Our focus in this lecture

is more on Kubernetes-related security.

What are the risks and what measures do you need to take

to secure the cluster?

Cube API server

is at the center of all operations within Kubernetes.

We interact with it through the kube control utility

or by accessing the API directly.

And through that, you can perform almost any operation

on the cluster.

So that's the first line of defense,

controlling access to the API server itself.

We need to make two types of decisions.

Who can access the cluster, and what can they do?

Who can access the API server

is defined by the authentication mechanisms.

There are different ways that you can authenticate

to the API server, starting with user IDs

and passwords stored in static files or tokens,

certificates, or even integration

with external authentication providers like LDAP.

Finally, for machines, we create service accounts.

We will look at these in more detail

in the upcoming lectures.

Once they gain access to the cluster,

what can they do is defined by authorization mechanisms.

Authorization is implemented using role-based access

controls where users are associated to groups

with specific permissions.

In addition, there are other authorization modules

like the attributed-based access control,

node authorizers, web pods, et cetera.

Again, we look at these in more detail

in the upcoming lectures.

All communication with the cluster

between the various components such as the etcd cluster,

the kube controller, manager scheduler, API server,

as well as those running on the worker nodes

such as the kubelet and the kube-proxy

is secured using TLS encryption.

We have a section entirely for this where we discuss

and practice how to set up certificates

between the various components.

What about communication

between applications within the cluster?

By default, all ports can access all other ports

within the cluster.

Now, you can restrict access between them

using network policies.

Instructor: Hello, and welcome to this lecture

on authentication in a Kubernetes cluster.

As we have seen already, the Kubernetes cluster consists

of multiple nodes, physical or virtual,

and various components that work together.

We have users like administrators

that access the cluster to perform administrative tasks,

the developers that access the cluster to test

or deploy applications,

we have end users who access the applications deployed

on the cluster,

and we have third party applications accessing the cluster

for integration purposes.

Throughout this section,

we will discuss how to secure our cluster

by securing the communication between internal components

and securing management access to the cluster

through authentication and authorization mechanisms.

In this lecture, our focus is on securing access

to the Kubernetes cluster with authentication mechanisms.

So we talked about the different users

that may be accessing the cluster.

Security of end users who access the applications deployed

on the cluster is managed

by the applications themselves internally.

So we will take them out of our discussion.

Our focus is on users access to the Kubernetes cluster

for administrative purposes.

So we are left with two types of users.

Humans such as the administrators and developers and robots

such as other processes or services

or applications that require access to the cluster.

Kubernetes does not manage user accounts natively.

It relies on an external source like a file

with user details or certificates

or a third party identity service like LDAP

to manage these users.

And so you cannot create users in a Kubernetes cluster

or view the list of users like this.

However, in case of service accounts,

Kubernetes can manage them.

You can create and manage service accounts using

the Kubernetes API.

We have a section on service accounts exclusively

where we discuss and practice more about service accounts.

For this lecture, we will focus on users in Kubernetes.

All user access is managed by the API server

whether you're accessing the cluster

through kube control tool or the API directly,

all of these requests go through the kube-apiserver.

The kube-apiserver authenticates the request

before processing it.

So how does the kube-apiserver authenticate?

There are different authentication mechanisms

that can be configured.

You can have a list of username and passwords

in a static password file,

or usernames and tokens in a static token file,

or you can authenticate using certificates.

And another option is to connect

to third party authentication protocols like LDAP,

Kerberos, et cetera.

We will look at some of these next.

Let's start with static password and token files

as it is the easiest to understand.

Let's start with the simplest form of authentication.

You can create a list of users

and their passwords in a CSV file

and use that as the source for user information.

The file has three columns: password, username, and user id.

We then pass the file name as an option

to the kube-apiserver.

Remember the kube-apiserver service

and the various options we looked at earlier in this course,

that is where you must specify this option.

You must then restart the kube-apiserver

for these options to take effect.

If you set up your cluster using the kubeadm tool,

then you must modify the kube-apiserver pod definition file.

The kubeadm tool will automatically restart

the kube-apiserver once you update this file.

To authenticate using the basic credentials

while accessing the API server,

specify the user and password in a curl command like this.

In the CSV file with the user details that we saw,

we can option optionally have a fourth column

with the group details to assign users to specific groups.

Similarly, instead of a static password file,

you can have a static token file.

Here, instead of password, you specify a token.

Pass the token file as an option token auth file

to the kube-apiserver.

While authenticating, specify the token

as an authorization barrier token to your request like this.

That's it for this lecture.

Remember that this authentication mechanism

that stores usernames, passwords, and tokens in clear text

in a static file is not a recommended approach

as it is insecure.

But I thought this was the easiest way to understand

the basics of authentication in Kubernetes.

Going forward, we will look

at other authentication mechanisms.

I also wanna point out that if you are trying this out

in a kubeadm setup, you must also consider volume mounts

to passing the auth file.

Details about these are available

in the article that follows,

and remember to set up authorization for the new users.

We will discuss about authorization later in this course.

**Article on Setting up Basic Authentication**

Setup basic authentication on Kubernetes (Deprecated in 1.19)

Note: This is not recommended in a production environment. This is only for learning purposes. Also note that this approach is deprecated in Kubernetes version 1.19 and is no longer available in later releases

Follow the below instructions to configure basic authentication in a kubeadm setup.

Create a file with user details locally at /tmp/users/user-details.csv

1. # User File Contents
2. password123,user1,u0001
3. password123,user2,u0002
4. password123,user3,u0003
5. password123,user4,u0004
6. password123,user5,u0005

Edit the kube-apiserver static pod configured by kubeadm to pass in the user details. The file is located at /etc/kubernetes/manifests/kube-apiserver.yaml

1. apiVersion: v1
2. kind: Pod
3. metadata:
4. name: kube-apiserver
5. namespace: kube-system
6. spec:
7. containers:
8. - command:
9. - kube-apiserver
10. <content-hidden>
11. image: k8s.gcr.io/kube-apiserver-amd64:v1.11.3
12. name: kube-apiserver
13. volumeMounts:
14. - mountPath: /tmp/users
15. name: usr-details
16. readOnly: true
17. volumes:
18. - hostPath:
19. path: /tmp/users
20. type: DirectoryOrCreate
21. name: usr-details

Modify the kube-apiserver startup options to include the basic-auth file

1. apiVersion: v1
2. kind: Pod
3. metadata:
4. creationTimestamp: null
5. name: kube-apiserver
6. namespace: kube-system
7. spec:
8. containers:
9. - command:
10. - kube-apiserver
11. - --authorization-mode=Node,RBAC
12. <content-hidden>
13. - --basic-auth-file=/tmp/users/user-details.csv

Create the necessary roles and role bindings for these users:

1. ---
2. kind: Role
3. apiVersion: rbac.authorization.k8s.io/v1
4. metadata:
5. namespace: default
6. name: pod-reader
7. rules:
8. - apiGroups: [""] # "" indicates the core API group
9. resources: ["pods"]
10. verbs: ["get", "watch", "list"]
12. ---
13. # This role binding allows "jane" to read pods in the "default" namespace.
14. kind: RoleBinding
15. apiVersion: rbac.authorization.k8s.io/v1
16. metadata:
17. name: read-pods
18. namespace: default
19. subjects:
20. - kind: User
21. name: user1 # Name is case sensitive
22. apiGroup: rbac.authorization.k8s.io
23. roleRef:
24. kind: Role #this must be Role or ClusterRole
25. name: pod-reader # this must match the name of the Role or ClusterRole you wish to bind to
26. apiGroup: rbac.authorization.k8s.io

Once created, you may authenticate into the kube-api server using the users credentials

curl -v -k https://localhost:6443/api/v1/pods -u "user1:password123"

Presenter: Hello and welcome to this lecture.

In this lecture, we look at kubeconfigs in Kubernetes.

So far, we have seen how to generate a certificate

for a user.

We have seen how a client uses the certificate file

and key to query the Kubernetes REST API,

for a list of pods using curl.

In this case, my cluster is called my-kube-playground.

So, send a curl request

to the address of the kube API server,

while passing in the pair of files,

along with the CA certificate,

as options.

This is then validated by the API server

to authenticate the user.

Now, how do you do that

while using the kubectl command?

You can specify the same information

using the options,

server, client key, client certificate

and certificate authority with the kubectl utility.

Obviously, typing those in every time is a tedious task.

So, you move these information

to a configuration file called as kubeconfig,

and then specify this file

as the kubeconfig option in your command.

By default, the kubectl tool looks for a file

named config under a directory, .kube,

under the user's home directory.

So, if you create the kubeconfig file there,

you don't have to specify the path

to the file explicitly in the kubectl command.

That's the reason you haven't been specifying any options

for your kubectl commands so far.

The kubeconfig file is in a specific format.

Let's take a look at that.

The config file has three sections,

clusters, users and contexts.

Clusters are the various Kubernetes clusters

that you need access to.

Say you have multiple clusters for development environment

or testing environment, or prod,

or for different organizations,

or on different cloud providers, et cetera.

All those go there.

Users are the user accounts

with which you have access to these clusters.

For example, the admin user,

a dev user, a prod user, et cetera.

These users may have different privileges

on different clusters.

Finally, contexts marry these together.

Contexts define which user account will be used

to access which cluster.

For example, you could create a context

named admin@production that will use the admin account

to access a production cluster.

Or, I may want to access the cluster I have set up on Google

with the dev user's credentials,

to test deploying the application I built.

Remember, you're not creating any new users,

or configuring any kind of user access

or authorization in the cluster with this process.

You're using existing users

with their existing privileges,

and defining what user you're going to use

to access what cluster.

That way, you don't have to specify the user certificates

and server address in each and every

kubectl command you run.

So, how does it fit into our example?

The server specification in our command

goes into the clusters section.

The admin user's keys and certificates

goes into the users section.

You then create a context that specifies

to use the my-kube-admin user

to access the my-kube-playground cluster.

Let's look at a real kubeconfig file now.

The kubeconfig file is in a YAML format.

It has API version set to V1.

The kind is config,

and then it has three sections, as we discussed.

One for clusters, one for contexts, and one for users.

Each of these is in an array format.

That way, you can specify multiple clusters,

users or contexts within the same file.

Under clusters, we add a new item

for our kube-playground cluster.

We name it my-kube-playground,

and specify the server address under the server field.

It also requires the certificate

of the certificate authority.

We can then add an entry into the users section

to specify details of my kube admin user.

Provide the location of the client certificate

and key pair, so we have now defined the cluster

and the user to access the cluster.

Next, we create an entry under the contexts section

to link the two together.

We will name the context my-kube-admin@my-kube-playground.

We will then specify the same name we used

for cluster and user.

Follow the same procedure to add all the clusters

you daily access.

The user credentials you use to access them,

as well as the contexts.

Once the file is ready, remember that you don't have to

create any object like you usually do

for other Kubernetes objects.

The file is left as-is, and is read by the kubectl command,

and the required values are used.

Now, how does kubectl know which context to choose from?

We've defined three contexts here.

Which one should it start with?

You can specify the default context to use

by adding a field current-context to the kubeconfig file.

Specify the name of the context to use.

In this case, kubectl will always use the context

dev-user@google to access the Google clusters,

using the dev user's credentials.

There are command line options available within kubectl

to view and modify the kubeconfig files.

To view the current file being used,

run the kubectl config view command.

It lists the clusters, contexts, and users,

as well as the current context that is set.

As we discussed earlier,

if you do not specify which kubeconfig file to use,

it ends up using the default file located

in the folder .kube, in the user's home directory.

Alternatively, you can specify a kubeconfig file

by passing the kubeconfig option

in the command line, like this.

We will move our custom config to the home directory,

so this becomes our default config file.

So, how do you update your current context?

So, you've been using my-kube-admin user

to access my-kube-playground.

How do you change the context to use prod user

to access the production cluster?

Run the kubectl config use-context command

to change the current context

to the prod user at production context.

This can be seen in the current context field in the file.

So yes, the changes made by kubectl config command

actually reflects in the file.

You can make other changes in the file,

update or delete items in it,

using other variations of the kubectl config command.

Check them out when you get time.

What about namespaces?

For example, each cluster may be configured

with multiple namespaces within it.

Can you configure a context to switch

to a particular namespace? Yes.

The context section in the kubeconfig file

can take additional field called namespace,

where you can specify a particular namespace.

This way, when you switch to that context,

you will automatically be in a specific namespace.

Finally, a word on certificates.

You have seen path to certificate files mentioned

in kubeconfig like this.

Well, it's better to use the full path,

like this, but remember, there's also another way

to specify the certificate credentials.

Let's look at the first one.

For instance, where we configure the path

to the certificate authority.

We have the contents of the ca.crt file on the right.

Instead of using certificate authority field

and the path to the file,

you may optionally use the certificate authority data field

and provide the contents of the certificate itself.

But, not the file as-is.

Convert the contents to a base64 encoded format

and then pass that in.

Similarly, if you see a file

with the certificate's data in the encoded format,

use the base64 decode option to decode the certificate.

Well, that's it for this lecture.

Head over to the practice exercises section,

and practice working with kubeconfig files

and troubleshooting issues.

-: Before we head into authorization,

it is necessary to understand

about API groups in Kubernetes.

But first, what is the Kubernetes API?

We learned about the Kube API server.

Whatever operations we have done so far with the cluster

we have been interacting

with the API server one way or the other,

either through the Kube control utility

or directly via REST.

Say we want to check the version.

We can access the API server at the master notes address

followed by the port, which is 6443 by default

and the API version. It returns the version.

Similarly, to get the list of pods

you would access the URL API slash V1 slash pods.

Our focus in this lecture is about these API pods:

the version and the API.

The Kubernetes API is grouped

into multiple such groups based on their purpose

such as one for APIs, one for health

one for metrics and logs, et cetera.

The version API is for viewing the version of the cluster,

as we just saw. The metrics and health API are

used to monitor the health of the cluster.

The logs are used

for integrating with third party logging applications.

In this video, we will focus

on the APIs responsible for the cluster functionality.

These APIs are categorized into two:

the core group and the named group.

The core group is where all core functionality exists

such as namespaces, pods, replication controllers,

events and points, notes, bindings, persistent volumes,

persistent volume claims, conflict maps,

secrets, services, et cetera.

The named group APIs are more organized.

And going forward, all the newer features

are going to be made available through these named groups.

It has groups under it for apps, extensions, networking,

storage, authentication, authorization, et cetera.

Shown here are just a few.

Within apps, you have deployments, replica sets,

StateFul sets.

Within networking, you have network policies.

Certificates have these certificate signing requests

that we talked about earlier in this section.

So the ones at the top are API groups, and the ones

at the bottom are resources in those groups.

Each resource in this has a set of actions

associated with them.

Things that you can do with these resources,

such as list the deployments, get information about one

of these deployments, create a deployment,

delete a deployment, update a deployment,

watch a deployment, et cetera.

These are known as verbs.

The Kubernetes API reference page can tell you

what the API group is for each object.

Select an object and the first section

in the documentation page shows its group details.

V1 core is just V1.

You can also view these on your Kubernetes cluster.

Access your Kube API server at port 6443 without any path

and it will list you the available API groups.

And then within the named API groups,

it returns all the supported resource groups.

A quick note on accessing the cluster API like that.

If you were to access the API directly

through curl as shown here, then you will not

be allowed access except for certain APIs like version,

as you have not specified any authentication mechanisms.

So you have to authenticate to the API

using your certified files by passing them

in the command line like this.

An alternate option is to start a Kube control proxy client.

The Kube control proxy command launches a proxy

service locally on port 8001

and uses credentials and certificates

from your Kube config file to access the cluster.

That way you don't have to specify those

in the curl command.

Now you can access the Kube control proxy service

at port 8001 and the proxy

will use the credentials from Kube config file

to forward your request to the Kube API server.

This will list all available APIs at root.

So here are two terms that kind of sound the same:

the Kube proxy and Kube control proxy.

Well, they're not the same.

We discussed about Kube proxy earlier in this course.

It is used to enable connectivity between parts

and services across different notes in the cluster.

We discuss about Kube proxy

in much more detail later in this course.

Whereas Kube control proxy is an HTTP proxy service created

by Kube control utility to access the Kube API server.

So what to take away from this.

All resources in Kubernetes are grouped

into different API groups.

At the top level,

you have core API group and named API group.

Under the named API group, you have one for each section.

Under these API groups, you have the different resources

and each resource has a set

of associated actions known as verbs.

In the next section on authorization,

we can see how we use these to allow

or deny access to users.

Well, that's it for this lecture.

I will see you in the next.

-: So far, we talked about authentication.

We saw how someone can gain access to a cluster.

We saw different ways that someone, a human or a machine

can get access to the cluster.

Once they gain access, what can they do?

That's what authorization defines.

First of all, why do you need authorization in your cluster?

As an administrator of the cluster,

we were able to perform all sorts of operations in it.

Such as viewing various objects like pods,

and notes, and deployments,

creating or deleting objects such

as adding or deleting pods or even notes in the cluster.

As an admin, we are able to perform any operation.

But soon, we will have others accessing the cluster as well

such as the other administrators, developers, testers,

or other applications like monitoring applications

or continuous delivery applications like Jenkins, et cetera.

So, we will be creating accounts

for them to access the cluster by creating user names

and passwords, or tokens, or signed TL certificates,

or service accounts as we saw in the previous lectures.

But we don't want all

of them to have the same level of access as us.

For example, we don't want the developers to have access to

modify our cluster configuration, like adding

or deleting notes or the storage or networking

configurations.

We can allow them to view, but not modify.

But they could have access to deploying applications.

The same goes with service accounts.

We only want to provide the external application

the minimum level of access to perform its required

operations.

When we share our cluster between

different organizations or teams, by logically

partitioning it using name spaces, we want to

restrict access to the users to their name spaces alone.

That is what authorization can help you within the cluster.

There are different authorization mechanisms supported

by Kubernetes, such as node authorization,

attribute based authorization, role-based authorization,

and Webhook.

Let's just go through these, now.

We know that the Kube API server is accessed by users

like customer management purposes, as well as the kubelets

on nodes within the cluster for management purposes

within the cluster.

The kubelet accesses the API server to

read information about services and points, nodes, and pods.

The kubelet also reports to

the Kube API server with information

about the node, such as its status.

These requests are handled by a special authorizer known

as the node authorizer.

In the earlier lectures, when we discussed

about certificates, we discussed that the kubelets should be

part of the system nodes group and have a name prefixed

with system node.

So any request coming

from a user with the name system node and

part of the system nodes group is authorized

by the node authorizer and are granted these privileges,

the privileges required for a kubelet.

So, that's access within the cluster.

Let's talk about external access to the API.

For instance, a user attribute based authorization

is where you associate a user

or a group of users with a set of permissions.

In this case,

we say the dev user can view, create and delete pods.

You do this, by creating a policy file

with a set of policies defined in adjacent format.

This way, you pass this file into the API server.

Similarly, we create a policy definition file

for each user or group in this file.

Now, every time you need to add or make a change

in the security, you must edit this policy file manually

and restart the Kube API server.

As such, the attributes based access control

configurations are difficult to manage.

We will look at role based access controls, next

Role based access controls make these much easier.

With role based access controls,

instead of directly associating a user

or a group with a set of permissions, we define a role.

In this case, for developers, we create a role

with the set of permissions required for developers.

Then, we associate all the developers to that role.

Similarly, create a role for security users

with the right set of permissions required for them.

Then, associate the user to that role, going forward.

Whenever a change needs to be made to the user's access

we simply modify the role

and it reflects on all developers immediately.

Role-based access controls provide a more standard approach

to managing access within the Kubernetes cluster.

We will look at role-based access controls

in much more detail in the next lecture.

For now, let's proceed

with the other authorization mechanisms.

Now, what if you want to outsource

all the authorization mechanisms?

Say you want to manage a authorization externally

and not through the built in mechanisms

that we just discussed.

For instance, open policy agent is a third party

tool that helps with admission control and authorization.

You can have Kubernetes make an API call

to the open policy agent with the information

about the user and his access requirements

and have the open policy agent decide

if the user should be permitted or not.

Based on that response, the user is granted access.

Now, there are two more modes

in addition to what we just saw.

Always allow and always deny.

As the name states always allow, allows all requests

without performing any authorization checks.

Always deny, denies all requests.

So, where do you configure these modes?

Which of them are active by default?

Can you have more than one at a time?

How does authorization work

if you do have multiple ones configured?

The modes are set using the authorization mode option

on the Kube API server.

If you don't specify this option, it is set to always allow.

by default.

You may provide a comma separated list

of multiple modes that you wish to use.

In this case, I want to set it to node , Rback and Webhook.

When you have multiple modes configured,

your request is authorized using each one

in the order it is specified.

For example, when a user sends a request

it's first handled by the node authorizer.

The node authorizer handles only node requests.

So, it denies the request.

Whenever a module denies a request,

it is forwarded to the next one in the chain.

The role-based access control module performs its checks

and grants the user permission.

Authorization is complete

and user is given access to the requested object.

So, every time a module denies the request,

it goes to the next one in the chain

and as soon as a module approves the request,

no more checks are done and the user is granted permission.

Well, that's it for this lecture.

In the upcoming lectures

we will discuss more about role-based access controls.

-: In this lecture,

we'll look at role-based access controls

in much more detail.

So how do we create a role?

We do that by creating a role object.

So we create a role definition file with the API version

set to R back dot authorization dot K etes dot io slash V1,

and kind set to role.

We name the role developer

as we are creating this role for developers,

and then we specify rules.

Each rule has three sections,

API groups, resources, and verbs.

The same things that we talked about

in one of the previous lectures.

For core group,

you can leave the API group section as blank.

For any other group, you specify the group name.

The resources that we want

to give developers access to are pods.

The actions that they can take are list, get, create,

and delete.

Similarly, to allow the developers to create config maps,

we add another rule to create config map.

You can add multiple rules for a single role like this.

Create the role using the kube control create role command.

The next step is to link the user to that role.

For this, we create another object called role binding.

The role binding object links a user object to a role.

We will name it dev user to developer binding.

The kind is role binding.

It has two sections.

The subject is where we specify the user details.

The role ref section is where we provide the details

of the role we created.

Create the role binding

using the kube control create command.

Also note that the roles and role bindings fall

under the scope of name spaces.

So here the dev user gets access to pods and config maps

within the default name space.

If you want to limit the dev user's access

within a different name space, then specify the name space

within the metadata of the definition file

while creating them.

To view the created roles,

run the kube control get roles command.

To list role bindings, run the kube control

get roll bindings command.

To view more details about the role,

run the kube control describe role developer command.

Here you see the details about the resources and permissions

for each resource.

Similarly, to view details about role bindings,

run the kube control describe role bindings command.

Here you can see details about an existing role binding.

What if you being a user would like to see

if you have access to a particular resource in the cluster?

You can use the kube control auth can I command

and check if you can say, create deployments,

or say, delete notes.

If you are an administrator,

then you can even impersonate another user

to check their permission.

For instance, say you were tasked

to create a necessary set of permissions

for a user to perform a set of operations

and you did that,

but you would like to test if what you did is working.

You don't have to authenticate as the user to test it.

Instead, you can use the same command

with the as user option like this.

Since we did not grant the developer permissions

to create deployments, it returns no.

The dev user has access to creating pods though.

You can also specify the name space

in the command like this.

The dev user does not have permission

to create a pod in the test name space.

A quick note on resource names.

We just saw how you can provide access

to users for resources like pods within the name space.

You can go one level down

and allow access to specific resources alone.

For example, say you have five pods in name space,

you wanna give access to a user to pods, but not all pods.

You can restrict access to the blue and orange pod alone

by adding a resource names field to the role.

Well, that's it for this lecture.

Head over to the practice exercises section

and practice working with role-based access controls.

Instructor: We discussed about roles

and role bindings in the previous lecture.

In this lecture, we will talk

about cluster roles and cluster role bindings.

When we talked about roles and role bindings,

we said that roles and role bindings are name spaced,

meaning they're created within name spaces.

If you don't specify a name space,

they're created in the default name space

and control access within that name space alone.

In one of the previous lectures, we discussed

about name spaces and how it helps in grouping

or isolating resources like pods, deployments and services.

But what about other resources like nodes?

Can you group or isolate nodes within the name space?

Like can you say node 01 is part

of the dev name space?

Now, those are cluster wide or clusters scoped resources.

They cannot be associated to any particular name space,

so the resources are categorized

as either name spaced or cluster scoped.

Now, we have seen a lot of name spaced resources

throughout this course like pods and replica sets

and jobs, deployments, services, secrets.

And in the last lecture,

we saw two new, roles and role bindings.

These resources are created

in the name space you specify when you created them.

If you don't specify a name space, they're created

in the default name space.

To view them or delete them or update them,

you always specify the right name space.

The cluster scoped resources are those

where you don't specify a name space when you create them

like nodes, persistent volumes,

the cluster roles and cluster role bindings

that we're going to look at in this lecture.

Certificate signing request we saw earlier

and name spaced objects themselves are

of course, not name spaced.

Note that, this is not a comprehensive list of resources.

To see a full list of name spaced

and non-name spaced resources,

run the cube control API resources command

with the name spaced option set.

In the previous lecture, we saw how to authorize a user

to name space resources.

We use roles and role bindings for that,

but how do we authorize users

to cluster via resources like nodes or persistent volumes?

That is where you use cluster roles

and cluster role bindings.

Cluster roles are just like roles except they are

for cluster scoped resources.

For example, a cluster admin role can be created

to provide a cluster administrator permissions

to view, create, or delete nodes in a cluster.

Similarly, a storage administrator role can be created

to authorize a storage admin to create persistent volumes.

Create a cluster role definition file

with the kind cluster role

and specify the rules as we did before.

In this case, the resources are nodes,

then create the cluster role.

The next step is to link the user to that cluster role.

For this, we create another object

called cluster role binding.

The role binding object links the user to the role,

we will name it cluster admin role binding.

The kind is cluster role binding.

Under subjects, we specify the user details,

cluster admin user, in this case.

The role ref section is where we provide the details

of the cluster role we created.

Create the role binding using

the cube control create command.

One thing to note before I let you go,

we said that cluster roles and bindings are used

for cluster scoped resources, but that is not a hard rule.

You can create a cluster role

for name spaced resources as well.

When you do that, the user will have access

to these resources across all name spaces.

Earlier, when we created a role to authorize a user

to access pods, the user had access

to the pods in a particular name space alone.

With cluster roles, when you authorize a user

to access the pods, the user gets access

to all pods across the cluster.

Kubernetes creates a number of cluster roles by default

when the cluster is first setup.

We will explore those in the practice tests

that are coming up. Good luck.

admission controllers.

So we've been running commands from our command line

using the kubectl utility to perform various kinds

of operations on our Kubernetes cluster.

And we know every time we send a request,

say to create a pod, the request goes to the API server

and then the pod is created

and the information is finally persisted

in the etcd database.

When the request hits the API server, we've learned

that it goes through an authentication process

and this is usually done through certificates.

If the request was sent through kubectl,

we know the kubeconfig file has the certificates configured

and the authentication process is responsible

for identifying the user who send the request

and making sure the user is valid.

And then the request goes through an authorization process,

and this is when we check if the user has permission

to perform that operation.

And we have learned that this is achieved

through role-based access controls.

So in this case if the user was assigned

this particular role of a developer

the user is allowed to list,

get, create, update or delete pods.

And so if the request that came in matched

any of these conditions, in this case

it does as the request is to create a pod.

It is allowed to go through, otherwise it's rejected.

So that's authorization with role based access control.

Now with role based access control,

you could place in different kinds of restrictions,

such as to allow or deny those with a particular role

to say create list or delete different kinds of objects

like pods, deployments or services.

We could even restrict access to specific resource names,

such as say a developer can only work on pods named blue

or orange, or restrict access within a namespace alone.

Now, as you can see, most of these rules that you can create

with role-based access control

is at the Kubernetes API level, and what user is allowed

access to what kind of API operations.

And it does not go beyond that.

But, what if you want to do more than just define

what kind of access a user has to an object?

For example, when a pod creation request comes in

you'd like to review the configuration file and look

at the image name and say that you do not want to

allow images from a public dock hub registry.

Only allow images from a specific internal registry.

Or to enforce that we must never use the latest tag

for any images.

Or say, for example you'd like to say,

if the container is running as the route user,

then you do not want to allow that request.

Or allow certain capabilities only,

or to enforce that the metadata always contains labels.

So, these are some of the things that you can't achieve

with the existing role-based access controls,

and that is where admission controllers comes in.

Admission controllers help us implement better

security measures to enforce how a cluster is used.

Apart from simply validating configuration,

admission controllers can do a lot more,

such as change the request itself

or perform additional operations

before the pod gets created.

We will go over some examples in the upcoming slides.

There are a number of admission controllers

that come prebuilt with Kubernetes,

such as always pull images that ensures that every time

a pod is created the images are always pulled.

The default storage class admission controller

that observes the creation of PVCs

and automatically adds a default storage class to them

if one is not specified.

The event rate limit admission controller can help set

a limit on the request with the APA server

can handle at a time, to prevent the APA server

from flooding with requests.

The namespace exists, admission controller rejects requests

to namespaces that do not exist,

and there are many more admission controllers available.

So let's take that as an example and look at it

in a bit more detail.

The namespace exists admission controller.

Say we want to create a pod in a namespace

called blue that doesn't exist.

If I run this command, it would throw an error

that says the namespace blue is not found.

What's happening here is that my request gets authenticated

then authorized, and it then goes through

the admission controllers.

The namespace exists that mission controller handles

the request and checks if the blue space is available.

If it is not, the request is rejected.

The namespace exists is a built in admission controller

that is enabled by default.

There's another admission controller that is not enabled

by default, and that is called as the

Namespace Auto Provision Admission controller.

This will automatically create the namespace,

if it does not exist.

We will see how it can be enabled in a minute.

First, to see a list of admission controllers enabled

by default run the kube API server/H command

and grip for enable admission plugins.

Now here you'll see a list of admission controllers

that are enabled by default

the ones that are highlighted in green.

Note that if you're running this in a Kube ADM based setup,

then you must run this command within

the Kube APA Server Control Plane Pod

using the kubectl exec command first like this.

To add an admission controller

update the enable admission plugins flag

on the Kube APA server service to add

the new admission controller.

So if you're in a kube ADM based setup

then update the flag within the kube APA server

manifest file as shown here on the right.

So the one on the left is

if you are updating the kube APA server service

and the one on the right is

if the APIs server is running as a pod

in a kube ADM based setup.

Similarly to disabled admission controller plugins,

you could use the disabled admission plugins flag.

Once updated the next time we run the command

to provision a pod in a namespace that does not exist yet,

the request goes through authentication, then authorization,

and then the namespace auto probation controller

at which point it realizes

that the namespace doesn't exist.

So it creates the namespace automatically

and the request goes through successfully to create the pod.

If you list the namespaces now,

you'll see that the blue namespace is automatically created.

So that's one example of how an admission controller works.

It cannot only validate and reject requests from users.

It can also perform operations in the backend

or change the request itself.

Note that the namespace auto provision

and the namespace exists admission controllers

are deprecated and is now replaced by the namespace

lifecycle admission controller.

The namespace lifecycle admission controller will make sure

that request to a non-existent namespace is rejected

and that the default namespaces such as

default kube system and kube public cannot be deleted.

Well, that's it for now. Head over to the labs and practice

are working with admission controllers and I will see you

in the next lecture.

-: In this lecture, we will take a closer look

at the different types of admission controllers

and how we can configure our own admission controller.

We looked at the namespace exists

or namespace lifecycle admission controller.

It can help validate if a name space already exists

and reject the request if it doesn't exist.

This is known as a validating admission controller.

Let's look at another type

of admission controller plugin

named as the default StorageClass plugin.

This is a plugin that is enabled by default.

Say for example,

you're submitting a request to create a PVC.

The request goes through authentication authorization

and finally, the admission controller.

The default StorageClass admission controller

will watch for request to create a PVC

and check if it has a StorageClass mentioned in it.

If not, which is true in our case,

it'll modify your request to add the default StorageClass

to your request.

This could be whatever StorageClass is configured

as the default StorageClass in your cluster.

So when the PVC is created and you inspect it,

you'll see that a StorageClass default is added to it

even though you hadn't specified it during the creation.

So this type of admission controller

is known as a mutating admission controller.

It can change or mutate the object itself

before it is created.

So those are two types of admission controllers.

Mutating admission controllers

are those that can change the request,

and validating admission controllers

are those that can validate the request

and allow or deny it.

And there may be admission controllers that can do both.

That can mutate a request as well as validate a request.

Now, generally, mutating admission controllers

are invoked first

followed by validating admission controllers.

This is so that any change made

by the mutating admission controller can be considered

during the validation process.

And this example,

the namespace auto provisioning admission controller

which is a mutating admission controller,

is run first followed by the validating controller

namespace exists.

If it was run the other way,

then the namespace exists admission controller

would always reject the request for a namespace

that does not exist

and the namespace auto provisioning controller

would never be invoked to create the missing namespace.

And when a request goes through these admission controllers,

if any admission controller rejects the request,

the request is rejected and an error message

is shown to the user.

Now, these are all built-in admission controllers

that are part of the Kubernetes source code

and are compiled and shipped with Kubernetes.

Now, what if we want our own admission controller

with our own mutations and validations

that has our own logic?

To support external admission controllers,

there are two special admission controllers available:

mutating admission webhook,

and then validating admission webhook.

And this is what we will look at next.

We can configure these webhooks

to point to a server that's hosted

either within the Kubernetes cluster or outside it,

and our server will have our own admission webhook service

running with our own code and logic.

After our request goes through,

all the built-in admission controllers

it hits the webhook that's configured.

We will see how to configure that in a bit.

And then once it hits the webhook,

it makes a call to the admission webhook server

bypassing in an admission review object in adjacent format.

This object has all the details

about the request such as the user that made the request,

and the type of operation the user is trying to perform,

and on what object and the details

about the object itself.

On receiving the request,

the admission webhook server responds

with an admission review object

with a result of whether the request is allowed or not.

If the allowed field in the response is set to true

then the request is allowed,

and if it's set to false it is rejected.

So how do we set this up?

First, we must deploy our admission webhook server

which will have our own logic,

and then we configure the webhook on Kubernetes

by creating a webhook configuration object.

So let's take a look at each of these steps next.

So the first step is to deploy our own webhook server.

Now, this could be an API server

that could be built on any platform.

An example, code of a webhook server written in Go,

now can be found here in the Kubernetes documentation pages.

It's written in the Go programming language.

You could develop your own server in other languages

as well if required.

The only requirement is that it must accept the mutate

and validate APIs and respond with the JSON object

that the web server expects.

So here's a pseudo code of a sample webhook server

written in Python.

There are two calls, a validate call and a mutate call.

The validate call receives the validation webhook request

and in this example compares the name of the object

and the name of the user who sent the request

and rejects the request if it's the same name.

Well, just a simple example use case to show what we can do

with the request that come in.

And if you look down, we'll see the mutating webhook

which gets the username and response

with a JSON Patch operation of adding the username

as a label to any request that was raised by anyone.

So if you take a closer look

at this particular piece of code,

a patch object is a list of patch operations

with each operation being add, remove, replace, move,

copy or test,

and we then specify the path

within the JSON object that needs to be targeted for change.

In this case, it is /metadata/label/users,

and then the value that needs to be added

if it is an add operation.

So we get the username from the request.

So that's gonna be the value of that particular label.

This is then sent as a base 64 encoded object

as part of the response.

On a side note, from an exam point of view,

you will not be asked to develop any code like this,

so don't worry if you don't fully understand

this piece of code.

All you need to take away from this

is that the admission webhook server

is a server that you deploy that contains the logic

or the code to permit or reject a request

and it must be able to receive

and respond with the appropriate responses

that the webhook expects.

So this is just a simple example to show

what kind of things that you can do

or what kind of things that you can code

and implement in the webhook server that you deploy.

Okay.

So moving on, once we have developed our own webhook server

the next step is to host it.

So we either run it as a server somewhere

or containerize it and deploy it

within Kubernetes cluster itself as a deployment.

If deployed as a deployment in a Kubernetes cluster,

then it needs a service for it to be accessed.

So we have a service named webhook service as well.

The next step is to configure our cluster

to reach out to the service

and validate or mutate the requests.

For this,

we create a validating webhook configuration object.

So we start with the API version,

kind, metadata and webhooks section.

API version is admissionregistration.k.io/v1,

kind is validating webhook configuration.

If we are configuring a mutating webhook,

this would be a mutating webhook configuration.

We then give it a name.

Now under webhooks, we configure the different webhooks.

So a webhook has a name,

a client config and a set of rules.

So the name we set it to podpolicy.example.com.

The client config is where we configure the location

of our admission webhook server.

If we deploy this server externally on our own

that is not a part of a deployment in Kubernetes cluster,

then we can simply provide a URL path

to that server like this.

Now instead, if we deployed the server as another service

on our own cluster as we see here as it is on the left,

then we can use the service configuration

and provide the namespace, and name of the service,

which in our case is webhook-service.

Now, of course, the communication between the API server

and the webhook server has to be over TLS.

So a certificate bundle should be configured.

So the server has to be configured

with a pair of certificates.

Then a CA bundle is to be created

and passed into this client config SSEA bundle.

Next, we must specify when to call our API server.

We could specify rules to configure exactly

when we want our web hook server

to be called for validation.

Now, we might not want to do that for all of the calls:

for example, we may only want it to be called

while creating pods, or deleting pods,

or creating deployments, et cetera.

Now, whatever that may be, it can be added as a rule

under the rules section using API groups,

API versions, operation types and resources.

In this example,

we are only going to call this webhook configuration

when calls are made to create pods.

And that should be it.

Once this object is created,

every time we create a pod

a call would be made to the webhook service,

and depending on the response,

it would be allowed or rejected.

Well, this is for this lecture.

Head over to the labs and practice working with webhooks

and I will see you in the next lecture.

Autoscroll

Course content

Overview

Q&AQuestions and answers

Notes

Announcements

Reviews

Learning tools

Instructor: We will now discuss about API versions.

We talked about APIs and API groups, resources and verbs.

Now we are going to talk about API versions.

We know that everything under APIs are the API groups

such as apps, extensions, networking, et cetera.

And each API group has different versions.

Now, what we see here is the V1 version.

When an API group is at V1

that means it is a GA stable version

generally available stable version.

The API group may have other versions

such as Beta or Alpha

as V1 Beta 1 or V1 Alpha 1 respectively.

So what are these different versions?

What do these each of these versions mean?

Now Alpha is when an API is first developed

and merged to the Kubernetes code base

and becomes part of the Kubernetes

released for the very first time.

At this stage, the API version has Alpha in its name

indicating that it is an Alpha release.

For example, it could be V1 Alpha 1

or V1 Alpha two, et cetera.

This API group is not enabled by default.

For example, at the time of this recording,

the API group internal dot API server dot case dot io,

which has the resource storage version

is only available as V1 Alpha 1.

This means that if you were to create this object now

you will have to say in the amul definition file

the API version

as internal API server dot case dot io/ V1 Alpha 1.

However, this Alpha version is not enabled by default.

So if you try to create this object

you will not be able to do that.

We'll discuss about how to enable a particular API version

specifically those in the Alpha version,

a few minutes later in this video.

Now, since this is an Alpha version

it may lack end to end tests and may have bugs.

As such, it is not very reliable.

Also, there is no guarantee

that this API will be available in the future releases.

It may be dropped without any notice in future releases.

As such, this is only really for expert users

who are interested in testing and giving early feedback

for the API group.

As of this recording

the API groups in Alpha phase

are the internal API server dot case dot IO group.

Now, after some time the Alpha version

and once all the major bug are fixed

and it has end to end test, it moves to the Beta stage.

This is where the API version name gets Beta in name.

For example,

V1 Beta one or V1 Beta two, et cetera.

The API groups in Beta stage

are enabled by default

and they have end-to-end tests.

But since it is not GA,

it may still have minor bugs.

However, there is commitment

from the project that it may be moved to GA in the future.

This is for users who are interested

in Beta testing and providing feedback for the features.

As of now, the flow control group is in Beta phase,

as as you can see from this chart.

Now, after being in the Beta stage

for a few months and with a few releases

the API group moves to the GA or stable version.

Now, this is when the version number no longer

has Alpha or Beta in it.

Instead, it is just V1, and it is of course enabled

by default in the API group,

and it is part of conformance tests

and has all the tests written.

Now, since most bugs are fixed

in Alpha and Beta stages in the API group,

this API group is highly reliable

and it'll be supported and present

in many feature releases to come.

And of course, this is when it is available to all users.

Now, as you might have noticed

most of the API groups such as apps, authentication

authorization certificates, coordination, et cetera

are GA and stable now.

However, if you look at some of them,

they have multiple versions.

For example,

autoscaling has V1, V2, Beta 2, V2, Beta 1, et cetera.

The node API group has V1

V1 Beta 1 and V1 Alpha versions.

And so why is that and what does that mean?

So an API group can support multiple versions

at the same time.

For example, the apps group can have..

V1 Beta 1 and V1 Alpha 1

all at the same time.

This would mean

that you will be able to create the same object

using any of these versions in your YAML file.

So say I want to create an nginx dot yaml,

I could create it

with the API version set to V1 Alpha 1

or V1 Beta 1 or V1.

So any of these will work.

But even though there are multiple versions supported

at the same time,

only one can be the preferred or storage version.

So what is a preferred version?

Now, when you have multiple versions enabled

and you run the cuckoo get deployment command

which version is the command going to query?

That's defined by the preferred version.

In this case, if V1 is set to the preferred version

then that is the command that it will query.

Or when you run the kubectl explain command

the version that it returns is the preferred API version.

Also, when multiple versions are present,

only one version can be the storage version.

This means if any object is created with the API version set

to anything other than the storage version,

such as V1 Alpha 1 or V1 Beta 1,

then those will be converted to the storage version,

which is V1

before storing them into the etcd database.

So that's what the preferred and storage versions mean.

Preferred is the default version used

when you retrieve information through the API

using kubectl get command or something like that.

And the storage version is the version

in which an object is stored

in etcd respective of the version you have used

in the yaml file while creating the object.

Remember, only one can be the preferred or storage version.

And even though the preferred

and storage versions are usually the same

they can be different.

They don't necessarily have to be the same.

So how do you identify the the preferred version

if there are multiple APIs?

So the preferred version is listed when you list

that specific API.

For example, when I list the APIs available

for the batch API group,

by going to the URL as shown here

I see that it has V1 and V1 Beta 1 versions available

which are the supported API versions

of which the preferred version is V1 as is shown below.

And as of now

it is not possible to see which is the storage version

of a particular API through an API or a command.

The one way to find that out

is by looking at the stored object

in etcd itself by querying the etcd database directly.

Here's a sample command where we are querying

the blue deployment object

using the etcd kubectl utility using the get command.

And in the output we see the API version

to be apps slash V1.

So to enable or disable a specific version

you must add it to the runtime config parameter

of the Kube API server service.

For example, we talked about a number of Alpha APIs earlier

and since these APIs are not enabled by default

you will not be able to create objects for them.

So if you'd like to try those out

what you could do is you could actually add those

to the runtime config parameter

and the APIs here are comma separated

and you would like to enable

and specify the APIs that you would like to enable.

And of course, once you do that

you must remember to always restart the API server service.

And that goes to whenever you edit these options.

And once done

you should be able to use that API

and test it out.

Well, that's all for now

and I will see you in the next lecture.

-: We will now discuss about API deprecations.

So we discussed

that a single API group can support multiple versions

at a time, but why do you need to support multiple versions

and how many should you support?

When can you remove an older version that is

no longer required?

This is answered by the API deprecation policy.

By looking at the answers to these questions

we will also understand the rules

in the API deprecation policy.

First, let's look at the why.

Let me give you a quick tour

of the life cycle of an API group.

Let's for example say that we are planning to contribute

to the Kubernetes project.

So we create an API group called code cloud.com

under which we have two resources called Cores and Webinar.

This is just an example.

So we develop it in-house and we test it

and when we are ready to merge it to the Kubernetes project

we raise a PR and hopefully it gets accepted

and we release it as an alpha version.

So we call it V1 Alpha one

because it's the first alpha version of the V1 version

you can create a course

or a webinar object using yaul file like this

and using the API version code cloud.com/v1 alpha one.

Now, let's say for instance

the webinar object didn't go well

with the users and we decided to remove it.

In the next Kubernetes release.

can we just remove it from the V one alpha one version?

No. That's where the first rule

of API application policy comes into play.

API elements may only be removed

by incrementing the version of the API group

meaning you can only remove the webinar element

from the WE one alpha two version of the API group.

It will continue to be present

in the V one alpha one version of the element.

So at this point, the resource

in database is still at B one Alpha one

but our API version has now changed to B one alpha two.

So this is now going to be a problem.

We will need to go back and change all API versions

in our files from B one alpha one to V one alpha two

which is why the new releases must support

both V one alpha one and V one alpha two.

But the preferred

or storage version could be we one alpha two.

This means that the users can use the same yamo

files to create the resources, but internally

it would be converted and stored as V one alpha two.

So that brings us

to the second rule of API deprecation policy.

API objects must be able to round trip between API versions

in a given release without information loss

with the exception

of whole risk resources that do not exist in some versions.

If we create an API, an object

in the V one alpha one version

and it is converted to V one alpha two

and then back to V one alpha one, it should be the same

as the original V one alpha one version.

For example, we have a course object in B one alpha one.

It has a spec field called type set two video.

This is then converted

to B one alpha two with B one alpha two

we introduced a new field called Duration.

This field was not there

in we one alpha one where we now convert this back

to B one Alpha one.

It will now have the new field, but the original one didn't.

So we must add an equivalent field

in B one alpha one version so

that the converted B one alpha one matches the

original B one alpha one version.

So continuing with our story

we fixed some more bugs and are now ready for beta.

Our first beta version called v1, BTA one is now ready

and then after a few months we release the next beta version

of v1 beta two.

And finally we release our stable version, the GA version

which we call as v1.

So that's kind of how an API evolves over time.

Starts with we want alpha one, then alpha two

you required more alpha versions and then to beta versions

beta one, beta two, and more beta versions if required.

And finally we want

now we don't have to have all the versions available

at all times like this.

Of course, we must deprecate

and remove older versions as we release newer versions.

So let's look at what are the rules

and best practices around that.

So let's say with the Kubernetes release of version X

and we released the B one alpha one

version of our API group.

So X here could be Kubernetes version release 12.1 0.1

for example.

And that's the first time our package was included.

So we made it version V one alpha one.

Now since it is the only version it is

the preferred and storage version.

With the Kubernetes release of x plus one

we released the V1 alpha two version of our API group.

Now, since we are in the alpha phase

we are not required to keep the older V1 alpha one version

as part of this new release.

And this is part of the rule

for a of the Kubernetes deprecation policy.

This rule states that other

than the most recent API versions in each track

older API versions must be supported

after they're announced deprecation for a duration

of no less than GA 12 months

or three releases for GA and nine months

or three releases for beta and zero releases for alpha.

So alpha versions need not be supported

for any release, but beta and GA versions once

released must be supported anywhere from nine to 12 months.

So that's why the release x

plus one does not have V1 alpha one.

So that may break things

for those who had previously used the V1 alpha one

as as we discussed earlier in this in this video.

And so you must mention about this change

in the release node of that Kubernetes version.

Here's an example of one such mention

of the removal of V1 alpha two configure API

and the release node requests users to

convert the V1 alpha two to V1 alpha three.

In that example, we will talk

about how to do just that a bit later in this video.

So now in the x plus two version

we release the first VI version

of our API group is called V one, beta one.

And now since the previous version was an alpha version

which is V one alpha two, it is not required

for that version to be part of the new release.

And same as before

the release notes must be updated to notify users to migrate

from B one alpha two to V one beta one.

Now that we are in beta going forward

things are going to change because rule four A states

that beta versions need to be supported

for nine months or three releases, whichever is longer.

So continuing to x plus three release.

Now on x plus three, we have, we want beta two released.

Now, as we discussed earlier from the beta version onwards

it is required to have the previous beta release

which is V1 beta one in this case as part of this release.

So this release has both the new beta version as well

as the previous beta version

as part of this release.

The V1 beta one version is now deprecated a note

that we want beta one version is only deprecated

but not removed.

It's still part of this release

it's still going to be there

for a couple of releases before it, it is actually removed.

If you were to use the we want beta one API at this point

it would display a deprecation warning.

Now also note that at this time we want beta one

is continuing to be the preferred version.

So why is that?

Why isn't the, v one beta two version

the preferred and storage version

because rule four B states that the preferred API version

and the storage version for a given group may not advance

until after a release has been made that supports

both the new version and the previous version.

So in this case

the current release is the first release where

both new and previous version

that's beta one and beta two are supported.

So we cannot change the preferred storage version yet.

Instead, we must wait till the next release.

The next release also has both versions except

beta two now becomes the preferred storage version.

But continuing on with version X plus five

we now finally have the V one GA stable version.

But along with that, we also have V one beta one

and V one beta two versions.

V one beta two is still the preferred or storage version

because this is the first release that has

the V one version, but it now becomes deprecated.

So now V one beta one and V one beta two are deprecated.

In the next release, we can remove V one beta one version

because as per rule four A

a beta version needs to be supported for three releases.

beta one version was deprecated in release X plus three

and it's been around for X plus four and X plus five.

So that's three releases

and it can now be removed.

With x plus six V one can now

be the preferred and storage version

while beta two is going to stay around

for another release to complete its three release support

with X plus seven version, nothing changes

as beta two continues to stay around

for the last time before it will be removed.

And in x plus eight version, beta two is removed

and we are left with just the GA stable V one version.

That's kind of how a version progresses.

Now, let's say for example with the next release, we are now

for the first time having the first version of V two

so we decided to start working on a V2 version

and we now have the V2 Alpha one version available

for the first time.

Now, if you look at the releases in the past

every time we had a new version available, we deprecated

we immediately deprecated the older version.

But now that we have the V two alpha one version ready

can we deprecate the V1 version now?

No, because that's where rule three comes in.

And rule three says an API version

in a given track may not be deprecated

until a new API version, at least as stable is released.

Meaning GA versions can deprecate beta or alpha versions

but not the other way around.

An alpha version aren't deprecate a GA version in this case

the V2 alpha one is an alpha version

and V one is a GA version, so we cannot deprecate V one now.

The V two alpha one version needs to go

through its life cycle of becoming a V two, alpha two

then V two beta one, then v2 beta two, and then V two

and and only can the V2 version deprecate the V1 version.

So that brings us

to the final topic of this video, the cube converts command.

Now, as we have been seeing when Kubernetes clusters

are upgraded, we have new APIs being added

and old ones being deprecated and removed.

Now, when an old API is removed, it is important to note

that you have to update your existing manifest files

to the new version.

For example, say I have a yamo file

with a deployment definition of v1 beta one

and when Kubernetes is upgraded

the beta one version of deployment is removed.

So I would need to use the V1 version going forward.

However, I may have a lot of yamo files in the old manifest

in the old version, which has V1 beta one in it.

So to convert my YAML files to newer version

you may use the cube cuttle convert command

and specify the old yamo file and the new output format.

For example, to convert this deployment definition file

to the new version, run the cube convert command

and specify the old file name followed by the new version

which happens to be app slash B one.

And this will output the new version

of the manifest definition in YAML format on screen.

Note that the cube convert command may not be available

on your system by default

and that's because it's a separate plugin.

So the Cube kele convert command is a special

plugin that you have to install.

The installation instructions can be found

in the Kubernetes documentation pages, along

with the instructions for installing the cube kele utility.

And so it's available at this link.

We will go over this in the upcoming labs.

Well, thanks for your time

and I will see you in the next one.

Instructor: Let's look at custom resource definitions

in Kubernetes.

Before we talk about custom resources,

let's first talk about resources.

So here's a deployment definition file.

And when we create a deployment,

Kubernetes creates a deployment and stores its information

in the ETCD data store.

And we can create this deployment

and then list the deployment and see the status.

And we can run the delete command to delete the deployment.

Now all of this is simply going to create, list,

and modify the deployment object or resource

in the ETCD data store.

But we know that when we create deployments

it creates pods equal to the number of replicas

defined in the deployment.

In this case, three.

So who or what is responsible for that?

And that's the job of a controller.

In this case, the deployment controller.

Now we don't have to create a controller

because the deployment controller

is built in to the Kubernetes and it's already available.

So the controller is a process that runs in the background

and its job is to continuously monitor the status

of resources that it's supposed to manage.

In this case, the deployment that we created.

And when we create, update, or delete the deployment,

it makes the necessary changes on the cluster

to match what we have done.

In this case, when we create a deployment object,

the controller creates a replica set

which, in turn, creates as many pods

as we have specified in the deployment definition file.

That's the job of a controller.

And here is how it looks in code.

So the deployment controller is written in Go

and is part of the Kubernetes source code.

You don't have to understand Go

or understand this code for now.

I'm just showing this

so you can see what a controller actually looks like.

Okay, mow we've learned about many resources

throughout this course such as replica sets, deployments,

jobs, cron jobs, stateful sets, namespace, et cetera.

These are just few of the many resources available

on the cluster.

And these have controllers that are responsible

for watching the status of these objects

and making the necessary changes on the cluster

to maintain the state as expected.

Now let's do something fun.

Let's just like how we created the deployment

to deploy multiple parts in a cluster,

we would like to create, say,

a flight ticket object to book a flight ticket.

I'm just picking this use case

to show that this could be really anything,

like literally anything.

And later towards the end of the video

or end of this section,

we will see some real life use cases.

Let's stick with this for now.

And we are first, we are going to see what we want

and later, we will see how to achieve it.

So what I want to do is create an object

called flight ticket.

We will say the API version is flights.com/v1.

The kind is FlightTicket. We'll name it my-flight-ticket.

The spec section will have some properties

required to book the ticket such as a from airport,

which we will set to Mumbai,

and a to address, which we'll set to London,

and also another property called number

to specify the number of tickets that I want to book

and that is two.

When I create this object,

I want to have a flight ticket resource created.

And when I list all flight tickets,

I want all flight tickets to be listed.

And when I delete a flight ticket,

I want the flight ticket booking to be deleted.

Now how we're going to do this, I'll explain in a bit.

For now, we're just discussing the what. What do we want?

So this is going to create or delete

the flight ticket object in the ETCD data store.

But it's not actually going to book a flight ticket, is it?

We want this to actually go out

and book a flight ticket for real.

Say for instance, there is this API available at, say,

book-flight.com/api

that we can call to book a flight ticket.

So how do we call this API

whenever we create a flight ticket object

to book a flight ticket?

So for that, we're going to need a controller.

So we'll create a flight ticket controller

and it's written in Go,

and we will watch for the creation or deletion

of the flight ticket resource.

And when we create a resource,

it will contact the book flight ticket API

to book a flight ticket.

And when we delete the resource,

it would make an API call to delete that booking.

And it might look something like this in code.

So that's how it works at a high level.

So the flight ticket object that we created

is a custom resource and the controller that we wrote

to book the actual flight ticket by calling the API,

that is called as a custom controller.

We have a custom resource and we have a custom controller.

Now let's see how we are going to achieve this.

If you tried to create a flight ticket resource now

on your Kubernetes cluster,

you will see that it fails with the error message

that says there are no matches for the kind of flight ticket

in version flights.com/v1.

Now this is because you can't simply create any resource

that you want without configuring it in the Kubernetes API,

without first telling Kubernetes that it should allow us

to create a flight ticket object.

So that's what we have to do first.

You have to first define what that resource is

that we want to create.

So for that, we need what is known as

a custom resource definition or CRD.

We're going to use CRD to tell Kubernetes

that we would like to create objects

of kind FlightTicket going forward.

So the CRD is a similar object

with API version, kind, metadata, and spec.

The API version is apiextensions.k8s.io/v1.

Kind is CustomResourceDefinition.

Metadata has name set to flighttickets.flights.com,

let's say.

And under the spec section, we will specify a scope.

Now scope defines if the object is namespaced or not,

and we know that there are namespaced

and non-namespaced scopes in Kubernetes.

For example, pods, replica sets, deployments are all scoped,

whereas persistent volumes, nodes,

and namespace itself are non-scoped objects.

So here we can define if this object

is going to be namespace scoped or not.

We'll set it to namespaced for now.

Then we define groups.

Groups is the API group that we provide in the API version.

It'll be flights.com.

And then we specify the names of the resource.

We define the kind, which is the kind that we used

in the flight ticket definition file

and we will set it to FlightTicket.

And we must also specify a singular

and plural versions of names.

So the singular name here is just the flight ticket

which is used to display the resource type

in the output of the kubectl commands.

The plural is what is used by the Kubernetes API resource.

And if you run the kubectl API resources command,

this is what is going to be shown here in the plural format.

So it's flight tickets there,

and, of course, you can provide a short version of the name.

Let's say FT.

This way we can just refer to the resource

as FT when we run the kubectl commands.

Next we have versions.

Each resource can be configured with multiple versions

as it passes through the various phases of its lifecycle.

For example, if it's a new resource type

that we are going to introduce it,

then we start at alpha or beta versions

before making its way to version v1.

This is something that we discussed

in the API versions lecture earlier in this course.

Let's name it version v1 for now.

And with multiple versions configured for the same resource,

we must choose which ones are served through the API server

and we also define which is the storage version.

If you have multiple versions,

only one version can be marked as the storage version.

We discussed about what are storage versions

in the lecture about APIs earlier in this course.

Next, we specify the schema.

The schema defines all the parameters that can be specified

under the spec section when you create the object.

It defines what fields are supported

and what type of value that fields supports, et cetera.

The schema uses OpenAPI V3 schema version.

We specify the different properties using an object type.

We have from, to, and number.

And from and to are string. Number is an integer.

We can also specify validations like the minimum

or maximum value that can be specified under the number.

If the value is entered by the user does not fall

within this range, the resource will not be created

and will return an error message.

We can now create the custom resource definition

by running the kubectl create command.

And once the custom resource definition is created,

you can now create the flight ticket object

and then get or delete it.

So that solves the first part of our problem,

being able to create any kind of object type

that we want on Kubernetes.

You can use custom resource definitions, or CRDs,

to create any kind of resource you want on Kubernetes

and specify a schema and add validations.

But it's only going to allow you to create these resources

and store them in ETCD.

It's not actually going to do anything

about these resources yet

because we don't yet have a controller for it

and that's what we will look at next.

The second part is building a custom controller

that can handle these resources that can watch

when these resources are created

and perform actions based on the resource specifications.

You see, without a controller,

the custom resource that we created

is just going to sit there.

It's just data locally in our ETCD data stores

and does not actually do anything.

In the upcoming video, we'll see how to get started

with creating such a controller.

-: In this lecture

we will look at developing Custom Controllers.

So to pick up from where we left off,

we have created a custom resource definition,

and we are able to create our Flight Ticket objects.

And the data about the ticket is stored in ETCD.

Now what we need to do is monitor the status

of the objects in ETCD

and perform actions such as making calls

to the flight booking API to book, edit,

or cancel flight tickets.

And that's why we need a Custom Controller.

So a controller is any process or code that runs in a loop

and is continuously monitoring the Kubernetes cluster

and listening to events of specific objects being changed,

in this case, the flight ticket object.

And now we could do that in any way we can.

Of course you need to write some code.

So say I know Python well.

So I could write a code in Python

that would query the Kubernetes API server

for objects in the Kubernetes API.

And then I could watch for changes

and make further calls to the API

to make changes to the system.

However, developing a controller in Python

may be challenging,

as the calls made to the APIs may become expensive,

and we will need to create our own queuing

and caching mechanisms.

Developing in Go with the Kubernetes Go Client,

provides support for other libraries

like shared informers that provide caching

and queuing mechanisms

that can help build controllers easily

and in the right way.

So how do you start building a Custom Controller?

So there's a GitHub repo named sample-controller.

So first clone this repo,

and then we modify the controller,

.go with your custom code.

Then we build and run it.

So make sure you have the Go programming language

installed on your machine.

Install it if it is and if it is isn't already installed.

And then clone the GitHub repo of the sample-controller.

We then see the into the sample controller directory,

and then we customize the controller.go

with our custom logic.

And we're not going to get into the details

about the code itself at this point.

We will probably go over it at some other time

where we will focus on advanced topics

that would include like building controllers and operators.

For now, let's just assume

that we've customized the controller.go code.

And somewhere within the code,

we're making a call to the flight booking API

and to book flight tickets.

And then once we do that, we then build the code,

and then we run it.

And when we run it, we specify the kube config file

that the controller can use to authenticate

to the Kubernetes API.

And then the control process starts locally

and it then watches for the creation

of the flight ticket objects

and makes the necessary pulse.

Once your controller is ready,

you can decide on how to distribute it.

Now you don't want to build and run it each time.

So you may package the Custom Controller

in a Docker image and then choose

to run it inside your Kubernetes cluster

as a pod or a deployment.

That's a high level overview

of building a Custom Controller.

Now in the exam, I don't expect them to ask a question

about building a Custom Controller

as it requires more coding knowledge.

So I don't anticipate a question on this.

However, there may be questions

to build custom resource definitions

and work with customer resource definition files,

right, or to work with existing controllers

that are already there.

It's good to know.

All right, so that's all for now.

And in the next video, we will talk about operators.

-: In this section, we will talk about Helm.

Now, Kubernetes is awesome

at managing complex infrastructures.

We humans tend to struggle with complexity

though applications we deploy

into our Kubernetes cluster can become very complicated.

A typical app is usually made up

of a collection of objects

that need to interconnect to make everything work.

For example, even a relatively simple WordPress

site might need the following,

a deployment to declare the pods that you wanna run

like with MySQL database server or the web servers.

A persistent volume to store the database,

a persistent volume claim,

a service to expose the web server running in a pod,

and a secret to store the admin password

and maybe even more

if you want extra stuff like periodic backups and so on.

And for every object, we might need a separate YAML file.

Then we need to kube cuddle apply to every YAML file.

Now, this can be a tedious task,

but it's not the end of the world.

Now imagine we download these YAML files from the internet.

We're not happy with the defaults,

so we start changing stuff.

Persistent volumes are 20GB,

but we know our website will need much more storage

and we go to the YAML files where persistent volumes

and claims are declared.

We change 20 to 100 and more stuff we wanna change.

We'll have to open up every YAML file

and edit each one according to our needs.

Well, not bad enough yet.

Now imagine two months go by.

We now have to upgrade some components in our app,

and back to editing multiple YAML files with great care.

So we don't change the wrong thing in the wrong place.

We need to delete the app.

We'll need to remember each object that belongs

to our app and delete them all one by one.

Now you might be thinking,

hey, it's not a big deal.

We can just write all object declarations

in a single YAML file.

Well, that's true, but it might make it even

harder to find the stuff that we are looking for.

We'd have to continuously search for stuff

we need to edit in something that could be 25 pages of text,

at least in multiple files,

it'd be somewhat organized

and we'd know we'll find deployment related stuff,

in the the MySQL deployment of YAML file.

Enter Helm.

Helm changes the paradigm.

Kubernetes doesn't really care about our app as a whole.

All that it knows is that

we declared various objects

and it proceeds to make each of them exist in our cluster.

It doesn't really know

that the persistent volume

and that deployment and that secret

and that service are all part

of a big application called WordPress.

It looks at all the little pieces

that the administrator wanted to have in the cluster

and takes care of each one individually.

Helm, however, is built from the ground up

to know about such stuff.

That's why it's sometimes called

a package manager for Kubernetes.

It looks at those objects as part of a big package

as a group, whenever we need to perform an action

we don't tell Helm the objects it should touch.

We just tell it what package we want it to act on,

like our WordPress app package.

And based on the package name,

it then knows what objects it should change and how.

And even if there are hundreds

of objects that belong to that package.

To make this easier to understand,

well think about this.

A computer game is contained

in hundreds or thousands of files.

There are a few files with the programs executable code,

other files with audio, game sounds, and music

and other files with graphics, textures, images

files with configuration data and so on.

Now, imagine what would happen

if we had to download each of these files separately.

That would be tedious.

Fortunately, we don't have to go

through such horrors as we get a game installer.

We run it.

We choose the directory where we want to install,

we press install

and the installer does the rest,

putting thousands of files in their proper location.

Helm does a similar thing and more

for the YAML files and Kubernetes objects

that make up our application.

Hence, we get advantages like this.

We use a single command to install our whole app,

even if it needs a hundred objects.

Helm proceeds to automatically add every necessary object

to Kubernetes without bothering us with the details.

We can customize the settings we want for our app or package

by specifying desired values at install time.

But instead of having to edit multiple files

in multiple YAML files,

we have a single location where

we can declare every custom setting.

It's called a value.yaml file,

and this is where

we can change the size of our persistent volumes,

choose the name of our WordPress website,

the admin password,

settings for the database engine and so on.

We can upgrade our application with a single command.

Helm will know what individual objects need to change

to make this happen.

We can roll back to the previous

so-called revision.

We use a single command to uninstall our app.

It keeps track of all the objects used by each app,

so it knows what to remove.

We don't need to remember each object

that belongs to one of our apps anymore

or use 10 separate commands to remove everything.

Helm does all the work.

We will look into these commands in more detail

in the upcoming lectures.

For now, understand that Helm works as a package manager,

with install or uninstall wizard,

and also a release manager,

helping us upgrade or roll back application.

The core thing is that it lets us treat our Kubernetes apps

as apps instead of just a collection of objects.

This takes a huge burden of our shoulders

as we don't have to micromanage

each Kubernetes object anymore.

Helm can do that for us.

-: Let us now understand the helm concepts.

So back to our WordPress application.

We have now discussed about the challenges.

Let us now see how Helm solves these challenges

using charts.

Here are the yaml files that we plan to use.

We have the deployment dot yaml

the secret dot yaml tv dot yaml, PVC dot yaml

and the service dot yaml.

Now each with its own definition to deploy a component

of the WordPress application on Kubernetes.

Now we know that some of these have values that might change

between different environments.

Well, users may prefer to use a different version

of the WordPress image that is used to

deploy the WordPress application or different size of disc.

And of course, the WordPress admin password is

going to be different as well.

So the first step is to convert these files

into templates where these values become variables.

The two curly braces indicates that these are variables

and that the values specified within our variable names

which will be used to fetch these values from another place.

So what is that other place where these values

are fetched from?

These values are stored in a file named Values dot yaml.

These, this file has the image storage

and password encoded variables defined

with the values We want these to have.

This way anyone who wants to deploy this application

can customize their deployment by simply changing the values

from the single file called Values dot yaml.

So a combination of templates plus values dot yaml gives us

the final version of definition files that can be used to

deploy the application on the Kubernetes cluster.

Together the templates and the values file forms

a helm chart.

A single helm chart may be used to deploy a simple

application like WordPress in our example

and it'll have all the necessary template files

for different services as well as the values file

with the variables.

It also has a chart dot yaml file that has information

about the chart itself, such as the name of the chart

the charts version, a description of what the chart is

some keywords associated with the application

and information about the ERs.

Now you can create your own chart for your own application

or you can explore existing charts from the artifact

hub@artifacthub.io and look for charts uploaded

by other users.

This hub is called as a repository that stores helm charts.

As of this recording, there are over 5,700 charts available.

Search for a chart for the application you're trying to

deploy. You

-: Can either search using this web interface

or you can search using the command helm search followed

by hub to indicate that you wish to search the Artifact Hub.

The artifact hub is the community driven chart repository

but there are other repositories as well

such as the Bit nami Helm repository.

To search for charts in other repositories

you must first add a repository to your local helm setup.

For this run the command Helm repo add to

add the bit NAMI repository

with the link to the Bitnami charts repository.

Then search the repository using the helm search

repo command instead of the search hub command.

And you can list existing repos using the helm

repo list command.

Now, once you find the chart

the next step is to install the chart on your cluster.

For this run, the helm install command followed

by a release name and the chart name.

Now, when this command is run

the helm chart package is downloaded

from the repository and extracted and installed locally.

So each installation of a chart is called a release

and each release has a release name.

So that's the release name that you specify

within the helm installed command.

For example, you can install the same application

using the same chart multiple times on a Kubernetes cluster

by running the helm Install command multiple times

and each time you run the helm install command.

A release is created

and each release is completely independent of each other.

Let's proceed to some additional helm commands.

To list installed packages, run the helm list command

to uninstall packages, run the helm uninstall command.

Now, we saw

that we could use the helm install command to download

and install a helm chart, but if we only need to download it

and not install it, then run the helm pool command.

Now use the dash UN option

because the chart is normally downloaded

in a TAR archive format.

The Antar option will extract its contents

after downloading it.

Now once extracted, you can find the contents of the chart.

When you list the files under the extracted folder

it's gonna be in the name, same name of the chart

and you may open and edit the values dot yaml

file to change any values that if required.

Now, once the changes are made

install the local chart using the Helm Install Command

but by specifying the path to that particular directory.

So that is all that we will be discussing

about Helm in this course.

There's so much to learn about Helm

so it requires an entire course of its own.

So we are working on an entire course on Helm for beginners

and we'll be releasing that soon.

So watch out for that.

From an exam perspective

we have already covered what is required for you to know.

So head over to the labs and practice working on help.