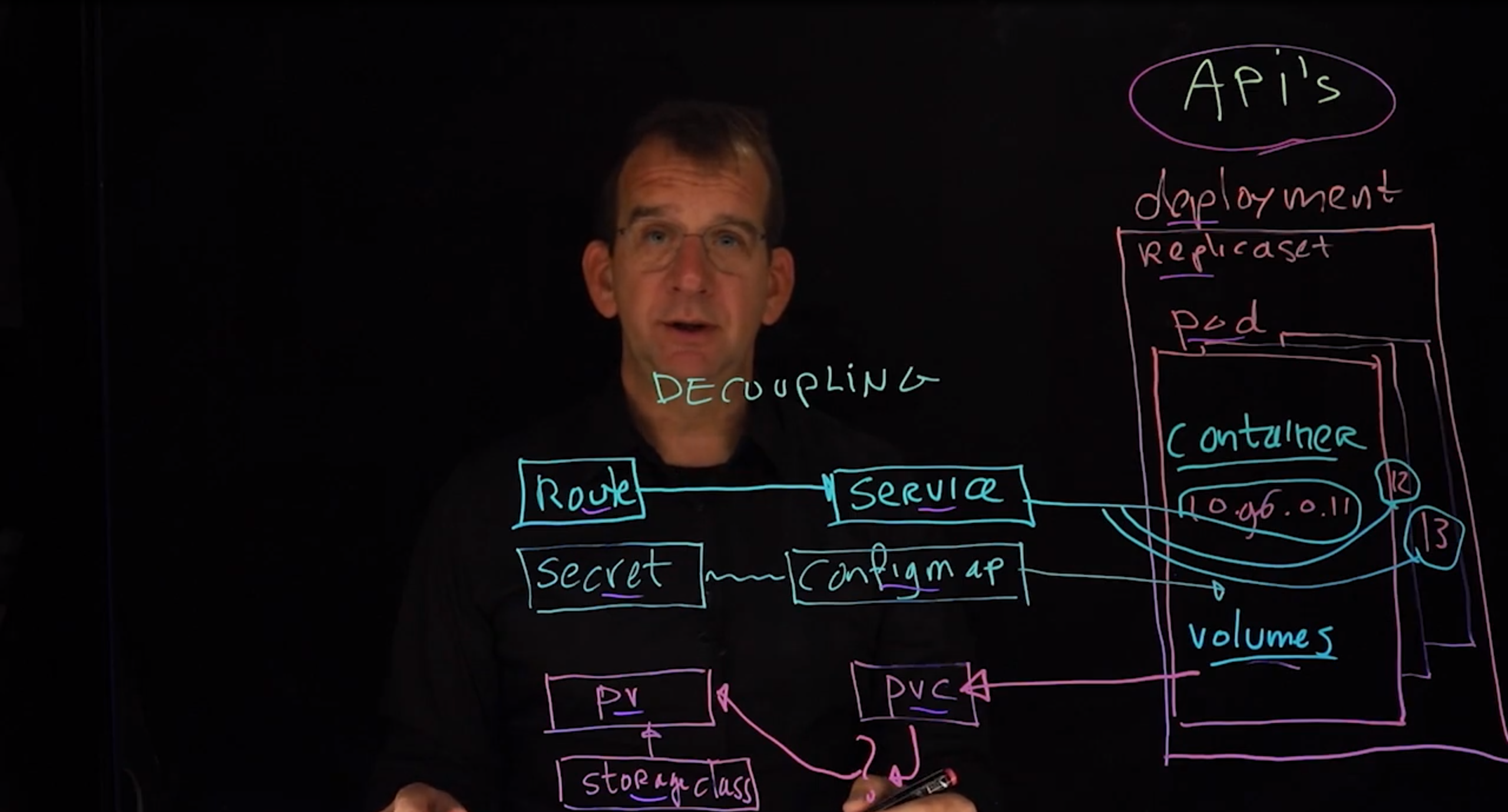
* 1. 7.3 Deploying an Application in OpenShift

1. Running Application in OpenShift
   1. Understanding OpenShift
   2. Exploring API
   3. Deploying an Application in OpenShift
      1. Understanding Projects
         * Applications should be deployed in isolated environments to enhance security and manageability.
         * These isolated working environments are known as projects.
         * Before creating applications, use **oc new-project** to create new projects.
         * OpenShift projects are stored as Kubernetes namespaces.
         * Us **oc projects** or **oc get ns** as administrators to get an overview of all projects.
         * Restricted users will see their own projects only.
         * - In this video, I will tell you what to do to deploy applications in OpenShift. So to start with, there is a project. What is the project? Well, the project is an isolated environment where applications should be deployed. Using projects is very important in OpenShift because the project is also the security boundary. And that means that when later you are going to work with role-based access control, you can use role-based access control to grant specific privileges to users, to a project. And you can do other limitations on project as well, such as quota to limit the availability of resources inside a project. Before you create your applications, you should always use oc new-project to create new projects. And I would really like to advise that you create one new project for every single application that you are going to add to OpenShift. OpenShift products are equivalent to Kubernetes namespaces. So if you have a Kubernetes background, you will find projects as namespaces in Kubernetes. There is one big difference though, and that is that projects in OpenShift due respect role-based access control namespaces don't. If you are an ordinary developer kind of user in OpenShift, you do have permission to see your own projects. You cannot see all the namespaces in the environment, but if you login as Cube Admin in the OpenShift environment, you will see all these namespaces as well. And that's kind of nice about projects. Projects is something that can be applied in the limited environment of what your user is doing. And I will show you how to use oc projects and oc get ns to get an overview of all these products and namespaces only.
         * So, as we have discussed before the Linux kernel is implementing different features that are very important for Kubernetes, as well as containers. And one of these features is the namespace. The namespace in the Linux kernel is providing a very strict level of isolation, and that means that containers that are in one namespace cannot access containers that are in another namespace. Now Linux kernel features are implemented in Kubernetes as well. So if you look at Kubernetes in Kubernetes, we have namespaces also, and the namespaces that you are using in Kubernetes can be used as different environments to keep different workloads together, but in the end, all of these namespaces are implemented by the Linux kernel namespace. Now in OpenShift, we don't really use namespaces directly, but you will be using projects. Now, while there's a project, a project is a namespace, but in OpenShift, the project is combined with role-based access control. And that means that ordinary users will only see their own projects and administrator users will see all projects. In fact, for an administrator, usually if the administrator usually is looking here at a project, the administrator user will see the namespace. But the nice thing about OpenShift projects is that these projects for ordinary users allow them to create their own environment. You could compare that to a chrt directly on Linux, where a user can only see the contents of their home directories and within their home directories, they are creating these different projects. Now what are projects used for? Well, two main reasons. One reason is for organization. Which is all about keeping resources together that belong together. Another reason is for restrictions, restrictions in the broadest sense of the word. You can use quota for instance, to limit your availability of resources within a specific project. So that customers that pay more, get more and customers that pay less get less. You can also use it to apply administrative roles to specific projects, and that makes projects in OpenShift to really a lot more than namespaces in Kubernetes because namespaces in Kubernetes, well, they're not normally used that thoroughly in combination with role-based access control.
         * Now let's talk about running applications in OpenShift. oc new-app is the primary tool for running applications. It's a tool that has been engineered to work with applications in an OpenShift environment. oc create deployment can be created as an alternative, but the main differences that oc create deployment was created for Kubernetes and as such oc create deployment, doesn't know about any of the OpenShift specifics. With both of these commands, use minus H to get usage help. We already talked about using minus H, which is very convenient. And after creating the applications, as we have already seen, you can use oc get all to find application resources. And if you want to get an overview of resources in all projects, you use oc get all minus upper case A. Now, before I'm going to show you how this works from the command line. Let's talk about oc new-app sources. oc new-app can built applications from different sources. And these include a Dockerfile, which you can use to ultimately to build your own images or an image from any image repository, or using directly from the source code or indirectly from the source code using Source to Image. Now, when you are going to work with them applications in OpenShift, you will find that there's a deployment. There's also a DeploymentConfig. Let me outline the similarities and differences. In older versions of OpenShift, the OpenShift API resource DeploymentConfig, was the default resource to run applications. The DeploymentConfig uses a ReplicationController to control port replications. In newer versions of OpenShift, the Kubernetes API resource deployment should be used instead. Now, how does that work? Well in OpenShift, the DeploymentConfig was introduced before the Kubernetes deployment. And that is why in older versions DeploymentConfig is the standard, but nowadays as OpenShift is a Kubernetes distribution, OpenShift tries to respect the standards that are imposed by Kubernetes, and for that reason, the Deployment is the preferred way. If you are using the Deployment, the Deployment uses ReplicaSet to control Pod replication instead of the DeploymentConfig ReplicationController. If you specifically want to create DeploymentConfigs using oc new-app, don't forget that there's the oc new-app with the dash dash as-deployment-config option that will run applications specifically as a DeploymentConfig. This is kind of tip that may come in handy during the exam. Also, you should realize that in the current version of OpenShift, some resources like templates, for instance, still are creating DeploymentConfigs instead of Deployments as the default. So be prepared for both of them, and don't be surprised if oc get deploy doesn't show your applications use oc get deploymentconfig instead in that case. Let's go check out these commands a little bit. So let me start with oc projects. oc projects is showing all the projects currently available. And as you can see in my user account, I have two projects. Next question, of course is what is my user account? Well, oc whomami will help you in figuring out there you can see I am developer. And if I would use oc get ns, then I'm trying to get information about namespaces, but that is not allowed for a developer user. So I have to login as a Cube Admin user. Now, if you need your Cube Admin credentials, the best way to do so is crc console dash dash credentials. The credentials are printed directly after installation, but using this dash dash credentials can easily get access to the Cube Admin credentials. Then you just copy and paste, and then you are Cube Admin. As you can see right now, I have access to 63 different projects. And that is because as Cube Admin I see all the administrative projects as well. These are projects that have a name that starts with OpenShift, and likewise being Cube Admin, I can use oc get ns, where we see the different namespaces. And just so you know, the namespaces are equivalent to the projects. So a project is just a filtered way of presenting information that is provided by Kubernetes namespaces. Here, you can see the projects that we have created earlier in this course. Right, let me get back to a developer view. Now, let's talk a bit about oc new-app, but before we do that, first oc new-project. It's really important to create new projects for all the applications that you are going to work with. And let's call it myfirstnewapp or whatever. The real reason, the most important reason why this is so important is for your own convenience. If you don't throw everything in the same project, it is so much easier to manage what's happening. Now you can see that the command line client is already giving me information on what we can do. So here it's telling me I can use oc new-app, or I can use oc create deployment. We have already seen oc create deployment, so I want to explore oc new-app a little bit now. So oc new-app minus H by bless showing me what well, showing me some explanation, and then it is telling me how to use it. So oc new-app works with an image, an image stream, a template, a path, a URL, and absolutely more of them. Template is something that we are going to discuss later. So let's not talk about that. Now let's talk about pretty straightforward commands like oc new-app, mySQL, mySQL user equals et cetera. Now, what is all of this? This is using oc new-app, based on the mySQL image. And this mySQL image is fetched from the image registries. If you want to be specific in the image registries, you can specify the specific URL to the image. And if you don't, then you will probably end up with the Docker registry. If you are looking for a specific command line arguments, that's also a convenient, you just go down and you'll get a list of all the command line arguments, like the name for instance. So that means that if I'm going to use oc new-app bitnami nginx dash dash name is bitginx then it is creating the new application for me. So this is the output that we get by default. It has found a container image from Docker Hub. So it's fetching this from Docker Hub by default, and it uses an image stream. We will talk about image stream separately later, but image streams are pretty important in OpenShift, especially if you are going to build new applications from your source code, using Source to Image. And then it's creating the resources which include the image stream, but also the Deployment and the service. And in the end, it's telling us that the application is not exposed. Now, there are two ways to look at the result. You can just type oc status, and oc status is giving kind of a hierarchical overview. So we have the service, the service bitginx which is exposing the application on port 80 80, and 8 4 4 3. Now we have the deployment, and the deployment number two, running for 39 seconds with one pod, and there is the deployment number one deployed 41 seconds ago. All right, now I also like investigating this by using oc get all and that is because oc get all is presenting the information more exactly as it is. So we have the Deployment. The Deployment was created 73 seconds ago, and here we can see that the Deployment has created a replica set. There's one 73 seconds old, and there's one 71 seconds old. This first replica set is the replica set that relates to the image stream. We'll talk about that later once more. And the second replica set is the replica set that learns my pod. And here we have the service that is exposing access to the application. And that's basically all you need to do to run applications in an OpenShift environment. Before we continue, let me do one more thing, and that is oc delete. oc delete what? Well, oc delete pod bitginx because this is the reason why you want to run applications in an OpenShift environment using Deployments. If you delete your pods, then oc get all is giving me what? It's giving me a new pod; it was generated nine seconds ago. The essence of OpenShift is that you are going to decouple your application, which means that your application code is one thing and the application data is running somewhere completely else. And as a result, if something bad happens to one of your learning instances of the applications, then that's not a big problem. OpenShift is just going to run a new instance, which will connect to your data. You will find out more about that as we proceed through the remainder of the lessons in this course.

7.2 Exploring the API

- In this video, we are going to talk about the API. To tell you and give you insight how the API is working, I would like to get started with a drawing. In this lesson, I'm going to tell you about OpenShift resources. These resources are all defined in the API's. And most of the resources that we are dealing with are Kubernetes resources. Now, what is it all about? Well, OpenShift is all about running a container. But a container is not a resource in OpenShift. In order to manage the container, OpenShift is putting the container in a box, and this box is what we know is the pod. The pod is the most essential resource in an OpenShift cluster. It contains properties that are necessary for managing your container in a clustered environment. Like for instance, container volumes that take care of storage. Now, the pod is a single instance of your application, and its scalability is a thing in an OpenShift environment, you don't wanna run single instances. An OpenShift is putting the pods in another resource, and this other resource is the deployment. In older versions of OpenShift, you will see it referred to as a deployment config, that is basically the same. Now the deployment has taken care of replication by using replicaset. And because you are using a replicaset, it becomes possible to run different instances of the same application. That's what the scalability is all about. But by introducing the scalability, we also have a new challenge, and that is that all these pods have their own IP address. So let's say this is 10.96.0.11, this one is 12, and right here, we have 13. Now, which one is the end-user going to address? Well, that is where another resource comes in. And that resource would be the service. So the service resource is all about accessibility. And that is because the service resource really is a load balancer, an API-based load balancer that makes sure the traffic can be load balanced between all these different IP addresses. So if user initiates a request, the request is directed to one of these containers, and the service has taken care that that is going to happen alright. Now service provide a single point of entry, they have a single IP address as well, but there's one thing that they don't do, they don't expose any routes, any URL, and that is why OpenShift is adding the route resource. So the route resource is what provides the URL, the user uses the route to address the service, and the service is pointing to one of the pods that is running in your Kubernetes environment. And if you think that is all, it's not, because there is another thing that is very important in OpenShift, and that is what we call decoupling. Decoupling is about separating site-specific data from your code. And the most important resource to implement decoupling is a ConfigMap. The purpose of the ConfigMap is to provide site-specific data like environment variables, like local storage, and this ConfigMap is connected to your pod, and that allows the pods to contain static data only that is not related to any specific site. And when you are running an application in a specific site, the ConfigMap is added to that. Apart from that, there's also the Secret. Now, what is the relation between a Secret and ConfigMap? Well a Secret is a base 64 encoded version of the ConfigMap. There is one more resource type that we need to talk about, and that is the resource type that talks about storage. So we discussed that in pods, you have these volumes. Volumes can directly connect to external storage. And if you do that, you make that your pod contains site-specific information, you don't wanna do that. So in OpenShift, you want to make sure that your site-specific information is contained in the PV, the PV is a persistent volume. And the persistent volume points to your sand or whatever type of storage that you have. This PV can be automatically generated by the storage class. So that means that as an administrator, you don't have to generate PVs manually, PVs can be generated on demand. The big question is how are you going to connect this PV to your pod storage? Well, you do that by using a PVC, the persistent volume claim. And the persistent volume claim is just a request for a specific amount of storage. And that means that the pod does not contain information about any site-specific volumes, no, the pod refers to the persistent volume claim. And next, persistent volume claim is going to query for storage on the cluster. And if some available storage exists, it will bind, if the storage does not exist, it will be generated by storage class. So let's summarize, what did we see? We have seen that the essential resource is the pod. The pod is managed by the deployment, the deployment is using the replicaset to take care of the scalability. To take care of accessing, we have to service, the services addressed by the route. The ConfigMap is used for storing variables. The Secret is for storing variables and configuration files in a secure way. The persistent volume represents storage at a specific site, and it can be dynamically created by the storage class. And a persistent volume claim is what the volume is using to connect to a specific storage type. And all of that is defined in the Kubernetes and OpenShift API's. In the next couple of videos, we'll discover all of these in much more detail. All right, let's summarize what we have just seen in the drawing about the API resources. There is the oc api-resources command, which shows the list of resources and the specific API collection that they are coming from. And there is oc explain which can be used to get more information about API resources. In oc explain, you find the exact definition of the API resource, and that comes in very handy if you want to work with YAML manifest files to define your applications. And notice that in some cases, Kubernetes resources as well as OpenShift resources can be used for the same purpose. Let me show you what the result of these commands looks like. All right, let me start with oc api resources. I'm piping that through less because it's giving a lot of results. So what do we see? We see a couple of columns. In the left column name, we find the name of the API resource. Then some of the API resources have a short name, some don't, but if they do have a short name, it's convenient to work with them. Then a very important part is the apiVersion. The apiVersion is telling us which apiVersion the resource is coming from. Now, it happens to be that in OpenShift, there is multiple APIs that are connected together. If you see apiVersion V1 like right here, that is a core Kubernetes API. apps/v1 is the most important extension to the Kubernetes API. But as you will see in just a little bit, there are other apiVersions SL, and in particular OpenShift is adding lots of apiVersions. Now we see whether or not the resource is namespaced. Namespace, if that is set to true, that means that the resource can be limited to the boundaries of a namespace, and that can be useful for role-based access control. The kind is the name of the resource in a way that it is used in a YAML code, that's not the most important one. What is important here, is to see if you scroll back a little bit. Look at this, here you can see a lot of stuff already happening. Now, the thing is that the Kubernetes API is extensible, and that means that in Kubernetes, you don't work with one single API, you work with a set of APIs. And it is relatively easy to add resources to this collection of APIs, and that's exactly what OpenShift is doing. And that is also about the differences between OpenShift and Kubernetes. OpenShift is adding lots of APIs to what Kubernetes is adding already. And that is cool, but it also means that it will be more complicated if ever you want to move away from OpenShift back to Kubernetes, because all of these API specifics that come from OpenShift APIs are not available in Kubernetes. Now you will notice that in some cases, the OpenShift API is providing functionality that more or less is provided by the Kubernetes API SL. Now, how is that possible? That's because of the open source dynamics. OpenShift is its own product, but OpenShift also works with Kubernetes. And that means that sometimes new features come from OpenShift. And if the new features are introduced in OpenShift making their way to Kubernetes, that may lead to situations that look like redundant features. You will see that in a couple of cases, but normally in the end, it all fixes itself and one of the features is going to win. Good. Let's get out of here, and let me show you oc explain. You can do oc explain on any API resource. Let me use oc explain on pod, what do we see? We see the fields. The fields you see here are always more or less the same. You will always find apiVersion kind metadata and spec. And status is what is filled in by the cluster itself, it's the current status of the resources. Now the most important parts of what you find in oc explain is in the spec part, that goes for all of the resources. So if ever you are wondering what is so specific about pods, what exactly are pods adding to my cluster resources? Well, you have look at oc explain pod.spec. And that is useful for explanation purposes, but also as these are parameters that you are going to put directly in the YAML codes that can be used to define your resources, and that we can see these resources that are so specific for the pods. So containers is where you include the containers, but you can do things like initContainers. And initContainers doing some additional work before the main container is started. Or in a cluster, very important, the nodeSelector, which is going to help your cluster decide where the node should be started, or priority, which is going to set a priority for your pod. And like that, there are many more properties that can be set for pods using the API specification. And you will find that oc explain is a very valuable help when defining your own resources in the OpenShift cluster.



Kubernetes and OpenShift define the following:

* Pods are managed by deployments.
* Deployments use replica sets to take care of scalability.
* Service exposes an application running on a set of pods as a network service. It provides a network endpoint of IP addresses and Ports that allows other services or external clients to communicate with the pods running the application regardless of which node the pods are running on.
* ConfigMap decoupling configuration from application code can be mounted as file or environment variables within a container, making it easier to manage and update configurations without the need to rebuild or redeploy applications.
* The Secret is for securely storing variables and configuration files.
* Persistent Volume represents storage at a specific site, which can be dynamically created by the storage class.
* A Persistent Volume claim is what the volume uses to connect to a specific storage type.

So, as we have discussed before, the Linux kernel is implementing different features that are very important for Kubernetes, as well as containers. One of these features is the namespace. The namespace in the Linux kernel provides

### 7.4 Displaying Information about Running Applications

- In this video, I'm going to show you how you can display information about running applications, which in fact is a key skill if you want to do troubleshooting as well. It's just about a demo, and you can see the most essential commands. I have elicit these commands for you on the slide, but let me just demonstrate them. So the first thing that you need to do to get information about your running applications is oc get all. That is because an application is composed of different elements, as a deployment, as a replica set as a pod, as a service. There might be more involved, and oc get all really is the only way to see all of it. As you can see, we have the deployment. Now if you look at it, the deployment is telling us that ready, one out of one, and up to date, one, and available, one. So this is a deployment that is expecting to see one pod, and we can see that this pod is up and running. Now we see the replica set. This is the active replica set where we have desired, current, and ready all set to one. That's what we are expecting to see. Now we also have a replica set which is desired, current, and ready all set to zero. Now as this is an application that has been started with oc new app, this replica set has had a temporary function in the build process, but that function is now over, and for that reason, we don't see anything anymore. You will notice that after replication updates, old replica set stay around to make it easy to revert an update in case it has gone wrong. So don't worry about it. Desired, current, and ready all set to zero means that this is not supposed to be any active. Then we have the pod. And in the pod, we see one out of one. Now, what is this one out of one? That's about containers in this pod, because pods are used to star containers, and you can start multiple containers by one pod. So one out of one means that this pod is starting one container, and this one container is present. And on top of that, we see the service. I'll explain later what this service is all about. And likewise, I will later about the image stream. Now in your applications, it's always about the pod because the pods, after all, is your running application. So let me show you more about the pod, but let me do that in a way where we see a failing pod. I'm going to use oc run -h. That's a command that really you should know about, but you should not use too much because oc run is starting an unmanaged pod, also known as a naked pod. And I'm going to use oc run to start mariadb. So I'm using oc run mymariadb --imagesmariadb. Oops, I'm almost forgetting something. Do you remember what you should do before running applications? Oc new project maria, and then we do oc run mymariadb imagesmariadb. We don't want to see this new pod being mixed everything else. So let's do an oc get all, which in this case is just showing pod information. And in the pod information, we see container creating. Container creating can take up to a minute or so, depending on your download speed, because images need to be fetched. So don't be surprised if container creating is taking a little bit of time. If you want to be more specific, oc get pod is what you use to get information about pods only. And we can see, well, no difference with previous commands because in this case we only have one resource. A convenient addition to oc get pod is -A where -A is listing all pods. But oops, you can see error from server forbidden. - A is getting pod information from all namespaces. And if you want to get information out of all namespaces, you need administrator privileges. And we don't have administrator privileges here, and that's why we are getting forbidden statement here. Okay, let's get back to the oc get pods where we see, oh no, mymariadb is zero out of one, and the status is error. That's on purpose because I wanted to show you what you can do in case things are going wrong. In your basic troubleshooting, if you are trying to start a workload, and that doesn't work out the way you expect it to, then the first thing to do is oc describe. So oc describe pod on mymariadb. Now what is oc describe? Well, when you create an application, the application is added to the OpenShift database. That's the ETCD database, and oc describe is telling you what exactly has happened in the ETCD database to create your application. And if anything cluster-wise went wrong, an oc describe will tell you, and otherwise oc describe will give you a very good overview of all the events that have happened. We want to start at the bottom. Because at the bottom, we can see a list of the events, and the last event is the most relevant because we can see 12 seconds ago, six times over 79 seconds, we have a back-off restarting failed container. So OpenShift has tried to start a container that didn't work out successfully. Now if starting a container in a pod doesn't work out successfully, you need to investigate. Now in oc describe you can find all the relevant information, starts right here. And this information includes information about node that is currently running it, about labels. You find the IP address. But most important of all of it, you find the current container state. Now what do we read in this container state? We read that the state is waiting. The reason is CrashLoopBackOff. CrashLoopBackOff means that OpenShift is trying to activate the work load, and that didn't work out all right. And the last state is terminated because of the reason error. Something is wrong, exit code one. Now what is this exit code one? Well, this exit code one is what your container application is generating. You know every container has a main application, the entry point application, which we have seen before. If this entry point application is starting and it doesn't work out all right, it generates an exit code one, and that means that the problem is not in OpenShift. The problem is in your application. And if the problem is in your application, the best thing to do is to use oc logs because oc logs connects to the standard out of your application. And if you use oc logs mymariadb in this case, then we see exactly what has been generated by mymariadb. It's telling us error database is uninitialized. Password option is not specified. You need to specify one of MYSQL root password and a couple of other variables. So I have not provided the required information, and that is why I'm getting this error. In order to fix this error, well, in this case, is you can pass variables to a running pod that is not managed by a deployment. We will need to delete the deployment and start it against later, but I will explain later how you can pass variables to containers while they are starting. There's one more thing that I want to show you in the context of displaying application information, and that is oc get pods on your pod -o yaml. Now when you run applications in OpenShift, the applications are writing their information to the ETCD database. This information is written in JSON format, but JSON is not something that reads easily. So if you want to read exactly what is happening, you can use -o json to print a JSON text. I'd rather go from -o yaml to see the YAML information about the running application. Now what do we see? We see API version metadata. We don't care about that too much. What we care about is right here, the spec in which you can see all the properties that have been used while starting this application. And as you can see, this is much more than what we have provided on the command line. All of these are default properties that have been set, and these properties are quite important for managing your application workloads in an OpenShift environment. What we also see towards the end is a status. And if something is not going right with your application, the status is the raw status information about your application. And this is information that's also included when we are using oc describe. Look, we have a lastState terminated. Reason is error and exit code is one. It's written in a slightly different way, but this is exactly the information that we've just seen in oc describe. Oc describe is a bit more readable, but the information that you get in the oc get pod -o yaml is just as good.

### 7.5 Using Labels

- In this video, we will talk about how OpenShift uses labels. So, what is a label? Okay, well, a label is an identifier that is added, either automatically or manually, to workloads in OpenShift. Labels are used by OpenShift as a selector, which means that you can use it as a filtering option. It also means that labels are used internally to make sure that resources match together. Like ReplicaSets, which are using labels to monitor the availability of Pods, or like Services, which are using labels to connect to Pods with a matching label. Administrators can set labels to make filtering or scheduling Pods easier. Let me show you a little bit about it. To make it easy, out of the next demo, I have summarized the most important commands, which you can see right here. All right, for this demo, we need to be in an environment where we have running deployment. So, let me use "oc projects", which is showing me the different projects that are available for my environment. And let me use "oc project lab", to get back to the lab project. "Oc get all" is showing the different resources running here, which include the deployment or ReplicaSet in a Pod. Now, if I'm using "oc get pods,rs", it's just to show you that you can use "oc get" to show multiple resources, as well. So, I just want to see Pods and ReplicaSets, "--show-labels" is showing all the different labels that have been set. Now, what do we see here? We see that the label "app=labnginx", or "app=labginx" is set to the Pod as well as the ReplicaSet. That is how ReplicaSet and Pods are being matched together. Now, if I'm using "oc get pods,rs --selector app=labginx", then we don't see any difference here, but this is what you can use to filter Pods and other applications based on a specific label. So here we only want to see resources that match the label "app=labginx". You can use "oc label" yourself, as well, so if I use "oc label pod" on "labginx", "storage=SSD", for instance, then we have added the label. And if I'm using the "--show-labels" again, we should see it. As you can see, it's added to the end of the labels for the Pod.

Now, let me do something else that's cool, and that is "oc label pod labginx app-". And now I'm using "oc get pods", and guess what? Do you see what's happening? We see that there is a new Pod being created. Now, why is that? Well, that is because of this. "Oc get pods --show-labels" is showing that we have two Pods running. One is "app=labginx", the other one does not have "app=labginx", and the replica is monitoring the availability of workloads by monitoring the labels. So, if, suddenly, you reset the label, or if, suddenly, the label that the ReplicaSet is monitoring goes away, then, according to the ReplicaSet, the application is no longer available, and the application needs to be restarted. And that's exactly what's happening here. So this is how you are using labels. Labels are important. Keep an eye on them because they can be used to set some additional metadata on your application, and sometimes it's good to figure out what exactly your labels are doing, to understand what's happening in your OpenShift environment.

### 7.6 Using OpenShift in the Declarative Way

- In this video, I will show you how to use OpenShift in a declarative way. Now, what is declarative way? Well, it's the opposite of the imperative way. In the imperative way, an operator types commands to get things done. There's nothing wrong with it, but it's not a very DevOps way of working. In a declarative way of working the configuration is managed as code. That brings some benefits because that means that you can store your configuration in a Git repository and ensure consistency as well as apply version management. And in DevOps environments, it's preferred to work in a declarative way just because it makes sure that everybody is doing the same and using the same methodology. While you are using a declarative approach, you can also easily manage versions of the configuration code in a virtual control system, and that makes it easy to manage it at an enterprise level. OpenShift resources can be defined in a declarative way using YAML files. So how does that work? Well, the recommended way to create YAML files is by just generating them. You can do very complicated and start from scratch, but that's not a good idea just generate them. Most commands that you will be using in an OpenShift environment, have some options. The --dry-run=client followed by -o yaml. And that generates YAML and just use Linux command line redirection to capture the YAML code inside a file. Alternatively YAML files can be written from scratch according to the directions that can be find in oc explain. We have already seen oc explain, and I will show you again. And also, but that's a bit more work, YAML files can be generated from running resources using oc get deploy mynginx -o =yaml. The disadvantage of this approach is that if you generate YAML code out of a running resource, you get a lot of status information and other information that you don't need at all. Let me show you how to work with YAML in an OpenShift environment. As I just mentioned the easiest way to work with YAML is like this. You use oc create deploy, or whatever you wanna do, mynginx --imagesbitnemi/nginx or whichever image you want to use. And then --dry-run=client. That's the important part because this guarantees that the application is not added to your closer, it is just doing a dry run. And in the dry run, it shows what it should be doing, but it's not doing it. This dry run only makes sense if you generate the output as YAML file. Now without anything else, the YAML code will be dropped on the screen. That's not very convenient so I'm using command line redirection. All right, so we use redirection to redirect to my nginx.yaml or whatever you want to call it. Now let's open this file so that we can talk about the structure in a YAML file a little bit as well. So my nginx.yaml, what do we see? We see API version kind metadata spec, and status as well. Hey, keep that in mind. API version kind metadata spec. If I use oc explain on deployment or deploy short also works. Look, there we have API version kind metadata spec and status as well. So everything you see in the YAML file corresponds to the output of the oc explain command. And if you want to investigate what is happening, use oc explain for more detail. API version refers to the API in its current version. It is important that if you are updating to a newer version of OpenShift, newer versions of the API may become available and what you have right here must correspond to the current version of the API. So the kind to set to a deployment, that's the API resource then we have the metadata. And in a metadata, we can see child resources. So we have creation timestamps, label, and name. Now the hierarchy between the parents and the child resource is indicated by using indentation in YAML. And indentation in JYAML is happening by using spaces, not tabs. It's common to use two spaces, it's not mandatory. You can use as many as you want, as long as you are consistent in it and all child items are indented the right way. So we can see the metadata as creation, timestamp, and labels and name and we can see that the labels as a child resource as well, which is set to app nginx. And likewise in spec, we can see all the resources that are a part of the deployment specification. So replica and selector and strategy, these are a part of the deployment specification then we get the template. The template is the pole that is managed by the deployment. So everything you find on the template really supports specification, where you will find metadata and spec as well. And in the spec, you can find the containers. Notice that this container has a hyphen in the beginning. The hyphen in the beginning indicates that this is a list item. You have simple key value pairs in YAML. You can also define a race or lists, which means that there can be multiple of them. And it happens to be that inside a deployment, you can have multiple containers. And that is why we see the containers as a hyphenated item. Every new thing starts with a hyphen, indicates a new container. Now, once you have a YAML this way, you can just use oc create -f mynginx.yaml to create the resource. And from there, everything is the same. Obviously the big benefit of using this YAML based approach is that you can put it in a Git repository and do some version management. Now, let me also show you oc get all, which is just showing me the mynginx, oops, which was not created in a new product, but hey, for this demo, it's not that important. Let me show you oc get deploy mynginx -o yaml and pipe that through less. I just want you to be aware if you catch the YAML of a running application that is possible, but there is so much information that you don't need to have in your own YAML. Like the creation timestamp and the generation, for instance, or the biggest part, the managed fields. So if you are going to store all of this in a YAML file, let me just do that. New.yaml is good enough as a name. And if we put all of this in this new YAML file, then we need to do some cleaning up. So creation timestamp can go, generation can go, manage fields can go and that's a lot as you can see. And that all stops right here. Then we have the resource version can go, the self link, the UID can go. Here we are in the deployment specification where we see all these different deployment properties. It's nice to have them in here, but not really necessarily, but let's only clean up the necessary. Last part that needs cleaning up it's status. Status is status information that is generated by the closer. And really, if you want to manage your applications by using YAML files, you don't need any status information. So this is a reasonably clean YAML file and using this YAML makes it possible to easily redeploy the same application with just a newer name, mynginx newer for instance. If you do that, don't forget to also change the labels because by default, every application that you generated using oc create new, will get a label that is sets to the key app and the value name of the application. And these values name of the application must be changed as well, because otherwise there's a risk that your new deployment is going to include and manage old ports as well. And that's gonna be a kind of messy, so please avoid it.

### 7.7 Using Services to Access Pods

- We have now talked about applications a little bit. Let's talk about providing access to applications using services. So what is this service all about? Well, in OpenShift, a Pod software-defined network is provided for Pod access. The fact that it is a Pod software-defined network, it means two things. It means that this SDN allows Pods to communicate to one another even if they are on different nodes. It also means that it's software-defined network and it cannot be reached directly by external users. Also Pod IP addresses are volatile because Pods are dynamically being edited and removed because of the scalability. So you never know which IP address is available. And if your Pod is going away, a new Pod is created, and this new Pod will have a new IP address. And for these reasons, Pods are not addressed directly, but services are used instead. Services are exposed on the cluster IP address. And by exposing them on the cluster IP address, we can create a unique entry point, a unique way of accessing these different services. And the service provides an IP address it can be used to access workloads that are running in Pods, either from within the cluster or from outside the cluster if a route is added. Services also provide load balancing when multiple Pods are used in a replicated setup. In that scenario, the user is accessing the service and the service is doing API based load balancing. That is load balancing without having any external LoadBalancer that runs the software but where the API is making a direct connection between the service and the Pods that is running in the background. In this drawing, I want to explain services. In order to understand services, we should start with nodes in a Kubernetes cluster. So if these are the physical nodes, these physical nodes are connected to physical network of course. This is a network where external users are coming in. Now on this physical nodes, there is the API server. The API server creates the cluster network. And this cluster network is a network where all of the nodes are connected to as well. The cluster network is a network that is used for administrative purposes. The cluster network hosts many things, but there's one thing that is not running on the cluster network and that is your Pods. So your Pods are running somewhere here on your Pod network. So if you are running different replicated Pods in a deployment, you can have Pod 1, and Pod 2, and Pod 3, for instance, being part of one deployment. And you can have a different set of Pods as well like Pod 4, and Pod 5, and Pod 6 being a part of the other deployment. Now there is one thing that you should realize, and that is from the outside, which is right here. This is where your user is living from the outside, there is no direct access to these Pods. Now, how does that work? Well, that works by adding services. So services are API resources that expose access to the different Pods. And in a service, you don't really care about deployments or whatever, services connect to the cluster net. And they can use two different types. One type is ClusterIP and the other type is NodePort. But regardless of type of service that you are using, the services are used as an API based LoadBalancer to provide access to the different Pods. You will see that in upcoming demos that in your services the service as endpoints and these endpoints allow the services to know about the different Pods. And in order to do so, the services have an IP address that is reachable on the ClusterIP only and not from the outside. Now, how does that make it usable? Well, if you would look at it like this it doesn't, but if you have NodePort service, in a NodePort service, something in addition can be added. And that is the node port. So what is the node port doing? The node port is connecting to ports that is allocated on the cluster nodes. That is dynamically allocated high port, it starts somewhere after port number 31,000. And the idea is that if traffic hits this specific port, the traffic will be forwarded to the NodePort. So it's port forwarding that it's all about. And that is one way of exposing traffic, not the preferred way though. We will talk about the preferred way later. Notice that this allows direct access from the outside by users that know which port the NodePort based service is connected on. It does not allow direct access to ClusterIP. ClusterIP type services are very convenient in microservices setup. So if you have microservices, then you may have a backend. The backend is not supposed to be accessible from the outside at all. Only the front end is supposed to be accessible from the outside and this scenario would work. So if these are your rep server front end, then you're expose them using node port. You connect the NodePort using variables in a ConfigMap about which we will talk later to connect to the ClusterIP, and that allows for this internal communication to happen. There's still one problem that we need to talk about. And that is how are you going to tell Mr. Happy user that he should connect to a random port ID, that's not going to happen. And that is why in OpenShift, there is routes as well, and the OpenShift route really connects to service. So if we create a route, the route exposes the service. The route exists as a LoadBalancer. So it's running as software in your cluster as a port actually. It exists as LoadBalancer, it has an API connection to the service. And because of the API connection to the service, the route knows what to connect to. So the route will port forward to these physical node ports and the route will make the load balancing decision to make sure that the user traffic is connected to somewhere in the cluster. So how does that work? Well, that works because the route is exposing an FQDN, a DNS name. So Mr. Happy user will connect using the FQDN. The DNS server that's involved which I have not included in this drawing will forward the traffic to the route. The route will use the LoadBalancer. The LoadBalancer will forward the traffic to the NodePort or, and that's the possibility of routes as well, directly to the ClusterIP. That is because the route has an API level connection with its backend service. You can use it to expose ClusterIP as well, no need to configure your service as the NodePort. But in this case, I'm using NodePort as an example. So what is going to happen? Well, based on the NodePort, the port forwarding through the NodePort services happening and the service on its turn is doing the port forwarding to the actual Pod and that makes that Mr. Happy user is in business and your application is exposed. And this is how exposing applications in an OpenShift environment is working. So now that we have a little bit of an understanding about services, let's talk about Pod to service connection. So services are using selector labels to find the Pods that they should connect to. It's like between replicas and ports as we have seen in the previous videos. Services are also looking for specific labels that are set on the Pods. And the Pods themselves know which services they are connected to by two environment variables that are automatically assigned to running Pods. That is SVC\_NAME\_SERVICE\_HOST which is a service IP address, in which SVC\_NAME is the name of the service and SERVICE\_PORT which is a service port. So you cannot as easily make a service believe that a Pod is connected to it because it also goes to the other way around, the Pod knows which service it is looking for. Services themselves are automatically registered with the Kubernetes internal DNS server, which makes them accessible through DNS. And by default that would be SVC\_NAME.PROJECT\_NAME.svc.clustername. And that's kind of convenient that there are these automatically generated DNS names so that the resources can reach out to one another in a very easy way. If you are wondering what a clustername is, you can use oc config get clusters or oc config current-context to find out the clustername, but you can also use use oc get svc to find these names. Now let's talk about the different Pod access options. That is because there are different types of services. ClusterIP provides an IP address that is only accessible on the ClusterIP. And this IP address cannot be addressed directly by external users. NodePort provides a node port on the cluster nodes which allows users to connect to the service directly. In OpenShift, services are not addressed directly, use routes instead. Also as an alternative, you can use oc port-forward to expose a Pod port on the local workstation where oc client is used. oc port-forward is good for administrator or developer access, but it's not what you want to use to expose workloads to external users. But if you want to do it, this is how you do it. Now, even if you are not going to address services directly, I want to demonstrate how they are working anyway. As long as you remember, just use routes to expose and to access your applications. But for having some insight is good if you know how Pods can be addressed using services directly as well. All right, let's explore some services. oc get all, is showing us the result of OC new-app. If you use OC new-app, services created automatically. So we can see that the service/bitginx is right here. The IP address is 10.217.4.176, but it's a ClusterIP. Now, what does that mean? That means that if I'm trying to use curl to 10.217.4.176:8080, because that's the default port at nginx is listening on, I'm getting nothing, that's because it's a ClusterIP. And we have just seen that ClusterIP is happening on the cluster node only. So the only way to access it will be to physically access the cluster node and from there address the ClusterIP address. That's not something that we are going to do. Instead, I'm going to use oc edit, that's a new command. And this new command is what I'm going to use to change a couple of properties on this service resource. So oc edit on svc bitginx. And there we can see the type which is set to ClusterIP, I am going to set that to NodePort. If you use oc edit, you better be very sure what you are doing. So next, there is the ports. So we have a port with a name, and a port, and a protocol, and a targetPort. I am going to add, no I'm not going to add anything, because the nice thing about NodePort is that if you are using NodePort, these ports are automatically going to be assigned. If I use oc get svc again, it's just the services that we care about, then you can see how this is working. So here we have port 8080 which is exposed by NodePort 30278 and port 8443 which is exposed by NodePort 32353. And you know what that means, that means that I can use curl on crc ip 30278. Why the dollar crc ip? Because the crc ip is printing the IP address of my CodeReady containers node. And I'm exposing the NodePort on that specific IP address. So curl on that specific IP address is giving me 30278. Okay, let me also show you crc ip, this is the IP address that my CodeReady containers machine is using. But again, this is not what you want to be using. You want to be using something else which is called routes. Before I'm showing you the route, I would also like to show you oc port forward. This is not something I'm using on a daily basis. So let me use minus h so that we can figure out what exactly we are going to port forward. So this is showing us, okay, we have pod/mypod and that is going to be port forwarded to locally and pod IP address. So if I'm using oc port-forward, we port forward a pod, and the pod is bitginx. Oh no tab completion here, okay, let me do it a different way. Bitgnx, now we have the local port 9999:8080. And there we go. Oh, oc port-forward, I don't have to specify the resource type. There we go. You can see that local port 9999 is redirected to 8080. I should have started this in the background. So control z and bg to continue in the background and now I should be able to use curl to localhost:9999. There we go, welcome to nginx, as you can see, that's also working. I think we now know enough about services and port-forwarding, let's explore routes in the next video.