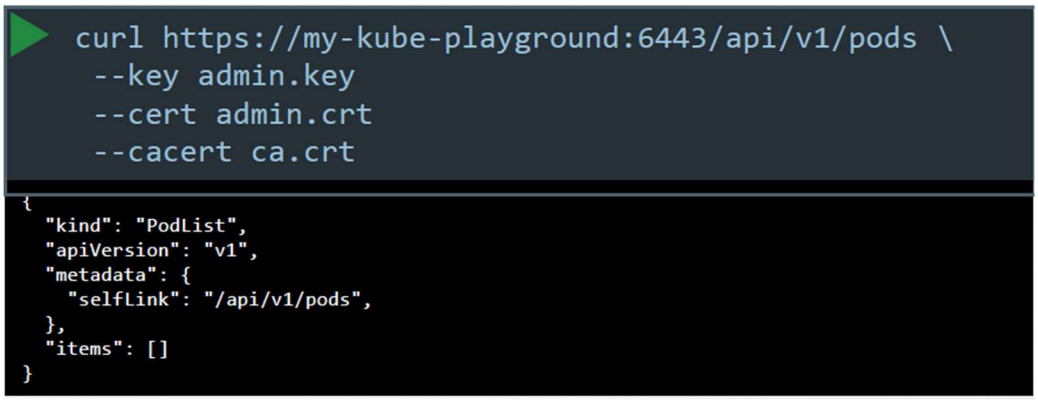
KUBECONFIG

So far, we have seen how to generate a certificate for a user. You've seen how a client uses the certificate file and key to query the Kubernetes REST API for a list of pods using cURL.

So send a cURL request to the address of the kube-apiserver while passing in the bearer files, along with the CA certificate as options. Like below



This is then validated by the API server to authenticate the user.

Now, how do you do that while using the kubectl command? Will 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 this information to a configuration file called as kubeconfig. And then specify this files 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.

$HOME/.kube/config

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.

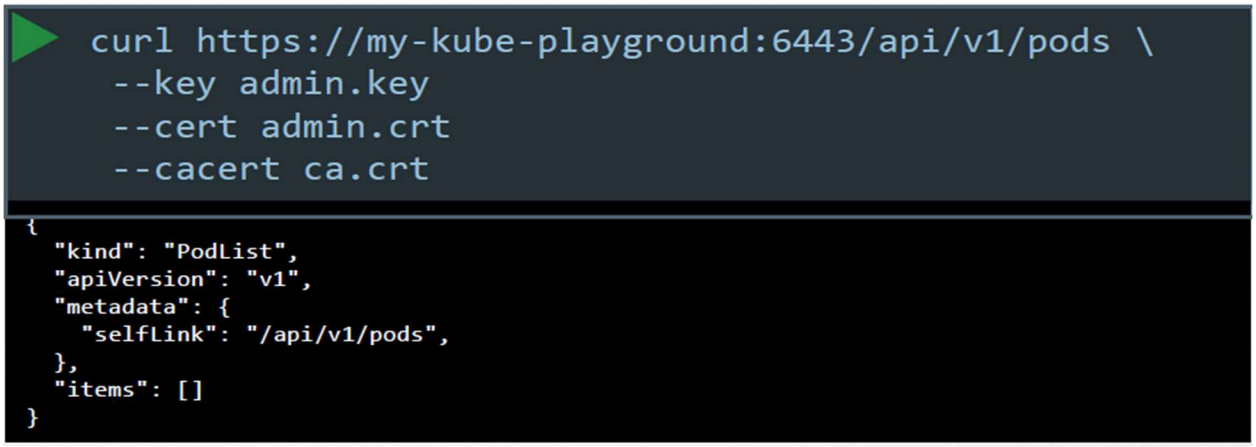
Users are the user accounts with which you have access to these clusters. For example, the admin user, a dev user, a prod user, et cetera. These users may have different privileges on different clusters.

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. For that you specify like **dev@Google**

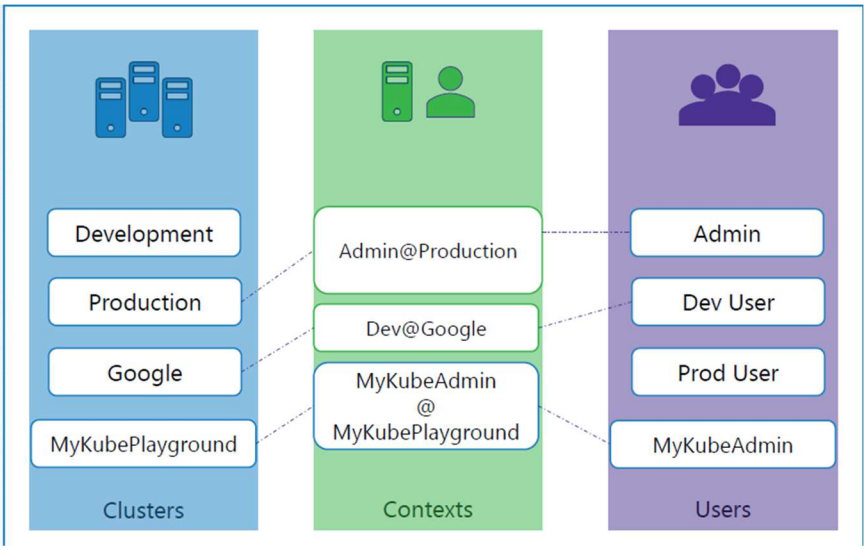
Remember, you're not creating any new users or configuring any kind of user access or authorization in the cluster with this process. You're using existing users with their existing privileges and defining what user you're going to use to access what cluster.

Like this 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 cluster section. The admin user's keys and certificates goes into the user section. You then create a context that specifies to use the my kube admin user to access the my kube playground cluster.



```
--server my-kube-playground:6443
--client-key admin.key
--client-certificate admin.crt
--certificate-authority ca.crt
```



Let's look at a real kubeconfig file now. The kubeconfig file is in a YAML format. it has three sections as we discussed: one for clusters, one for contexts and one for users. Each of these is an array format. That way you can specify multiple clusters, users or contexts within the same file.

```yaml
apiVersion: v1
kind: Config

clusters:
- name: my-kube-playground
  cluster:
    certificate-authority: ca.crt
    server: https://my-kube-playground:6443


contexts:
- name: my-kube-admin@my-kube-playground
  context:
    cluster:
    user:


users:
- name: my-kube-admin
  user:
    client-certificate: admin.crt
    client-key: admin.key
```

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 user's 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 context section to link the two together. We will name the context **my-kube-admin@my-kube-playground**.

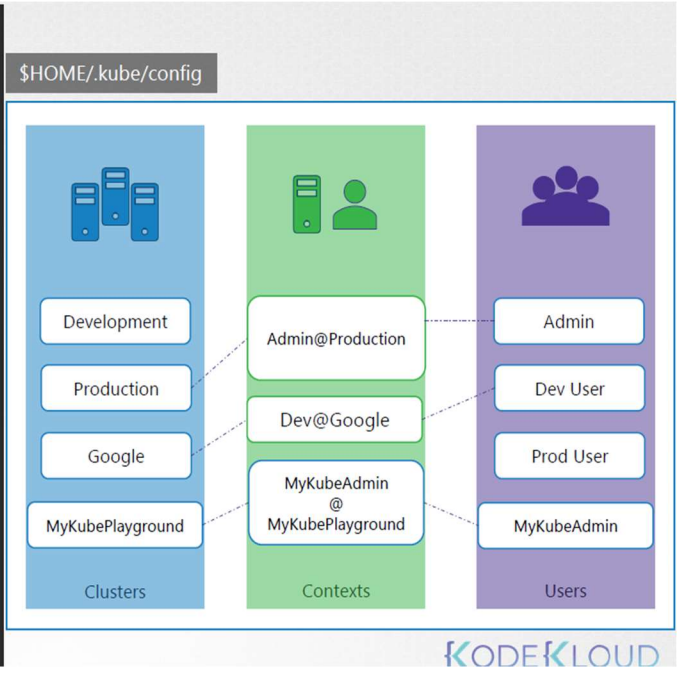
We will then specify the same name we used for cluster and user.

Follow the same procedure to add all the clusters you daily access, the user credentials you use to access them, as well as the context.

Once the file is ready, remember, 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 used 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

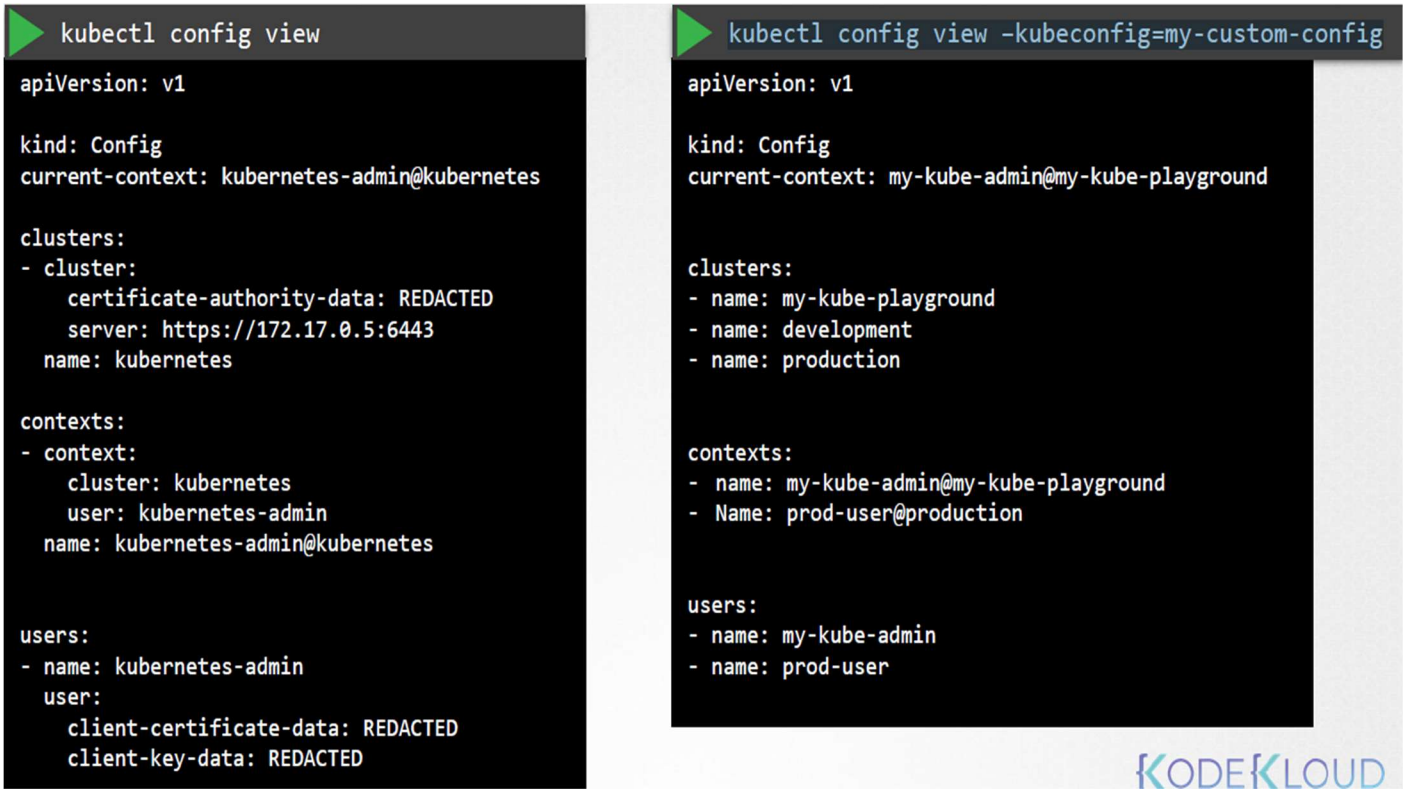
```
apiVersion: v1
kind: Config
current-context: dev-user@google

clusters:
- name: my-kube-playground   (values hidden…)
- name: development
- name: production
- name: google

contexts:
- name: my-kube-admin@my-kube-playground
- name: dev-user@google
- name: prod-user@production

users:
- name: my-kube-admin
- name: admin
- name: dev-user
- name: prod-user
```



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 cluster's contexts and users, as well as the current context that is set.

If you do not specify which kubeconfig file to use, it ends up using the default file located in the **$HOME/.kube/config.** Alternatively, you can specify a kubeconfig file by passing the kubeconfig option in the command line like below

$ kubectlconfig view –kubeconfig=my-custom-config



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? To do this run below command, then changes can be seen in the current context field in the file.

$kubectl config use-context prod-user@production

```
kubectl config use-context prod-user@production
```

```
apiVersion: v1

kind: Config
current-context: prod-user@production


clusters:
- name: my-kube-playground
- name: development
- name: production


contexts:
-  name: my-kube-admin@my-kube-playground
-  Name: prod-user@production


users:
- name: my-kube-admin
- name: prod-user
```

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.

```
kubectl config -h
```

```
Available Commands:
  current-context Displays the current-context
  delete-cluster  Delete the specified cluster from the kubeconfig
  delete-context  Delete the specified context from the kubeconfig
  get-clusters    Display clusters defined in the kubeconfig
  get-contexts    Describe one or many contexts
  rename-context  Renames a context from the kubeconfig file.
  set             Sets an individual value in a kubeconfig file
  set-cluster     Sets a cluster entry in kubeconfig
  set-context     Sets a context entry in kubeconfig
  set-credentials Sets a user entry in kubeconfig
  unset           Unsets an individual value in a kubeconfig file
  use-context     Sets the current-context in a kubeconfig file
  view            Display merged kubeconfig settings or a specified kubeconfig file
```

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.

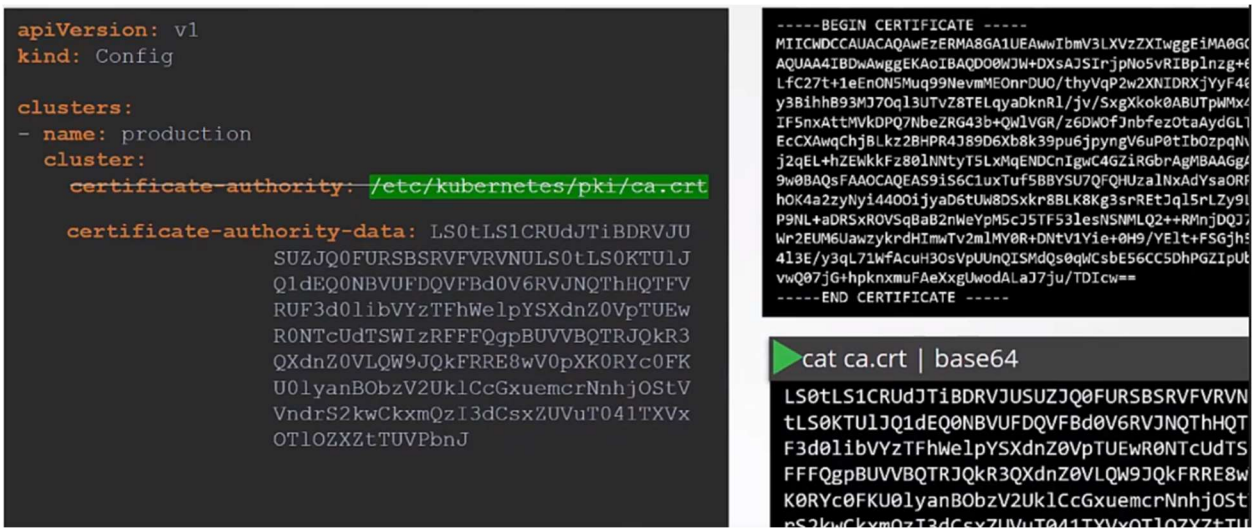
Finally, a word on certificates. You have seen paths to certificate files mentioned in kubeconfig like below

```
apiVersion: v1
kind: Config

clusters:
- name: production
  cluster:
    certificate-authority: ca.crt
    server: https://172.17.0.51:6443

contexts:
- name: admin@production
  context:
    cluster: production
    user: admin
    namespace: finance

users:
- name: admin
  user:
    client-certificate: admin.crt
    client-key: admin.key
```

Well, it's better to use the full path

```
client-certificate: /etc/kubernetes/pki/users/admin.crt
client-key: /etc/kubernetes/pki/users/admin.key
```

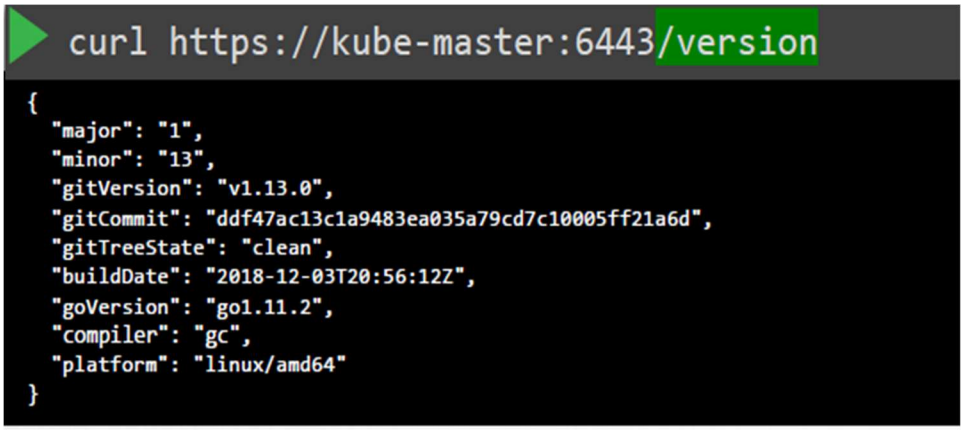
But remember, there's also another way to specify the certificate credentials. Instead of using certificate-authority field and the path to the file, you may optionally use the **certificate-authority-data** field and provide the contents of the certificate itself but not the file as is. Convert the contents to a Base64 encoded format and then pass that in.



API GROUPS

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've 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.



Similarly, to get a list of pods you would access the URL api/v1/pods.

```
curl https://kube-master:6443/api/v1/pods
```

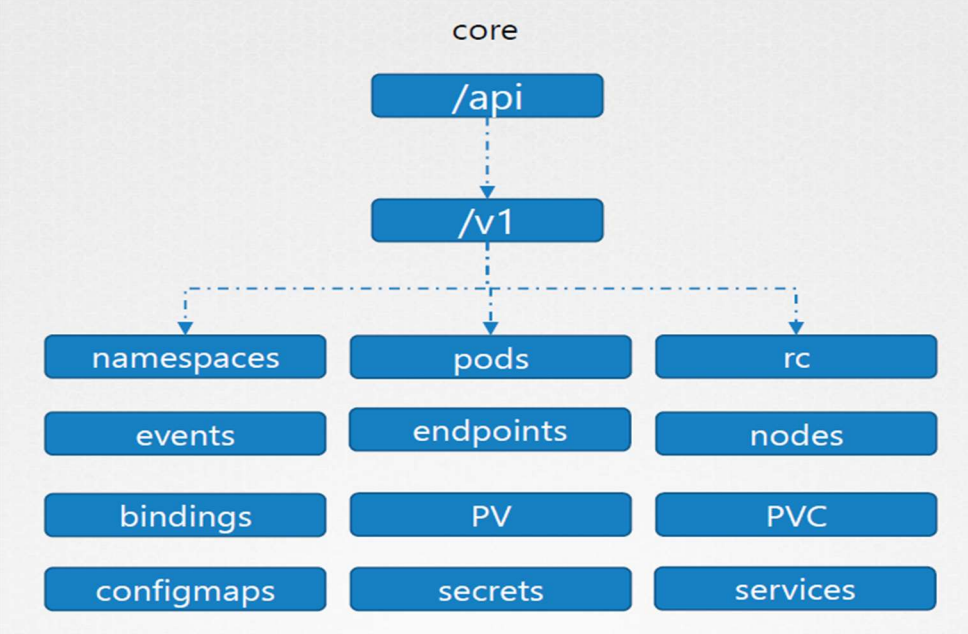

```
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/pods",
    "resourceVersion": "153068"
  },
  "items": [
    {
      "metadata": {
        "name": "nginx-5c7588df-ghsbd",
        "generateName": "nginx-5c7588df-",
        "namespace": "default",
        "creationTimestamp": "2019-03-20T10:57:48Z",
        "labels": {
          "app": "nginx",
          "pod-template-hash": "5c7588df"
        },
        "ownerReferences": [
          {
            "apiVersion": "apps/v1",
            "kind": "ReplicaSet",
            "name": "nginx-5c7588df",
            "uid": "398ce179-4af9-11e9-beb6-020d3114c7a7",
            "controller": true,
            "blockOwnerDeletion": true
          }
        ]
      },
```


Our focus in this lecture is about these API parts, 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 etc.

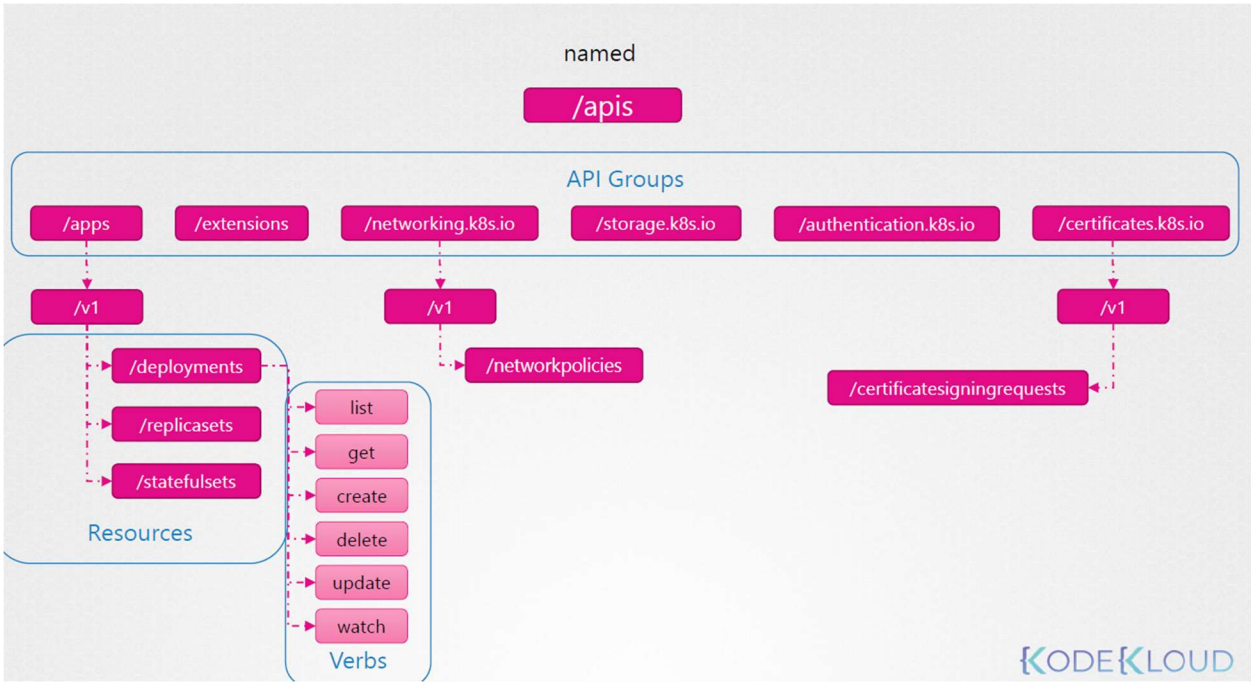
The version API is for viewing the version of the cluster, as we just saw. The Metrics and Health API are used to monitor the health of the cluster. The logs are used for integrating with third-party logging applications.

Here, 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 name, spaces, pods, replication controllers, events and points, nodes, bindings, persistent volumes, persistent volume claims, conflict maps, secrets, services, etc.



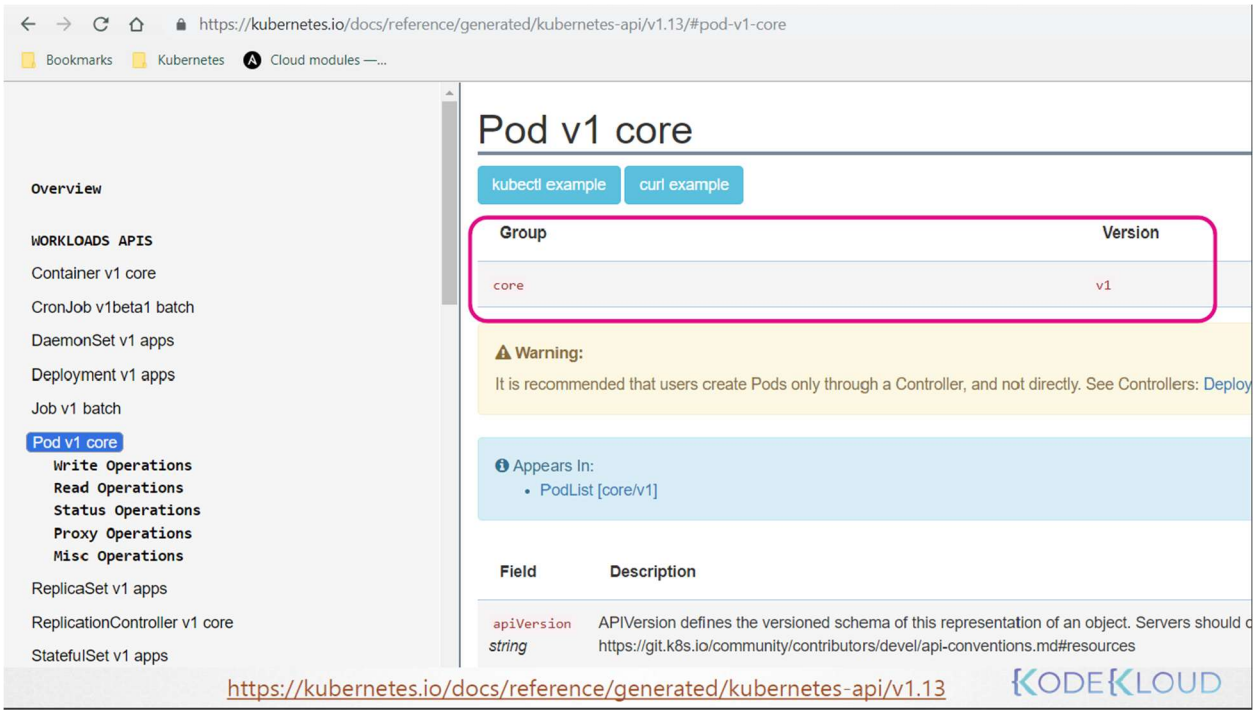
The named group APIs are more organized and going forward, all the newer features are going to be made available through these named groups.



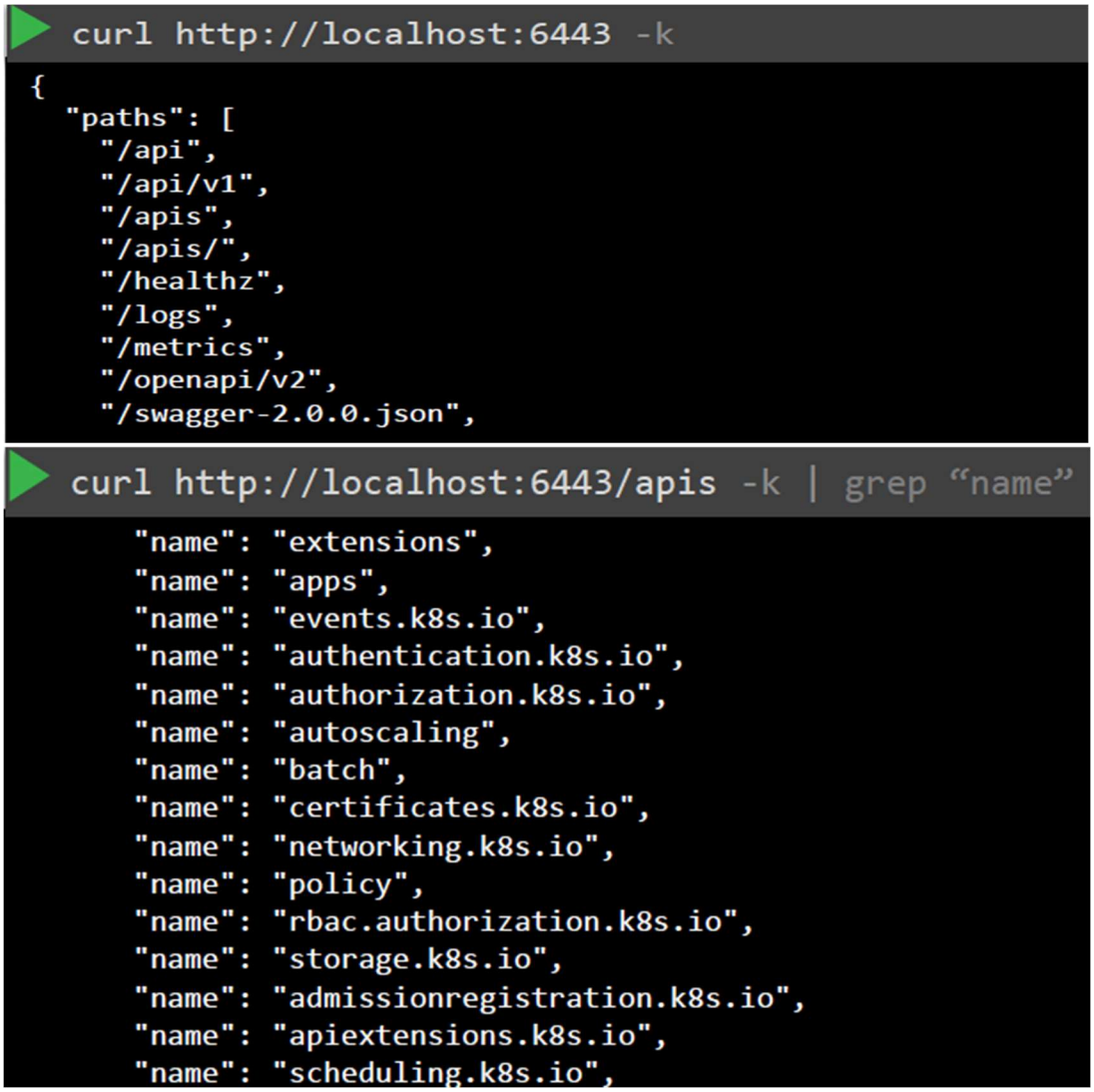
It has groups under it for apps, extensions, networking, storage, authentication, authorization, etc. Shown here are just a few. Within apps, you have deployments, replica sets, stateful sets. Within networking, you have network policies. Certificates have these certificate signing requests that we talked about earlier.

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 it's group details, v1 core is just v1. You can also view these on your Kubernetes cluster.



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.

```
curl http://localhost:6443 -k
```

```
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {

  },
  "status": "Failure",
  "message": "forbidden: User \"system:anonymous\" cannot get path \"/\"",
  "reason": "Forbidden",
  "details": {

  },
  "code": 403
}
```

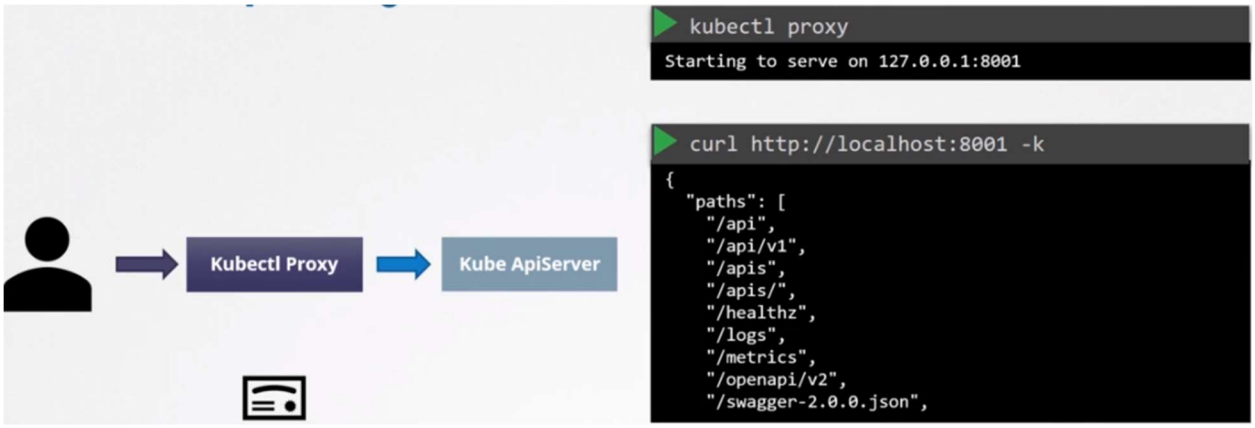
So you have to authenticate to the API using your certificate files by passing them in the command line like below.

```
curl http://localhost:6443 -k
     --key admin.key
     --cert admin.crt
     --cacert ca.crt
```

```
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/",
    "/healthz",
    "/logs",
```

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.



Kube control proxy is an ACTP proxy service created by Kube control utility to access the Kube API server

Finally, 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.

Authorization

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.

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 nodes and deployments, creating or deleting objects such as adding or deleting pods or even nodes 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 usernames and passwords or tokens, or signed TL certificates or service accounts.

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 nodes 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 ☺

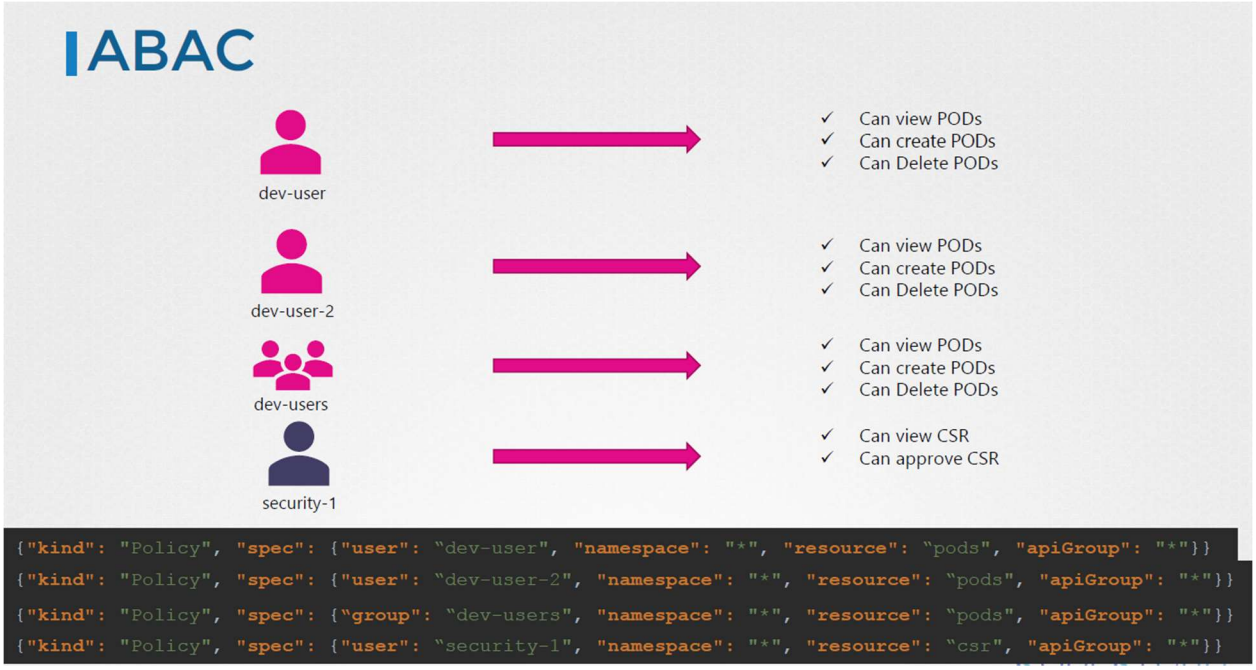
There is different authorization mechanism supported by Kubernetes, such as node authorization, attribute-based authorization, role-based authorization and webhook. Let's just go through these now.

We know that the Kube API Server is accessed by users like us for management purposes, as well as the kubelets on node within the cluster for management process 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 privilege is required for a kubelet.

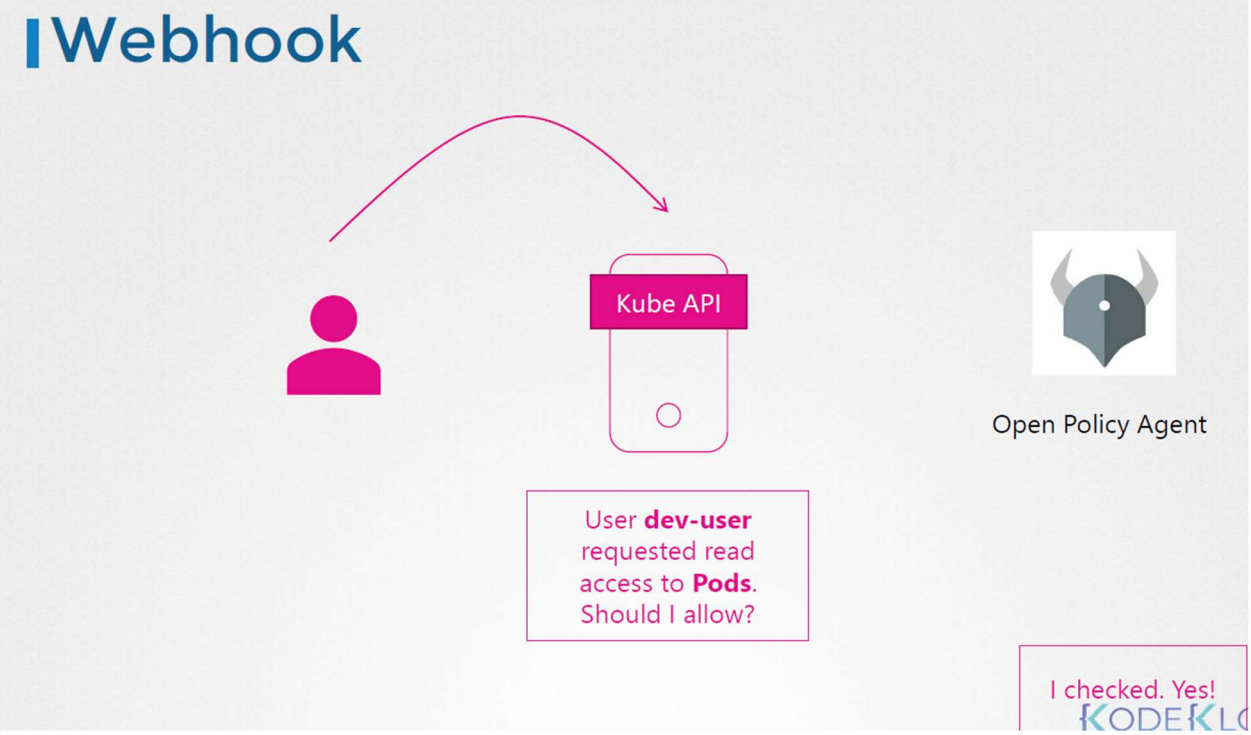
So **Node Authorizer is** access within the cluster.

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 json format show below, 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 **attribute-based access control** configurations are difficult to manage.

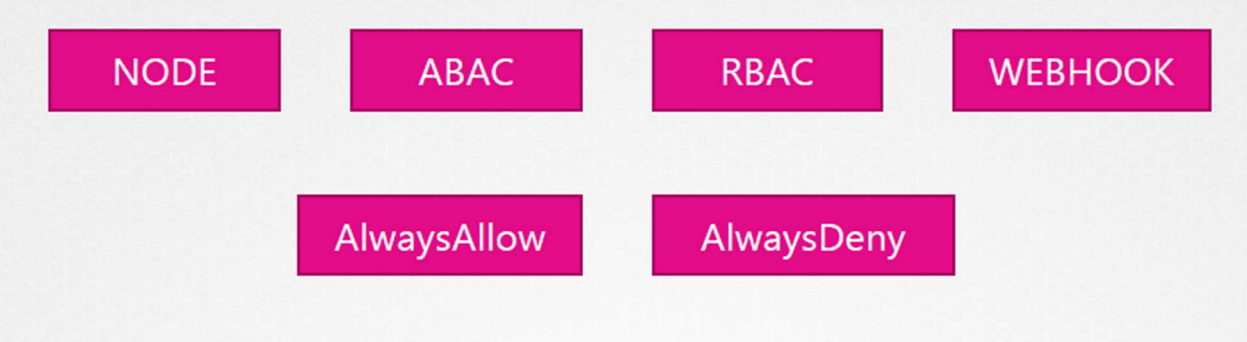


Role-based access controls make these much easier. With role-based access controls, instead of directly associating a user or a group with a set of permissions, we define a role, in this case for developers. We create a role with the set of permissions required for developers then we associate all the 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.

Now, what if you want to outsource all the authorization mechanisms? Say you want to manage 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.



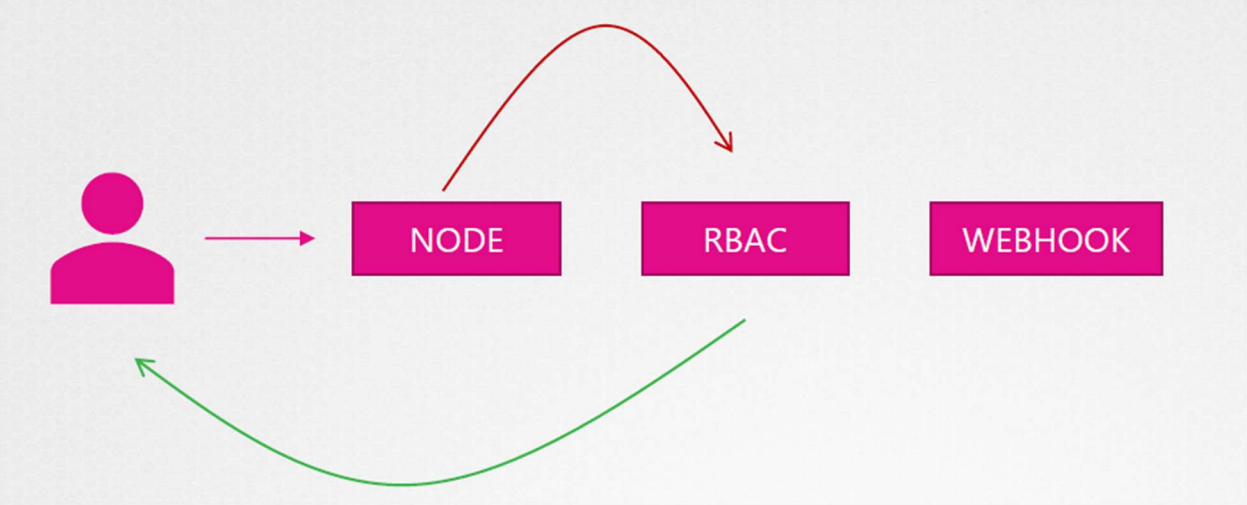
So, where do you configure these modes? Which of them are active by default? Can you have more than one at a time? How does authorization work if you do have multiple ones configured?

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, rbac and webhook. When you have multiple modes configured your request is authorized using each one in the order it is specified.

```
ExecStart=/usr/local/bin/kube-apiserver \\
  --advertise-address=${INTERNAL_IP} \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --authorization-mode=Node,RBAC,Webhook \\
  --bind-address=0.0.0.0 \\
  --enable-swagger-ui=true \\
  --etcd-cafile=/var/lib/kubernetes/ca.pem \\
  --etcd-certfile=/var/lib/kubernetes/apiserver-etcd-client.crt \\
  --etcd-keyfile=/var/lib/kubernetes/apiserver-etcd-client.key \\
  --etcd-servers=https://127.0.0.1:2379 \\
  --event-ttl=1h \\
  --kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \\
  --kubelet-client-certificate=/var/lib/kubernetes/apiserver-etcd-client.crt \\
  --kubelet-client-key=/var/lib/kubernetes/apiserver-etcd-client.key \\
  --service-node-port-range=30000-32767 \\
  --client-ca-file=/var/lib/kubernetes/ca.pem \\
  --tls-cert-file=/var/lib/kubernetes/apiserver.crt \\
  --tls-private-key-file=/var/lib/kubernetes/apiserver.key \\
  --v=2
```

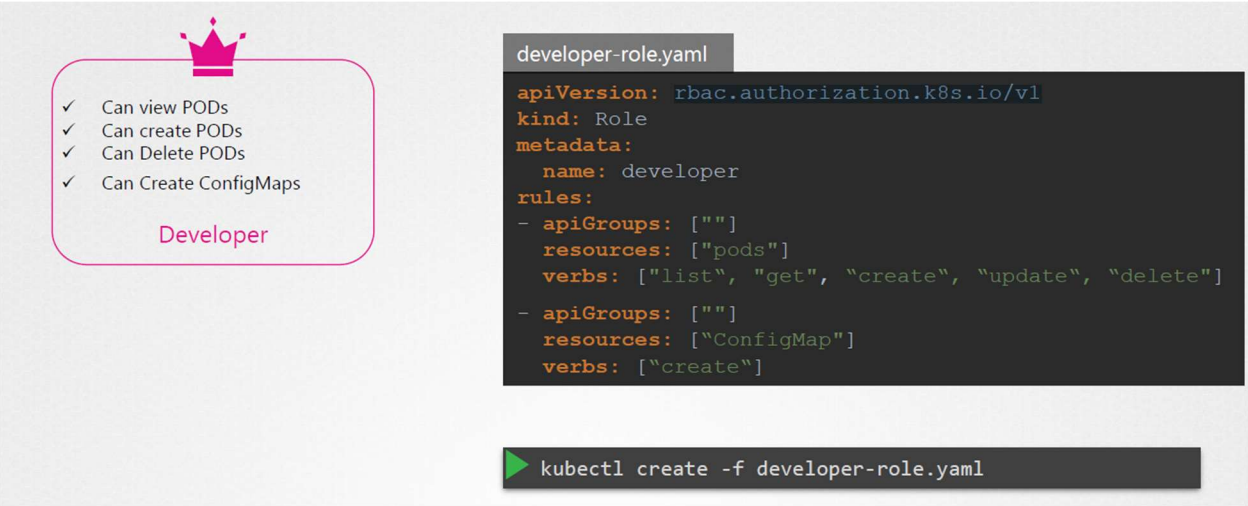
For example, when a user sends a request it's first handled by the Node Authorizer. The Node Authorizer handles only node requests, so it denies the request. Whenever a module denies a request it is forwarded to the next one in the chain. The role-based access control module performs its checks and grants 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.



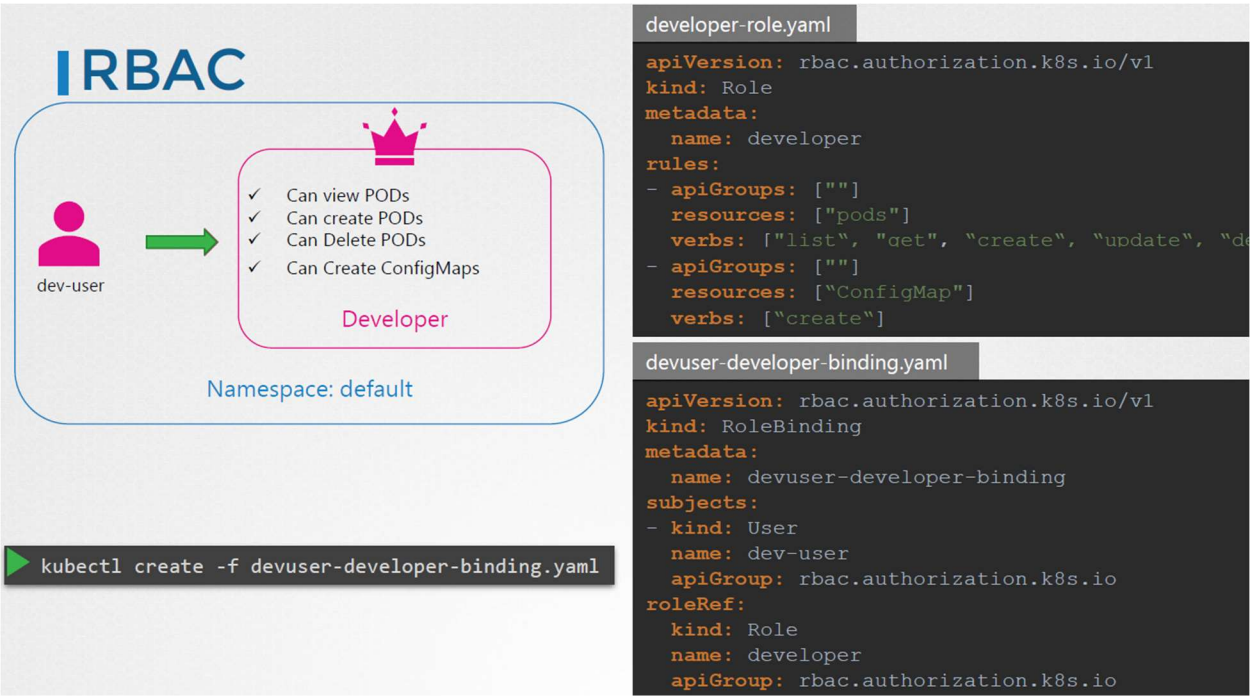
RBAC

We 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 **rbac.authorization.k8s.io/v1** and kind said 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 cube control create roll command, shown below.



The next step is to link the user to that role. For this, we create another object called **roll binding**. The roll 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 **subjects** is where we specify the user details. The **roleRef** section is where we provide the details of the roll we created. Create the role binding using the cube 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 pot and convict maps within the default name space. If you want to limit the dev user's access within a different namespace, then specify the name space within the metadata of the definition file while creating them.

To view the created roles, run the **kubectl get roles** command. To list role bindings, run **kubectl get rolebindings** command. To view more details about the role, run the **kubectl describe role** command. Here you see the details about the resources and permissions for each resource. Similarly, to view details about role bindings run the **kubectl describe rolebindings** command. Here you can see details about an existing role binding.

```
kubectl get roles
NAME        AGE
developer   4s
```

```
kubectl get rolebindings
NAME                        AGE
devuser-developer-binding   24s
```

```
kubectl describe role developer
Name:         developer
Labels:       <none>
Annotations:  <none>
PolicyRule:
  Resources  Non-Resource URLs  Resource Names  Verbs
  ---------  -----------------  --------------  -----
  ConfigMap  []                 []              [create]
  pods       []                 []              [get watch list create delete]
```

```
kubectl describe rolebinding devuser-developer-binding
Name:         devuser-developer-binding
Labels:       <none>
Annotations:  <none>
Role:
  Kind:  Role
  Name:  developer
Subjects:
  Kind  Name      Namespace
  ----  ----      ---------
  User  dev-user
```

CHECK ACESS

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 **kubectl auth can-i** command and check if you can say, create deployments, or say delete nodes.

```
kubectl auth can-i create deployments
yes
```

```
kubectl auth can-i delete nodes
no
```

If you're an administrator, then you can even impersonate another user to check their permission. For instance, say you were tasked to create 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 below.

```
kubectl auth can-i create deployments --as dev-user
no
```

You can also specify the name space in the command

```
kubectl auth can-i create pods --as dev-user --namespace test
no
```

The dev-user does not have permission to create a pod in the test name space.

Well, 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 namespace you want to 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 rule shown below.

Cluster roles

When we talked about roles and role bindings, we said that roles and role bindings are namespaced, meaning they are created within namespaces. If you don't specify a namespace, they are created in the default namespace and control access within that namespace alone.

We discussed about namespaces 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 a namespace? Like can you say node 01 is part of the dev namespace? No, those are cluster-wide or cluster-scoped resources. They cannot be associated to any particular namespace.

So the resources are categorized as either **namespaced or cluster-scoped**. Now we have seen a lot of namespaced 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 namespace you specify when you created them. If you don't specify a namespace, they are created in the default namespace. To view them, or delete them, or update them, you always specify the right namespace.

The cluster-scoped resources are those where you don't specify a namespace when you create them, like nodes, persistent volumes, cluster roles and cluster role bindings, certificate signing request we saw earlier, and namespace objects themselves, of course, not namespaced.

To see a full list of namespaced and non-namespaced resources, run the kubectl API resources command with the namespaced option set.

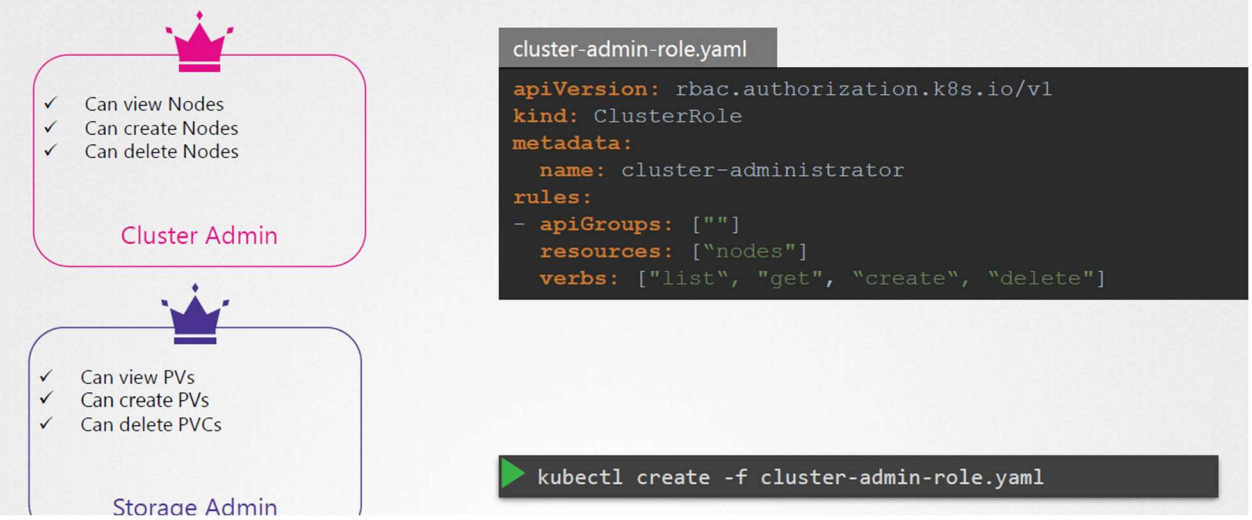
```
kubectl api-resources --namespaced=true
```

```
kubectl api-resources --namespaced=false
```

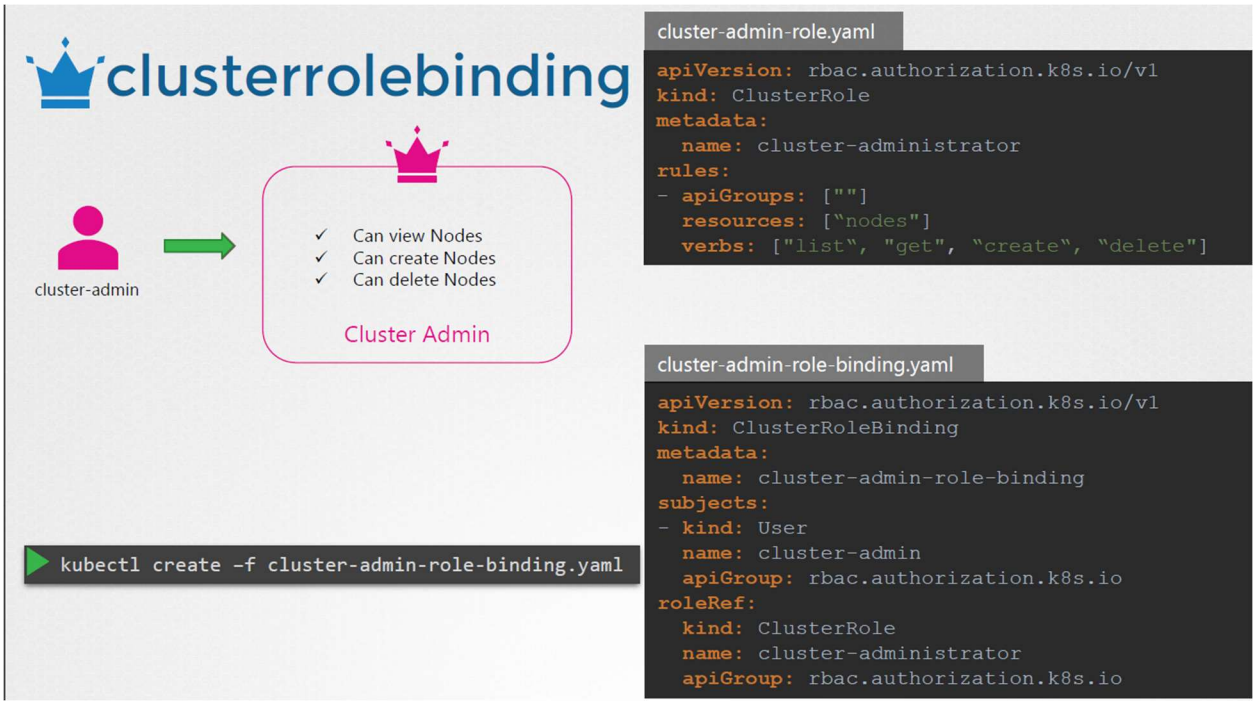
In the previous lecture, we saw how to authorize a user to namespace resources. We used roles and role bindings for that. But how do we authorize users to cluster-wide resources like nodes or persistent volumes. That is where you use cluster roles and cluster role bindings. Cluster roles are just like roles, except they are for cluster-scoped resources.

For example, a cluster admin role can be created to provide a cluster administrator permission 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 and claims.

Create a cluster role definition file with the kind **clusterRole** 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 **ClusterRoleBinding**. Under **subjects**, we specify the user details, **cluster-admin** user in this case. The **roleRef** section is where we provide the details of the cluster role we created. Create the role binding using the kubectl create command.



One thing to note before I let you go. We said that cluster roles and binding are used for cluster-scoped resources, but that is not a hard rule. You can create a cluster role for namespaced resources as well. When you do that, the user will have access to these resources across all namespaces. Earlier, when we created a role to authorize a user to access pod, the user had access

to pods in a particular namespace alone. With cluster roles, when you authorize a user to access the pods, the user gets access to all pods across the cluster.

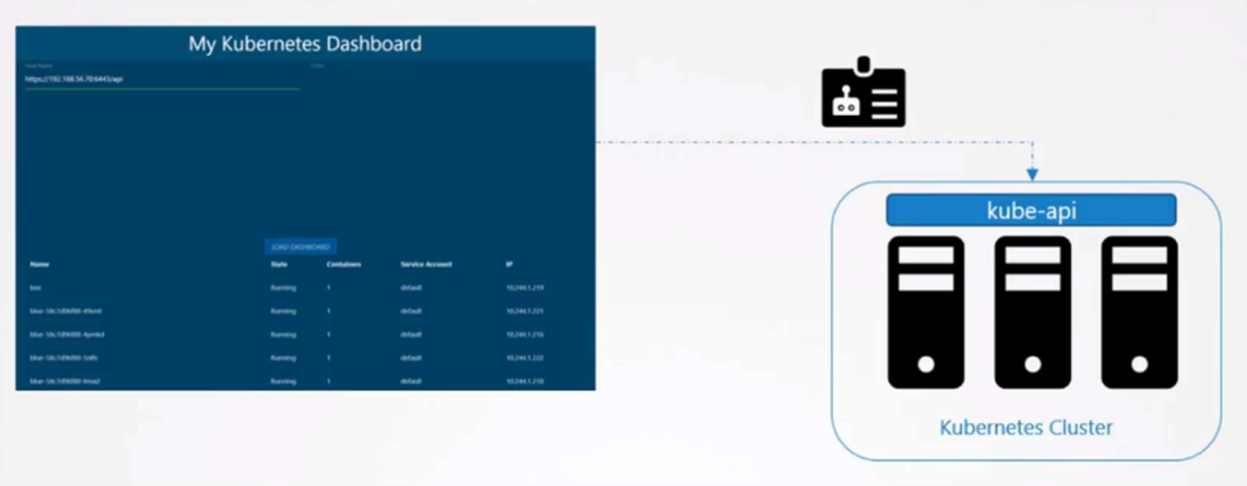
Service Account

So there are two types of accounts in Kubernetes, a user account and a service account. As you might already know, the user account is used by humans, and service accounts are used by machines.

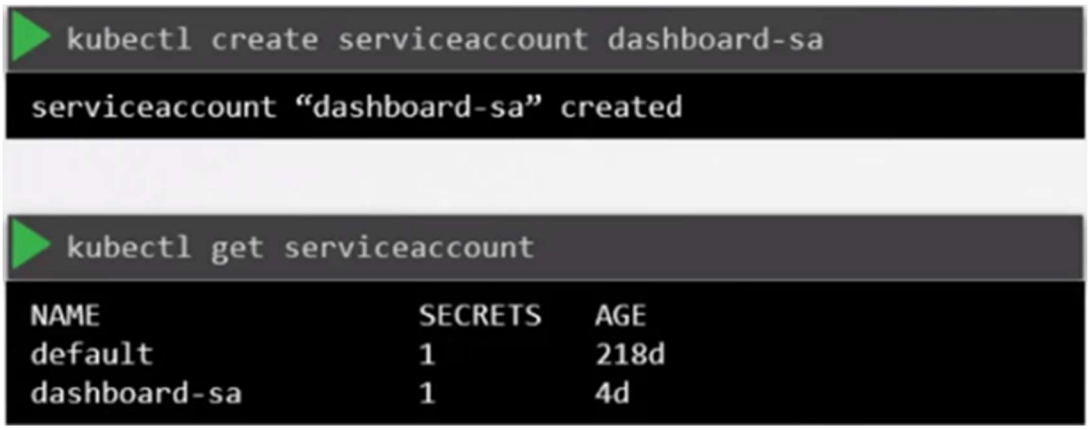
A user account could be for an administrator accessing the cluster to perform administrative tasks, or a developer accessing the cluster to deploy applications, etc.

A service account could be an account used by an application to interact with a Kubernetes cluster. For example, a monitoring application like Prometheus uses a service account to pull the Kubernetes API for performance metrics. An automated build tool like Jenkins uses service accounts to deploy applications on the Kubernetes cluster.

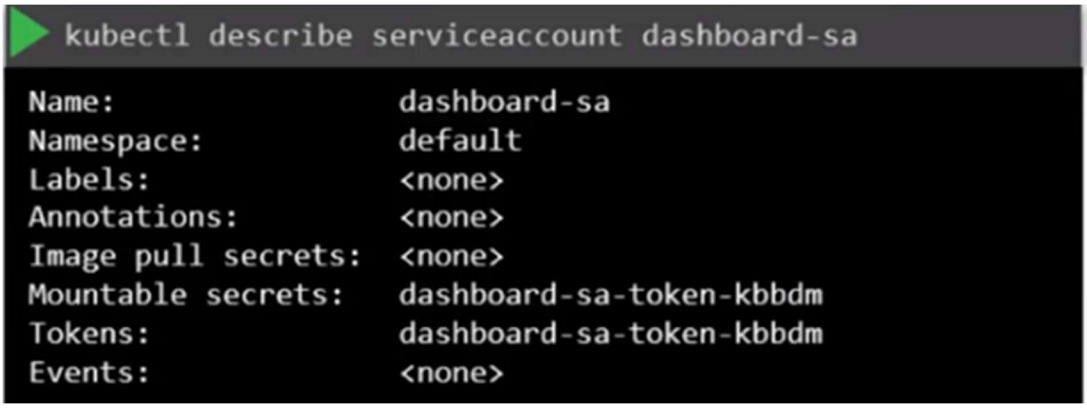
Let's take an example. I've built a simple Kubernetes dashboard application named "My Kubernetes Dashboard." It's a simple application built in Python and all that it does when deployed is retrieve the list of pods on a Kubernetes cluster by sending a request to the Kubernetes API, and displayed on a webpage. In order for my application to query the Kubernetes API, it has to be authenticated. For that, we use a service account.



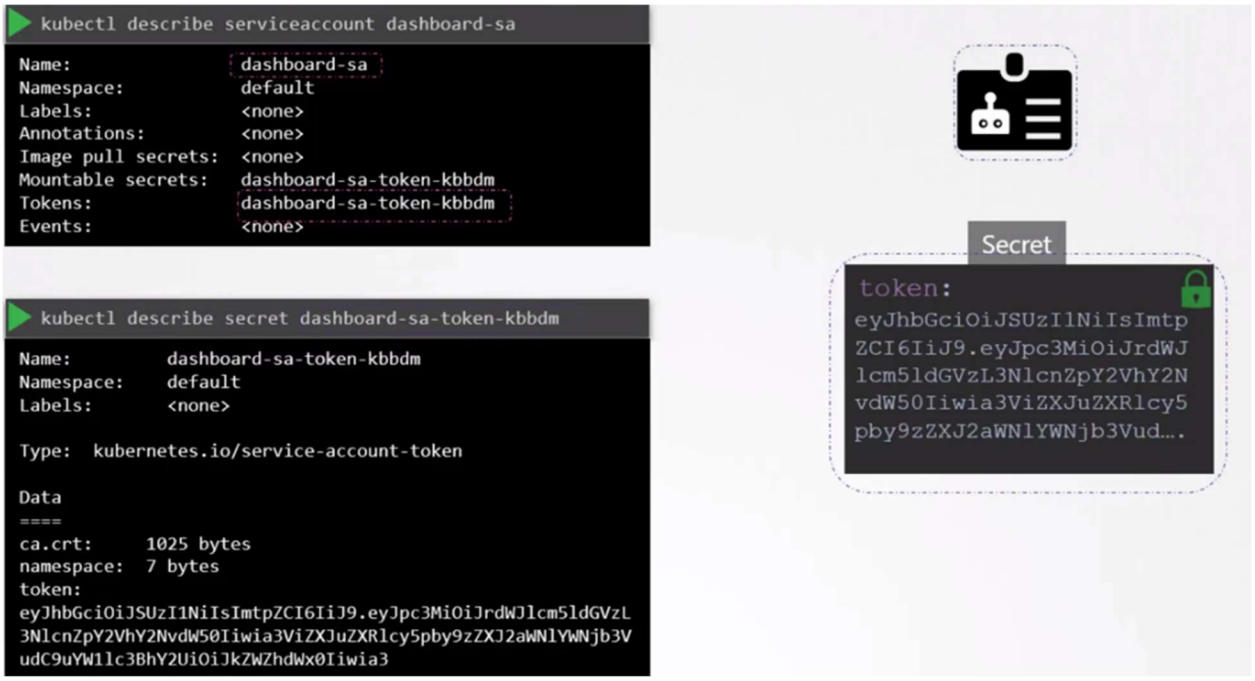
To create a service account, run the command **kubectl create serviceaccount**, followed by the account name, which is dashboard-sa in this case. To view the service accounts from the kubectl, get service account command. This will list all the service accounts.



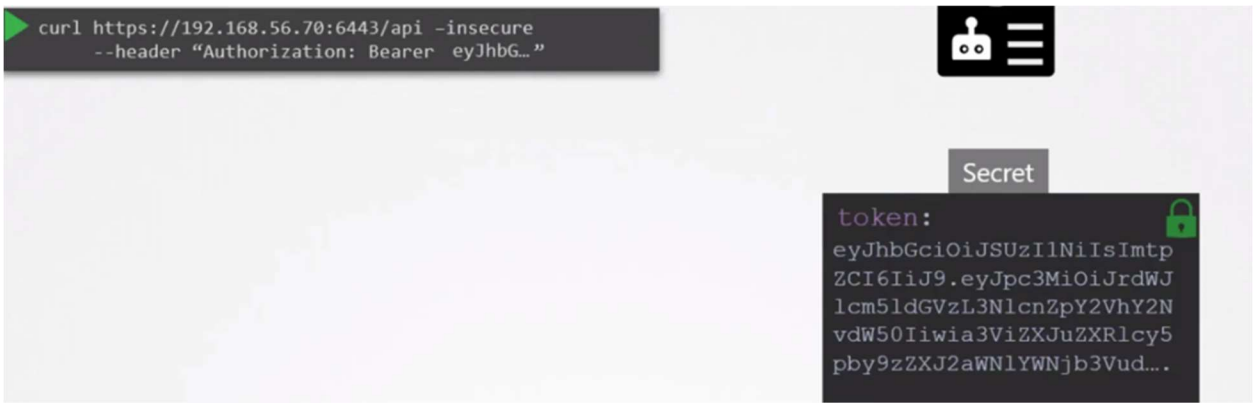
When the service account is created, it also creates a token automatically. The service account token is what must be used by the external application while authenticating to the Kubernetes API. **The token however, is stored as a secret object**. In this case, it's named dashboard-sa-token-kbbdm.



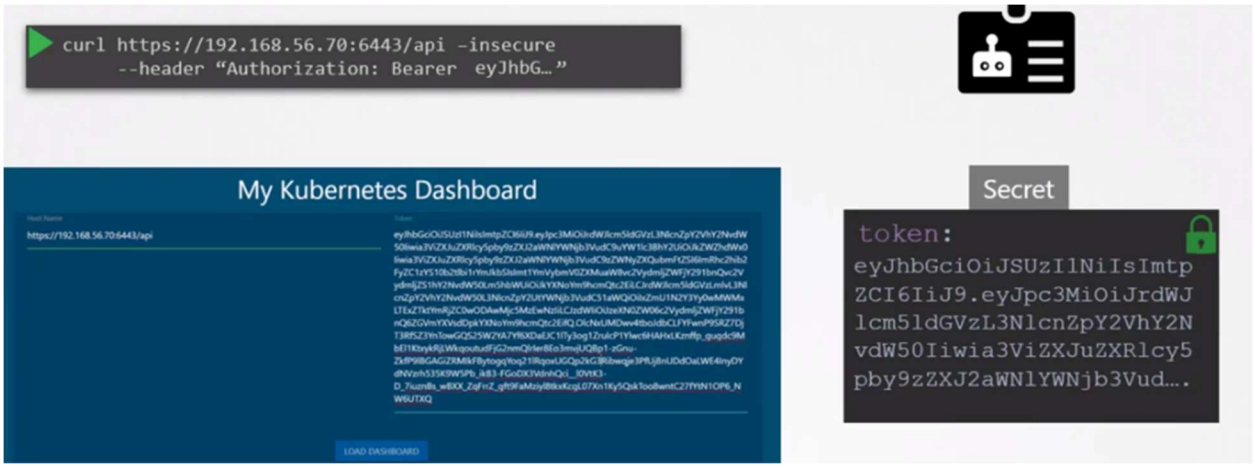
So when a service account is created, it first creates the service account object and then generates a token for the service account. It then creates a secret object and stores that token inside the secret object. The secret object is then linked to the service account. To view the token, view the secret object by running the command kubectl describe secret. This token can then be used as an authentication bearer token, while making a rest call to the Kubernetes API.



For example, in this simple example using curl, you could provide the bearer token as an authorization header while making a risk call to the Kubernetes API.

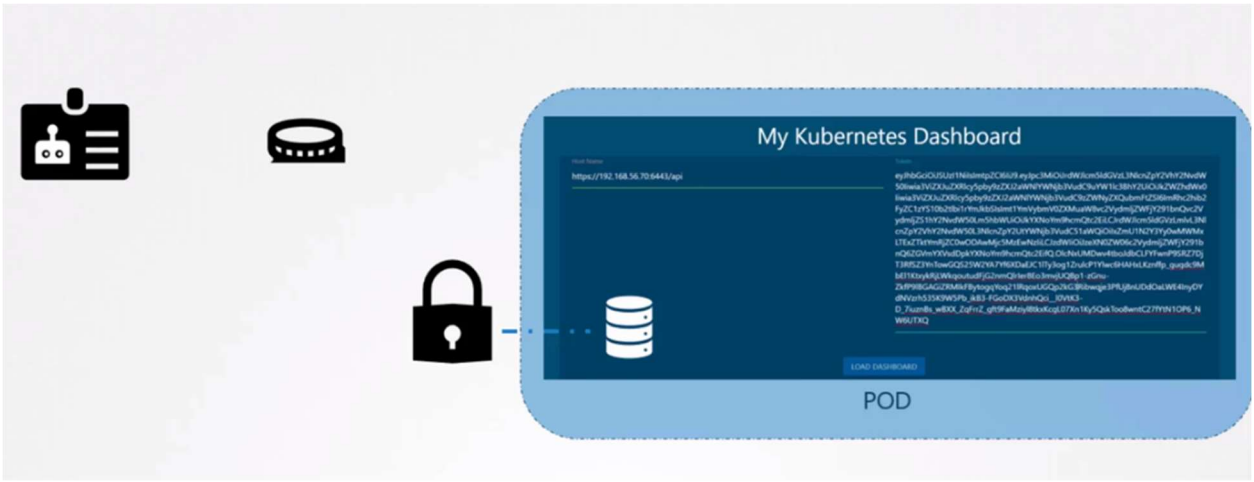


In case of my custom dashboard application, copy and paste the token into the tokens field to authenticate the dashboard application.



So that's how you create a new service account and use it. You can create a service account, assign the right permissions using role-based access control mechanisms, and export your service account tokens and use it to configure your third party application. to authenticate to the Kubernetes API.

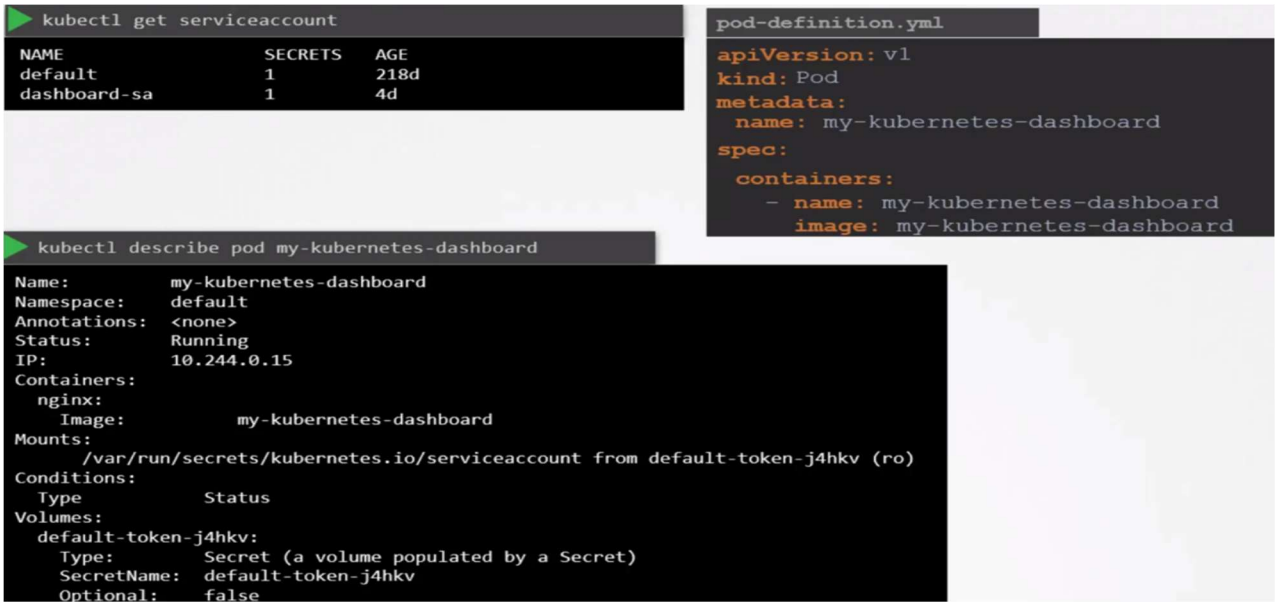
But what if your third party application is hosted on the Kubernetes cluster itself? For example, we can have our custom Kubernetes dashboard application or the Prometheus application, deployed on the Kubernetes cluster itself. In that case, this whole process of exporting the service account token and configuring the third party application to use it, can be made simple by automatically mounting the service token secret as a volume inside the pod hosting the third party application. That way the token to access the Kubernetes API is already placed inside the pod and can be easily read by the application. You don't have to provide it manually.



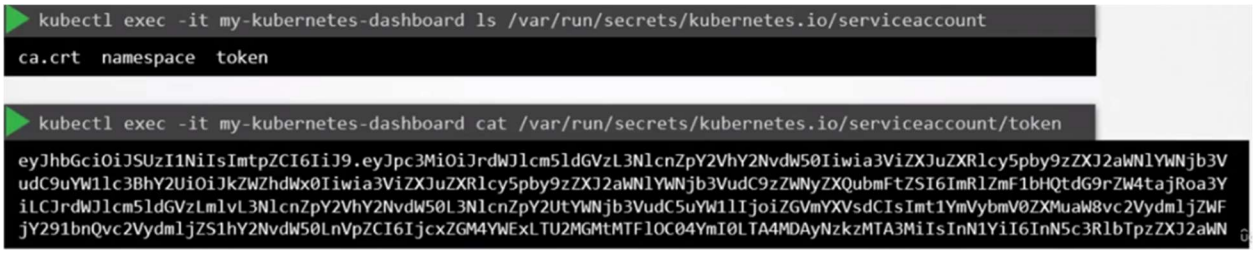
If you go back and look at the list of service accounts, you will see that there is a default service account that exists already. For every name space in Kubernetes, a service account named default is automatically created.

Each name space has its own default service account. Whenever a pod is created, the default service account and its token are automatically mounted to that pod as a volume mount.

For example, we have a simple pod definition file that creates a pod using my custom Kubernetes dashboard image. We haven't specified any secrets or volume mounts in the definition file. However, when the pod is created if you look at the details of the pod by running the kubectl describe pod command, you see that a volume is automatically created from the secret named default token, which is in fact the secret containing the token for this default service account.



The secret token is mounted at location var/run/secrets/kubernetes.io/serviceaccount inside the pod. So from inside the pod, if you run the LS command to list the contents of the directory, you will see the secret mounted as three separate files. The one with the actual token is the file named token. If you view contents of that file, you will see the token to be used for accessing the Kubernetes API.



Now, remember that the default service account is very much restricted. It only has permission to run basic Kubernetes API queries. If you'd like to use a different service account, such as the one

we just created, modify the pod definition file to include a service account field and specify the name of the new service account.

pod-definition.yml

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

Remember, you cannot edit the service account of an existing pod. You must delete and recreate the pod. However, in case of a deployment, you will be able to the service account as any changes to the pod definition file will automatically trigger a new rollout for the deployment. So the deployment will take care of deleting and recreating new pods with the right service account.

So remember, Kubernetes automatically mounts the default service account if you haven't explicitly specified any. may choose not to mount a service account automatically by setting the auto mount service account token field to false in the pods back section.

Service Account (1.22 and 1.24 changes)

As we discussed every name space has a default service account and that service account has a secret object with a token associated with it. When a pod is created, it automatically associates the service account to the pod and mounts the token to a well-known location within the pod. In this case, it's under a var/run/secrets/kubernetes.io/serviceaccount. This makes the token accessible to a process that's running within the pod and that enables that process to query the Kubernetes API.

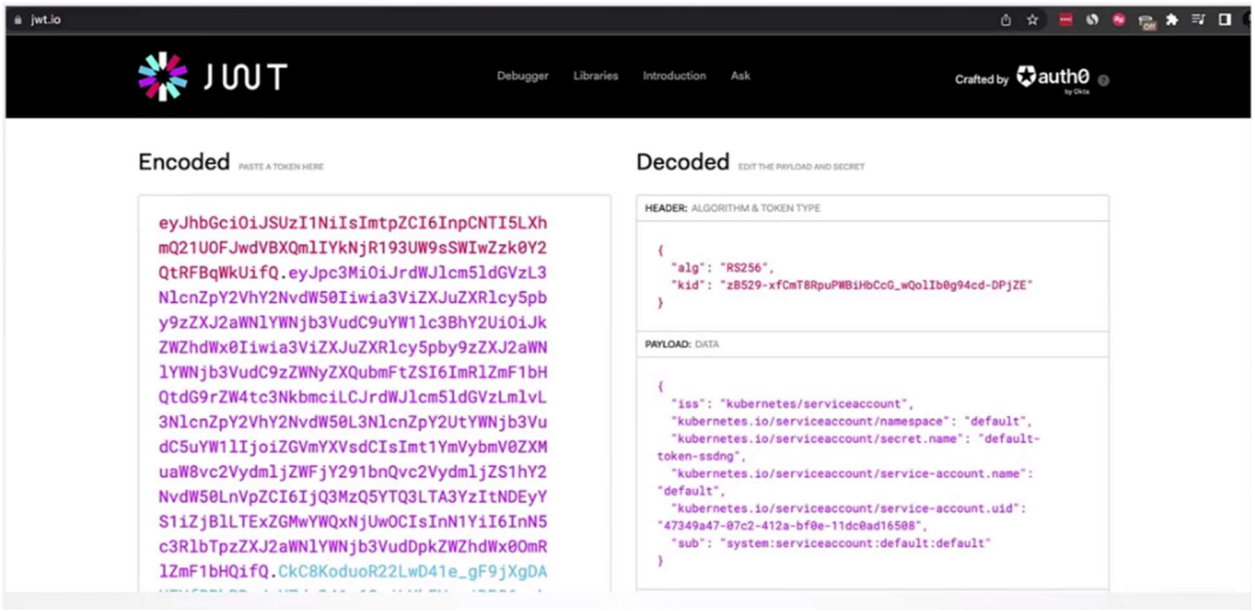
Now, if you list the contents of the directory inside the pod, you will see the secret mounted as three separate files. The one with the actual token is the file named token. So if you see the contents of that file, you'll see the token to be used for accessing the Kubernetes API.

So all of that remains same. This is exactly what we discussed in the previous.

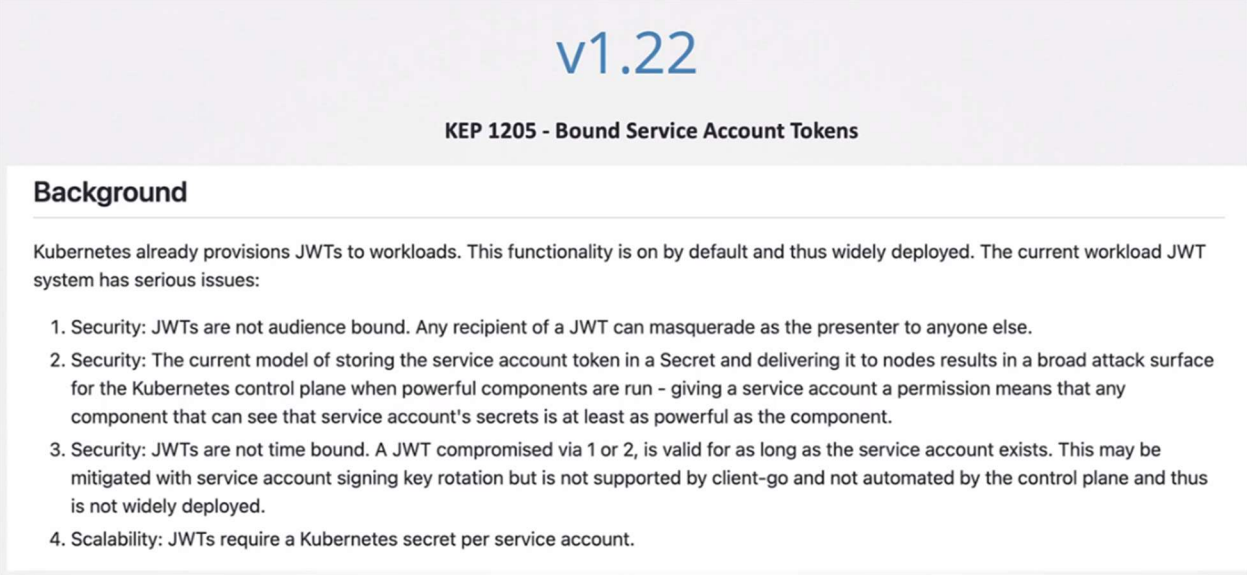
Now let's take that token that we just saw and if you decode this token using command or you could just copy and paste this token in the JWT website at jwt.io, you'll see that it has no expiry

date defined in the payload section here on the right. So this is a token that does not have an expiry date set.

```
jq -R 'split(".") | select(length > 0) | .[0],.[1] | @base64d | fromjson' <<< eyJhbGciOiJ...
{
  "iss": "kubernetes/serviceaccount",
  "kubernetes.io/serviceaccount/namespace": "default",
  "kubernetes.io/serviceaccount/secret.name": "default-token-ssdng",
  "kubernetes.io/serviceaccount/service-account.name": "default",
  "kubernetes.io/serviceaccount/service-account.uid": "47349a47-07c2-412a-bf0e-11dc0ad16508",
  "sub": "system:serviceaccount:default:default"
}
```



So this excerpt from the Kubernetes enhancement proposal for creating bound service account tokens describes this form of JWT to be having some security and scalability related issues. So the current implementation of JWT is not bound to any audience and is not time bound. As we just saw, there was no expiry date for the token. So the JWT is valid as long as the service account exists. Moreover, each JWT requires a separate secret object per service account, which results in scalability issues.

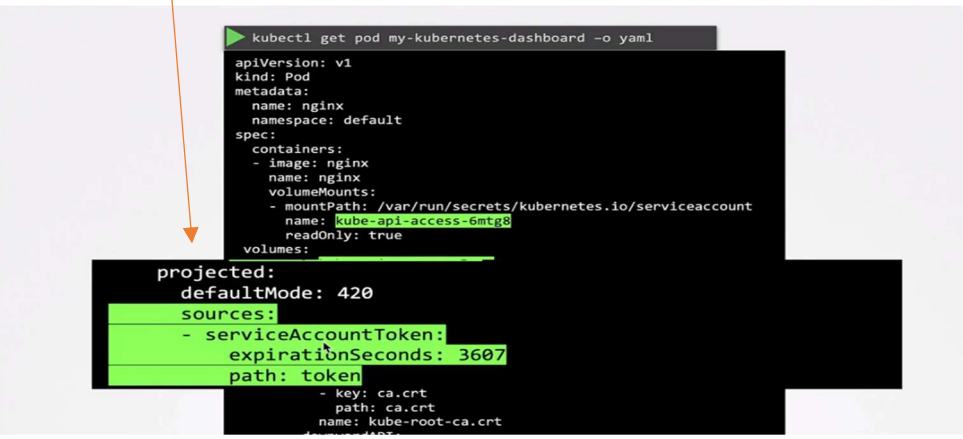


And as such, in version 1.22 the token request API was introduced as part of the Kubernetes enhancement proposal 1205, that aimed to introduce a mechanism for provisioning Kubernetes service account tokens that are more secure and scalable via an API.

So tokens generated by the token request API are audience bound, they're time bound and object bound, and hence are more secure.

Now since version 1.22, when a new pod is created it no longer relies on the secret token that we just saw. Instead, a token with a defined lifetime is generated through the token request API by the service account admission controller when the pod is created.

And this token is then mount as a projected volume onto the pod. So in the past, if you look at this space here you'd see as the secret that's part of the service account mount as a secret object, but now as you can see it's a projected volume that actually communicates with the token controller API, token request API and it gets a token for the pod.

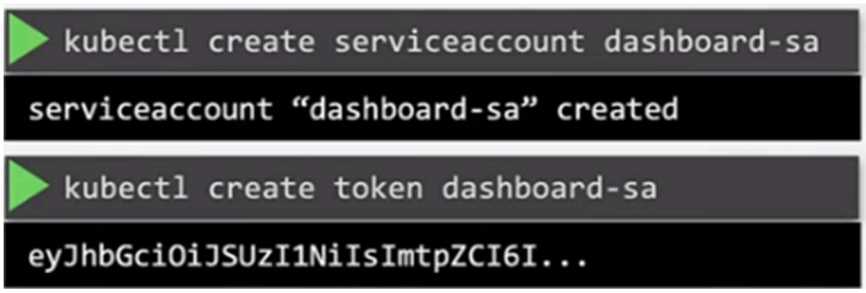


V1.24

Now with version 1.24, another enhancement was made as part of the Kubernetes enhancement proposal 2799, which dealt with the reduction of secret based service account tokens.

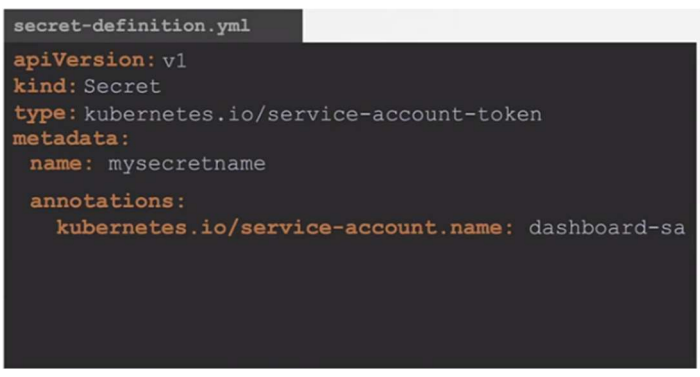
So in the past when a service account was created, it automatically created a secret with a token that has no expiry and is not bound to any audience. This was then automatically mount as a volume to any pod that uses that service account, and that's what we just saw. But in version 1.22 that was changed, the automatic mounting of the secret object to the pod was changed and instead it then moved to the token request API.

So with version 1.24, a change was made where when you create a service account, it no longer automatically creates a secret or a token as a secret. So you must run the command kubectl create token followed by the name of the service account to generate a token for that service account if you needed one and it will then print that token on screen.



Now, if you copy that token and then if you try to decode this token, this time you'll see that it has an expiry date defined. And if you haven't specified any time limit, then it's usually one hour from the time that you ran the command. You can also pass in additional options to the command to increase the expiry of the token.

So now post version 1.24, if you would still like to create secrets the old way with non-expiring token, then you could still do that by creating a secret object with the type set to **kubernetes.io/service-account-token** and the name of the service account specified within annotations in the metadata section. So this is how the secret object will be associated with that particular service account. So when you do this, just make sure that you have the service account created first and then create a secret object. Otherwise, the secret object will not be created.



So this will create a non-expiring token in a secret object and associated with that service account.

Now, you have to be sure if you really want to do that because that's where the Kubernetes documentation pages on service account token secrets. **It says you should only create service account token secrets if you can't use the token request API to obtain a token**.

Service account token Secrets

A `kubernetes.io/service-account-token` type of Secret is used to store a token credential that identifies a service account.

Since 1.22, this type of Secret is no longer used to mount credentials into Pods, and obtaining tokens via the TokenRequest API is recommended instead of using service account token Secret objects. Tokens obtained from the `TokenRequest` API are more secure than ones stored in Secret objects, because they have a bounded lifetime and are not readable by other API clients. You can use the `kubectl create token` command to obtain a token from the `TokenRequest` API.

You should only create a service account token Secret object if you can't use the `TokenRequest` API to obtain a token, and the security exposure of persisting a non-expiring token credential in a readable API object is acceptable to you.

And also, you should only create service account token request if the security exposure of persisting a non-expiring token credential is acceptable to you.

Now, the token request API is recommended instead of using the service account token secret objects, as they are more secure and have a bounded lifetime, unlike the service account token secrets that have no expiry.

Securing Images

In this lecture, we will talk about securing images. We will start with the basics of image names and then work our way towards secure image repositories and how to configure your pods to use images from secure repositories.

We deployed a number of different kinds of pods hosting different kinds of applications throughout this course, like web apps and databases and Redis cache, et cetera.

Let's start with a simple pod definition file. For instance, here we have used the Nginx image to deploy a Nginx container. Let's take a closer look at this image name. The name is Nginx but what is this image and where is this image pulled from?

This name follows Docker's image naming convention. Nginx here is the image or the repository name. When you say Nginx, it's actually **library/Nginx**. The first part stands for the user or the account name. So, if you don't provide a user or account name, it assumes it to be library.

Library is the name of the default account where Docker's official images are stored. These images promote best practices and are maintained by a dedicated team who are responsible for reviewing and publishing these official images.

Now, if you were to create your own account and create your own repositories or images under it, then you would use a similar pattern. So instead of library, it would be your name or your company's name.

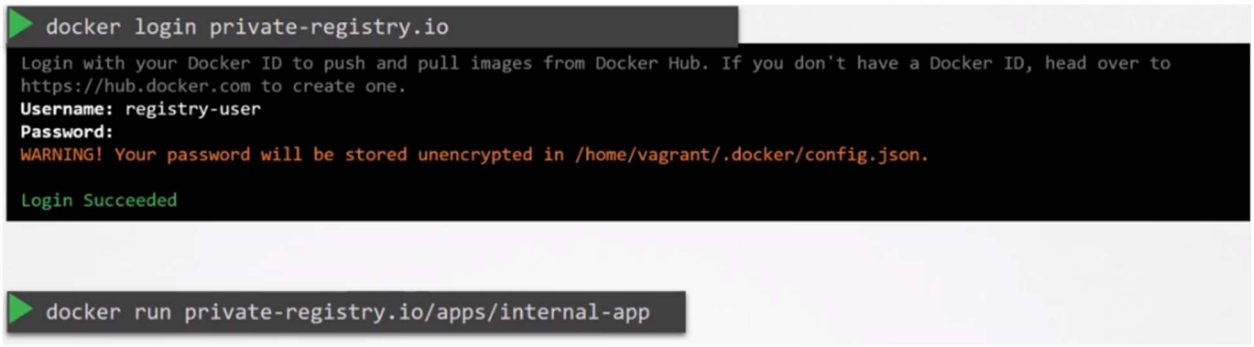
Now, where are these images stored and pulled from? Since we have not specified the location where these images are to be pulled from, it is assumed to be Docker's default registry, Docker hub. The DNS name for which is docker.io.

The registry is where all the images are stored. Whenever you create a new image or update an image, you push it to the registry and every time anyone deploys this application, it is pulled from the registry.

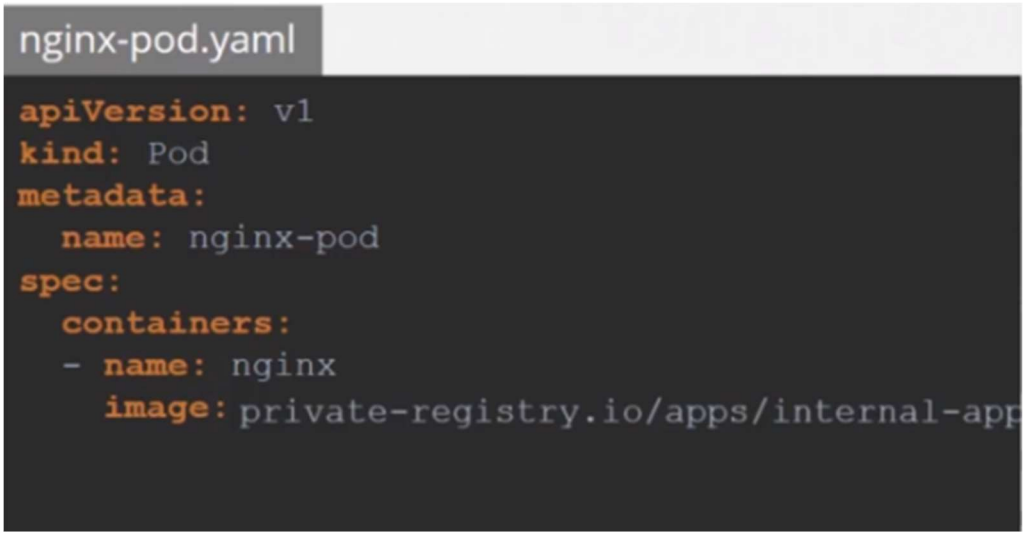
There are many other popular registries as well. Google's registry is at gcr.io, where a lot of Kubernetes related images are stored, like the ones used for performing end-to-end tests on the cluster. These are all publicly accessible images that anyone can download and access.

When you have applications built in-house that shouldn't be made available to the public, hosting an internal private registry may be a good solution. Many cloud service providers, such as AWS, Azure or GCP, provide a private registry by default. On any of these solutions, be it on Docker hub or Google's registry or your internal private registry, you may choose to make a repository private so that it can be accessed using a set of credentials.

From Docker's perspective, to run a container using a private image, you first log into your private registry using the Docker login command. Input your credentials. Once successful, run the application using the image from the private registry.

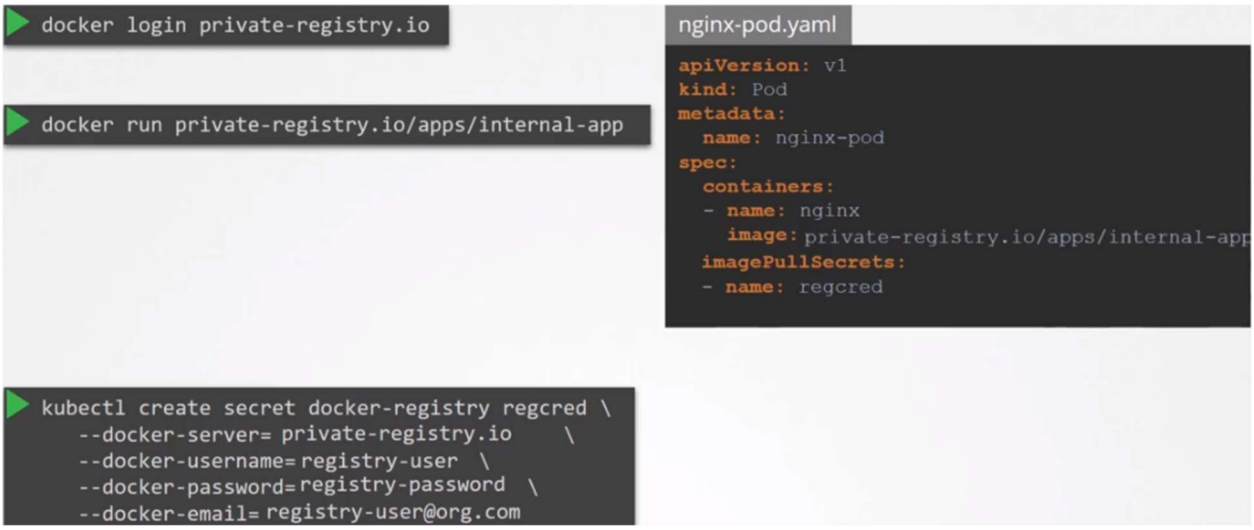


Going back to our pod definition file, to use an image from our private registry, we replace the image name with the full path to the one in the private registry.



But how do we implement the authentication, the login part? How does Kubernetes get the credentials? to access the private registry? Within Kubernetes, we know that the images are pulled and run by the Docker runtime on the worker nodes. How do you pass the credentials to the docker run times on the worker nodes?

For that, we first create a secret object with the credentials in it. The secret is of type **docker-registry** and we name it regcred, **docker-registry** is a built in secret type that was built for storing Docker credentials. We then specify the registry server name, the username to access the registry, the password, and the email address of the user. We then specify the secret inside our pod definition file under the image pull secret section.



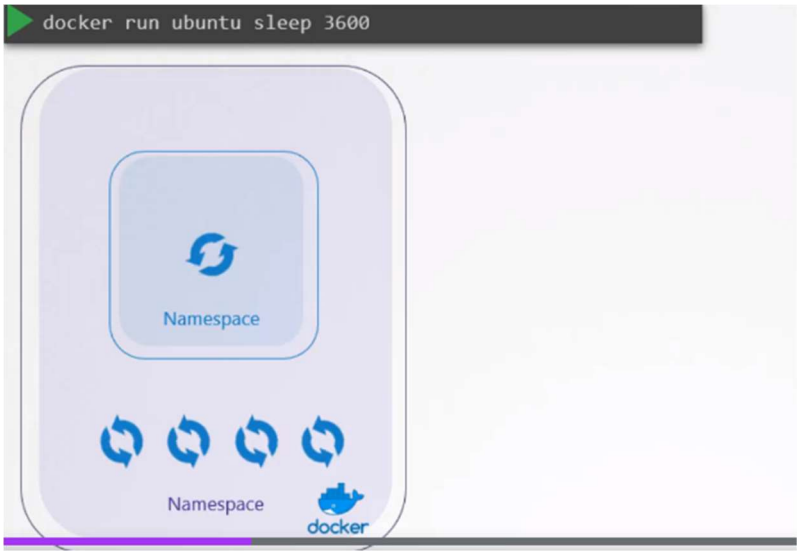
When the pod is created, Kubernetes or the kubelets on the worker node uses the credentials from the secret to pull images.

Security in Docker

Let us start with a host with Docker installed on it. This host has a set of its own processes running, such as a number of operating system processes, the Docker daemon itself, the SSH server, et cetera.

We will now run an Ubuntu Docker container that runs a process that sleeps for an hour. We have learned that unlike virtual machines, containers are not completely isolated from their host. Containers and the host share the same kernel. Containers are isolated using namespaces in Linux.

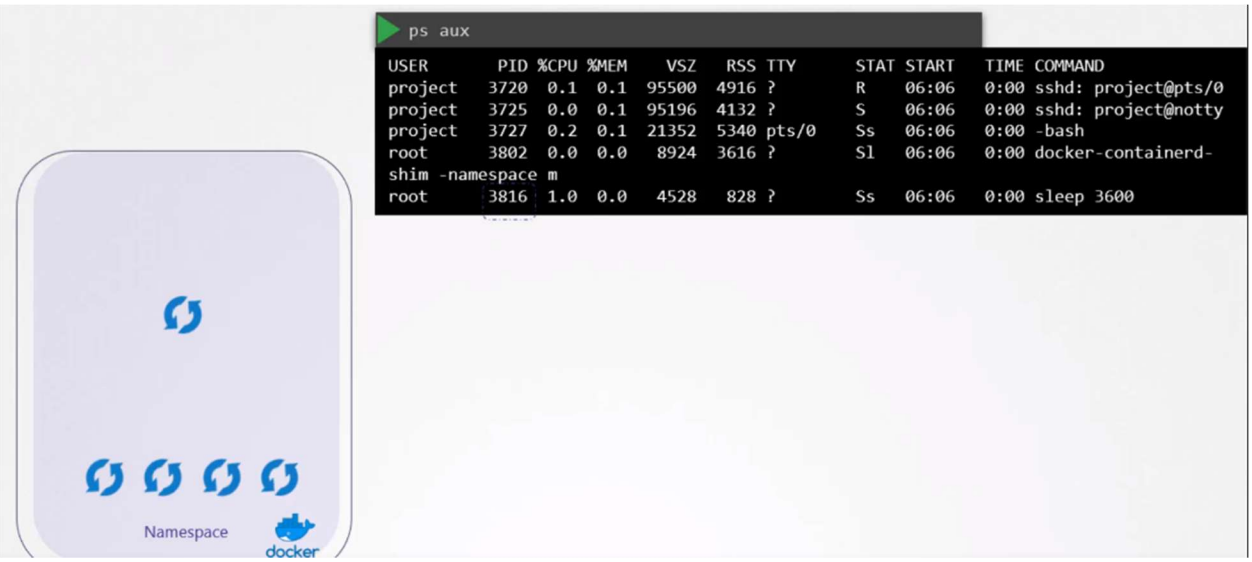
The host has a namespace and the containers have their own namespace. All the processes run by the containers are in fact run on the host itself but in their own namespace.



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

For the Docker host, all processes of its own, as well as those in the child namespaces are visible as just another process in the system.

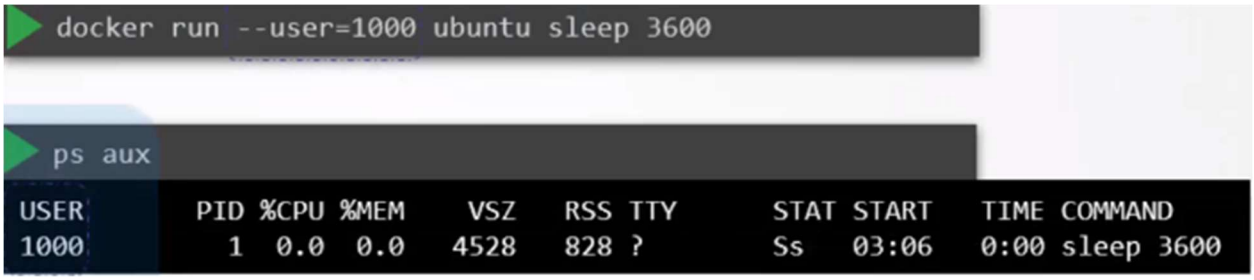
So when you list the processes on the host, you see a list of processes, including the sleep command but with a different process ID. This is because the processes can have different process IDs in different namespaces, and that's how Docker isolates containers within a system. So that's process isolation.



Let us now look at users in context of security. The Docker host has a set of users, a root user, as well as a number of non-root users. **By default, Docker runs processes within containers as**

the root user. This can be seen in the output of the commands we ran earlier. Both within the container and outside the container on the host, the process is run as the root user.

Now, if you do not want the process within the container to run as the root user you may set the user using the user option within the Docker run command and specify the new user ID. You will see that the process now runs with the new user ID.

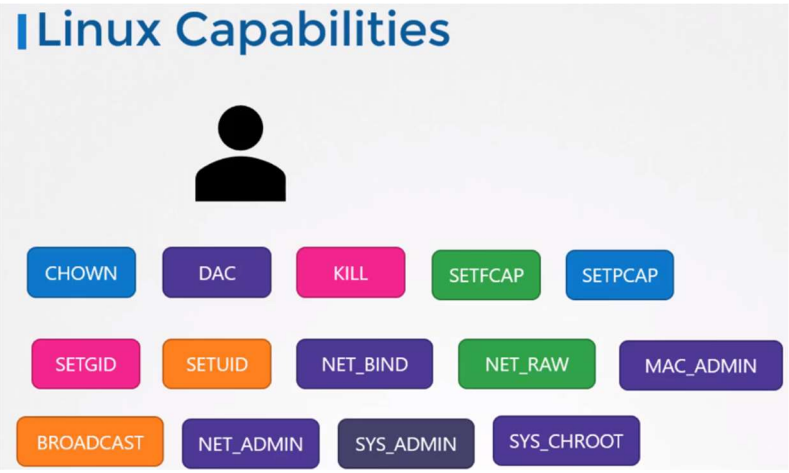


Another way to enforce user security is to have this defined in the Docker image itself at the time of creation. For example, we will use the default Ubuntu image and set the user ID to 1,000 using the user instruction. Then build the custom image. We can now run this image without specifying the user ID and the process will be run with the user ID 1,000.

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

implements a set of security features that limits the abilities of the root user within the container. So the root user within the container isn't really like the root user on the host.

Docker uses Linux capabilities to implement this. As we all know, the root user is the most powerful user on a system. The root user can literally do anything, and so does a process run by the root user. It has unrestricted access to the system, from modifying files and permissions on files, access control, creating or killing processes, setting group ID or user ID, performing network-related operations, such as binding two network ports, broadcasting on a network, controlling network ports, system-related operations, like rebooting the host, manipulating system clock, and many more. All of these are the different capabilities on a Linux system. You can now control and limit what capabilities are made available to a user.



By default, Docker runs a container with a limited set of capabilities. And so the processes running within the container do not have the privileges to say reboot the host or perform operations that can disrupt the host or other containers running on the same host.

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

```
docker run --cap-add MAC_ADMIN  ubuntu
```

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

```
docker run --cap-drop KILL      ubuntu
```

Or in case you wish to run the container with all privileges enabled, use the privileged flag.

```
docker run --privileged    ubuntu
```

Security Context

As we saw in the previous lecture, when you run a Docker container, you have the option to define a set of security standards, such as the ID of the user used to run the container, the Linux capabilities that can be added or removed from the container, et cetera.

These can be configured in Kubernetes as well. As you know already, in Kubernetes, containers are encapsulated in Pods. You may choose to configure the security settings at a container level or at a Pod level. If you configure it at a Pod level, the settings will carry over to all the containers within the Pod. If you configure it at both the Pod and the container, the settings on the container will override the settings on the Pod.

Let us start with a Pod definition file. This pod runs an ubuntu image with the sleep command. To configure security context on the container, add a field called **securityContext** under the **spec** section of the Pod. Use the **runAsUser** option to set the user ID for the Pod.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: web-pod
spec:
 securityContext:
   runAsUser: 1000


 containers:
      - name: ubuntu
        image: ubuntu
        command: ["sleep", "3600"]
```

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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: web-pod
spec:
  containers:
      - name: ubuntu
        image: ubuntu
        command: ["sleep", "3600"]
        securityContext:
          runAsUser: 1000
```

To add capabilities, use the capabilities option, and specify a list of capabilities to add to the Pod.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: web-pod
spec:
  containers:
      - name: ubuntu
        image: ubuntu
        command: ["sleep", "3600"]
        securityContext:
          runAsUser: 1000
          capabilities:
              add: ["MAC_ADMIN"]
```