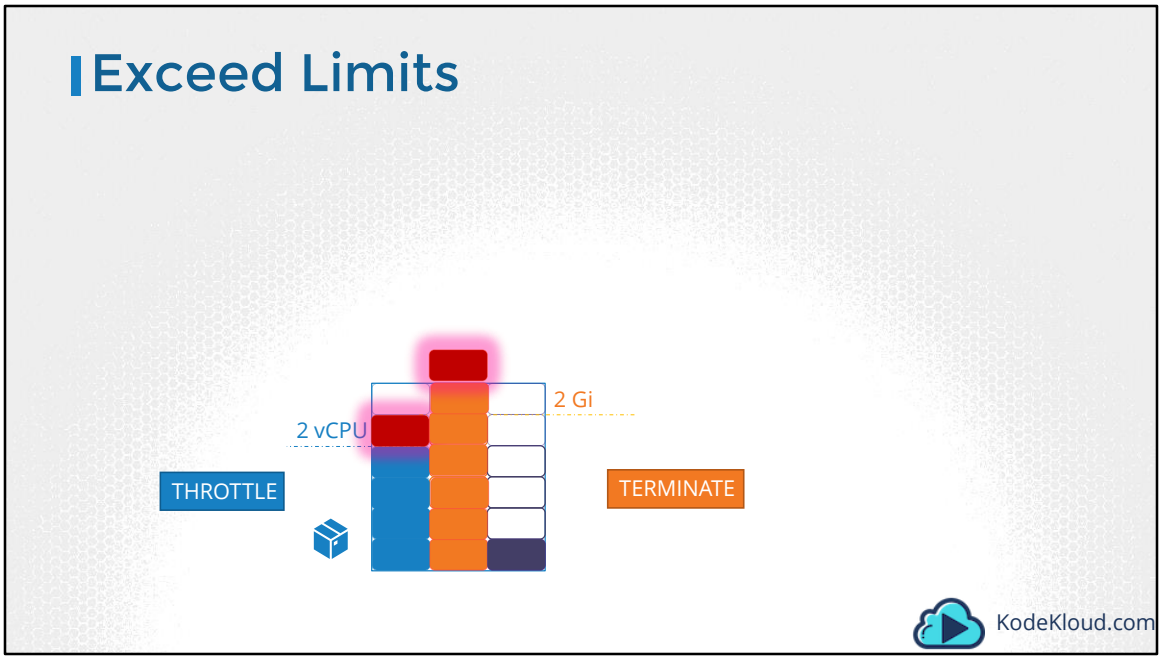


pod-definition.yaml

```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
```

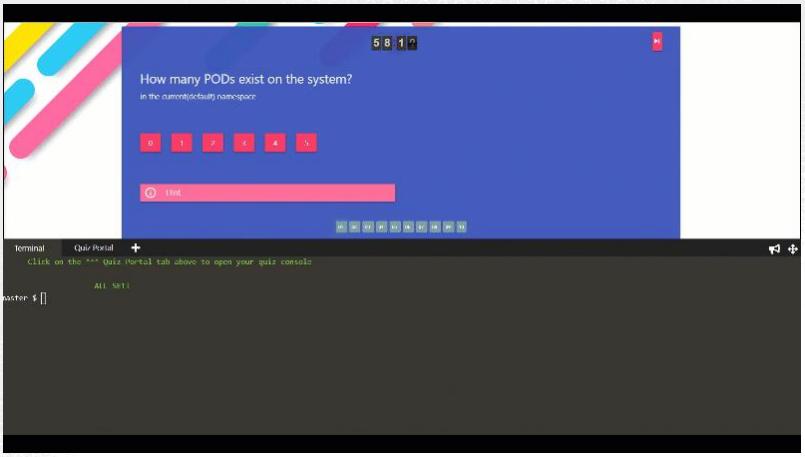
KodeKloud.com

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



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 that 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

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/

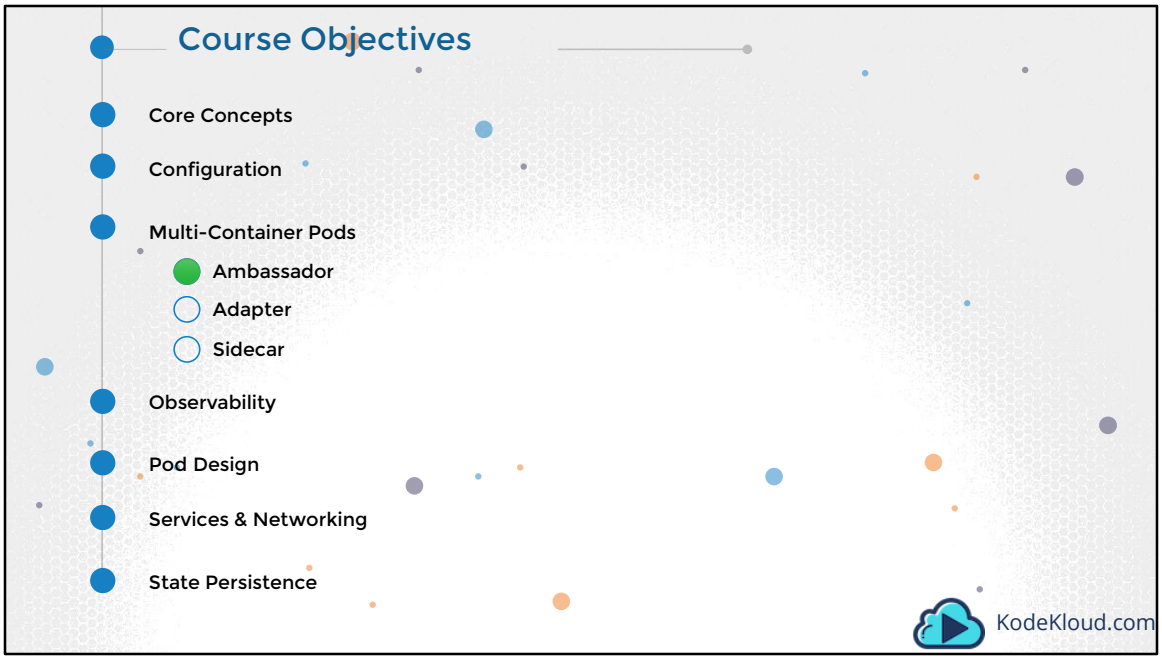
https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/cpu-default-namespace/

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/

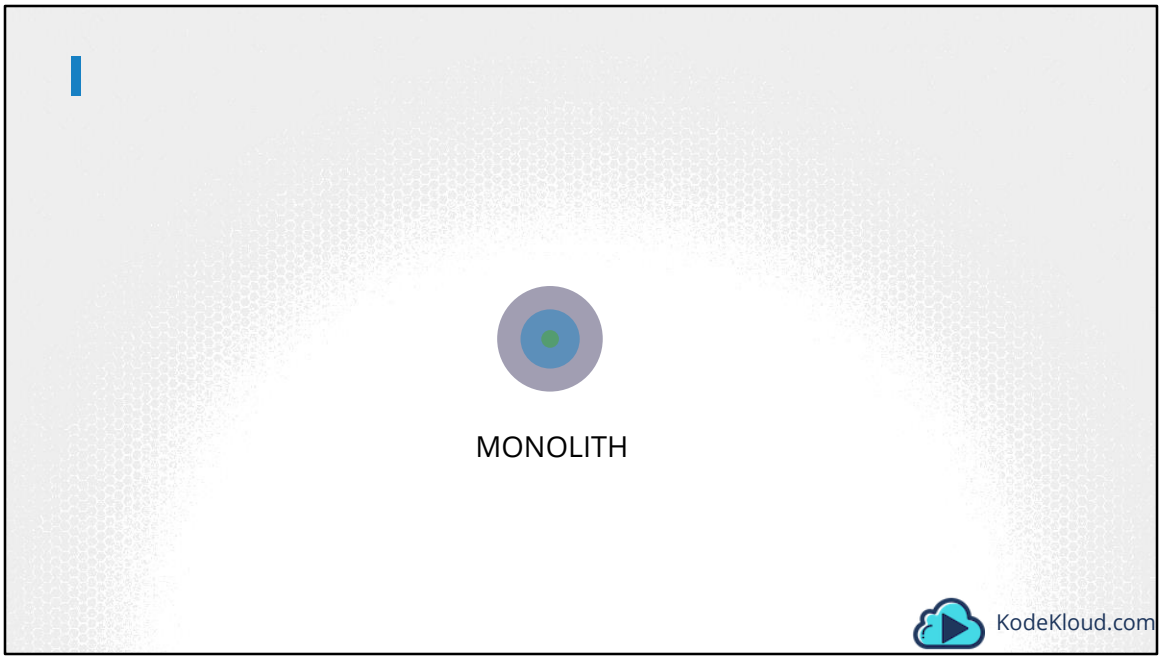
KodeKloud.com



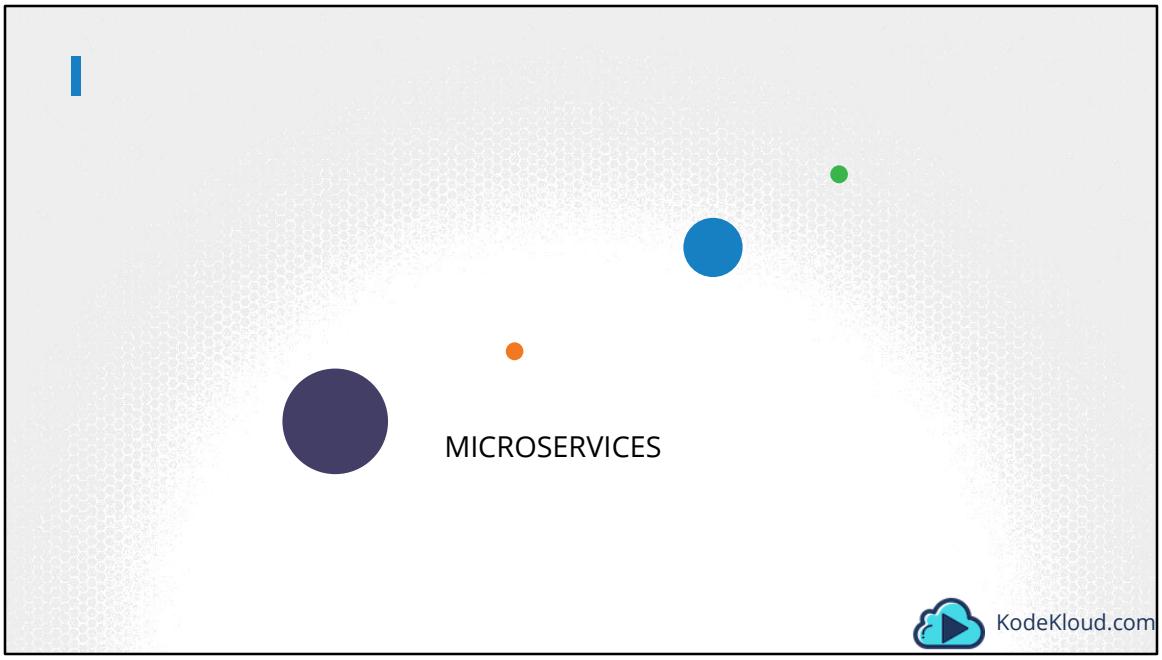
Hello and welcome to this section on Multi-Container Pods. My name is Mumshad Mannambeth and we are going through the Certified Kubernetes Applications Developer course.

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.

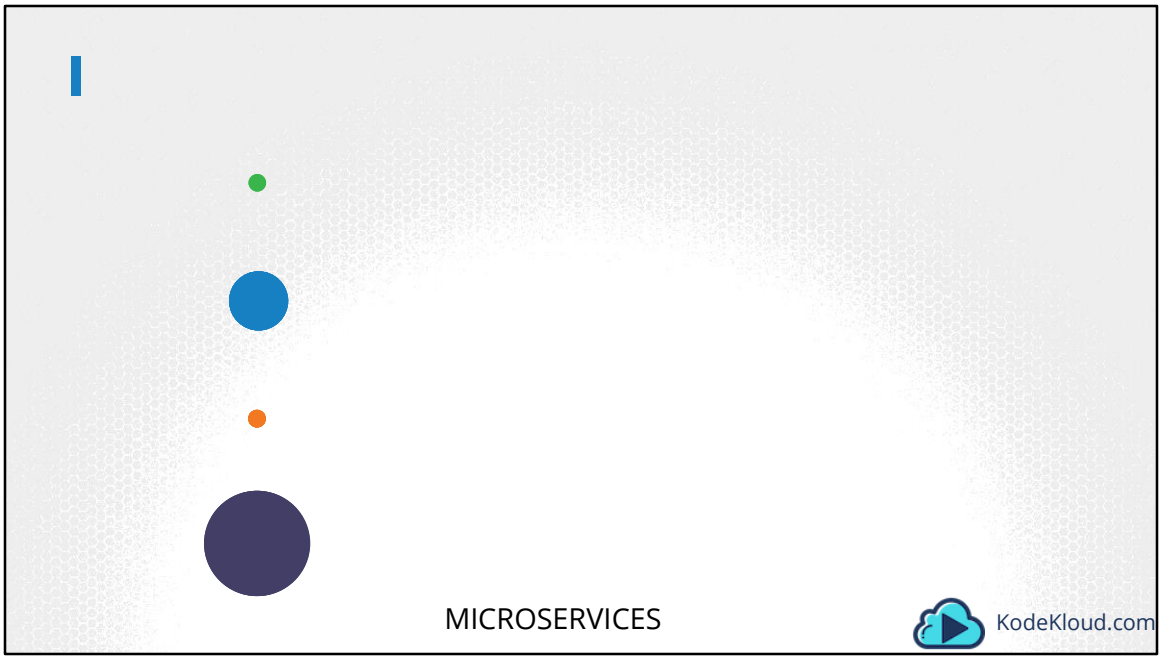
Before we head into each of these, let us start with the basic type of POD.



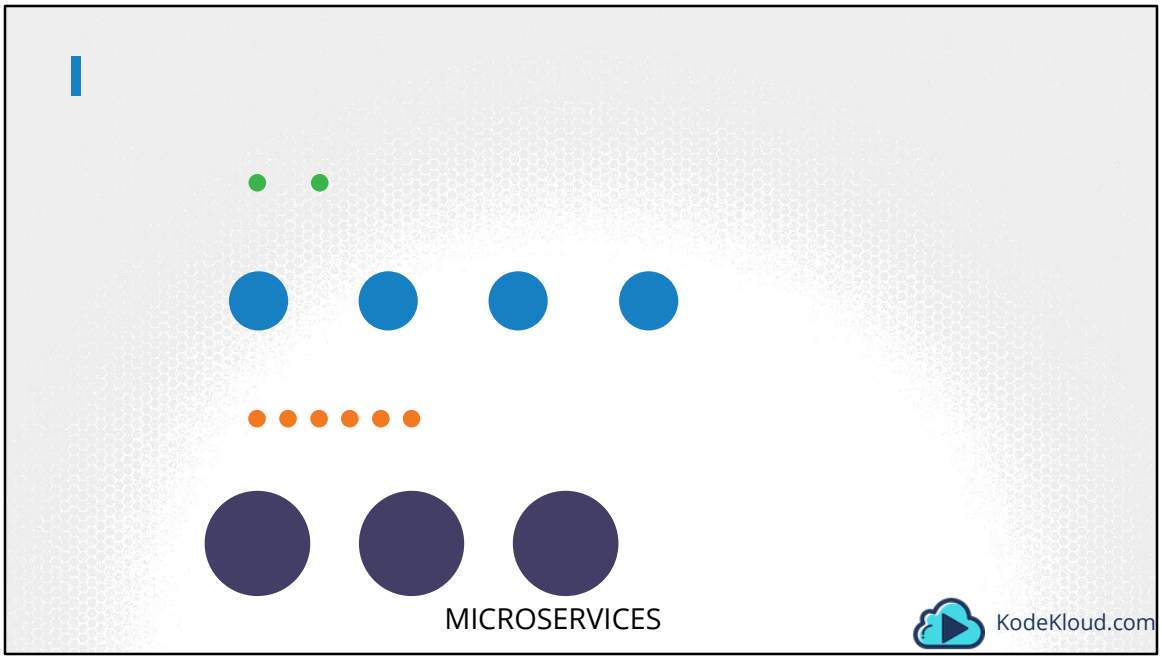
The idea of decoupling a large monolithic application into …



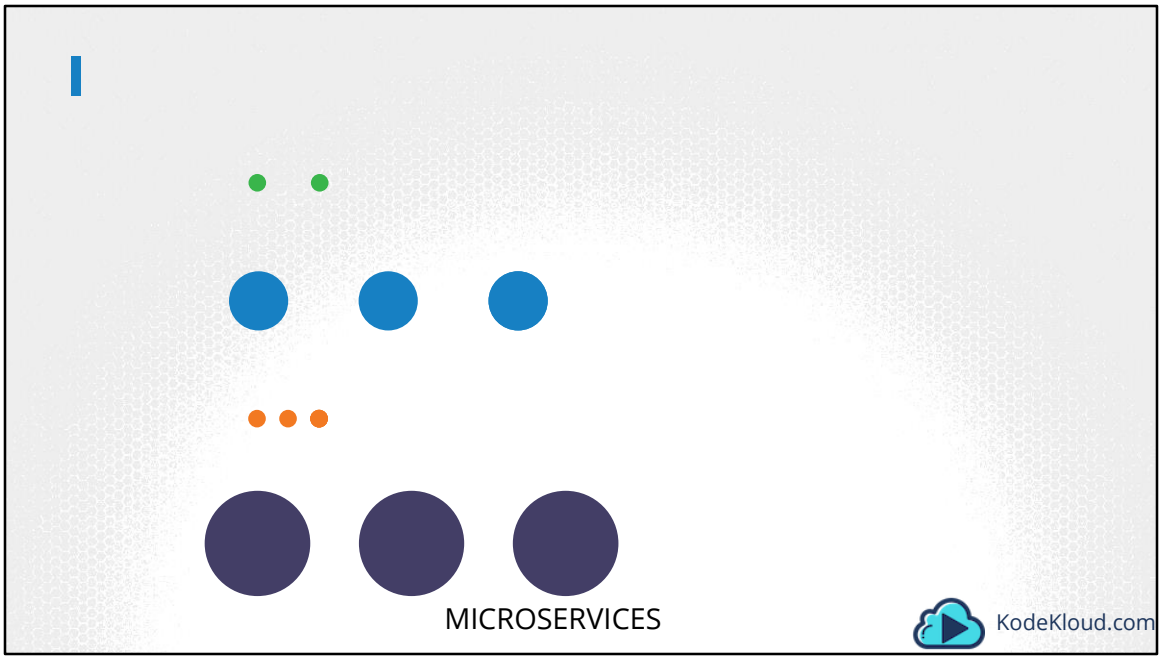
.. subcomponents known as microservices enables us to develop and deploy a set of independent, small and reusable code.



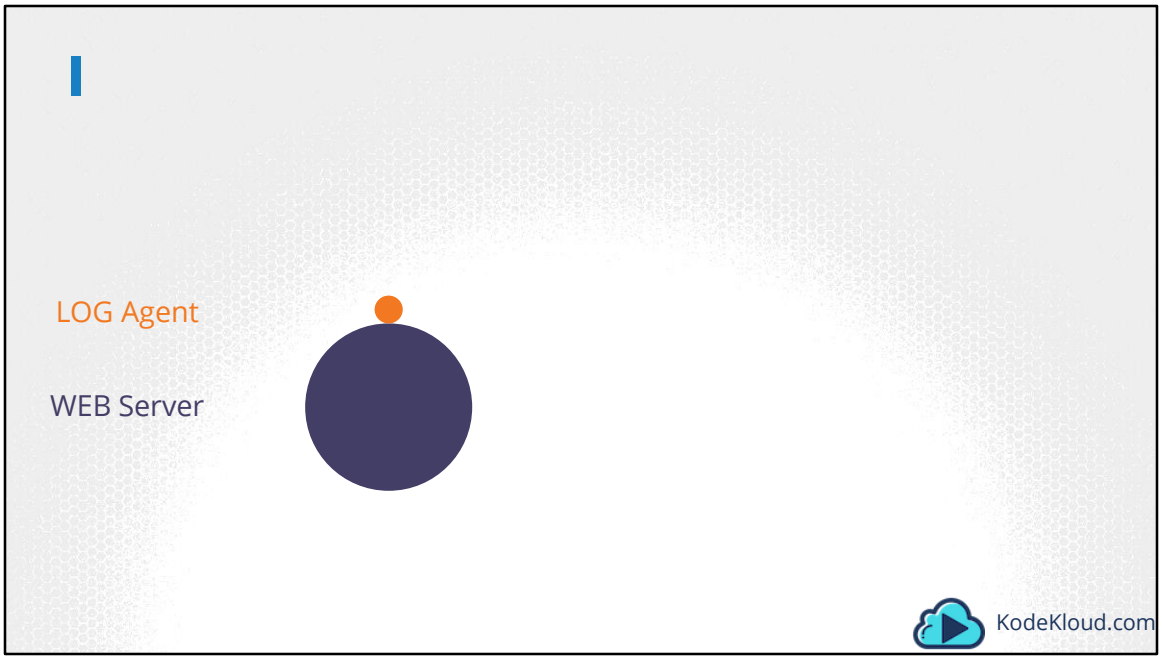
This, architecture then helps us



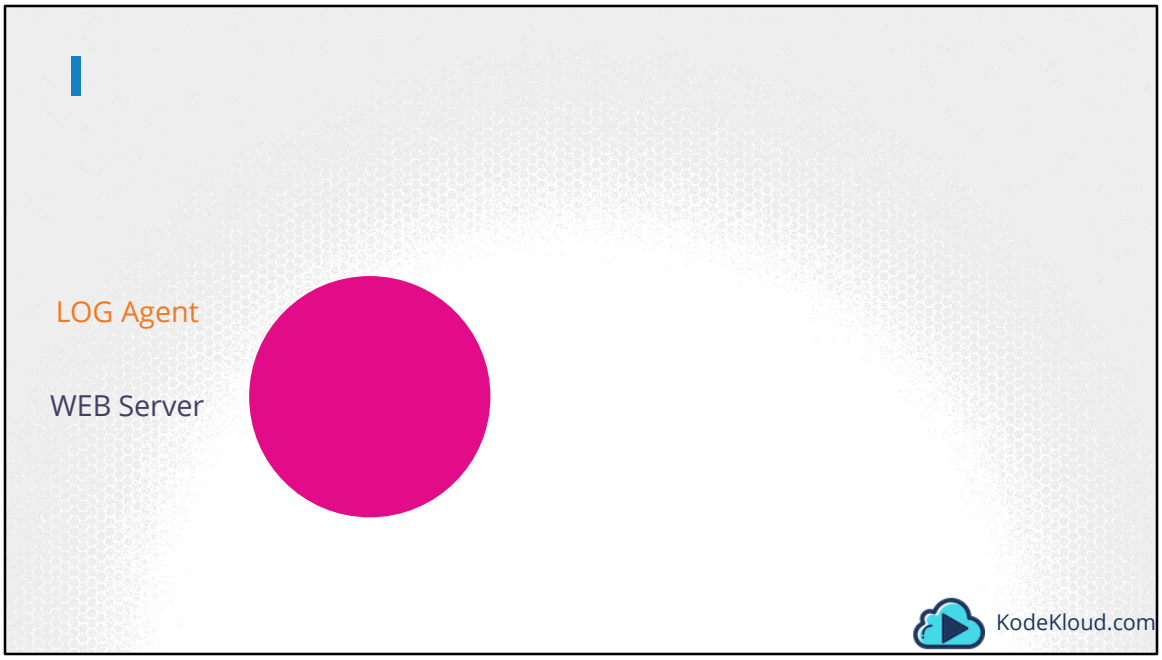
scale up..



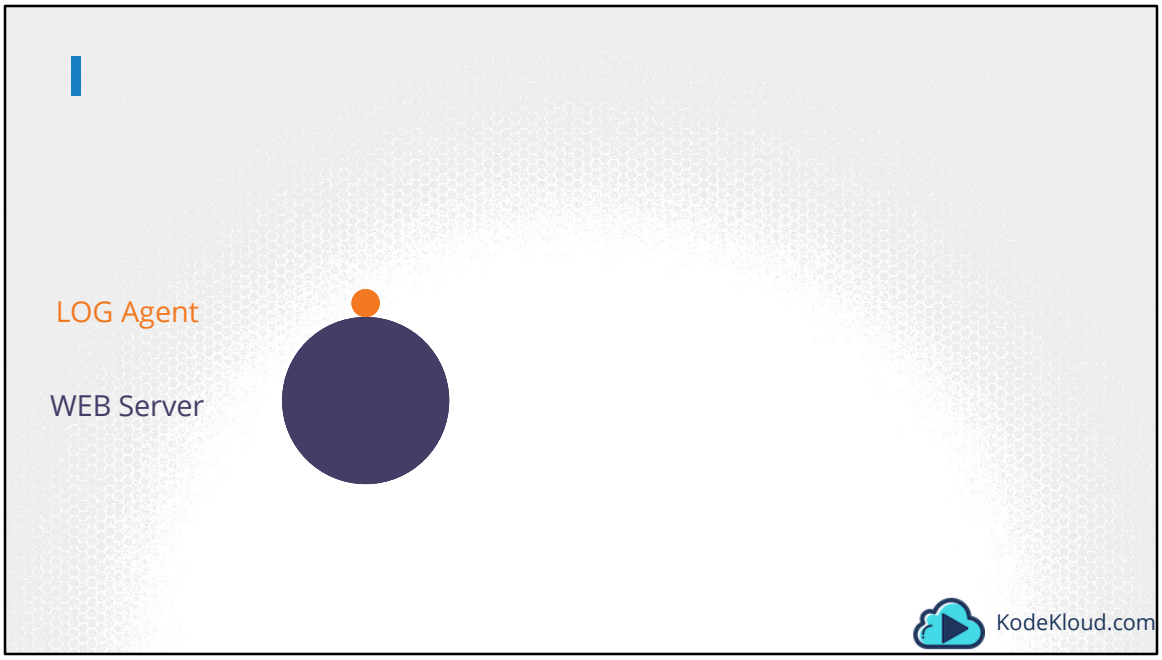
… , down as well as modify each service as required, as opposed to modifyin the entire application.



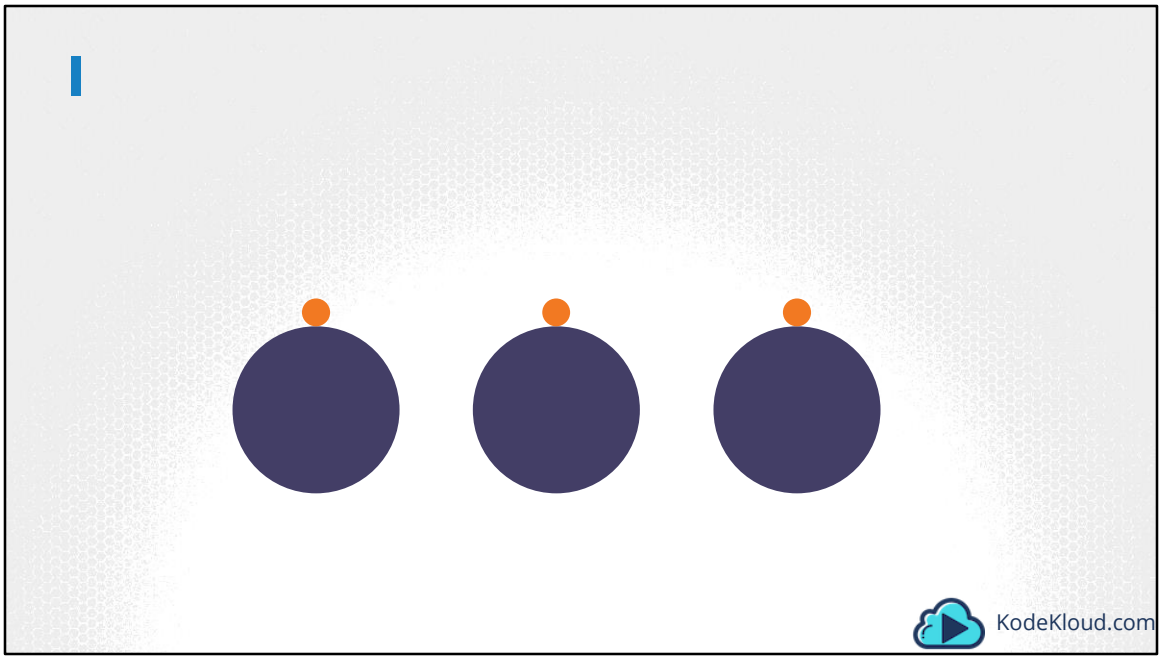
However, at times you may need two services to work together. Such as a web server and a logging service. You need one agent per web server instance paired together …



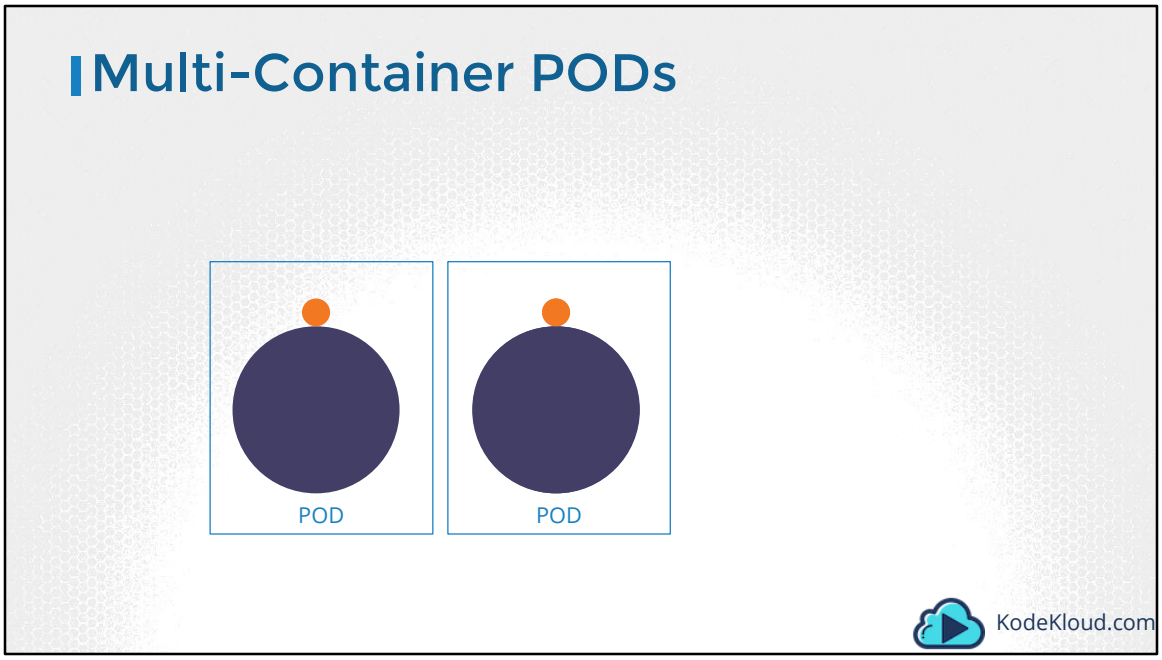
You don’t want to merge and bloat the code of the two services, as each of them target different features, and you’d still like them to be developed and deployed separately.



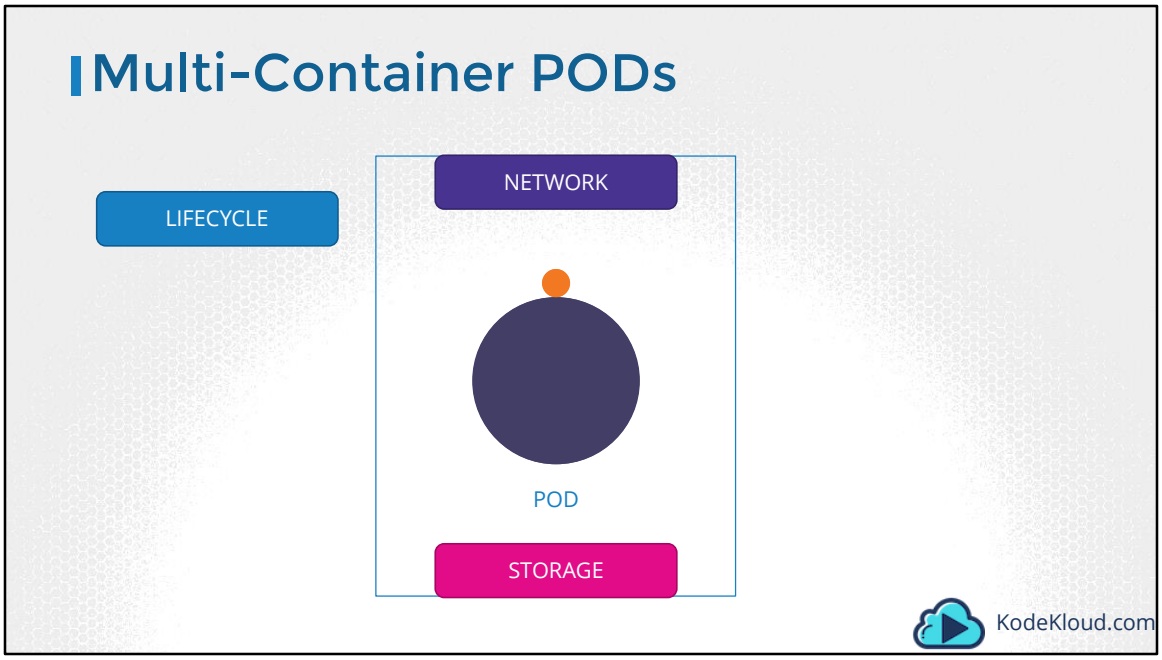
You only need the two functionality to work together. You need one agent per web server instance paired together …



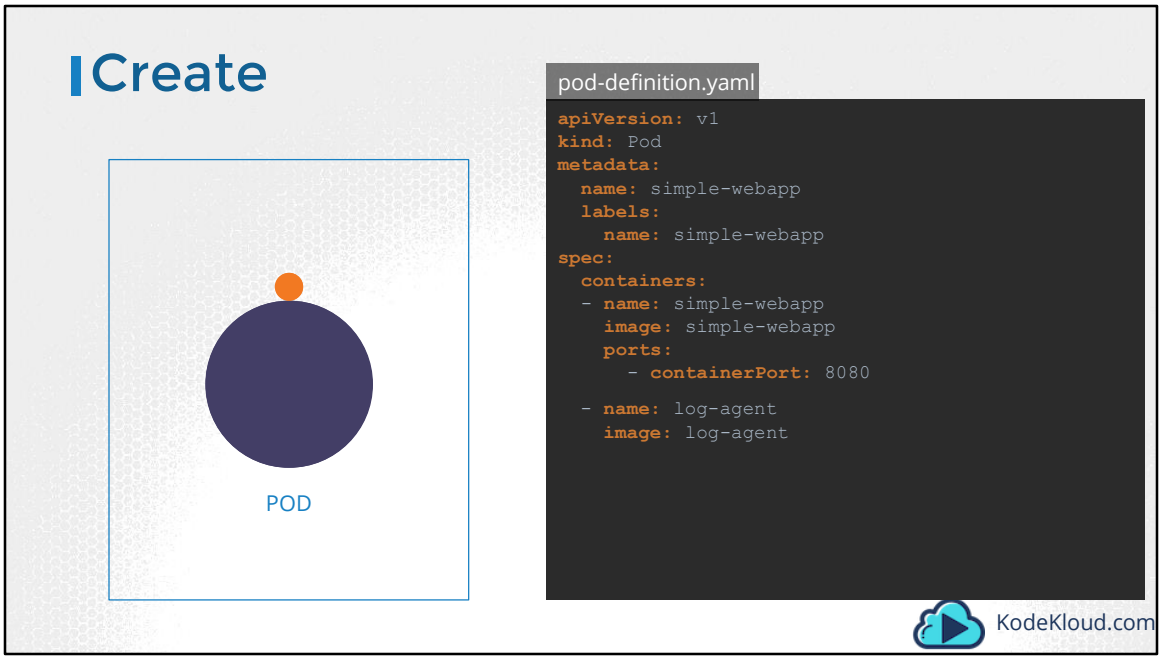
.. that can scale up …



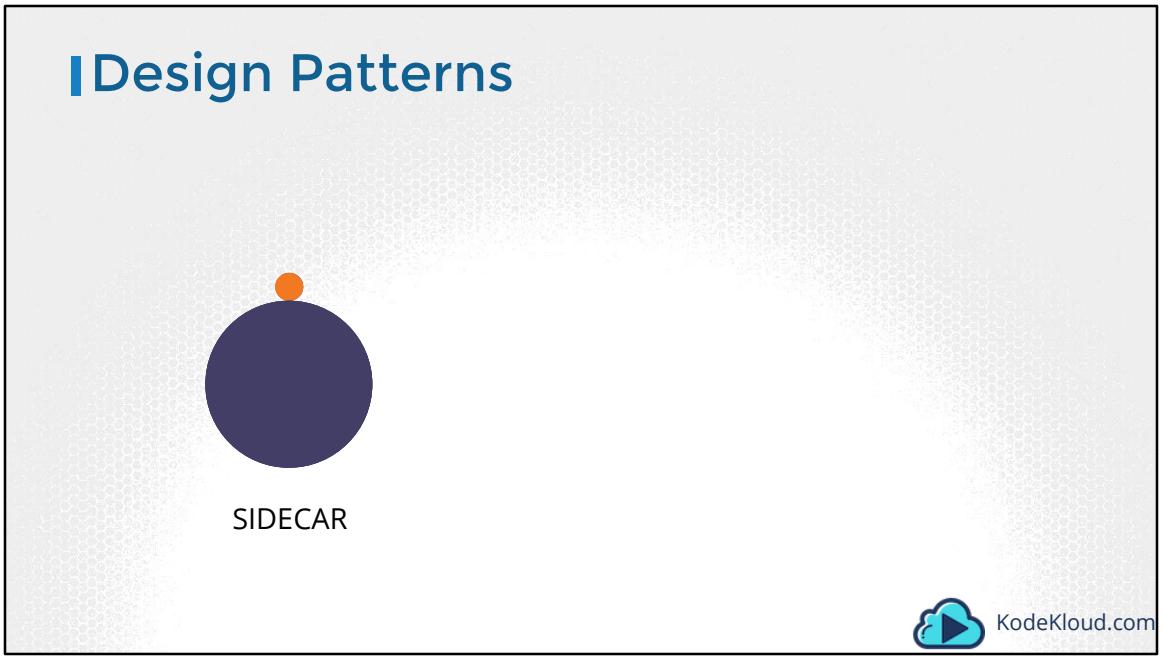
.. and down together. And that is why you have multi-container PODs….



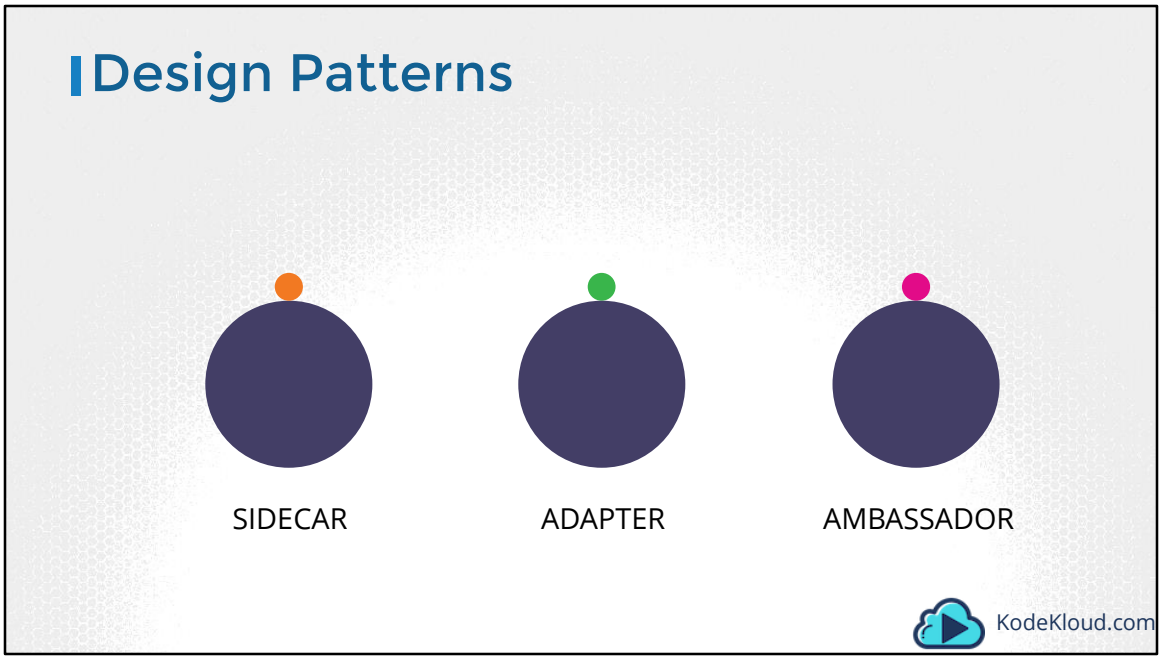
that share the same lifecycle – which means they are created together and destroyed together. They share the same network space, which means they can refer to each other as localhost. And they have access to the same storage volumes. This way, you do not have to establish, volume sharing or services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



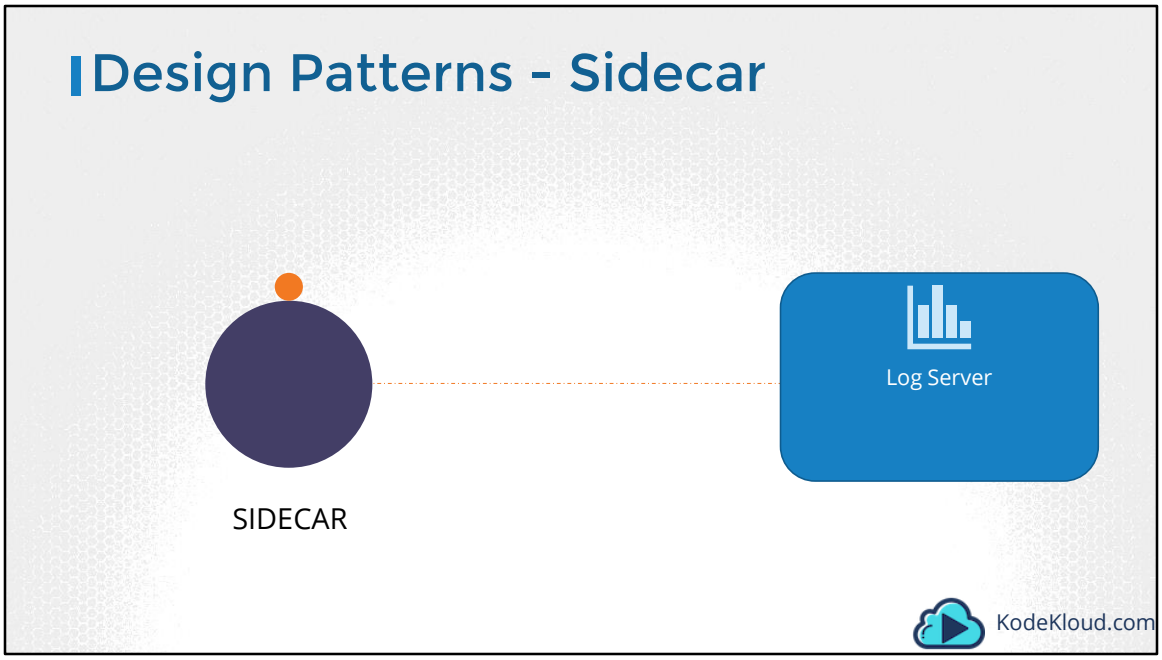
To create a multi-container pod, add the new container information to the pod-definition file. Remember, the containers section under the spec section in a pod definition file is an array and the reason it is an array is to allow multiple containers in a single POD. In this case we add a new container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



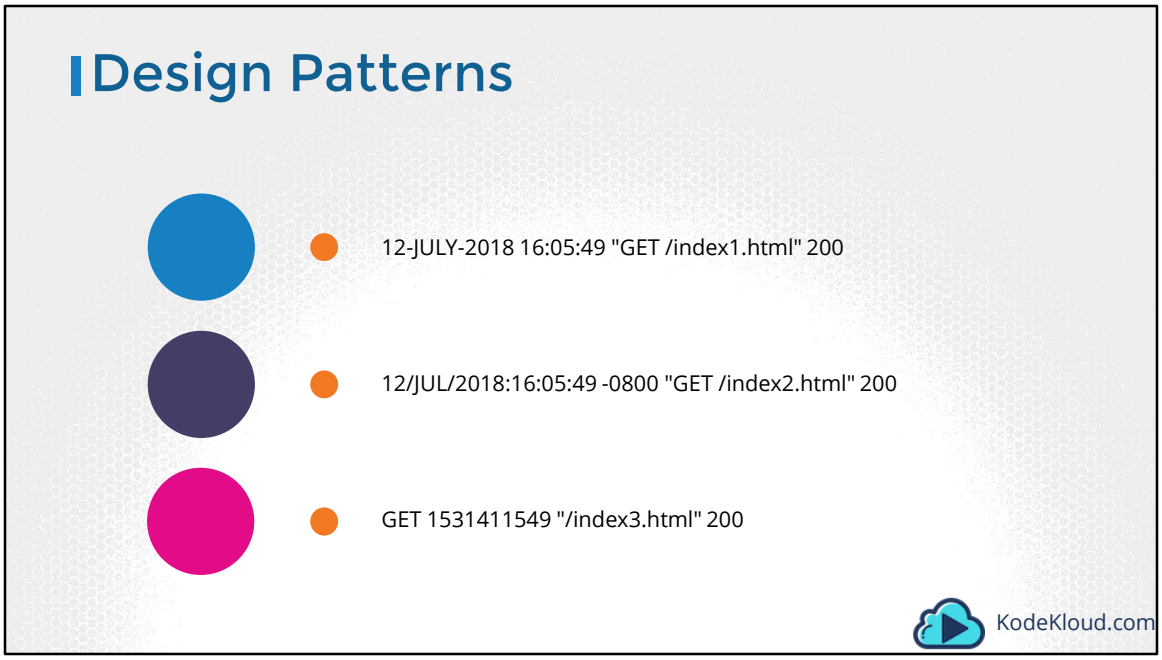
There are 3 common patterns, when it comes to designing multi-container PODs. The first and what we just saw with the logging service example is known as a side car pattern.



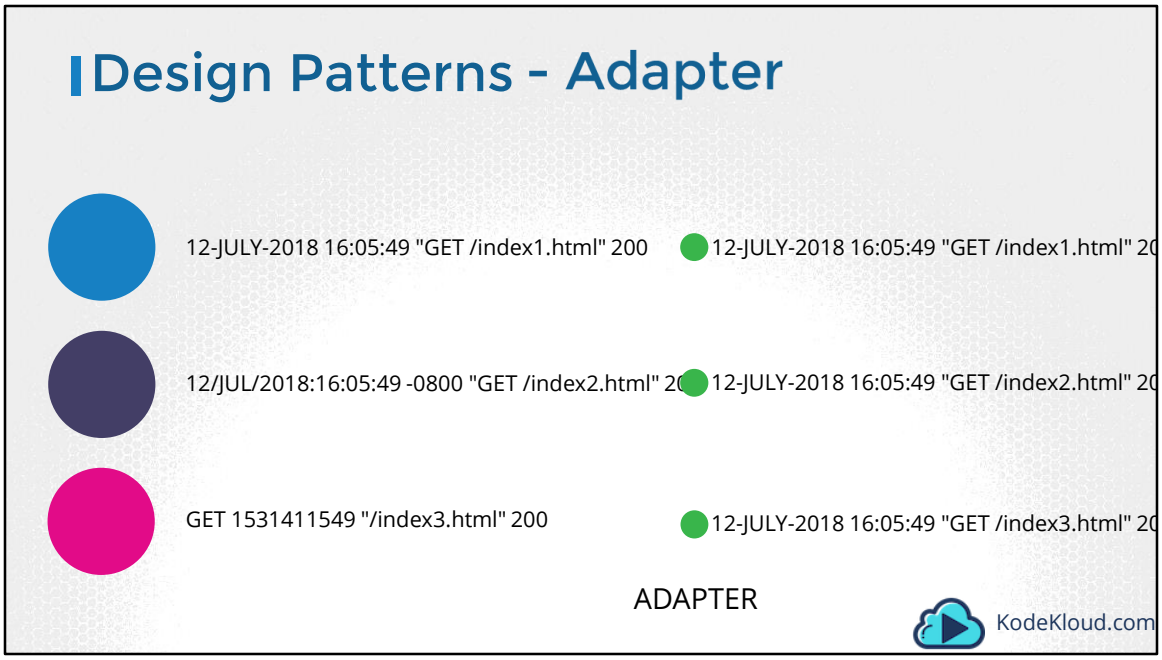
The others are the adapter and the ambassador pattern.



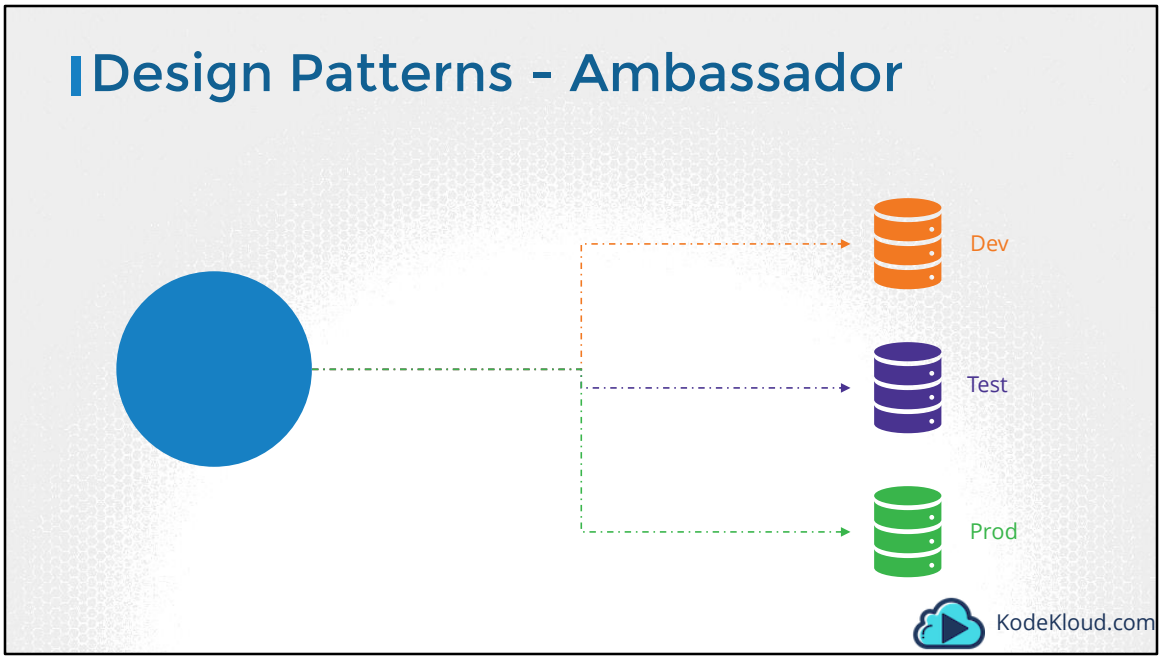
A good example of a side car pattern is deploying a logging agent along side a web server to collect logs and forward them to a central log server.



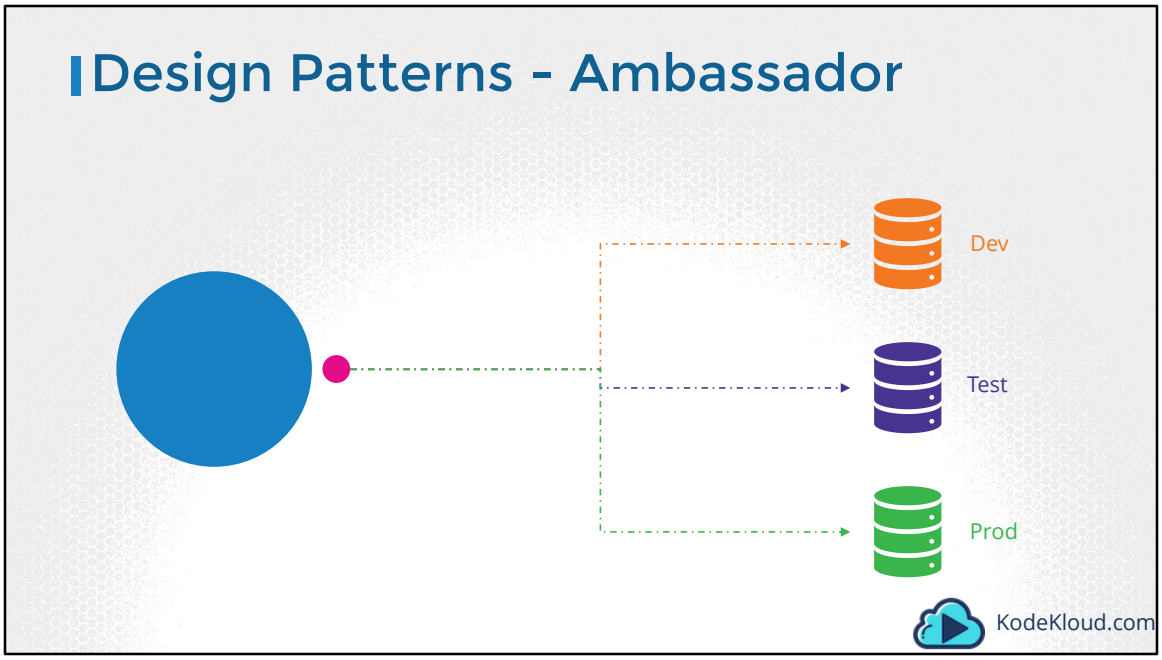
Building on that example, say we have multiple applications generating logs in different formats. It would be hard to process the various formats on the central logging server.



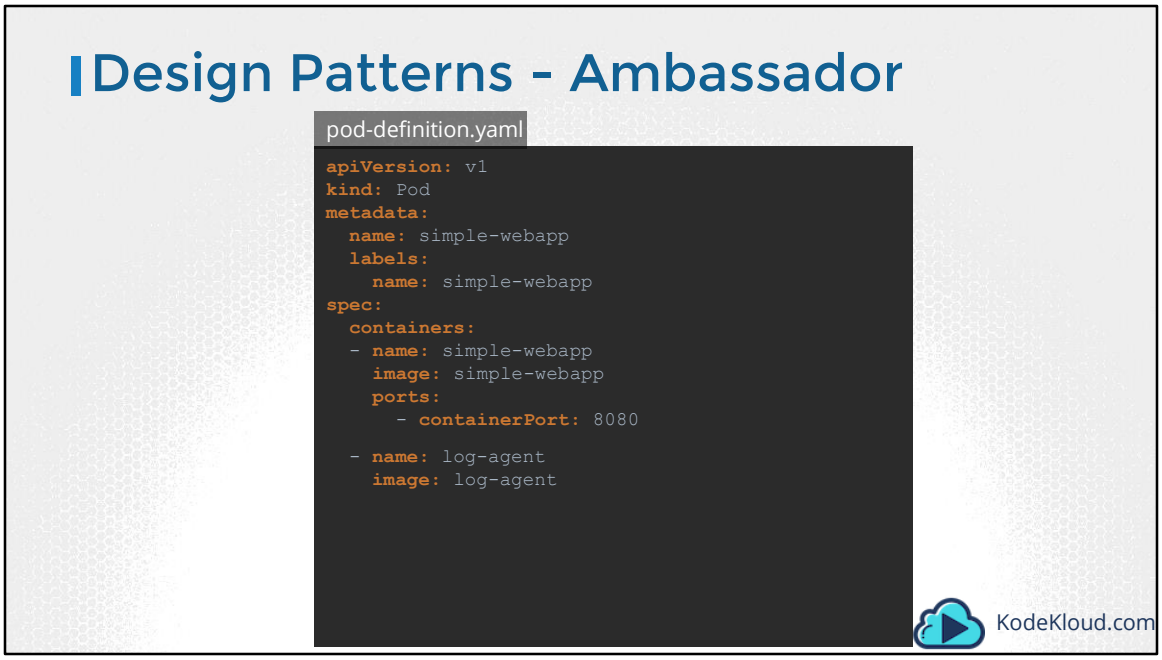
So, before sending the logs to the central server, we would like to convert the logs to a common format. For this we deploy an adapter container. The adapter container processes the logs, before sending it to the central server.



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.



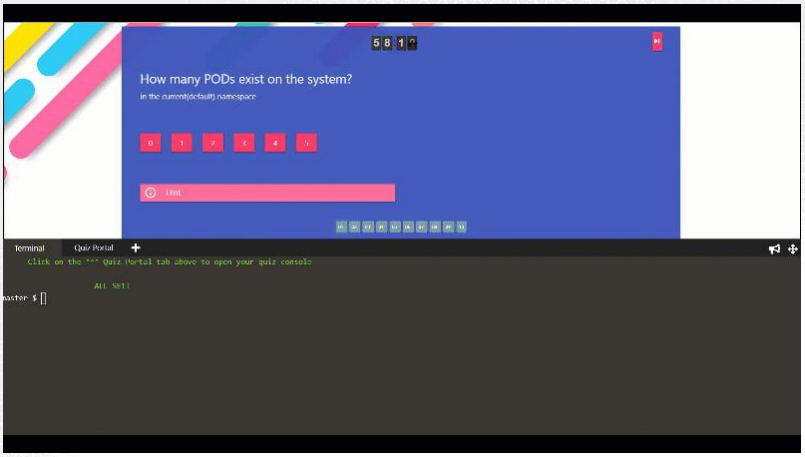
You may chose to outsource such logic to a separate container within your POD, so that your application can always refer to a database at localhost, and the new container, will proxy that request to the right database. This is known as an ambassador container.



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

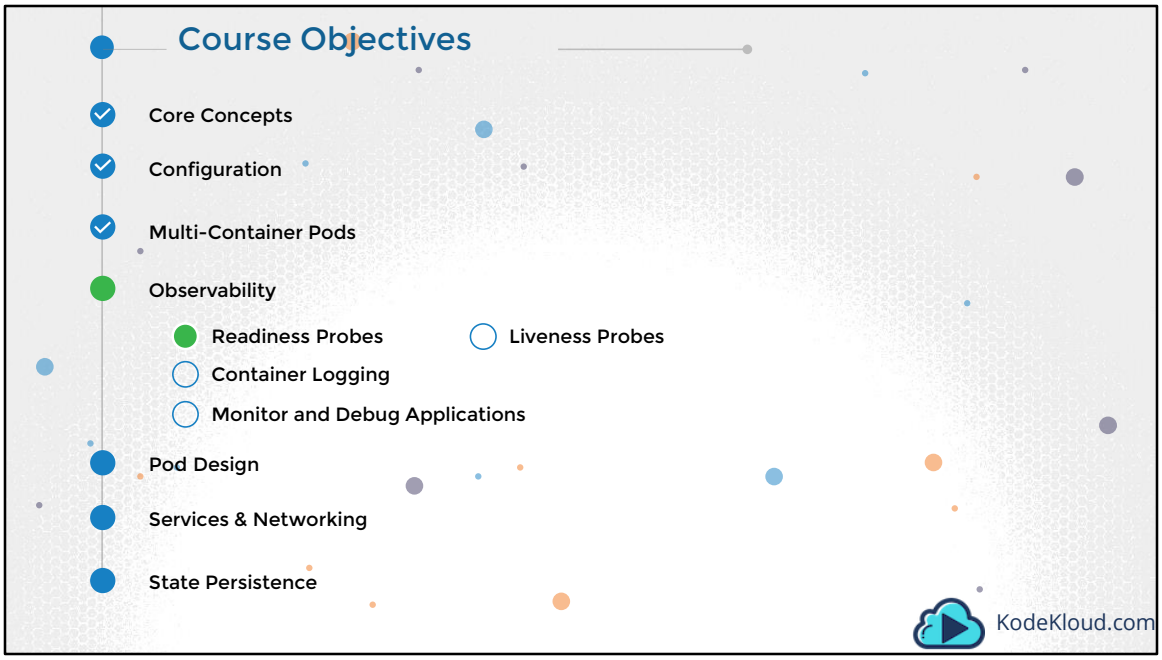
Access Test Here: https://kodekloud.com/courses/kubernetes-certification-course/lectures/6743661

References

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

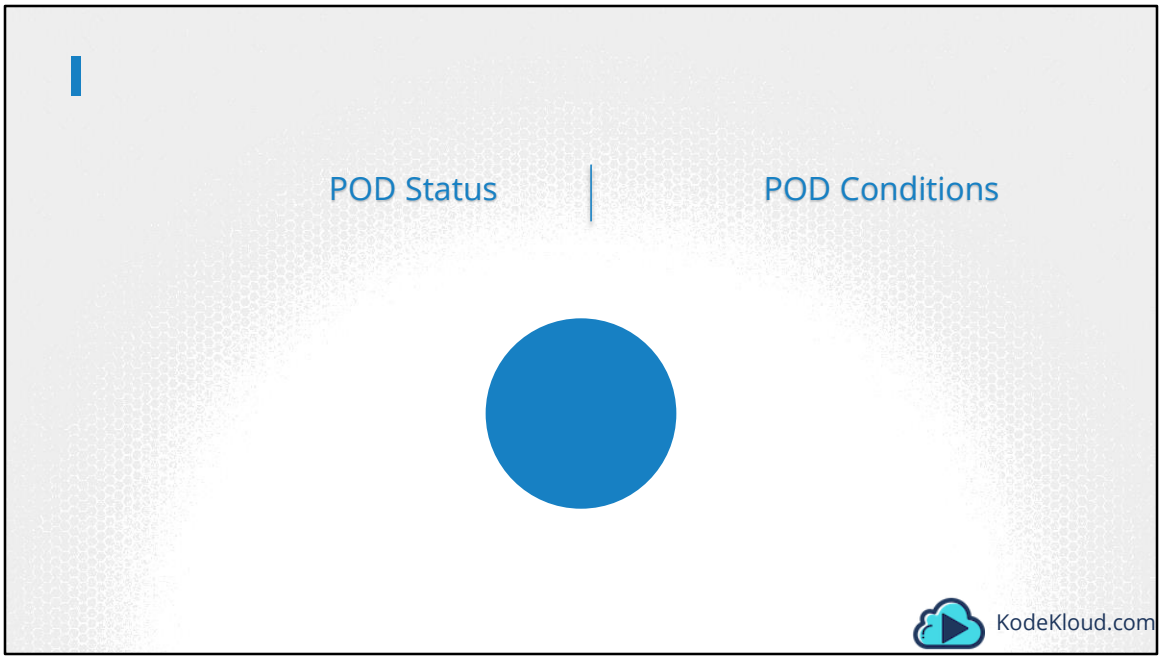
https://kubernetes.io/docs/tasks/access-application-cluster/communicate-containers-same-pod-shared-volume/

KodeKloud.com

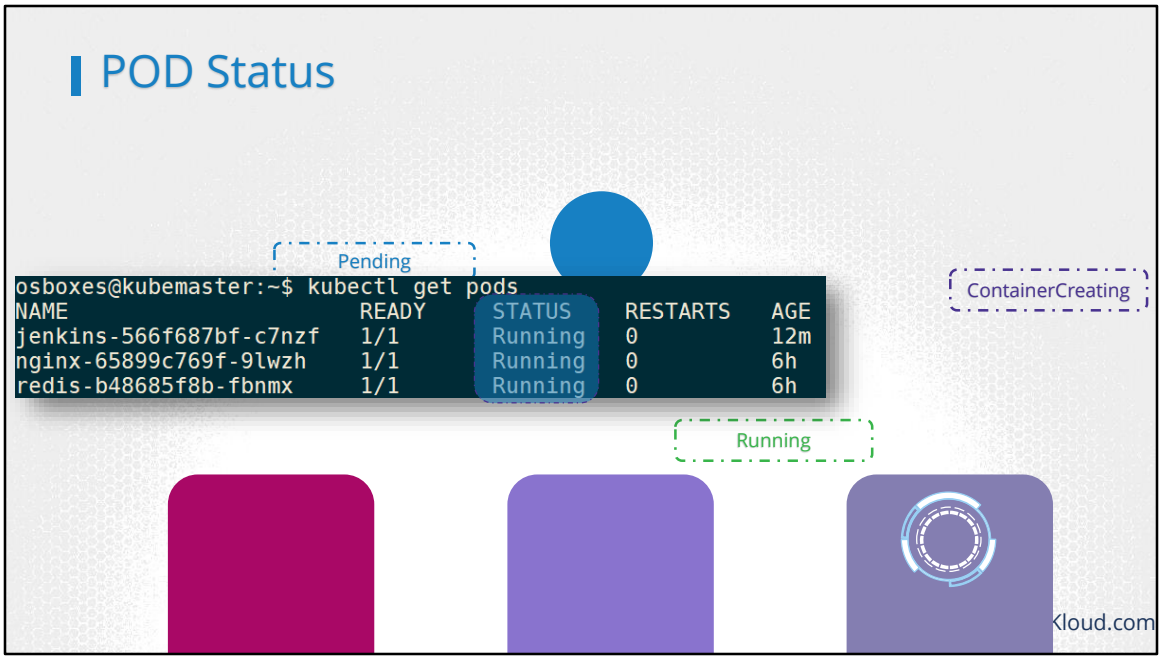


Hello and welcome to this section. In this section we learn about Observability in Kubernetes. We will discuss about Readiness and Liveness Probes, Logging and Monitoring concepts.

Let us start with Readiness Probes



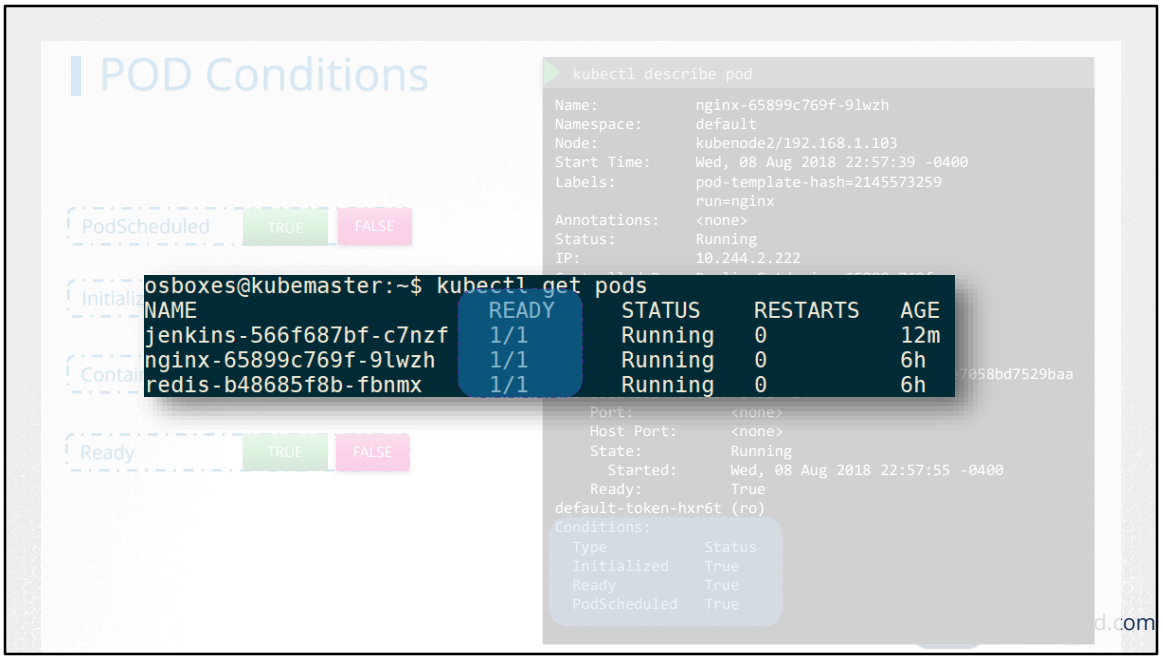
We discuss about Pod Lifecycle in detail in another lecture. However, as a pre-requisite for this lecture, we will quickly recap few different stages in the lifecycle of a POD. A POD has a pod status and some conditions.



The POD status tells us were 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 were 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, were the images required for the application are pulled and the container starts. Once all the containers in a POD starts, it goes into a running state, were 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.

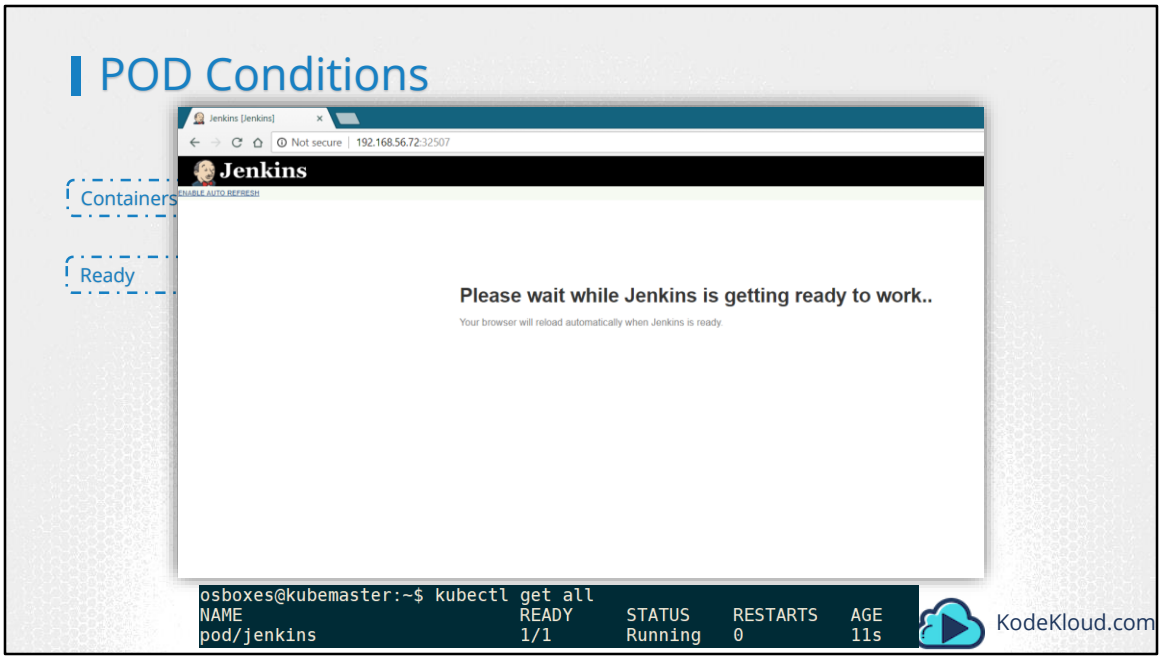


Conditions compliment POD status. It is an array of true or false values that tell us the state of a POD. When a POD is scheduled on a Node, the PodScheduled condition is set to True. When the POD is initialized, it's value is set to True. We know that a POD has multiple containers. When all the containers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

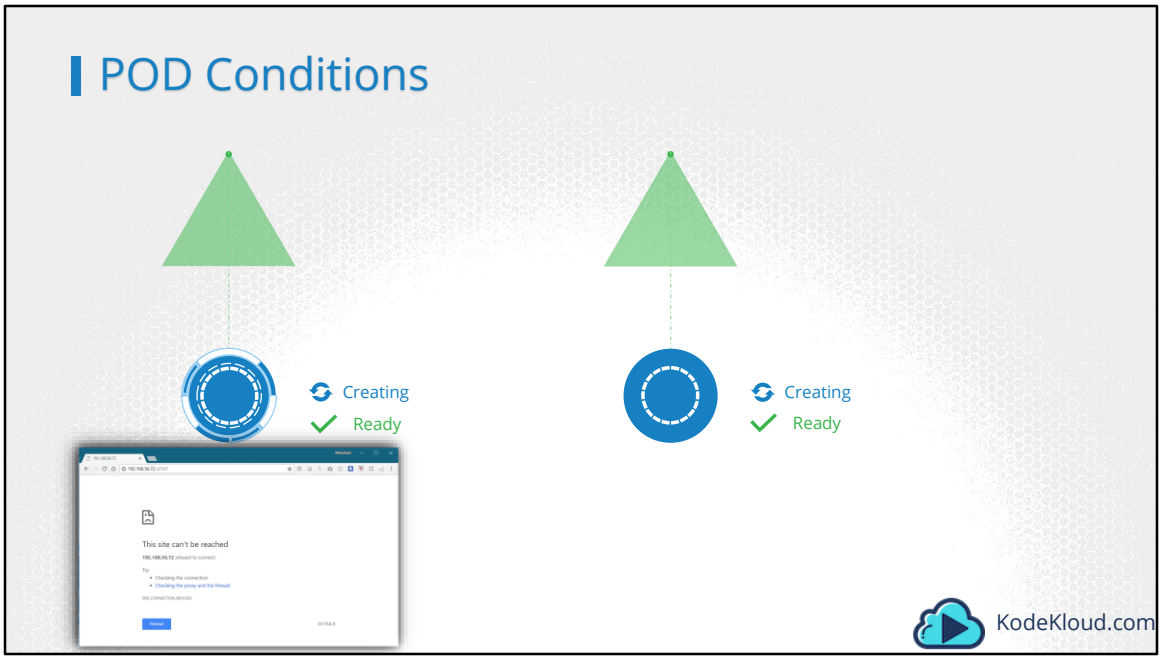
You can also see the Ready state of the POD, in the output of the kubectl get pods command.

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



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.

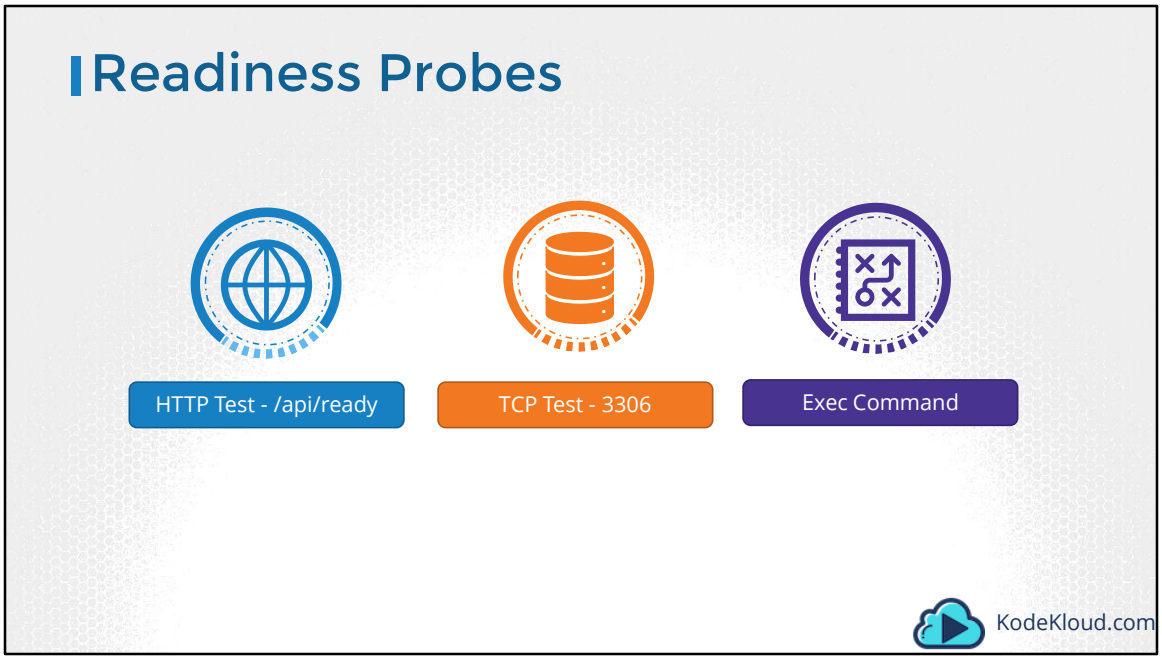
So why is that happening and how does kubernetes know weather that the application inside the container is actually running or not? But before we get into that discussion, why does it matter if the state is reported incorrectly.



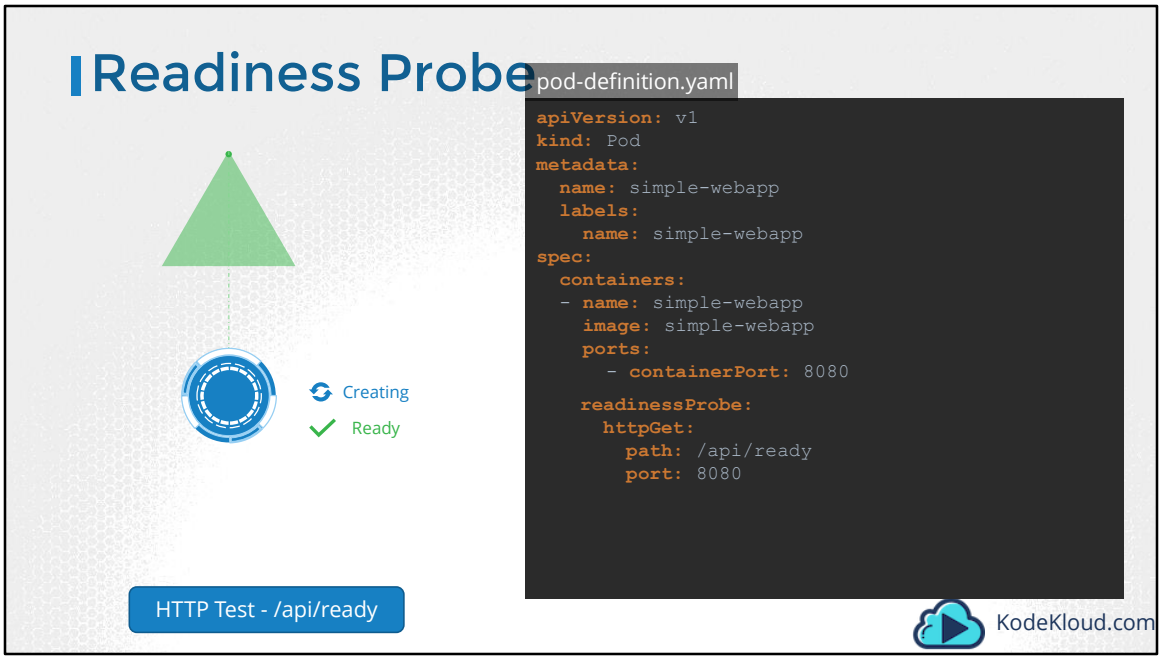
Let us look at a simple scenario were you create a POD and expose it to external users using a service. The service will route traffic to the POD immediately. The service relies on the pod's READY condition to route traffic.

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.

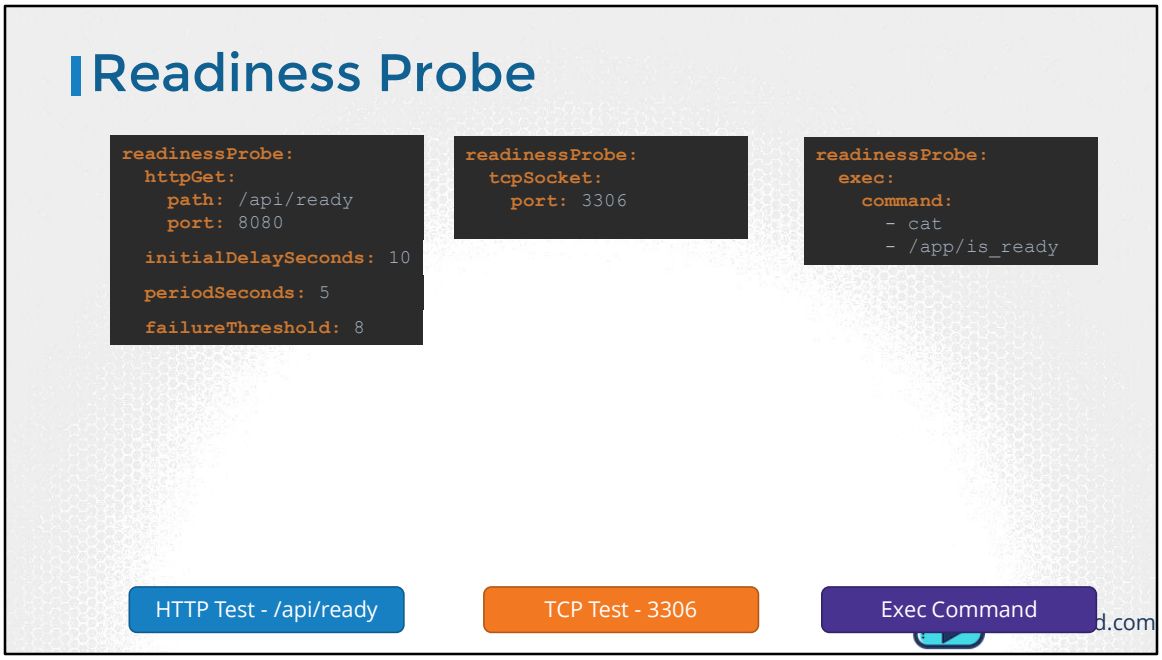
What we need here is a way to tie the ready condition to the actual state of the application inside the container. As a Developer of the application, YOU know better what it means for the application to be ready.



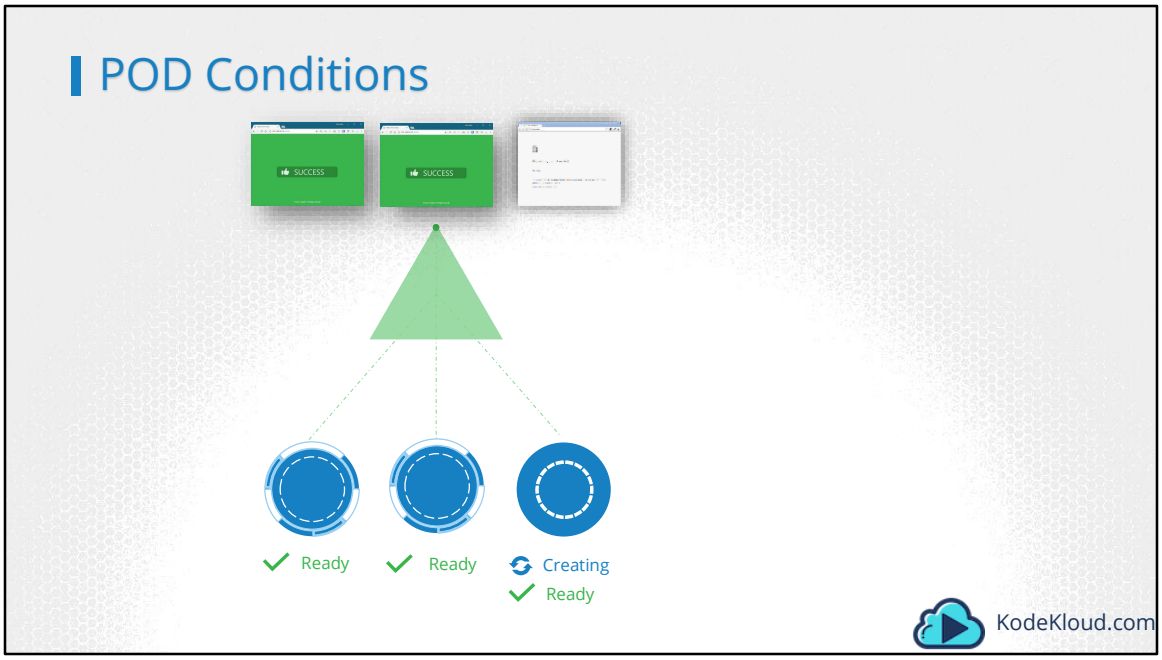
There are different ways that you can define if an application inside a container is actually ready. You can setup different kinds of tests or Probes, which is the appropriate term. In case of a web application it could be when the API server is up and running. So you could run a HTTP test to see if 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.



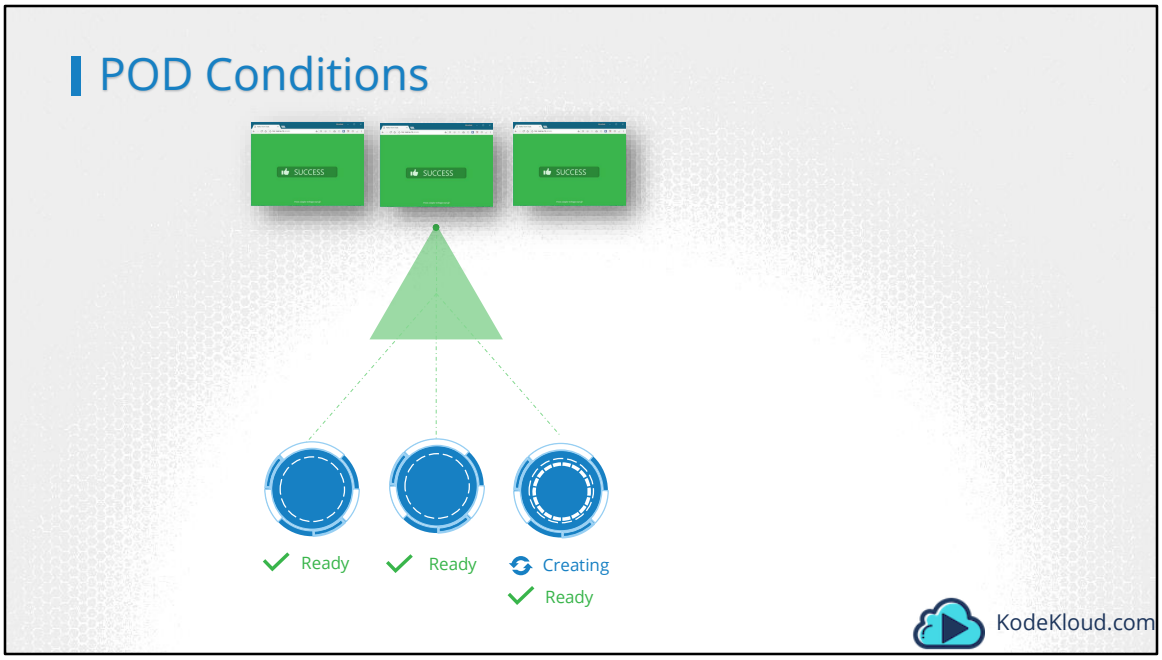
So how do you configure that test? In the pod definition file, add a new field called readinessProbe and use the httpGet option. Specify the port and the ready api. Now when the container is created, kubernetes does not immediately set the ready condition on the container to true, instead, it performs 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.



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.



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.



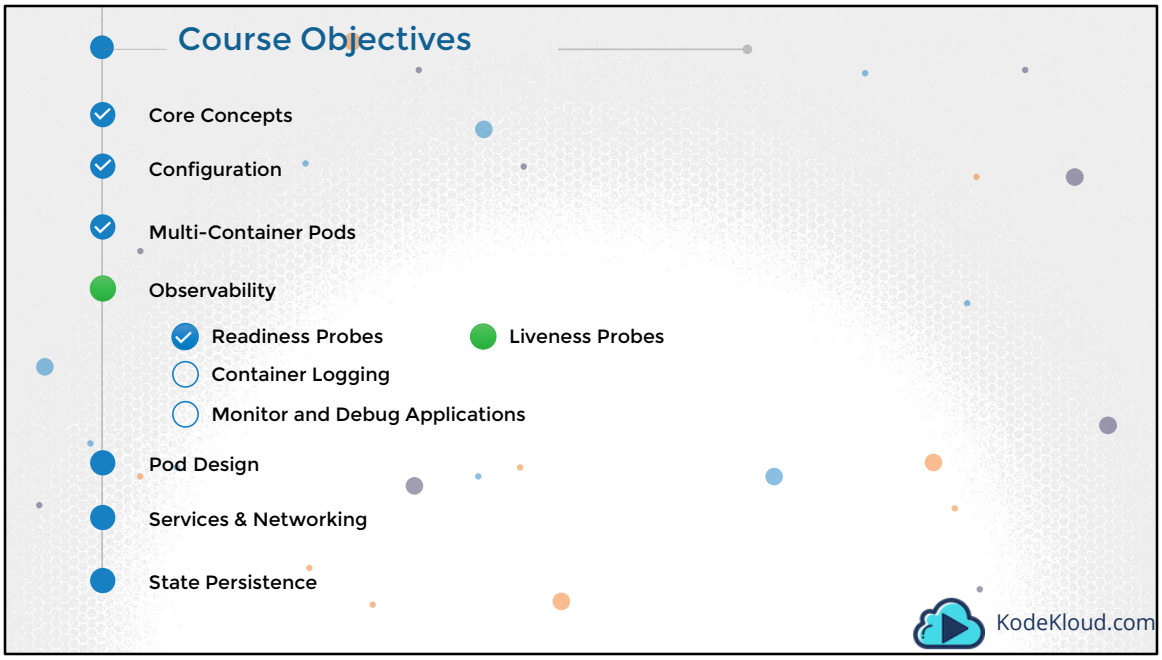
Instead if the pods were configured with the correct readinessProbe, the service will continue to serve traffic only to the older pods and wait until the new pod is ready. Once ready, traffic will be routed to the new pod as well, ensuring no users are affected.

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

References

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

KodeKloud.com



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.