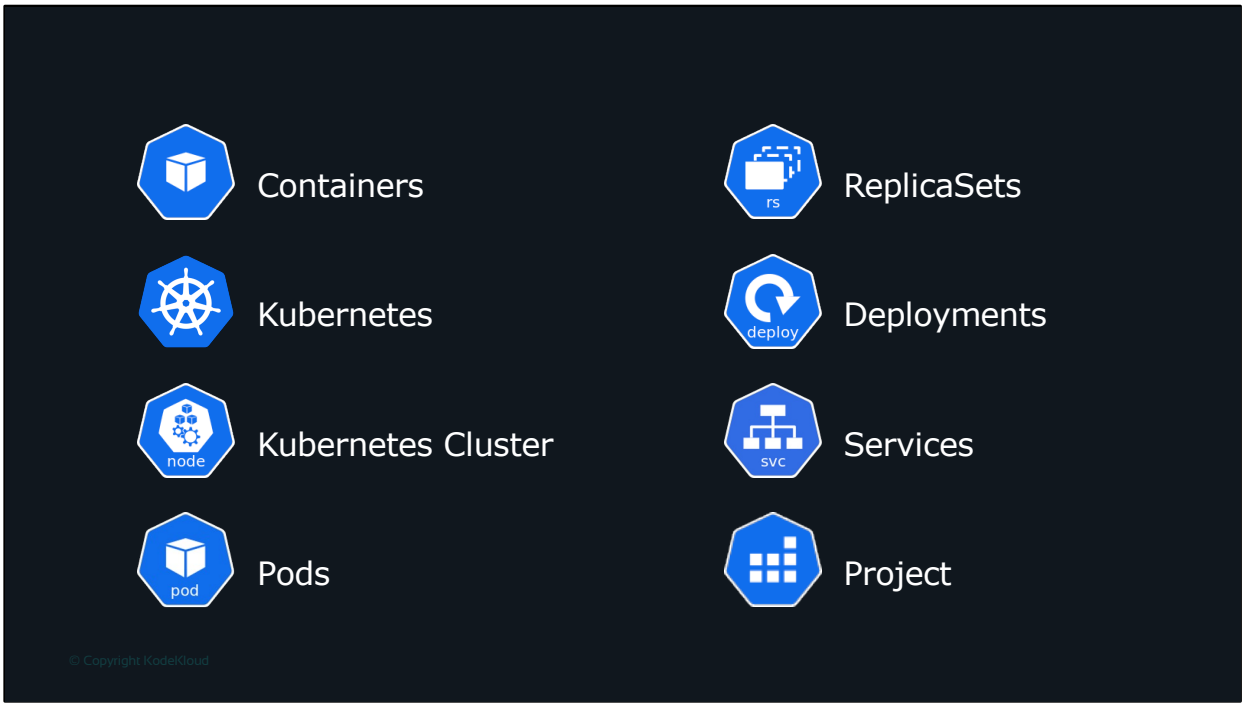
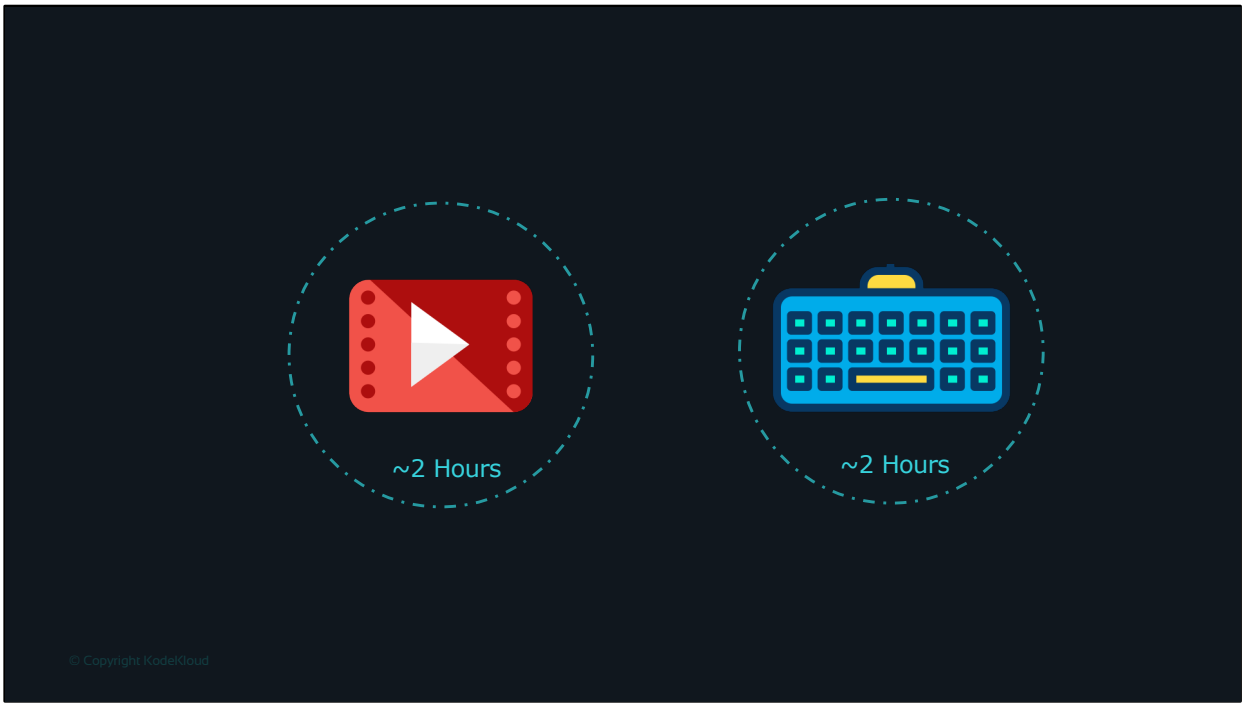
In this course you are going to learn the absolute basics of Kubernetes.

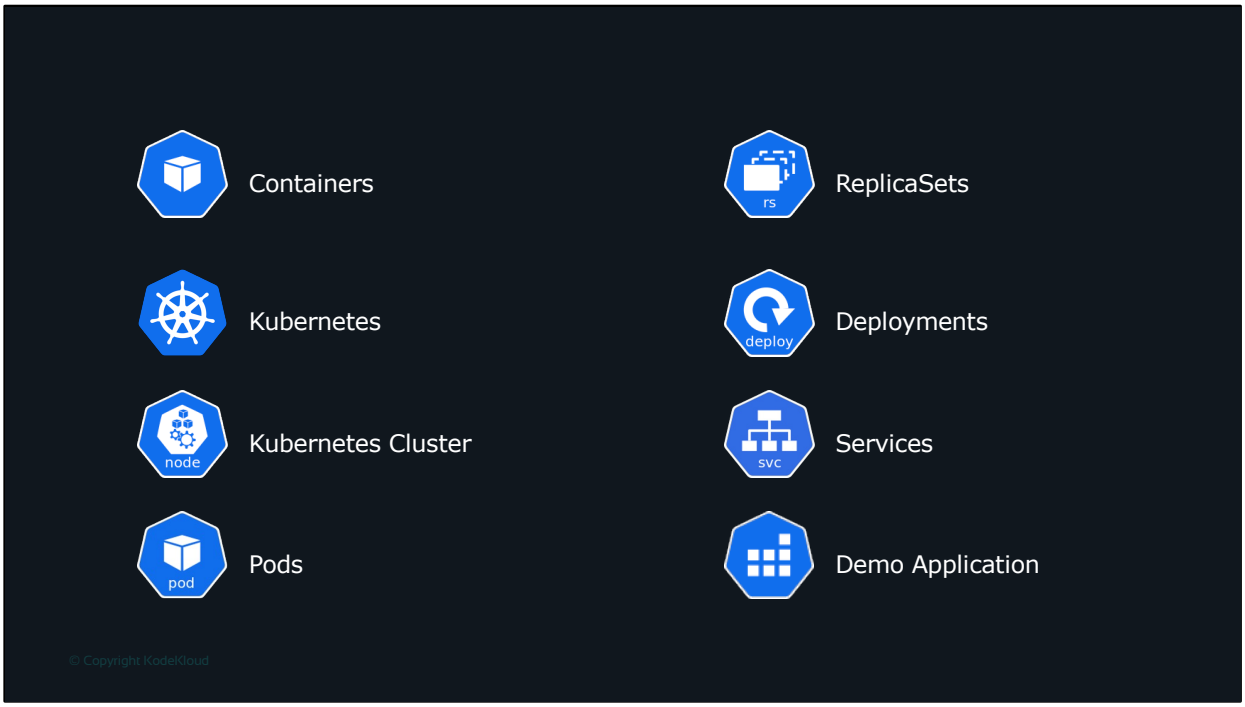
Access Labs at:
https://kode.wiki/kubernetes-labs



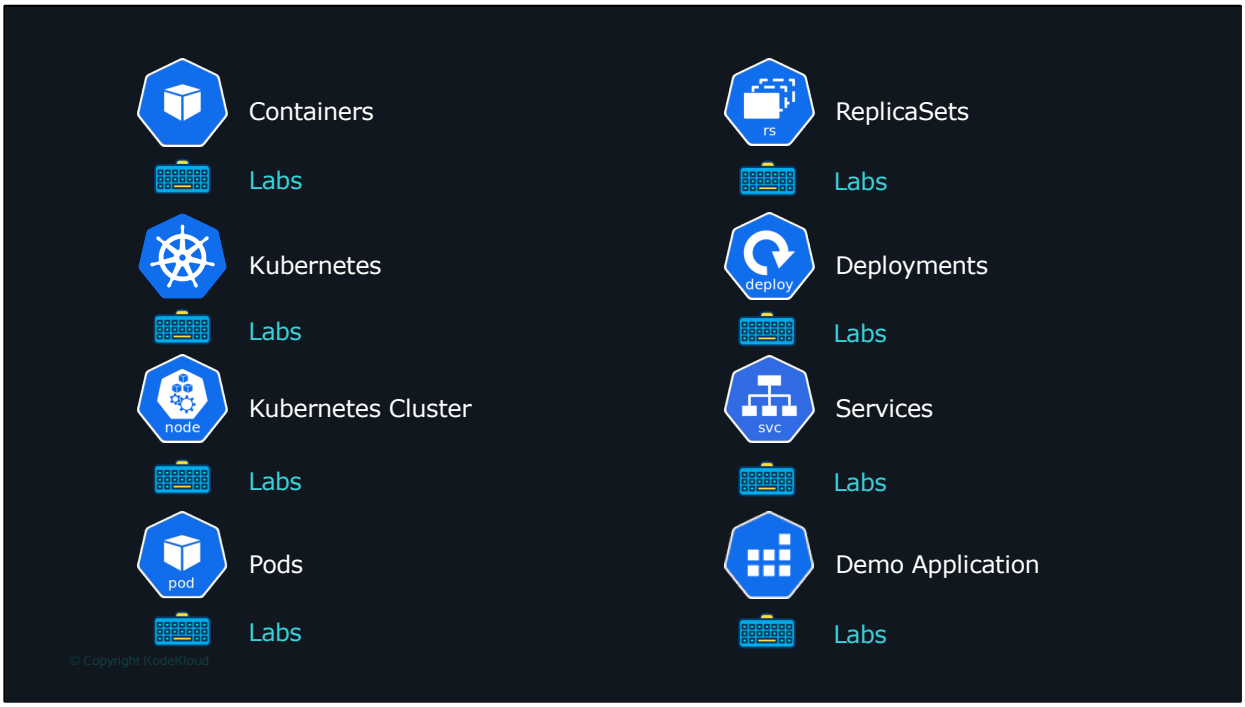
We'll start with a quick introduction to containers, and then we'll understand why you need container orchestration, what Kubernetes is, and then dive into Kubernetes concepts such as Pods, ReplicaSets, Deployments, Services and finally a project on deploying a micro-services application to a Kubernetes cluster.



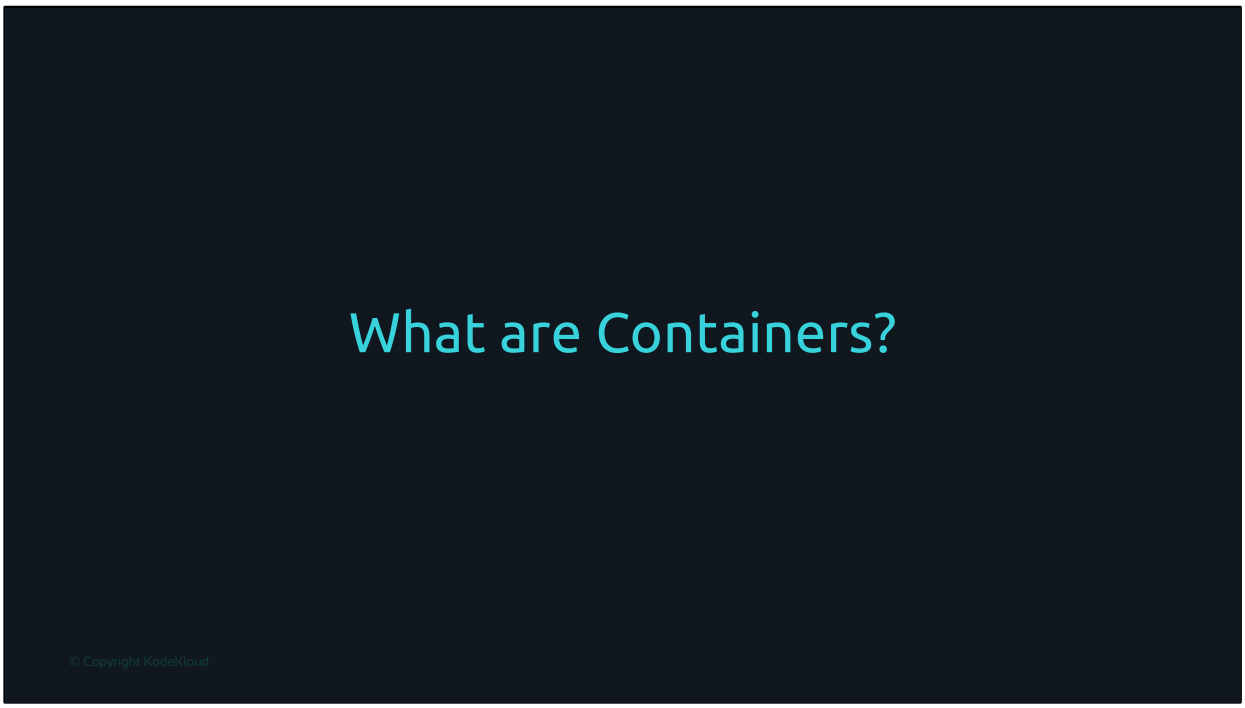
So here's how I recommend you to take this course. This course is about 2 hours of video and 2 hours of lab time. By the end of this course you should aim to get a high level understanding of Kubernetes, not just theory but with hands-on practice.



Each concept taught in this video

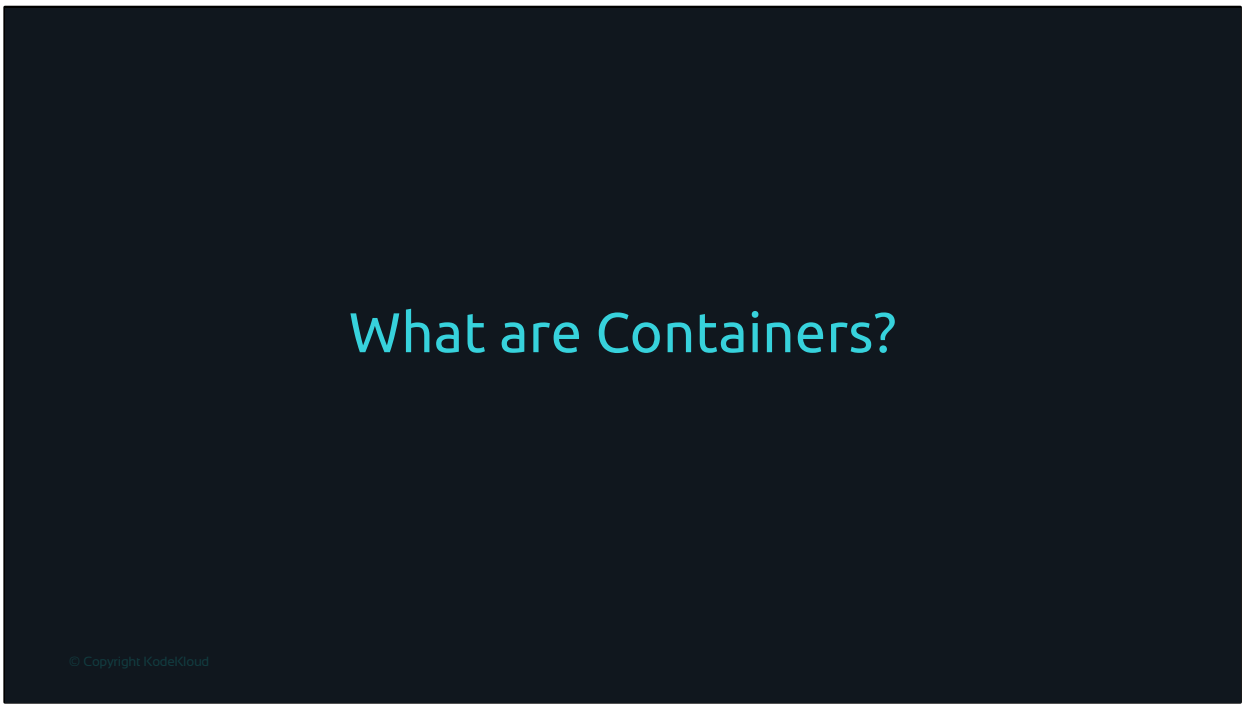


is followed by hands-on labs.

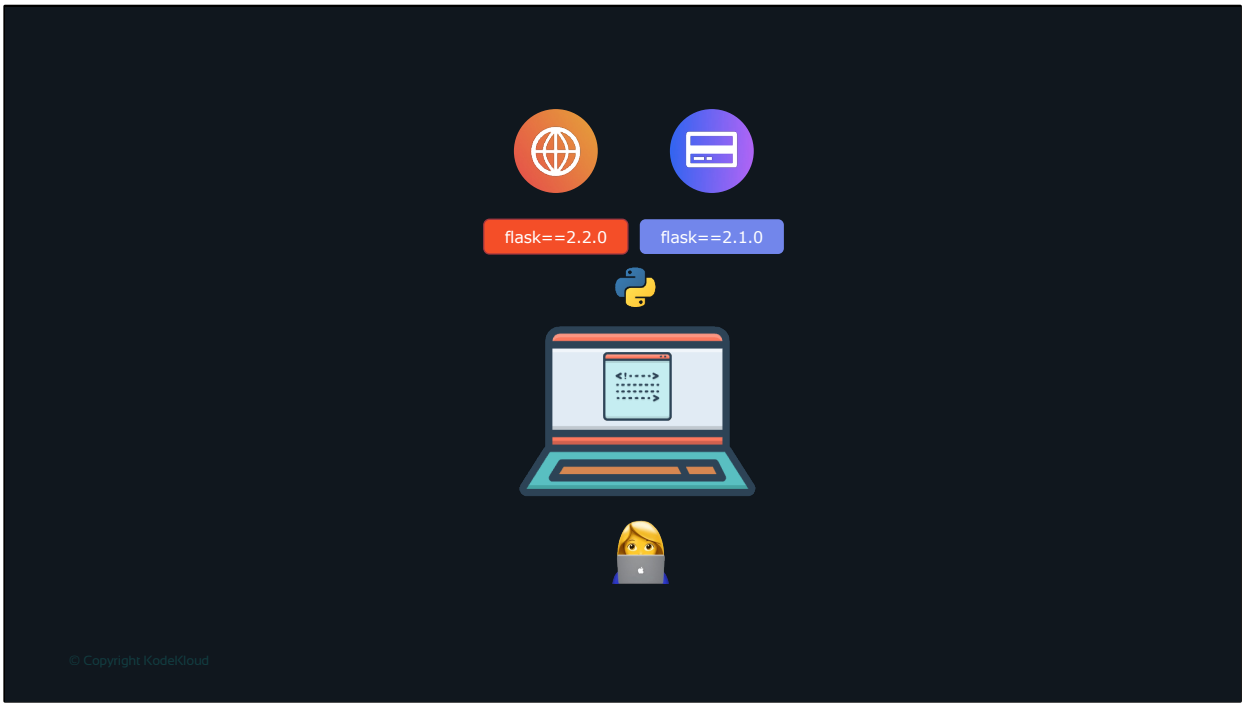


Let's start by refreshing our memory on containers.

So before we begin, head over to this link to download the deck used in this course, so you can keep a local copy for your own reference, as well as to access the labs that comes free with this course. Go to kode.wiki/kubernetes-labs. Or scan this QR code.

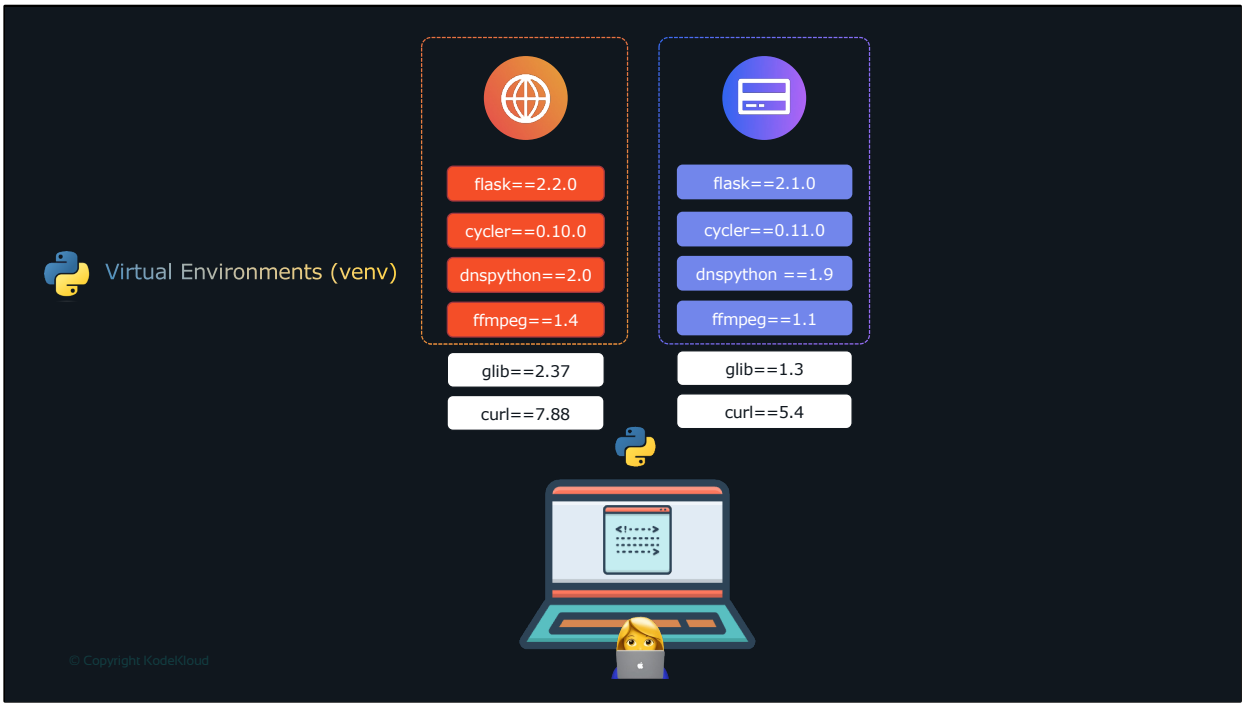


Let's start by refreshing our memory on containers.



So that's you and you are developing an application on your laptop. Say it's written in Python. It has certain dependencies such as the flask framework for serving a website. You are also working on another part of the application – say a payment service that also uses the flask framework, but relies on a different version of the library.

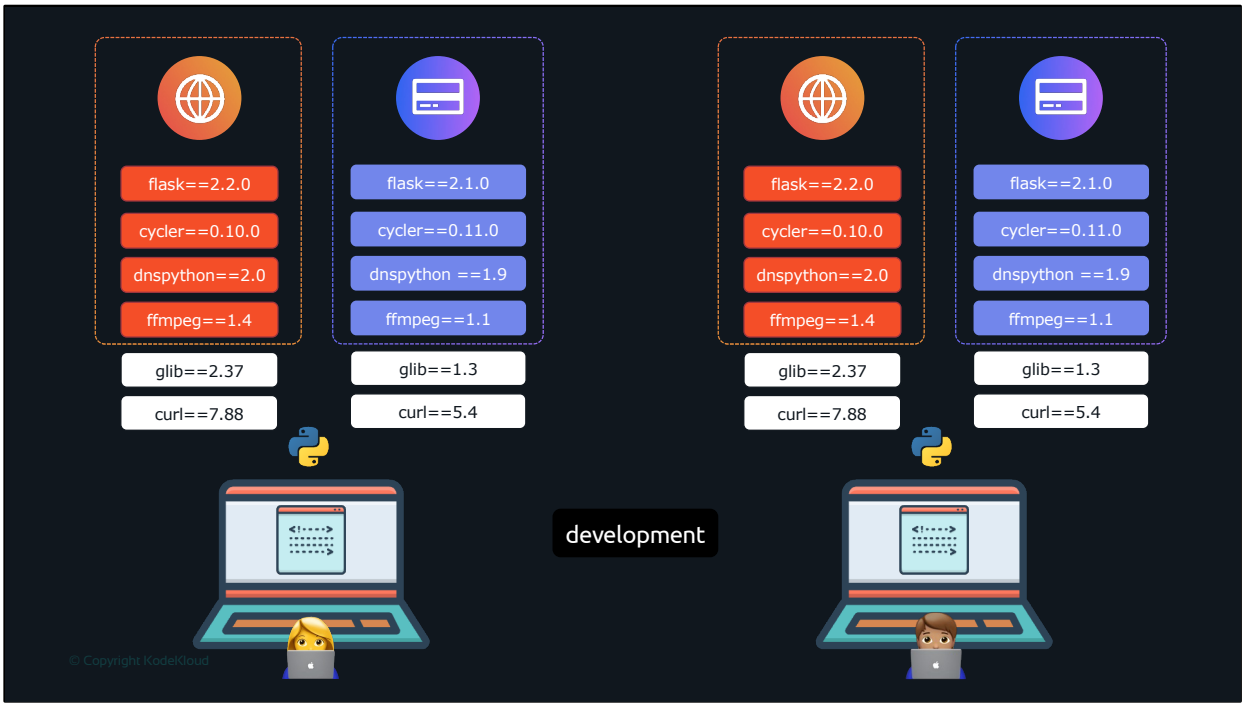
Applications typically add many such packages as they grow. And they may all be different in different applications. That's going to be a challenge if you are trying to run the same application on the same machine. Now certain programming languages provide solutions to tackle these. For example Python has the Python Virtual Environments concept that helps isolate python packages into virtual environments. That way you can have different versions of dependencies in different virtual environments. However that doesn't help you separate dependencies outside of the programming language libraries. For example, your app may rely on a specific package on the operating system, such as a



say the glib system library or utilities like curl. What if your application relies on certain binaries and packages on the system that require different versions? It's going to be hard to manage different versions of dependencies on the same OS.

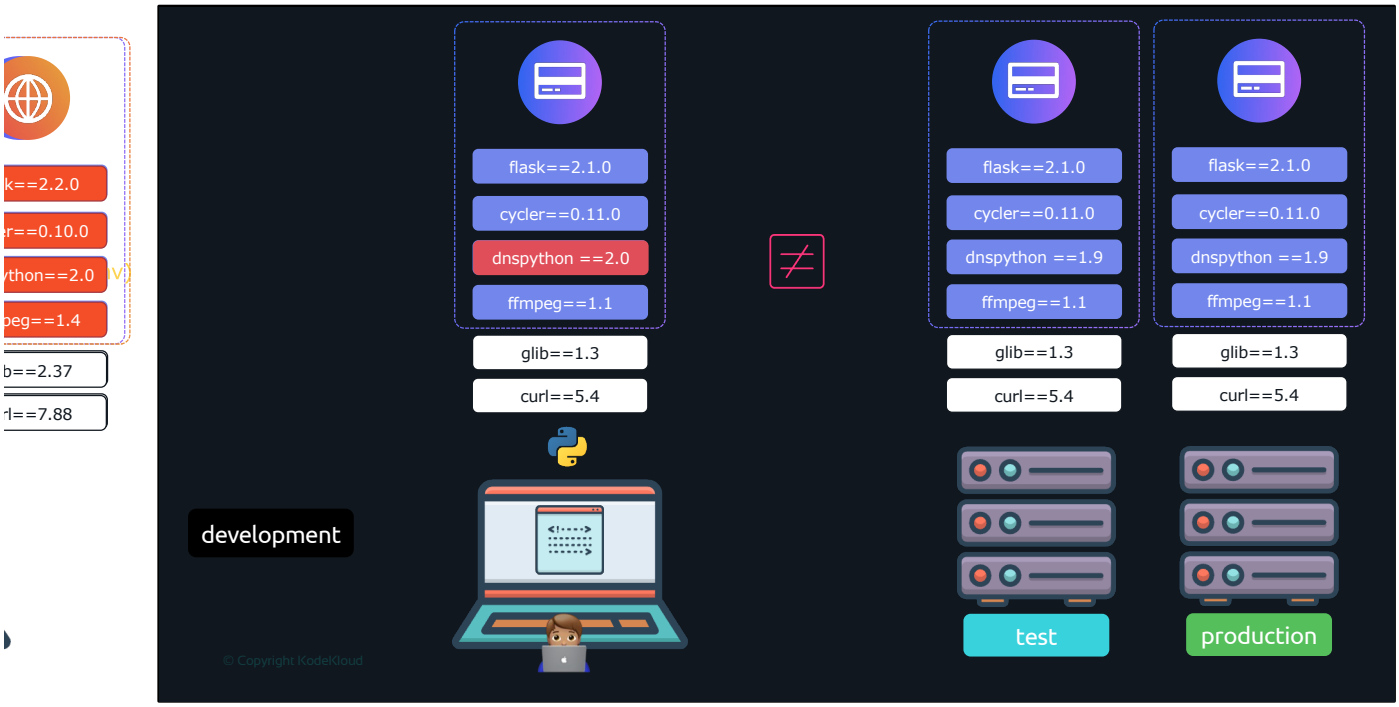
Moreover, let's say you bring in a friend to help you develop the application.

nents (venv)



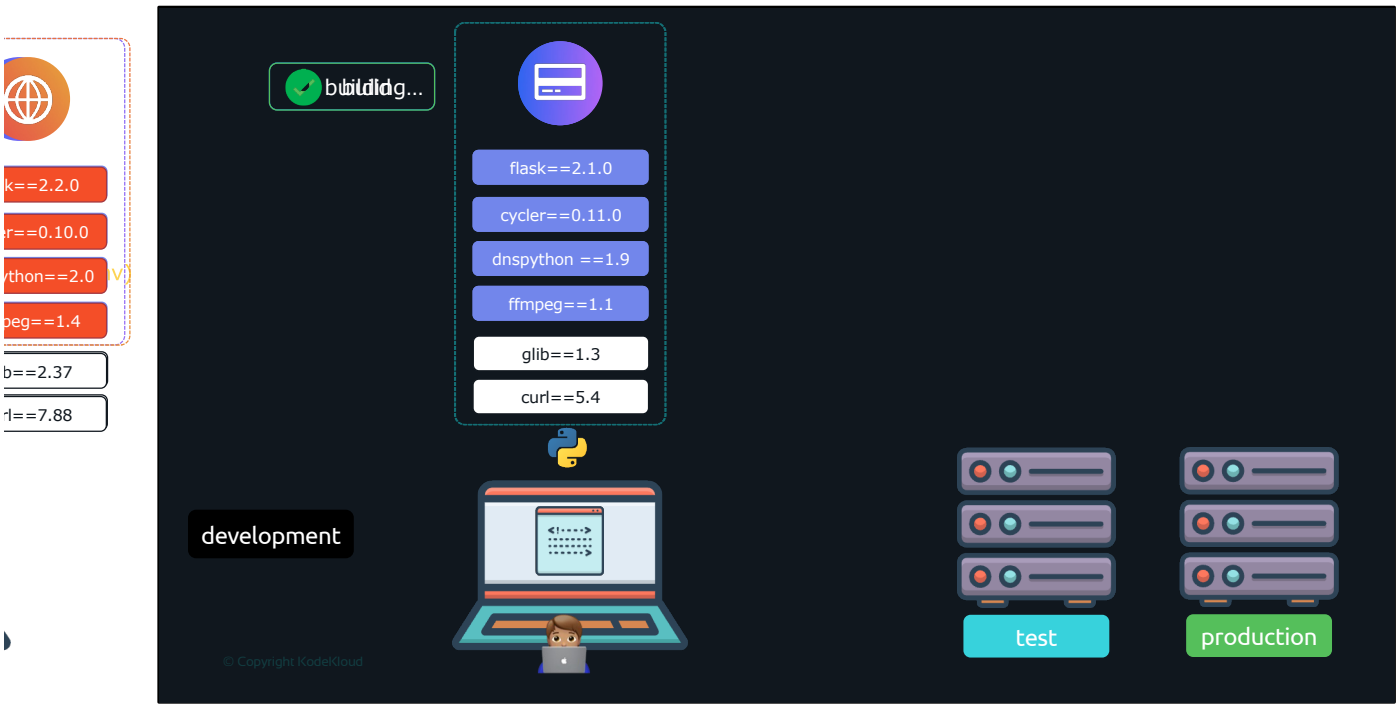
That person need to setup the exact same environment in the exact same way with the exact same versions of dependencies and libraries. And if the other person uses a completely different operating system, then a whole different nightmare awaits. Now you'll need to figure out what the different dependencies are for that operating system.

Now we have only been talking about development environment.



What happens when the application is deployed to a test environment or to a production environment. You'll have to make sure you setup the environment in the exact same way with the exact same dependencies. If you change something in the <c> development environment you'll have to make sure it gets updated in the test and prod environments.<c> Otherwise things end up working in one environment and not working in the other.

And no matter how much you try it's impossible to make sure these environments remain the same. At some point in time, someone is going to make a change to a dependency or add another dependency and forget to update them in the other environments and things are going to break.



What if you could build an image that consists of the app itself and all of its dependencies at both the app level and system level and package it…

And use the same exact image in all of the different environments? <c> That way anytime you make a change in the future, the image is rebuilt…

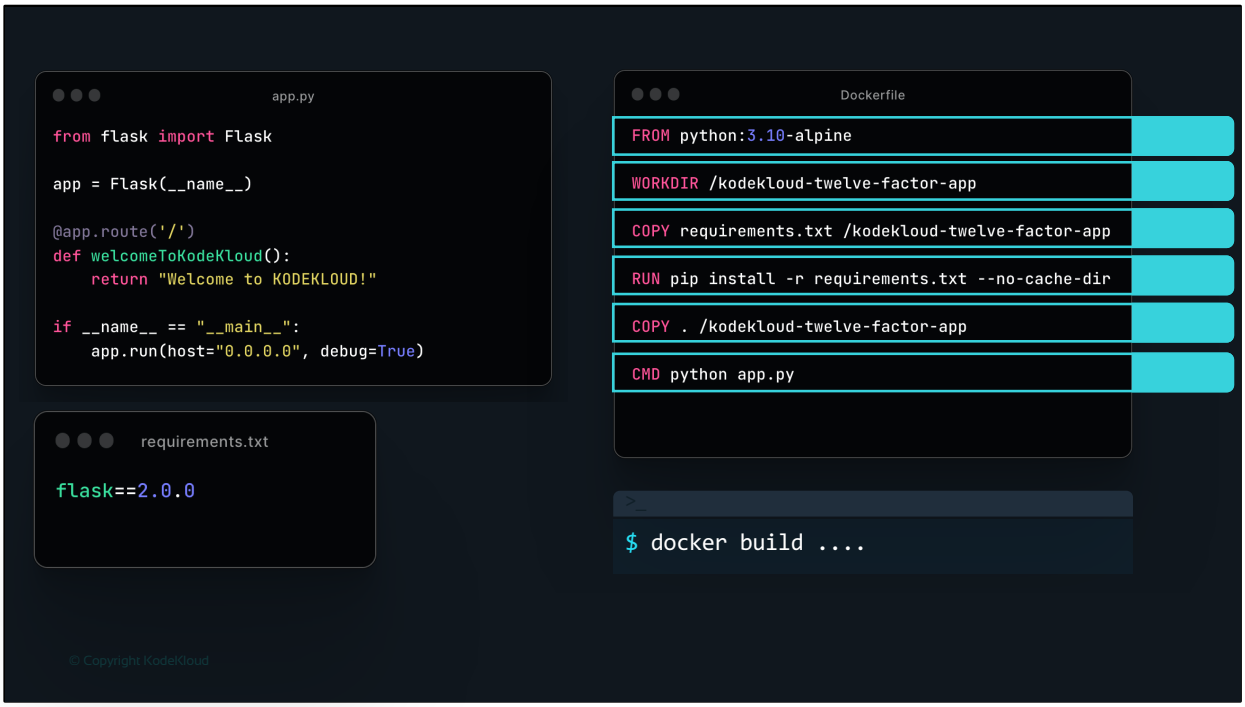
and the same image is used in all of the different environments. No more differences between different environments.

So that's what containers can help us with.

Containers help us create isolated environments on our systems to run applications completely isolated from each other. <c> You could run a different web application with different versions of dependencies or a PostgreSQL server or a MySQL server or Redis server. All on the same system with their own libraries and dependencies. But not worrying about any impact to each other. Each of these maybe based on different operating systems too.

Containers allow you to build images based on specific operating systems, then add all system level and application level dependencies to it to finally have a lean and clean image for each application that can run anywhere. On your linux machine you can run any of these applications even if they are based on a different OS flavor.

And one of the most popular tools to containerize applications and run containers is Docker.

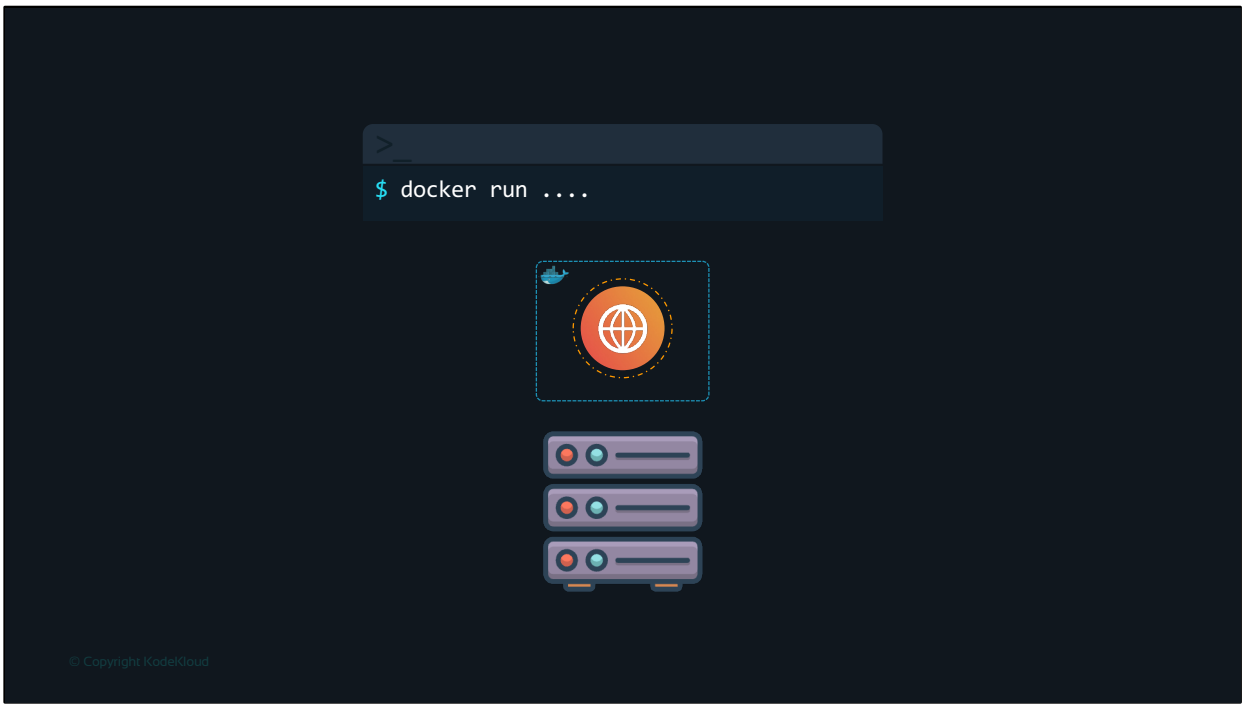


So here we have our application code, the requirements.txt file with the list of dependencies and we now build a Dockerfile to package the application with it's dependencies into a Docker container.

<c> The first line creates an image from the python base image. <c> Then sets the right working directory. <c> Then copies the requirements.txt file to the working directory. <c> And then installs the dependencies. This is where you could add any other

dependency to it<c> And then copies the application code into the image. <c> And finally defines the command to run the application using the CMD instruction.

Now by running the docker build command we build an image.

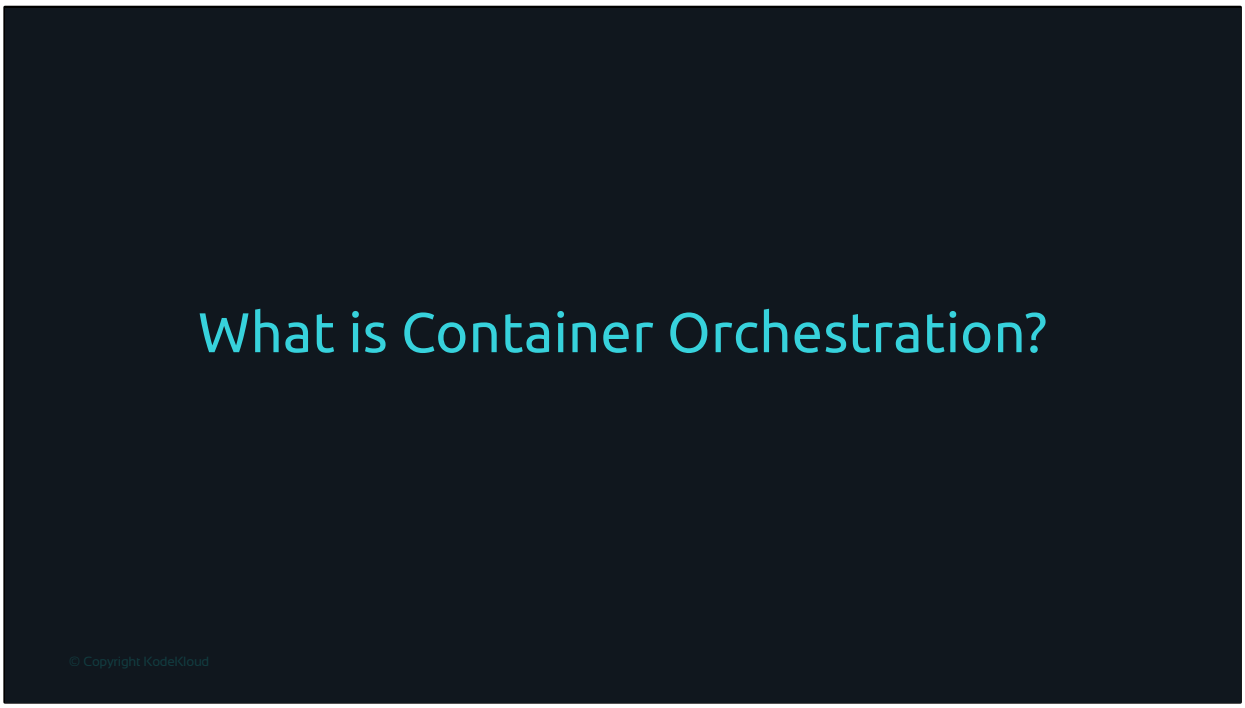


and by running the docker run command we run one instance of our application.

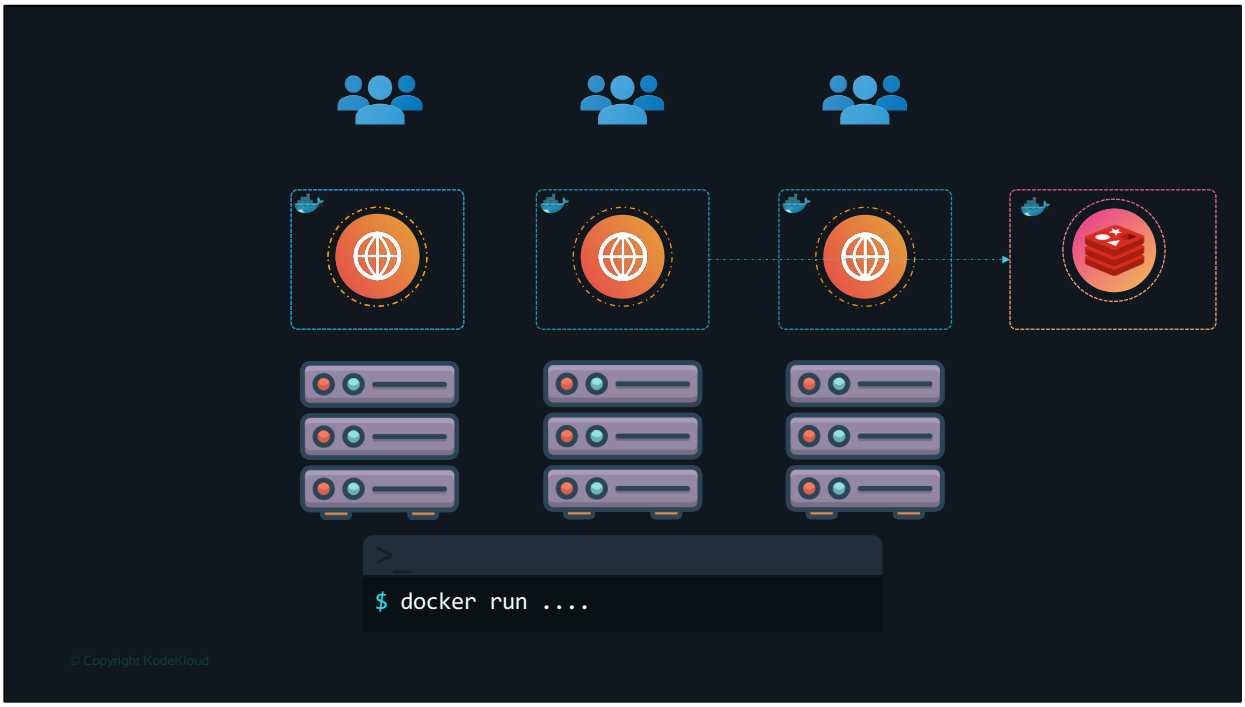
So that was a quick introduction to containers and Docker.

If you are new to Docker I would recommend checking out our free Docker for Beginners course on KodeKloud using the link given here. You'll learn with hands-on labs using our interactive learning environment by working on real systems exclusive to you.

At any time during this course if you feel you need assistance, head over to our community group. We have a thriving community on slack where our instructors and teaching assistances hang out. So go to kode.wiki/community or scan this QR code to get an invite. Explore the various channels available for learning different topics and feel free to post your questions in the respective channels.



So we have learned about containers, let us now see what Container Orchestration is.



By running the docker run command we were able to run one instance of our application. This instance can serve one set of users.

<c> But what if users increase? We want to be able to run more instances.

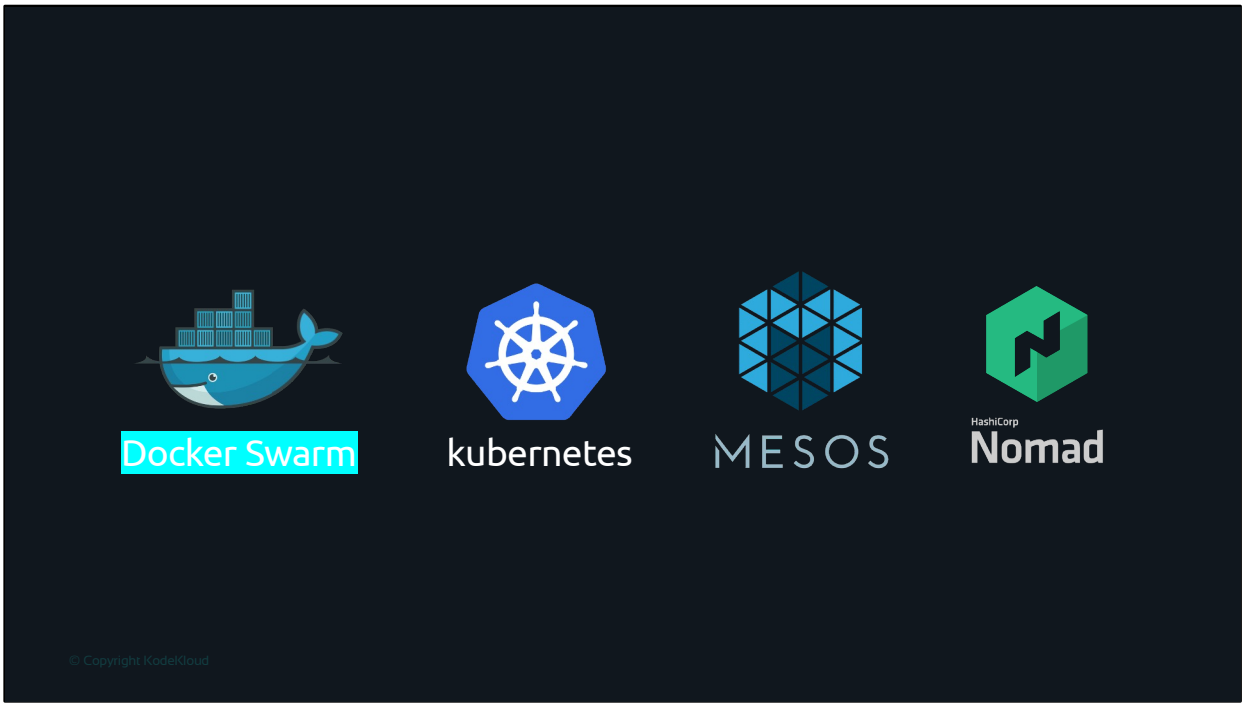
<c> If one instance crashes for some reason we want it to be able to automatically restart.

If there isn't enough resources on the server, we want to be able to scale out by adding more servers. And then run more instances of our application on those servers.

<c> And when users decrease, we want to be able to remove instances and reduce the number of servers.

So that's what container orchestration is at a high level. We want to be able to orchestrate multiple containers to scale up or down and also define the relation ship between the containers declaratively.

<c> If our app is dependent upon on a database container orchestrators can help us define that relationship.



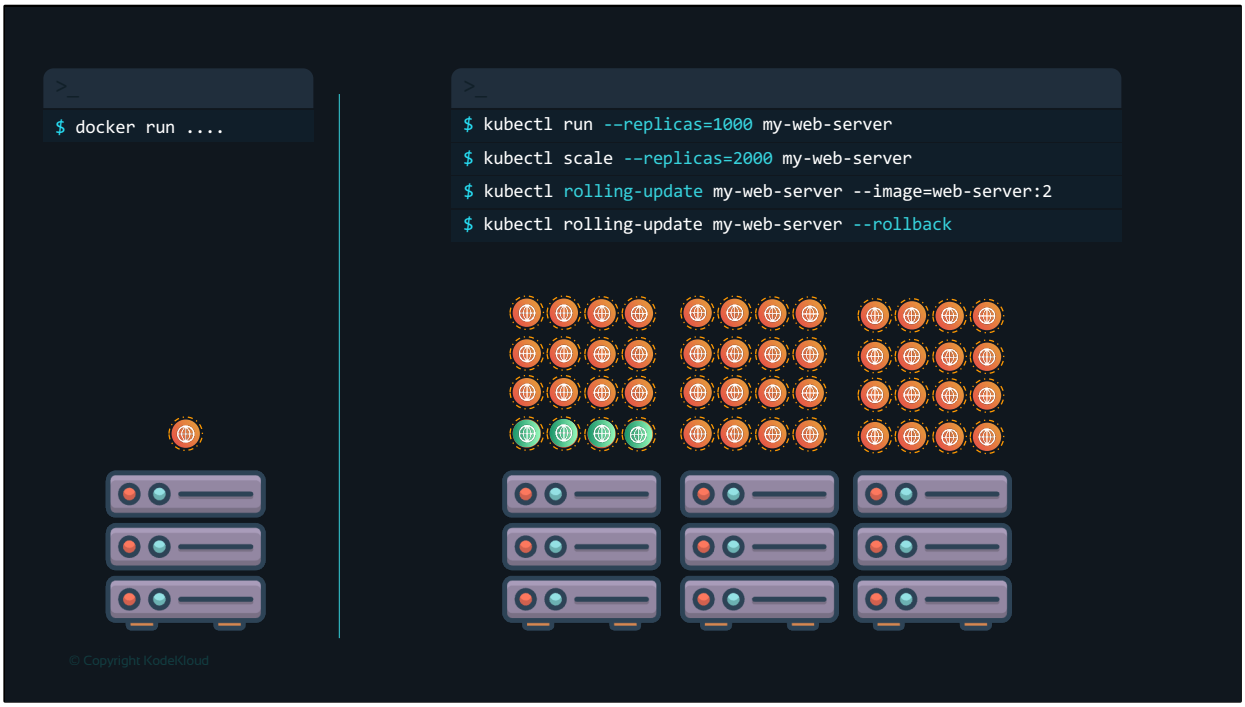
There are different container orchestration solutions out there today – such as Docker Swarm, Kubernetes, Mesos and now Nomad to name a few. Kubernetes is the most popular of it all and

In the rest of the video we will focus on Kuberentes.

Swarm

MESO

So what is Kubernetes?



With docker you were able to run a single instance of an application using the docker run command. Which is great! Running an application has never been so easy before.

With kubernetes, using the kubernetes CLI known as kubectl, <c> you can run a 1000 instance of the same application with a single command. <c> Kubernetes can scale it up to 2000 with another command.

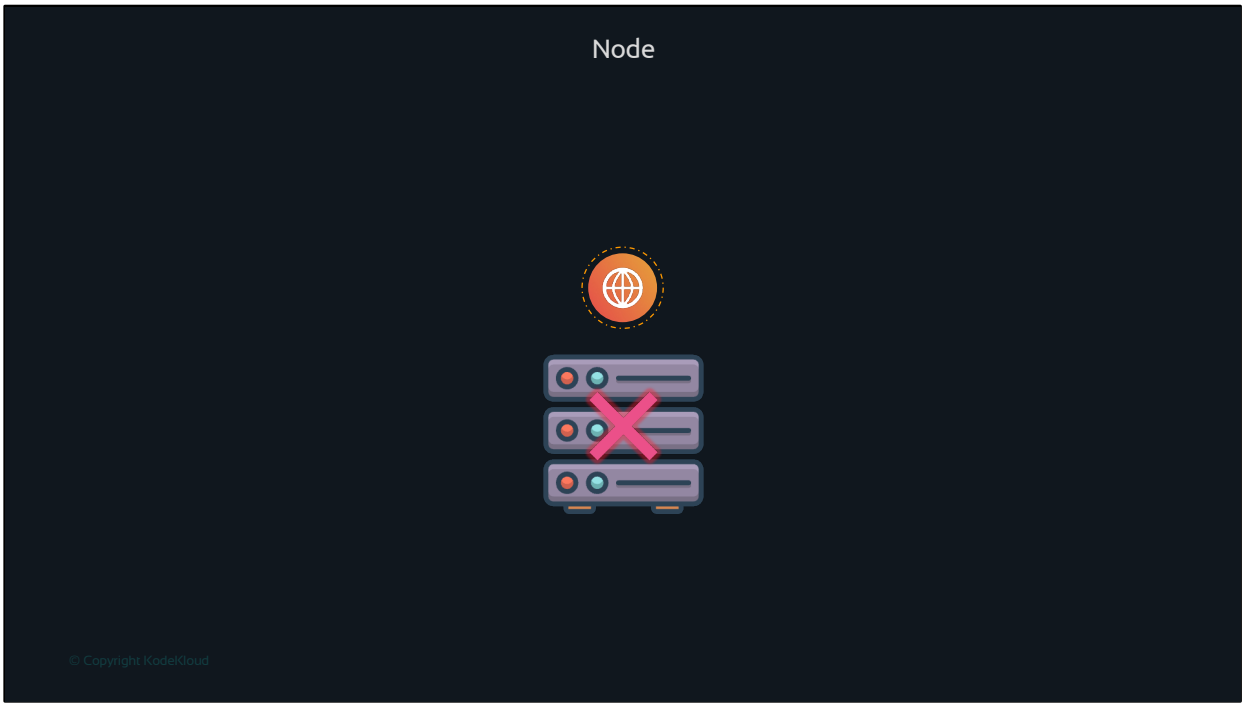
Kubernetes can even be configured to do these automatically so that instances and the infrastructure itself can scale up and down based on user load.

<c> Kubernetes can upgrade these 2000 instances of application in a rolling fashion one at a time, with a single command. <c> If something goes wrong, it can help you roll back these images with a single command.

<c> Kubernetes can help you test new features of your application by only upgrading a percentage of these instances through A/B testing methods.

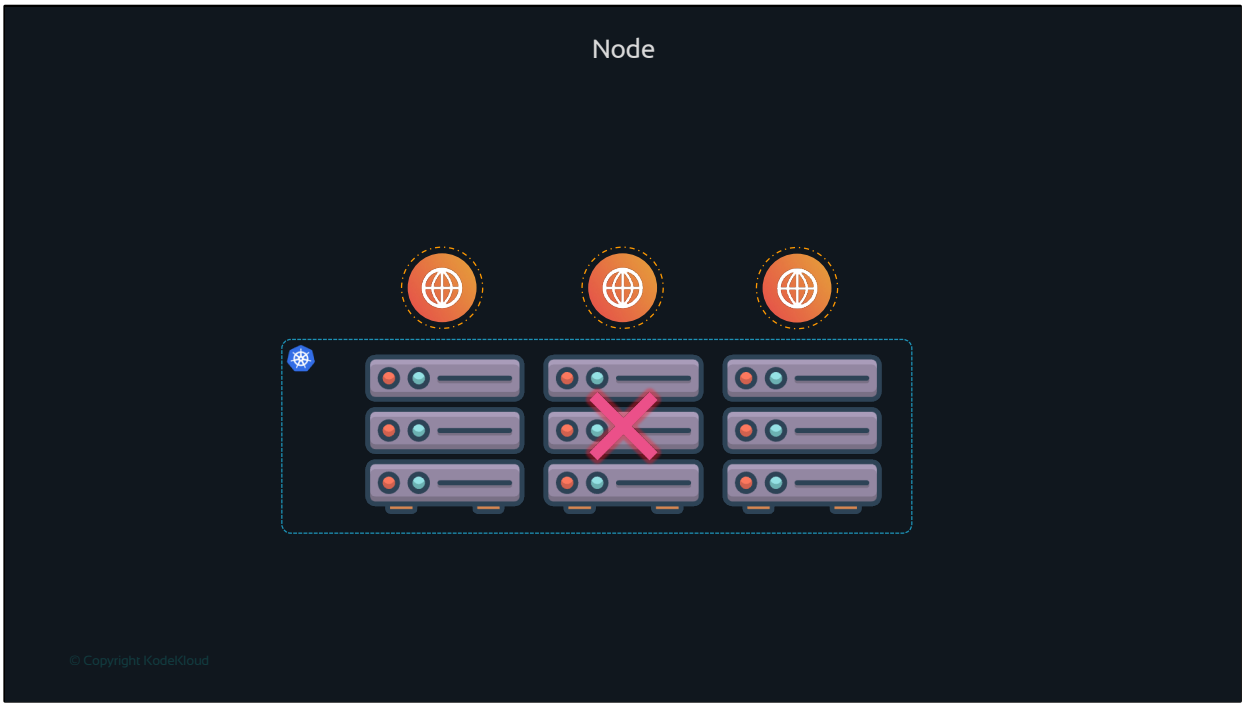
Don't worry about the command line tool for now. We will take a closer look at it soon.

With Kubernetes you are able to define the expected state of your application. For example you are able to define that your application consists of 4 different services. The webserver must have 3 instances running, the payment service to have 2. There should be a redis service with 3 instances running and a database service to which these services connect to. And you are able to define these in code and Kubenetes will ensure that the state you have defined for your application is maintained at all times.



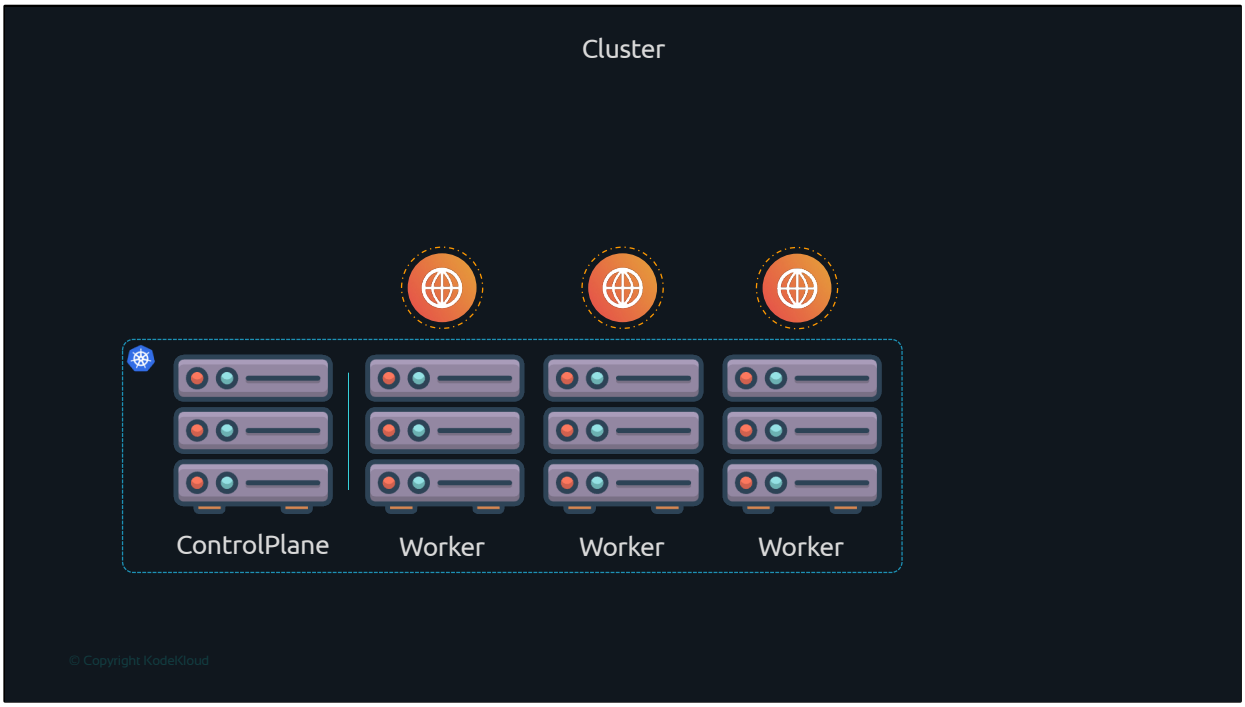
Let's us understand the basic components in a Kubernetes Cluster first. Let us start with Nodes. A node is a machine – physical or virtual – on which kubernetes is installed. A node is a worker machine and this is were containers will be launched by kubernetes.

<click>
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. <click>

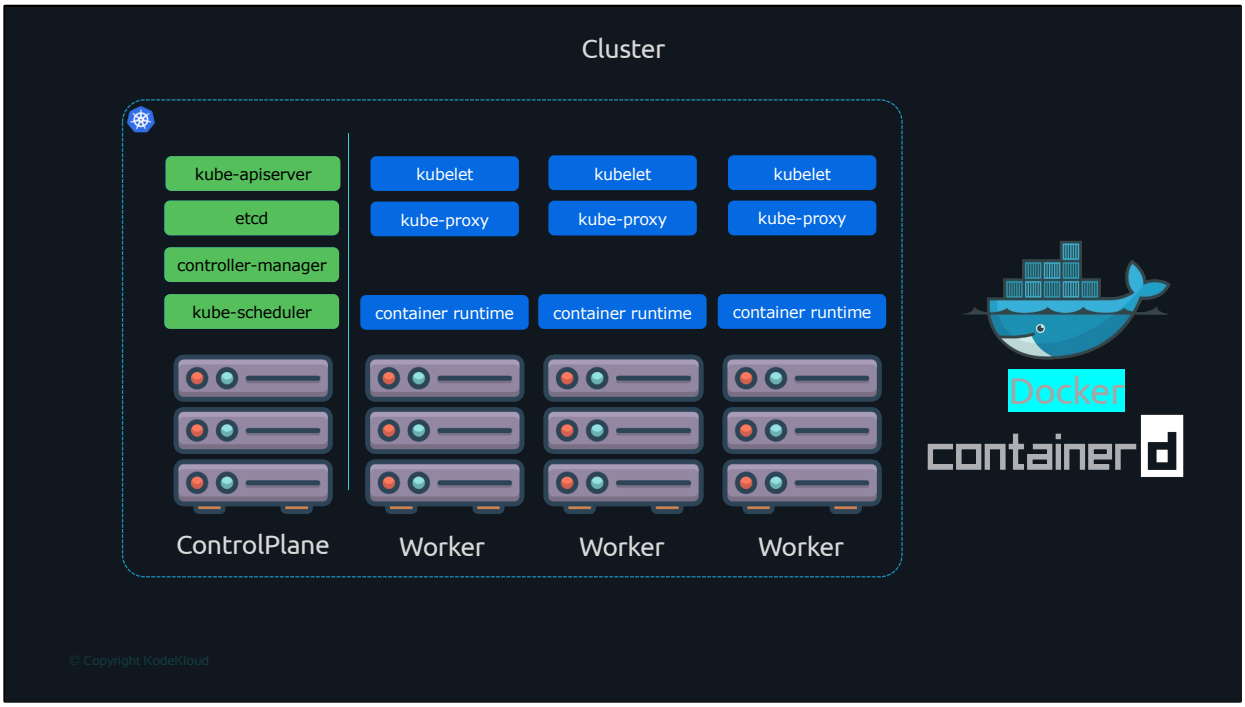


<click>
A cluster is a set of nodes grouped together. This way even if one <c> node fails you have your application still accessible from the other nodes. Moreover having multiple nodes helps in sharing load as well.

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 ControlPlane comes in. Also previously known as the master node.



The controlplane is another node with Kubernetes components installed in it. The controlplane 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. 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. This is where information about the nodes in the cluster, the application running on the cluster are stored.

<click><click>
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.

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

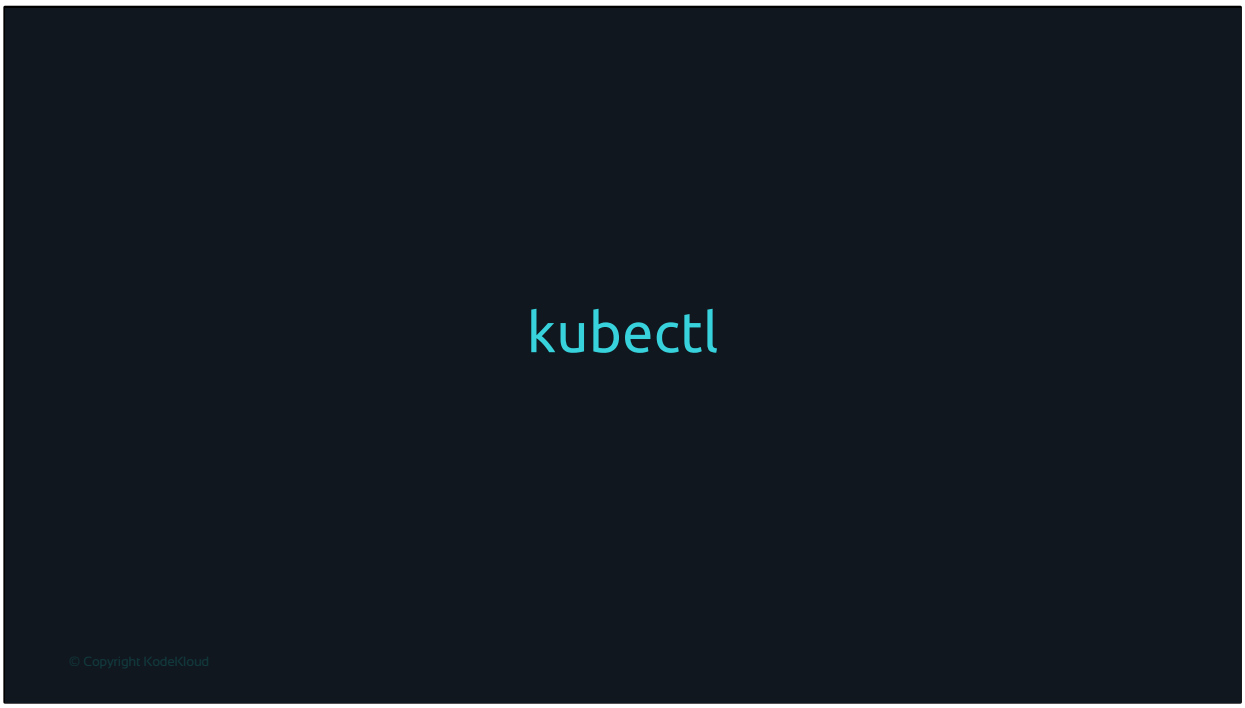
<click><click>
On the worker nodes you have the kubelet which is the agent that runs on each node in the cluster. The agent is responsible for making sure that the containers are running on the nodes as expected.

You also have the kube-proxy that is responsible for maintaining rules on the nodes.

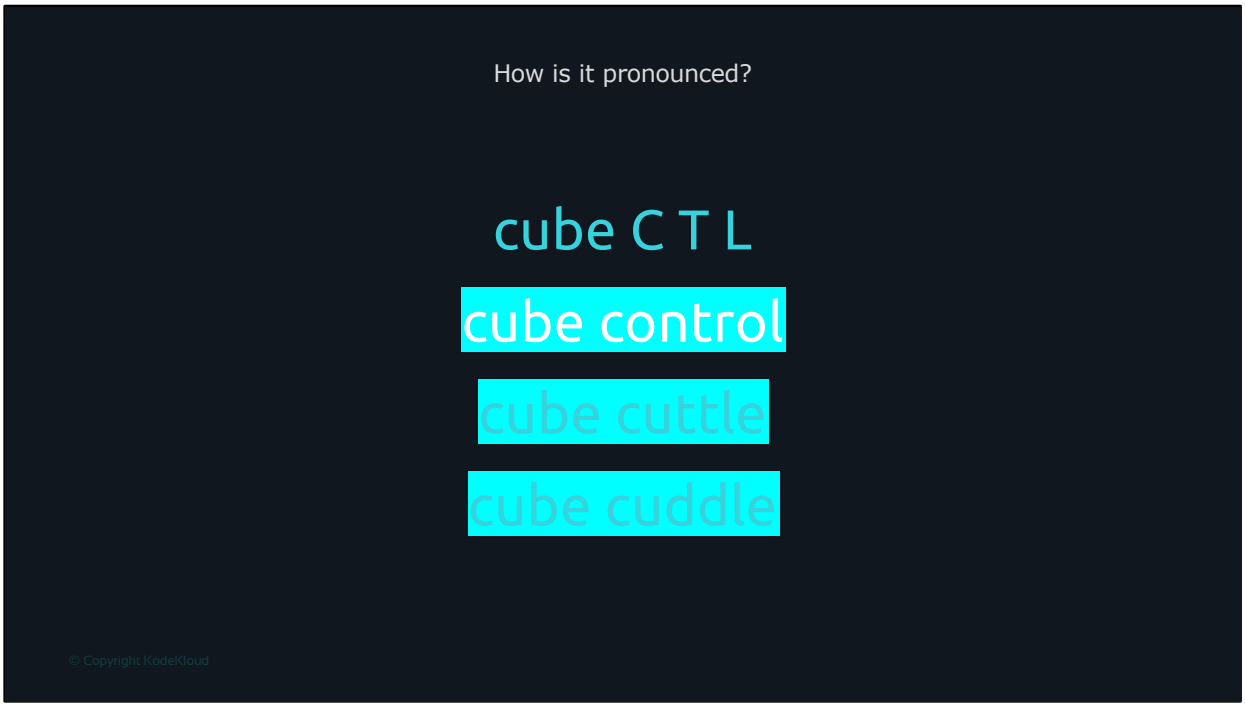
On the worker nodes you also have container runtime that is responsible for running containers. And one example of that is Docker. Now, it used to be Docker for a long time in the past – because Kubernetes was originally built to orchestrate Docker containers specifically. However over a period of time it evolved to support other container runtimes. So it no longer supports Docker directly, but supports the runtime component of Docker which today is managed by ContainerD. There is a separate video that talks about the whole history of Kubernetes and Docker and how they started out together and what changed. So check it out in the link given below.

So going forward we are going to refer to container runtime in kubernetes as containerD.

And that's the high level architecture of a kubernetes cluster. And next we will look into the kubernetes CLI.

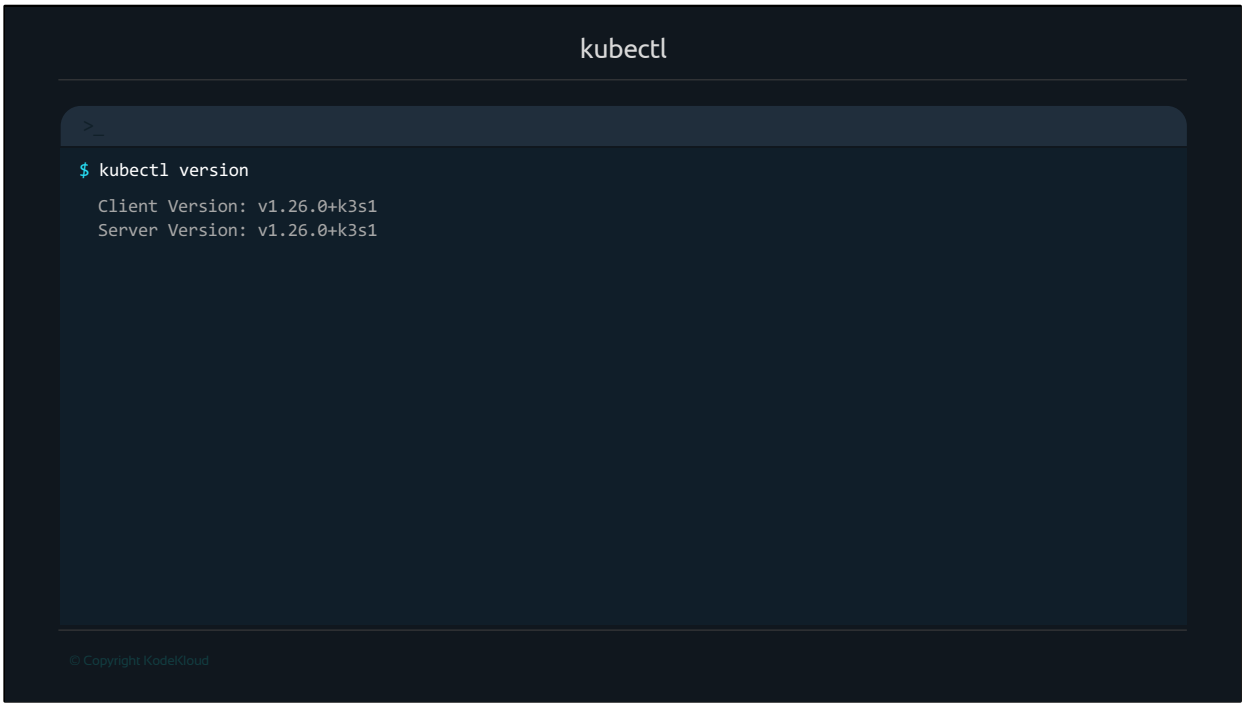


Let's take a look at the Kubectl utility. Kubectl is the command line utility of Kubernetes. This is the tool or command you would use to operate the kubernetes cluster such as to view the status of the cluster, to provision application, to scale up, scale down, delete and many other things.

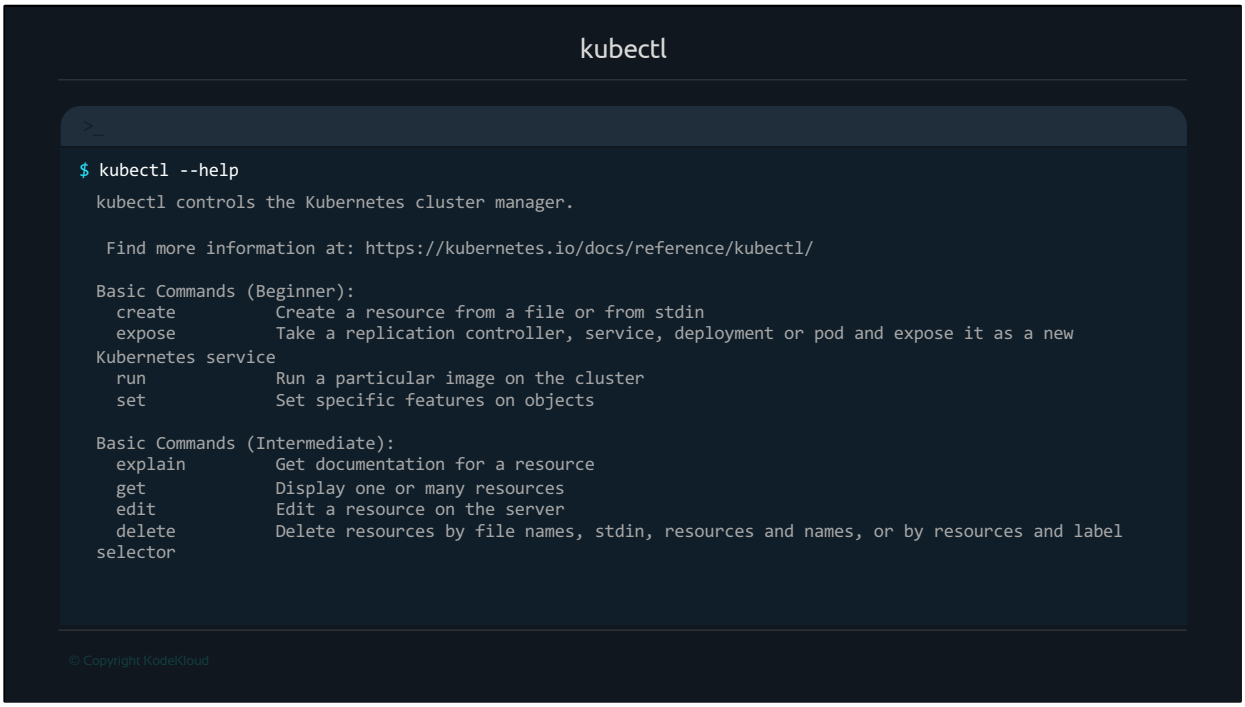


One of the questions I get asked often is how to pronounce this. Different people pronounce it differently. Some say kube C T L, other's say kube control, some say kube cuttle or kube cuddle. The canonical pronounciation is "cube control" though. So I'll try to stick to that. I myself have changed the way I pronounce it over the years. Kube Cuttle came easy to me so you'll hear me say that at times. Forgive me if you hear me mix it up at different times.

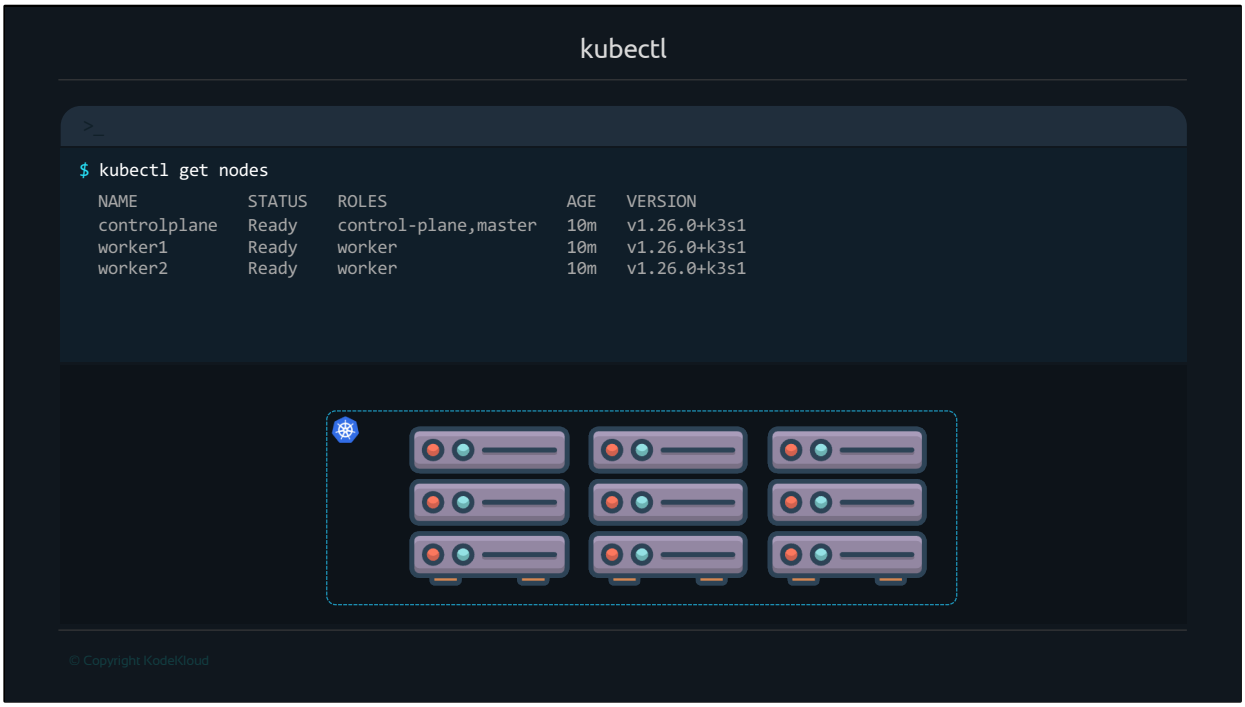
Now that it is out of the way let's get started.



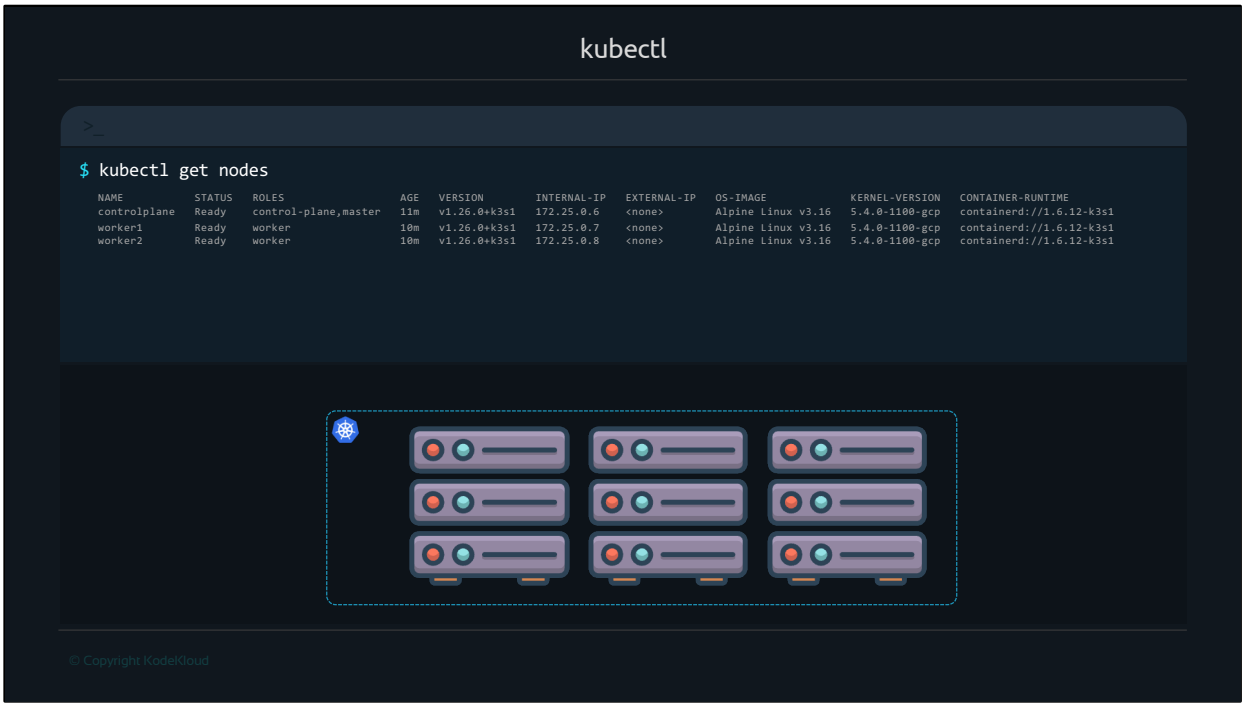
To identify the version of the kubectl client and the kubernetes server, run the kubectl version command. This lists the client and server version along with the version of any other tool installed in the system.



Thehelp option lists basic help information such as the basic commands that can be run. We will dig into these commands later in this video.



Let's begin with a few simple commands. To see a list of nodes in your cluster run the kubectl get nodes command. The output shows you the name of the node, it's status, the roles, how long the machine has been up and the version of Kubernetes running on that system.



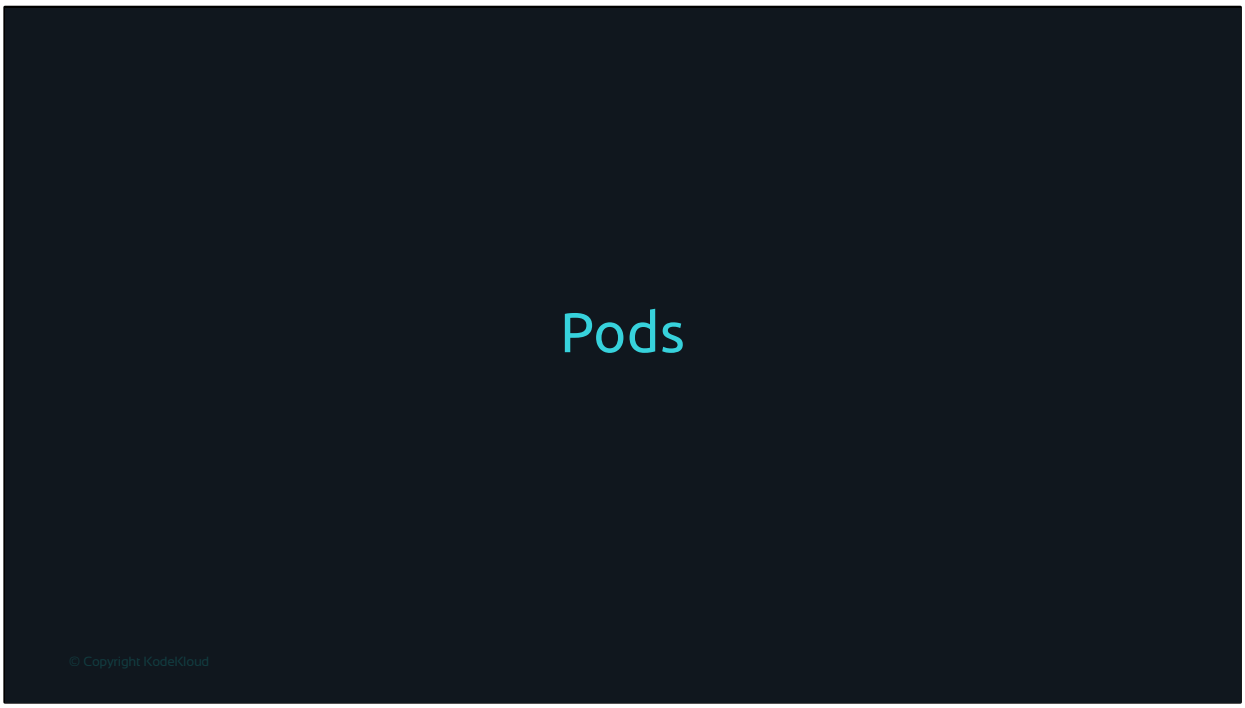
To get a more verbose output with more details such as internal IP, OSImage, kernel version, container runtime etc, run the same command with the –o wide option.

It's time for our first hands-on labs activity. Use the link given here to access the labs. As mentioned before the labs come free of cost with the course. All you need to do is signup for the free course and start the lab named "Familiarize with the Lab environment". In this lab you will use the kubectl commands to identify the cluster setup and the nodes available in it.

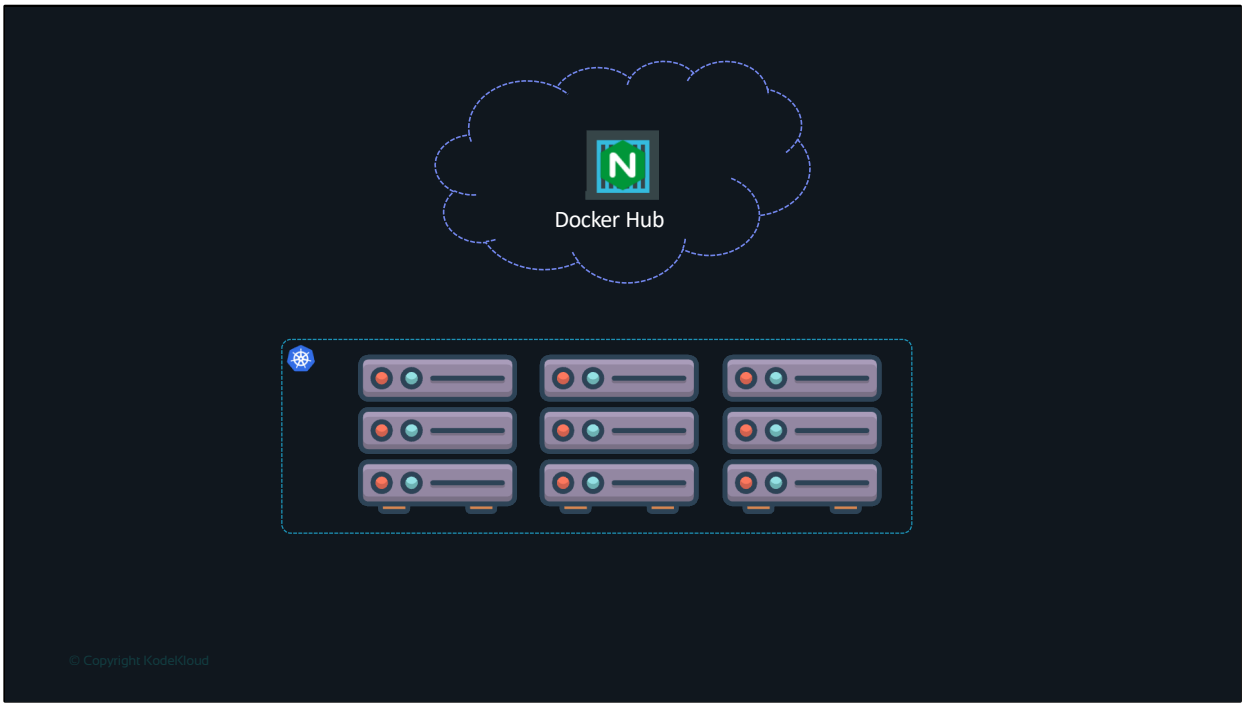
https://kodekloud.com/topic/labs-familiarize-with-lab-environment/

This course is designed for you to have a seamless experience from start to finish. And that's why we have labs after each concept that will help you gain hands-on experience on exactly what you learned until then. So we are going to work on an existing Kubernetes cluster that's already setup and get familiarized with the cluster, the kubrnetes command line interface, deploy applications to the cluster with pods, deployments, services etc. At the end of this course we'll share instructions on setting up your own local environment for you to continue your studies. Meanwhile we do not want you to be distracted with any issues that might come up when you try to build your own cluster. So my recommendation is to aim to complete this course with our labs and go from start to finish without any interruption. If this is a 3 hours course, you should aim to complete it in 3 hours or max 5 hours from now.

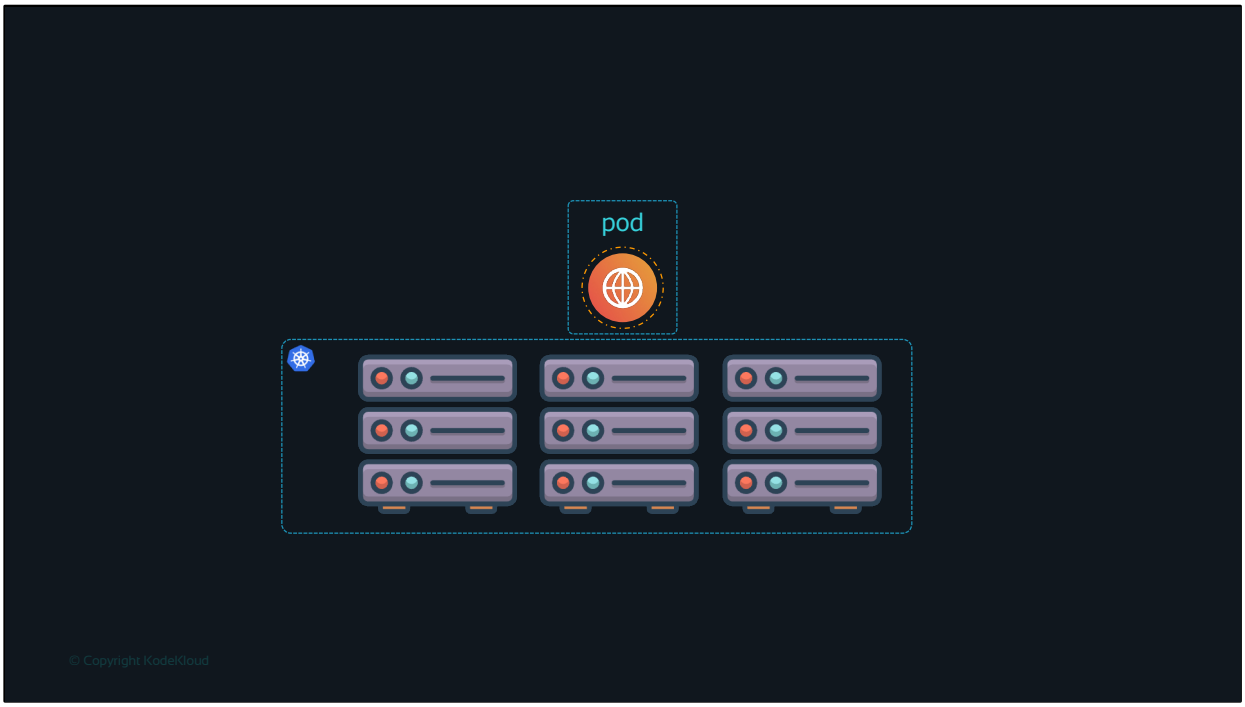
So head over to the labs using the links given below and come back here one you are done.



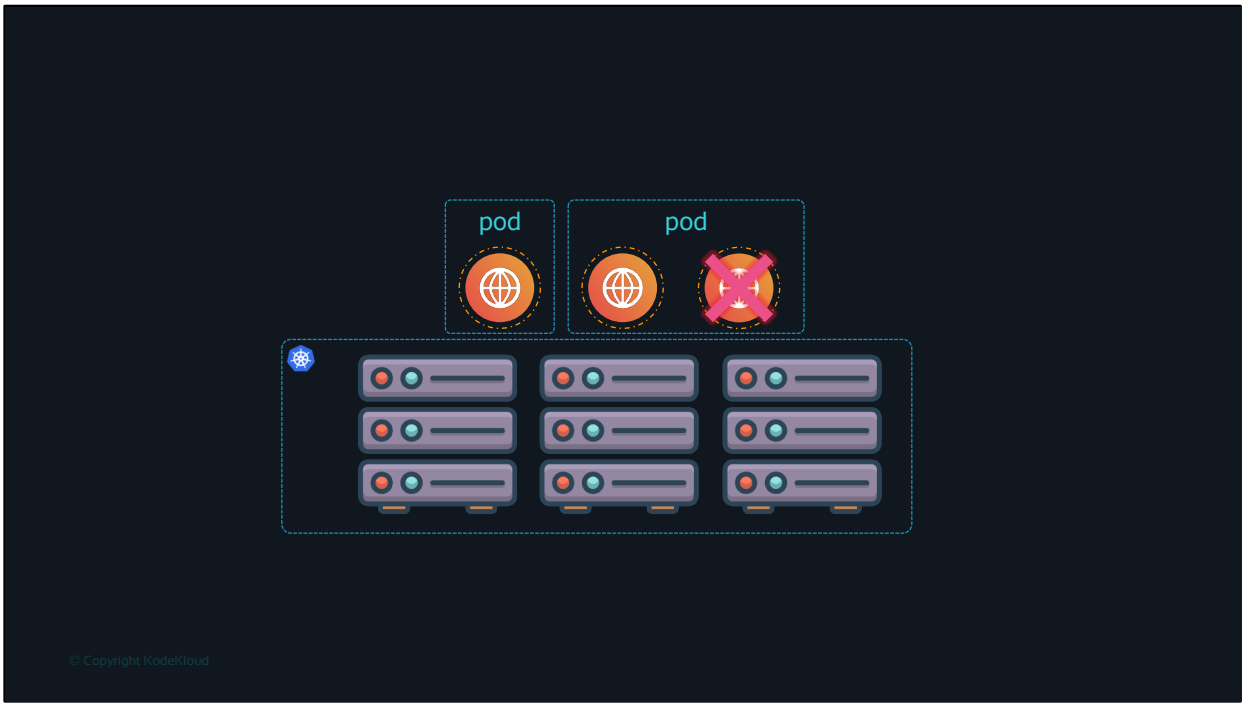
Let's take a look at Kubernetes Pods now. But before we begin, we would like to assume that the following have been setup already.



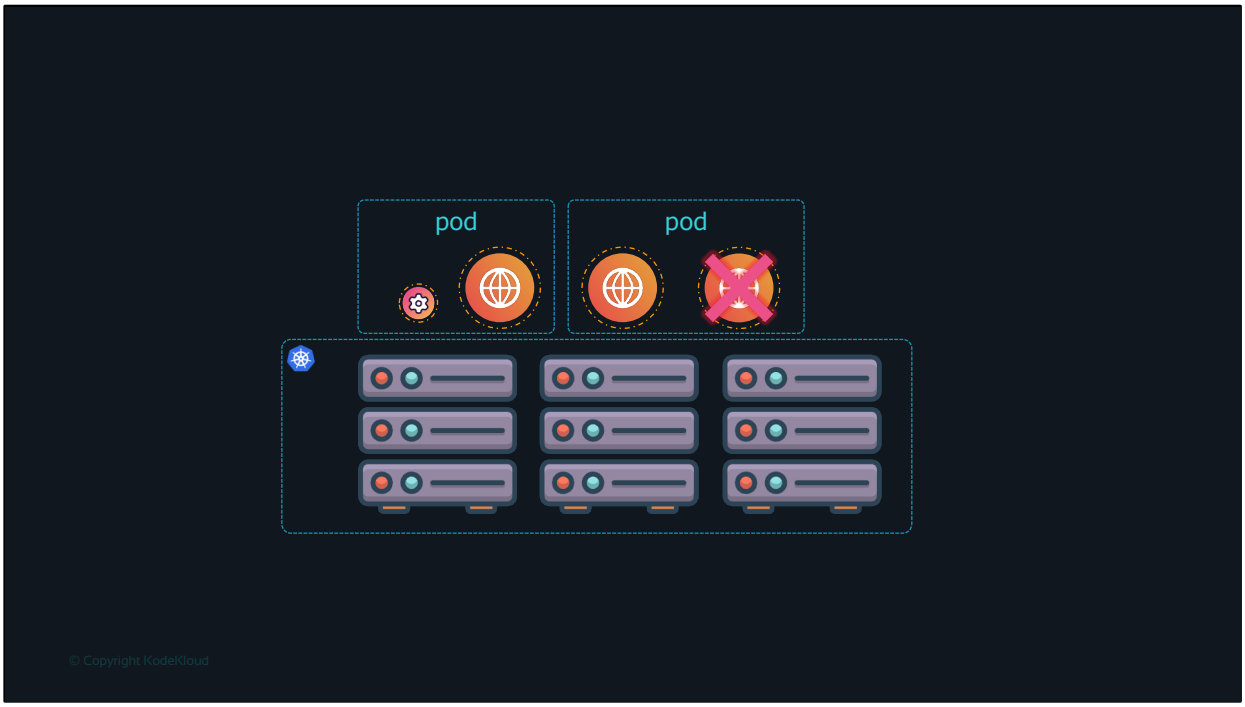
At this point, we assume that the application is already developed and built into Docker Images and it is available on a Docker repository like Docker hub, so kubernetes can pull it down. We also assume that the Kubernetes cluster has already been setup and is working.



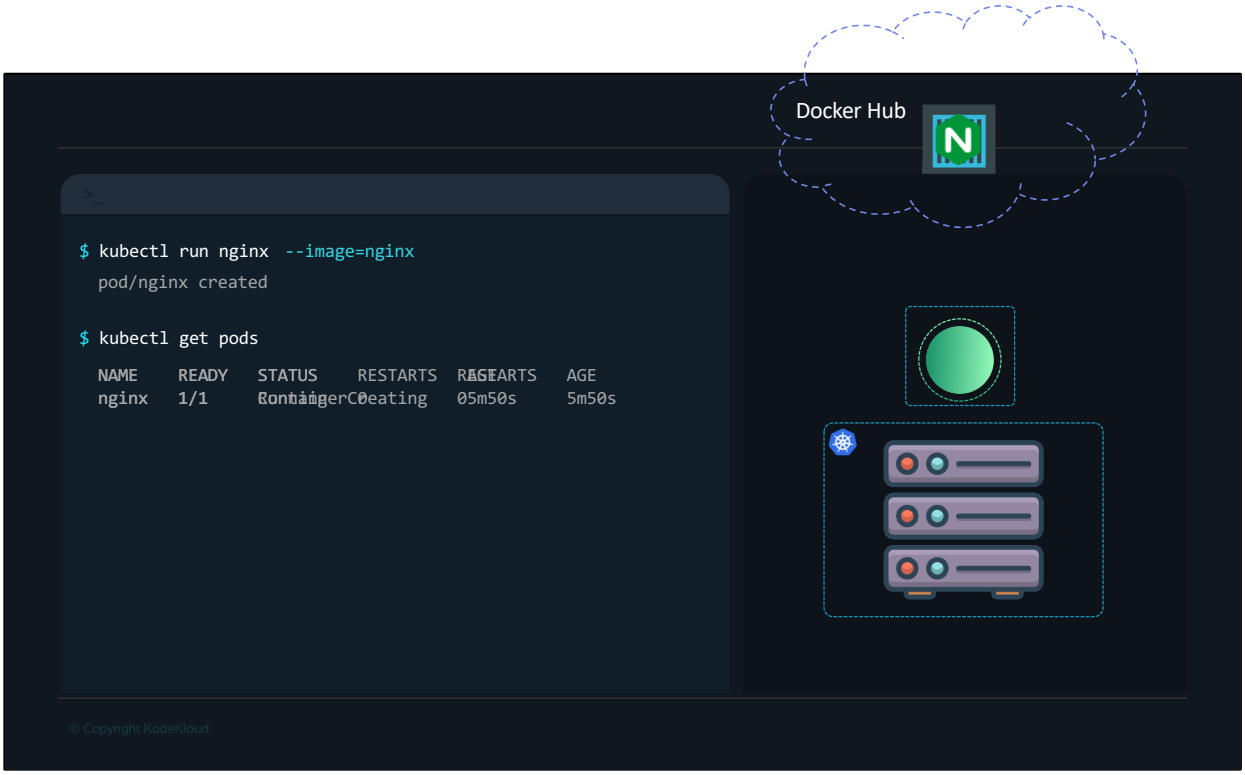
As we discussed before, <click> with kubernetes our ultimate aim is to deploy our application in the form of containers on a set of machines that are configured as worker nodes in a cluster. <click> However, kubernetes does not deploy containers directly on the worker nodes. The containers are encapsulated 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. So what happens when you want to scale up?



Do you add more containers to the same pod? No! You create more pods. Typically an application instance running as a container has a 1-to-1 relationship with a Pod. To create more instances of application you create more Pods. However the 1-to-1 relationship is not a strict rule.



It is a common practice to have a helper container or a sidecar container along with the main application. Such as an agent that collects logs or monitors the application and reports to a third party. And that's absolutely fine.



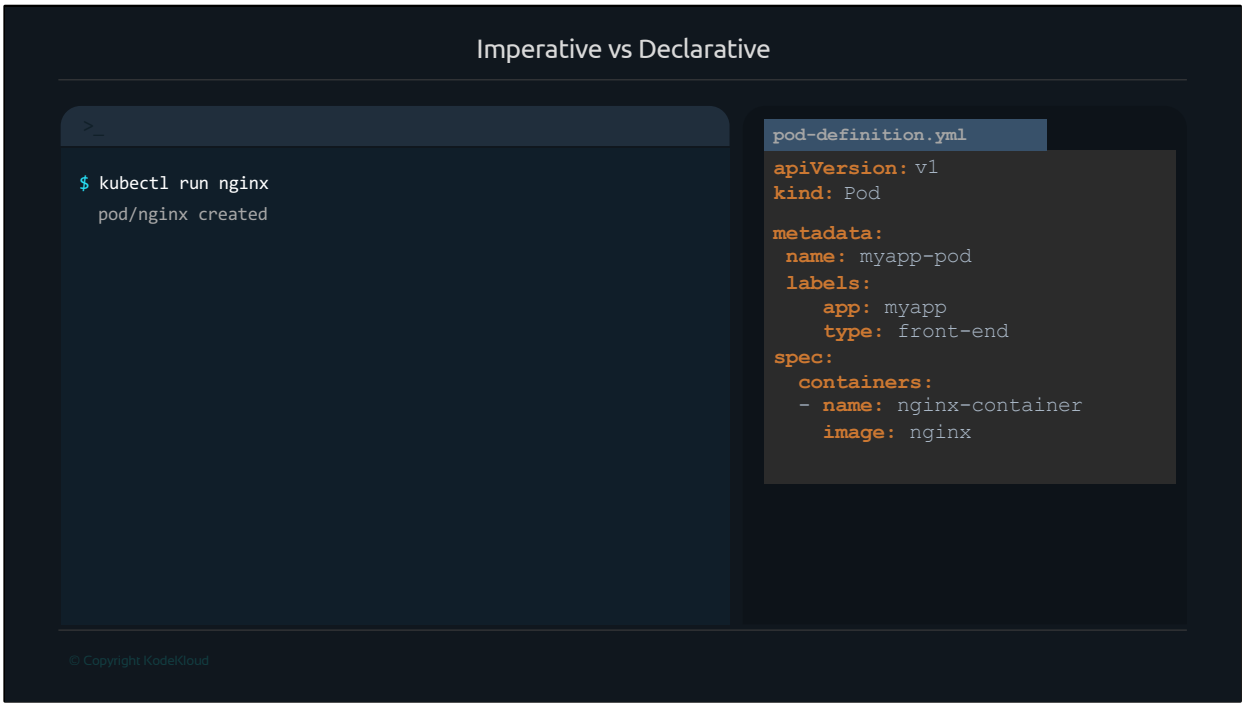
Let us now look at how to create PODs. <click> For this we run the kubectl run command. We specify a name for the pod and the image to be used to create the pod. What this command really does is it deploys a container by creating a POD. So it first creates a POD automatically and deploys an instance of the nginx docker image. <click> But were does it get the application image from? <click> 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? <click> 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.

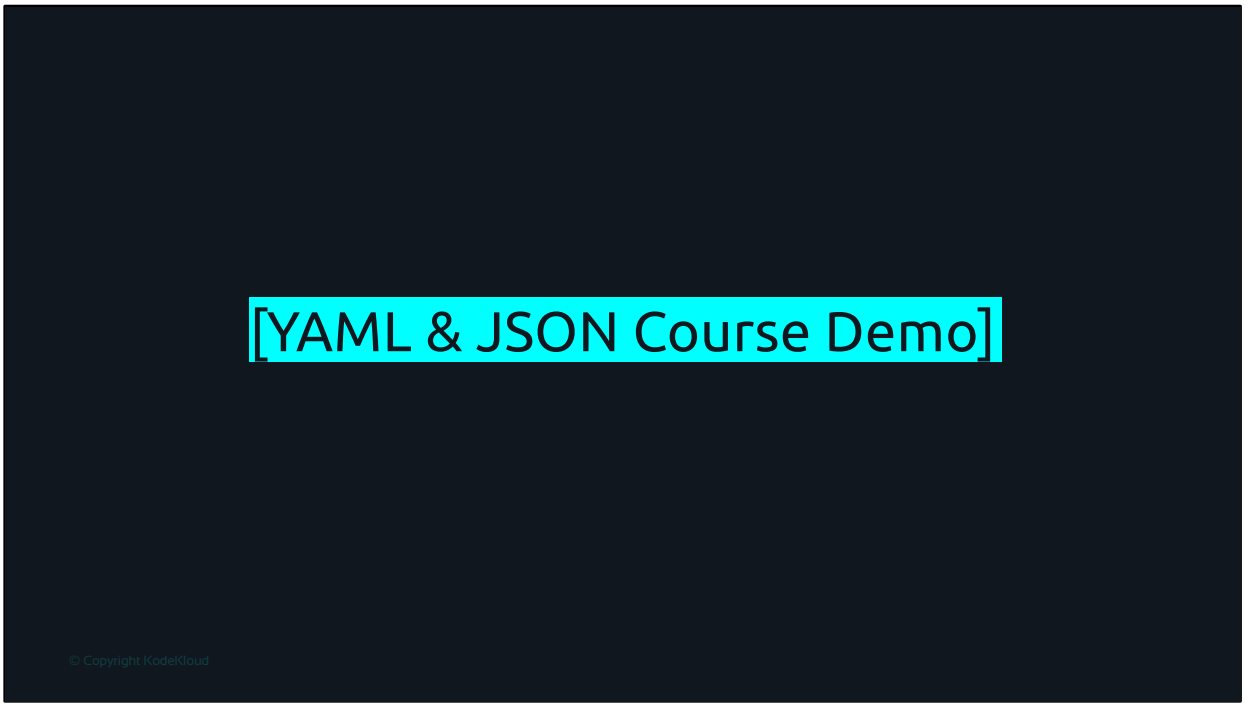
<click> For now we will just see how to deploy a POD and in a later lecture once we learn about networking and services we will get to know how to make this service accessible to end users.

Now this is called as the imperative way to create a POD. Let us now see the declarative way to create a POD.

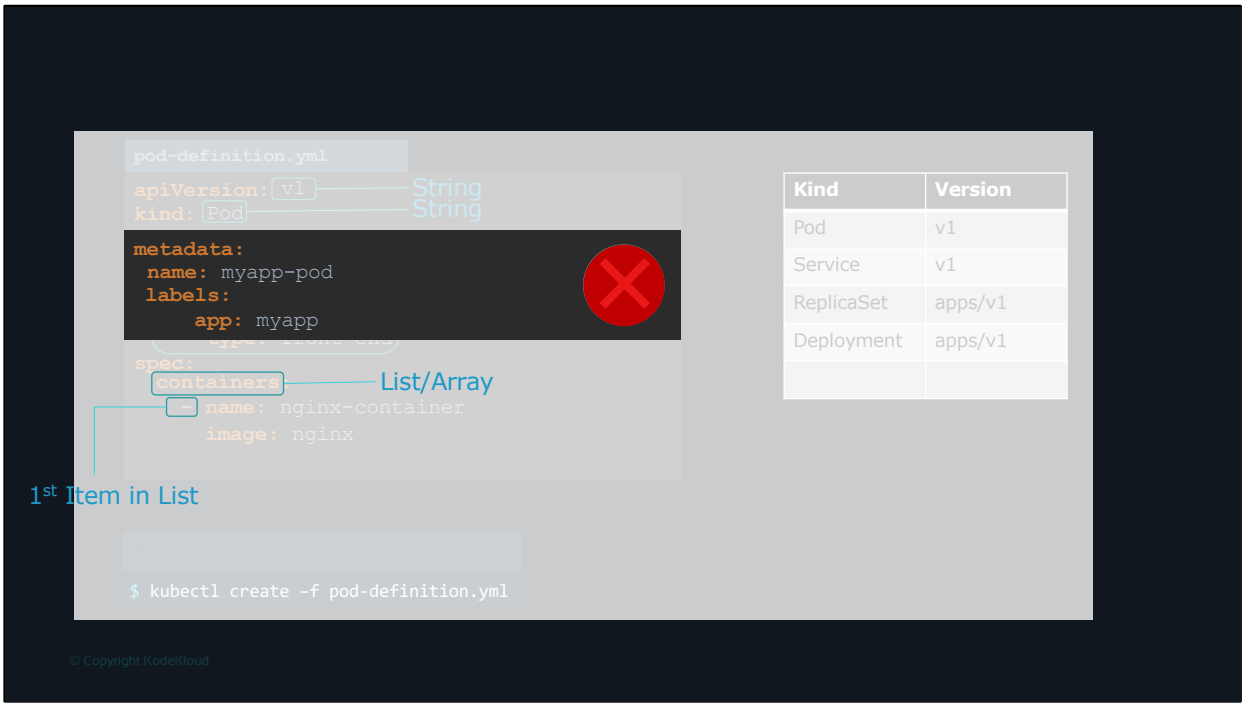


So that was the imperative way of creating an object in Kubernetes. You run a command to create one object at a time. When there are many objects and services in your application this is not a viable option.

The more preferred approach is the declarative way, where you create a YAML file with the specifications of the object – the pod in this case. And have kubernetes apply that configuration. This way you can define the state of your application and its services as code and store it in source code repositories and version them. This approach enables version control, CI/CD and sharing these with others and collaborating together.



If you are new to YAML check out our free course on YAML and JSON available on KodeKloud. There are hands-on activities that can help you get very comfortable with YAML soon. Because it's going to be an important necessity going forward.



Now we will learn how to develop YAML files specifically for Kubernetes. Kubernetes uses YAML files as input for the creation of objects such as PODs, Replicas, Deployments, Services etc. All of these follow similar structure. A kubernetes definition file always contains 4 top level fields. <click> 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, <click 2> we will set the apiVersion as v1. If you are creating a service, replicaset or deployments you will use the versions listed here. We will see what these are 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 <c> 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.

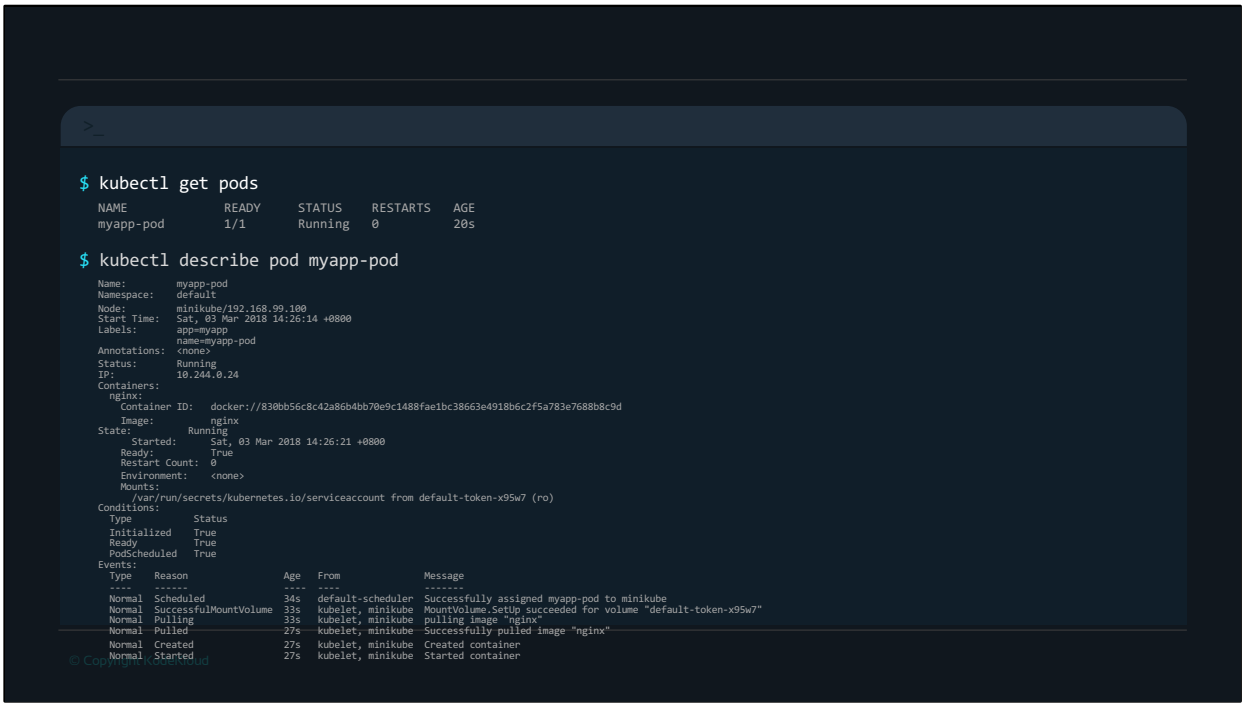
The next is metadata. <click> The metadata is data about the object like its name, labels etc. <click> As you can see unlike the first two were you specified a string value, this, is in the form of a dictionary. <click> So everything under metadata is intended to the right a little bit and so names and labels are children of metadata. <click> The number of spaces before the two properties name and labels doesn't matter, <click> but they should be the same as they are siblings. In this case labels has more spaces on the left than name and so it is now a child of the name property instead of a sibling. <click> Also the two properties must have MORE spaces than its parent, which is metadata, so that its intended to the right a little bit. In this case all 3 have the same number of spaces before them and so they are all siblings, which is not correct. <click> 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, <click> 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, <click> 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. <click>

<click>
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.

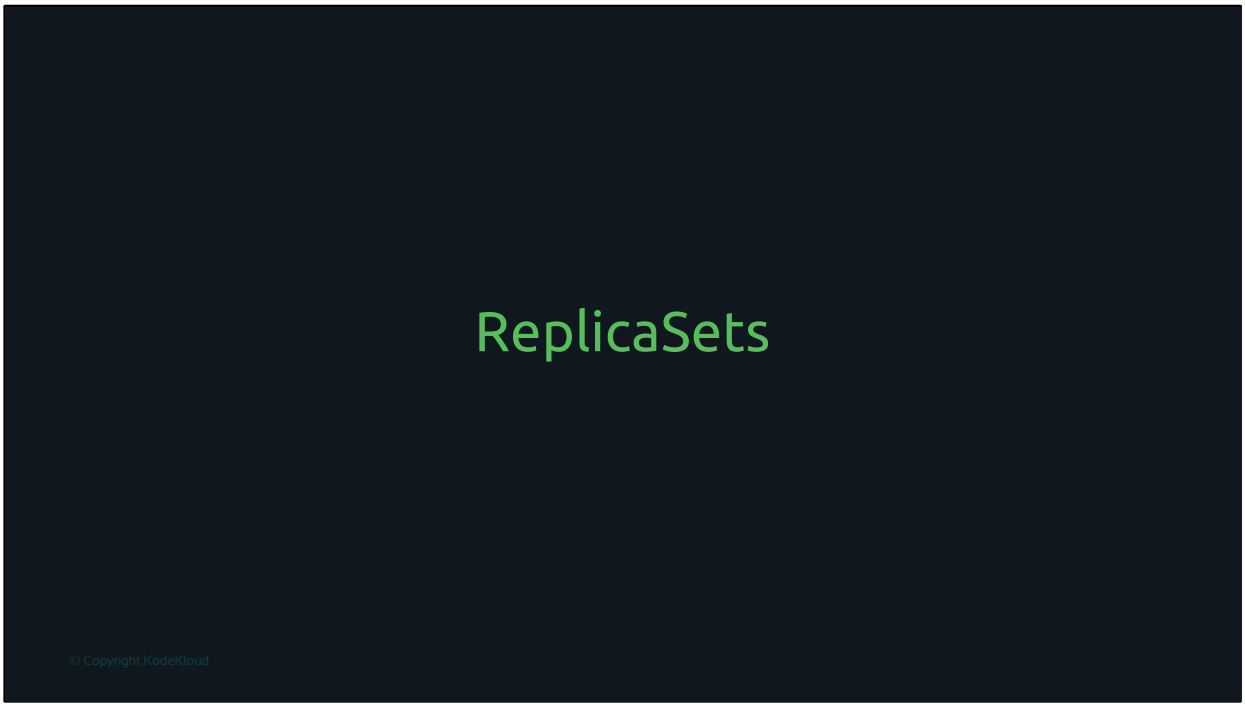
So to summarize remember the 4 top level properties. apiVersion, kind, metadata and spec. Then start by adding values to those depending on the object you are creating.



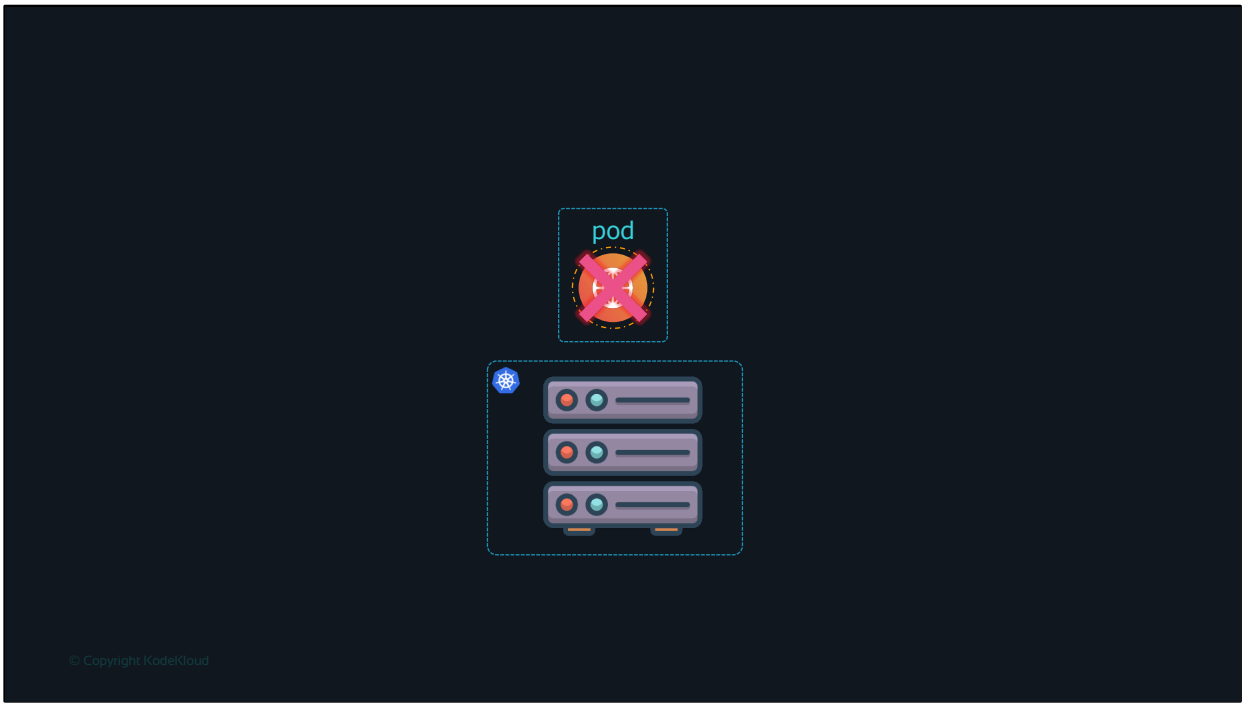
Once we create the pod, how do you see it? Use the <click> kubectl get pods command to see a list of pods available. In this case its just one. To see detailed information about the pod run the <click> kubectl describe pod command. This will tell you information about the POD, when it was created, what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

It's time for our second hands-on labs activity. Go back to the labs and access the labs for PODs. Or click on the link given here. In this lab you will create pods and also explore creating YAML files for PODs. Once done come back here and we will resume the course.

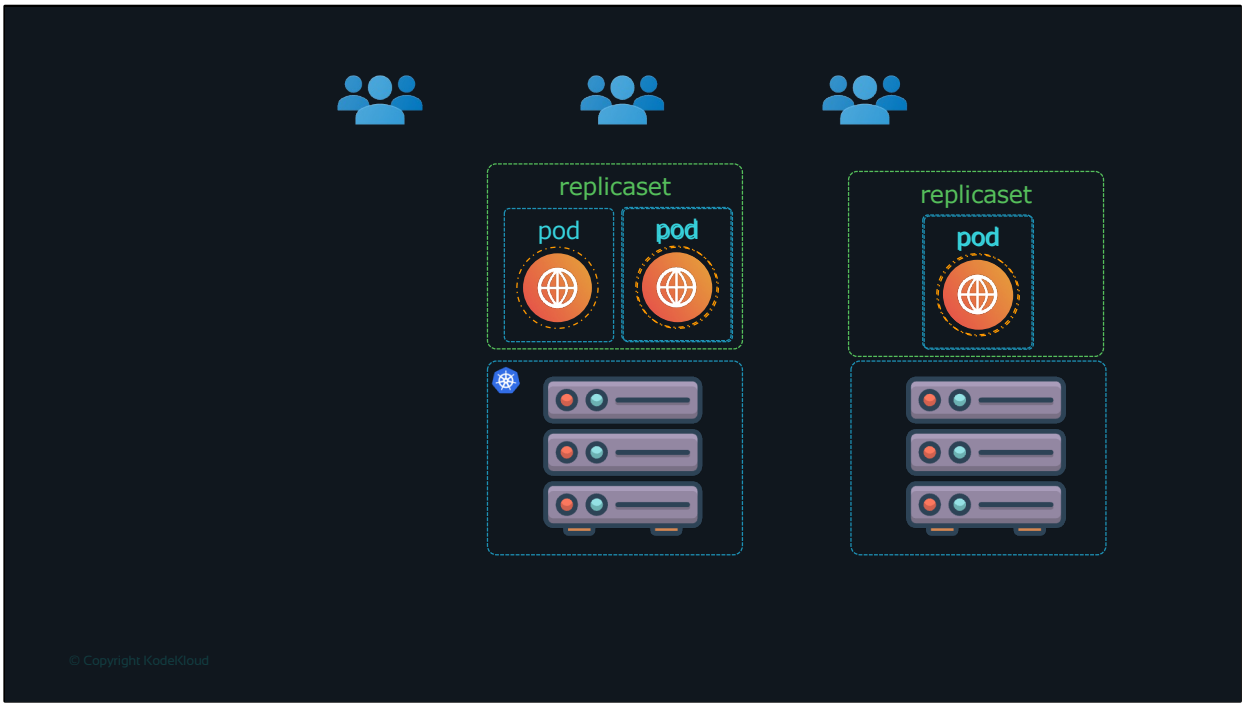
https://kodekloud.com/topic/labs-pods-with-yaml/



Let's now talk about ReplicaSets



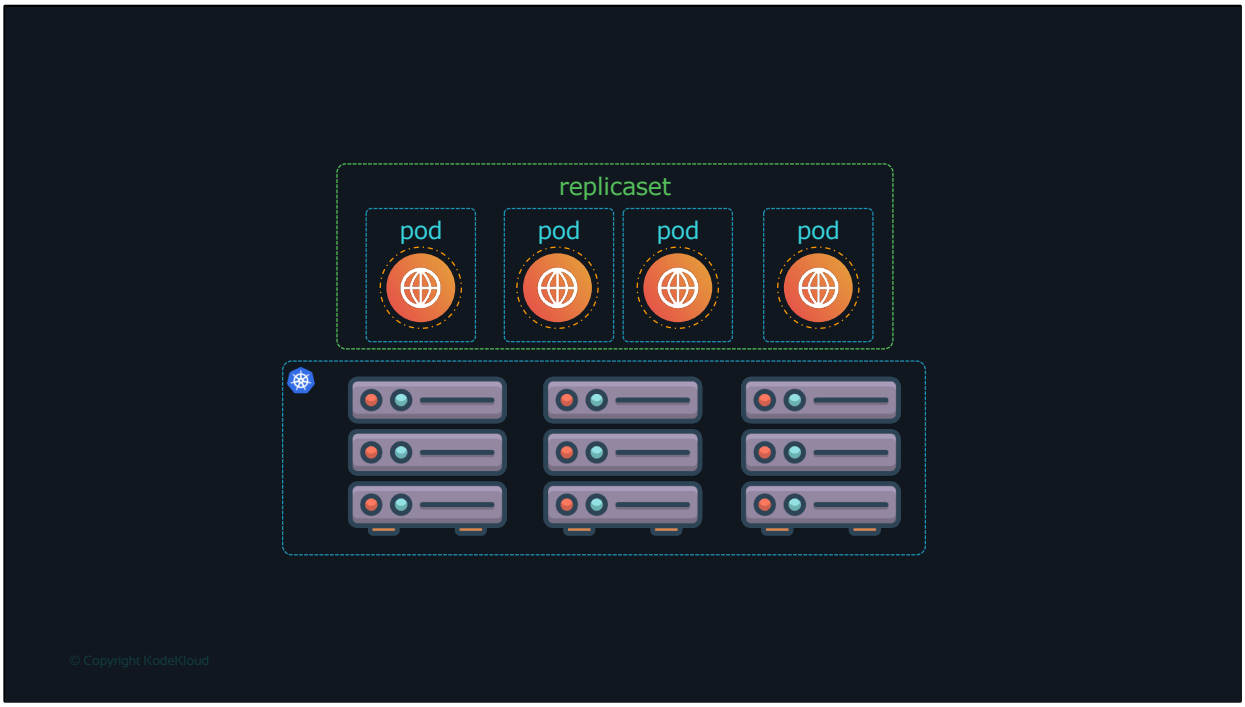
So what is a replica and why do we need a replicaset? <click> Let's go back to our first scenario were we had a single POD running our application. <click> What if for some reason, our application crashes and the POD fails? Users will no longer be able to access our application.



To prevent users from losing access to our application, we would like to have more than one instance or POD running at the same time. That way if one fails we still have our application running on the other one. And the replicaset brings the failed one back to ensure a pre-defined number of replicas are always running <click> The replicaset helps us run multiple instances of a single POD in the kubernetes cluster thus providing <click> High Availability.

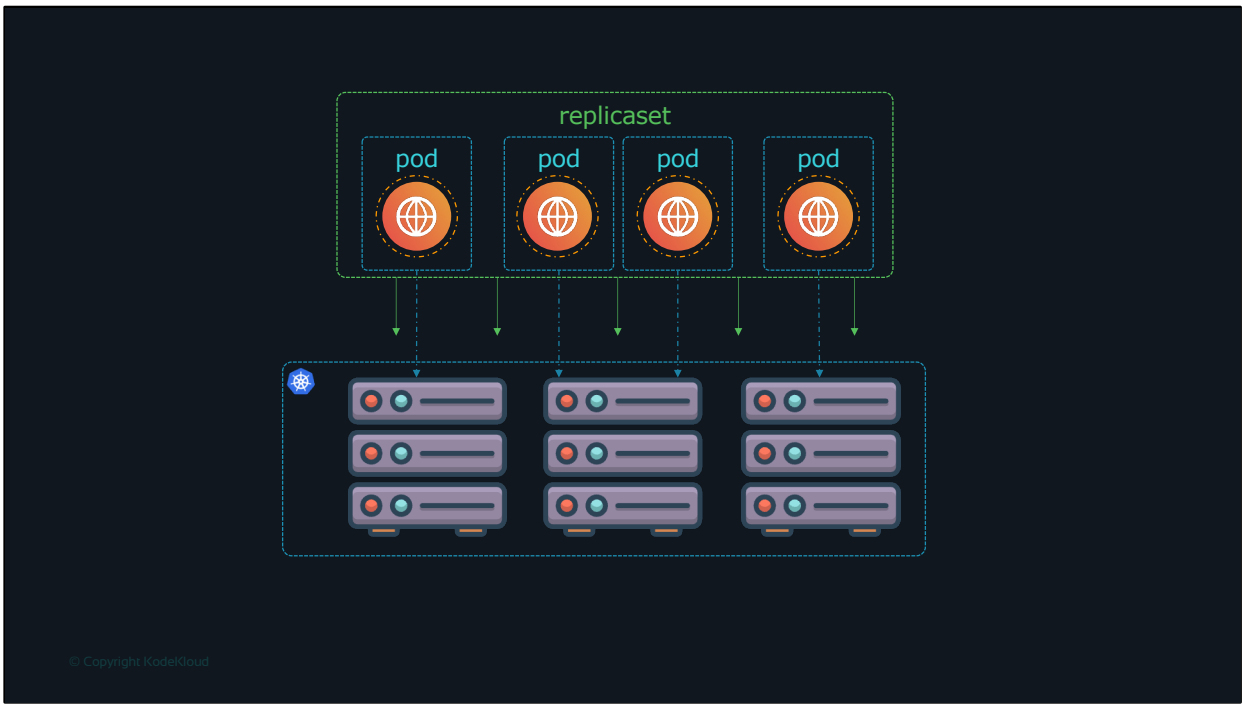
So does that mean you can't use a replicaset if you plan to have a single POD? No! Even if you have a single POD, <click> the replication controller can help by automatically bringing up a new POD when the existing one fails. Thus the replicaset ensures that the specified number of PODs are running at all times. Even if it's just 1 or 100.

Another reason we need replicaset is to create multiple PODs to share the load across them. For example, <click> in this simple scenario we have a single POD serving a set of users. <click> When the number of users increase and If we were to run out of resources on the first node,



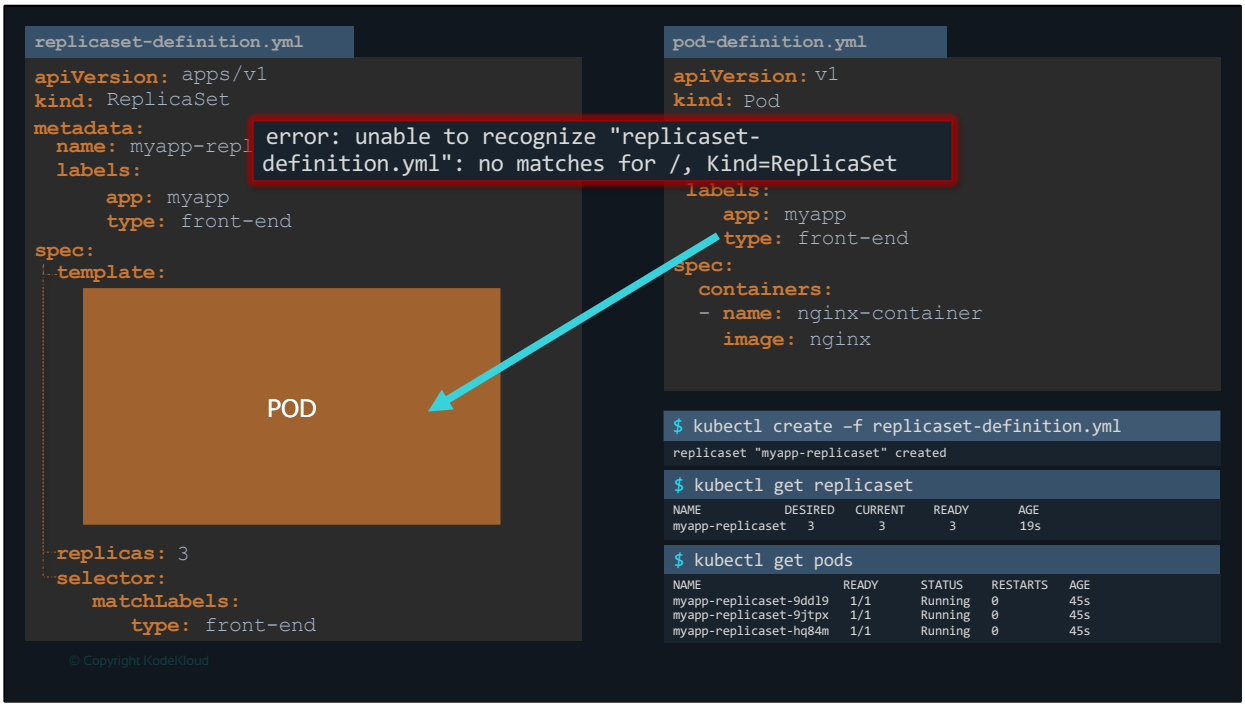
we could deploy additional PODs across other nodes in the cluster. As you can see, the replicaset spans across multiple nodes in the cluster. It helps us balance the load across multiple pods on different nodes as well as scale our application when the demand increases.

So a Pod has a one-to-one relationship with a node. A pod can only run on one node. You cannot move a pod from one node to the other. . A replicaset spans across the entire cluster.



So a Pod has a one-to-one relationship with a node. A pod can only run on one node at a time. You cannot move a pod from one node to the other. You'll have to kill it and recreate it on another node. Well, technically the scheduler decides which node a pod gets assigned to and there are ways for you to control that which is out of scope for this crash course. We discuss those in much detail in our CKA course. For now we will just stick to the basics. So a pod lives on one node.

A replicaset, spans across the entire cluster. And a replicaset can deploy a pod on any node in the cluster. It monitors the number of pods in the cluster and ensures enough is deployed at all times.



Let us now look at how we create a replicaset. <click> As with the previous lecture, we start by creating a replicasetdefinition file. We will name it replicaset-definition.yml. As with any kubernetes definition file, we will have 4 sections. The apiVersion, kind, metadata and spec. The apiVersion is specific to what we are creating. In this case replicaset is supported in kubernetes apiVersion apps/v1.. <click> If you get this wrong, you are likely to get an error that looks like this. It would say no match for kind ReplicaSet, because the specified kubernetes api version has no support for ReplicaSet.

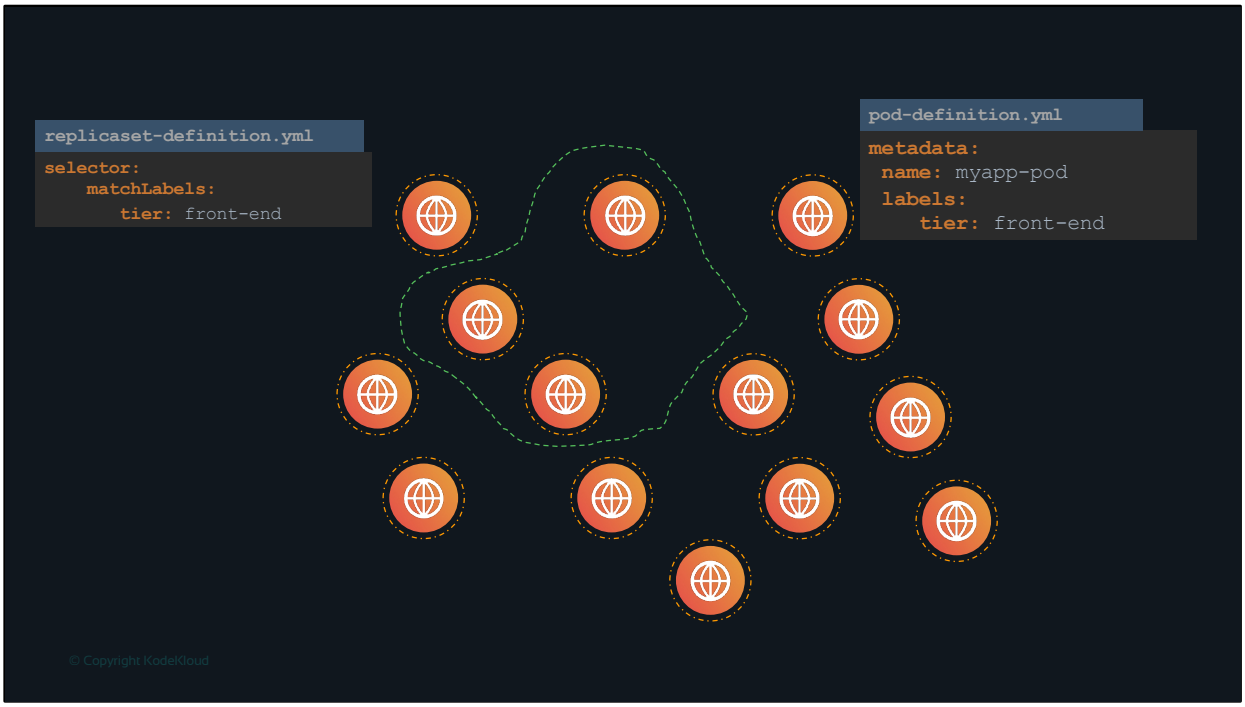
<click> The kind as we know is ReplicaSet. Under metadata, we will add a name and we will call it myapp-replicaset. And we will also add a few labels, app and type and assign values to them. So far, it has been very similar to how we created a POD in the previous section. The next is the most crucial part of the definition file and that is the specification written as spec. For any kubernetes definition file, the spec section defines what's inside the object we are creating. In this case we know that the replicaset creates multiple instances of a POD. But what POD? <click> We create a template section under spec to provide a POD template to be used by the replicaset to create replicas. Now <pause> how do we DEFINE the POD template? It's not that hard because, we have already done that in the previous exercise. <click> Remember,

we created a pod-definition file in the previous exercise. We could re-use the contents of the same file to populate the template section. <click> Move all the contents of the pod-definition file into the template section of the replication controller, except for the first two lines – which are apiVersion and kind. <click> Remember whatever we move must be UNDER the template section. Meaning, they should be intended to the right and have more spaces before them than the template line itself. <click> Looking at our file, we now have two metadata sections – one is for the ReplicaSet and another for the POD and <click> we have two spec sections – one for each. We have nested two definition files together. The replication controller being the parent and the pod-definition being the child.

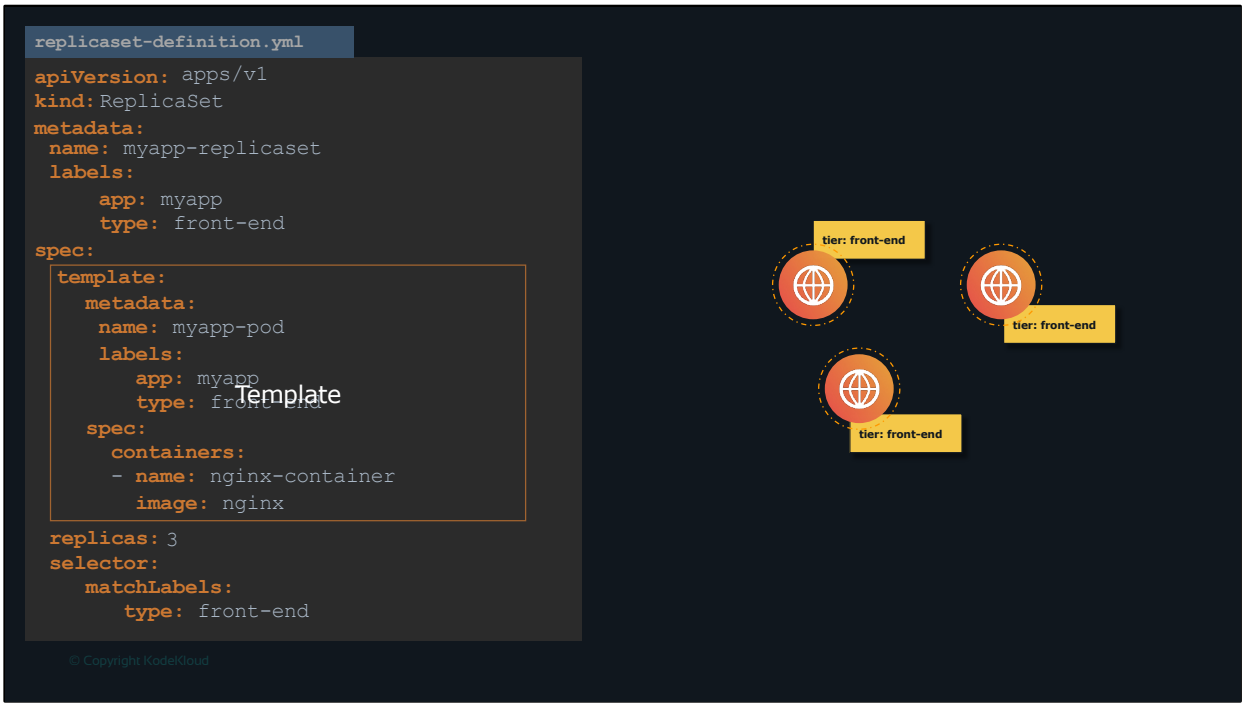
<click> Now, there is something still missing. <click> We haven't mentioned how many replicas we need in the replication controller. For that, <click> add another property to the spec called replicas and <click> input the number of replicas you need under it. Remember that the template and replicas are direct children of the spec section. So they are siblings and must be on the same vertical line : having equal number of spaces before them.

Replica set requires a selector definition. The selector section helps the replicaset identify what pods fall under it. But why would you have to specify what PODs fall under it, if you have provided the contents of the pod-definition file itself in the template? It's BECAUSE, replica set can ALSO manage pods that were not created as part of the replicaset creation. Say for example, there were pods created BEFORE the creation of the ReplicaSet that match the labels specified in the selector, the replica set will also take THOSE pods into consideration when creating the replicas. I will elaborate this in the next slide. For now know that it has to be written in the form of matchLabels as shown here. The matchLabels selector simply matches the labels specified under it to the labels on the PODs. The replicaset selector also provides many other options for matching labels that were not available in a replication controller.

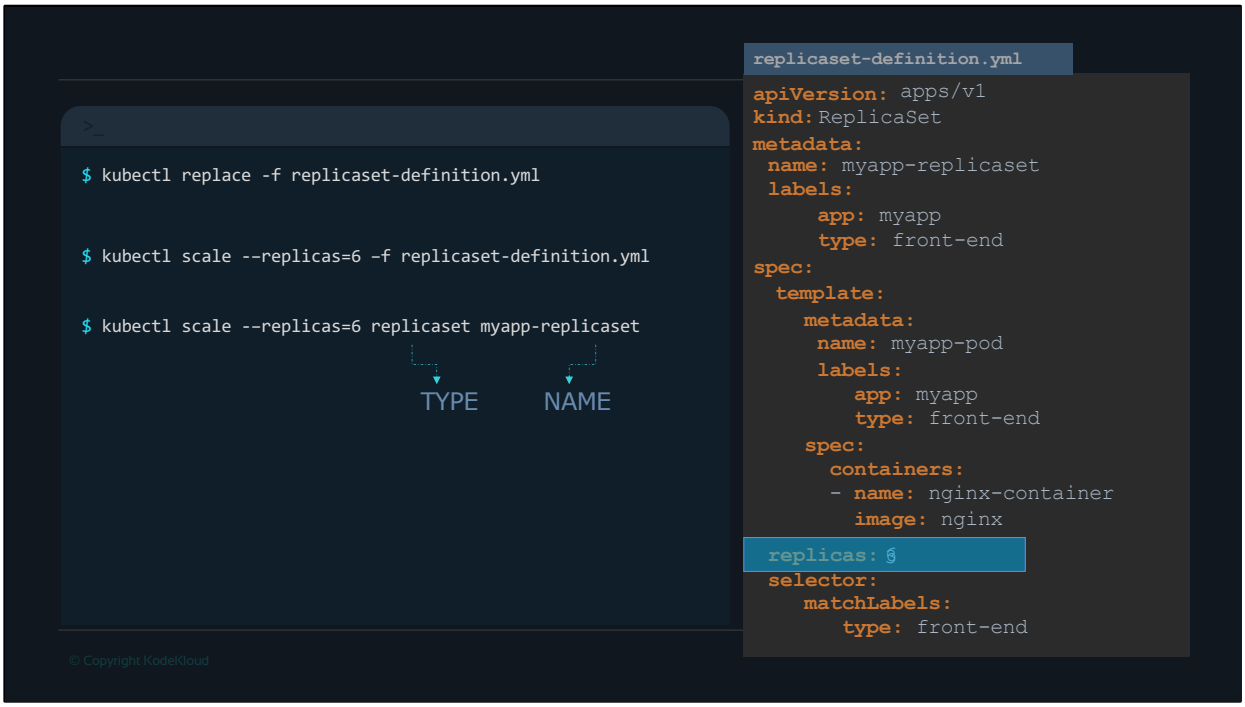
Once the file is ready, <click> run the kubectl create command and input the file using the –f parameter. The replicaset Is created. When the replicaset is created it first creates the PODs using the pod-definition template as many as required, which is 3 in this case. To view the list of created replicaset <click> run the kubectl get replicaset command and you will see the replicaset listed. We can also see the desired number of replicas or pods, the current number of replicas and how many of them are ready. If you would like to see the pods that were created by the replicaset , run the <click> kubectl get pods command and you will see 3 pods running. Note that all of them are starting with the name of the replicaset which is myapp-replicaset indicating that they are all created automatically by the replicaset.



So what is the deal with Labels and Selectors? Why do we label our PODs and objects in kubernetes? Let us look at a simple scenario. <click> Say we deployed 3 instances of our frontend web application as 3 PODs. <click> We would like to create a replication controller or replica set to ensure that we have 3 active PODs at anytime. And YES that is one of the use cases of replica sets. You CAN use it to monitor existing pods, if you have them already created, as it IS in this example. In case they were not created, the replica set will create them for you. The role of the replicaset is to monitor the pods and if any of them were to fail, deploy new ones. The replica set is in FACT a process that monitors the pods. Now, how does the replicaset KNOW what pods to monitor. <click> There could be 100s of other PODs in the cluster running different application. <click> This is were labelling our PODs during creation comes in handy. <click> We could now provide these labels as a filter for replicaset. Under the selector section we use the matchLabels filter and provide the same label that we used while creating the pods. This way the replicaset knows which pods to monitor.



Now let me ask you a question along the same lines. In the replicaset specification section we learned that there are 3 sections: Template, replicas and the selector. <click> We need 3 replicas and we have updated our selector based on our discussion in the previous slide. Say for instance we have the same scenario as in the previous slide were we have 3 existing PODs that were created already and we need to create a replica set to monitor the PODs to ensure there are a minimum of 3 running at all times. When the replication controller is created, it is NOT going to deploy a new instance of POD as 3 of them with matching labels are already created. <click> In that case, do we really need to provide a template section in the replica-set specification, since we are not expecting the replicaset to create a new POD on deployment? Yes we do, BECAUSE in case one of the PODs were to fail in the future, the replicaset needs to create a new one to maintain the desired number of PODs. And for the replica set to create a new POD, the template definition section IS required.

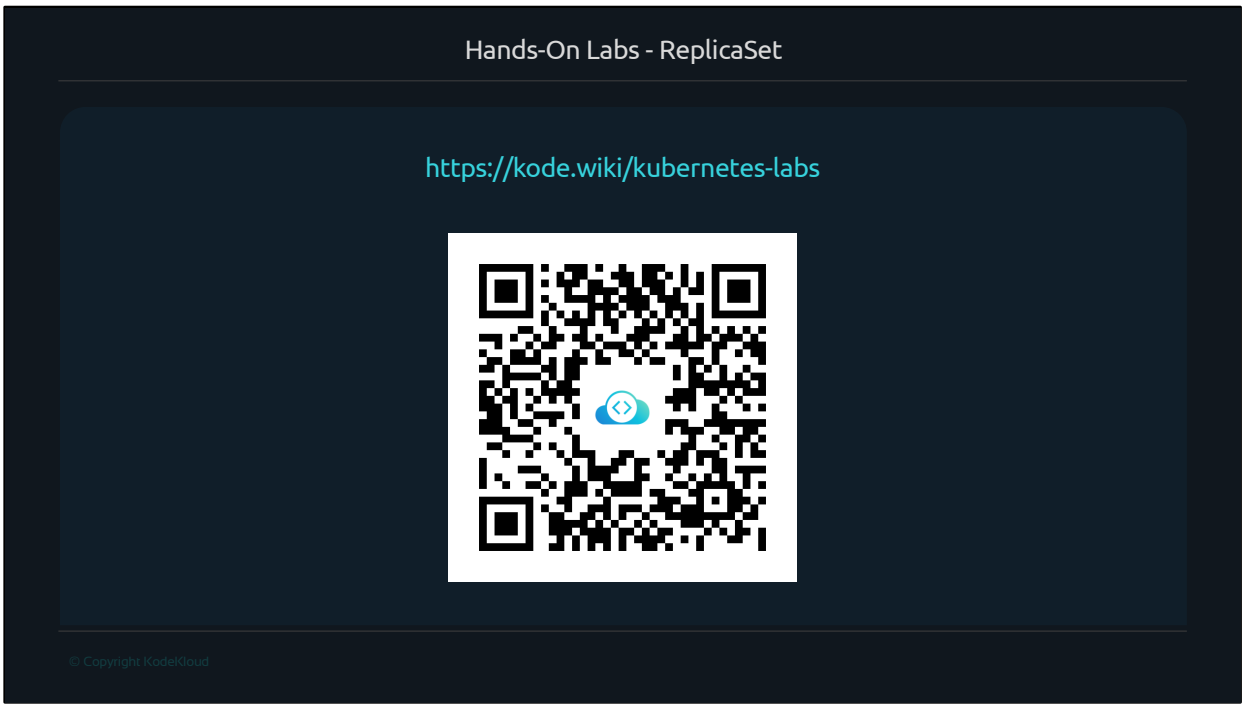


Let's look at how we scale the replicaset. Say we started with 3 replicas and in the future we decide to scale to 6. How do we update our replicaset to scale to 6 replicas. Well there are multiple ways to do it. The first, is to update the number of replicas in the definition file to <click> 6. Then <click> run the kubectl replace command specifying the same file using the –f parameter and that will update the replicaset to have 6 replicas.

The second way to do it is to run the <click> kubectl scale command. Use the replicas parameter to provide the new number of replicas and specify the same file as input. <click> You may either input the definition file or provide the replicaset name in the TYPE Name format. However, Remember that using the file name as input will not result in the number of replicas being updated automatically in the file. In otherwords, the number of replicas in the replicaset-definition file will still be 3 even though you scaled your replicaset to have 6 replicas using the kubectl scale command and the file as input.

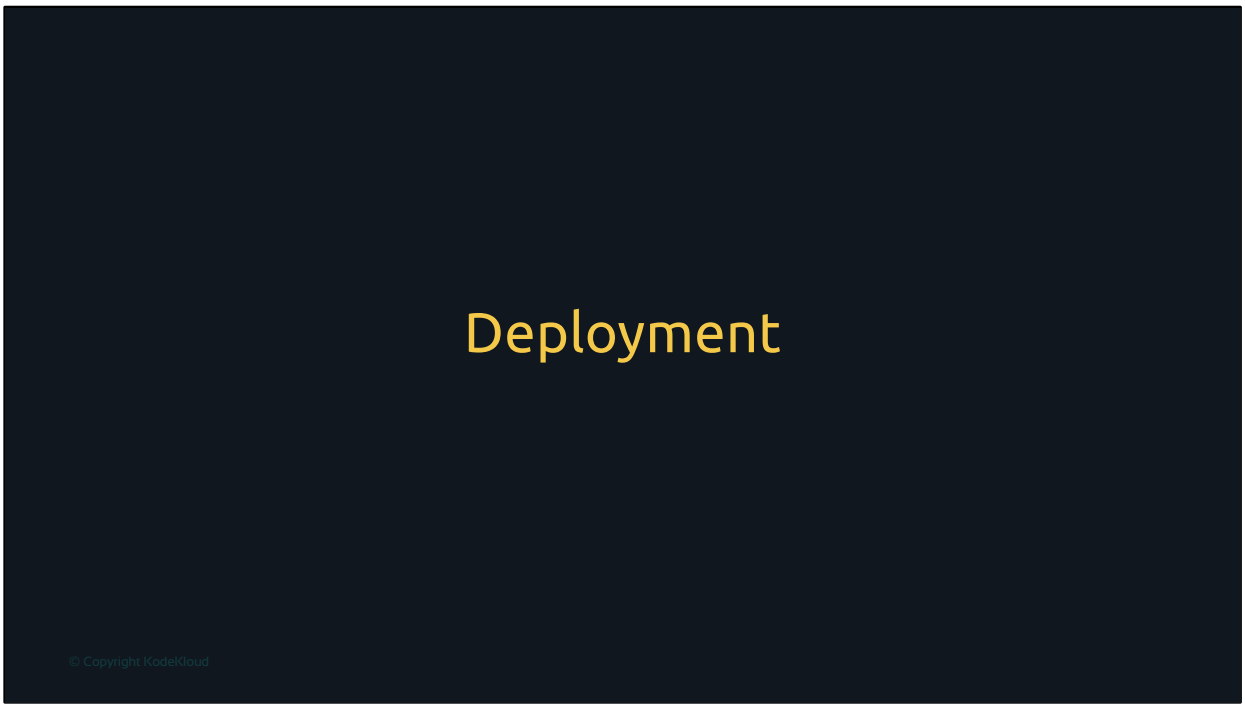
There are also options available for automatically scaling the replicaset based on load, but that is an advanced topic and we will discuss it at a later time.

Let us now review the commands real quick. <click> The kubectl create command, as we know, is used to create a replca set. You must provide the input file using the –f parameter. <click> Use the kubectl get command to see list of replicasets created. <click> Use the kubectl delete replicaset command followed by the name of the replica set to delete the replicaset. <click> And then we have the kubectl replace command to replace or update replicaset and also the <click> kubectl scale command to scale the replicas simply from the command line without having to modify the file.

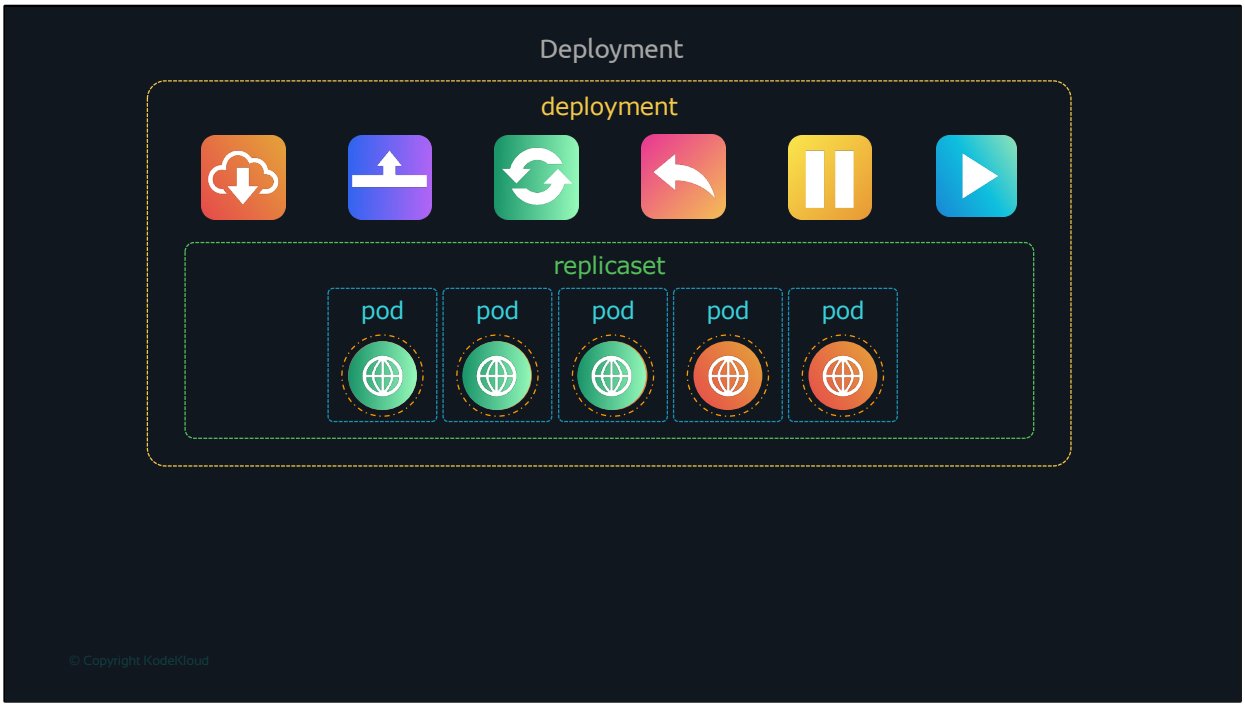


It's time for labs activity. Click on the link to go directly to the labs. If you haven't enrolled already enroll for free.

https://kodekloud.com/topic/labs-pods-with-yaml/



Let's now talk about Deployments.



So we saw how to deploy an application to Kubernetes by creating a pod and deploying multiple instances using replicasets. But deploying and managing the number of replicas won't cut it when it comes to deploying applications for production use cases.

<click> when newer versions of application is released, you would like to UPGRADE your application instances seamlessly.

when you upgrade your instances, you may want to upgrade them one after the other. And that kind of upgrade is known as Rolling Updates.

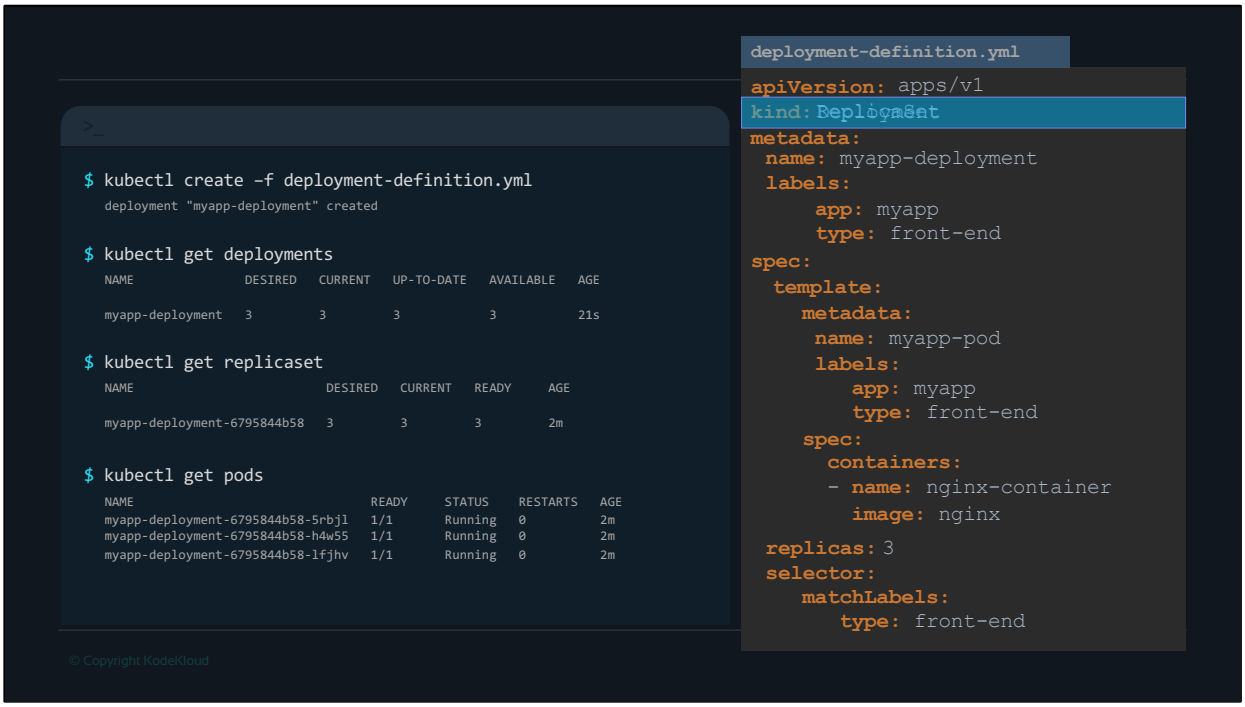
Suppose one of the upgrades you performed resulted in an unexpected error and you are asked to undo the recent update. You would like to be able to <click> rollBACK the changes that were recently carried out.

Finally, say for example you would like to make multiple changes to your environment such as upgrading the underlying WebServer versions, as well as scaling your environment and also modifying the resource allocations etc. You do not want each change to be applied immediately after the command is run, instead you would like

to apply a <click> pause to your environment, make the changes and then resume <click> so that all changes are rolled-out together.

<click> All of these capabilities are available with the kubernetes Deployments.

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



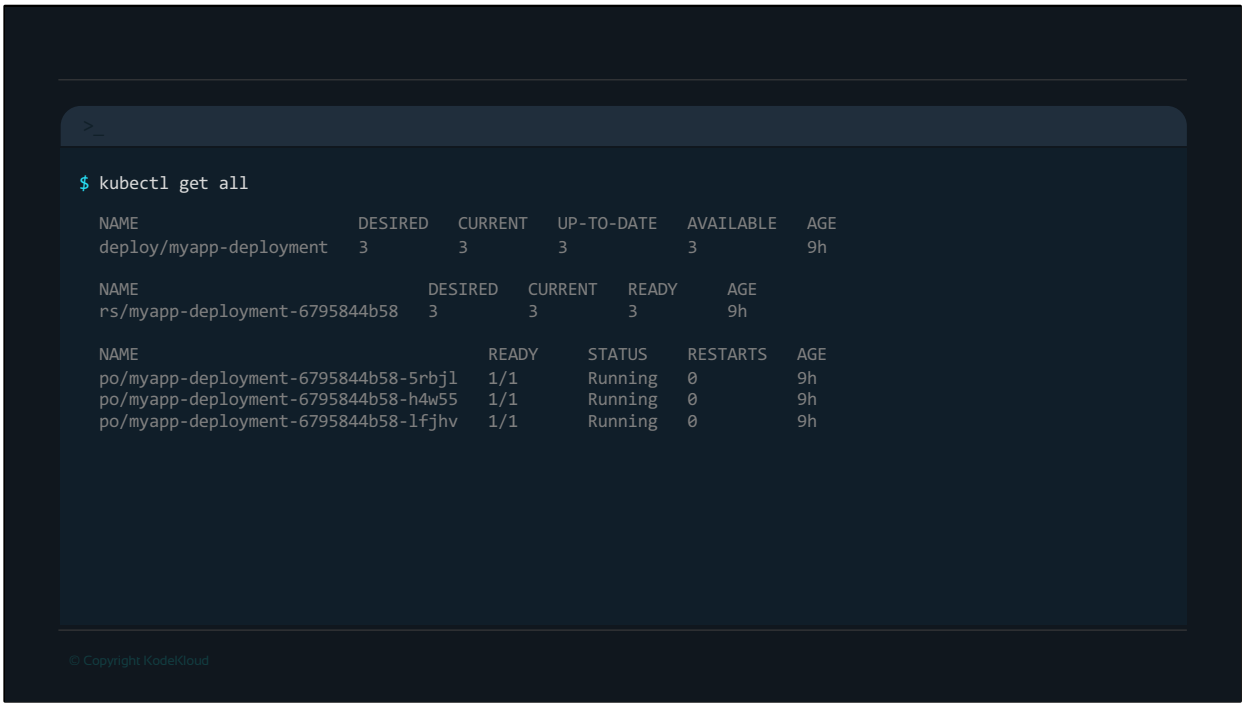
So how do we create a deployment. <click> As with the previous components, we first create a deployment definition file. The contents of the deployment-definition file are exactly similar to the replicaset definition file, except for the kind, which is now going to be Deployment.

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

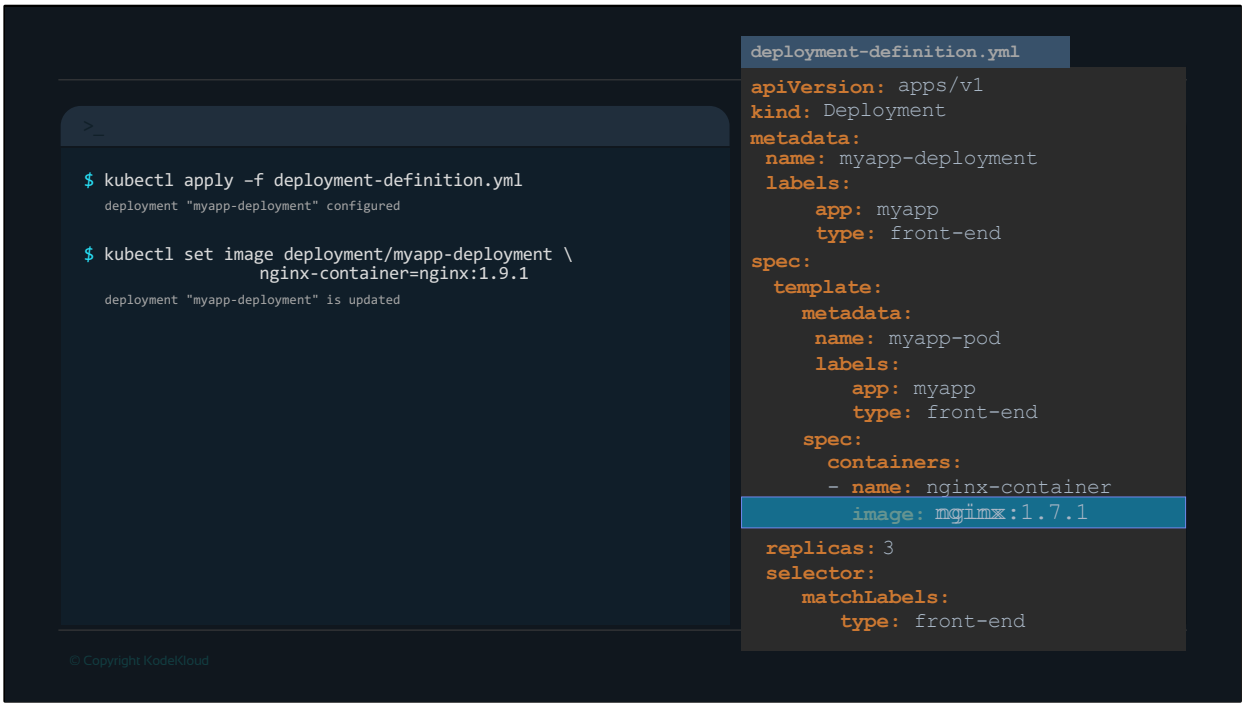
<click> Once the file is ready run the kubectl create command and specify deployment definition file. <click> Then run the kubectl get deployments command to see the newly created deployment. The deployment automatically creates a replica set. <click> So if you run the kubectl get replcaset command you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

So far there hasn't been much of a difference between replicaset and deployments, except for the fact that deployments created a new kubernetes object called

deployments. We will see how to take advantage of the deployment using the use cases we discussed in the previous slide in the upcoming lectures.



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

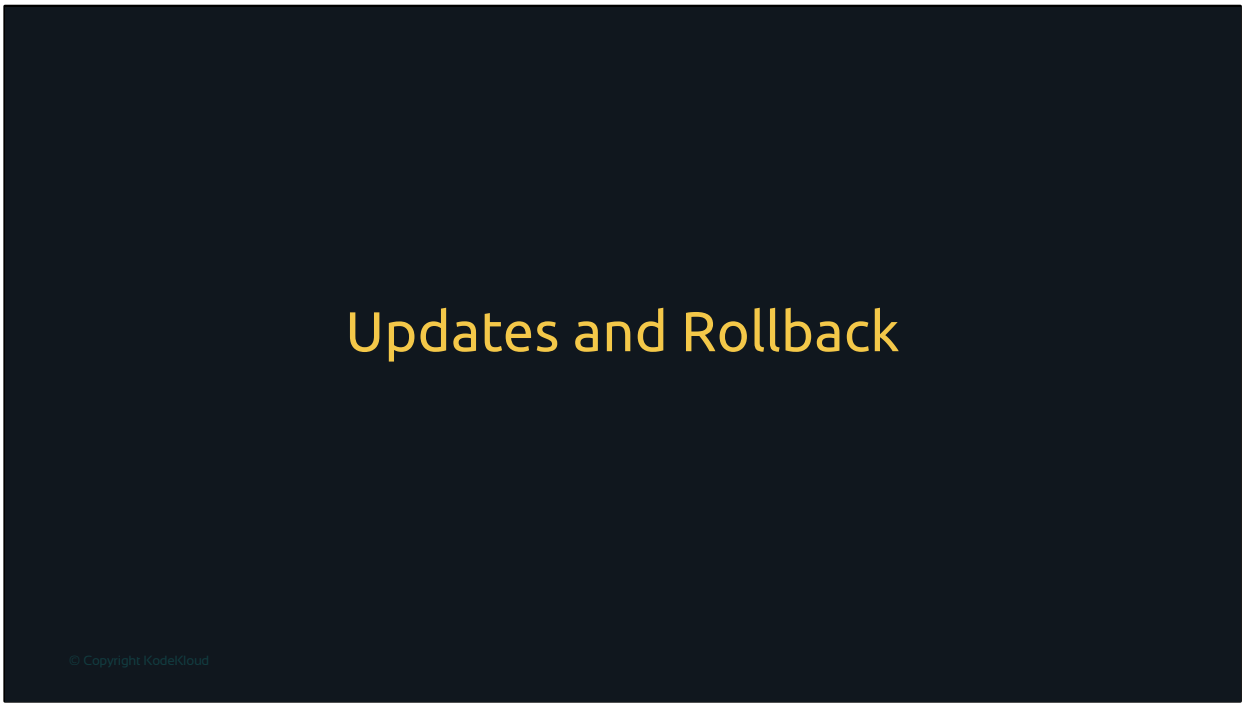


Now once a deployment is created and you have a newer version of the app available, how do you upgrade your application? As before one way is to updated the deployment definition file to update the new image name with the newer version of the app.

The imperative approach would be to use the kubectl set image command and specify the deployment name and the image name like this.

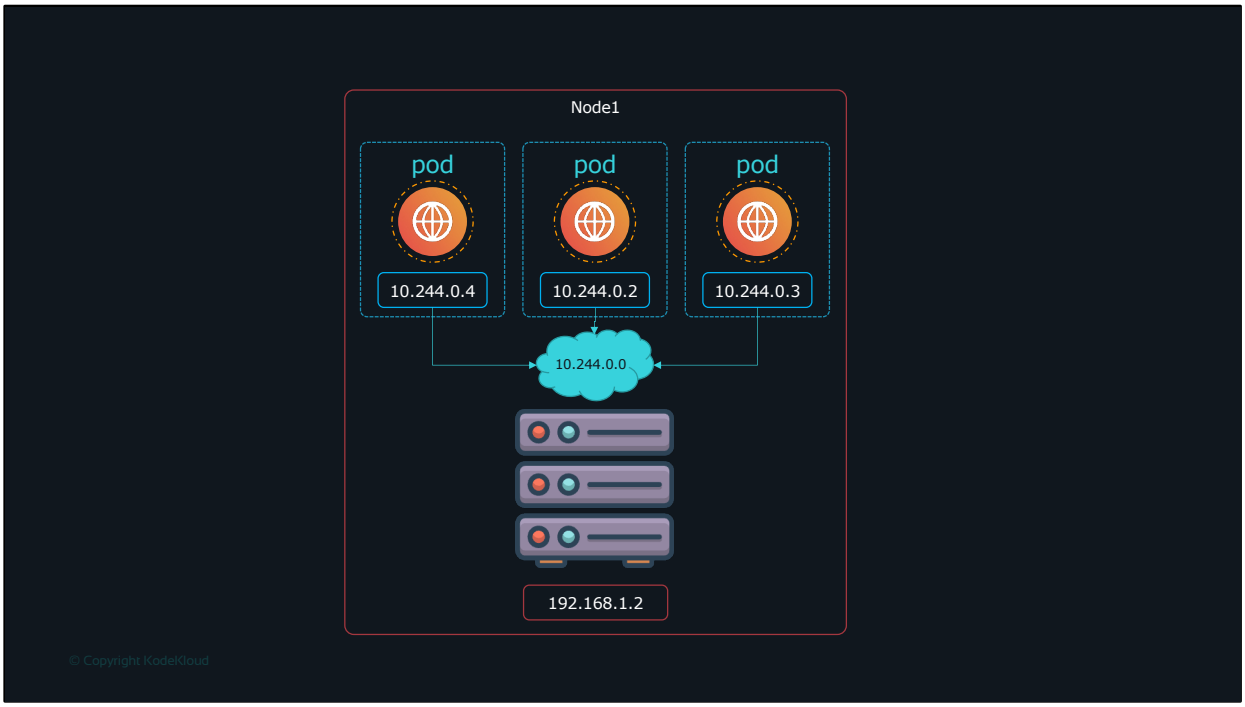
It's time for labs activity. Click on the link to go directly to the labs. If you haven't enrolled already enroll for free. In this lab you will work on creating deployments and deploying applications to a kubernetes cluster.

https://kodekloud.com/topic/labs-pods-with-yaml/


Updates and Rollback
© Copyright KodeKloud

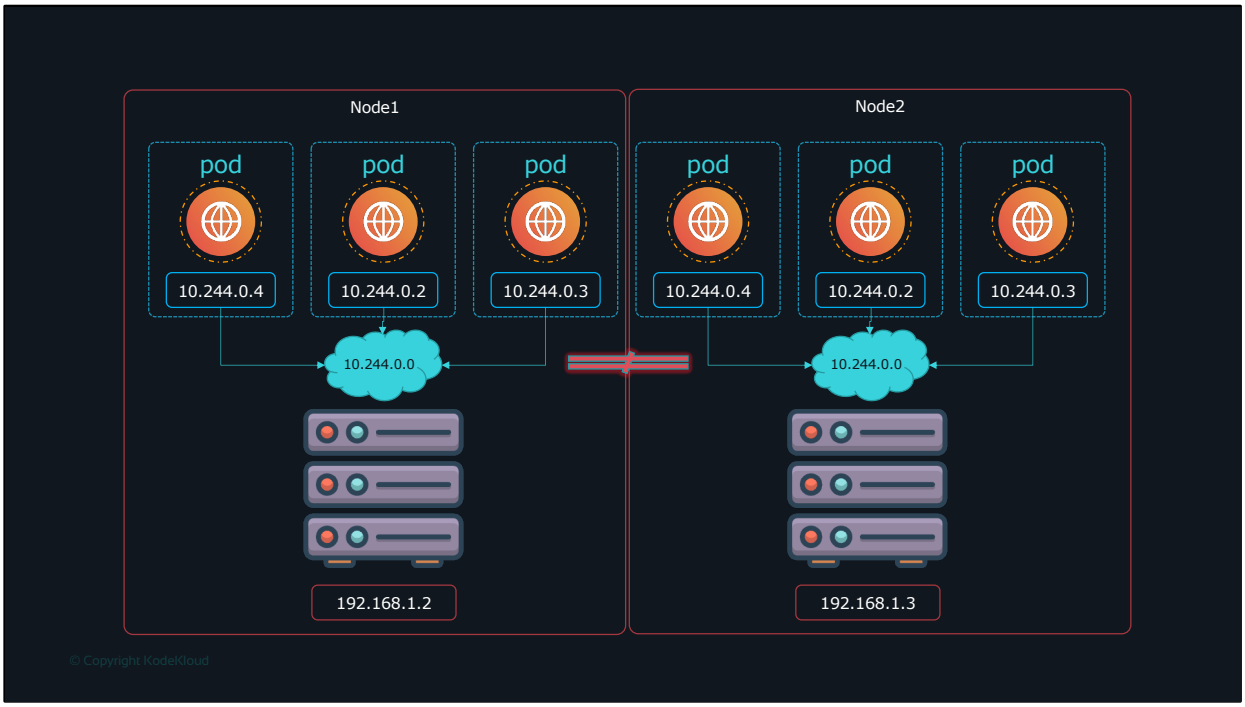

© Copyright KodeKloud

Let us now talk about Networking 101 with Kubernetes.



Let us look at the very basics of networking in Kubernetes. <click> We will start with a single node kubernetes cluster. The node has an IP address, say it is 192.168.1.2 in this case. This is the IP address we use to access the kubernetes node, SSH into it etc.

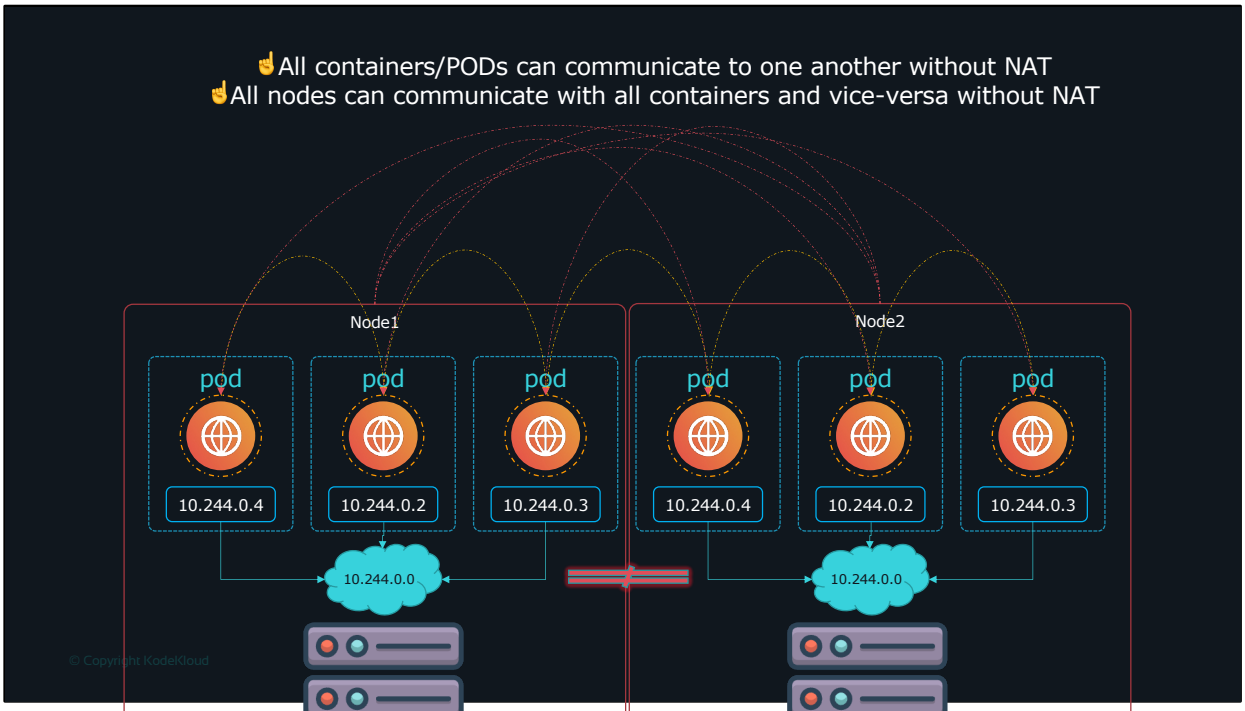
So on the single node kubernetes cluster we have created a Single POD. As you know a POD hosts a container. Unlike in the docker world were an IP address is always assigned to a Docker CONTAINER, <click>
in Kubernetes the IP address is assigned to a POD. Each POD in kubernetes gets its own internal IP Address. In this case its in the range 10.244 series and the IP assigned to the POD is 10.244.0.2. So how is it getting this IP address? When Kubernetes is initially configured it creates an internal private network with the address 10.244.0.0 <click>
and all PODs are attached to it. <click> When you deploy multiple PODs, they all get a separate IP assigned. The PODs can communicate to each other through this IP. But accessing other PODs using this internal IP address MAY not be a good idea as its subject to change when PODs are recreated. We will see BETTER ways to establish communication between PODs in a while. For now its important to understand how the internal networking works in kubernetes.



So it's all easy and simple to understand when it comes to networking on a single node. But how does it work when you have multiple nodes in a cluster? In this case we have two nodes running kubernetes and they have IP addresses 192.168.1.2 and 192.168.1.3 assigned to them. Note that they are not part of the same cluster yet. <click> Each of them has a single POD deployed. As discussed in the previous slide these pods are attached to an internal network and they have their own IP addresses assigned. HOWEVER, if you look at the network addresses, <click> you can see that they are the same. The two networks have an address 10.244.0.0 and the PODs deployed have the same address too.

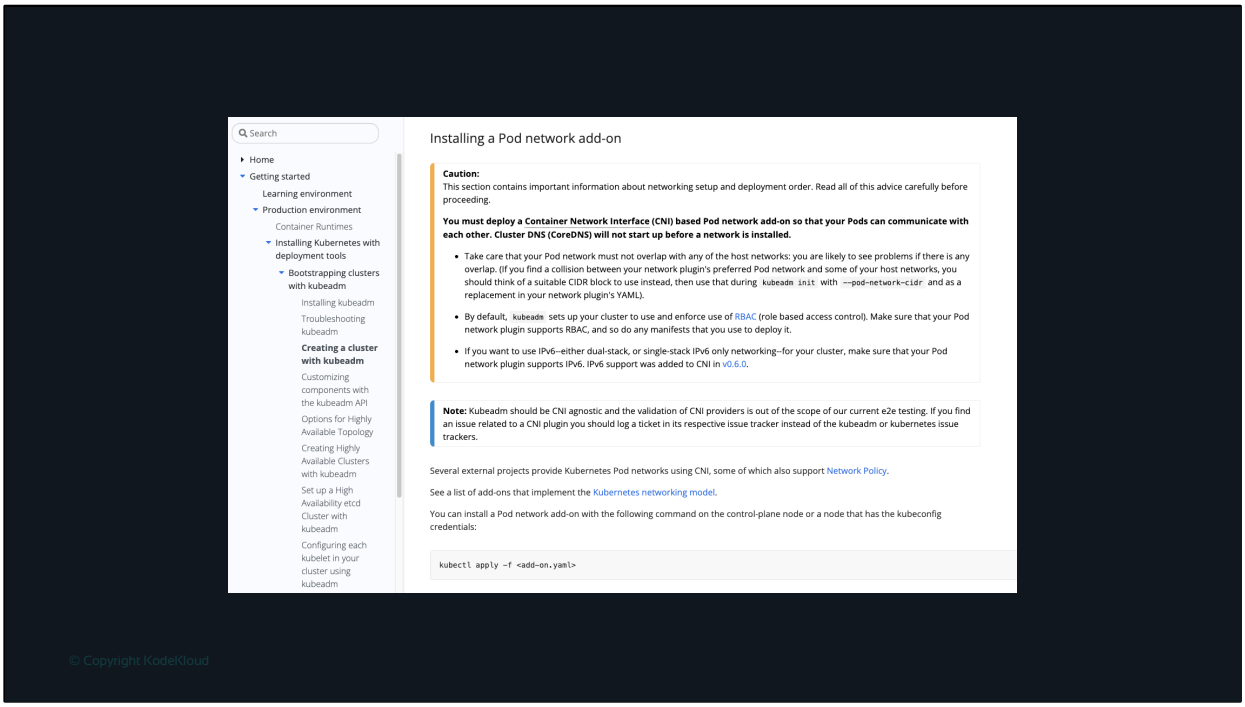
<click> This is NOT going to work well when the nodes are part of the same cluster. The PODs have the same IP addresses assigned to them and that will lead to IP conflicts in the network. Now that's ONE problem. When a kubernetes cluster is SETUP, kubernetes does NOT automatically setup any kind of networking to handle these issues. As a matter of fact, kubernetes expects US to setup networking to meet certain fundamental requirements. Some of these are that <click> all the containers or PODs in a kubernetes cluster MUST be able to communicate with one another without having to configure NAT. <click> All nodes must be able to communicate with containers and all containers must be able to communicate with the nodes in the

cluster. Kubernetes expects US to setup a networking solution that meets these criteria.

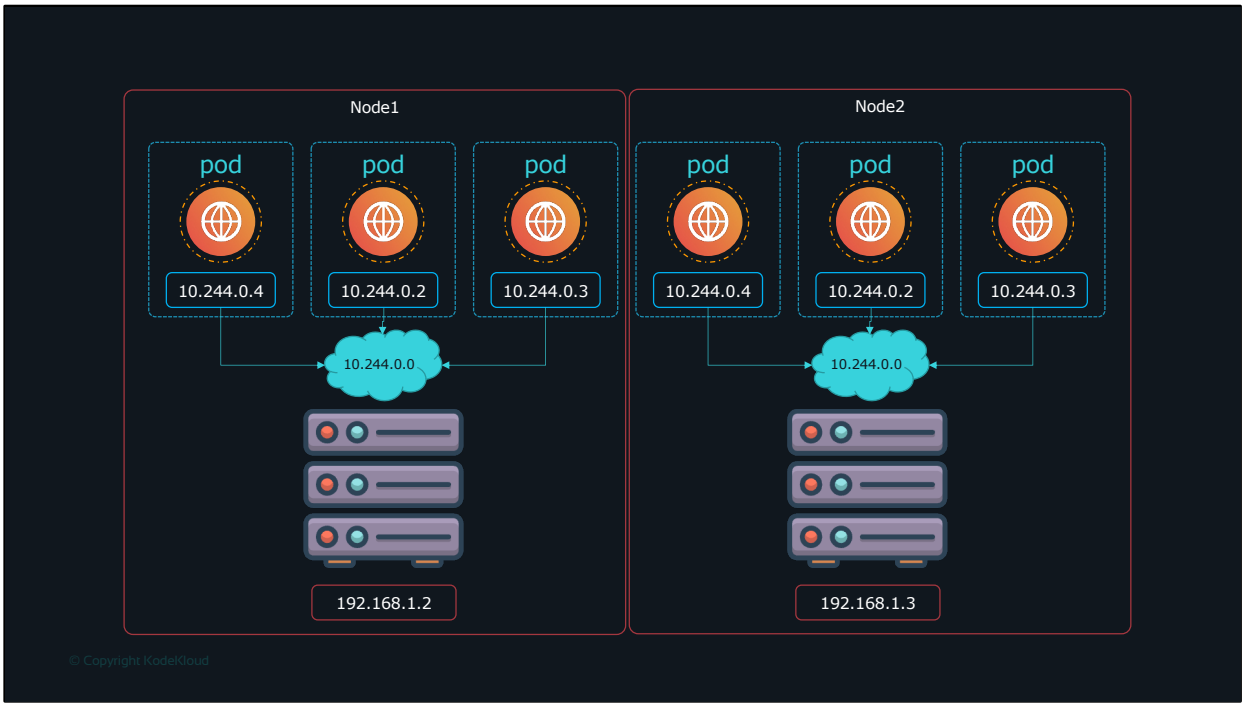


Some of these are th[illegible] all the containers or POD[illegible]ernetes cluster MUST be able to communicate with one another without having to configure NAT. <click> All nodes must be able to communicate with containers and all containers must be able to communicate with the nodes in the cluster. Kubernetes expects US to setup a networking solution that meets these criteria.

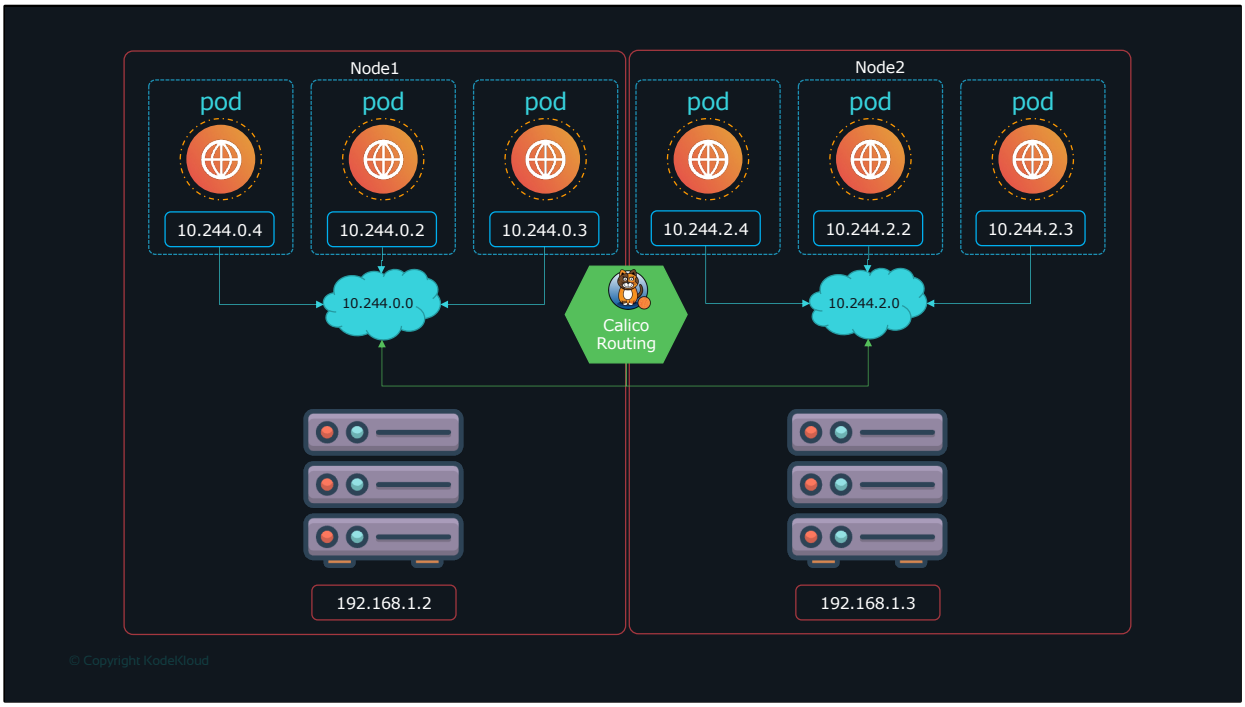
Fortunately, we don’t have to set it up ALL on our own as there are multiple pre-built solutions available. Some of them are the cisco ACI networks, Cilium, Big Cloud Fabric, Flannel, Vmware NSX-t and Calico. Depending on the platform you are deploying your Kubernetes cluster on you may use any of these solutions. For example, if you were setting up a kubernetes cluster from scratch on your own systems, you may use any of these solutions like Calico, Flannel etc. If you were deploying on a Vmware environment NSX-T may be a good option. If you look at the play-with-k8s labs they use WeaveNet. In our demos in the course we used Calico. Depending on your environment and after evaluating the Pros and Cons of each of these, you may chose the right networking solution.



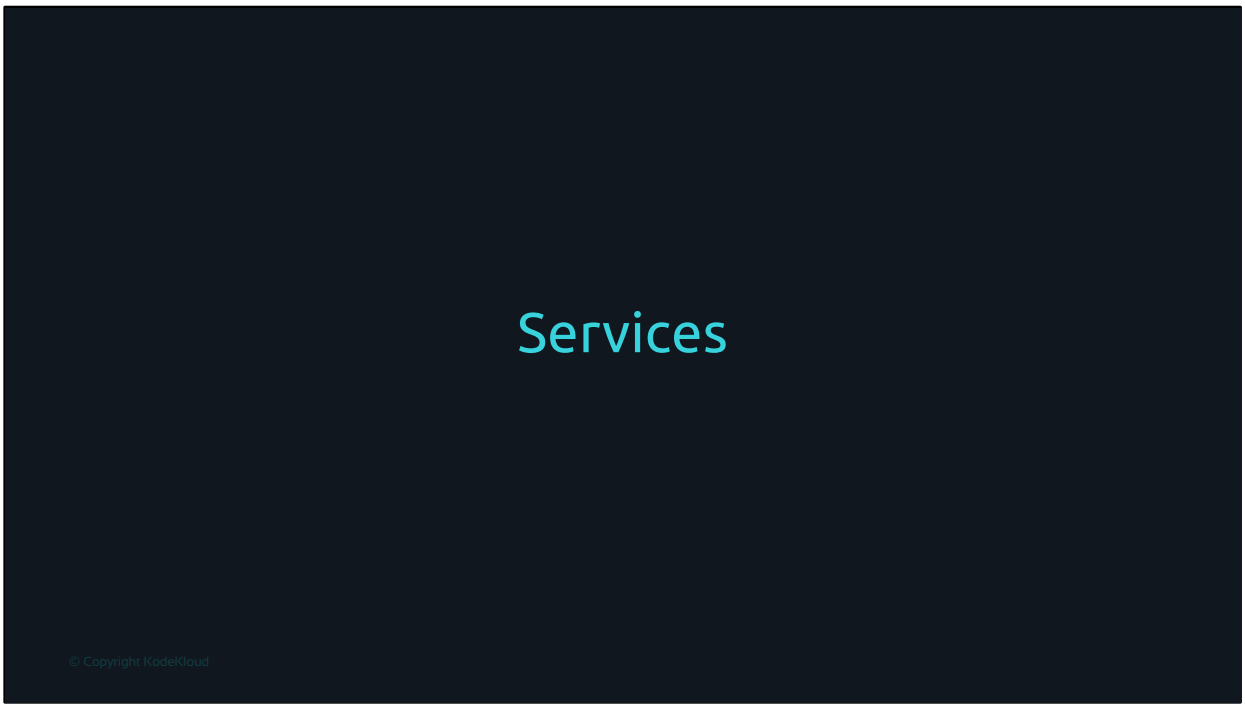
The step to install a pod network add-on is part of the kubernetes cluster creation process.

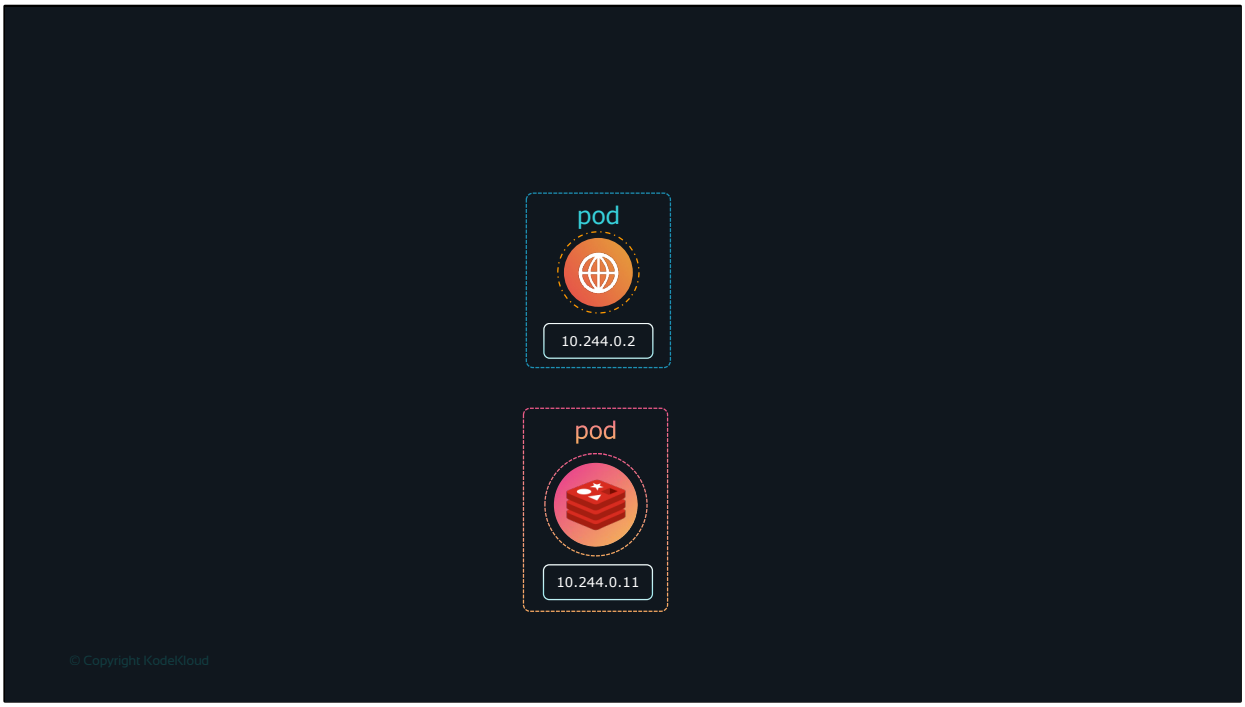


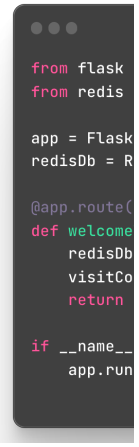
So back to our cluster, with the <click> Calico networking setup,



it now manages the networks and Ips in my nodes and assigns a different network address for each network in the nodes. This creates a virtual network of all PODs and nodes were they are all assigned a unique IP Address. And by using simple routing techniques the cluster networking enables communication between the different PODs or Nodes to meet the networking requirements of kubernetes. Thus all PODs can now communicate to each other using the assigned IP addresses.


Services
© Copyright KodeKloud



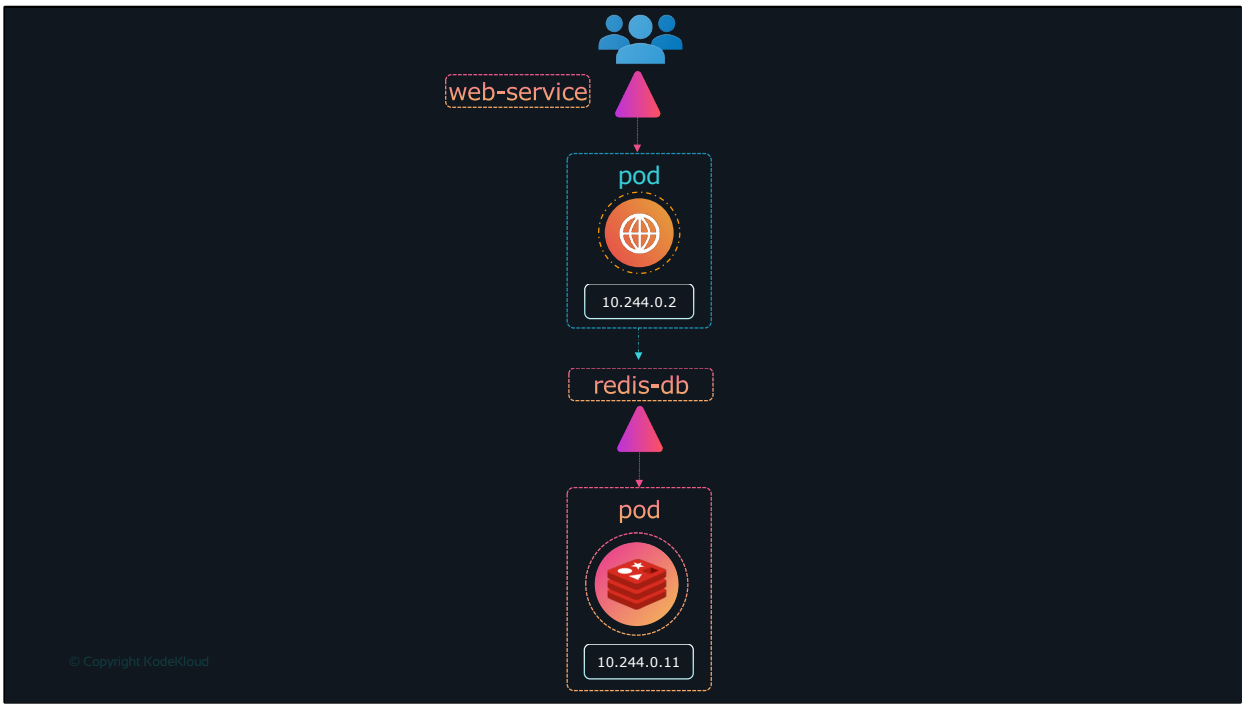


So we have two sets of services deployed in our application. A webserver and a redis service. Kubernetes assigns a unique IP address to each pod in the cluster. The webserver has 10.244.0.2 and the pod has 10.244.0.11

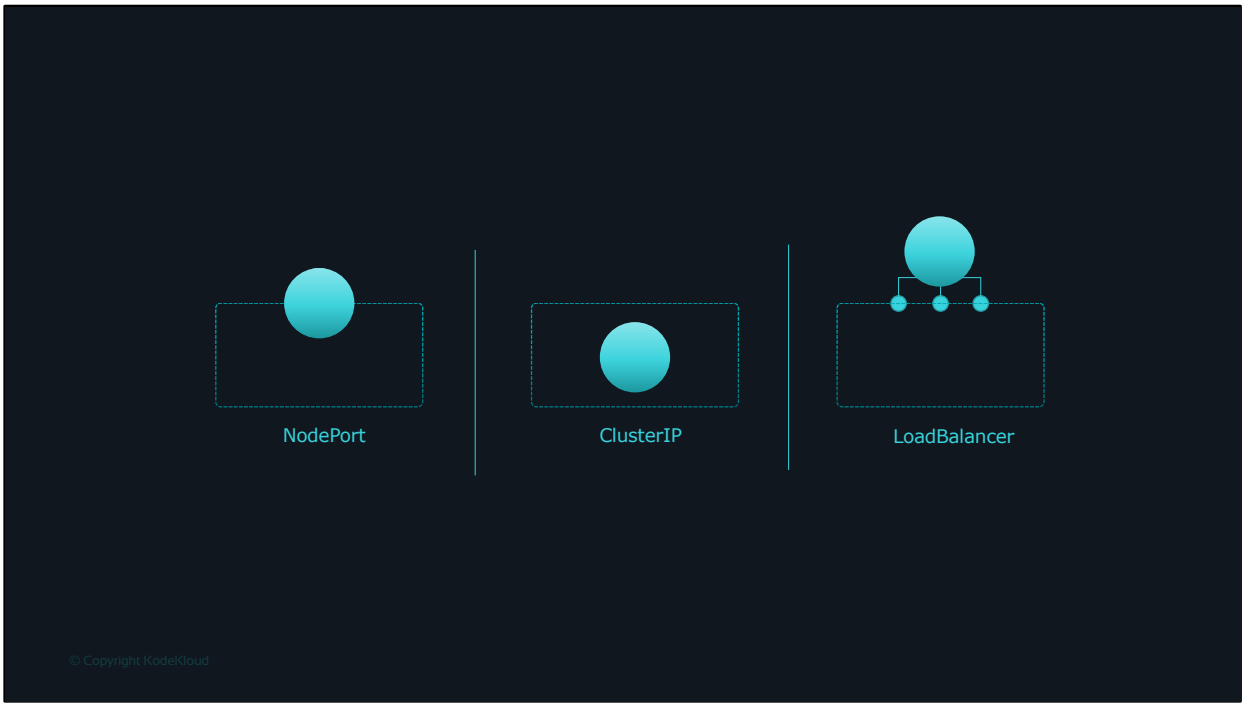
The web server needs to access the redis service.

So what would the webserver address the redis deployment as? The redis pod has an IP address. Can the webserver address the redis service using it's IP address? It can, but It shouldn't because the IP is for each POD and it is bound to change if the pod where to crash or restart for some reason. That's where a service comes in. A service enables communication between applications within a kubernetes cluster. Think of a service as a proxy or a load balancer – although it technically is not in a traditional sense. And it provides an endpoint for other services to connect to.

In this case we create a service named redis-db and now the web application can refer tot his service with the name redis-db.



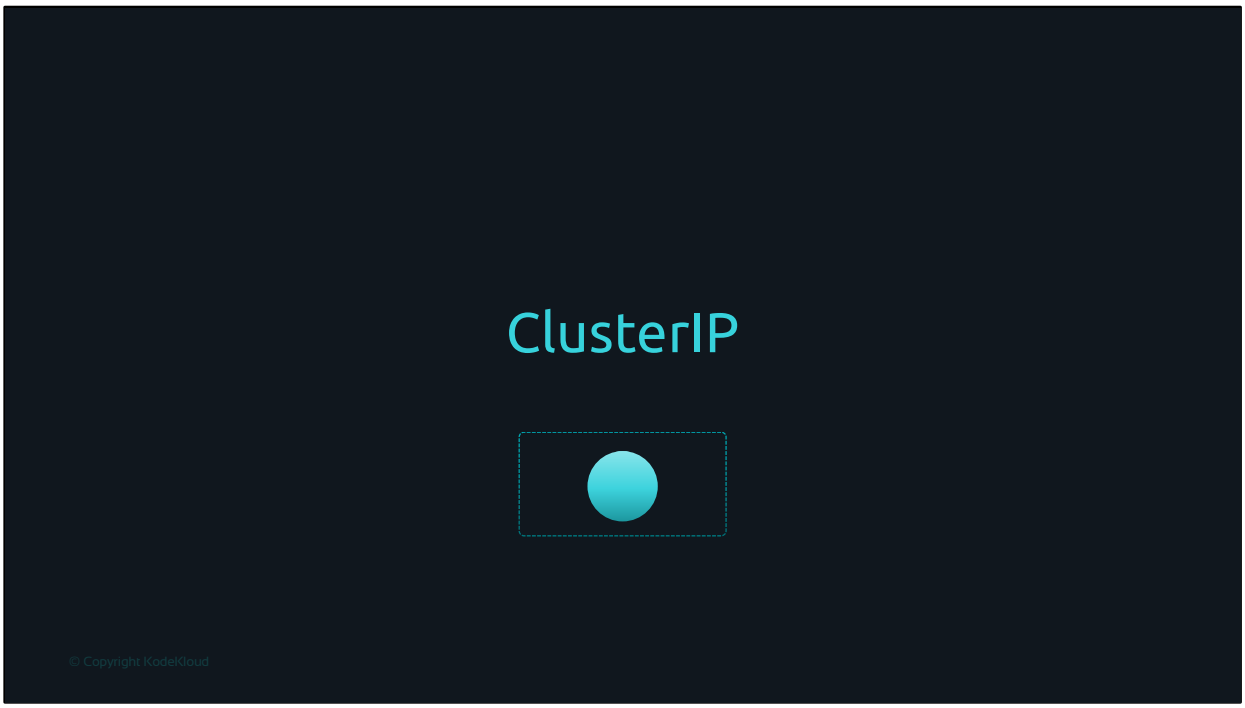
Similarly to expose the webservice outside to the external users you would create another service for the web server. We will call it the web-service. So a service enables connectivity between applications within the cluster as well as to expose applications outside the cluster to end users. We'll see how to create service in a few minutes, but first let's understand the different kinds of services.



The first one we discussed is the clusterip service. This is a service within the cluster that is not exposed externally and helps different services communicate with each other. This is the example we saw about the web server reaching the redis service. the redis db service is a clusterIP type of service.

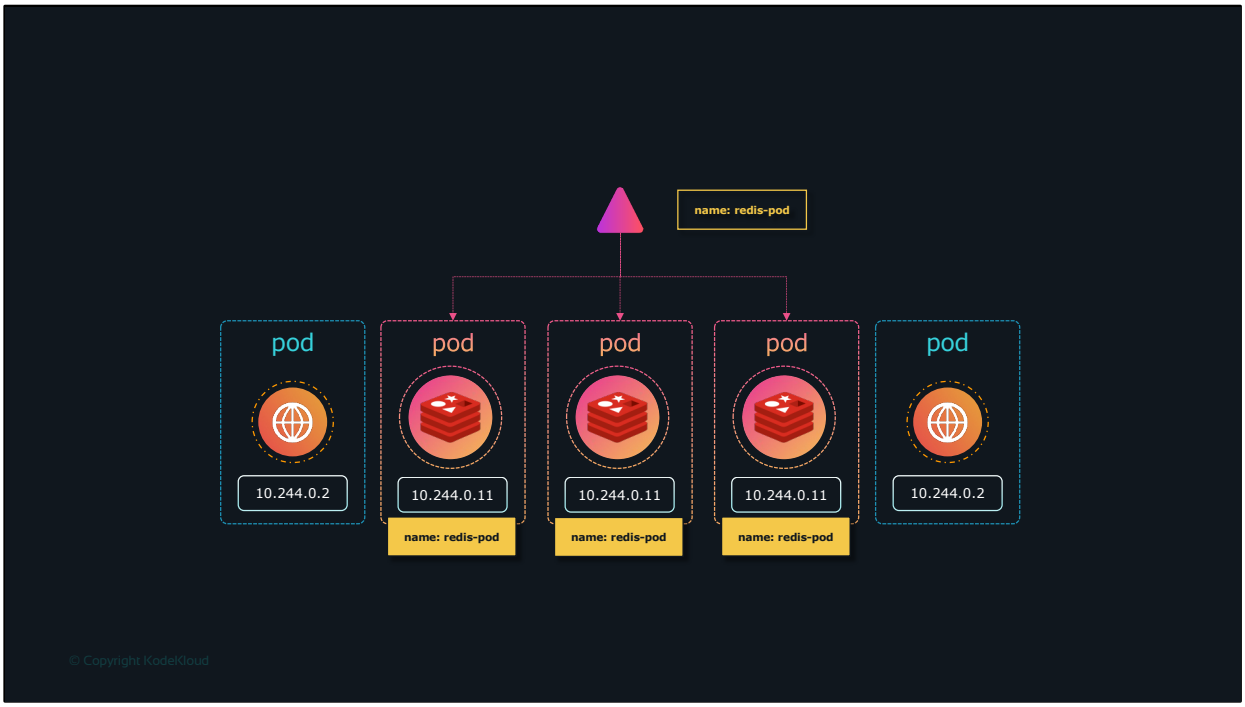
The second is NodePort – <click> and in this case the service exposes the application on a port on the Node to be made accessible to external users.

<click> The third type is a LoadBalancer, were it provisions a load balancer for our service in supported cloud providers – like Google Cloud, AWS or Azure. A good example of that would be to distribute load across different web servers. In the scope of this course we will look at the ClusterIP and NodePort services.



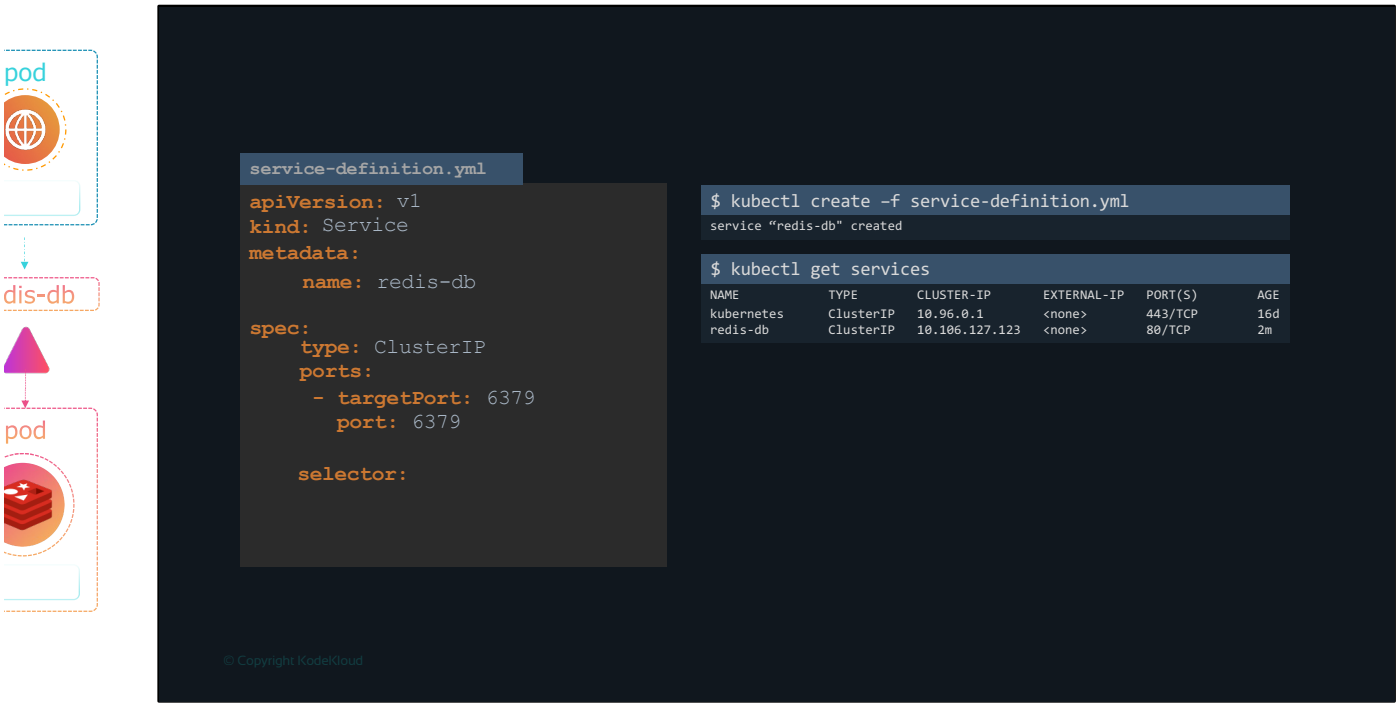
Let's look at the ClusterIP type of service.

The service we talked about earlier – where the web server tries to connect to the redis service is the ClusterIP tyep of service. This is pretty straight forward.



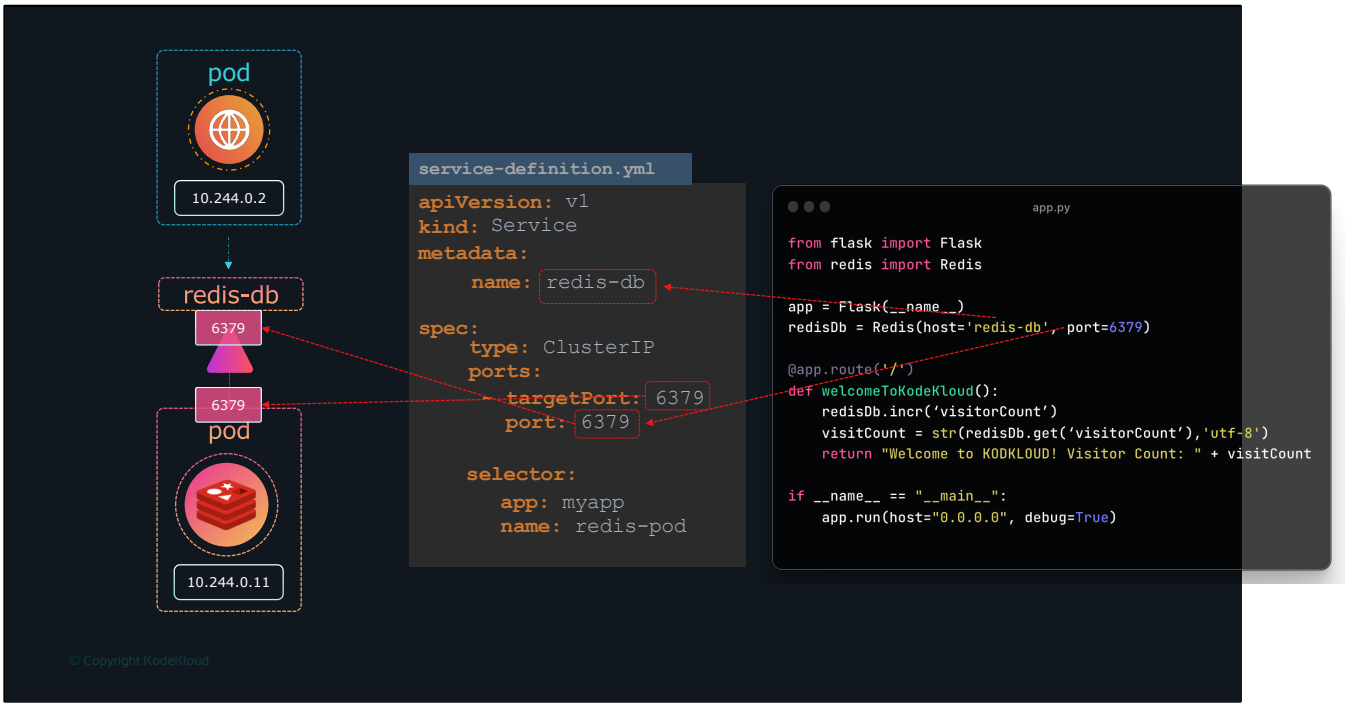
So here we have a redis pod that needs to be exposed within the cluster for the web application. We do that by <c> creating a service. But we know that pods are usually deployed in replicas. <c> Multiple instances. <c> And there could 100s of other pods in the cluster. How can a service identify which are the pods that it should be routing traffic to?

Again same as before we have labels and selectors. The pods have a label with the name set to redis-pod. We define the same label as a selector on the service. <c> The service identifies all pods with the same label and configures them as it's endpoints.



To create such a service, as always, use a definition file. In the service definition file , first use the default template which has apiVersion, kind, metadata and spec. The apiVersion is v1 <click> , kind is Service and we will give a name to our service – we will call it redis-db. <click> Under spec we have type and ports. The type is ClusterIP. In fact, ClusterIP is the default type, so even if you didn't specify it, it will automatically assume it to be ClusterIP. Under ports we have a targetPort and port. The target port is the port were the back-end is exposed, which in this case is 6379. And the port is were the service is exposed. Which is 6379 as well. I'll explain that in a bit more detail in a sec <click> To link the service to a set of PODs, we use selector.

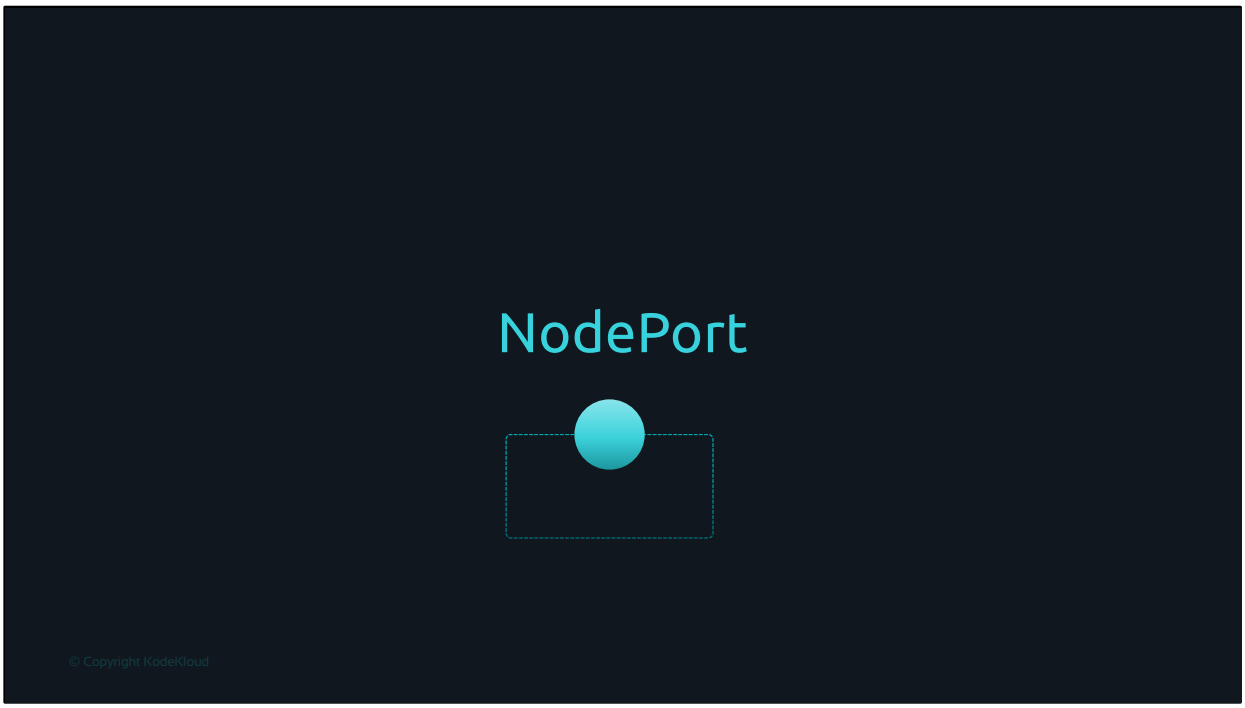
We will refer to the pod-definition file <click> and copy the labels from it and move it under selector. And that should be it. <click> We can now create the service using the kubectl create command and then check its status using <click> the kubectl get services command. The service can be accessed by other PODs using the ClusterIP or the service name. Preferebly the service name.



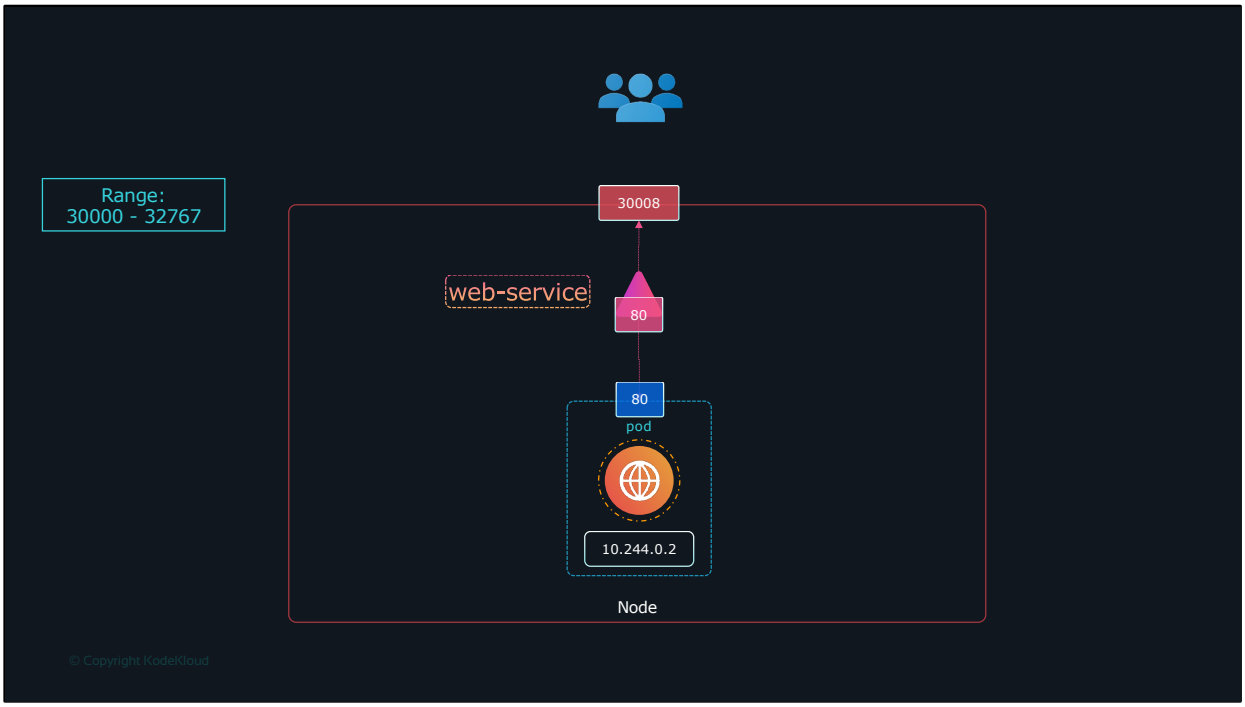
So let's talk about ports. When creating a service we must specify what port the application running inside the pod is listening on. And that's defined as the target port here.

We also need to specify which port the service must serve on. And these could be different. The applications could be listening on one port and the service could be listening on another. However in this case since any application connecting to redis expects it to be at 6379 we are going to stick to the same port.

So if you look at the code of the webserver, to connect to the redis service it must use the name of the service as the host which in this case is redis-db. And use the same port defined as the port on the service. Which in this case is 6379.

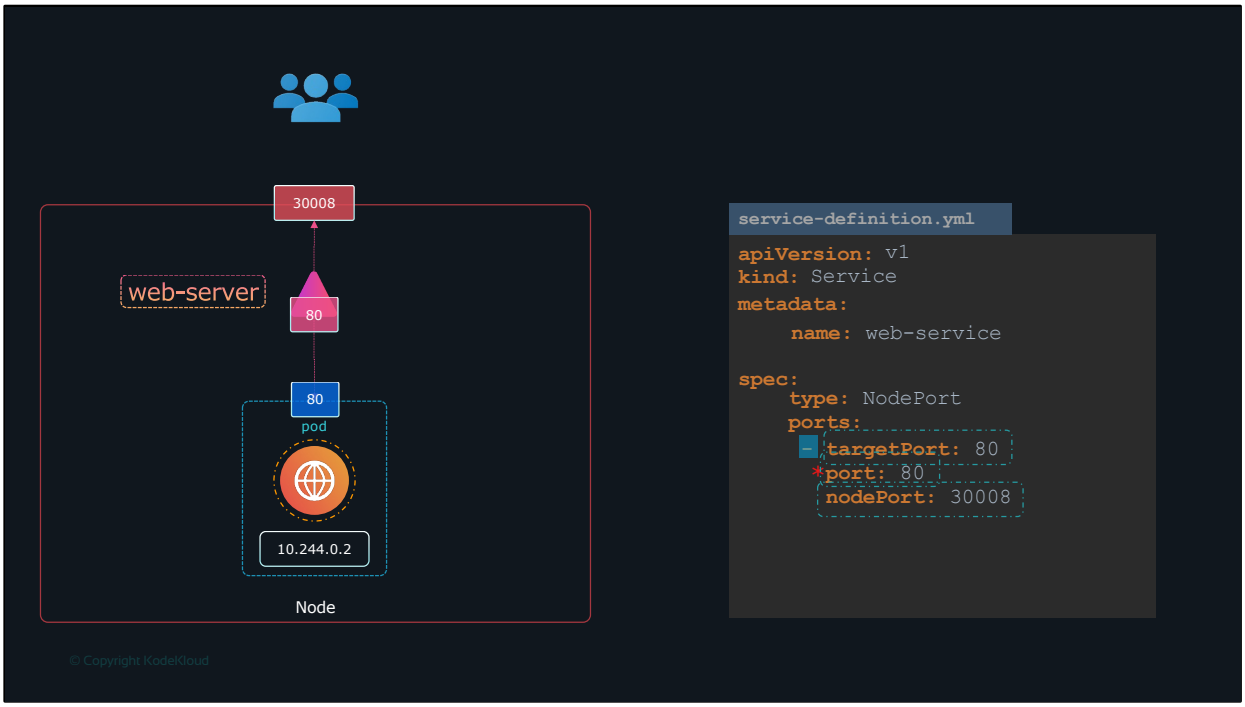


Let's look at what a NodePort service is.



A nodeport is a type of service where a <c> normal service is created first. And is then exposed to external users through a port on the node.

Let’s take a closer look at the Service. If you look at it, there are 3 ports involved. The port on the POD were the actual web server is running is port <click> 80. And it is referred to as the targetPort, because that is were the service forwards the requests to. The second port is the port on the <click> service itself. It is simply referred to as the port. Remember, these terms are from the viewpoint of the service. And finally we have the port on the Node itself <click> which we use to access the web server externally. And that is known as the NodePort. As you can see it is 30008. That is because NodePorts can only be in a valid range which is from 30000 to 32767.

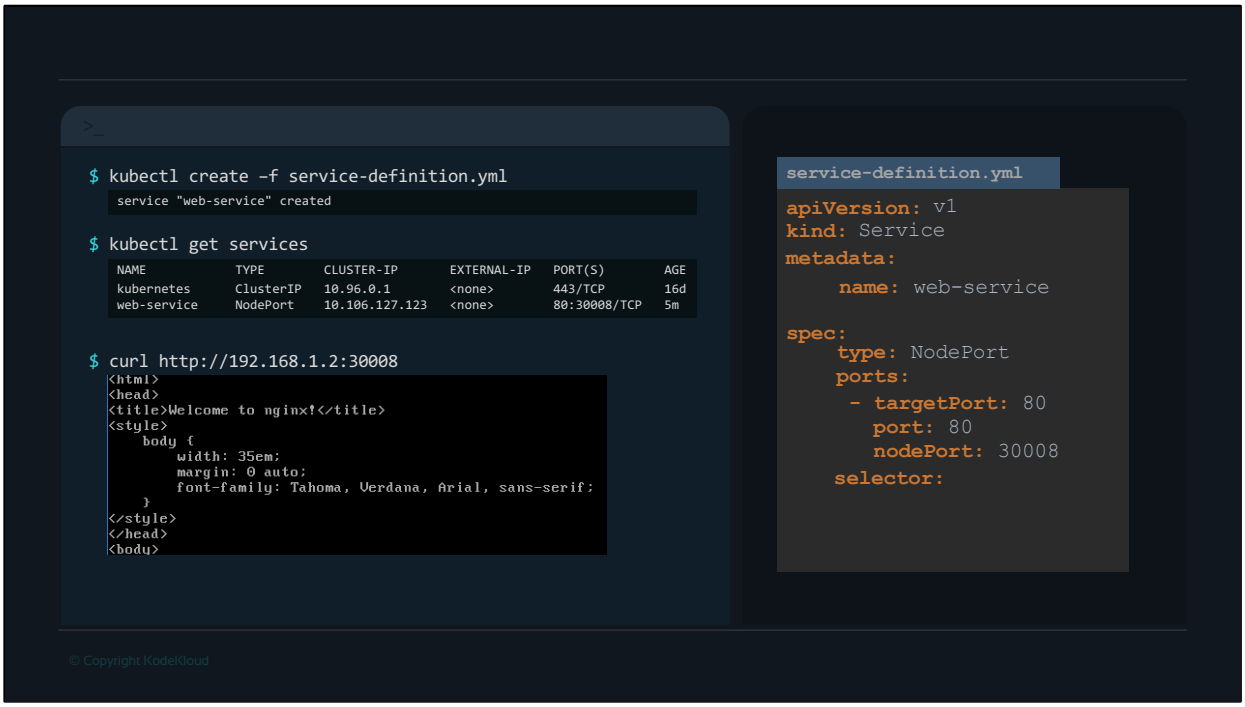


Let us now look at how to create the service. As before we will use a definition file to create a service.
The high level structure of the file remains the same. <click> we have apiVersion, kind, metadata and spec sections. <click> The apiVersion is going to be v1. <click> The kind is ofcourse service. <click> The metadata will have a name and that will be the name of the service. It can have labels, but we don't need that for now. Next we have spec. and as always this is the most crucial part of the file as this is were we will be defining the actual services and this is the part of a definition file that differs between different objects. <click> In the spec section of a service we have type and ports. The type refers to the type of service we are creating. As discussed before it could be ClusterIP, NodePort, or LoadBalancer. In this case since we are creating a NodePort we will set it as NodePort. The next part of spec is ports. This is were we input information regarding what we discussed on the left side of this screen. The first type of port is the targetPort, <click> which we will set to 80. The next one is simply port, <click> which is the port on the service object and we will set that to 80 as well. The third is NodePort <click> which we will set to 30008 or any number in the valid range. Remember that out of these, the only mandatory field is port <click>. If you don't provide a targetPort it is assumed to be the same as port and if you don't provide a nodePort a free port in the valid range between 30000 and 32767 is automatically

allocated. Also note that ports is an array. So note the dash <click> under the ports section that indicate the first element in the array. You can have multiple such port mappings within a single service.

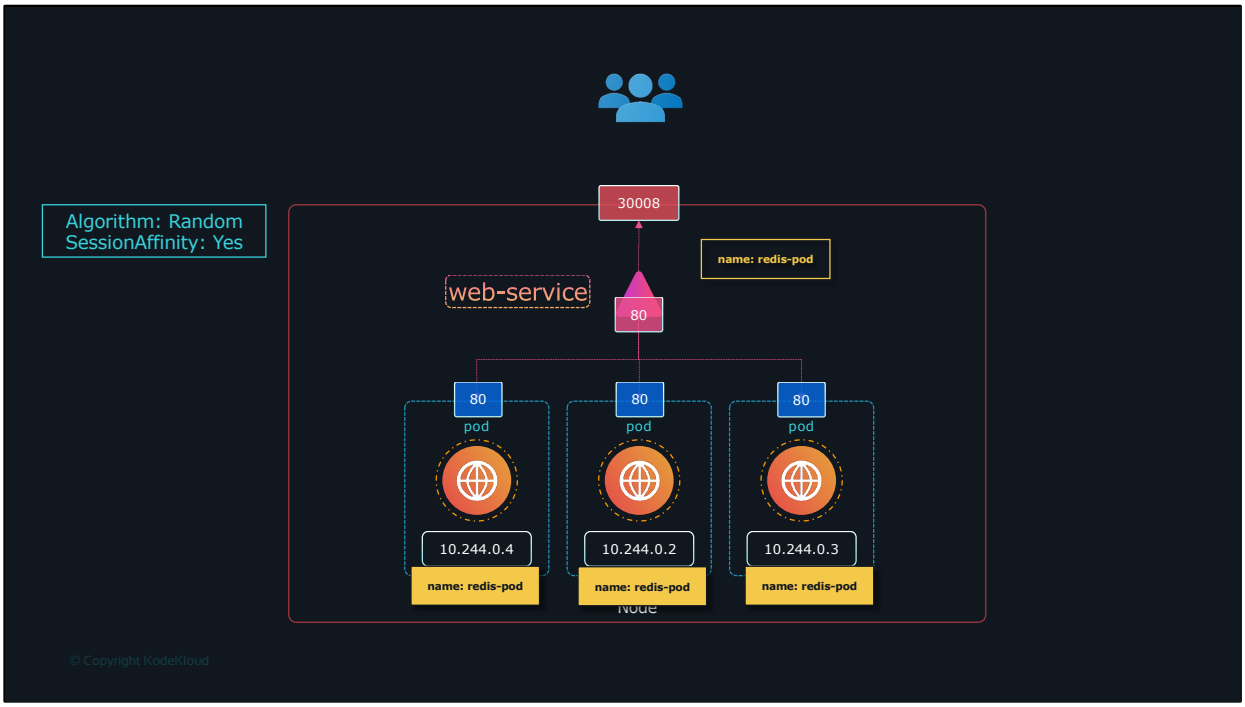
So we have all the information in, but something is really missing. There is nothing here in the definition file that connects the service to the POD. We have simply specified the targetPort but we didn't mention the targetPort on which POD. There could be 100s of other PODs with web services running on port 80. So how do we do that?

As we did with the replicasets previously and a technique that you will see very often in kubernetes, we will use labels and selectors to link these together. We know that the POD was created with a label. We need to bring that label into this service definition file.



So we have a new property in the spec section and that is <click> selector. Under the selector provide a list of labels to identify the POD. <click> For this refer to the pod-definition file used to create the POD. <click> Pull the labels from the pod-definition file and place it under the selector section. This links the service to the pod. <click> Once done create the service using the kubectl create command and input the service-definition file and there you have the service created.

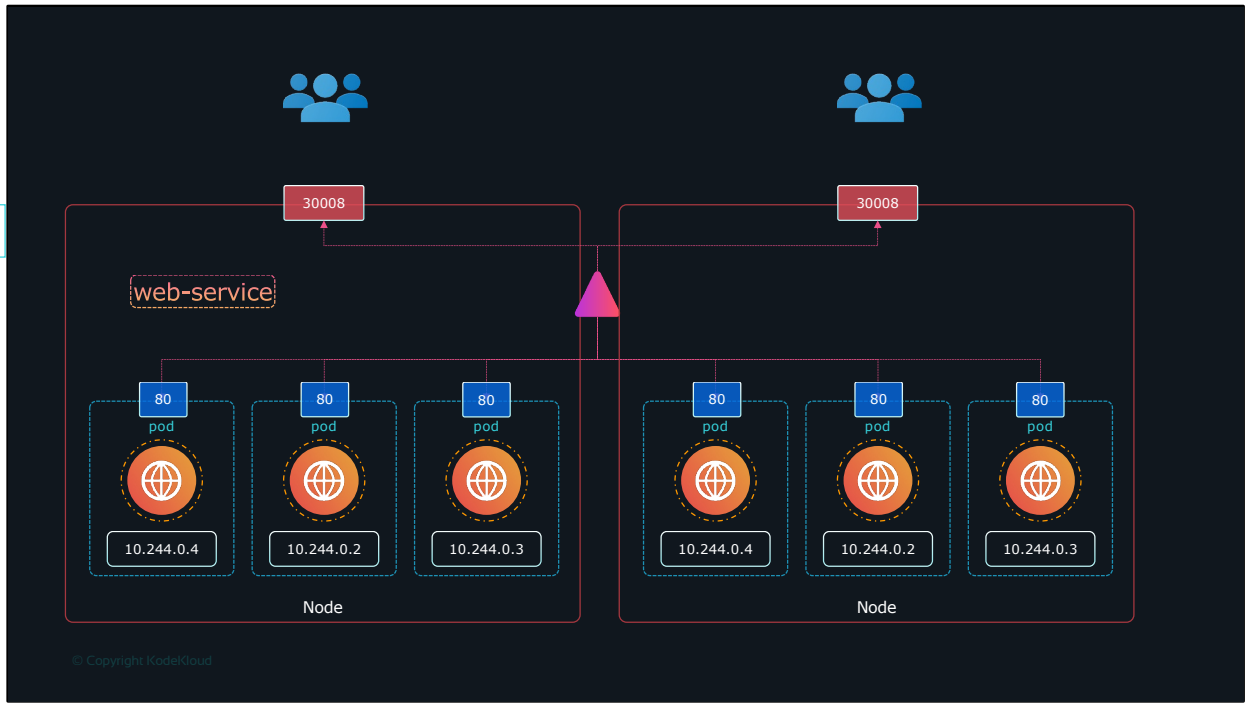
To see the created service, run the kubectl get services command that lists the services, their cluster-ip and the mapped ports. The type is NodePort as we created and the port on the node automatically assigned is 32432. We can now use this port to access the web service using curl or a web browser.



So far we talked about a service mapped to a single POD. But that's not the case all the time, what do you do when you have multiple PODs? <click> In a production environment you have multiple instances of your web application running for high-availability and load balancing purposes.

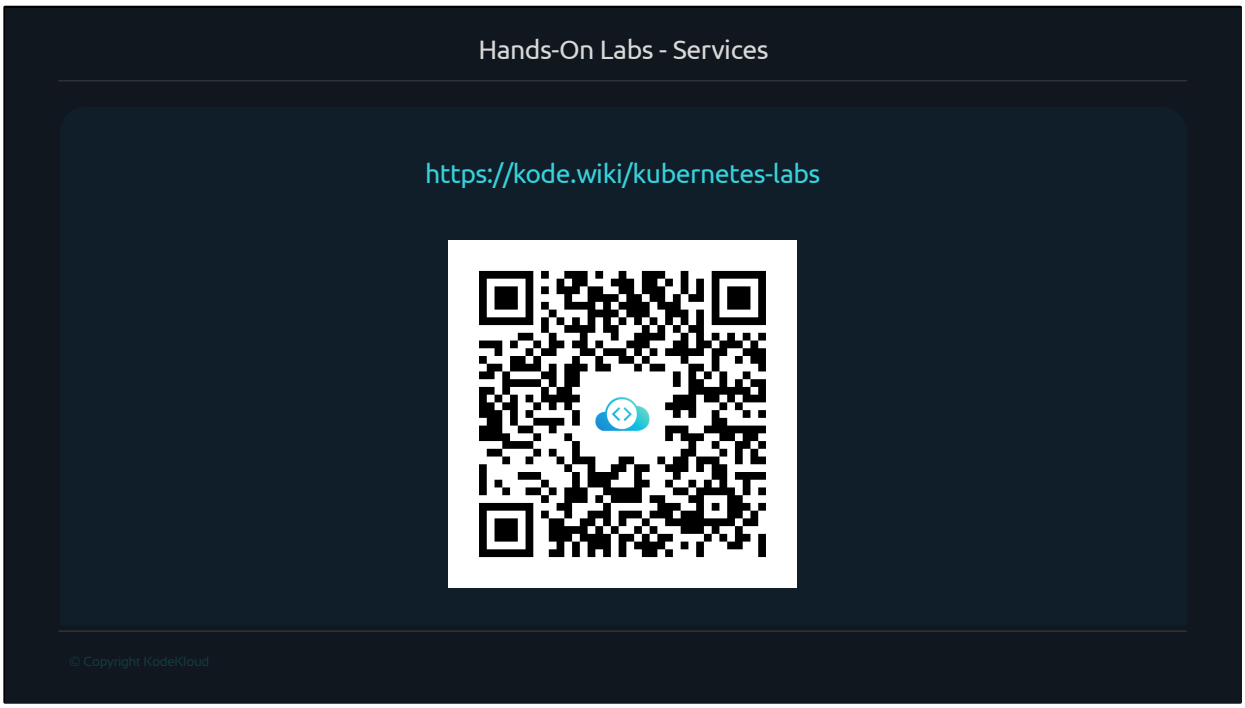
In this case we have multiple similar PODs running our web application. <click> They all have the same labels with a key name set to redis-pod. The same label is used as a selector during the creation of the service. So when the service is created, it looks for matching PODs with the labels and finds 3 of them. <click> The service then automatically selects all the 3 PODs as endpoints to forward the external requests coming from the user. You don't have to do any additional configuration to make this happen. <click> And if you are wondering what algorithm it uses to balance load, it uses a random algorithm. Thus, the service acts as a built-in load balancer to distribute load across different PODs.

ithm: Random
onAffinity: Yes



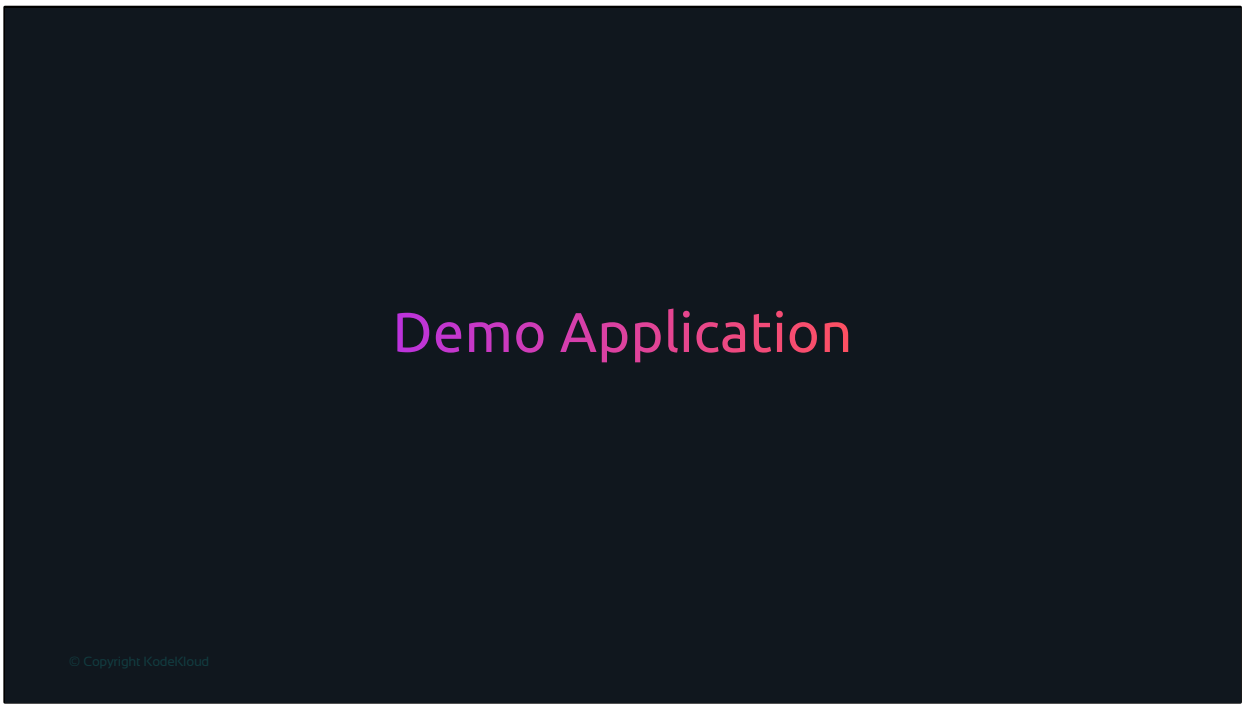
And finally, lets look at what happens when the PODs are distributed across multiple nodes. In this case we have the web application on PODs on separate nodes in the cluster. When we create a service , without us having to do ANY kind of additional configuration, <click> kubernetes creates a service that spans across all the nodes in the cluster and maps the target port to the SAME NodePort on all the nodes in the cluster. <click> This way you can access your application using the IP of any node in the cluster and using the same port number which in this case is 30008.

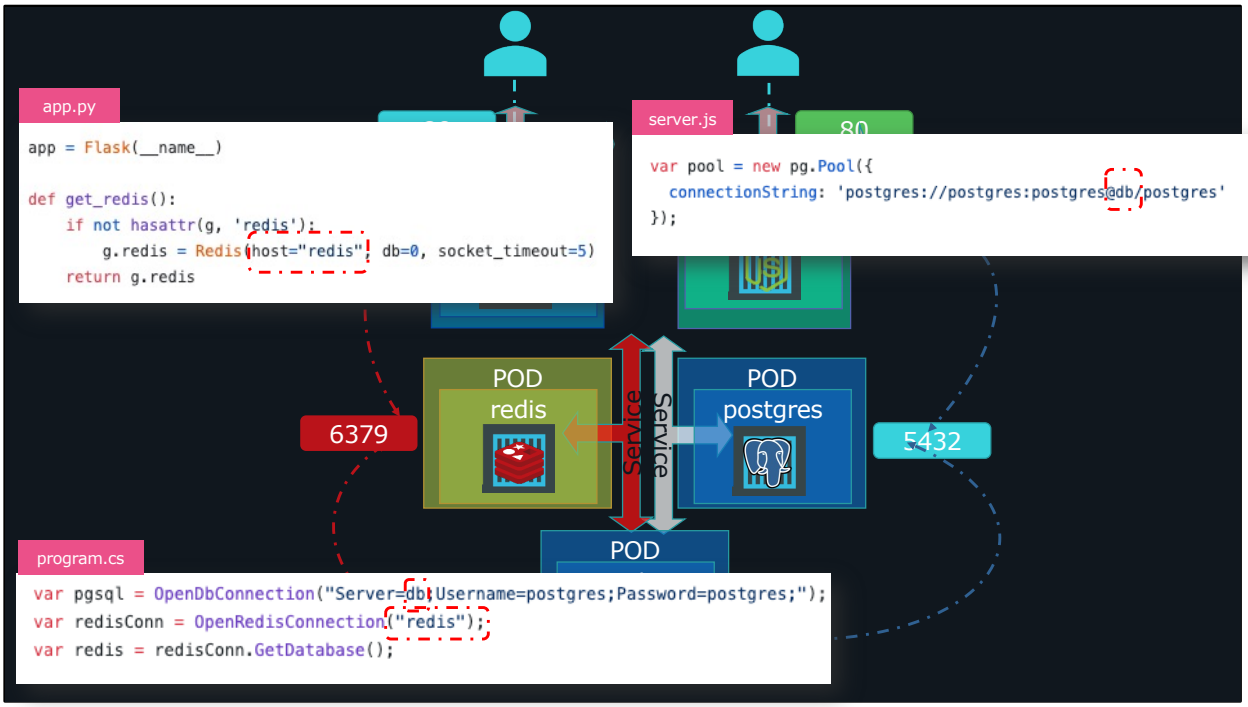
To summarize – in ANY case weather it be a single pod in a single node, multiple pods on a single node, multiple pods on multiple nodes, the service is created exactly the same without you having to do any additional steps during the service creation. When PODs are removed or added the service is automatically updated making it highly adaptive. Once created you won't typically have to make any additional configuration changes.



It's time for labs for services. Click on the link to go directly to the labs. If you haven't enrolled already enroll for free. In this lab you will work on creating services to expose applications within and outside a kubernetes cluster.

https://kodekloud.com/topic/labs-pods-with-yaml/


Demo Application
© Copyright KodeKloud



So we just saw how the voting application works on Docker. Let's now see how to deploy it on Kubernetes.

So it's very important to have a clear idea of what we are trying to achieve and plan accordingly before getting started. So we already know how the applications works, and it's a good idea to write down what we plan to do.

So our goal is to deploy these containers on a Kubernetes cluster, then enable connectivity between the containers so that the applications can access the databases and then enable external access for the external facing applications, which are voting and result app. So how do we go about this?

We know that we cannot deploy containers directly on Kubernetes, we learned that the smallest object we can create on a Kubernetes cluster is a POD. So we must deploy these applications as PODs on our Kubernetes cluster. Or we could deploy as replicasets or deployments. But first we will stick to pods, and later we will see how to easily convert that to a deployment. So once the PODs are deployed the next step is to enable connectivity between the services. It's important to first know what the connectivity requirements are. We must be very clear about what application

requires access to what services. We know that the redis database is accessed by the voting-app and the worker app. The voting app saves the vote to the redis database and the worker app reads the vote from the redis database.

We know that the PostgreSQL database is accessed by the worker app to update it with the total count of votes. And it's also accessed by the result-app to read the total count of votes to be displayed in the result web page in a browser.

We know that the voting app is accessed by the external users, the voters and the result app is also accessed by the external users to view the results.

So most of the components are being accessed by another component, except for the worker app. Note that the worker app is not being accessed by anyone. The worker app simply reads the count of votes from redis and updates the total count on the postgresql database. None of the other components nor the external users ever access the worker app. While the voting-app has a python web server that listens on port 80, and the result-app has a webserver that listens on port 80, and the redis database has a service that listens on port 6379, and the postgresql database has a service that listens on port 5432, the worker app has no service. Because it's just a worker and is not accessed by any other service or external users. So keep that in mind.

So how do you make one component accessible by another? Say for example how do you make the redis database accessible by the voting app? Should the voting-app use the IP address of the redis pod? No because that can change if the pod restarts. The right way to do it is to use a service. We learned that a service can be used to expose an application to other applications or users for external access. So we will create a service for the redis pod so that it can be accessed by the voting-app and the worker app. We will call it a redis service and it will be accessible anywhere within the cluster by the name of the service - redis. Why is that name important? The source code within the voting-app and the woker-app are hardcoded to point to a redis database running on a host by the name redis. So it's important to name your service as redis. So that these applications can connect to the redis database. And this is not a best practice to hard code stuff like this within the source code of application, instead you should be using environment variables or something. But for the sake of simplicity we will just follow this approach.

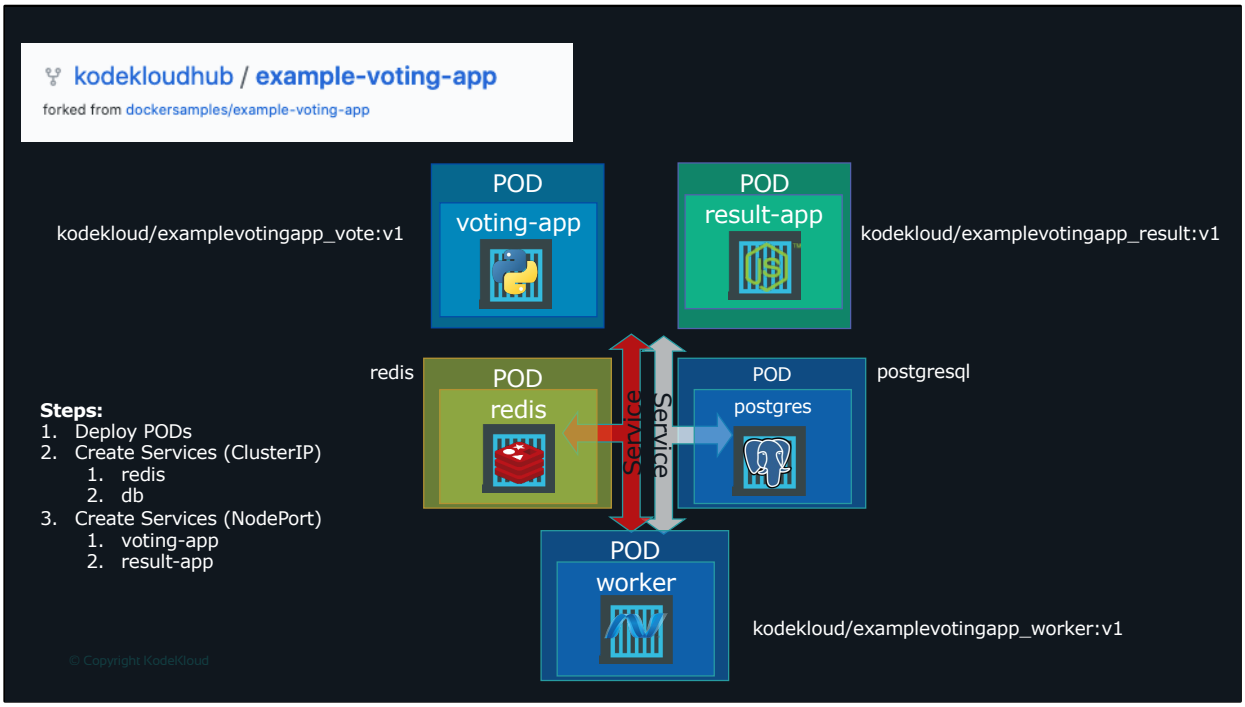
Now, these services are not to be accessed outside the cluster, so they should be of the type ClusterIP.

We will follow the same step of creating a service for the Postgresql pod so that the postgresql DB can be accessed by the worker and the result-app. So what should we

name the postgresql service? If you look at the source code of the result-app and the worker app you will see that they are looking for a database at address db. So the service we create for postgresql should be named db. Also note that while connecting to the database the worker and result apps pass in a user name and password to connect to the database. Both of which are postgres. So when we deploy the postgres db pod we must make sure that we set that for it.

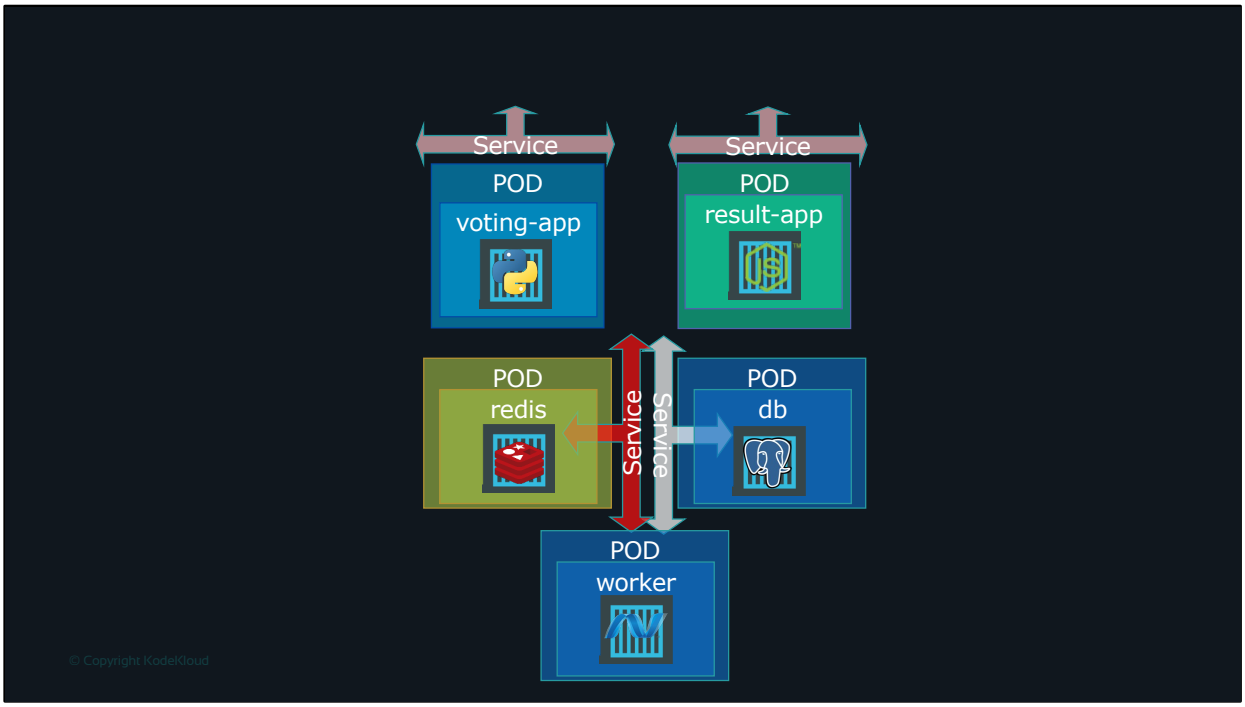
The next task is to enable external access. For this we saw that we could use the service type NodePort. So we create services for voting-app and result app and set their type to NodePort. We could decide on what port we are going to make them available. So we are done and we have the high level steps ready.

So to summarize we will be deploying 5 pods in total and we have 4 services, one for redis another for postgres both of which are internal services. So they are of type ClusterIP. We then have external facing services for voting app and result app. However we have no service for the worker pod. This is because it is not running any service that must be accessed by another application or external users. It is just a worker process that reads from one database and updates another. So it does not require a service. I say that again as that's a common question I get when we talk about services, why does this worker not require a service. A service is only required if the application has some service – like a web server or database server - that needs to be exposed.

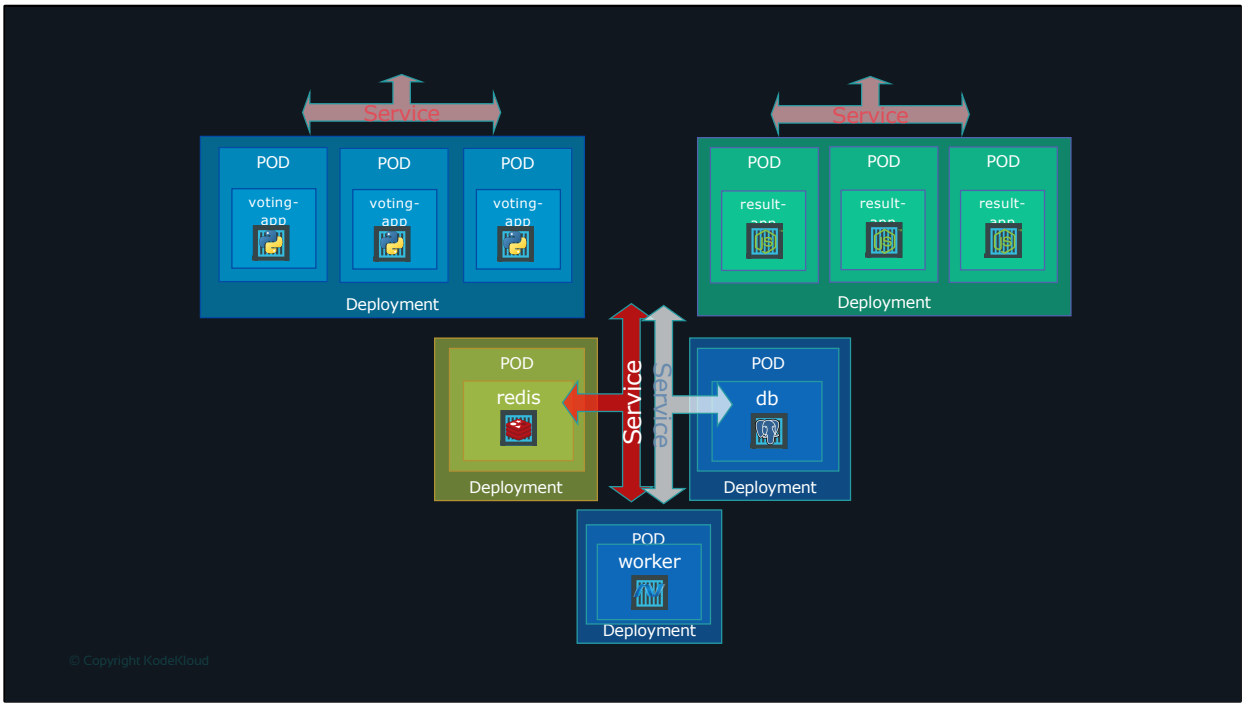


Before we get started with the deployment note that we we will be using the following docker images for these applications. These images are built from a fork of the original developed at the dockersamples repository. The image names are kodekloud/examplevotingapp_vote:v1 worker:v1, result:v1 and for the databases we will use the official redis and postgresql ones.

So that’s it for now and we will see this in action in the upcoming demo.

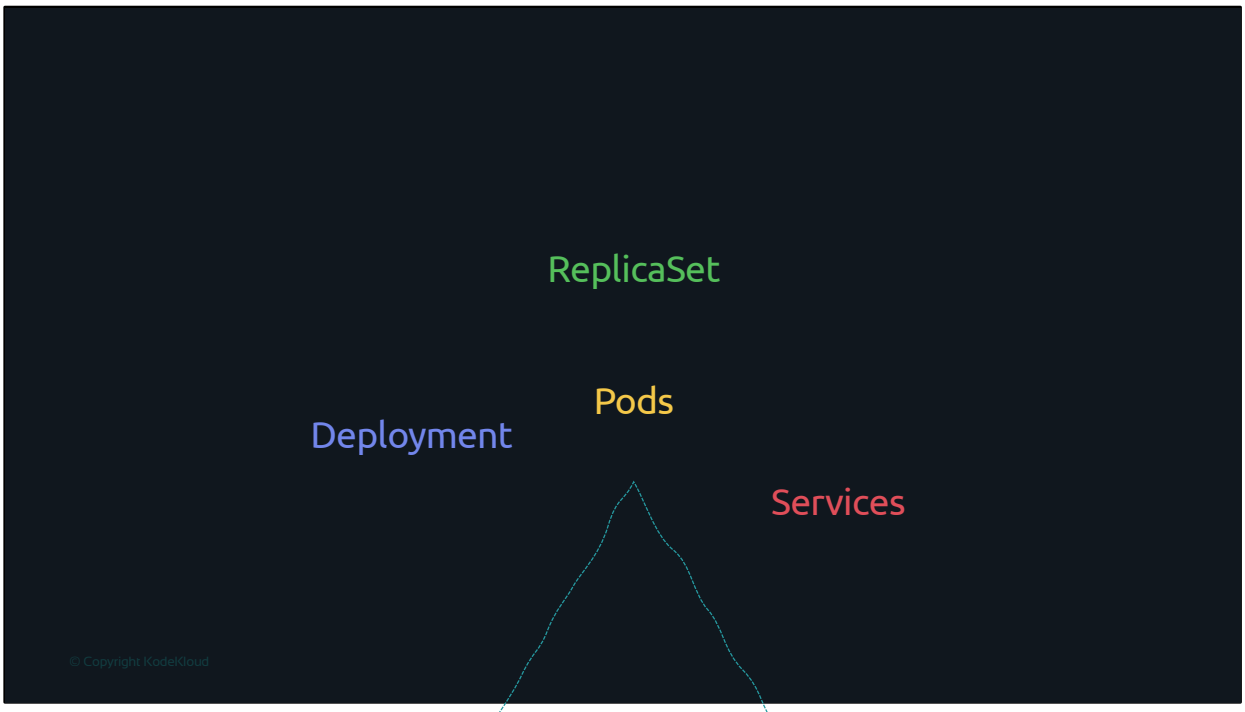


So this is what we saw in the last demo. We deployed PODs and services to keep it really simple. But this has its own challenges. Deploying PODs doesn't help us scale our application easily. If you wanted to update the image used in the application then your application will have to be taken down while a new pod is created. The right approach is to use Deployments to deploy an application.

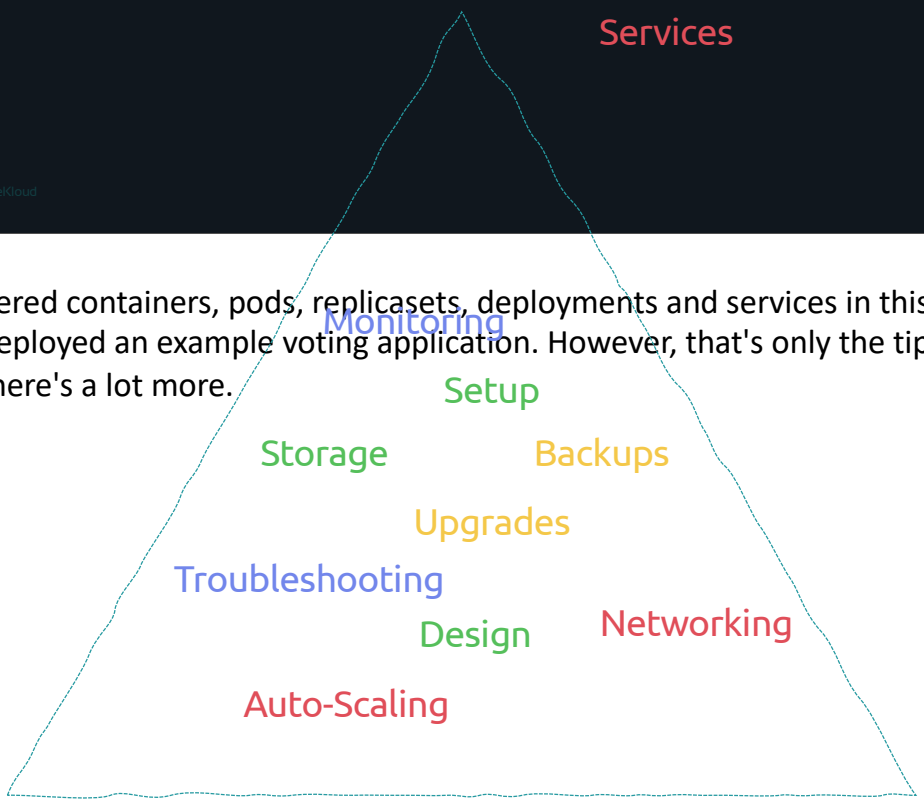


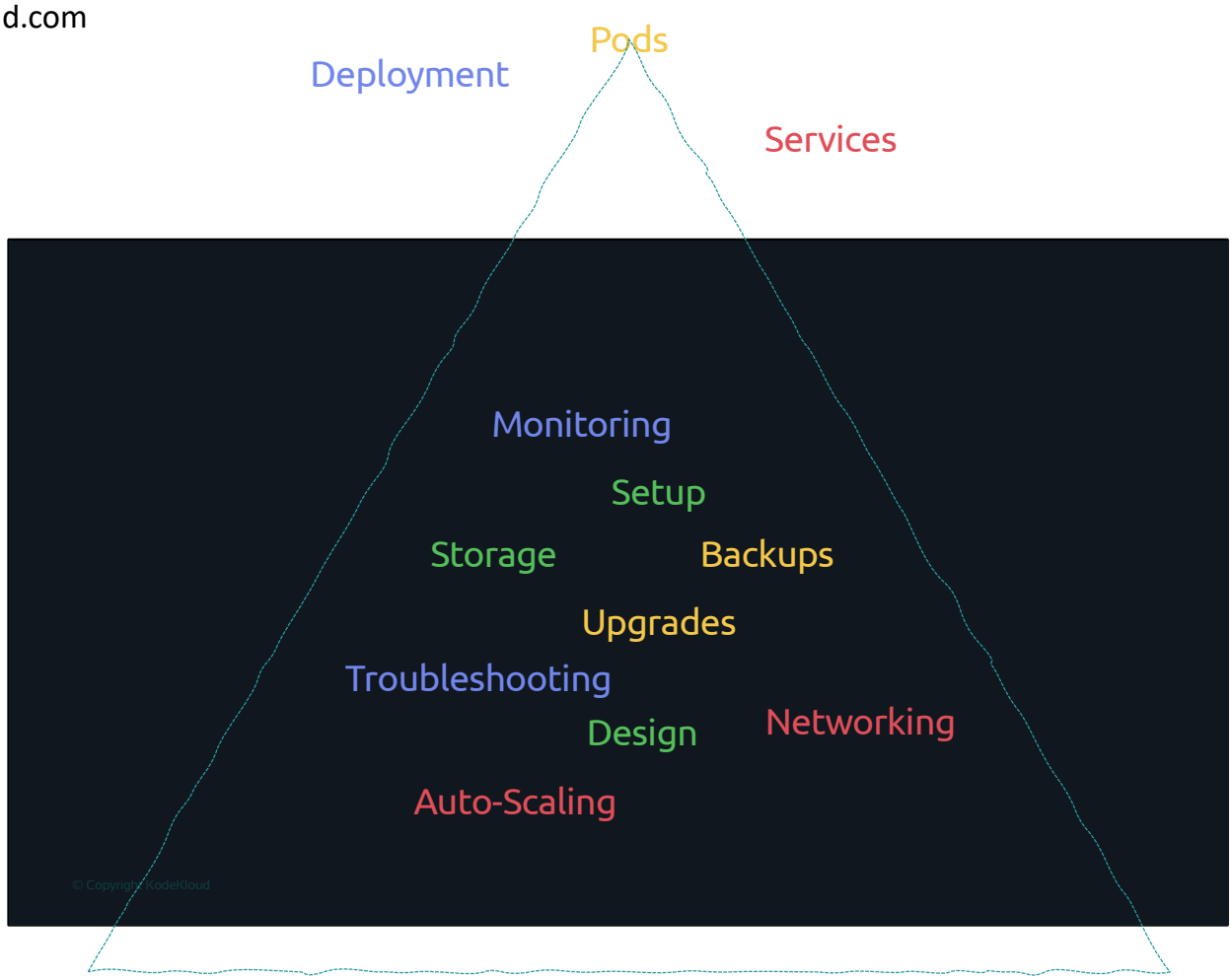
Let us now improvise our setup using Deployments. We chose deployments over ReplicaSets as Deployments automatically create replica sets as required and it can help us perform rolling updates and roll backs and maintain a record of revisions and the cause of change as we have seen in the previous demos. Deployments are the way to go. So we add more PODs for the front end applications voting-app and result-app by creating a deployment with 3 replicas. We also encapsulate databases and worker applications in deployments. Let's take a look at that now.

So here we are at the end of this crash course. I hope you enjoyed the course.

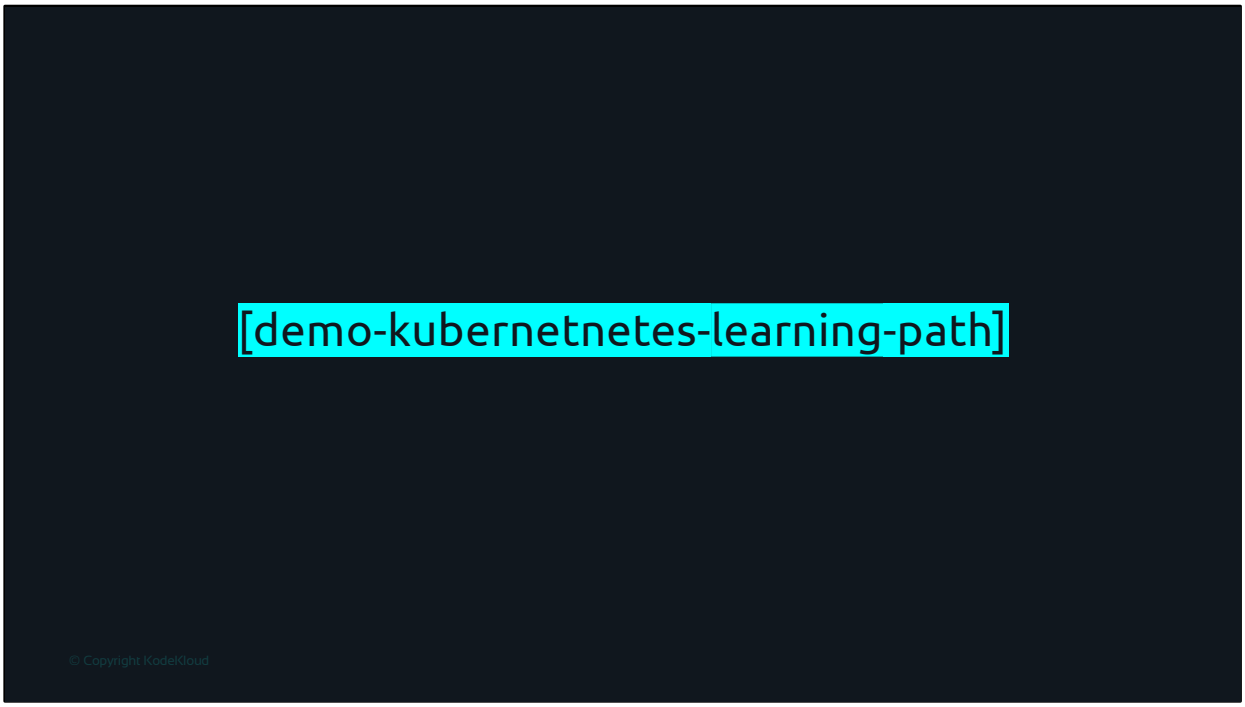


We've covered containers, pods, replicasets, deployments and services in this course and also deployed an example voting application. However, that's only the tip of the iceberg. There's a lot more.

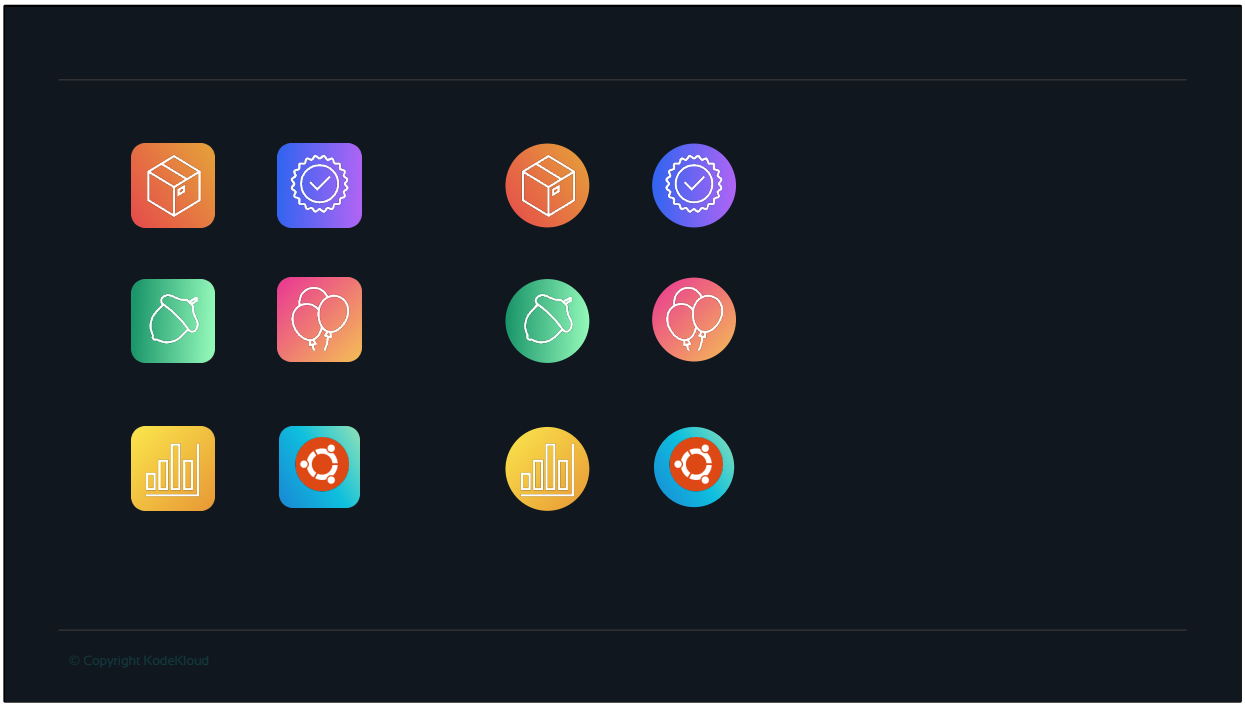




The different ways of provisioning a cluster, administering a cluster, maintaining and upgrading a cluster, logging and monitoring, security, backups, storage, networking, auto scaling, designing a kubernetes cluster and much more.



All of these are covered in our Kubernetes Learning Path. This involves – the CKAD – Certified Kubernetes Application Developer course, the CKA course – the certified kubernetes administrator, the CKS course – certified Kubernetes security specialist, and others like Helm, Kustomize, Istio etc.


© Copyright KodeKloud