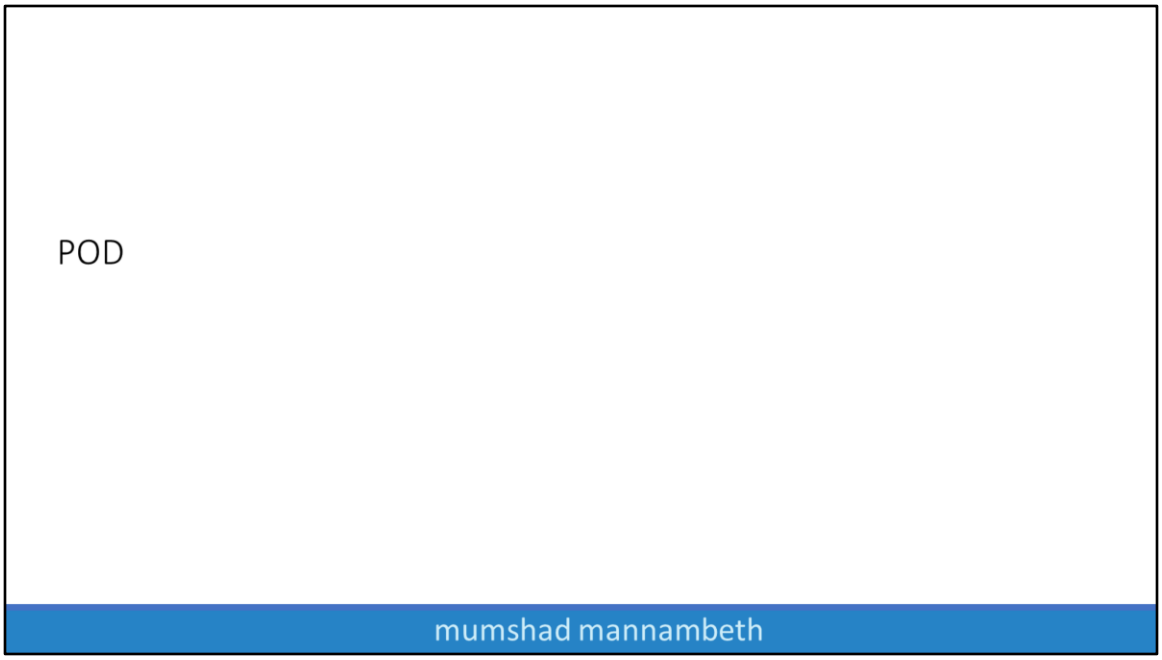
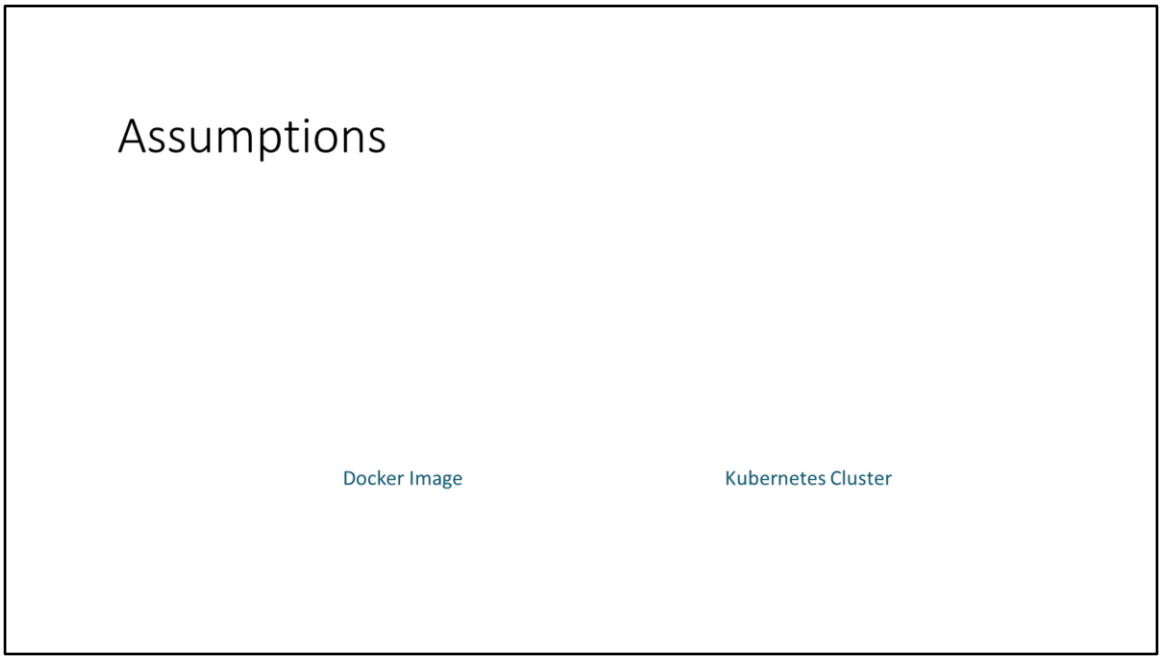
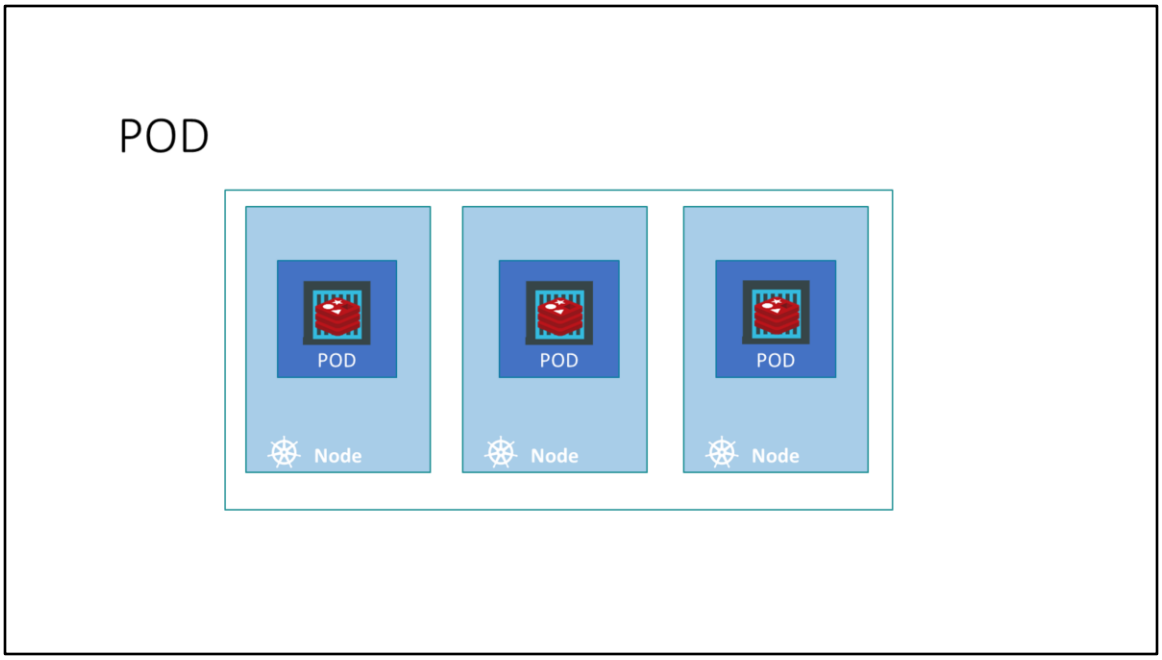

KodeKloud



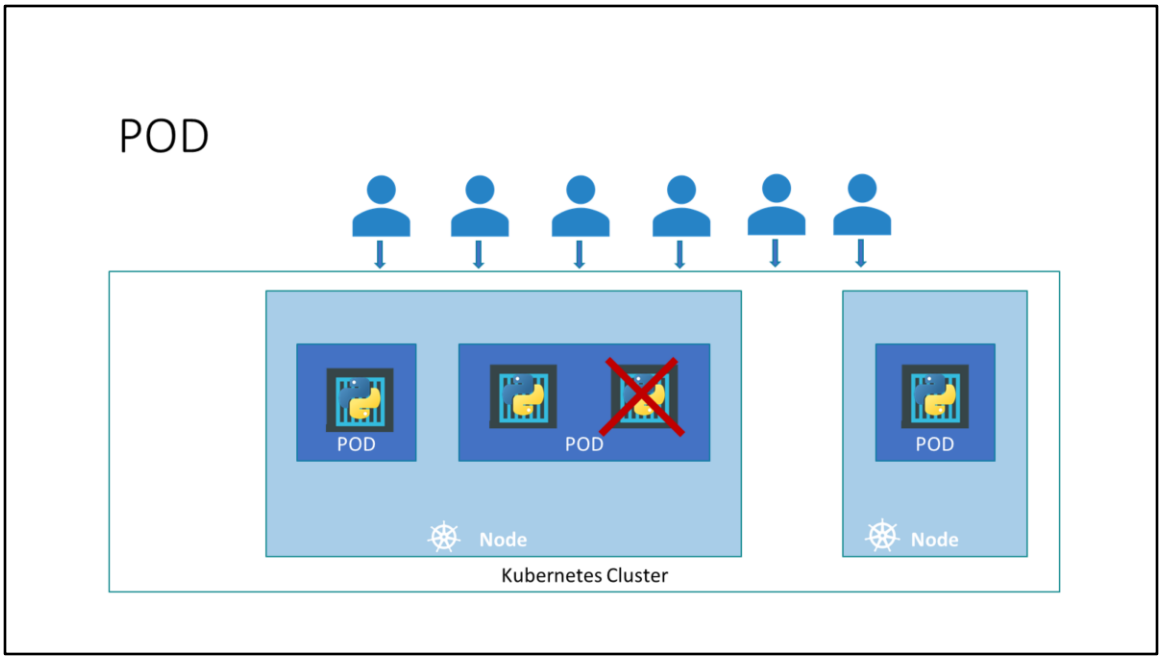
Hello and welcome to this lecture on Kubernetes PODs. My name is Mumshad Mannambeth and we are learning Kubernetes for Beginners. In this lecture we will discuss about Kubernetes PODs.



Before we head into understanding PODs, we would like to assume that the following have been setup already. At this point, we assume that the application is already developed and built into Docker Images and it is availalble 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. This could be a single-node setup or a multi-node setup, doesn't matter. All the services need to be in a running state.



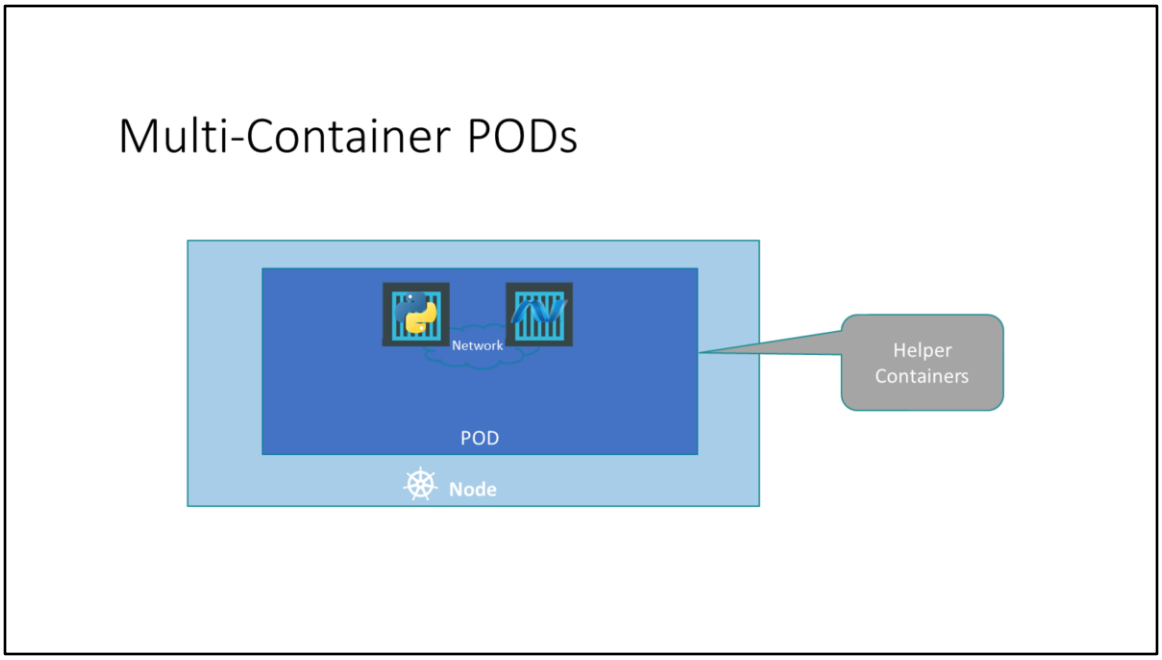
As we discussed before, <click> with kubernetes our ultimate aim is to deploy our application in the form of containers on a set of machines that are configured as worker nodes in a cluster. <click> However, kubernetes does not deploy containers directly on the worker nodes. The containers are encapsulated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



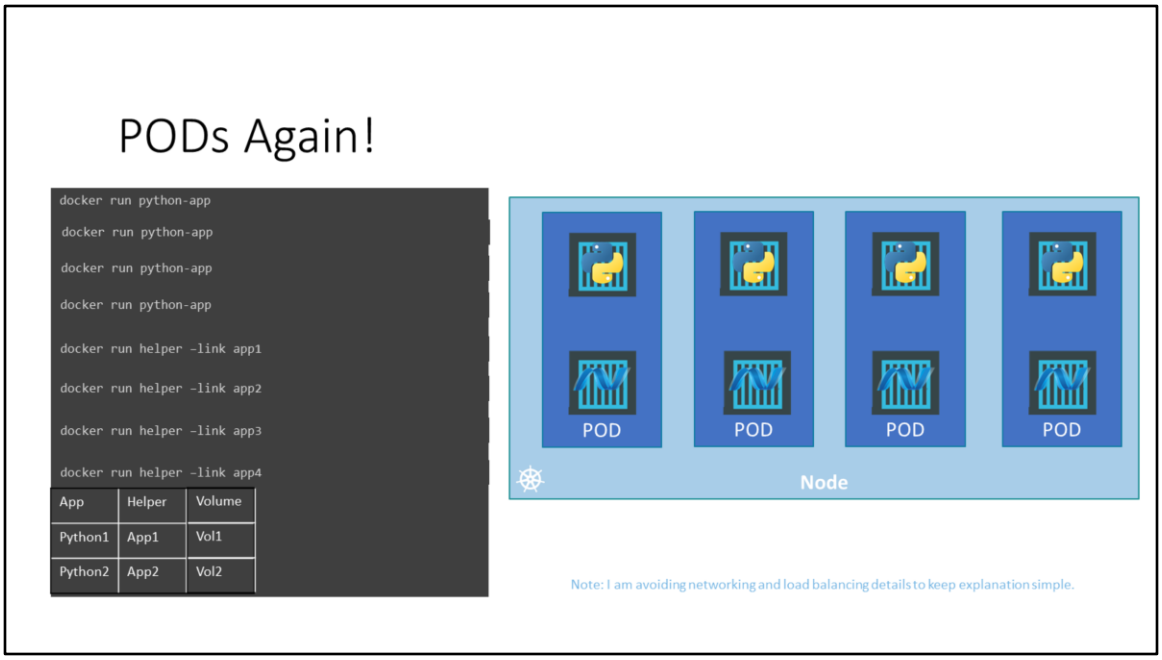
Here we see the simplest of simplest cases were you have a single node kubernetes cluster with a single instance of your application running in a single docker container encapsulated in a POD. <click> What if the number of users accessing your application increase and you need to scale your application?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

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

For now we are ONLY trying to understand the basic concepts.



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



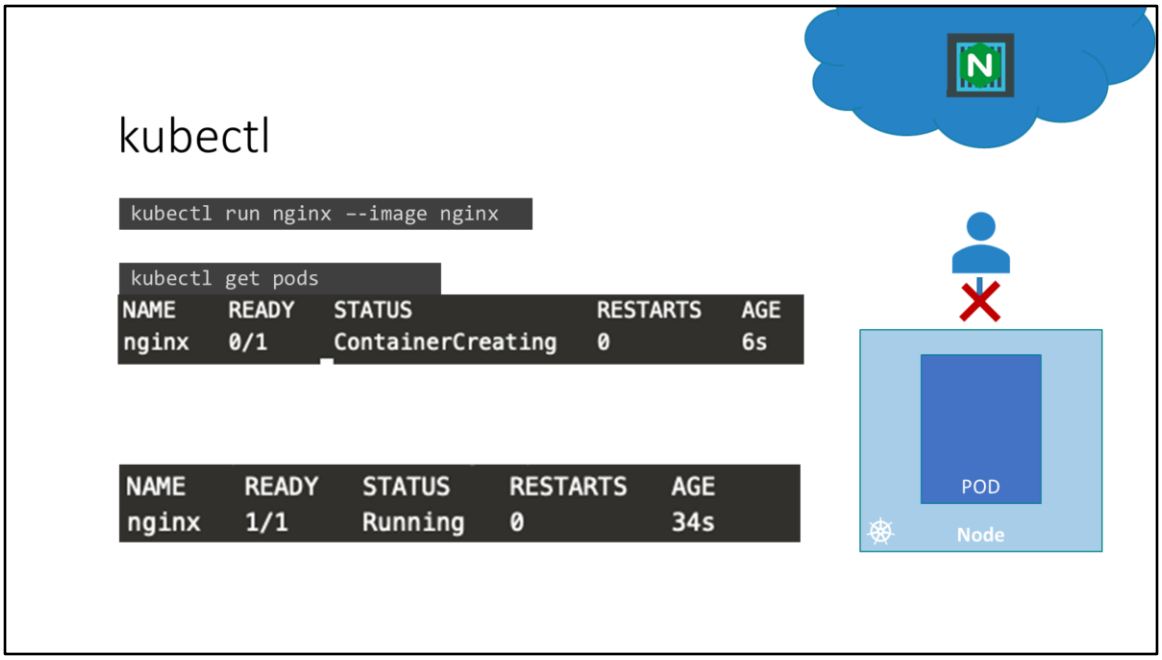
If you still have doubts in this topic (I would understand if you did because I did the first time I learned these concepts), we could take another shot at understanding PODs from a different angle. Let's, for a moment, keep kubernetes out of our discussion and talk about simple docker containers. Let's assume we were developing a process or a script to deploy our application on a docker host. <click> Then we would first simply deploy our application using a simple docker run python-app command and the application runs fine and our users are able to access it. <click> When the load increases we deploy more instances of our application by running the docker run commands many more times. This works fine and we are all happy. <click> Now, sometime in the future our application is further developed, undergoes architectural changes and grows and gets complex. We now have new helper containers that helps our web applications by processing or fetching data from elsewhere. These helper containers maintain a one-to-one relationship with our application container and thus, needs to communicate with the application containers directly and access data from those containers. <click> For this we need to maintain a map of what app and helper containers are connected to each other, we would need to establish network connectivity between these containers ourselves using links and custom networks, we would need to create shareable volumes and share it among the containers and <click> maintain a map of that as well. And most

importantly we would need to monitor the state of the application container and <click> when it dies, manually kill the helper container as well as its no longer required. When a new container is deployed we would need to deploy the new helper container as well.

<click> With PODs, kubernetes does all of this for us automatically. We just need to define what containers a POD consists of and the containers in a POD by default will have access to the same storage, the same network namespace, and same fate as in they will be created together and destroyed together.

Even if our application didn't happen to be so complex and we could live with a single container, kubernetes still requires you to create PODs. But this is good in the long run as your application is now equipped for architectural changes and scale in the future.

However, multi-pod containers are a rare use-case and we are going to stick to single container per POD in this course.



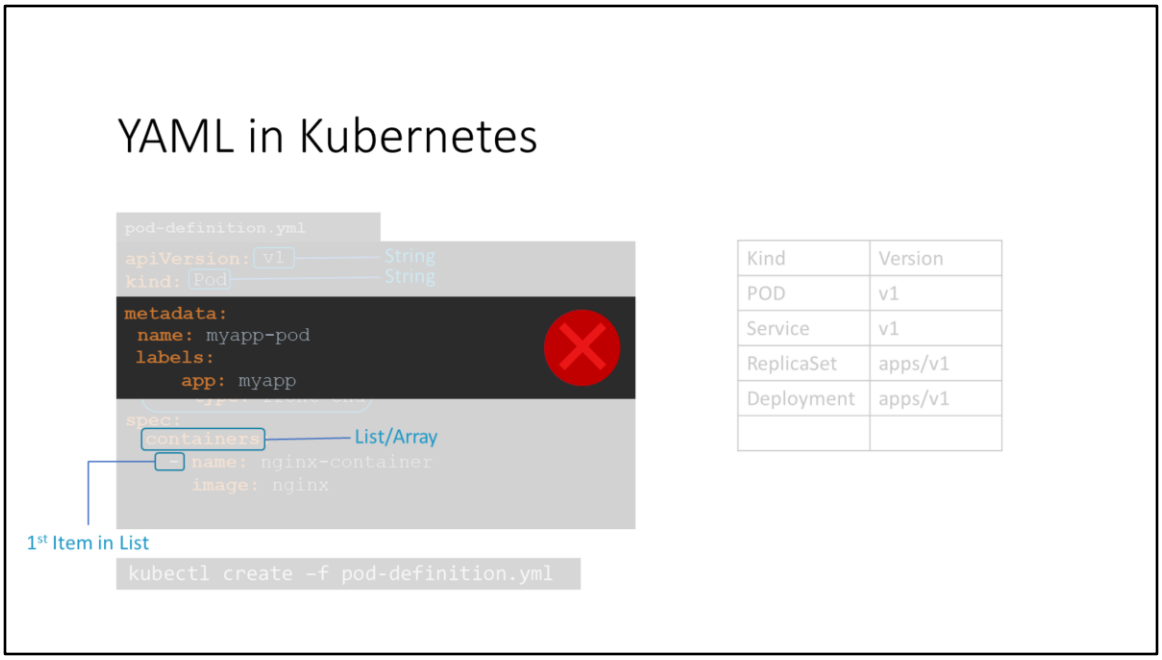
Let us now look at how to deploy PODs. <click> Earlier we learned about the kubectl run command. What this command really does is it deploys a docker container by creating a POD. So it first creates a POD automatically and deploys an instance of the nginx docker image. <click> But were does it get the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

Hello and welcome to this lecture, my name is Mumshad Mannambeth and we are learning kubernetes for beginners. In this lecture we will talk about creating a POD using a YAML based configuration file.



In the previous lecture we learned about YAML files in general. 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. Few other possible values for this field are apps/v1beta1, extensions/v1beta1 etc. We will see what these are for later in this course.

Next is the kind. The kind refers to the type of object we are trying to create, which in this case happens to be a POD. So we will set it as Pod. Some other possible values here could be ReplicaSet or Deployment or Service, which is what you see in the kind field in the table on the right.

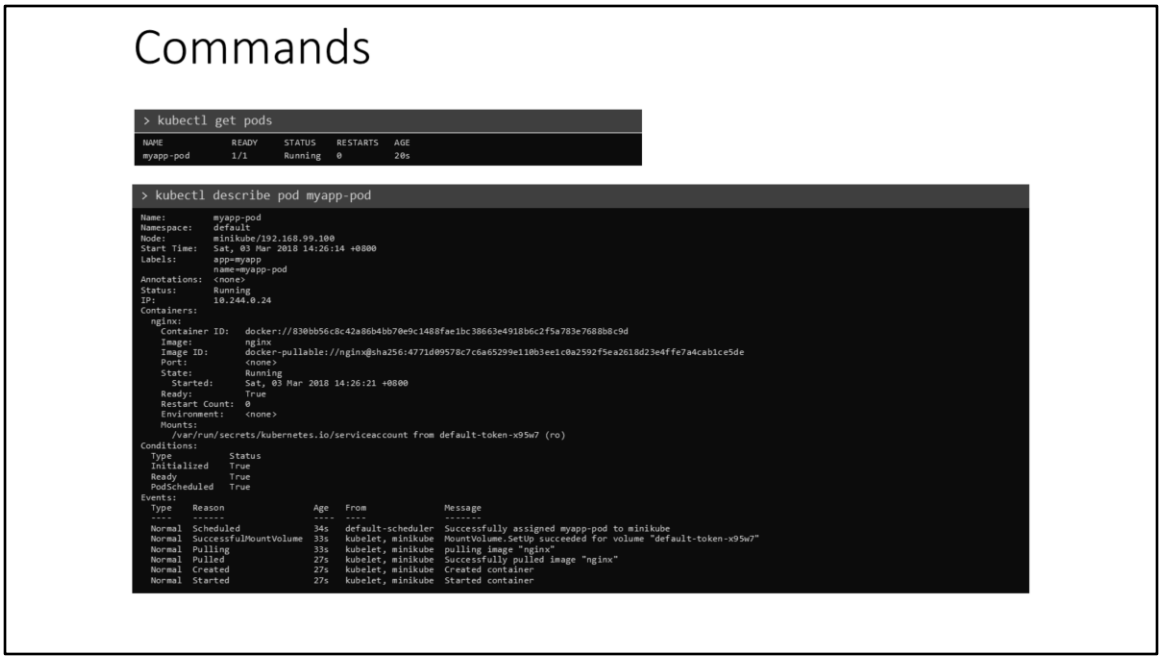
The next is metadata. <click> The metadata is data about the object like its name, labels etc. <click> As you can see unlike the first two were you specified a string value, this, is in the form of a dictionary. <click> So everything under metadata is intended to the right a little bit and so names and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

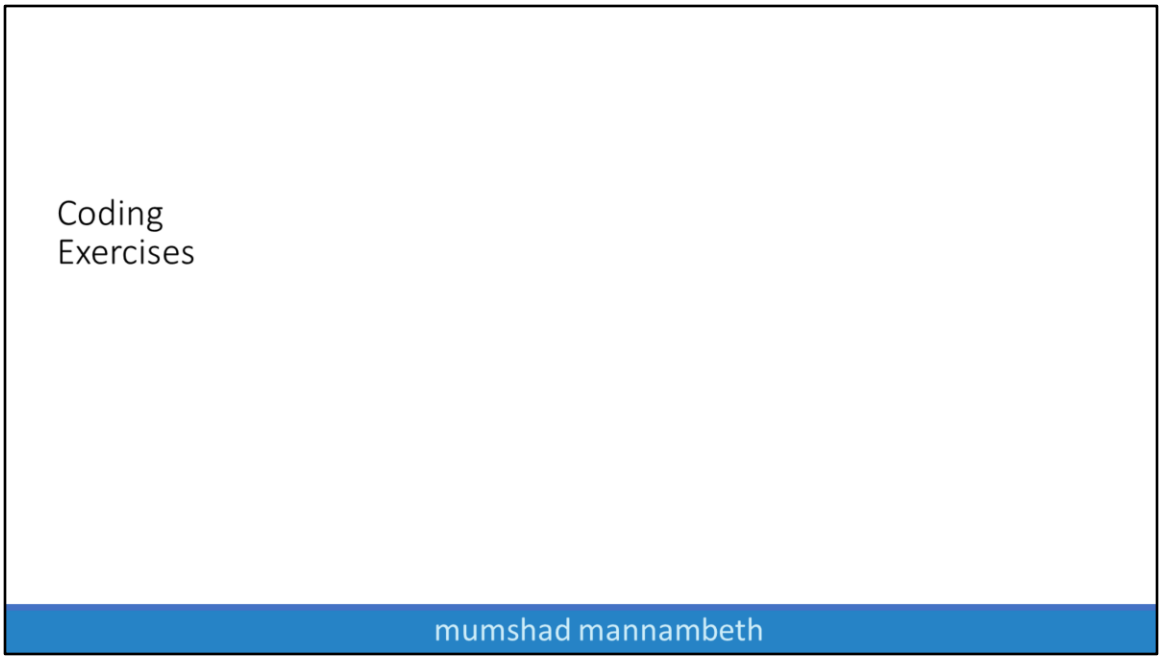
So far we have only mentioned the type and name of the object we need to create which happens to be a POD with the name myapp-pod, but we haven't really specified the container or image we need in the pod. The last section in the configuration file is the specification which is written as spec. Depending 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.



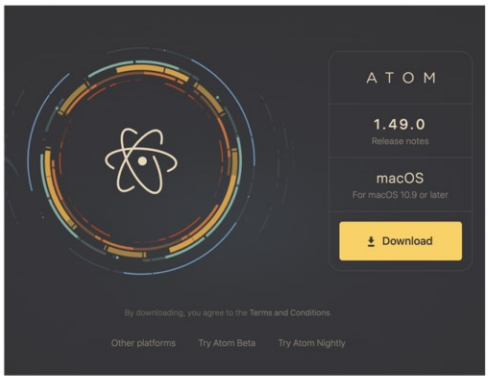
Once we create the pod, how do you see it? Use the <click> kubectl get pods command to see a list of pods available. In this case its just one. To see detailed information about the pod run the <click> kubectl describe pod command. This will tell you information about the POD, when it was created, what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

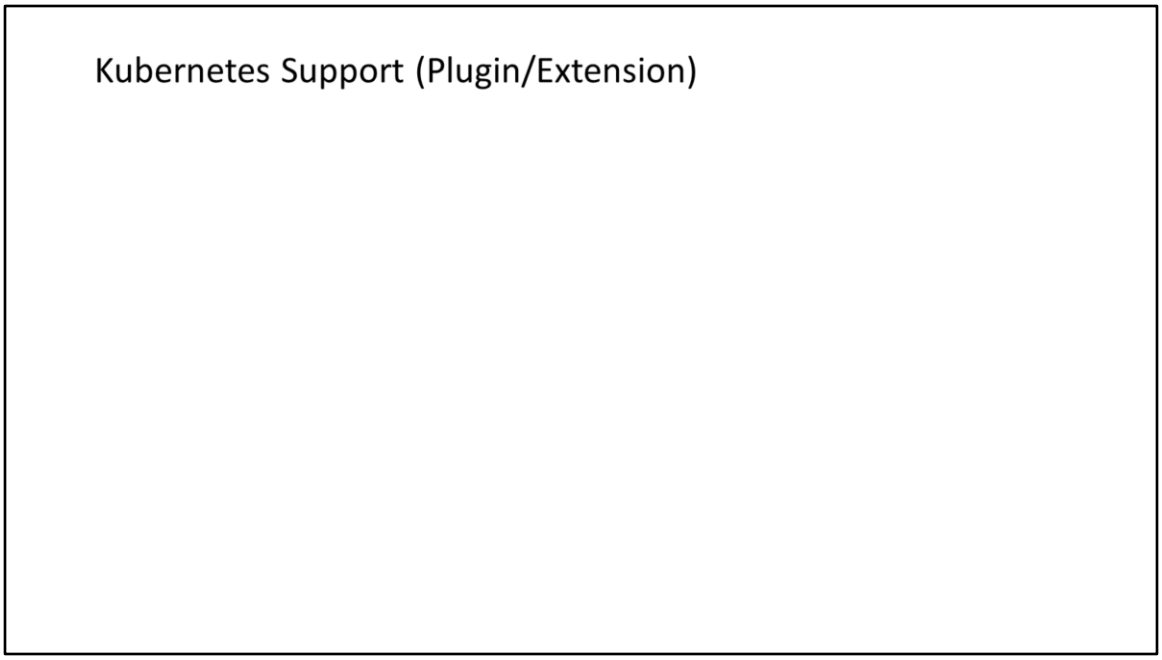


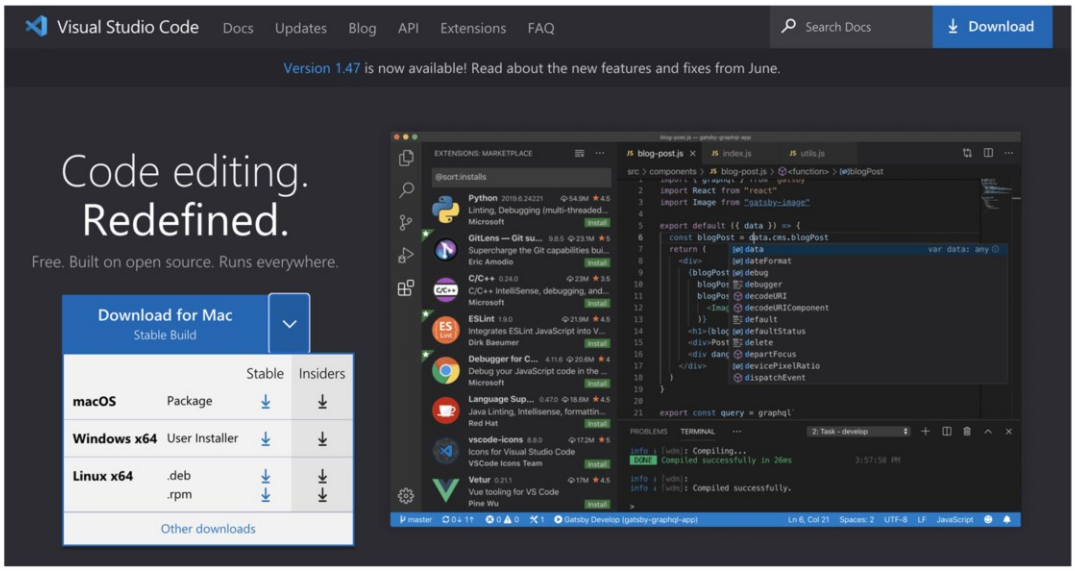
We just saw how to create a POD from command line. But you can also create a POD using a POD definition file. In fact most of Kubernetes configuration is done in definition files based in YAML format. When the application configuration gets complex, you will see that it is EASIER to use YAML files to define the configuration. As such at this point we will make sure that we are all comfortable with YAML files and its structure. This is important because going forward we will be working with YAML files extensively. The next lecture is going to be a very basic introduction to YAML file. And that will be followed by some coding exercises that will help you get familiar with YAML files using simple examples. If you are already familiar with YAML, please skip this lecture and coding exercises and move forward to the next lecture.

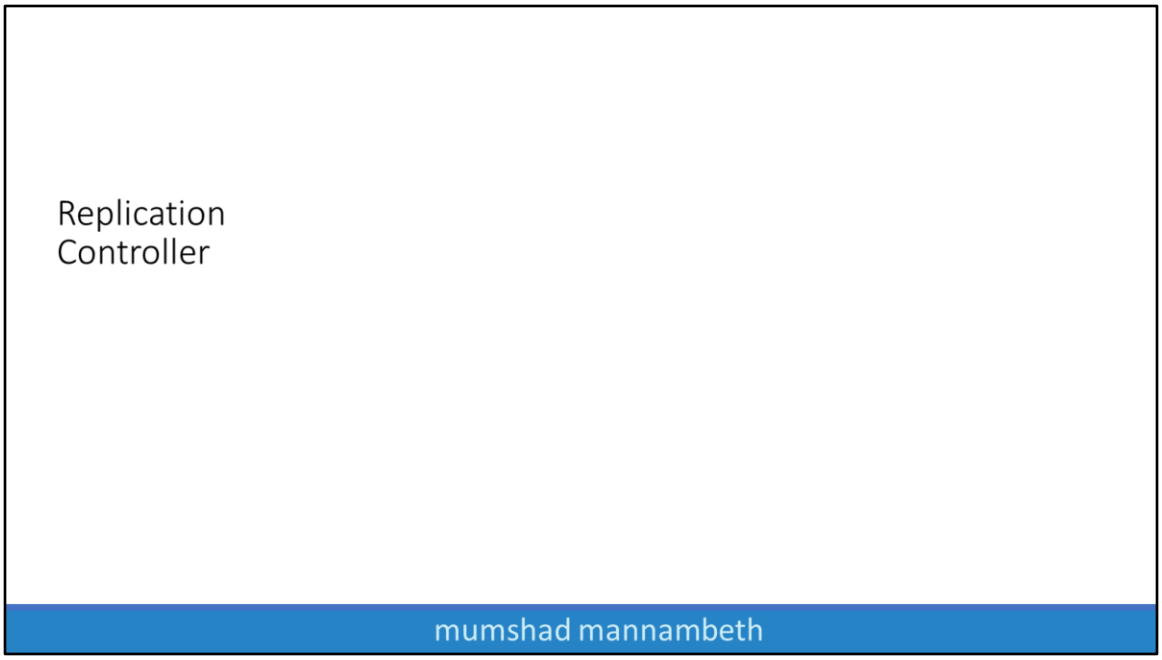
IDE (Integrated Development Environment)

Integrated development environment Software / JetBrains

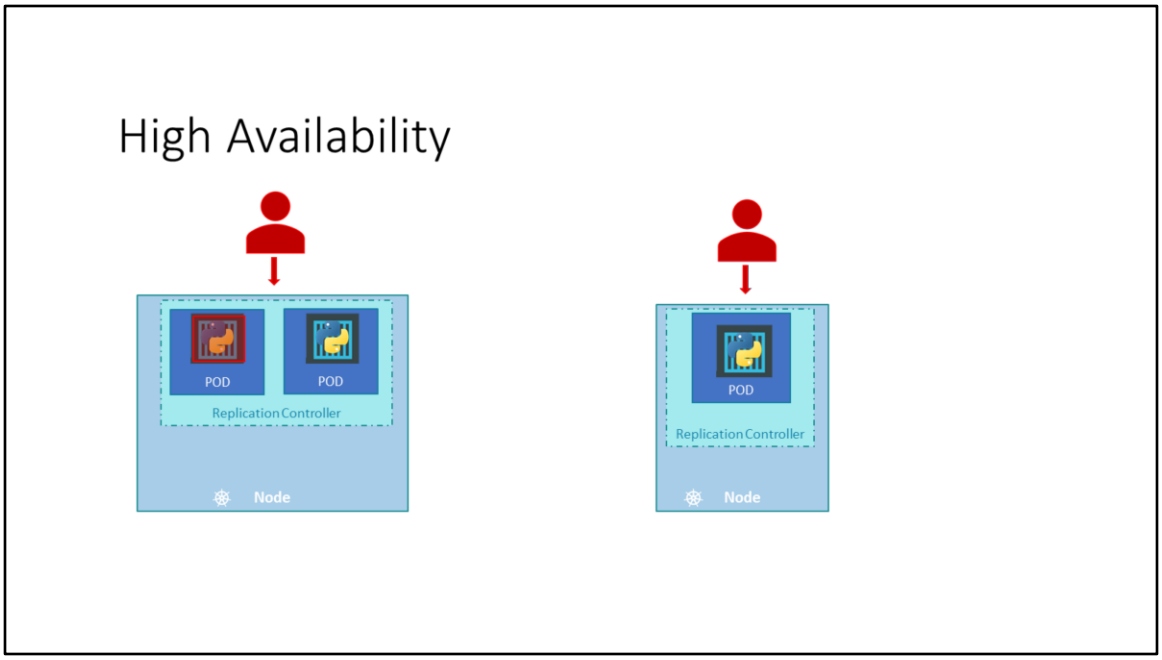



Kubernetes Support (Plugin/Extension)


Visual Studio Code
Docs
Updates
Blog
API
Extensions
FAQ
Search Docs
Download
Version 1.47 is now available! Read about the new features and fixes from June.
Code editing.
Redefined.
Free. Built on open source. Runs everywhere.
Download for Mac
Stable Build
Stable
Insiders
macOS
Package
Windows x64
User Installer
Linux x64
.deb
.rpm
Other downloads

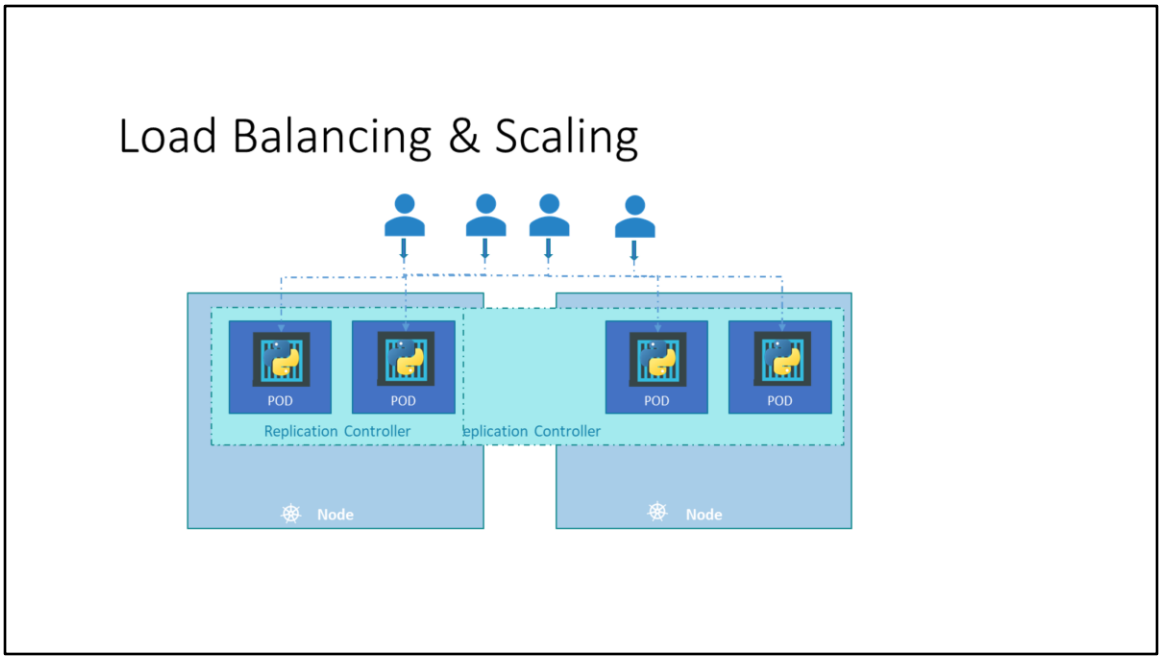


Hello and welcome to this lecture on Kubernetes Controllers. My name is Mumshad Mannambeth and we are learning Kubernetes for Beginners. In this lecture we will discuss about Kubernetes Controllers. Controllers are the brain behind Kubernetes. They are processes that monitor kubernetes objects and respond accordingly. In this lecture we will discuss about one controller in particular. And that is the Replication Controller.

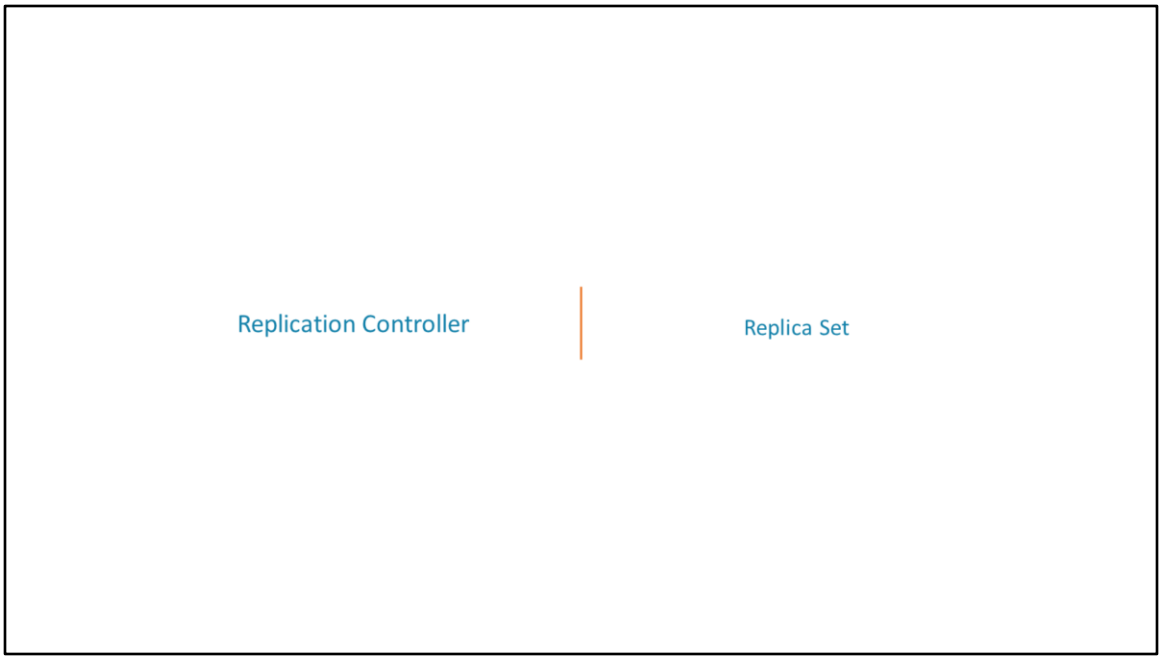


So what is a replica and why do we need a replication controller? <click> Let's go back to our first scenario were we had a single POD running our application. <click> What if for some reason, our application crashes and the POD fails? Users will no longer be able to access our application. To prevent 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. <click> The replication controller helps us run multiple instances of a single POD in the kubernetes cluster thus providing <click> High Availability.

<click> So does that mean you can't use a replication controller 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 replication controller ensures that the specified number of PODs are running at all times. Even if it's just 1 or 100.



Another reason we need replication controller 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 we deploy additional POD to balance the load across the two pods. <click> If the demand further increases 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 replication controller 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.



It’s important to note that there are two similar terms. Replication Controller and Replica Set. Both have the same purpose but they are not the same. Replication Controller is the older technology that is being replaced by Replica Set. Replica set is the new recommended way to setup replication. However, whatever we discussed in the previous few slides remain applicable to both these technologies. There are minor differences in the way each works and we will look at that in a bit.

As such we will try to stick to Replica Sets in all of our demos and implementations going forward.

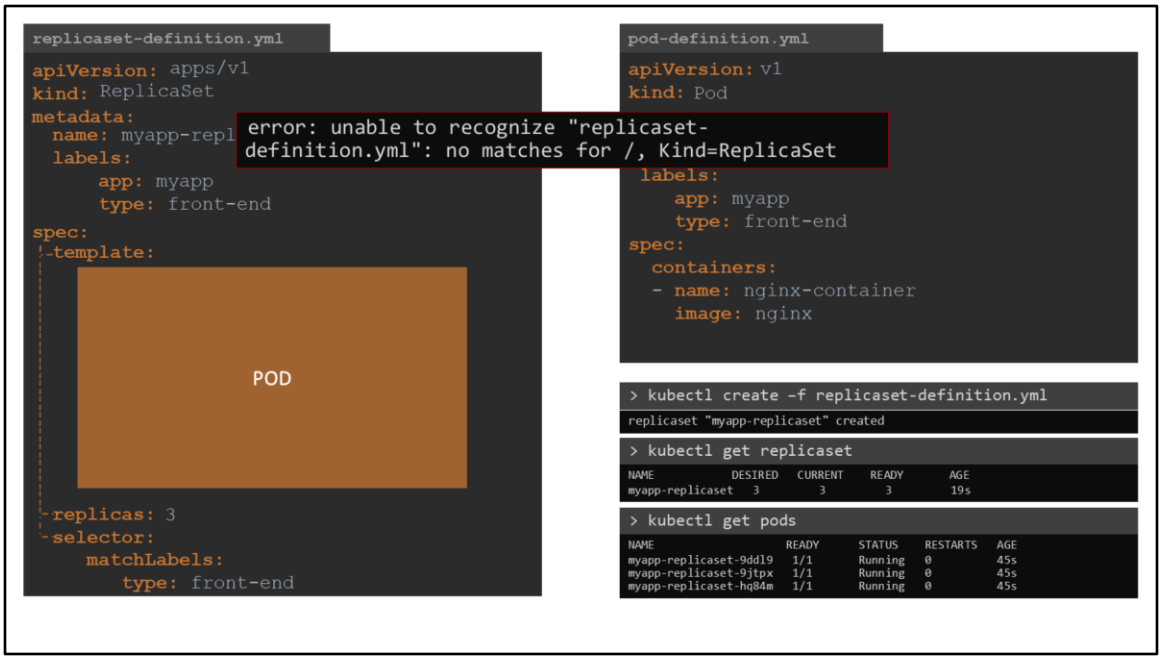
<Discuss more if possible>

Let us now look at how we create a replication controller. <click> As with the previous lecture, we start by creating a replication controller definition file. We will name it rc-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 replication controller is supported in kubernetes apiVersion v1. <click> So we will write it as v1. <click> The kind as we know is ReplicationController. Under metadata, we will add a name and we will call it myapp-rc. 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 replication controller 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 replication controller 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 Replication Controller 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.

Once the file is ready, <click> run the kubectl create command and input the file using the –f parameter. The replication controller Is created. When the replication controller 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 replication controllers <click> run the kubectl get replication controller command and you will see the replication controller 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 replication controller, 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 replication controller which is myapp-rc indicating that they are all created automatically by the replication controller.



What we just saw was ReplicationController. Let us now look at ReplicaSet. It is very similar to replication controller. <click> As usual, first we have apiVersion, kind, metadata and spec. The apiVersion though is a bit different. It is apps/v1 which is different from what we had before for replication controller. For replication controller it was simply 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>

<click> The kind would be ReplicaSet and we add <click> name and labels in metadata.

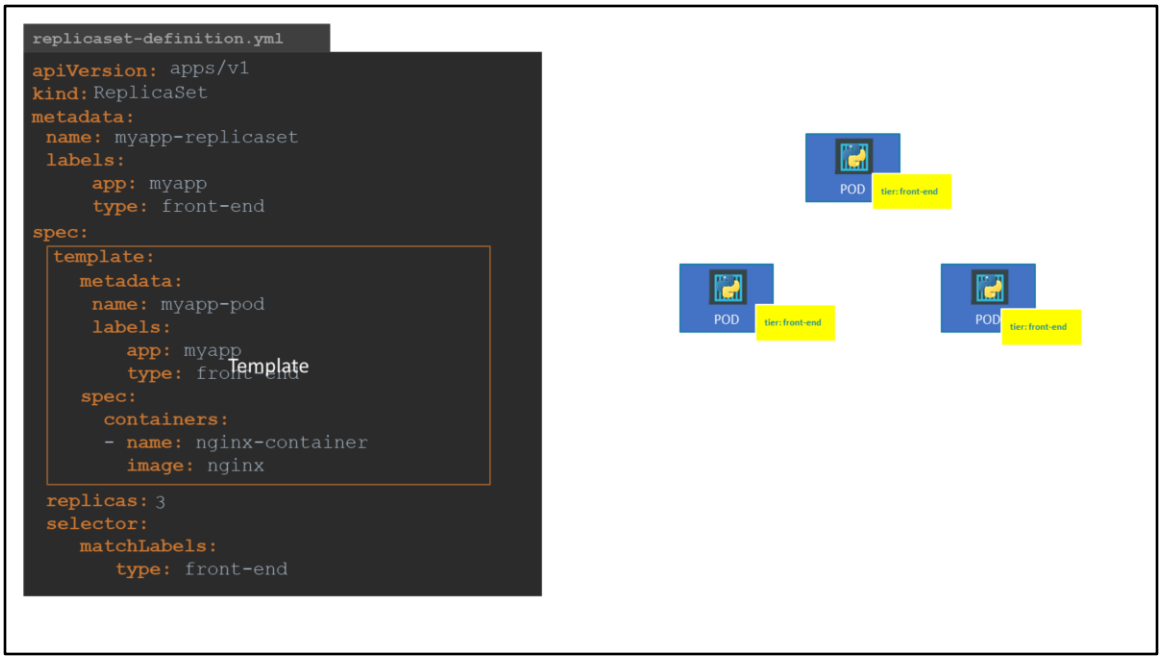
The specification section looks very similar to replication controller. <click> It has a template section were we provide pod-definition as before. <click> So I am going to copy contents over from pod-definition file. And we have number of replicas set to 3. However, there is one major difference between replication controller and replica set. <click> 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.

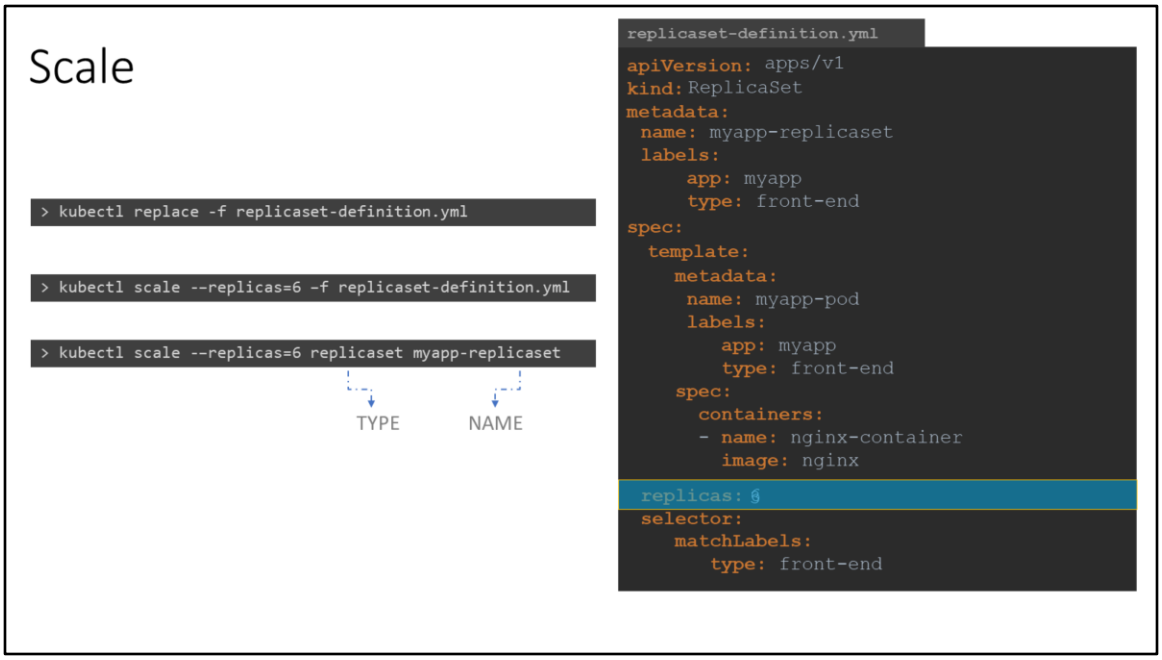
But before we get into that, I would like to mention that the selector is one of the major differences between replication controller and replica set. The selector is not a REQUIRED field in case of a replication controller, but it is still available. When you skip it, as we did in the previous slide, it assumes it to be the same as the labels provided in the pod-definition file. In case of replica set a user input IS required for this property. And 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.

<click> And as always to create a ReplicaSet run the kubectl create command providing the definition file as input and to see the created <click> replicasets run the kubectl get replicaset command. To get list of pods, <click> simply run the kubectl get pods command.

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.

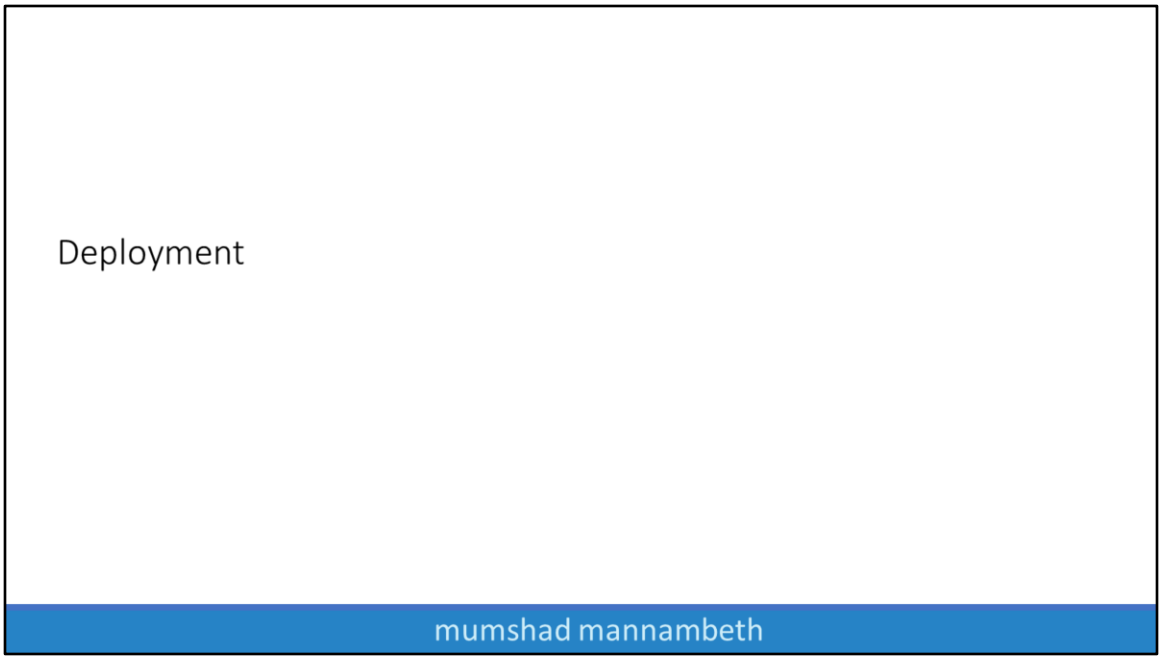


Let's look at how we scale the replicaset. Say we started with 3 replicas and in the future we decide to scale to 6. How do we update our replicaset to scale to 6 replicas. Well there are multiple ways to do it. The first, is to update the number of replicas in the definition file to <click> 6. Then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

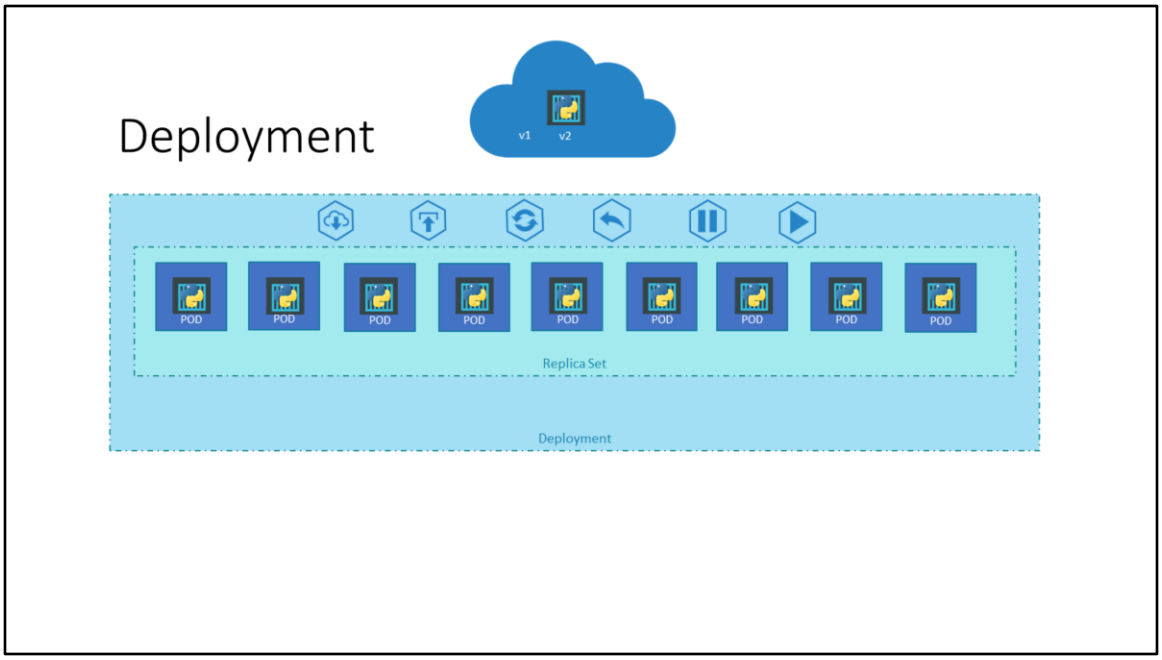
The second way to do it is to run the <click> kubectl scale command. Use the replicas parameter to provide the new number of replicas and specify the same file as input. <click> You may either input the definition file or provide the replicaset name in the TYPE Name format. However, Remember that using 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.



Hello and welcome to this lecture. My name is Mumshad Mannambeth and we are learning Kubernetes for Beginners. In this lecture we will discuss about Kubernetes Deployments.



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

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

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

Suppose one of the upgrades you performed resulted in an unexpected error and you are asked to undo the recent update. You would like to be able to <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 to apply each change 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 Replication Controllers or 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 deployments.

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.

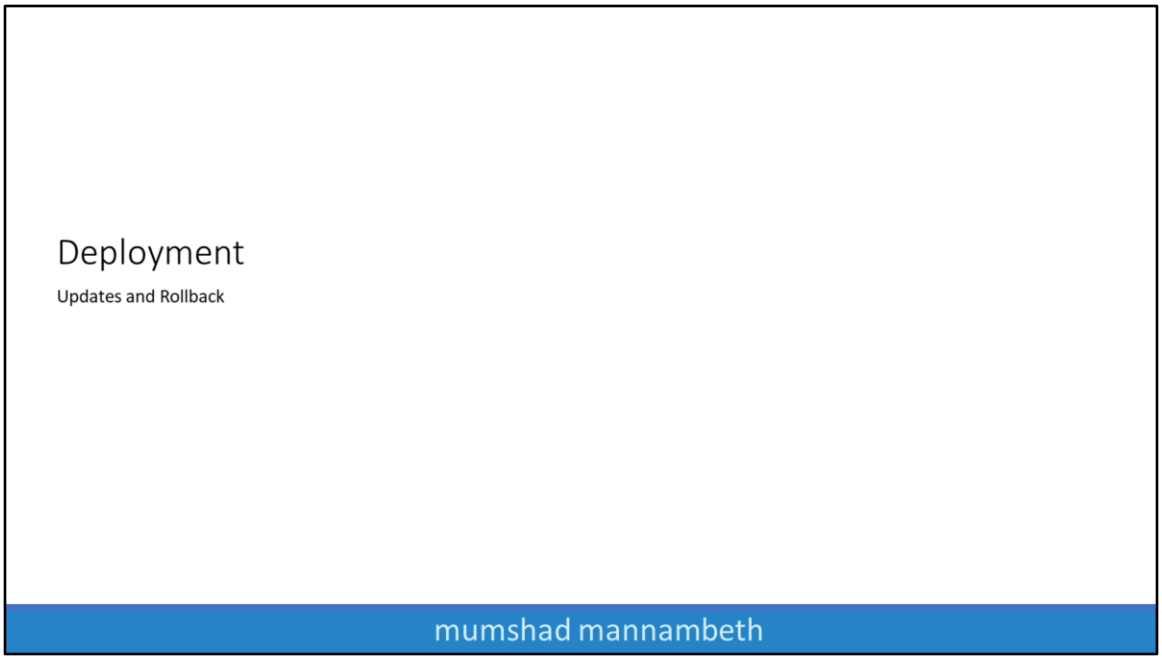
So far there hasn't been much of a difference between replicaset and deployments, except for the fact that deployments created a new kubernetes object called

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.

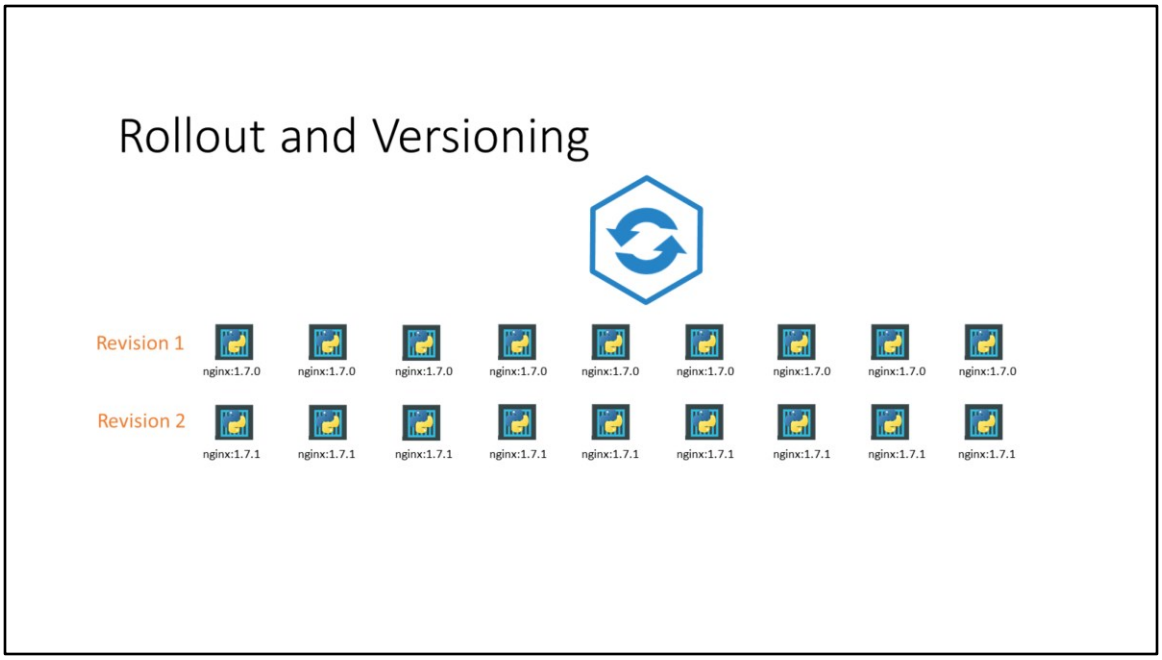
commands

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

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

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

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



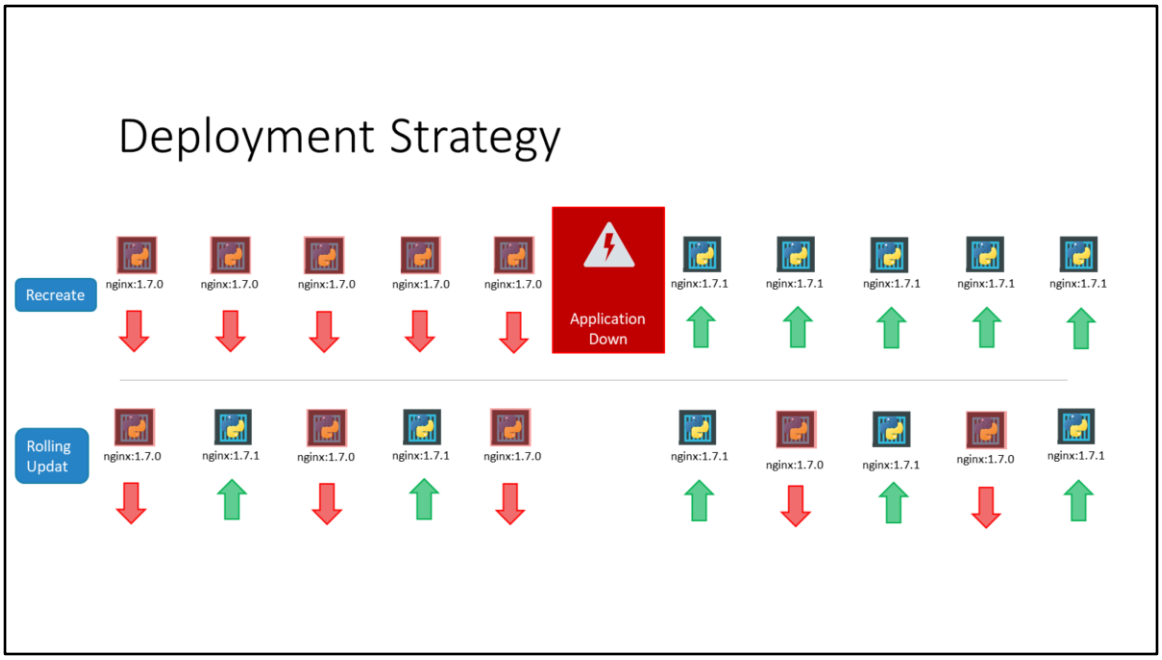
In this lecture we will talk about updates and rollbacks in a Deployment.



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

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

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



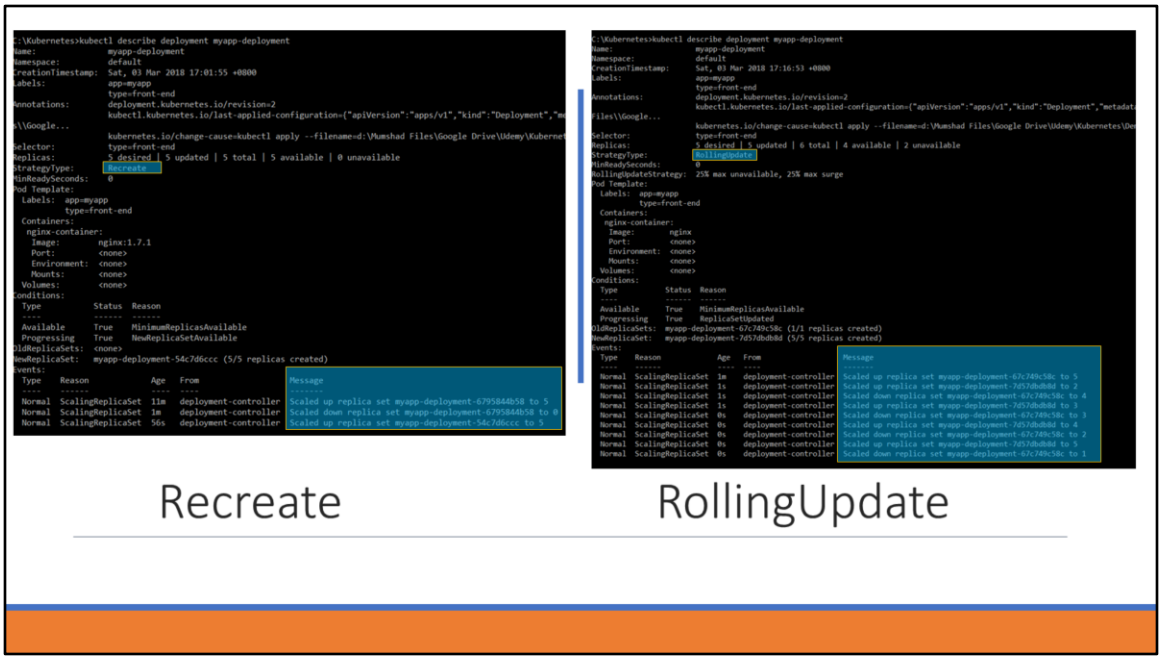
There are two types of deployment strategies. Say for example you have 5 replicas of your web application instance deployed. One way to upgrade these to a newer version is to destroy all of these and then create newer versions of application instances. <click> Meaning first, destroy the 5 running instances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

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

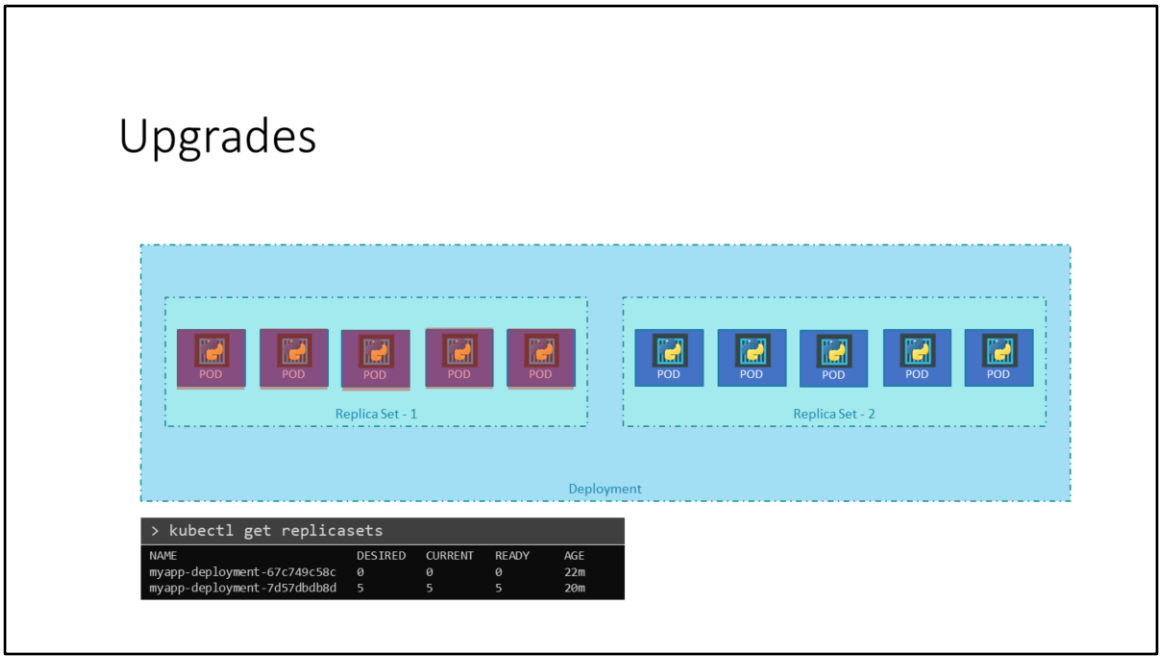
Remember, if you do not specify a strategy while creating the deployment, it will assume it to be Rolling Update. In other words, RollingUpdate is the default Deployment Strategy.

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

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

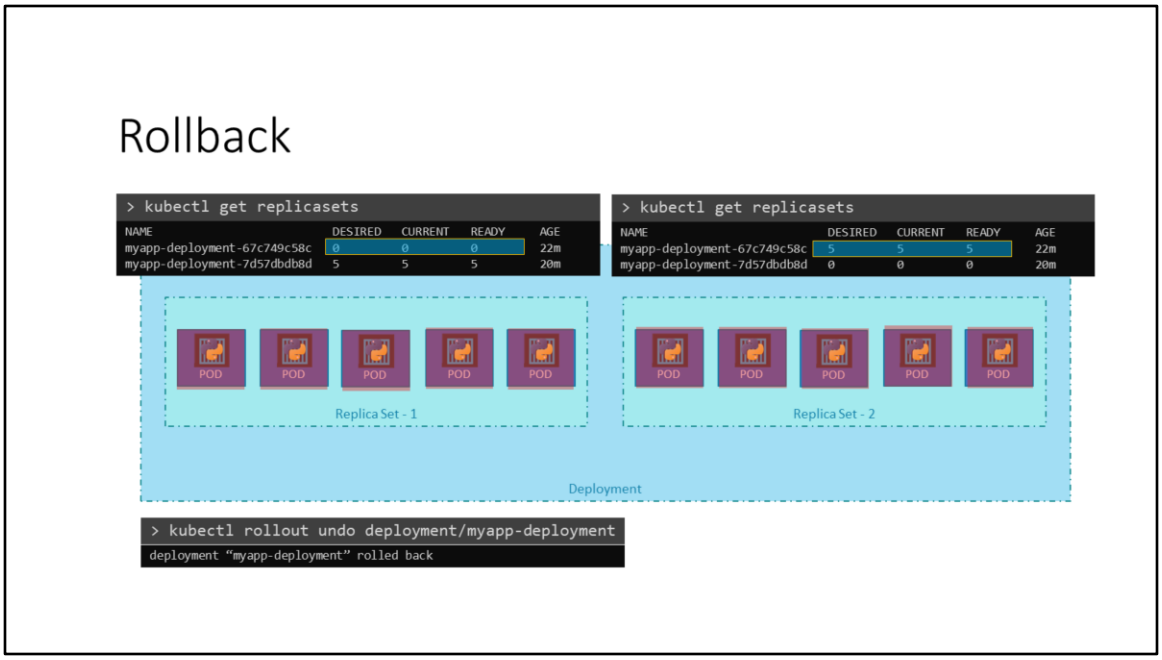


The difference between the recreate and rollingupdate strategies can also be seen when you view the deployments in detail. Run the kubectl describe deployment command to see detailed information regarding the deployments. <click> You will notice when the Recreate strategy was used the events indicate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



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

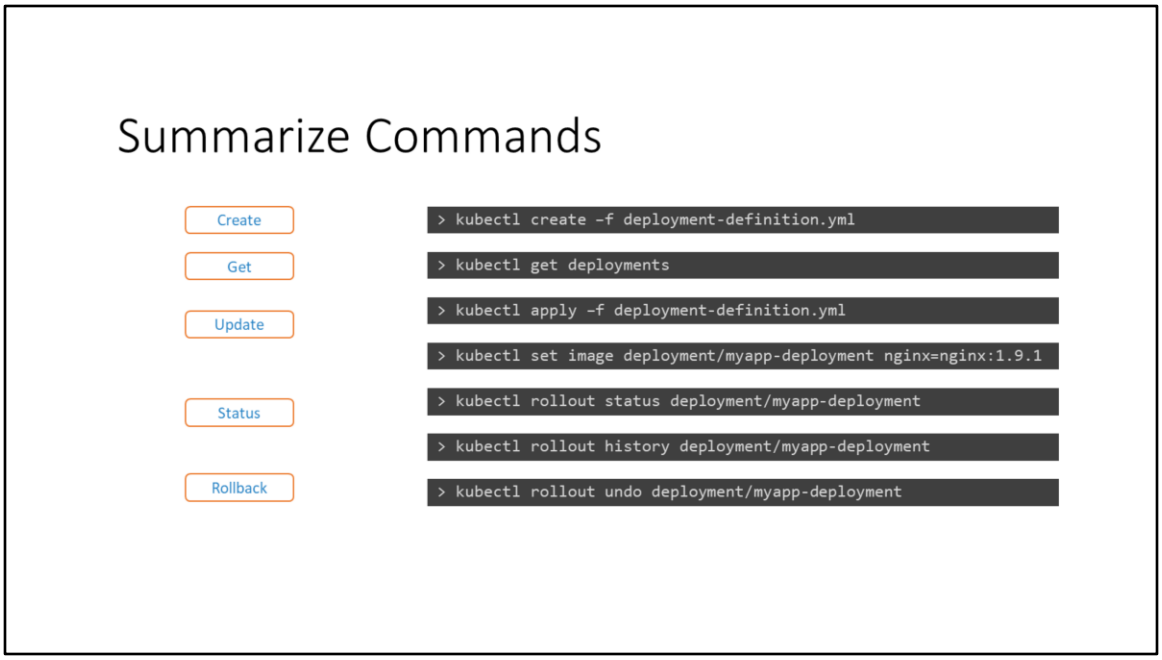
This can be seen when you try to list the replicasets using the kubectl get replicasets command. Here we see the old replicaset with 0 PODs and the new replicaset with 5 PODs.



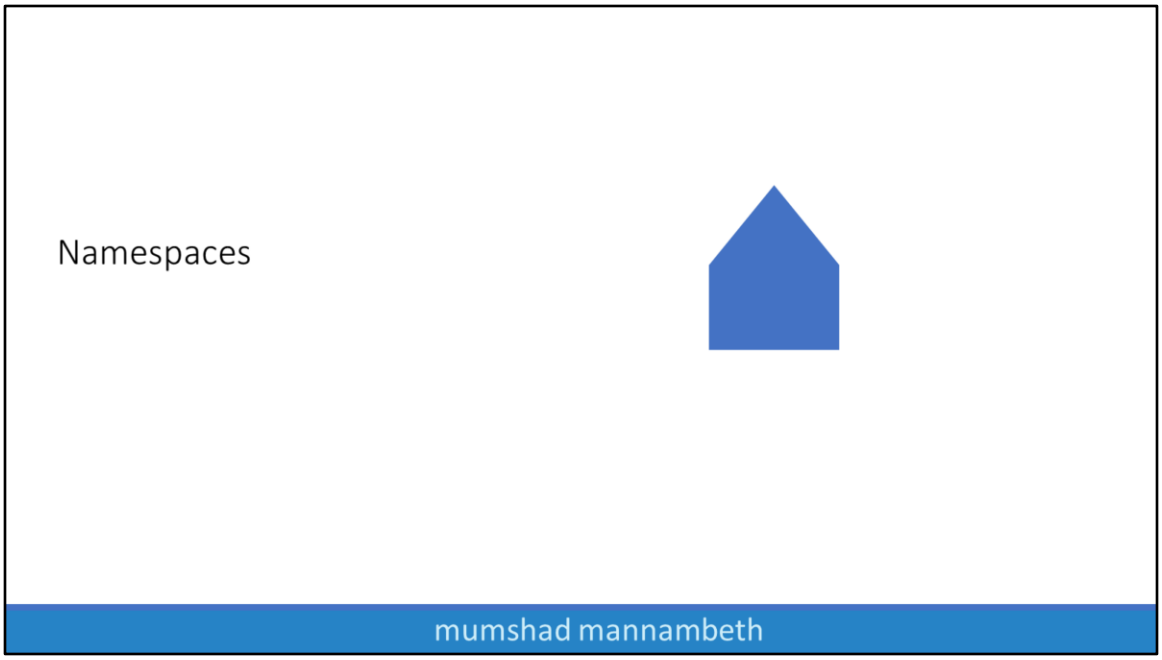
Say for instance once you upgrade your application, you realize something isn't very right. Something's wrong with the new version of build you used to upgrade. So you would like to rollback your update. Kubernetes deployments allow you to rollback to a previous revision. <click> To undo a change run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

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

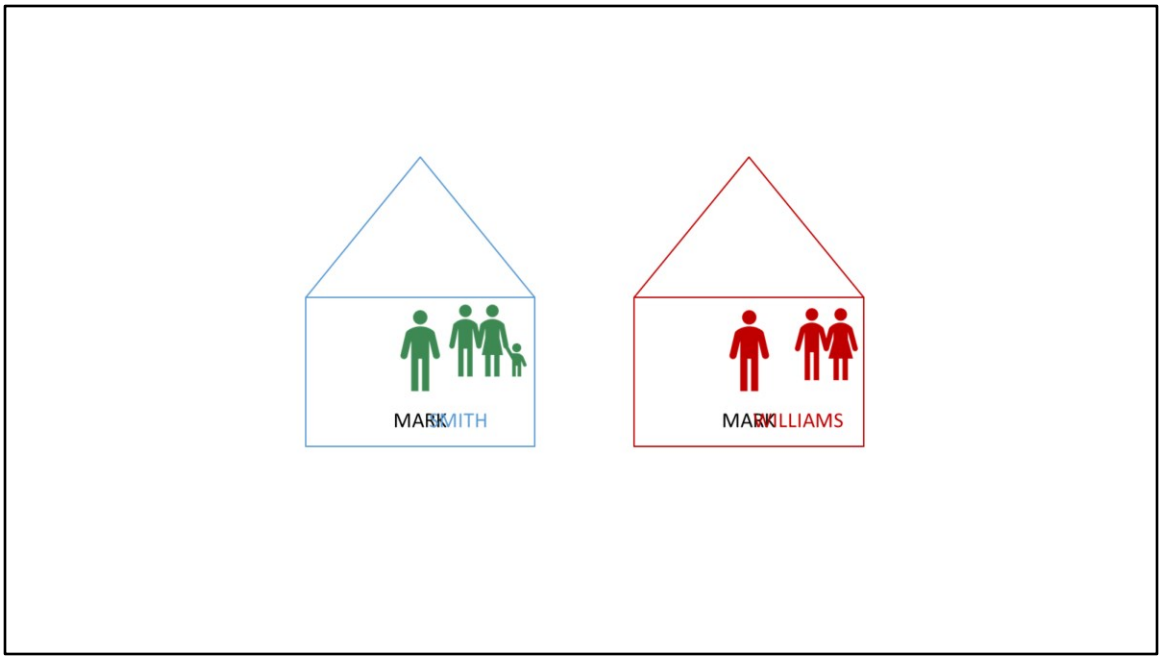
And finally let's get back to one of the commands we ran initially when we learned about PODs for the first time. <click> We used the kubectl run command to create a POD. This command infact creates a deployment and not just a POD. This is why the output of the command says Deployment nginx created.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



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



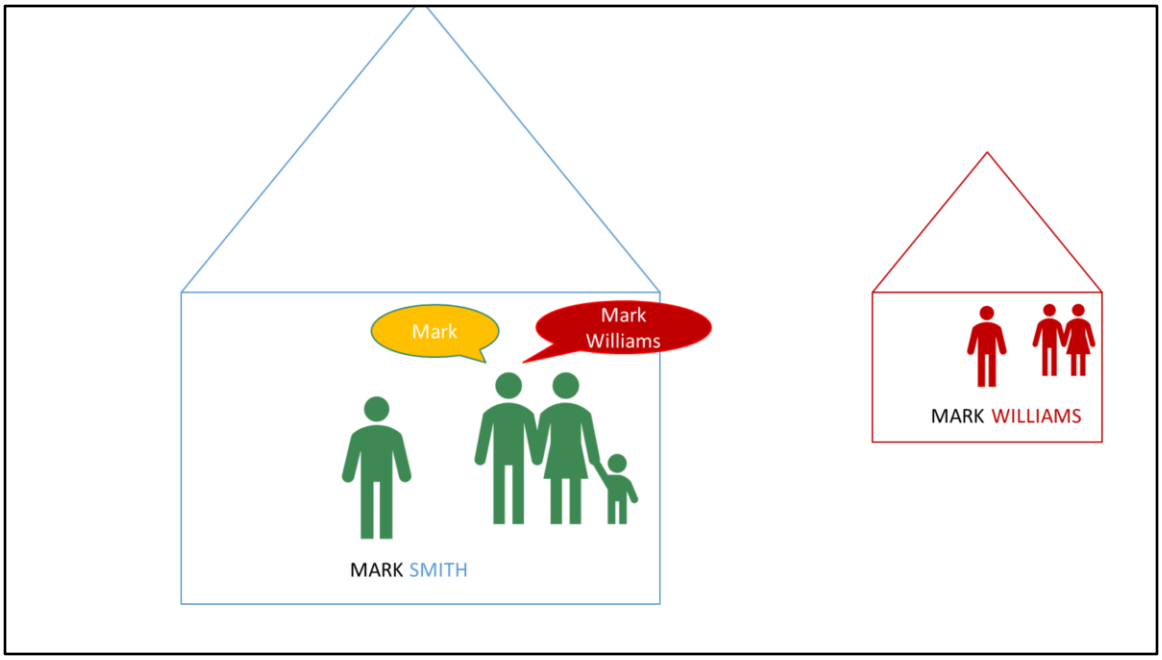
In this lecture we will discuss about namespaces in Kubernetes.



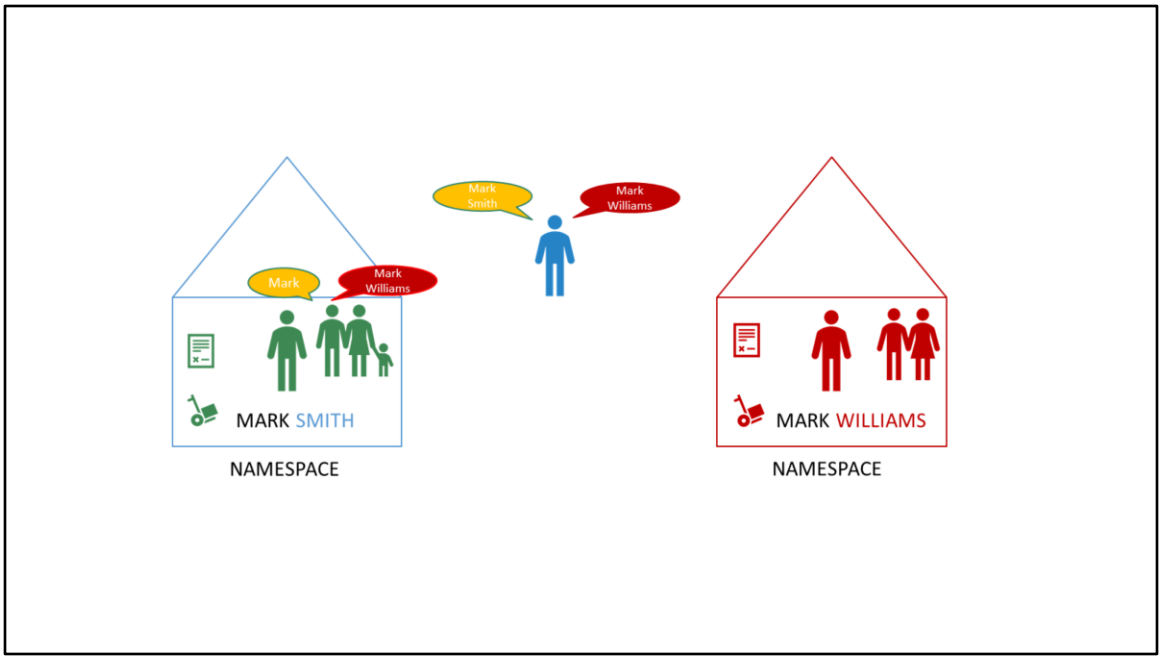
Let us begin with an analogy. There are two boys named Mark. To differentiate them from each other, we call them by their last names. Smith and Williams.

They come from different houses. Of course, the Smith's and the William's.

There are other members in the house.



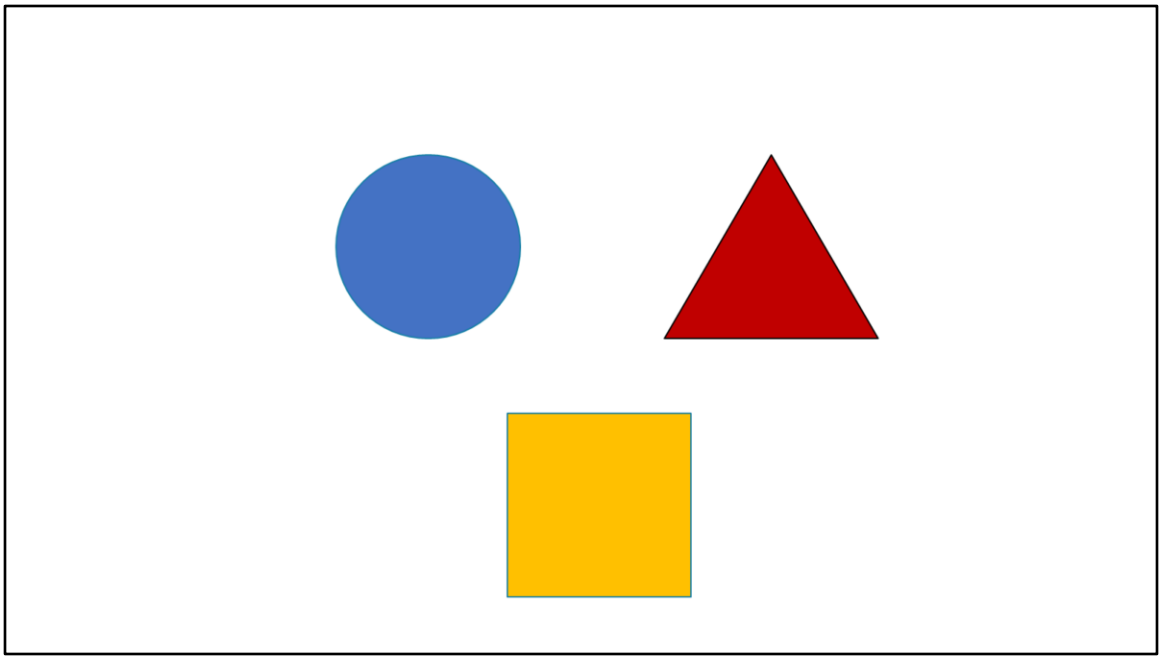
The individuals within a house refer to each other simply by their first names. For example, the father addresses mark simply as Mark. However, if the father wishes to address the mark in the other house he would use the full name.



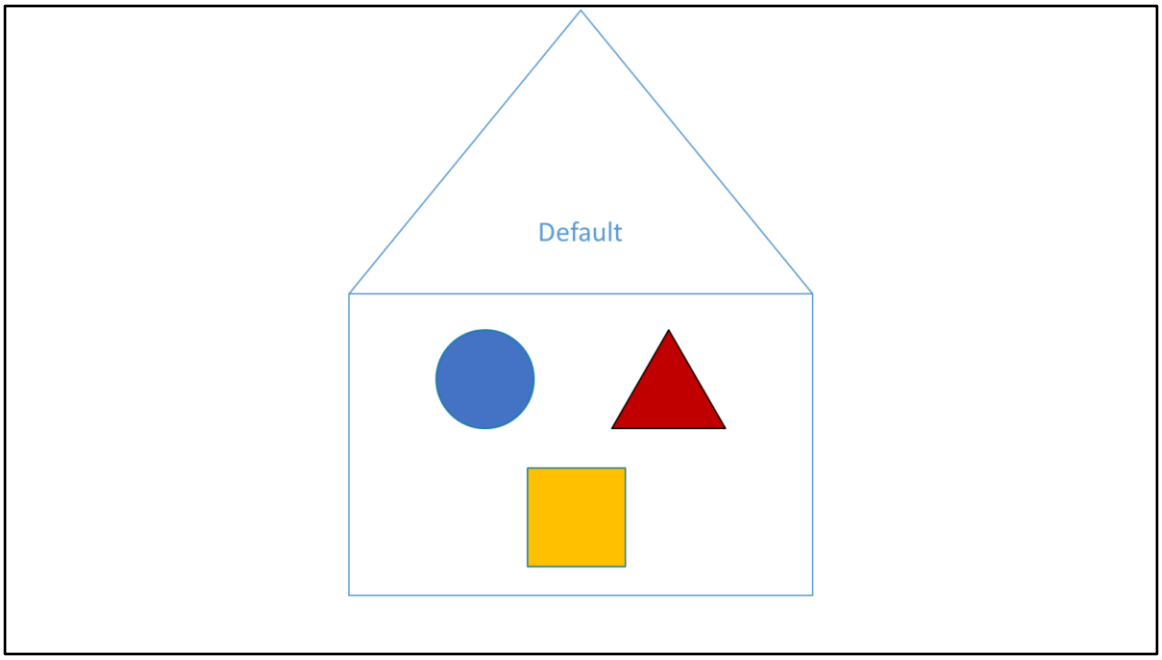
Someone outside of these houses would also use the full name to refer to the boys or anyone within these houses.

Each of these houses have their own set of rules that defines who does what. Each of these houses have their own set of resources that they can consume.

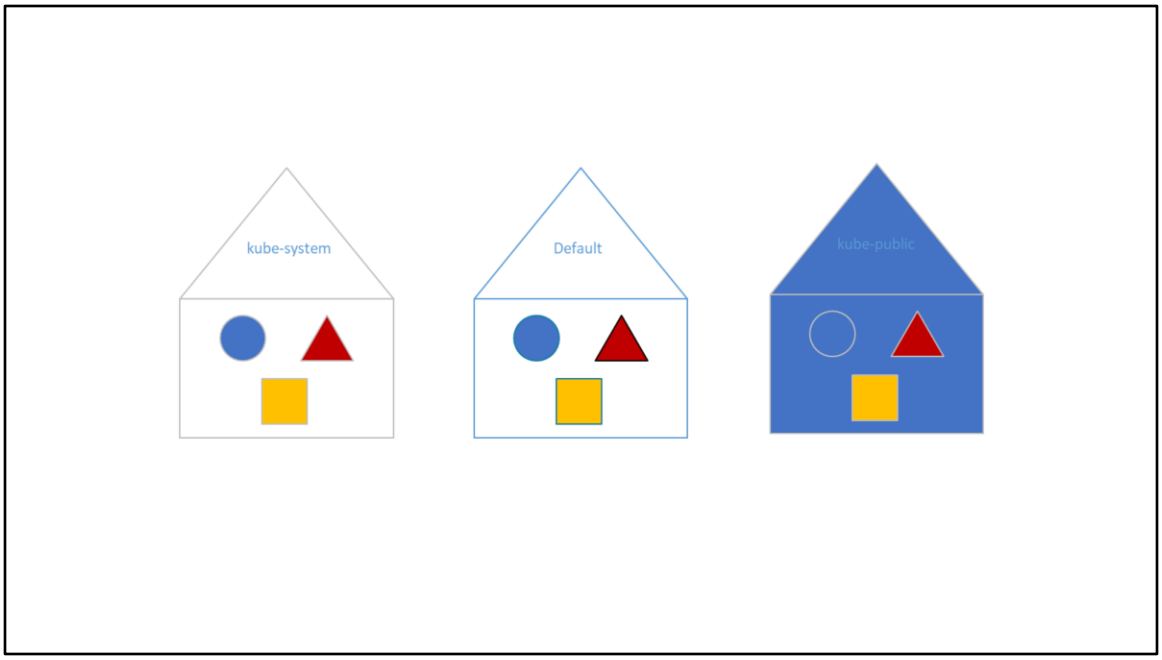
Let get back to Kubernetes. These houses correspond to namespaces in Kubernetes.



So far in this course we have created objects such as PODs, deployments and services in our cluster.

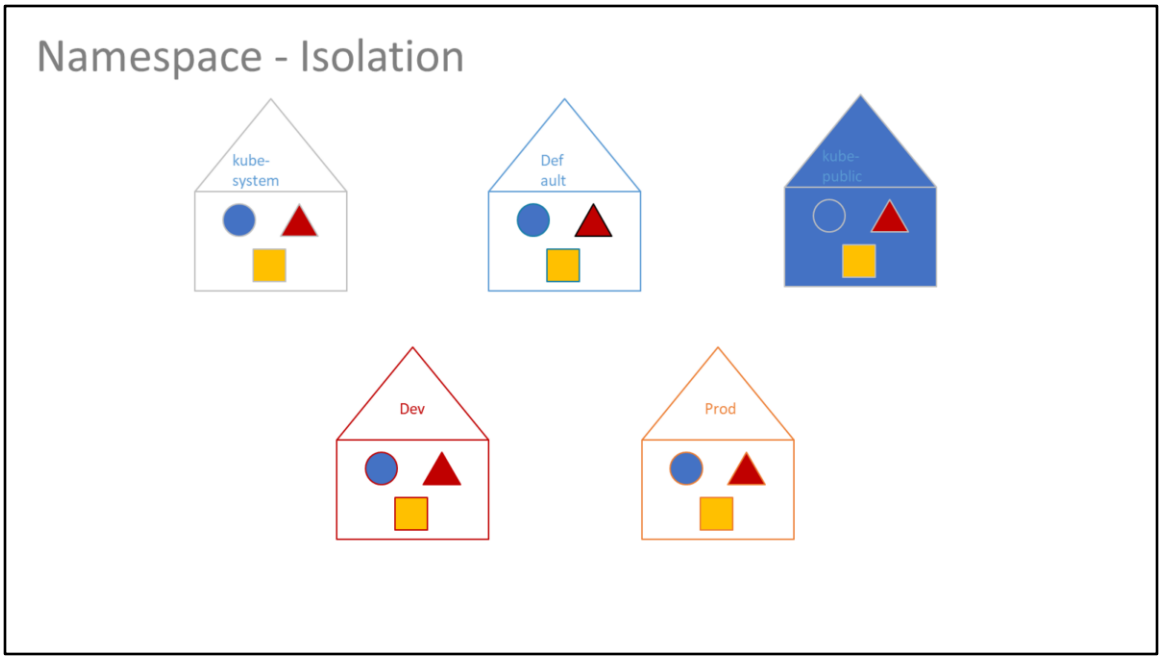


Whatever we have been doing, we have been doing it within a namespace. We were inside a house all this while. This namespace is known as the default namespace. And it is created automatically by kubernetes when the cluster is setup.

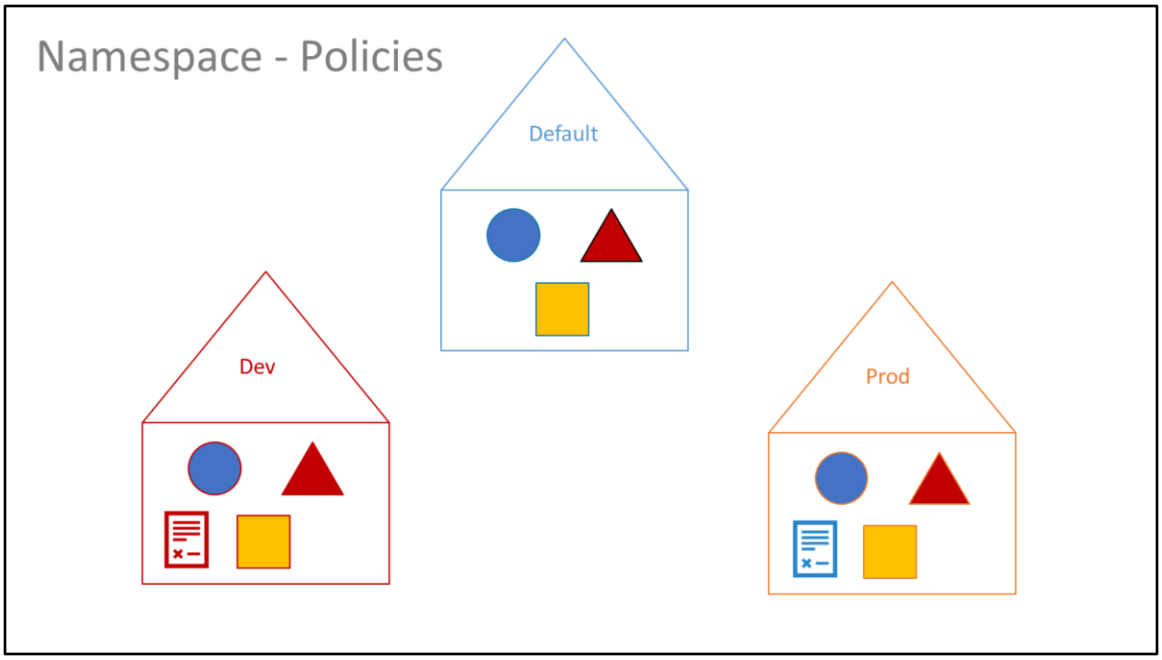


Kubernetes creates a set of PODs and services for its internal purpose such as those required by the Networking solution, the DNS service etc. To isolate these from the user and to prevent you from accidentally deleting or modifying these services, kubernetes creates them under another namespace created at cluster setup, named kube-system. A third namespace created by kubernetes automatically is called kube-public. This is where resources that should be made available to all users are created.

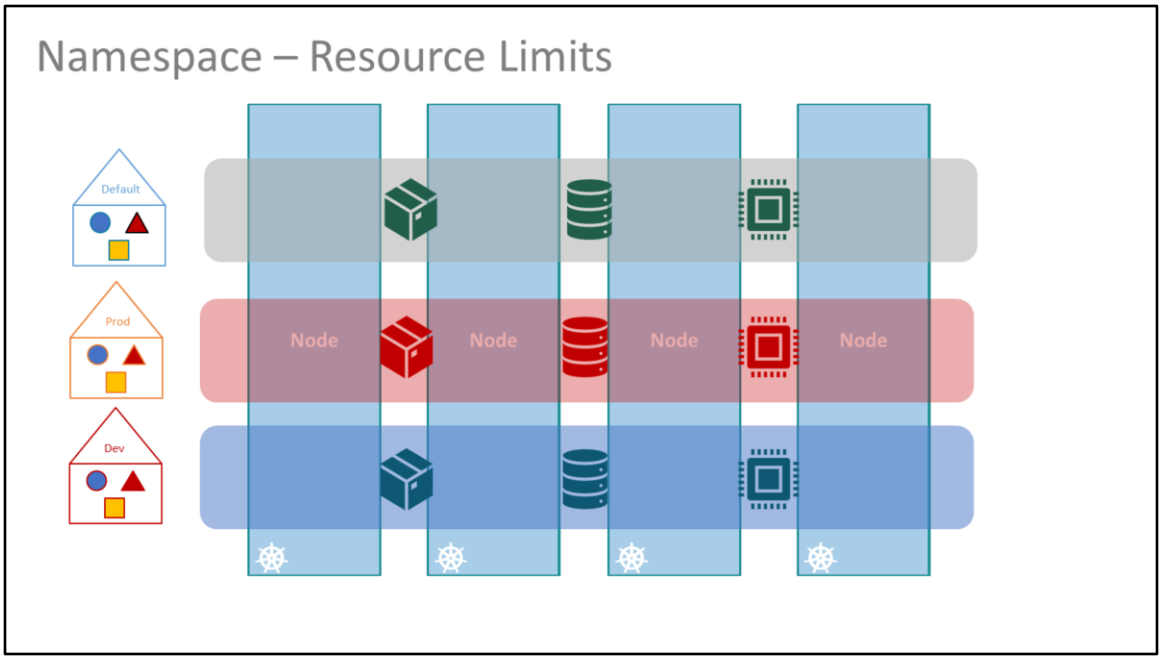
If your environment is small or you are learning and playing around with a small cluster, you shouldn't really have to worry about namespaces. You could continue to work in the default namespace. However when you grow and use a kubernetes cluster for enterprise purposes, you may want to consider the use of namespaces.



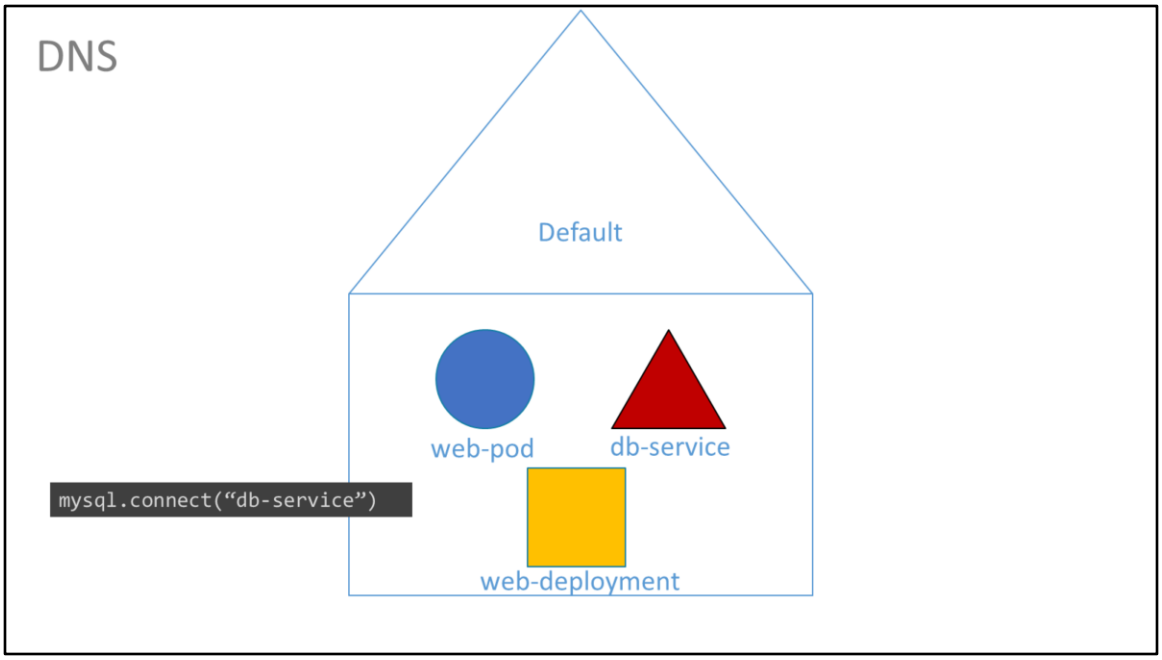
You can create your own namespaces as well. For example, if you wanted to use the same cluster for both dev and production environment, but at the same time isolate the resources between them, you can create a different namespace for each of them. That way while working in the Dev Environment you don't accidentally modify resources in production.



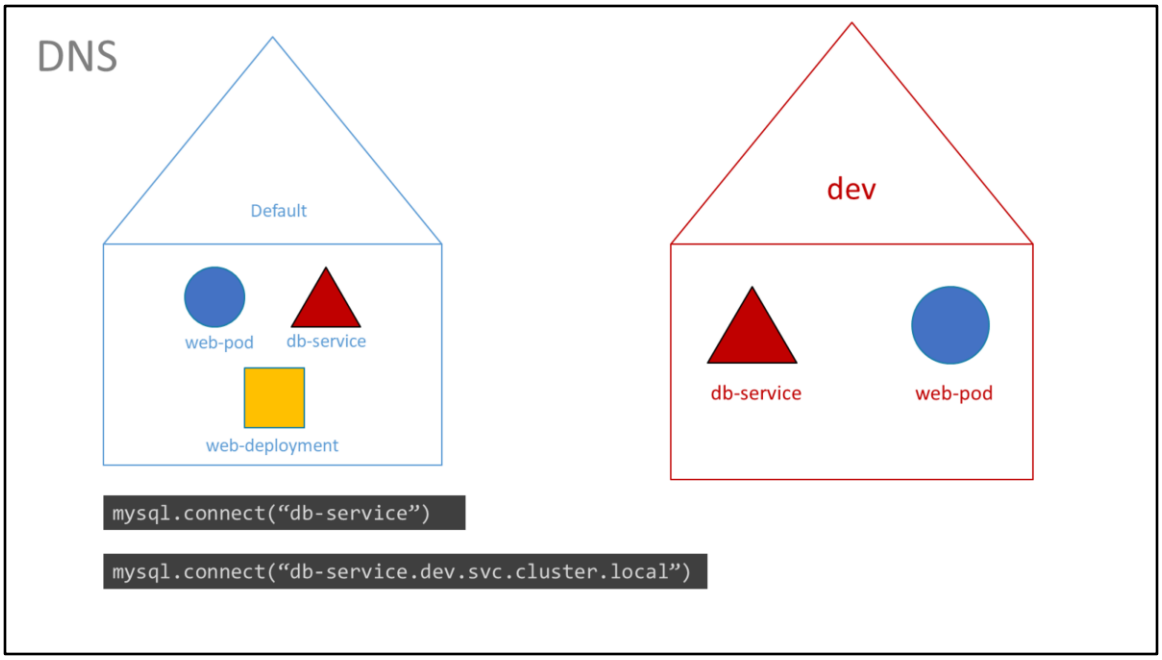
Each of these namespaces can have its own set of policies that defines who can do what.



You can also assign quota of resources to each of these namespaces, that way each namespace is guaranteed a certain amount and does not use more than its allowed limit.



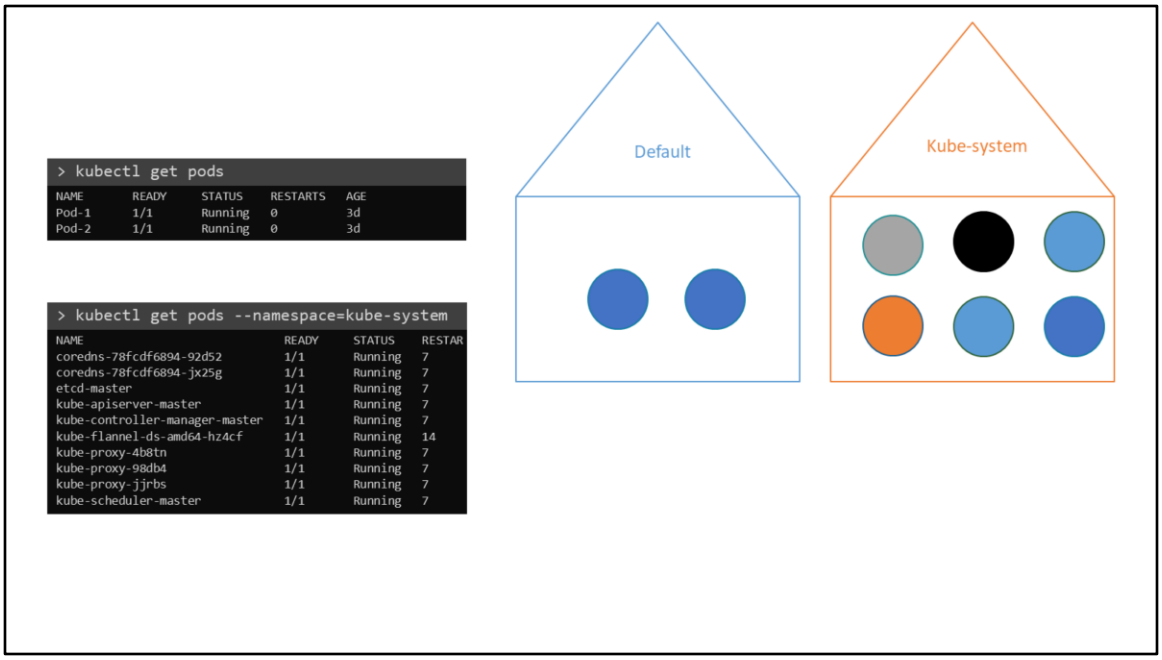
Going back to the default namespace that we have been working on. Just like how the members within a house refer to each other by their first names, the resources within a namespace can refer to each other simply by their names. In this case the webapp-pod can reach the db-service simply using the hostname "db-service".



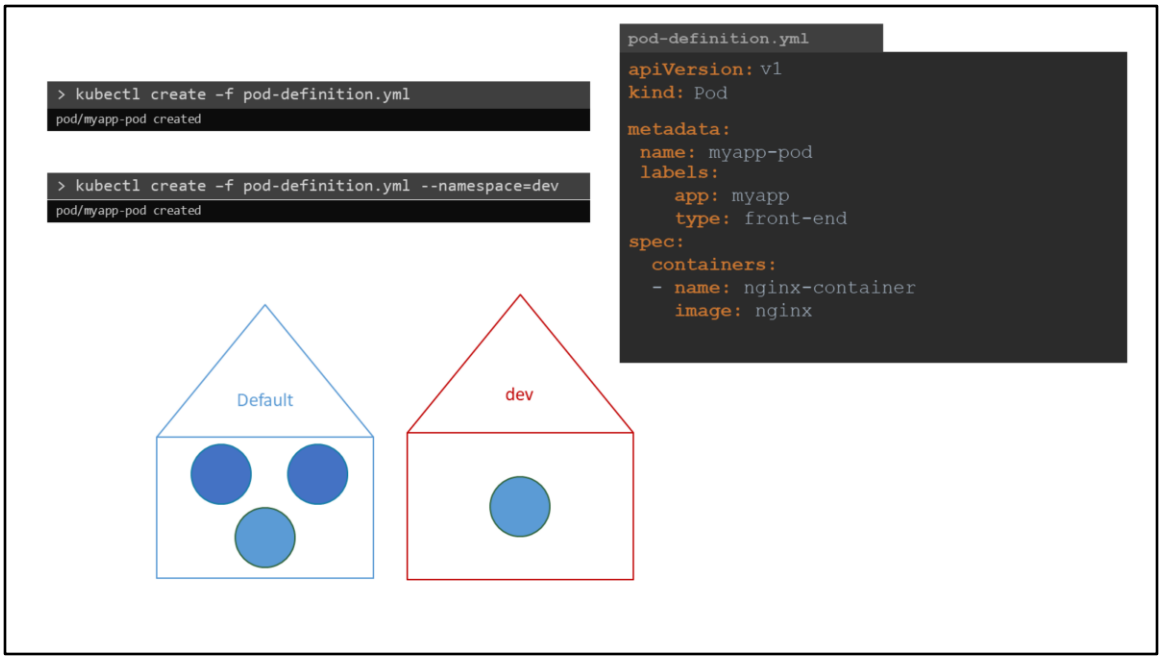
If required, the web-pod can reach a service in another namespace as well. For this you must append the name of the namespace to the name of the service as well. For example, for the web-pod in the default namespace to connect to the database in the Dev environment or namespace, use the servicename.namespace name. svc.cluster.local. That would be db-service.dev.svc.cluster.local. You are able to do this because when the service is created a DNS entry is added automatically.

DNS

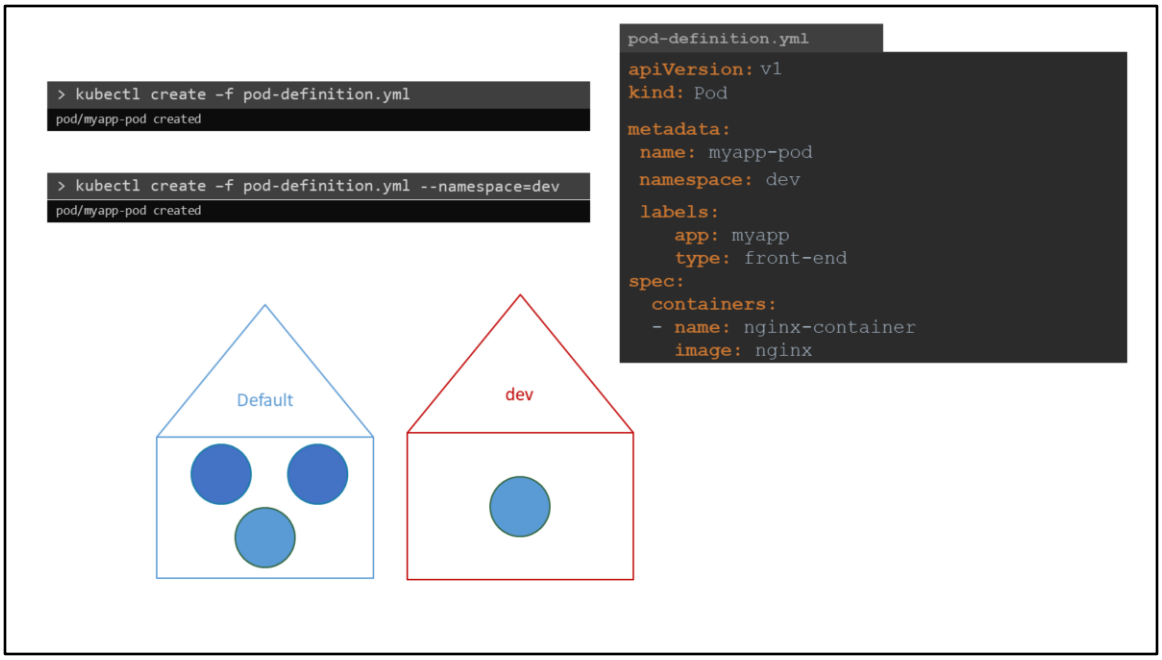
Looking closely at the dns name of the service, the last part – cluster.local – is the default domain name of the kubernetes cluster. Svc is the subdomain for service. Followed by the namespace and then the name of the service itself.



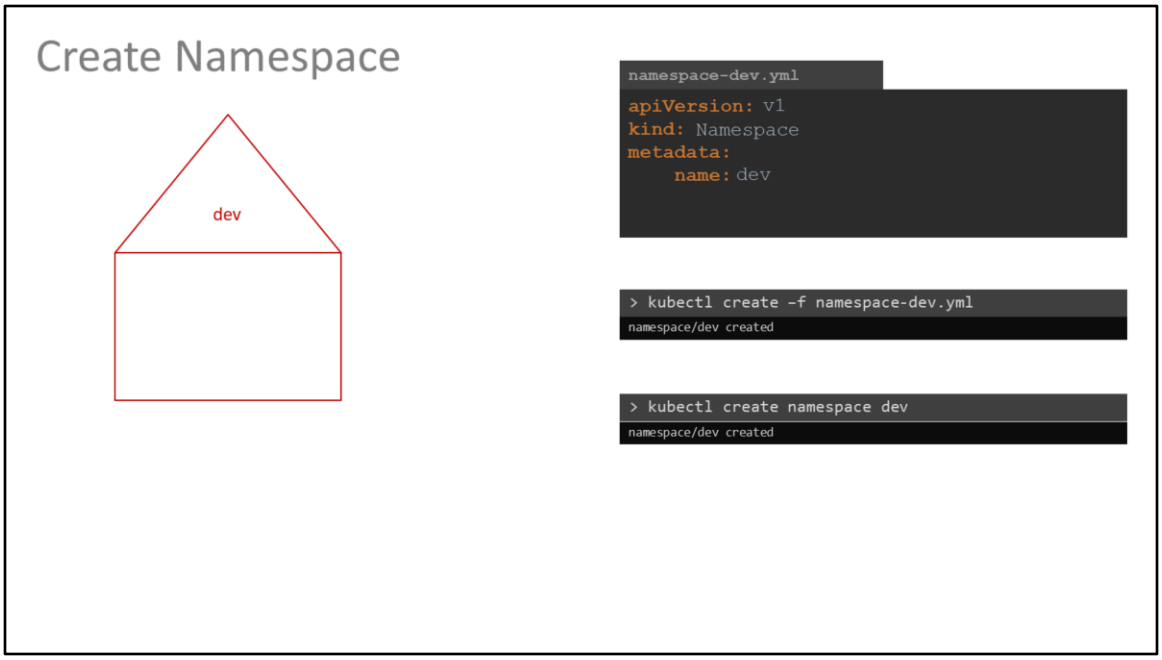
Let us now look at some of the operational aspects of Namespaces. Let us start with the kubectl commands. For example this command is used to list all the pods. But it only lists the pods in the default namespace. To list pods in another namespace use the namespace option in the command along with the name of the namespace - in this case kube-system.



Here I have a pod definition file. When you create a POD using this file, the pod is created in the default namespace. To create the pod in another namespace use the namespace option.

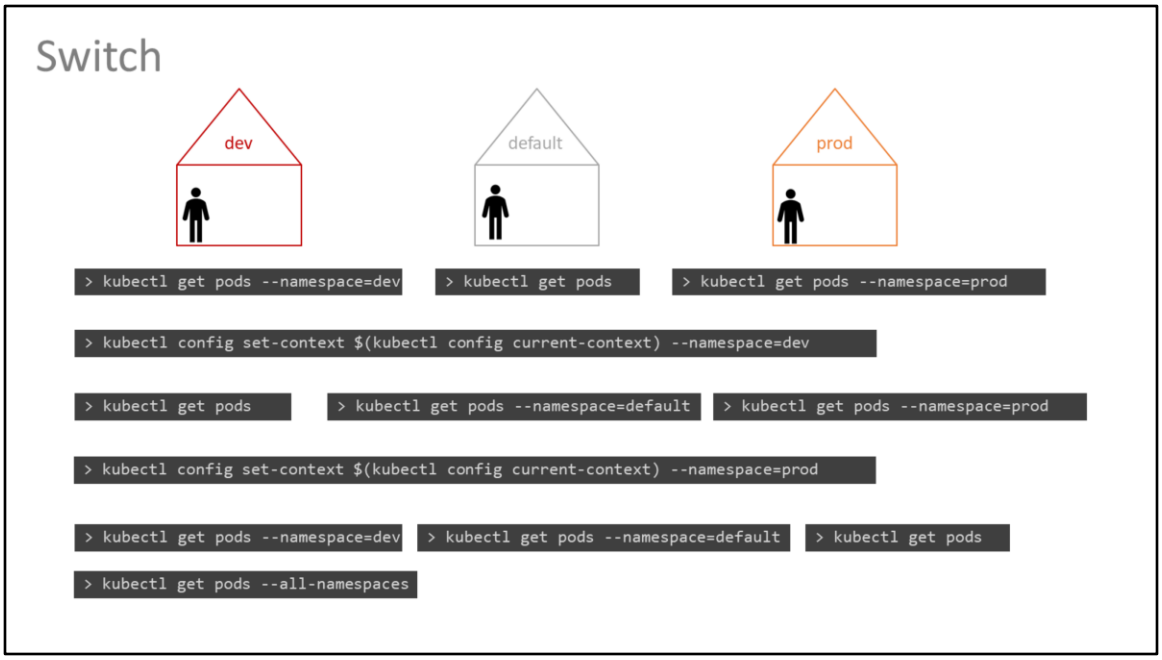


If you want to make sure that this pod gets created in the dev environment all the time, even if you don't specify the namespace in the command line, you can move the namespace definition into the pod definition file, like this under the metadata section. This is a good way to ensure your resources are always created in the same namespace.



So how do you create a new namespace? Like any other object, use a namespace definition file. The api Version is v1, kind is Namespace and under metadata, specify the name. In this case dev. Run the kubectl create command to create the namespace.

Another way to create a namespace is by simply running the command kubectl create namespace.

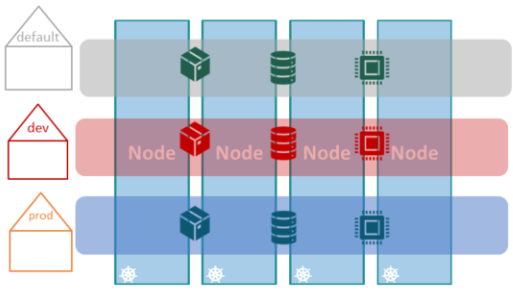


Now, say we are working in 3 namespaces. As we discussed before, by default we are in the default namespace. Which is why we can see the resources inside the default namespace using the kubectl get pods command and to view those in the dev namespace we have to use the namespace option. But what if we want to switch to the dev namespace permanently? So that we don't have to specify the namespace option anymore? Well in that case use the kubectl config command to set the namespace in the current context to dev. You can then simply run kubectl get pods command without the namespace option to list pods in the dev environment. But you will need to specify the option for other environments such as default or prod. Similarly you can switch to the prod namespace the same way.

```
> kubectl config set-context $(kubectl config current-context) --namespace=dev
```

Taking a closer look at the command, this command first identifies the current context and then sets the namespace to the desired one. Well, contexts are used to manage multiple clusters and multiple environments from the same management system. It is a totally separate topic to discuss and requires its own lecture. So we will discuss contexts in another lecture.

Resource Quota



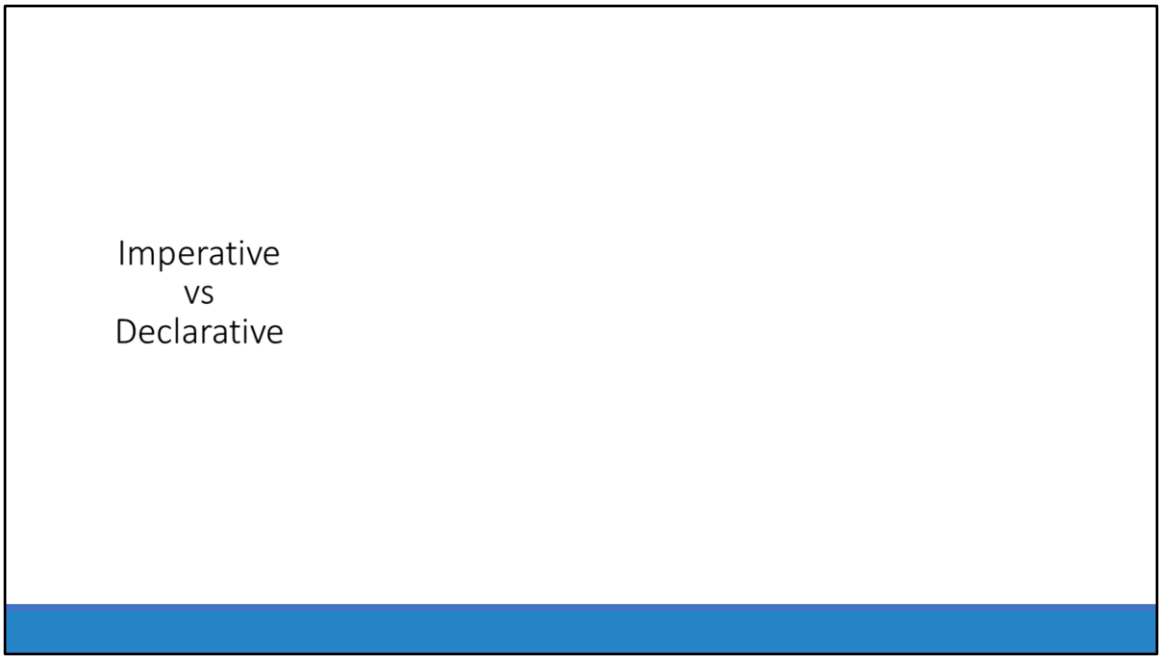
Compute-quota.yaml

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
    name: compute-quota
    namespace: dev

spec:
  hard:
    pods: "10"
    requests.cpu: "4"
    requests.memory: 5Gi
    limits.cpu: "10"
    limits.memory: 10Gi
```

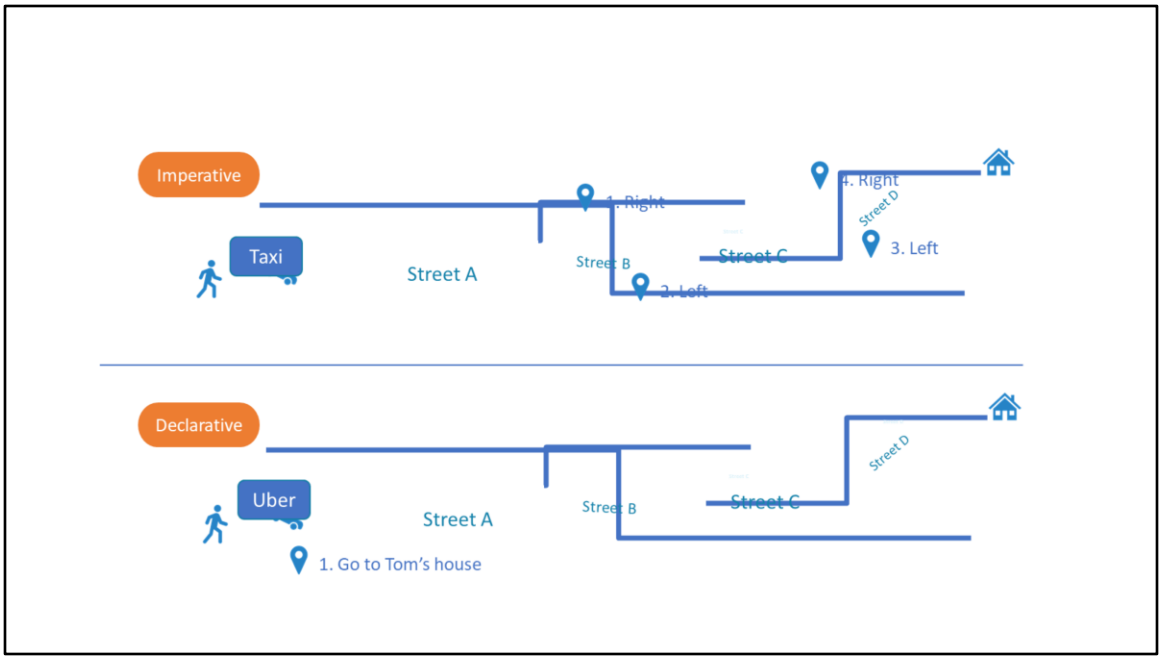
```
> kubectl create -f compute-quota.yaml
```

To limit resources in a namespace, create a resource quota. To create one start with a definition file for ResouceQuota. Specify the namespace for which you want to create the quota. And then under spec provide your limits, such as 10 pods, 10 CPU units, 10 Gibibyte of memory etc.

Well that's it for this lecture. Head over to the coding exercises section and practice working with namespaces. I will see you in the next lecture.



In this lecture we will discuss about imperative and declarative approaches in Kubernetes. Towards the end of this lecture, we will talk about some tips that you can use in the exam.



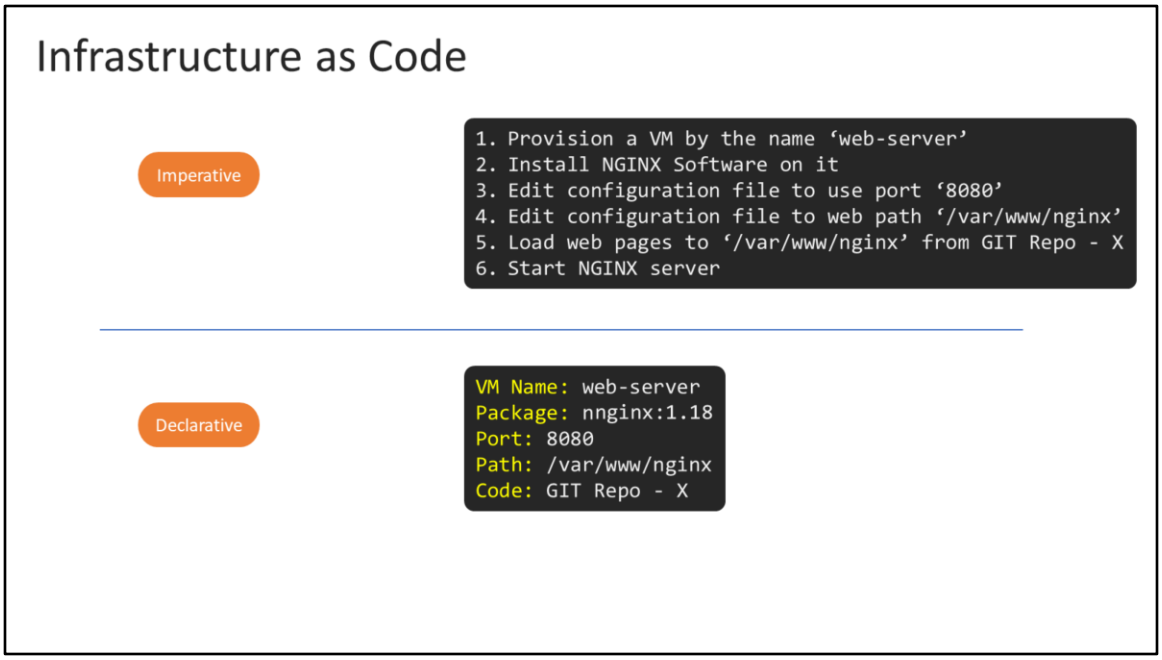
So far we have seen different ways of creating and managing objects in Kubernetes. We created objects directly by running commands, as well as using object configuration files.

In the infrastructure as code world there are different approaches in managing the infrastructure. And they are classified into imperative and declarative approaches.

Let's understand this with an analogy. Let's say, you want to visit a friend's house located at "Street D". In the past you would hire a taxi and give step by step directions to the driver on how to reach the destination - Like Take Right to StreetB, then take left to go to StreetC, and then take another left and then right to go to Street D and Stop at the house. "Specifying what to do and how to do" is the imperative approach.

On the other hand today when you book a cab, say through Uber, and just specify the final destination like "drive to Tom's house" – this is the Declarative approach. In this case we are not giving step by step directions, instead we are just specifying the Final destination. And the system figures out the right path to reach the destination. Specifying "what to do, not how to do" is the Declarative approach.

So what’s that got to do with what we are learning.



In the infrastructure as code world, an example of an imperative approach of provisioning infrastructure would be a set of instructions written step by step such as provisioning a VM named web-server, installing NGINX software on it, editing configuration file to use port 8080 and setting the path to the web files, and downloading source code of repositories from GIT. And finally starting the NGINX server. Here we are saying what is required and also how to get things done.

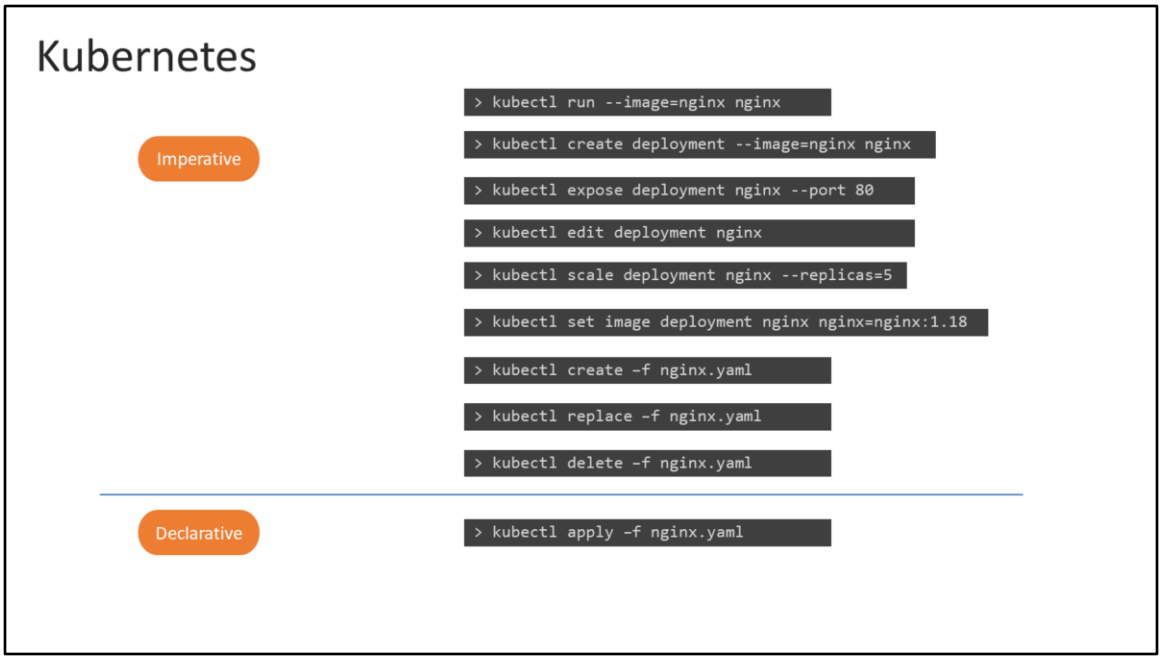
In the declarative approach we declare our requirements. For instance all we say is that we need a VM by the name web-server, with the nginx software, with the port set to 8080 and the path to the web files and where to download the code from. And everything that's needed to be done to get this infrastructure in place, is done by the system or the software. You don't have to provide step by step instructions. Orchestration tools such as Ansible, Puppet, Chef, Terraform fall into this category.

In the imperative approach, what happens if the first time only half of the steps were executed? What happens if you provide the same instructions again to complete the remaining steps? To handle such situations there would be many additional steps involved such as checks to see if something already exists and taking an action based on the results of that check. For instance while provisioning a vm what would happen

if a vm by the name web-server already exists? The same goes with creating a database, importing data etc. Should it fail or should it continue since the VM is already there.

What if we decide to upgrade the version of software – say to nginx-1.18 in the future? It should be as simple as updating the version of nginx in the configuration file and the system should take care of the rest.

Ideally the system should be intelligent enough to know what has already been done and apply the necessary changes only. That's the declarative way of doing things.



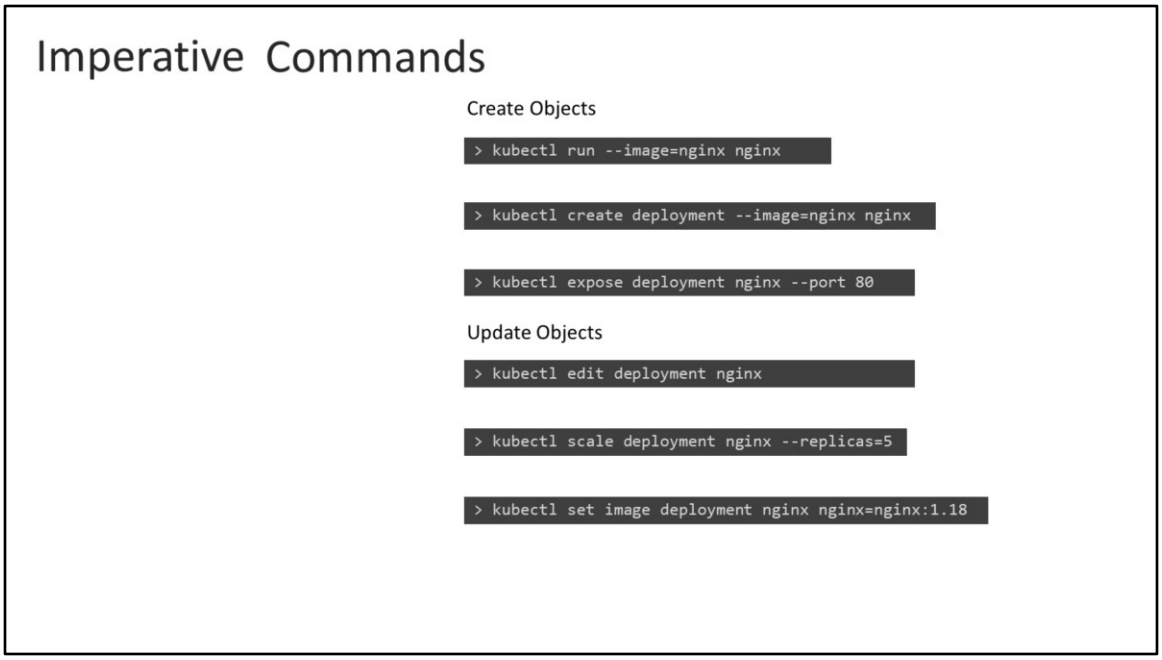
In the Kubernetes world the imperative way of managing infrastructure is using commands like kubectl run to create a pod, the kubectl create deployment command to create a deployment, kubectl expose command to create a service to expose a deployment. The kubectl edit command may be used to edit an existing object. For scaling a deployment or replicaset use the kubectl scale command, updating the image on a deployment use the kubectl set image command etc.

We have also used object configuration files to manage objects. Such as creating an object using the kubectl create –f command with the f option to specify the object configuration file and editing an object using the kubectl edit commands and deleting an object using the kubectl delete command. All of these are imperative approaches to managing objects in Kubernetes. We are saying exactly how to bring the infrastructure to our needs by creating, updating or deleting objects.

The declarative approach would be to create a set of files that defines the expected state of the applications and services on a Kubernetes cluster. And with a single kubectl apply command Kubernetes should be able to read the object configuration files and decide by itself what needs to be done to bring the infrastructure to the expected state. So in the declarative approach you will run the kubectl apply

command for creating updating or deleting an object. The apply command will look at the existing configuration and figure out what changes need to be made to the system.

So let’s look at these in a bit more detail.

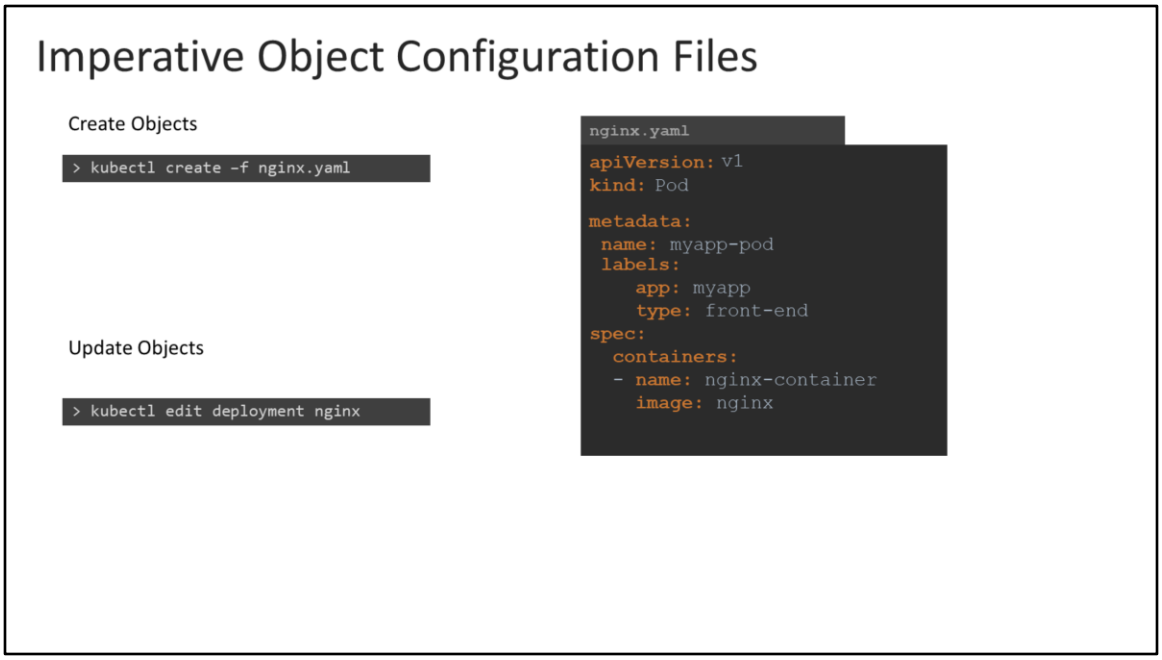


Within the imperative approach there are 2 ways. The first is using imperative commands such as the run, create, expose commands to create new objects. And the edit, scale, and set commands to update existing objects. These commands help in quickly creating or modifying objects as we don't have to deal with YAML files. And these are helpful during the certification exams.

However they are limited in functionality and will require forming long and complex commands for advanced use cases. Such as creating a multi container pod or deployment.

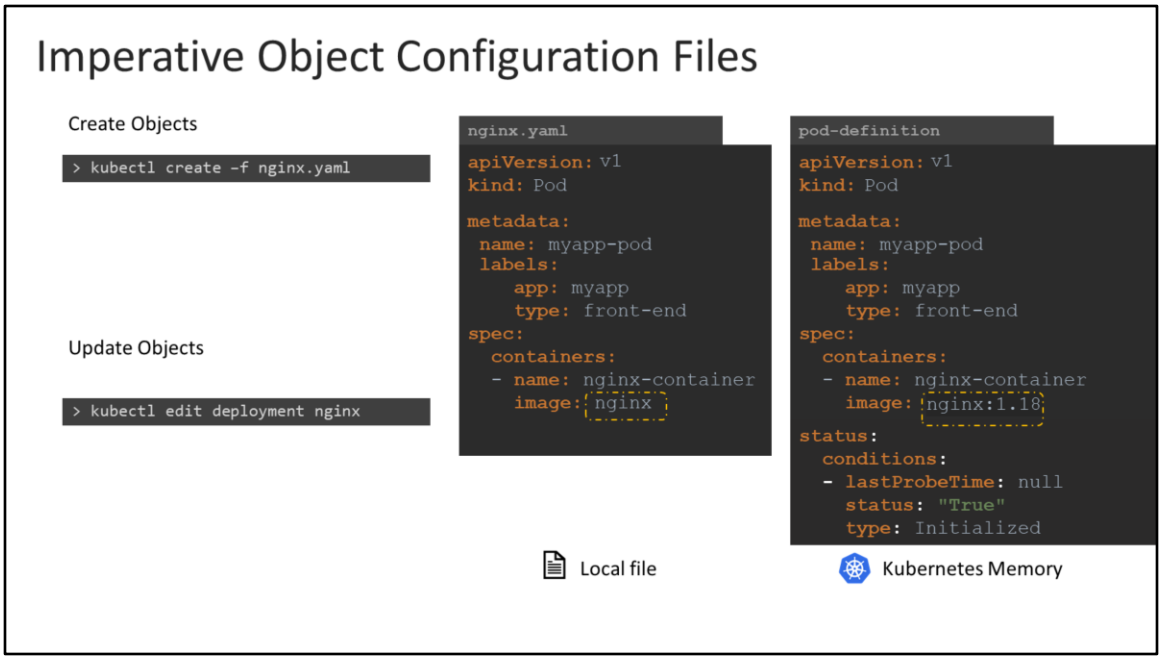
Secondly these commands are run once and forgotten. They are only available in the session history of the user who ran these commands. It's hard for another person to figure how these objects were created, so it is hard to keep track of. And so it's difficult to work with these commands in large or complex environments.

That's where managing objects with object configuration files can help.

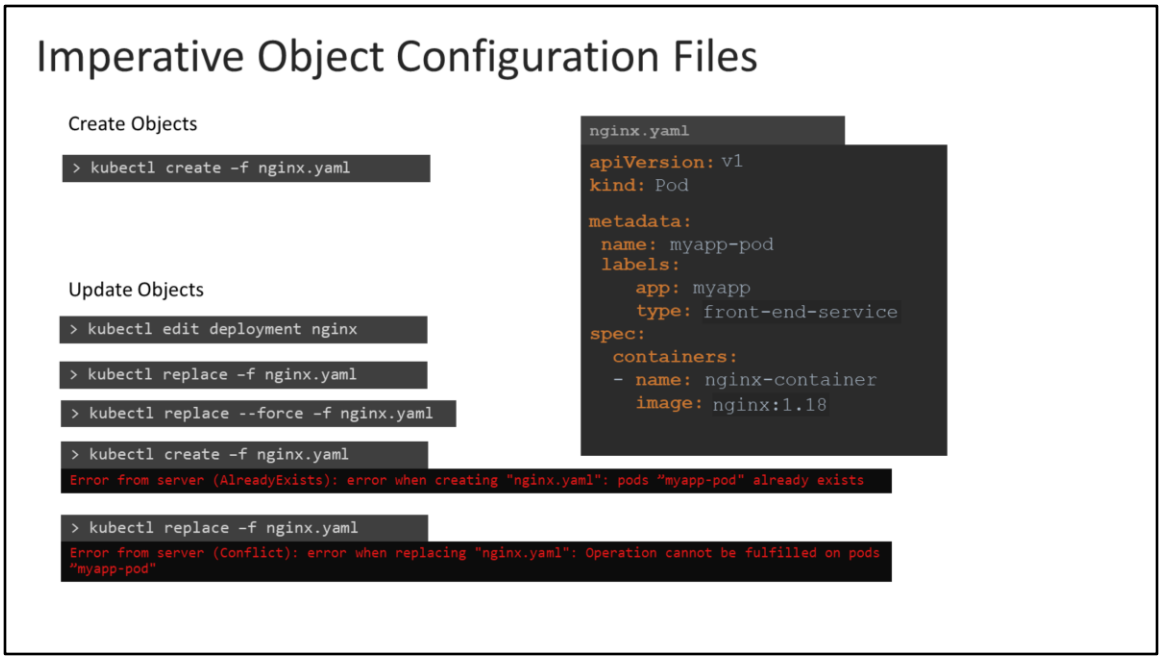


That's where managing objects with object configuration files can help. Creating object definition files or configuration files or manifest files as it's also called, can help us write down exactly what we need the object to look like in a YAML format and use the kubectl create command to create the object. We now have the YAML file with us always and it can be saved in a code repository like GIT. We can put together a change review and approval process around these files, so that a change made is reviewed and approved before it is applied to production.

In the future if a change is to be made, for instance editing the image name to another version, there are different ways to go about it. One way is to use the kubectl edit command and specify the object name. When this command is run, it opens a YAML definition file similar to the one you used to create the object, but with some additional fields.



it opens a YAML definition file similar to the one you used to create the object, but with some additional fields. This is not the file you used to create the object. This is a similar pod definition file within the Kubernetes memory. You can make changes to this file and save and quit, and those changes will be applied to the live object. However note that now there is a difference between the live object and the definition file you have locally. The change you made using the kubectl edit command is not really recorded anywhere. After the change is applied you are only left with your local definition file which in fact has the old image name in it.



In the future say you or a team mate decide to make a change to this object, unaware that a change was made using the kubectl edit command, when the new change is applied the previous change to the image is lost. So you can use the kubectl edit command if you are making a change and you are sure you are not going to rely on the object configuration file in the future.

A better approach to that is to first edit the local version of object configuration file with the required changes and then run the kubectl replace command to update the object. This way, going forward the changes made are recorded and can be tracked as part of the change review process.

At times you may want to completely delete and recreate objects. In such cases you may run the same command but with the force option.

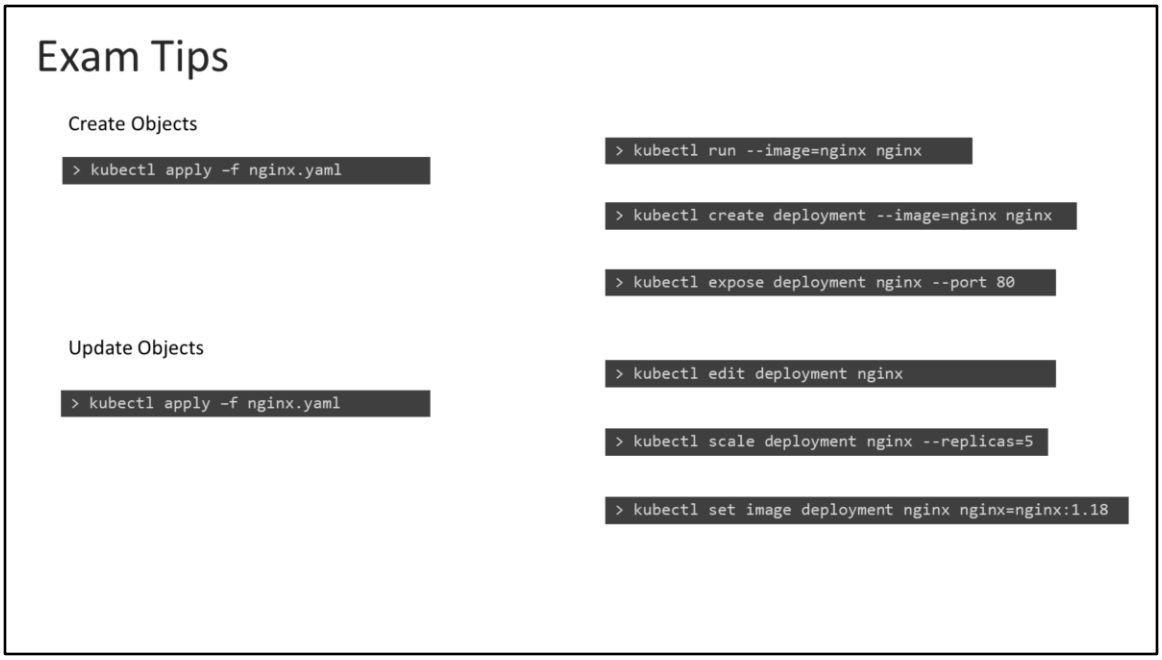
Now this is still the imperative approach because you are still instructing Kubernetes how to create or update these objects. First you run the kubectl create command to create the object. And then you run the replace command to replace or the delete command to delete the object.

What if you run the create command if the object already exists? Then it would fail with an error that says the pod already exists. When you update an object, you should always make sure it exists first before running the replace command. If an object doesn't exist the replace command fails with an error message. So the imperative approach is very taxing for you as an administrator as you must always be aware of the current configurations and perform checks to make sure things are in place before making a change.

The declarative approach is where you use the same object configuration files, but use the kubectl apply command to manage objects. The kubectl apply command is intelligent enough to create an object if it doesn't already exist. If there are multiple object configuration files as you would usually, then you may specify a directory as the path instead of a single file. That way all the objects are created at once.

When changes are to be made, simply update the object configuration file and run the kubectl apply command again. This time it knows that the object exists, and so only updates the object with the new changes.

Going forward for any changes made on the application, whether they are updating images or adding new configuration files, simply update your local repository with the changes and run the kubectl apply command. We will discuss more about how the kubectl apply command works in the next lecture.



From an exam perspective you could use the imperative approach to save time as much as possible. For example if the question is to just create a pod or a deployment with a given image, then one of these imperative commands can help you achieve that quickly.

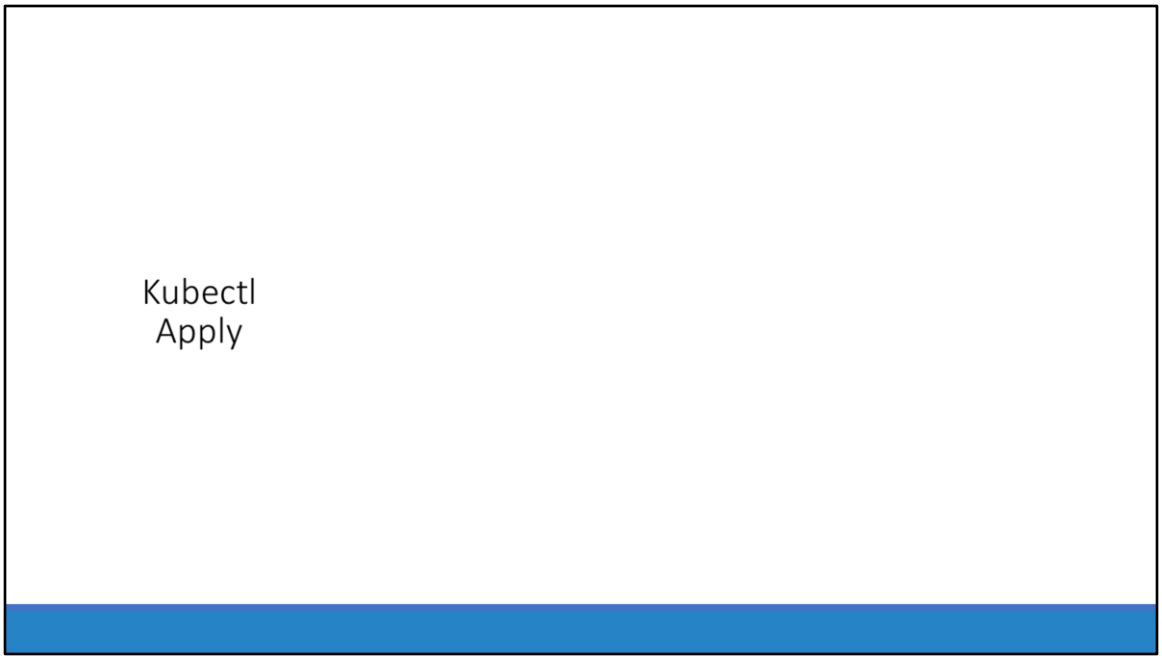
If you need to edit a property of an object then using the kubectl edit command may be the quickest way.

If you have a complex requirement with multiple containers, environment variables, commands, initContainers etc, then using an object configuration file to create the object would be preferred. This way if you see that you made a mistake, you can easily update the file and apply it again.

For more details on the different approaches to managing a Kubernetes cluster, get yourself familiarized with the Kubernetes documentation pages.

Labs

In the upcoming lab exercises try to use imperative approach in solving questions.
And I will see you in the next lecture.



In this lecture we will understand more about how the kubectl apply command works.

Declarative

Create Objects

```
> kubectl apply -f nginx.yaml
```

```
> kubectl apply -f /path/to/config-files
```

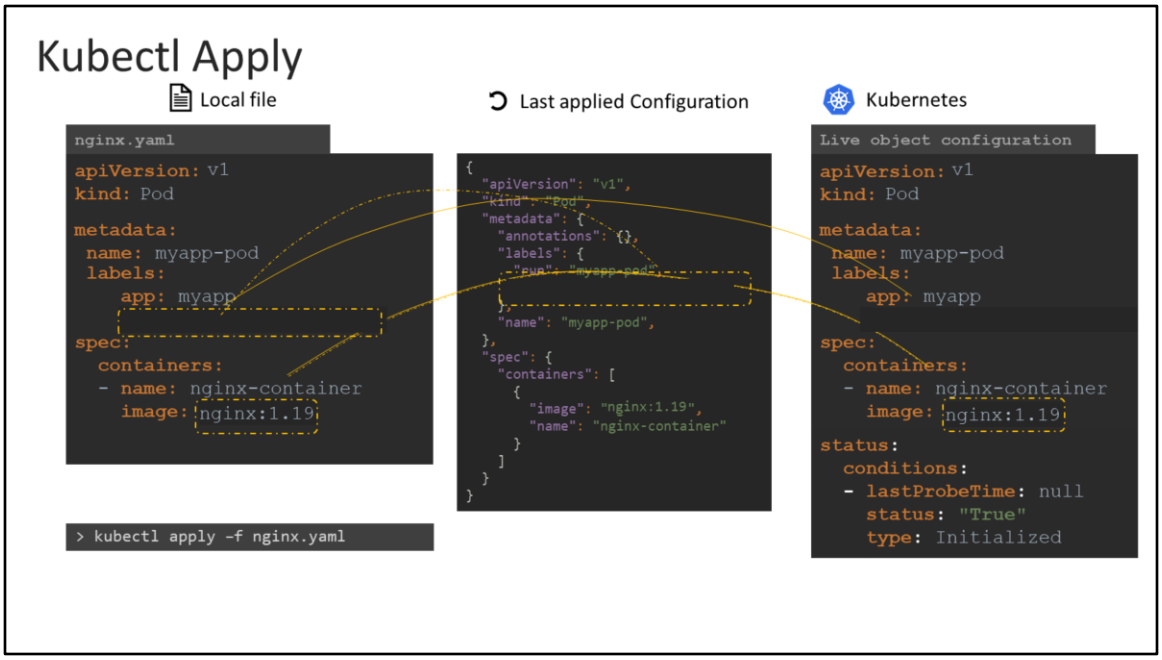
Update Objects

```
> kubectl apply -f nginx.yaml
```

nginx.yaml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end-service
spec:
  containers:
  - name: nginx-container
    image: nginx:1.18
```

In the previous lecture we saw how a kubectl apply command can be used to manage objects in a declarative way. In this lecture we wil see a bit more about how the apply command works internally.



The apply command takes into consideration the local configuration file, the live object definition on Kubernetes, and the last applied configuration before making a decision on what changes are to be made.

When you run the apply command, if the object does not already exist, the object is created. When the object is created an object configuration similar to what we created locally is created within Kubernetes but with additional fields to store status of the object. This is the live configuration of the object on the Kubernetes cluster. This is how Kubernetes internally stores information about an object no matter what approach you use to create the object.

But when you use the kubectl apply command to create an object it does something more. The YAML version of the local object configuration file we wrote is converted to a JSON format and stored as the last applied configuration.

Going forward for any updates to the object, all the three are compared to identify what changes are to be made on the live object.

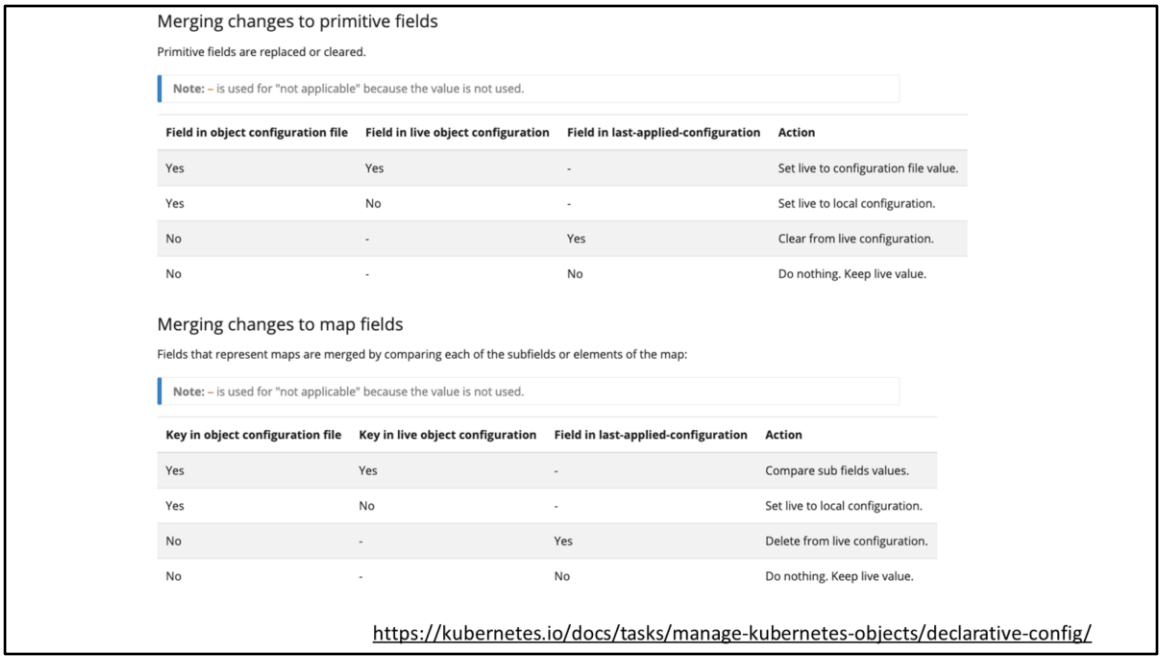
For example when the nginx image is updated to 1.18 in our local file and we run the

kubectl apply command, this value is compared with the value in the live configuration and if there is a difference the live configuration is updated with the new value. After any change the last applied json format is always updated to the latest.

<c>

So why do we really need the last applied configuration. If a field was deleted, say for example the type label was deleted. Now when we run the kubectl apply command, we see that the last applied configuration had a label type but it's not present in the live configuration. This means that the field needs to be removed from the live configuration. If a field was present in the live configuration and not present in the local or last applied configuration then it is left as is.

But where is this JSON object stored?



Merging changes to primitive fields

Primitive fields are replaced or cleared.

> **Note:** – is used for "not applicable" because the value is not used.

Field in object configuration file	Field in live object configuration	Field in last-applied-configuration	Action
Yes	Yes	-	Set live to configuration file value.
Yes	No	-	Set live to local configuration.
No	-	Yes	Clear from live configuration.
No	-	No	Do nothing. Keep live value.

Merging changes to map fields

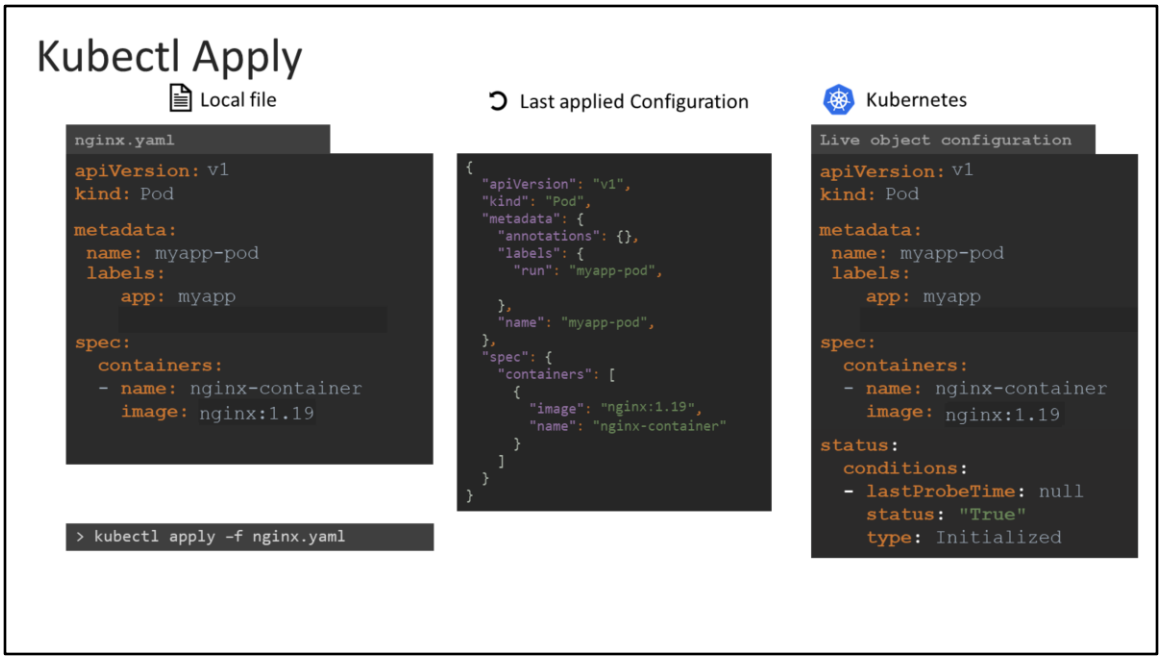
Fields that represent maps are merged by comparing each of the subfields or elements of the map:

> **Note:** – is used for "not applicable" because the value is not used.

Key in object configuration file	Key in live object configuration	Field in last-applied-configuration	Action
Yes	Yes	-	Compare sub fields values.
Yes	No	-	Set live to local configuration.
No	-	Yes	Delete from live configuration.
No	-	No	Do nothing. Keep live value.

https://kubernetes.io/docs/tasks/manage-kubernetes-objects/declarative-config/

What we just discussed is available for your reference in detail in the Kubernetes documentation pages.



So we saw the 3 sets of files – and we know that the local file is what's stored on our local system and the live object configuration is in the Kubernetes memory.

But where is this JSON file that has the last applied configuration stored?

It's stored on the live object configuration on the cluster as an annotation named last-applied-configuration. Remember that this is only done when you use the apply command. The kubectl create or replace commands do not store the last applied configuration like this. So you must bear in mind not to mix imperative and declarative approaches.

Once you use the apply command, going forward whenever a change is made the apply command compares all 3 sections – the local pod definition file, the live object configuration and the last applied configuration stored within the live object configuration file for deciding what changes are to be made to the live configuration.

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