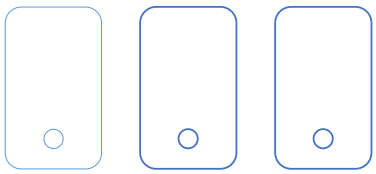

KodeKloud

SECURITY PRIMITIVES

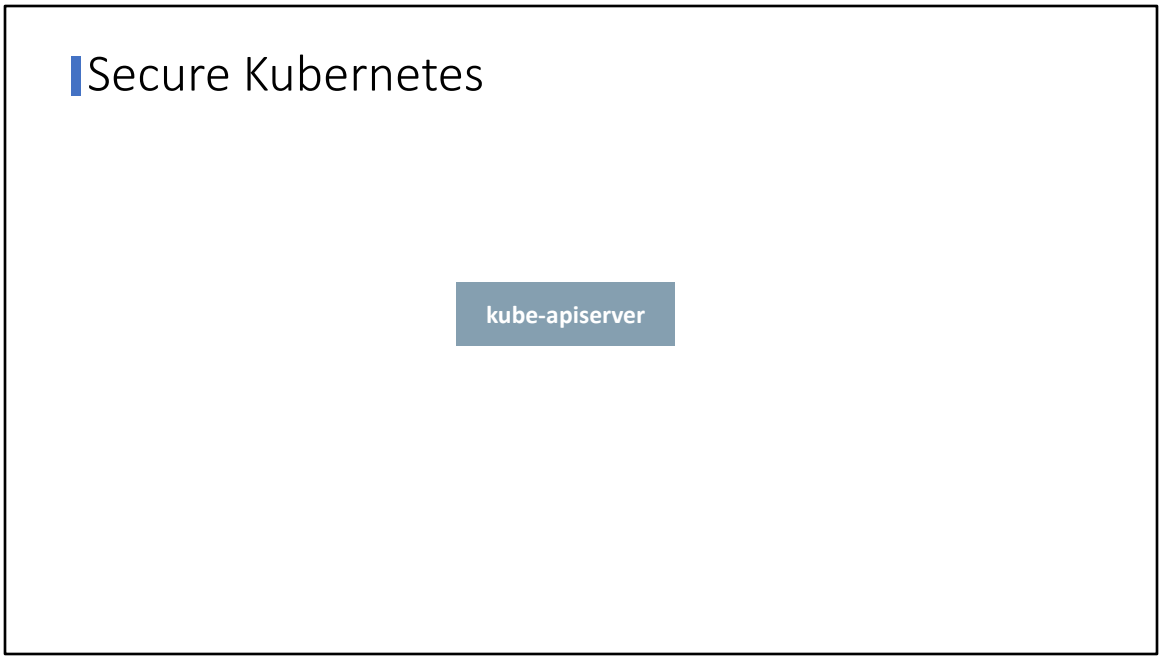
Secure Hosts



- ❑ Password based authentication disabled
- ❑ SSH Key based authentication

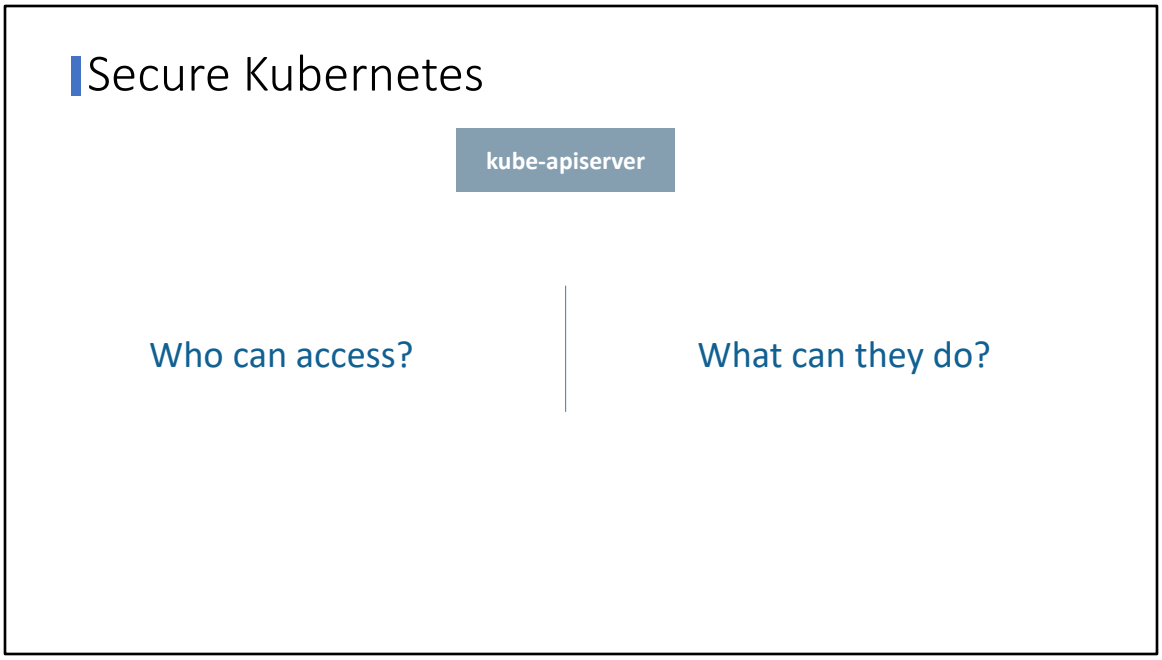
Kubernetes being the de-facto platform for hosting production grade applications, security is of prime concern. In this lecture we look at the various security primitives in Kubernetes at a high level before diving deeper into those in the upcoming lectures.

Let's begin with the hosts that form the cluster itself. <c> Of course all access to these hosts must be secured, root access disabled, password based authentication disabled, and only SSH key based authentication to be made available. And of course any other measures you need to take to secure your physical or virtual infrastructure that hosts kubernetes. Of course if that is compromised, everything is compromised.



But 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.

As we have seen already, the kube-api server is at the center of all operations within kubernetes. We interact with it through the kubectl 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.



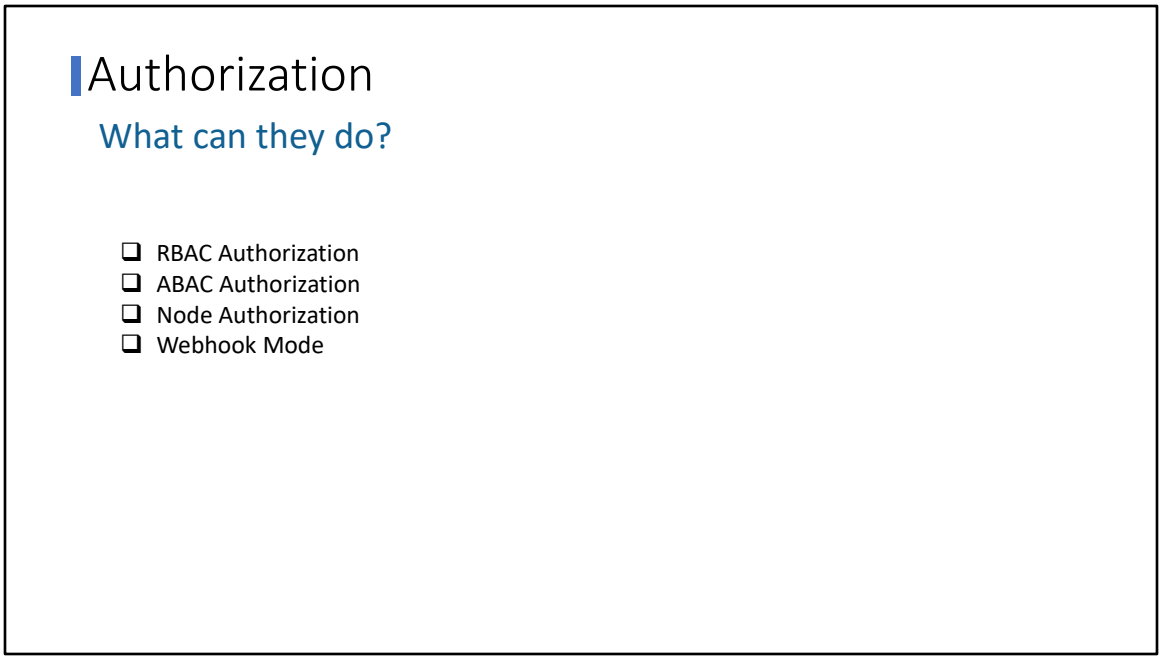
We will need to make two types of decisions. <c> Who can access the cluster? And what can they do?

Authentication

Who can access?

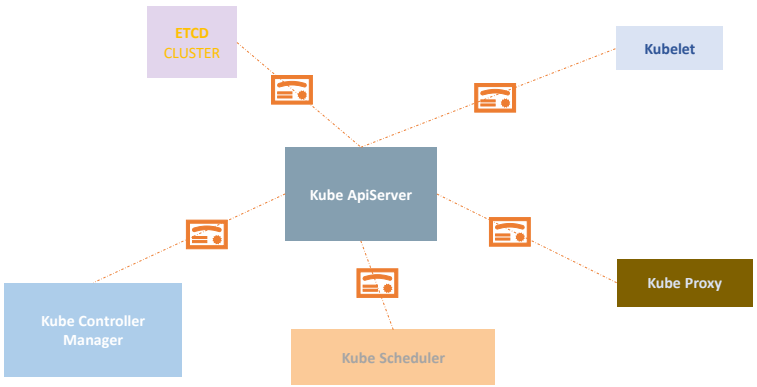
- ❑ Files – Username and Passwords
- ❑ Files – Username and Tokens
- ❑ Certificates
- ❑ External Authentication providers - LDAP
- ❑ Service Accounts

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 a static file, to tokens, certificates or even integration with external authentication providers like LDAP. And finally for machines we create service accounts. We will look at these in more detail in the upcoming lectures.



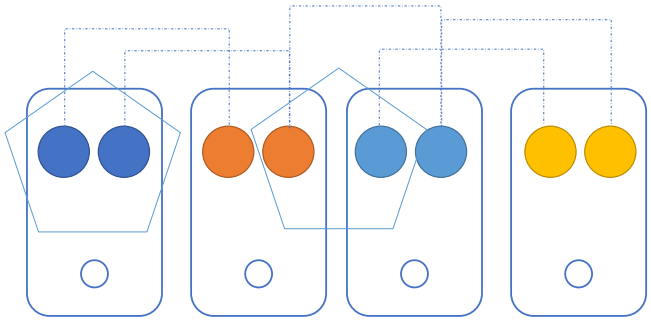
Once they gain access to the cluster, what can they do is defined by the Authorization mechanisms. Authorization is implemented using Role Based Access Control, where users are associated to groups with specific permissions. In addition there are other authorization modules like Attribute based access control, Node Authorizers, webhooks etc. Again we look at these in more detail in the upcoming lectures.

TLS Certificates



All communication with the cluster, between the various components such as the ETCD cluster, kube controller manager, scheduler, api server, as well as those running on the worker nodes such as the kubelet and kubeproxy is secured using <c> TLS Encryption. We have a section entirely for this where we discuss and practice how to setup the certificates between the various components.

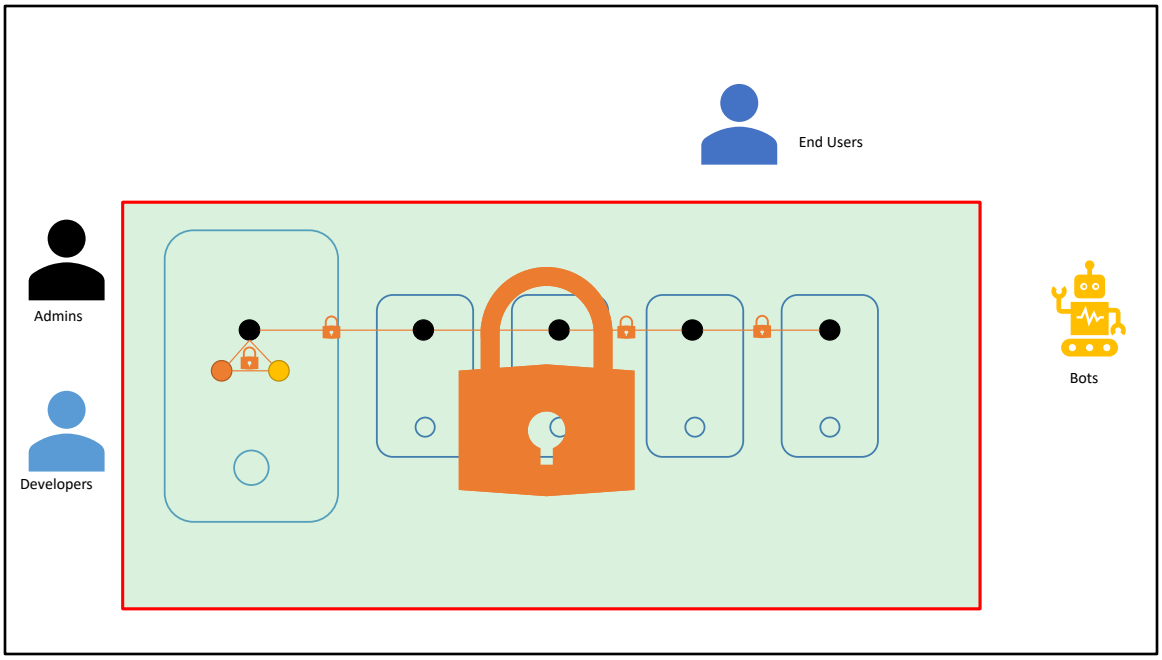
Network Policies



What about communication between applications within the cluster? By default all PODs can access all other PODs within the cluster. You can restrict access between them using <c> Network Policies. We will look at how exactly that is done later.

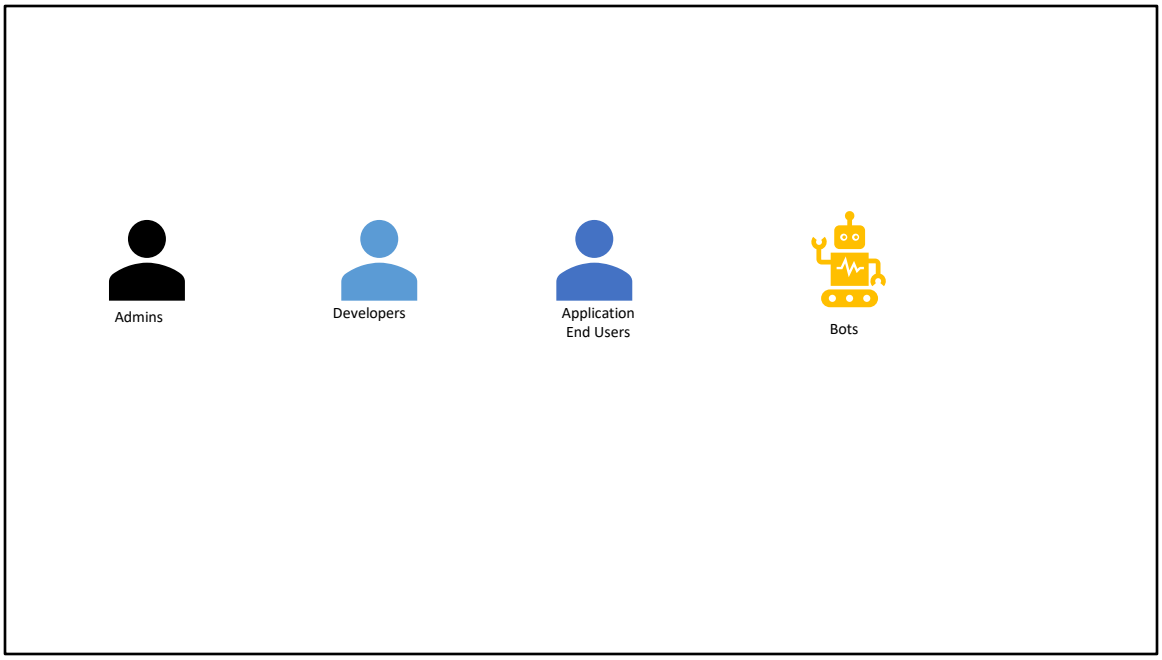
So that was a high level overview of the various security primitives in kubernetes. We will now look at these in much more detail going forward.


10
AUTHENTICATION

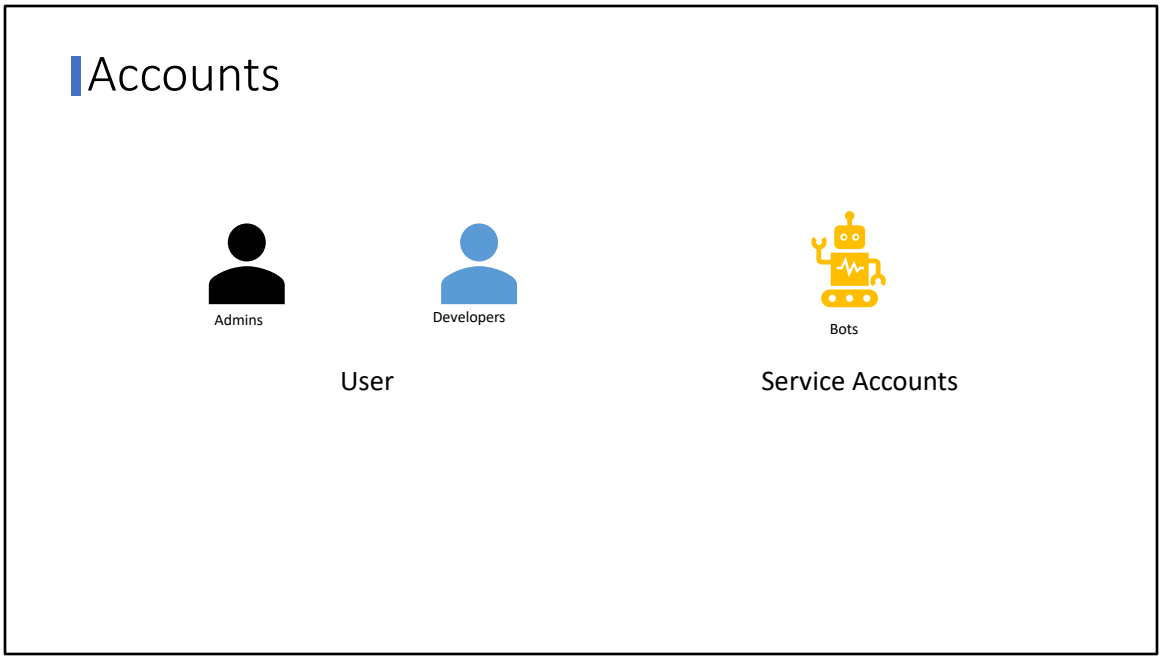


As we have seen already, the kubernetes cluster consists of multiple nodes physical/virtual and various components that work together. We have users like Administrators that access the cluster to perform administrative tasks and Developers that access the cluster to test or deploy applications. We have end users who access the applications deployed on the kubernetes cluster. And we have third party applications accessing the cluster for integration purposes.

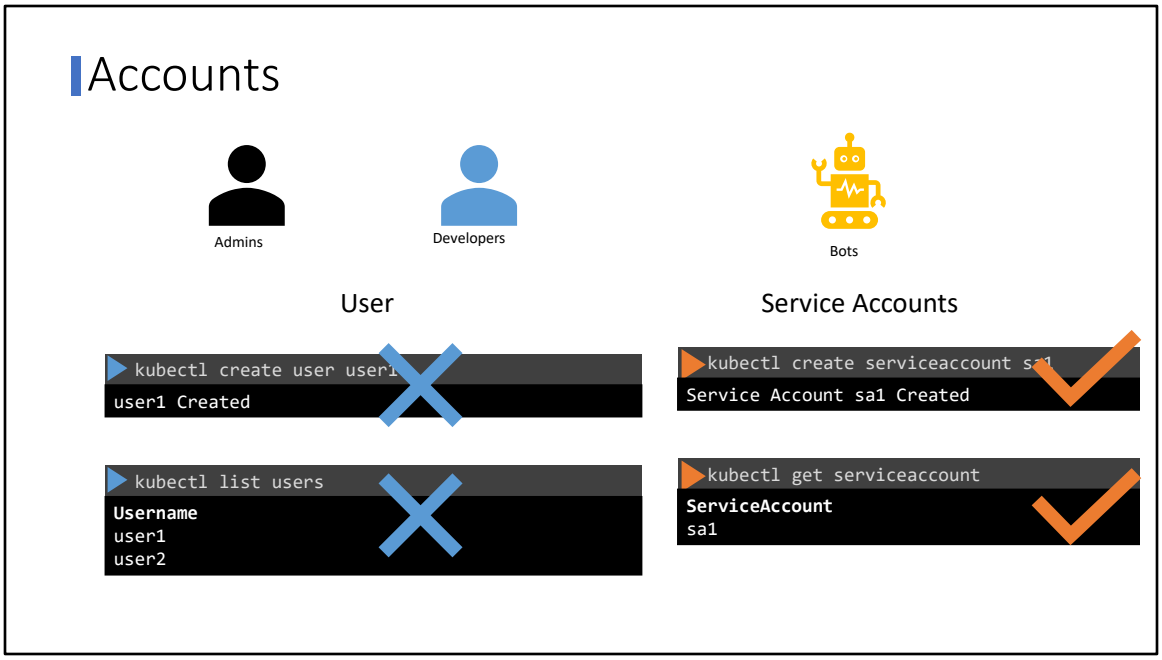
Throughout this section we will discuss how to secure our cluster by securing the communication between internal components, and securing management access to the kubernetes cluster through authentication and authorization mechanisms. In this lecture our focus is on securing access to the kubernetes cluster with authentication mechanisms.



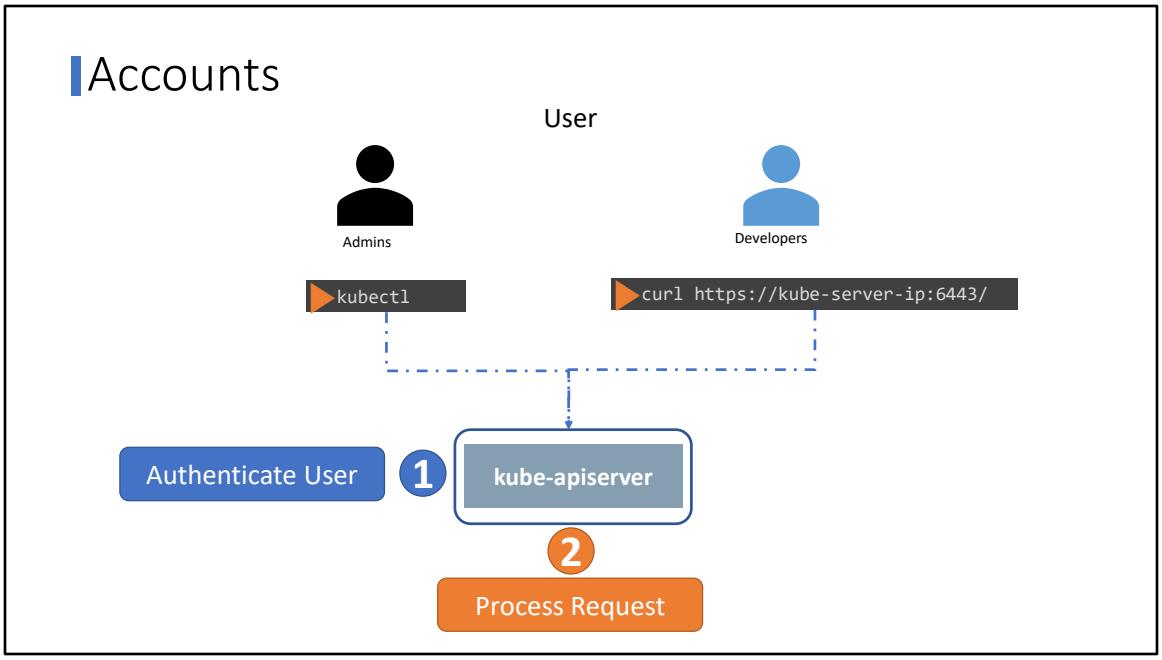
So we talked about the different users that may be accessing the cluster. Security of end users who access the applications deployed on the cluster is managed by the applications themselves internally.



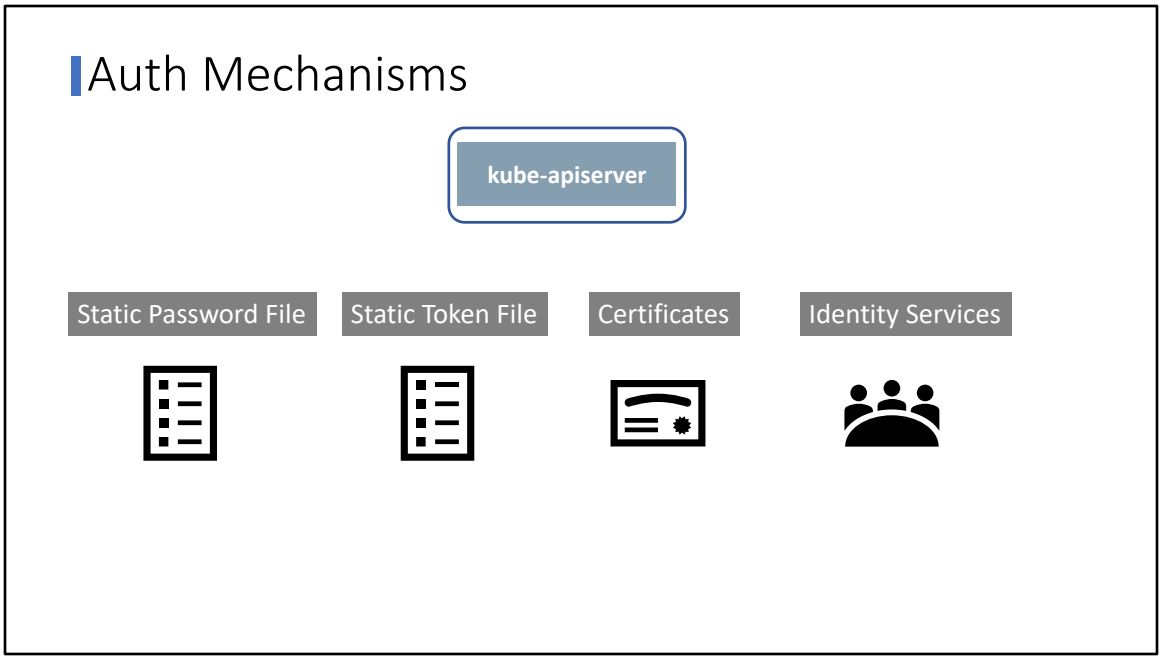
So we will take them out of our discussion. Our focus is on users access to the kubernetes cluster for administrative purposes. So we are left with two types of users. Humans, such as the Administrators and Developers and Robots such as other processes/services or applications that requires access to the cluster.



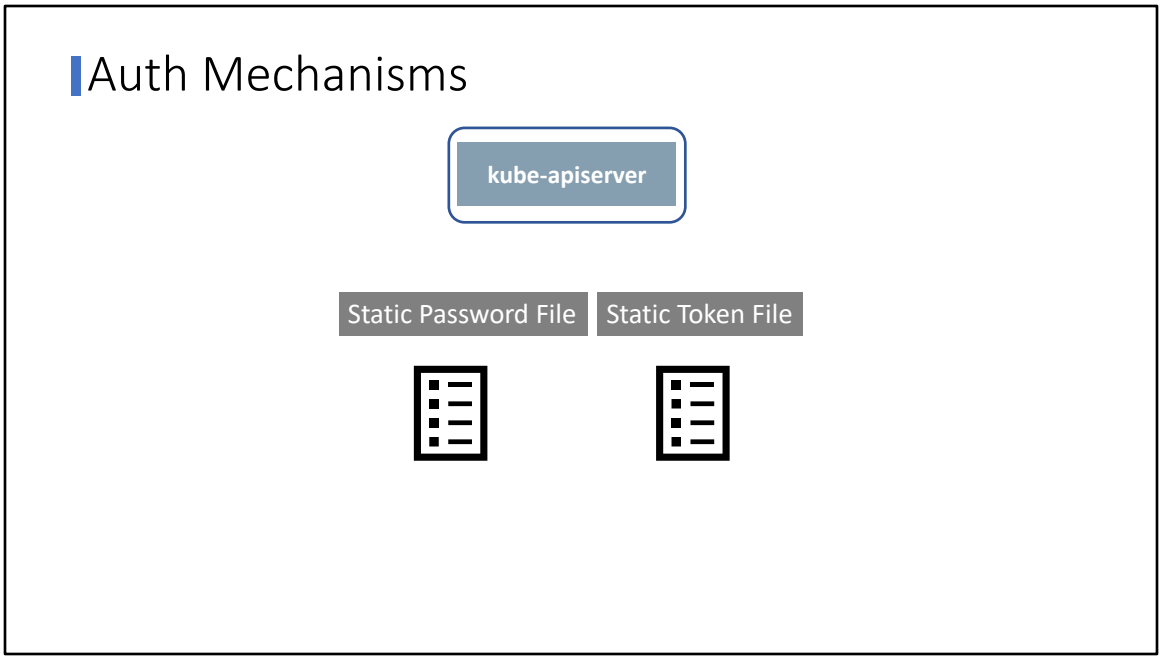
Kubernetes does not manage user accounts natively. It relies on an external file with user details or certificates or a third party identity service like LDAP to manage these users. And so you cannot create users in a kubernetes cluster or view the list of users like this. However in case of Service Accounts kubernetes can manage them. You can create and manage service accounts using the Kubernetes API. We have a section on ServiceAccounts exclusively where we discuss and practice more about Service Accounts.



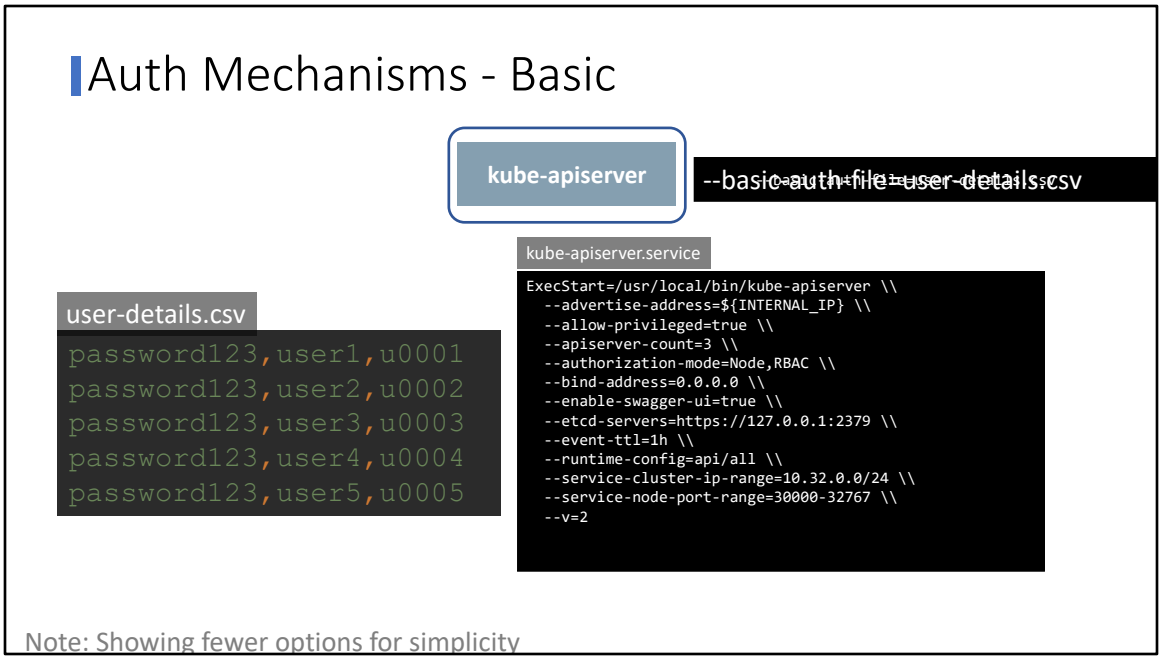
For this section we will focus on users in kubernetes. All user access is managed by the API server. Weather you are accessing the cluster through kubectl tool or the API directly, all of these requests go through the kube api server. The kube-api server <c> authenticates the requests before <c> processing it.



So how does the kube-api server 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 etc. 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 3 colums – password, username and userid. We then pass the file name as an option to the kube-api server. Remember the kube-api server 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-api server.

Kube-api Server Configuration

kube-apiserver.service

```
ExecStart=/usr/local/bin/kube-apiserver \\
  --advertise-address=${INTERNAL_IP} \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --authorization-mode=Node,RBAC \\
  --bind-address=0.0.0.0 \\
  --enable-swagger-ui=true \\
  --etcd-servers=https://127.0.0.1:2379 \\
  --event-ttl=1h \\
  --runtime-config=api/all \\
  --service-cluster-ip-range=10.32.0.0/24 \\
  --service-node-port-range=30000-32767 \\
  --v=2
  --basic-auth-file=user-details.csv
```

/etc/kubernetes/manifests/kube-apiserver.yaml

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --authorization-mode=Node,RBAC
    - --advertise-address=172.17.0.107
    - --allow-privileged=true
    - --enable-admission-plugins=NodeRestriction
    - --enable-bootstrap-token-auth=true

    image: k8s.gcr.io/kube-apiserver-amd64:v1.11.3
    name: kube-apiserver
```

Note: Showing fewer options for simplicity

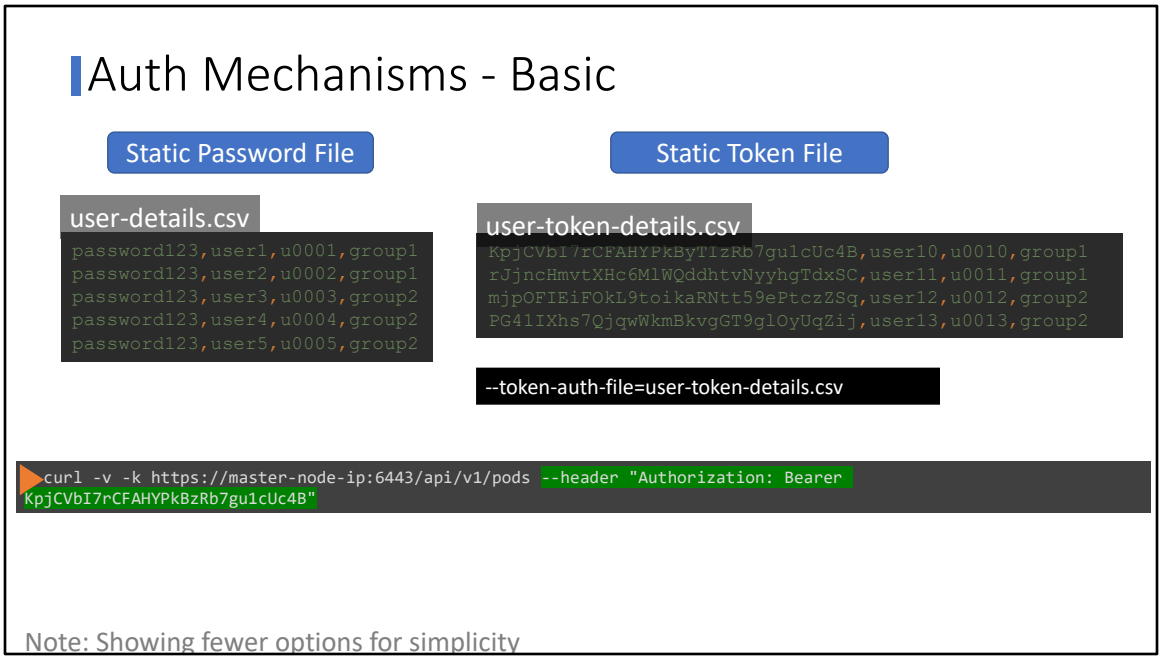
If you setup your cluster using the kubeadm tool, then you must modify the kube-apiserver POD definition file. Kube-adm tool will automatically restart the kube-api server once you update the file.

Authenticate User

```
curl -v -k https://master-node-ip:6443/api/v1/pods -u "user1:password123"
```


```
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/pods",
    "resourceVersion": "3594"
  },
 "items": [
    {
      "metadata": {
        "name": "nginx-64f497f8fd-krkg6",
        "generateName": "nginx-64f497f8fd-",
        "namespace": "default",
        "selfLink": "/api/v1/namespaces/default/pods/nginx-64f497f8fd-krkg6",
        "uid": "77dd7dfb-2914-11e9-b468-0242ac11006b",
        "resourceVersion": "3569",
        "creationTimestamp": "2019-02-05T07:05:49Z",
        "labels": {
          "pod-template-hash": "2090539498",
          "run": "nginx"
```


To authenticate using the basic credentials while accessing the API server, specify the user and password in a curl command like this.



We saw the user details csv file earlier. This file can optionally have a 4th 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-api server.

While authenticating specify the token as an Authorization Bearer token in your request like this.

Note

- This is not a recommended authentication mechanism
- Consider volume mount while providing the auth file in a kubeadm setup
- Setup Role Based Authorization for the new users

That's it for this lecture. Remember that this authentication mechanism that stores user names, 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.

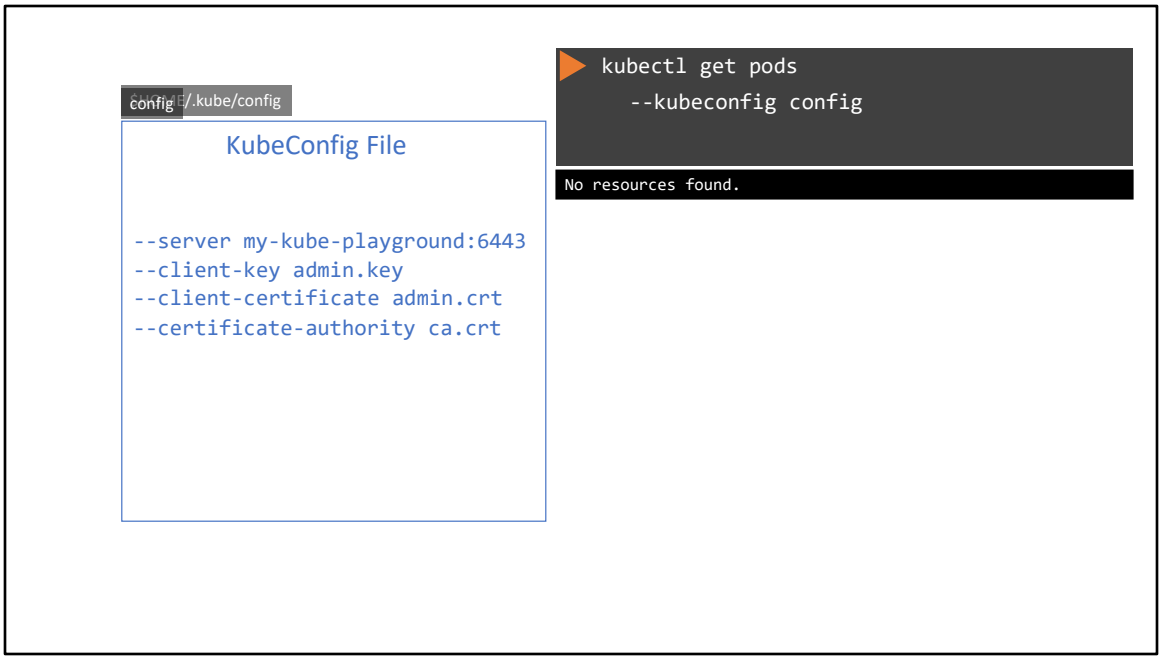
I also want to point out that if you were trying this out in a kubeadm setup you must also consider volume mounts to pass in the auth file. Details about these are available in the article that follows. And remember to setup Authorization for the new users. We will discuss about Authorization later in this course.

In the upcoming lectures we will discuss about Certificate based authentication and how the various components within the kubernetes cluster are secured using certificates.

In this lecture we look at how to manage certificates and what the Certificate API is.

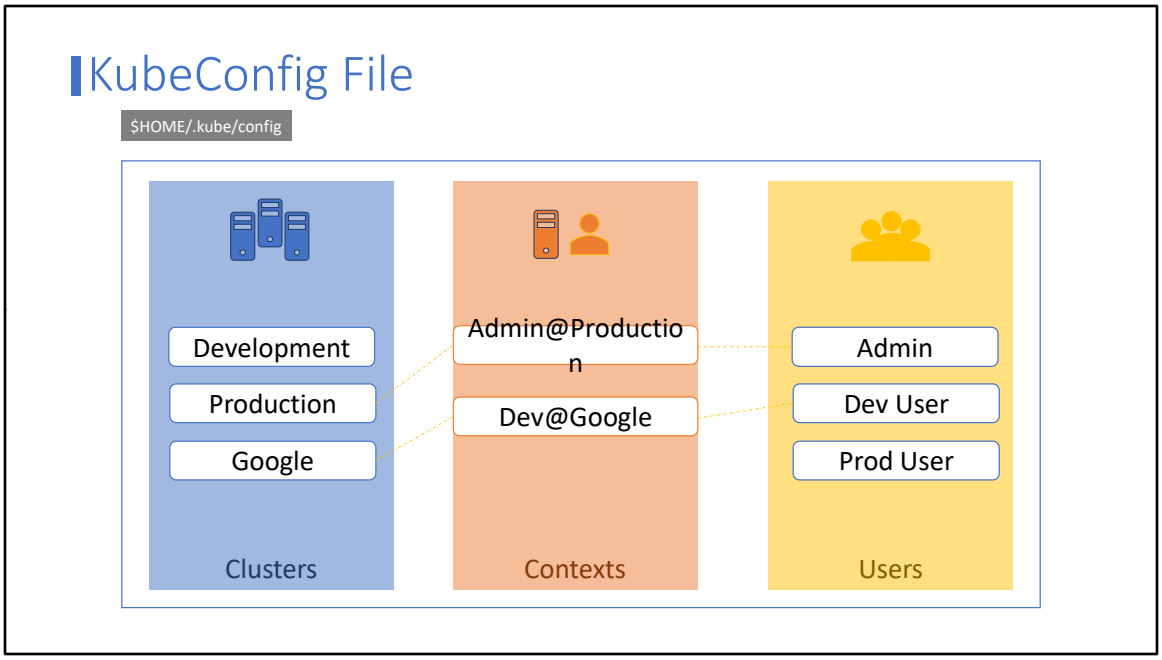
So far we have seen how to generate a certificate for a user. We have seen how a client uses the certificate file and key to query the kubernetes Rest API for a list of PODs using Curl. In this case my cluster is called my-kube-playground, so send a CURL request to the address of the kube-api server 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 utililty.



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. By default the kubectl tool looks for a file named config under a directory .kube under the users 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 3 sections. Clusters, Users and Contexts.

Clusters are the various kubernetes clusters that you need access to. Say you have a multiple clusters for a Development environment , or testing environment or prod, or for different organizations or on different cloud providers etc. All those go here.

Users are the user accounts with which you have access to these clusters. For example the admin user. A Dev User, a prod user etc. 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.

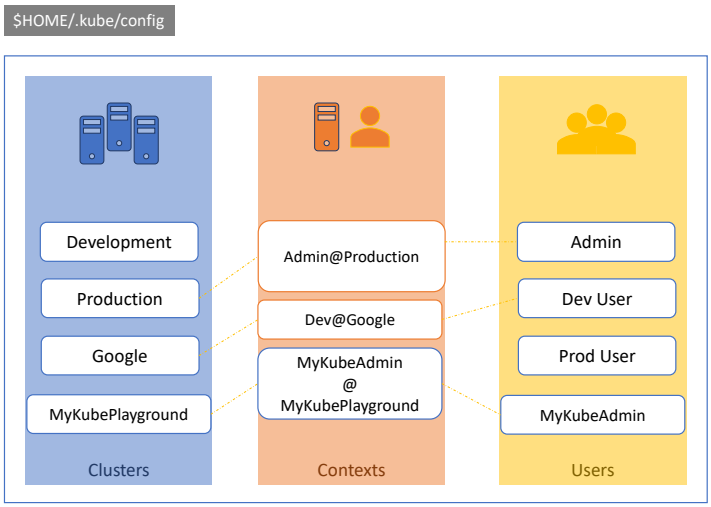
Or I may want to access the cluster I have setup on Google with the devusers credentials to test deploying the application I built.

Remember, you are not creating any new users or configuring any kind of user access or authorization in the cluster. You are using existing users with their existing

privileges and defining what user you are 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.

KubeConfig File

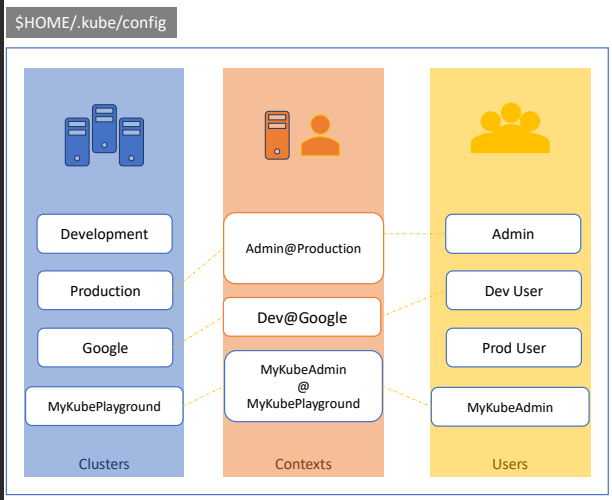
```
--server my-kube-playground:6443
--client-key admin.key
--client-certificate admin.crt
--certificate-authority ca.crt
```



So how does it fit into our example? The server specification in our command goes into the clusters section. The admin users keys and certificates goes into the users section. You then create a context that specifies to use the MyKubeAdmin user to access the MyKubePlayground cluster.

KubeConfig File

```
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
```



Let's look at a real KubeConfig file now. The kubeConfig is in a YAML format. It has apiVersion set to v1. The kind is config. And then it has 3 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.

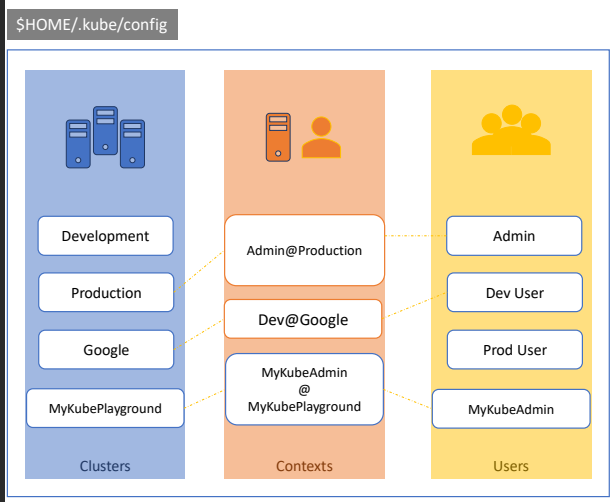
Under clusters we add a new item for our kube-playground cluster. We name it mukubeplayground and specify the server address under the server field. It also requires the certificate of the certificate authority.

We then add an entry into the users section to specify details of my kube admin user. Provide the location of the client's certificate and key pair. So we have now defined the cluster and the user to access the cluster.

Next create an entry under context 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.

KubeConfig File

```
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
```



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 chose from? We have 3 defined 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.

Kubectl config

```
kubectl config view
```

```
apiVersion: v1

kind: Config
current-context: kubernetes-admin@kubernetes

clusters:
- cluster:
    certificate-authority-data: REDACTED
    server: https://172.17.0.5:6443
  name: kubernetes

contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes


users:
- name: kubernetes-admin
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

```
kubectl config view –kubeconfig=my-custom-config
```

```
apiVersion: v1

kind: Config
current-context: my-kube-admin@my-kube-playground


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
```

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 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 users home directory. Alternatively you can specify a kubeconfig file by passing the kubeconfig option in the command line like this.

Kubectl config

```
kubectl config view
```

```
apiVersion: v1

kind: Config
current-context: my-kube-admin@my-kube-playground


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
```

```
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
```

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? Say you have 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@production context. This change 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.

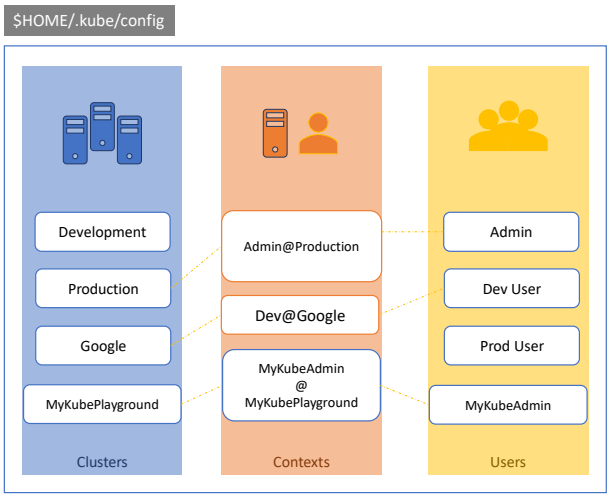
Kubectl config

```
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
```

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.

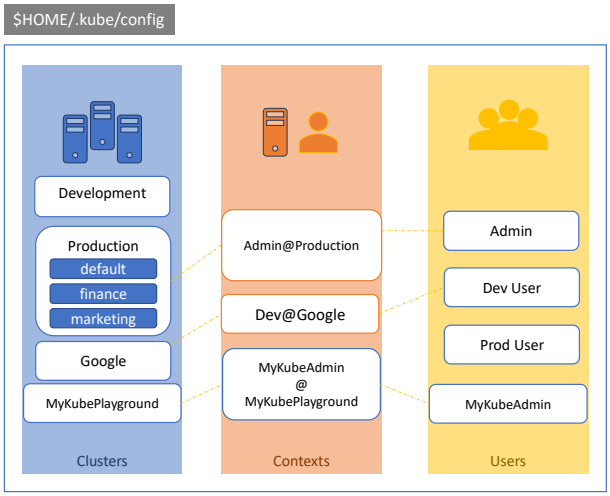
Namespaces



What about namespaces? For example, each cluster may be configured with multiple namespaces within it.

Namespaces

```
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
```



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.

Certificates in KubeConfig

```
apiVersion: v1
kind: Config

clusters:
- name: production
  cluster:
    certificate-authority: /etc/kubernetes/pki/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: /etc/kubernetes/pki/users/admin.crt
    client-key: /etc/kubernetes/pki/users/admin.key
```

Finally, a word on certificates. You have seen path to certificate files mentioned in kubeconfig like this. Well, its better to use the full path like this.

Certificates in KubeConfig

```
apiVersion: v1
kind: Config

clusters:
- name: production
  cluster:
    certificate-authority: /etc/kubernetes/pki/ca.crt

    certificate-authority-data:
```

```
-----BEGIN CERTIFICATE -----
MIICWDCCAUACAQAwEzERMA8GA1UEAwwIbmV3LXVzZXIwggEiMA0GC
AQUAA4IBDwAwggEKAoIBAQDO0WJW+DXsAJSIrjpNo5vRIBplnzg+6
LfC27t+1eEnON5Muq99NevmMEOnrDUO/thyVqP2w2XNIDRXjYyF40
y3BihhB93MJ7Oql3UTvZ8TELqyaDknRl/jv/SxgXkok0ABUTpWMx4
IF5nxAttMVkDPQ7NbeZRG43b+QWlVGR/z6DWOfJnbfezOtaAydGLT
EcCXAwqChjBLkz2BHPR4J89D6Xb8k39pu6jpyngV6uP0tIbOzpqNv
j2qEL+hZEWkkFz80lNNtyT5LxMqENDCnIgwC4GZiRGbrAgMBAAGgA
9w0BAQsFAAOCAQEAS9iS6C1uxTuf5BBYSU7QFQHUzalNxAdYsaORR
hOK4a2zyNyi44OOijyaD6tUW8DSxkr8BLK8Kg3srREtJql5rLZy9L
P9NL+aDRSxROVSqBaB2nWeYpM5cJ5TF53lesNSNMLQ2++RMnjDQJ7
Wr2EUM6UawzykrdHImwTv2mlMY0R+DNtV1Yie+0H9/YElt+FSGjh5
4l3E/y3qL71WfAcuH3OsVpUUnQISMdQs0qWCsbE56CC5DhPGZIpUb
vwQ07jG+hpknxmuFAeXxgUwodALaJ7ju/TDIcw==
-----END CERTIFICATE -----
```

cat ca.crt | base64

```
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRVFVRVNU
tLS0KTUlJQ1dEQ0NBVUFDQVFBd0V6RVJNQThHQTF
F3d0libVYzTFhWelpYSXdnZ0VpTUEwR0NTcUdTSV
FFFQgpBUVVBQTRJQkR3QXdnZ0VLOW9JQkFRRE8wV
K0RYe0FKU01yanBObzV2UklCcGxuemcrNnhjOStV
rS2kwCkxmQzI3dCsxZUVuT041TXVxOTlOZXZtTUV
J
nBObzV2UklCcGxuemcrNnhjOStVVndrS
kwCkxmQzI3dCsxZUVuT041TXVxOTlOZX
```

But remember there is 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 the 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 certificates data in the encoded format, use the base64 --decode option to decode the certificate.

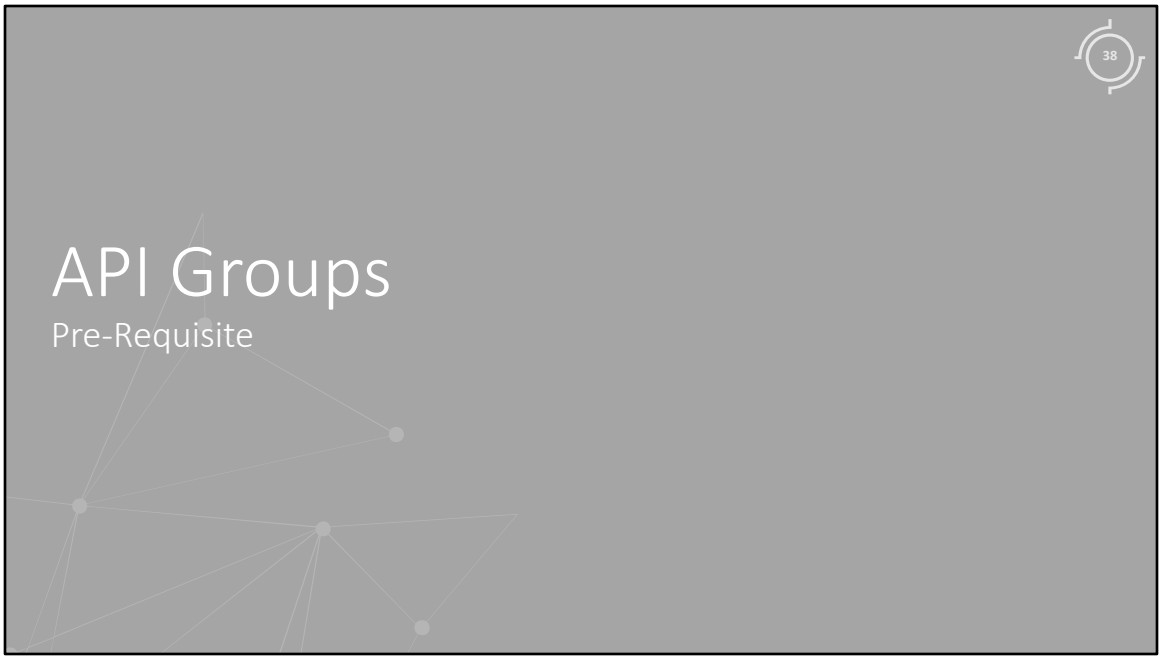
Certificates in KubeConfig

```
apiVersion: v1
kind: Config

clusters:
- name: production
  cluster:
    certificate-authority: /etc/kubernetes/pki/ca.crt

    certificate-authority-data: LS0tLS1CRUdJTiBDRVJU
                    SUZJQ0FURSBSRVFVRVNULS0tLS0KTUlJQ
                    1dEQ0NBVUFDQVFBd0V6RVJNQThHQTFVRU
                    F3d0libVYzTFhWelpYSXdnZ0VpTUEwR0N
                    TcUdTSWIzRFFFQgpBUVVBQTRJQkR3QXdn
                    Z0VLQW9JQkFRRE8wV0pXK0RYc0FKU0lya
                    nBObzV2UklCcGxuemcrNnhjOStVVndrS2
                    kwCkxmQzI3dCsxZUVuT041TXVxOTlOZXZ
                    tTUVPbnJ
```

```
echo "LS0…bnJ" | base64 --decode

-----BEGIN CERTIFICATE -----
MIICWDCCAUACAQAwEzERMA8GA1UEAwwIbmV3LXVzZXIwggEiMA0GCSq
AQUAA4IBDwAwggEKAoIBAQDO0WJW+DXsAJSIrjpNo5vRIBplnzg+6xc
LfC27t+1eEnON5Muq99NevmMEOnrDUO/thyVqP2w2XNIDRXjYyF40Fb
y3BihhB93MJ7Oql3UTvZ8TELqyaDknRl/jv/SxgXkok0ABUTpWMx4Bp
IF5nxAttMVkDPQ7NbeZRG43b+QWlVGR/z6DWOfJnbfezOtaAydGLTZF
EcCXAwqChjBLkz2BHPR4J89D6Xb8k39pu6jpyngV6uP0tIbOzpqNv0Y
j2qEL+hZEWkkFz80lNNtyT5LxMqENDCnIgwC4GZiRGbrAgMBAAGgADA
9w0BAQsFAAOCAQEAS9iS6C1uxTuf5BBYSU7QFQHUzalNxAdYsaORRQN
hOK4a2zyNyi44QOijyaD6tUW8DSxkr8BLK8Kg3srREtJql5rLZy9LRV
P9NL+aDRSxROVSqBaB2nWeYpM5cJ5TF53lesNSNMLQ2++RMnjDQJ7ju
Wr2EUM6UawzykrdHImwTv2mlMY0R+DNtV1Yie+0H9/YElt+FSGjh5L5
4l3E/y3qL71WfAcuH3OsVpUUnQISMdQs0qWCsbE56CC5DhPGZIpUbnK
vwQ07jG+hpknxmuFAeXxgUwodALaJ7ju/TDIcw==
-----END CERTIFICATE -----
```

Similarly if you see a file with the certificates 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.

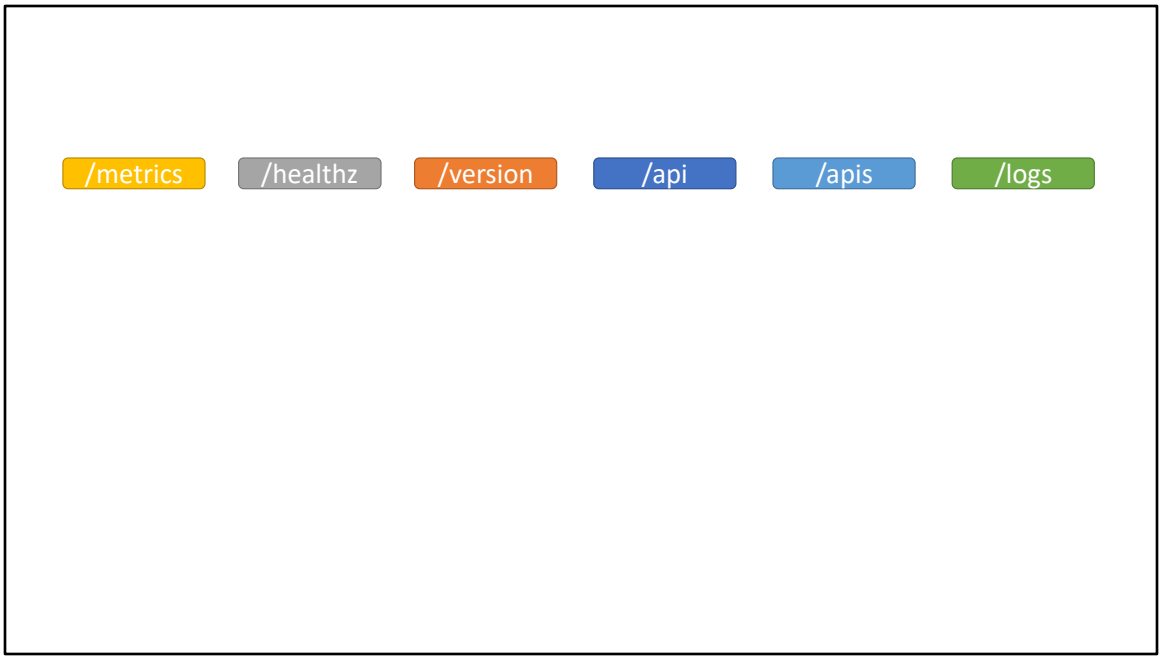


Let us talk about API Groups in Kubernetes.

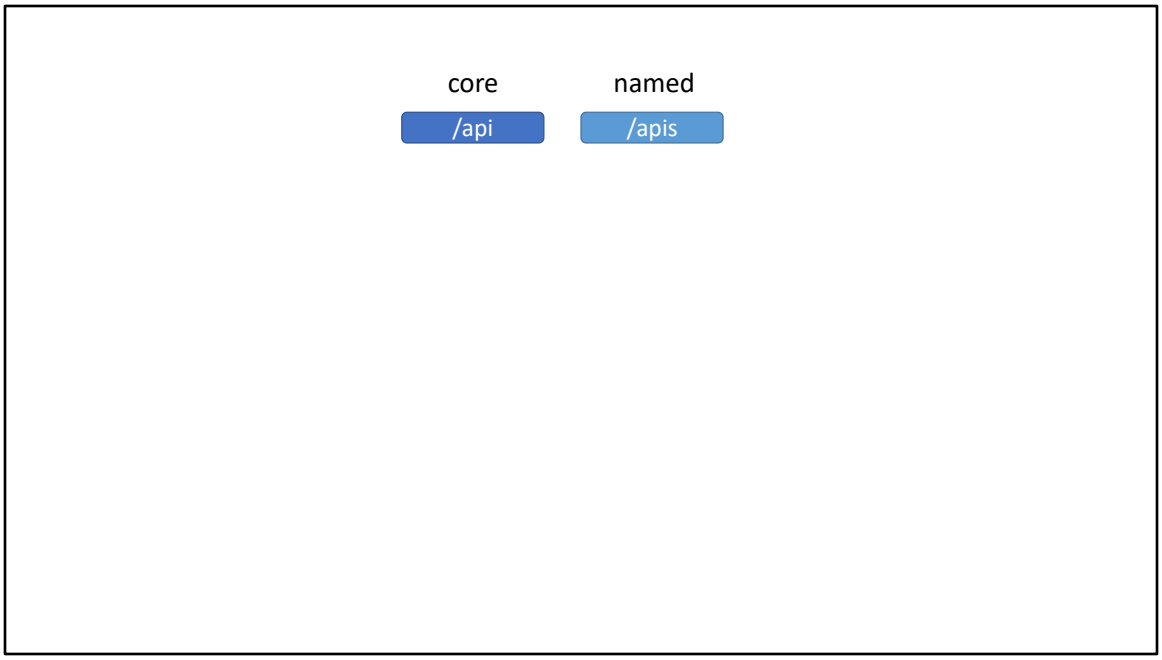
But first, what is the Kubernetes API? We learned about the Kube API server. Whatever operations we have done so far with the cluster, we have been interacting with the API server one way or the other. Either through the kubectl utility or directly. Say we want to check the version, we can access the api server at the master nodes 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/v1/pods.

Our focus in this lecture is about these API paths. The /version and /api.

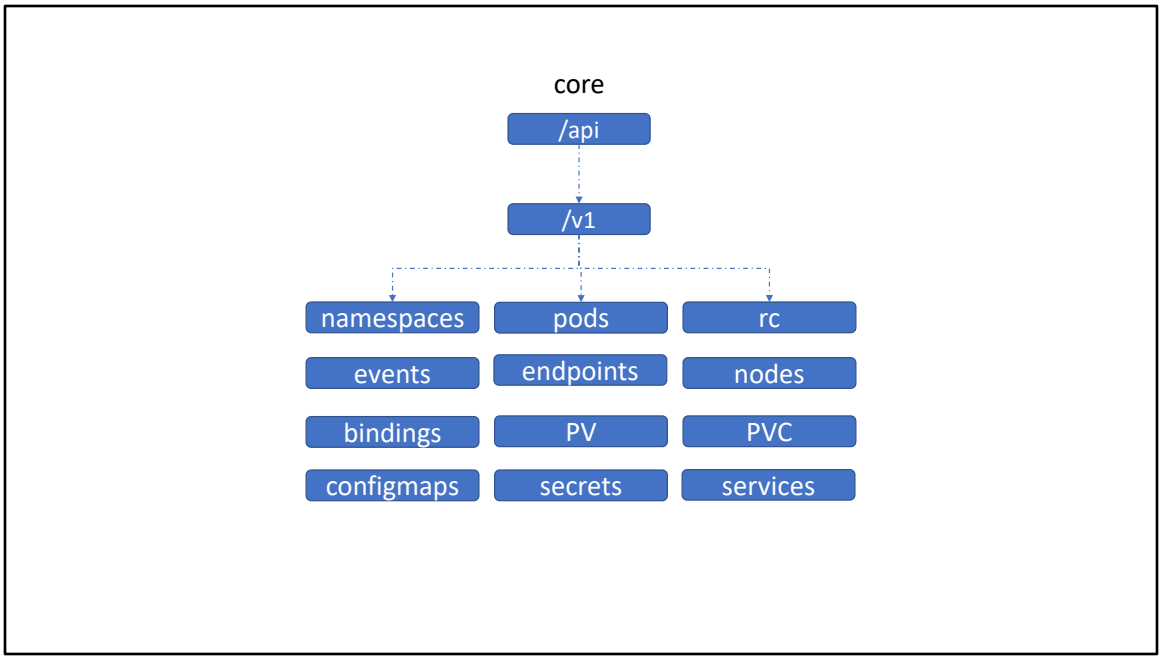


The Kubernetes API is grouped into multiple such groups based on their purpose. Such as one for apis, one for healthz, metrics and logs. The version API is for viewing the version of the cluster as we just saw. The metrics and healthz api are used to monitor the health of the cluster. The logs for integrating with third party logging application. In this video we will focus on the APIs responsible for the cluster functionality.

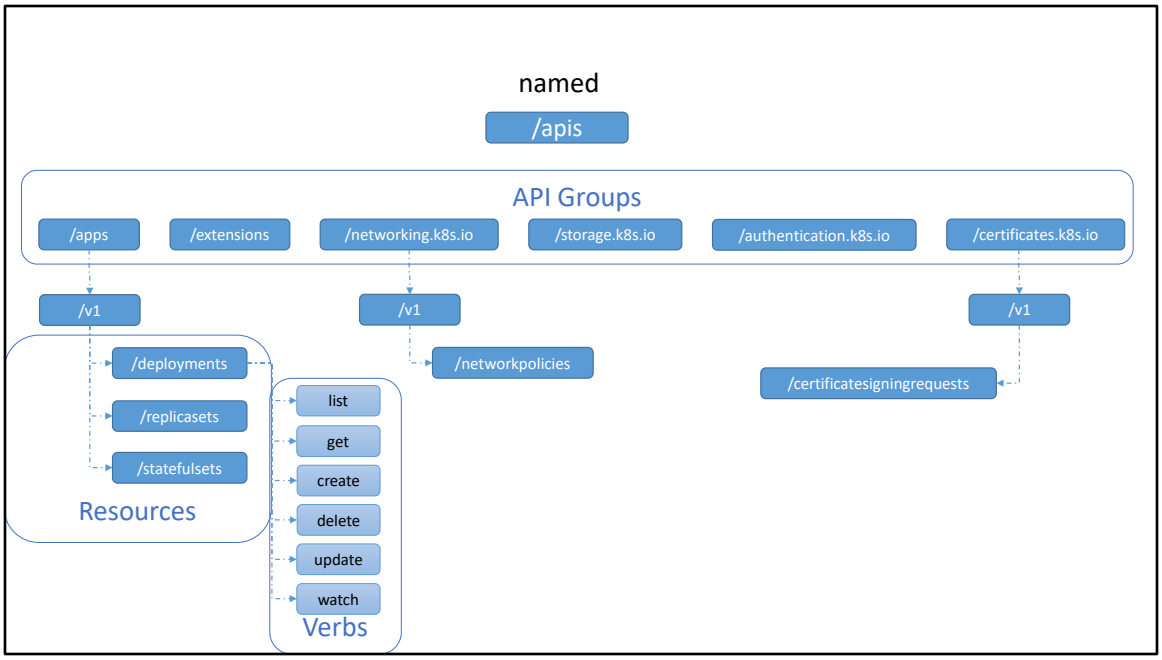


These APIs are categorized into two. The core group and the named group. The core group is where all core functionality exists. Such as namespaces, pods, replication controllers, events, endpoints, nodes, bindings, Persistent volumes, persistent volume claims, configmaps, secrets, services etc.

The named group contains



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, endpoints, nodes, bindings, Persistent volumes, persistent volume claims, configmaps, secrets, services etc.

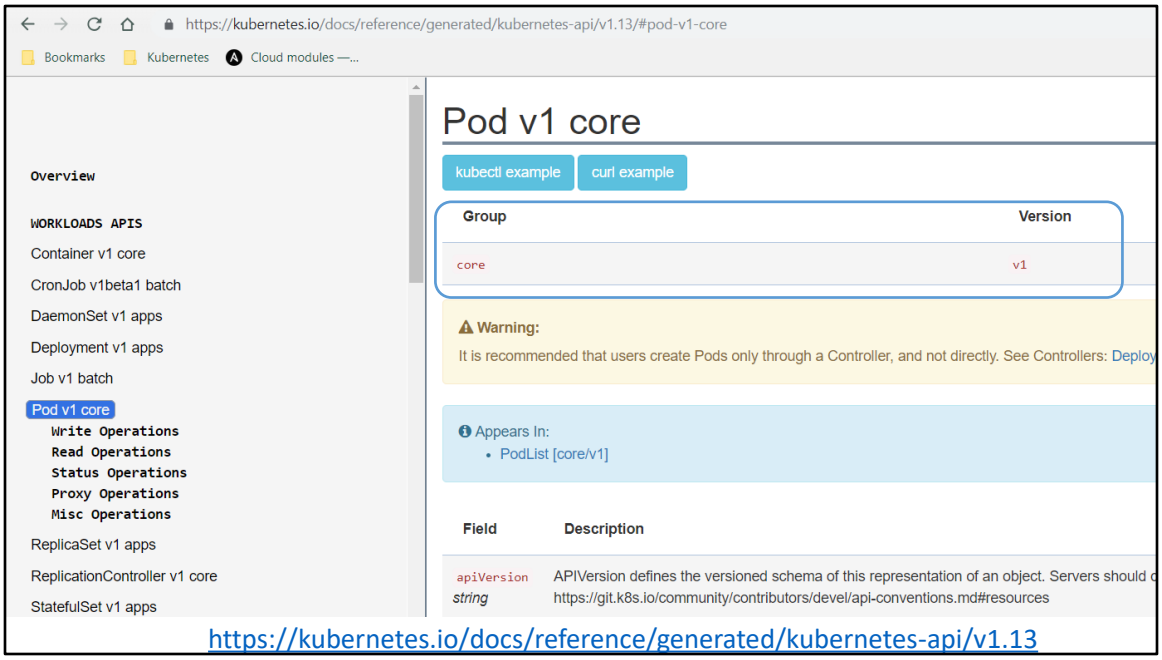


The named group apis is more organized. And going forward all the newer features are going to be made available through these. It has groups under it for apps, extensions, networking, storage, authentication, authorization certificates etc. Shown here are just a few. Within apps, you have deployments, replicasets, statefulsets etc.

Networking has network policies, certificates have the certificatesigningrequests that we talked about earlier in this section.

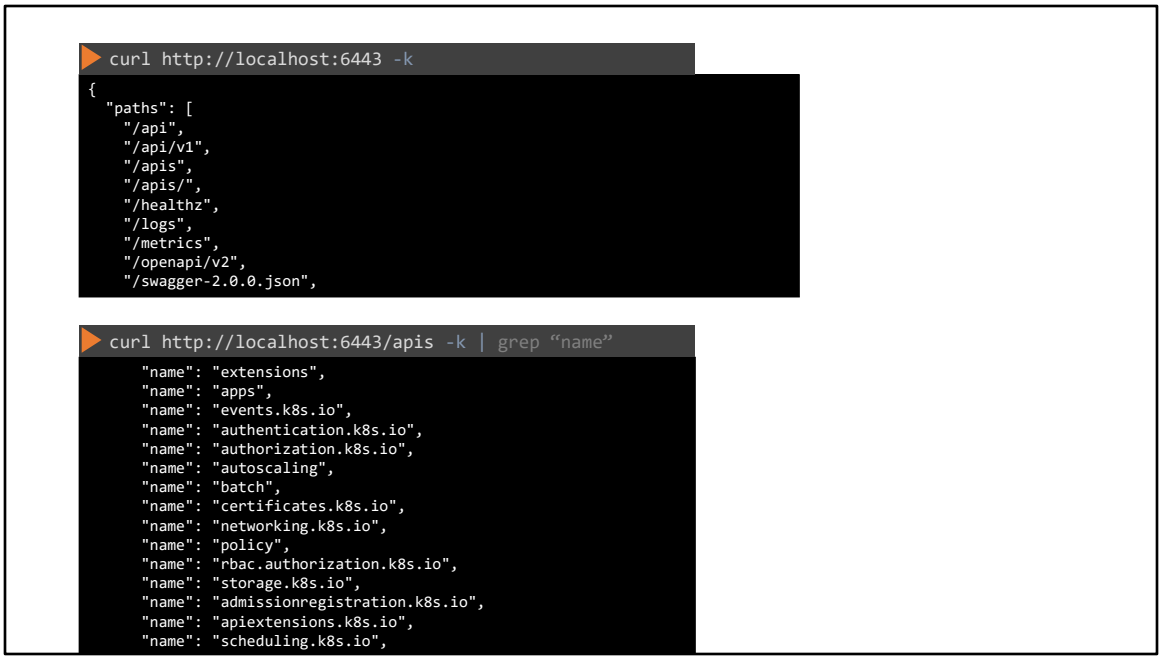
So the ones at the top are <c> API Groups and the ones at the bottom are <c> resources in those groups.

Each resource in this has a set of actions associated with it. 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 etc. These are known as verbs.

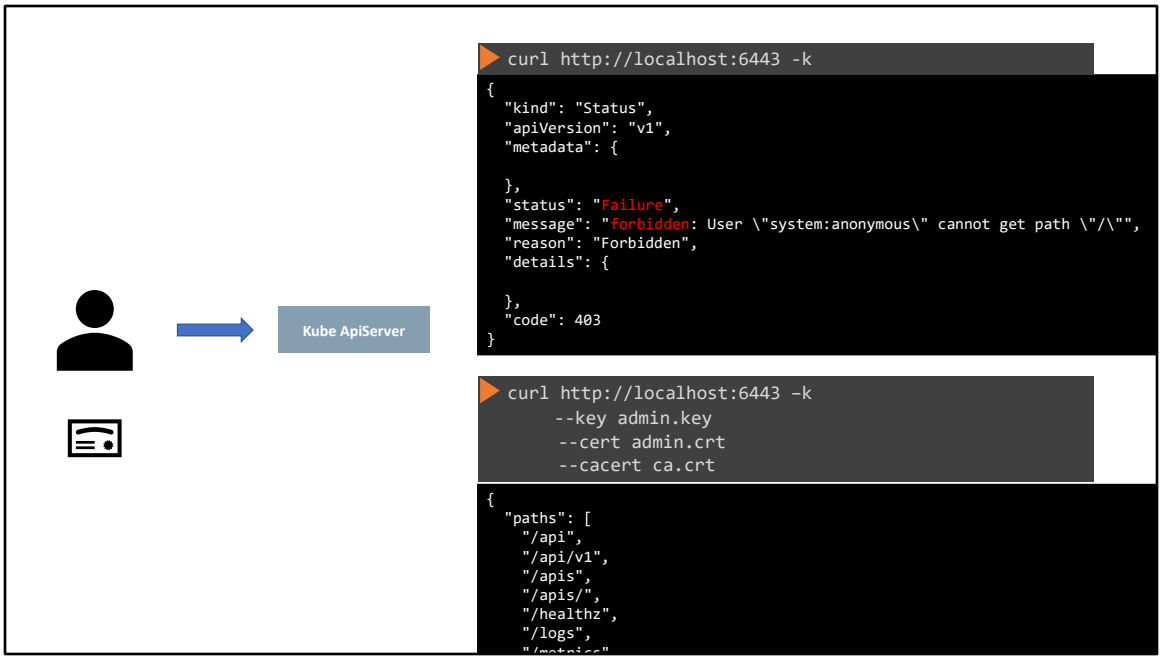


https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.13

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.

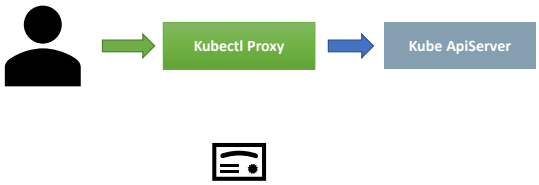


You can also view these on your Kubernetes cluster. Access your kube-api server at port 6443, without any path and it will list you the available api groups. And then within the named api groups it returns all supported groups.



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 certificate files by passing them in the command line like this.

kubectl proxy

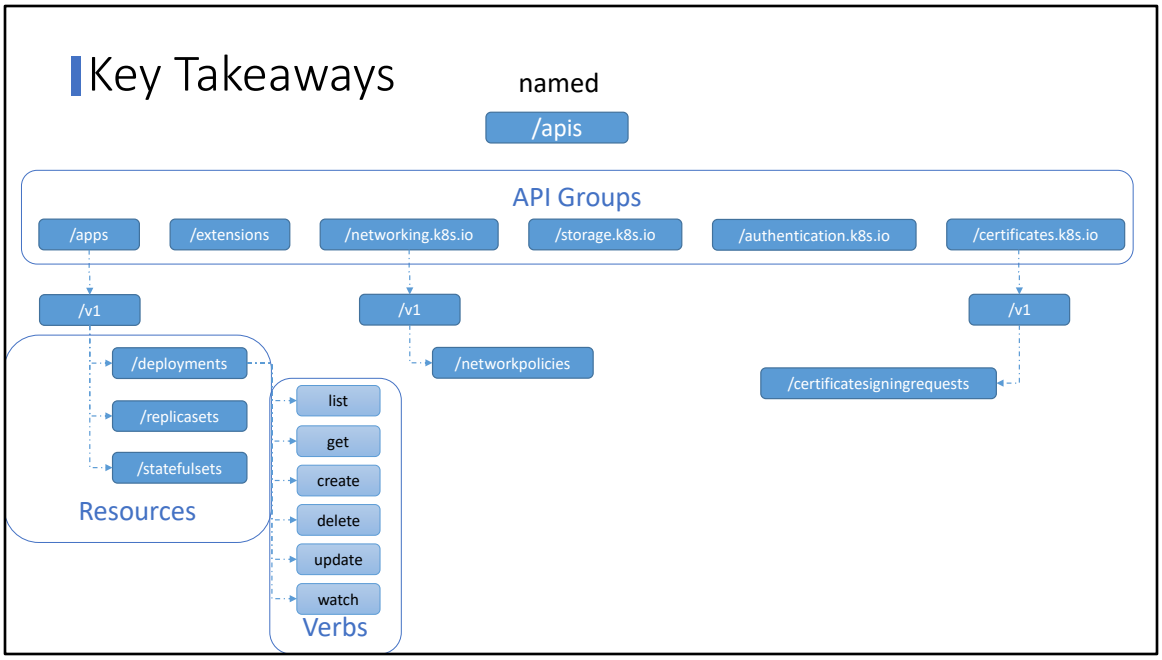



```
kubectl proxy
Starting to serve on 127.0.0.1:8001
```

```
curl http://localhost:8001 -k
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/",
    "/healthz",
    "/logs",
    "/metrics",
    "/openapi/v2",
    "/swagger-2.0.0.json",
```

An alternate option is to start a kubectl proxy client. <c> The kubectl proxy command, launches a proxy service locally on port 8001 and uses credentials and certificates from your kubeconfig file to access the cluster. That way you don't have to specify those in the curl command. <c> Now you can access the kubectl 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.

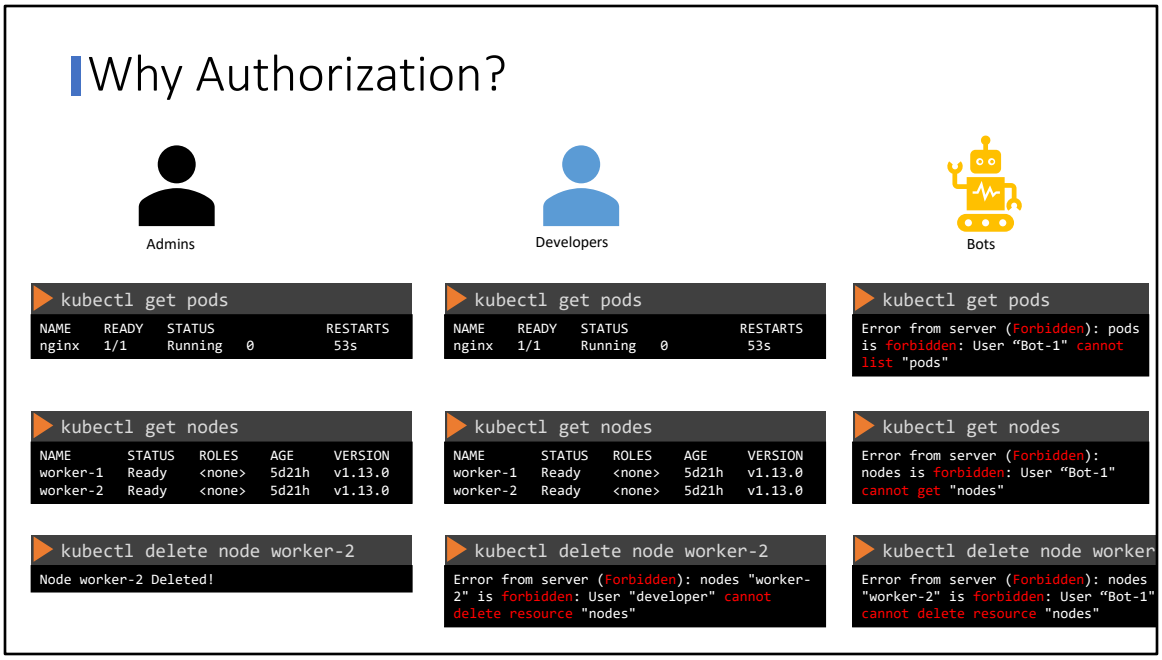
I want to point out 2 terms that kind of sound the same. Kube proxy and kubectl proxy. They are not the same. We discussed about kube-proxy earlier in this course. It is a used to enable connectivity between pods and services across different nodes in the cluster. We discuss about kube-proxy in much more detail later in this course. Whereas kubectl proxy is an HTTP proxy service created by kubectl 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 different resources. 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 get access to a cluster. We saw different ways that someone, a human or machine can get access to the cluster. Once they gain access, what can they do? That's what Authorization defines.



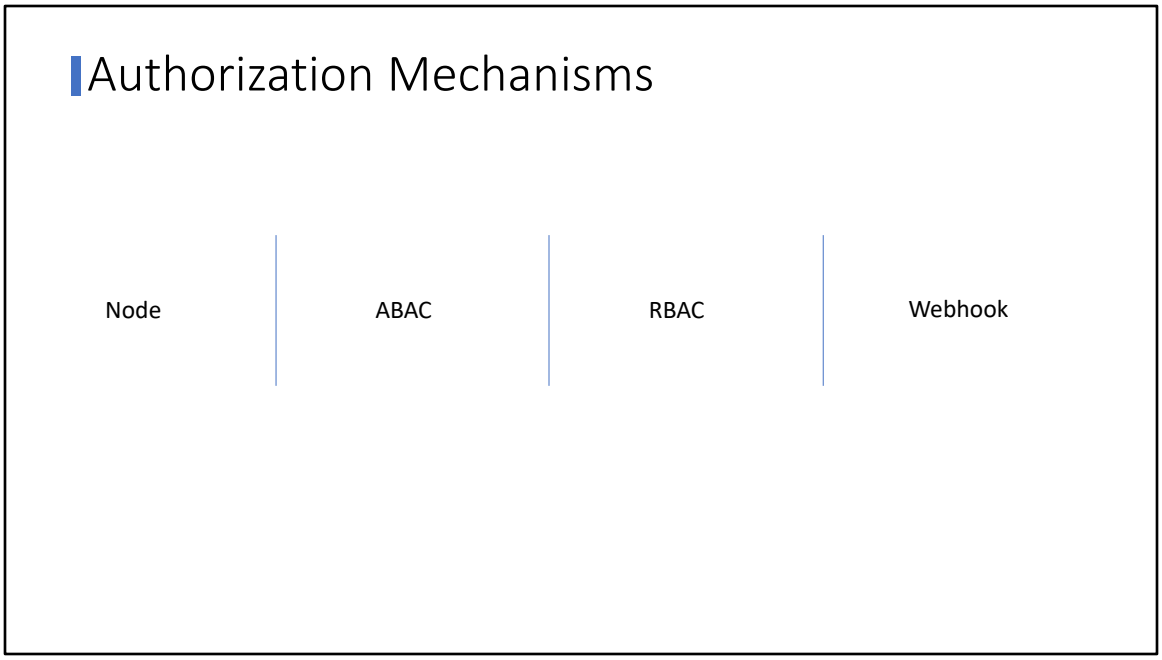
First of all why do you need Authorization in your cluster? As an administrator of the cluster, we were able to perform all sorts of operations in it. Such as viewing various objects <c> like nodes, pods or deployments or secrets, creating or deleting objects such as adding or deleting pods, or even 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 other administrators, developers, testers or other applications like monitoring applications or Continuous delivery applications like jenkins etc.

<c> So we will be creating accounts for them to access the cluster, by creating usernames and passwords or tokens, or signed TLS certificates or service accounts as we saw in the previous lectures in this section.

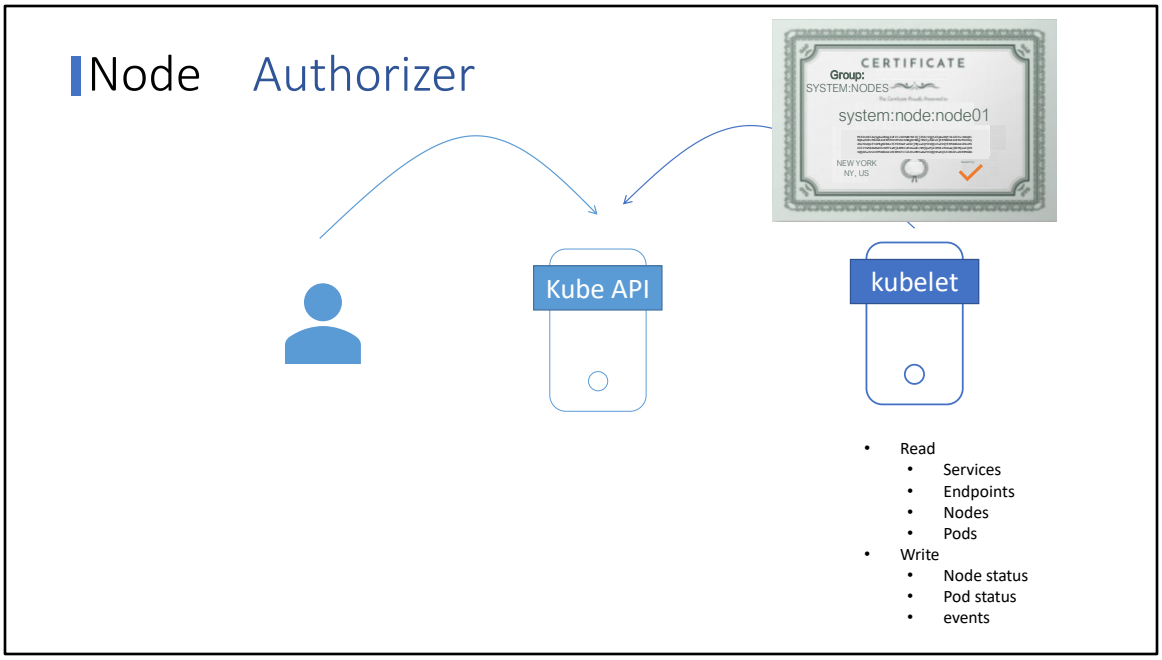
But we don't want all of them to have the same level of access as us. For example, <c> we don't want the developers to have access to modify our cluster configuration like adding or deleting nodes. Or the storage or networking configurations. We can allow them to view, but not modify. But they could have access to deploy applications. The same goes with service accounts, we only want to provide the external application the necessary access to perform its operations.

When we share our cluster between different organizations or teams by logically partitioning it using namespaces, we want to restrict access to the users to their namespaces alone. Again that is why you need authorization within the cluster.

Authorization Mechanisms



There are different authorization mechanisms supported by kubernetes. Such as Node Authorization, authorization, attribute based authorization, role based authorization and Webhook. Let us go through these now.



We know that the kube-api server is accessed by <c> users like us for management purposes, as well as the <c> kubelets on nodes within the cluster for management purposes within the cluster. <c> The kubelet accesses the api-server to read information about services, endpoints, nodes, pods etc. The kubelet also reports to the kube-api server with information about the ndoe such as its status, the status of the pods, events etc. These requests are handled by a special authorizer known <c> 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 requests 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.

ABAC

dev-user

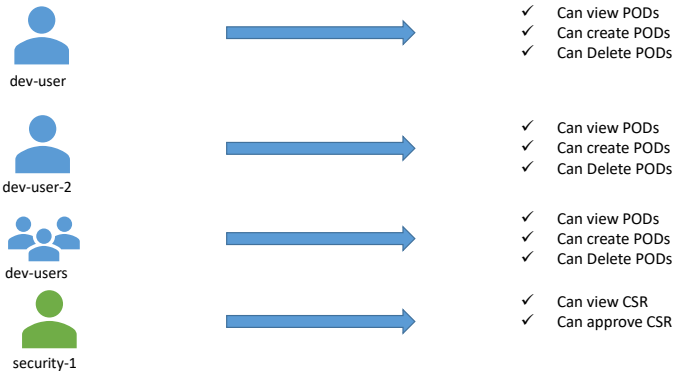
- ✓ Can view PODs
- ✓ Can create PODs
- ✓ Can Delete PODs

```
{"kind": "Policy", "spec": {"user": "dev-user", "namespace": "*", "resource": "pods", "apiGroup": "*"}}
```

Let's talk about external access to the API. For instance a user. Attribute based authorization is were you associate a user or a group of users, with a policy or a set of permissions. <c> In this case we say the dev-user can view create or delete pods.

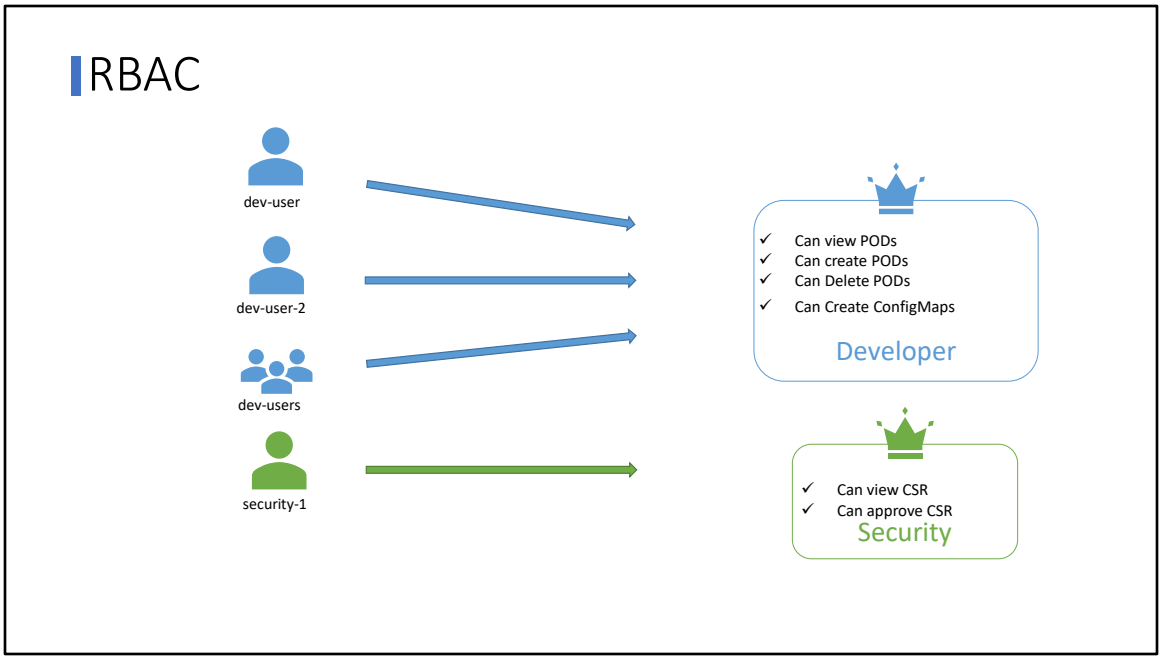
<c> You do this by creating a policy file with a set of policies defined in a JSON format this way. You pass this file in to the API server.

ABAC




```
{"kind": "Policy", "spec": {"user": "dev-user", "namespace": "*", "resource": "pods", "apiGroup": "*"}}
{"kind": "Policy", "spec": {"user": "dev-user-2", "namespace": "*", "resource": "pods", "apiGroup": "*"}}
{"kind": "Policy", "spec": {"group": "dev-users", "namespace": "*", "resource": "pods", "apiGroup": "*"}}
{"kind": "Policy", "spec": {"user": "security-1", "namespace": "*", "resource": "csr", "apiGroup": "*"}}
```

Similarly we create a policy definition for each user or group in the file. Everytime you need to add or make a change you must edit this policy file manually and restart the kube-api server. As such the ABAC configurations are difficult to manage. We will look at RBAC next.



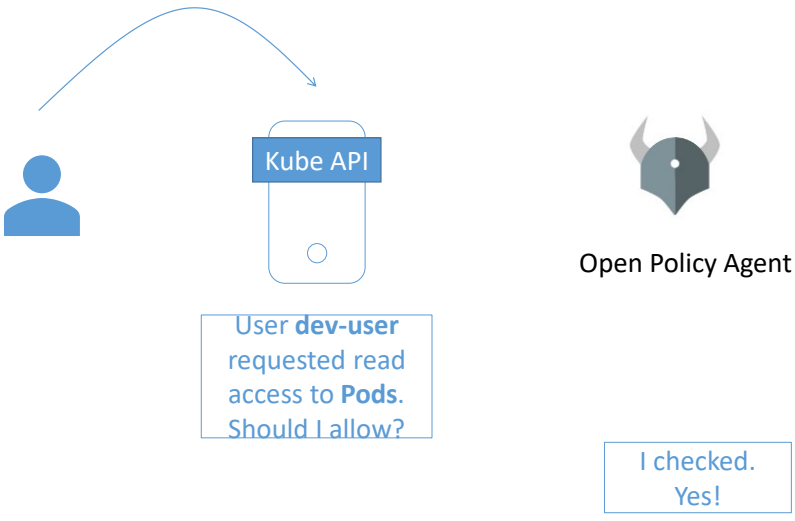
Role based access controls make these much easier. With RBACs, instead of directly associating a user or a group with a set of permissions, we define a role. In this case for developers, we create a role with the set of permissions required for developers. Then we associate all the developers to that 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 users access, <c> we simply modify the role and it reflects on all developers immediately. RBAC's provide a more standard approach to managing access within the Kubernetes cluster.

We will look at RBACs in much more detail in the next lecture.

Webhook



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 about. 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 external application decide if the user should be permitted or not. Based on the response the user is granted access

Authorization Mode

We saw the different authorization modes. <c> There are 2 in addition to this. AlwaysAllow and AlwaysDeny. As the name states, always allow allows all requests without performing any authorization checks. AlwaysDeny 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?

Authorization Mode

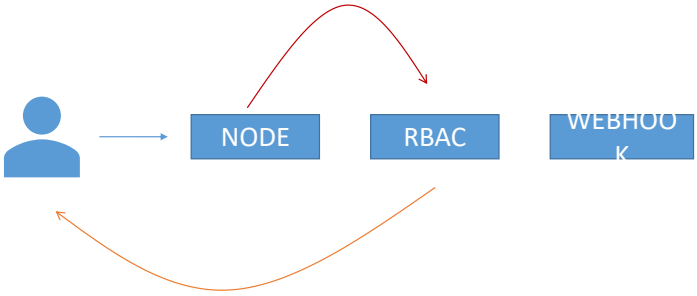
AlwaysAllow | NODE | ABAC | RBAC | WEBHOOK | AlwaysDeny

```
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
```

The modes are set using the <c> authorization-mode option on the kube-apiserver. <c> If you don't specify this option it is set to AlwaysAllow by default.

You may <c> 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.

Authorization Mode




```
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 \\
```

When you have multiple modes configured, your request is authorized using each one in the order it is specified. For example, <c> when a user sends a request, it's first handled by the node authorizer. The node authorizer handles only node requests, <c> so it denies the request. Whenever a <c> module denies a request it is forwarded to the next one in the chain. The RBAC module performs its check and grants the user permission.

So everytime a module denies a request it goes to the next one in chain. And as soon as a request is approved 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 look at Role Based Access Controls in much more detail.

RBAC

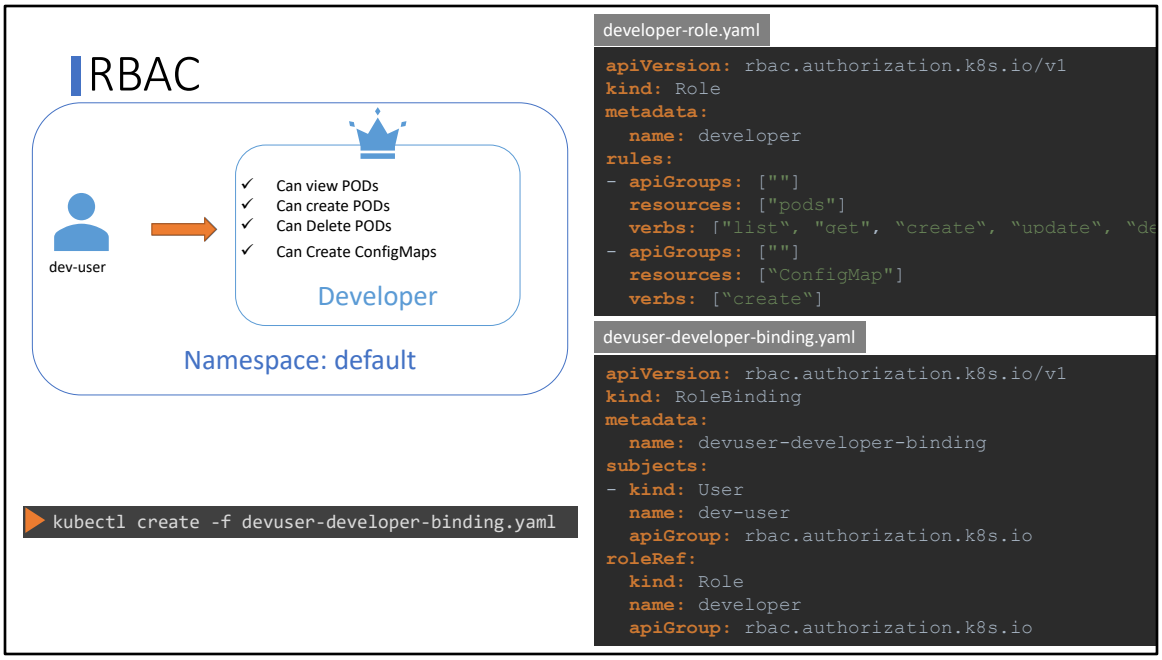
developer-role.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: developer
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["list", "get", "create", "update", "delete"]
- apiGroups: [""]
  resources: ["ConfigMap"]
  verbs: ["create"]
```

```
kubectl create -f developer-role.yaml
```

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 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- apiGroups, resources, and verbs. The same things that we talked about in the previous lecture. For core group you can leave it blank. The resources that we want to give developers access to are pods. The actions that they can take are list, get, create and delete. To allow the developers to create configmaps, we add another rule to create ConfigMap. You can add multiple rules for a single role like this.

Create the role using the kubectl create role command.



<c> The next step is to link the user to that role. For this we create another object called RoleBinding. The role binding object links the user object to the role. We will name it devuser-developer-binding.

The kind is RoleBinding. It has two sections. The subjects is where we specify the user details. The roleRef section is where we provide the details of the role we created.

Creating the role binding using the kubectl create command.

Also note that the roles and rolebindings fall under the scope of namespaces. So here the dev-user gets access to pods and configmaps within the default namespace. If you want to limit the dev-users access within a different namespace, then specify the namespace within the metadata of the definition file while creating them.

View RBAC

```
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]
```

To view the created roles run the kubectl get roles command. To list role bindings run the kubectl get rolebindings command. To view more details about the role, run the kubectl describe role developer command. Here you see details about the resources and permissions for each resource.

View RBAC

```
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
```

Similarly to view details about rolebindings run the kubectl describe rolebindings command. Here you can see details about an existing role binding.

Check Access

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

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

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

```
kubectl auth can-i create pods --as dev-user
yes
```

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

[pause-2]

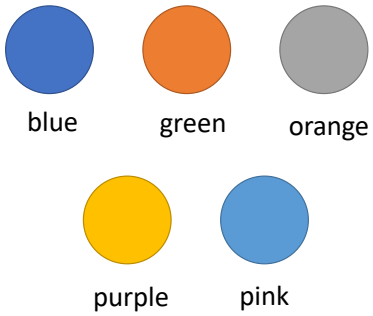
What if you, being a user, would like to see if you have access to a particular resource in the cluster? You can use the kubetl auth can-I command and check if you can, say create deployments. Or say delete nodes.

If you are an administrator then you can even impersonate another use 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 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 namespace in the command like this. The dev-user does not have permission to create a pod in the test namespace.

Well that's it for this lecture. Head over to the practice tests and practice viewing, configuring and troubleshooting Role based access controls in Kubernetes.

Resource Names



developer-role.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: developer
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "create", "update"]
  resourceNames: ["blue", "orange"]
```

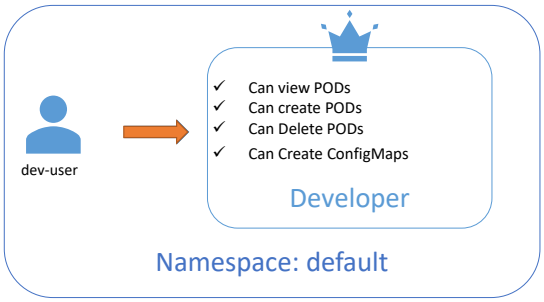
A quick note on resource names. We just saw how you can provide access to users for resources like pods within a namespace. You can go one level down, and allow access to specific resources alone.

For example, say you have 5 pods in a 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 resourceNames field to the rule.

Finally a note on resource names.

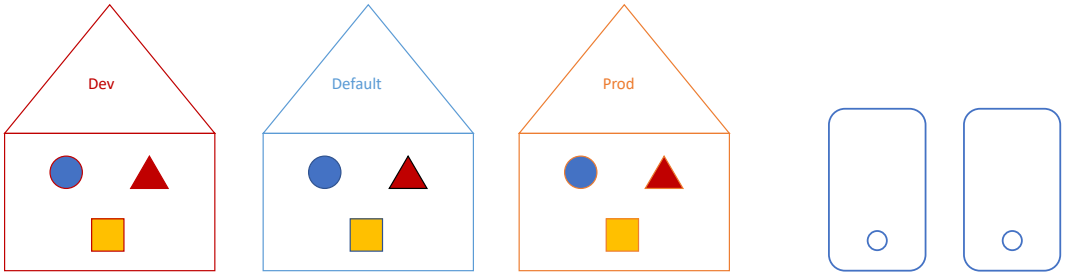
In this lecture we will talk about Cluster Roles and bindings.

Roles



We discussed about Roles and Role Bindings in the previous lecture. In this lecture we will talk about cluster roles and cluster role bindings. We also 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.

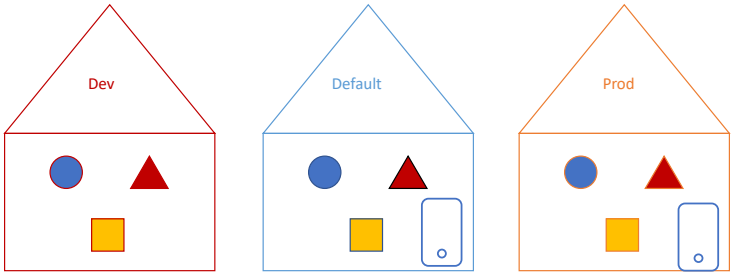
Namespace



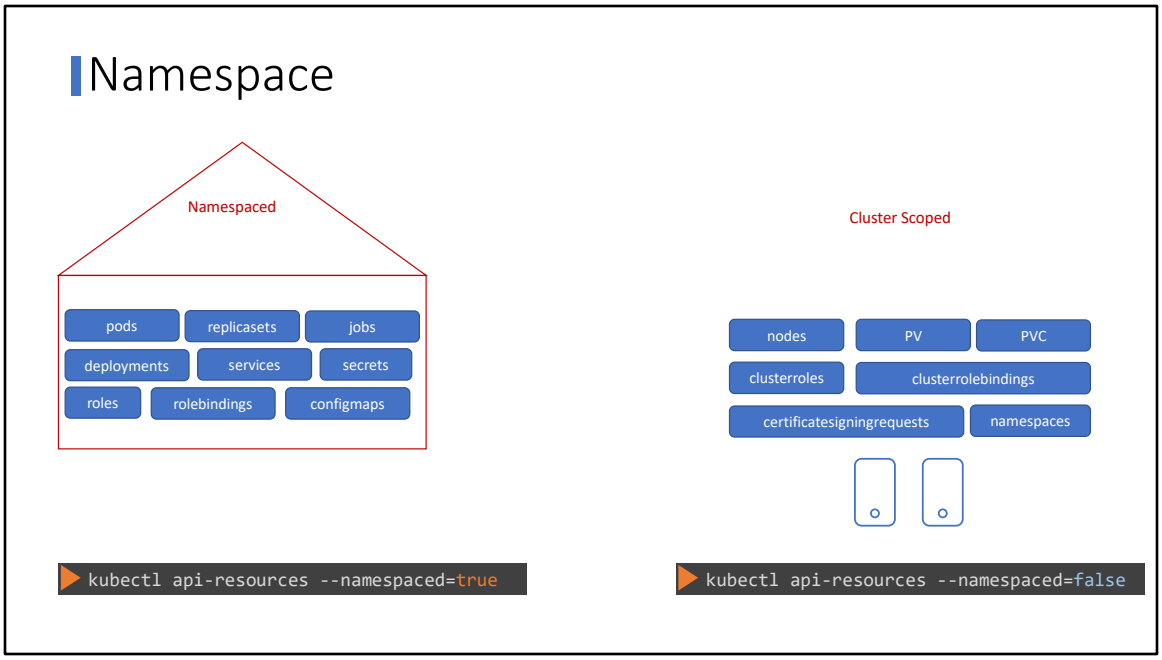
In one of the previous lectures we discussed about namespaces and how it helps in grouping or isolating resources like pods, deployments and services.

<c> But what about other resources like nodes?

Namespace



Can you group or isolate nodes within a namespace? No, those are cluster wide or cluster scoped resources. They cannot be associated to any namespaces. So the resources are categorized as either namespaced or cluster scoped.

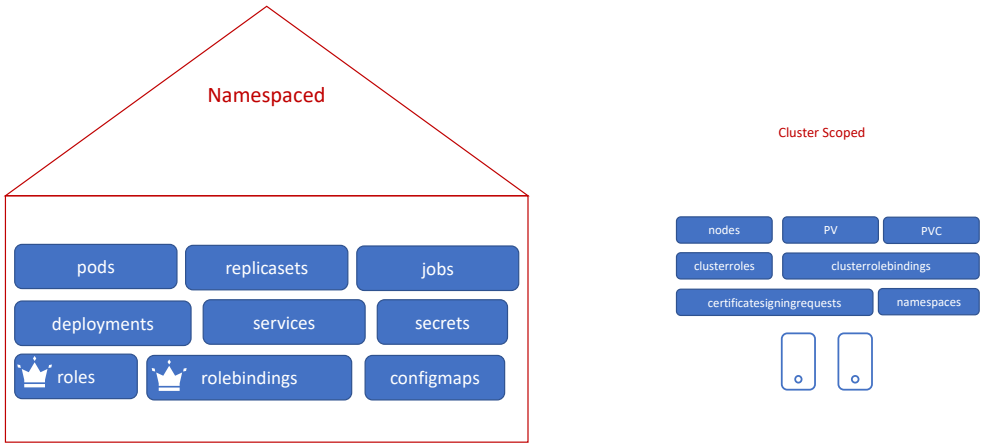


We have seen a lot of namespaced resources throughout this course. Like pods, replicasets, jobs, deployments, services, secrets, and in the last lecture we saw 2 new. Roles and Rolebindings. These resources are created in the namespace you specify when you create them. If you don't specify a namespace they are created in the default namespace. To view them or delete them or update them, you always specify the right namespace.

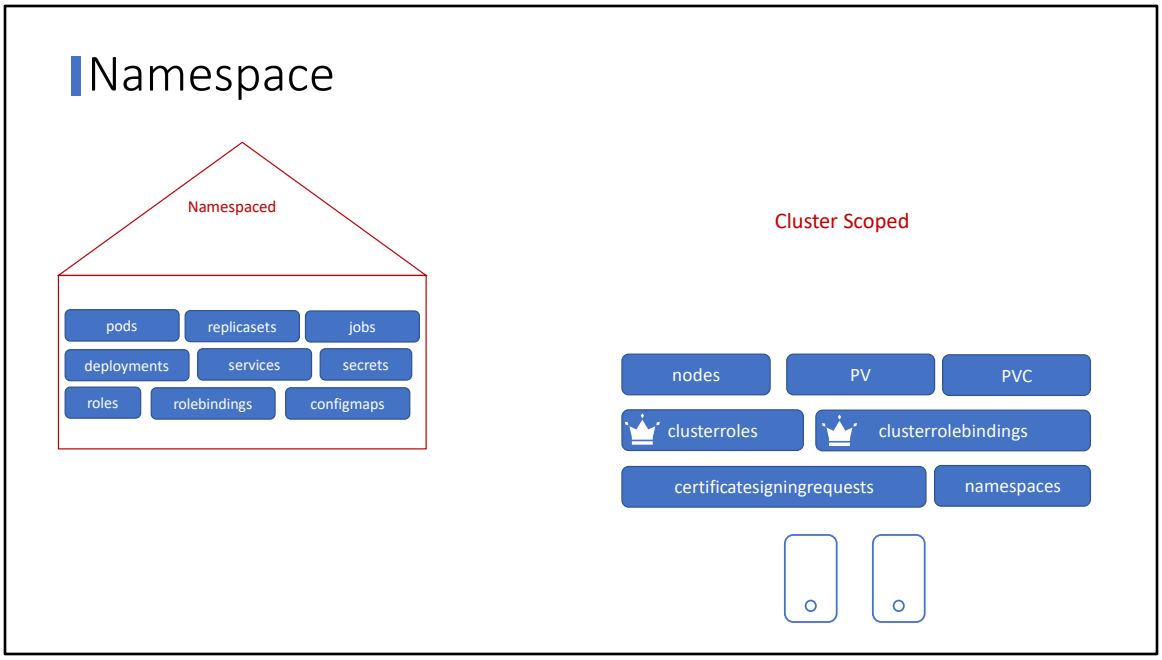
<c> The cluster scoped resources are those where you don't specify a namespace when you create them. Like nodes, persistent volumes, persistent volume claims, the clusterroles and clusterrolebindings that we are going to look at in this lecture, certificatesigningrequests we saw earlier and namespace objects themselves are not, of course, namespaced.

<c> Note that this is not a comprehensive list of resources. To see a full list of namespaced and non namespaced resources run the kubectl api-resources command with the namespaced option set.

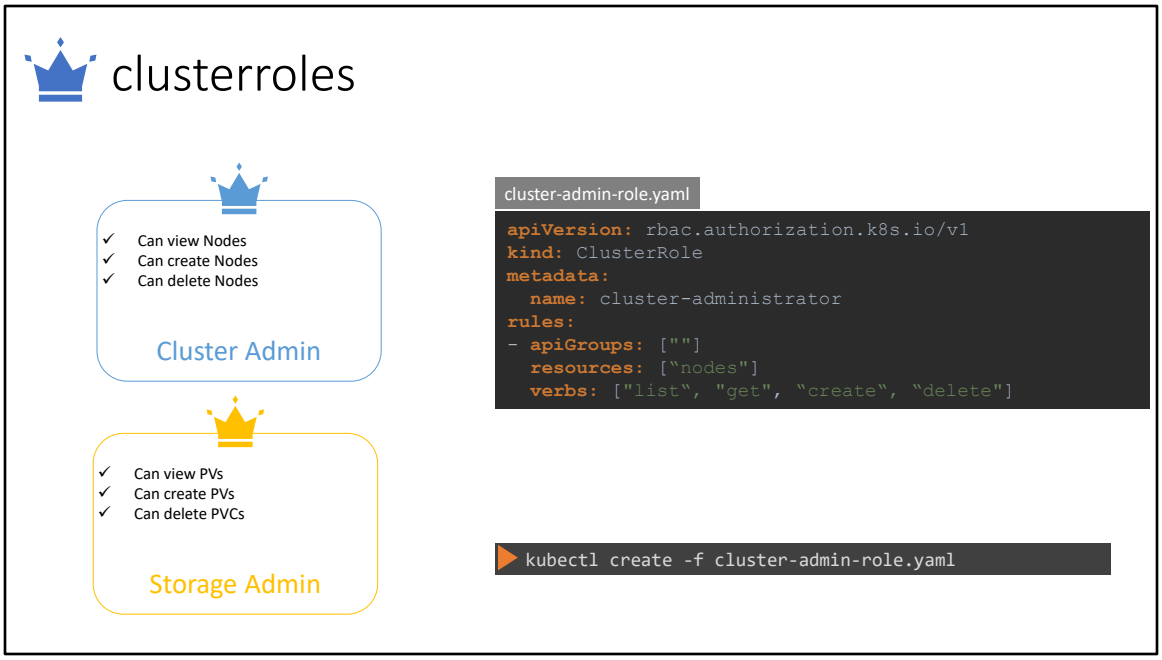
Namespace



In the previous lecture we saw how to authorize a user to namespaced resources liked pods and configmaps. We used Roles and Rolebindings that were namespaced too. But how do we authorize users to cluster wide resources likes nodes, or persistent volumes or pvcs?



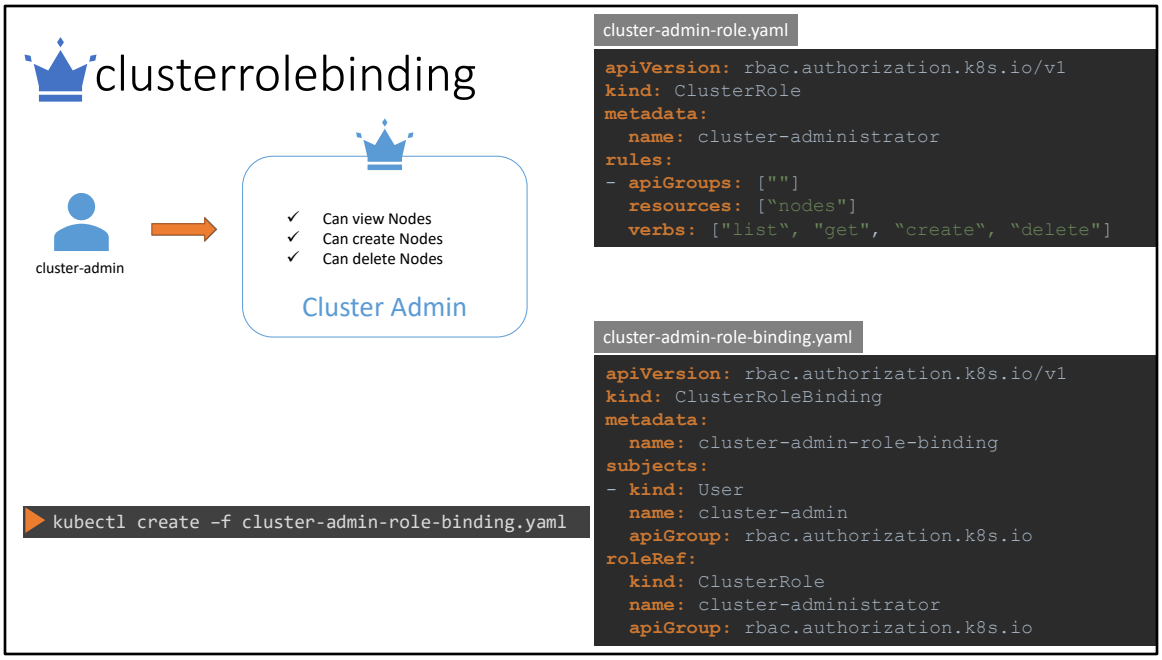
Well that is where you use cluster roles and cluster role bindings.



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.

<c> Similarly a storage administrator cluster role can be created to authorize a storage admin to create persistent volumes and claims.

<c> Create a cluster role definition file with the kind ClusterRole and specify the rules as we did previously. In this case the resources is nodes. Then create the ClusterRole.

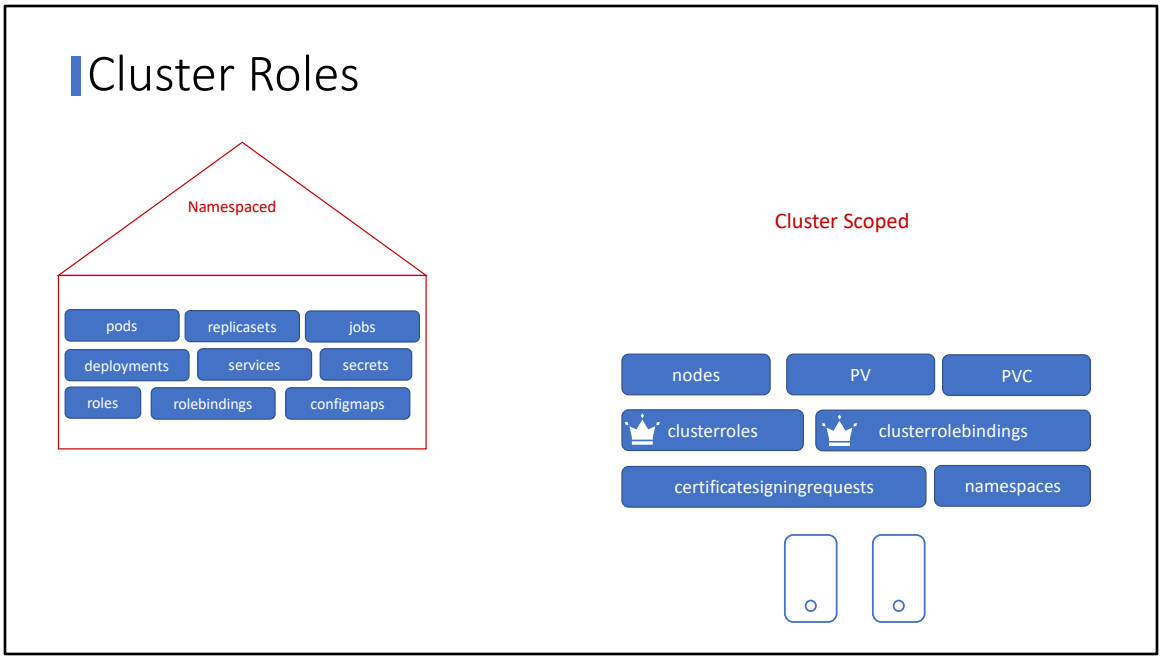


<c> The next step is to link the user to that ClusterRole. For this we create another object called ClusterRoleBinding this time. The role binding object links the user object to the role. We will name it cluster-admin-role-binding.

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.

Creating the role binding using the kubectl create command.

Also note that the roles and rolebindings fall under the scope of namespaces. So here the dev-user gets access to pods and configmaps within the default namespace. If you want to limit the dev-users access within a different namespace, then specify the namespace within the metadata of the definition file while creating them.



One thing to note. 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 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 pods, the user had access to the 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.

Kubernetes creates a number of cluster roles by default when the cluster is first setup. We will explore those in the practice tests coming up. Good luck!

In this lecture we will talk about securing images.

Image

nginx-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: nginx
```

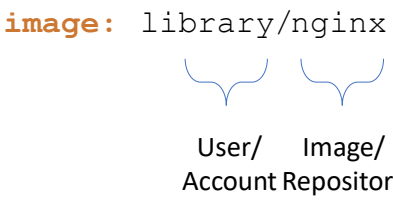
We will start with basics of image names and then work our way towards secure images repositories and how to configure your pods to use images from secure repositories. We deployed a lot of different kinds of pods hosting different kinds of applications throughout this course, like webapps, databases, redis cache etc. Let's look at a pod definition file for instance.

Here we have used the nginx image to deploy an nginx container.

Image

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.

Image

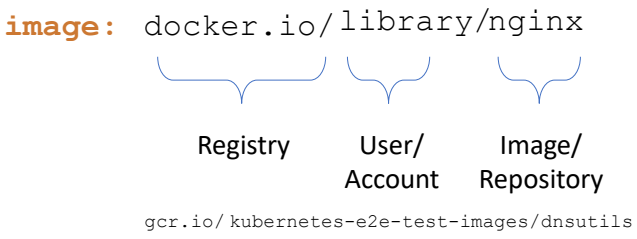


When you say nginx, its actually library/nginx. The first part stands for the user or account name. So if you don't provide a repository name, it assumes it to be library. Library is the name of the default account where the Docker official images are stored. These images promote best practices and are maintained by a dedicated team who are responsible for reviewing and publishing these official images.

<c> If you where to create your own account and create your own repositories or images under it, then you would use a similar pattern. Instead of library it would be your or your companies name.

Now where are these images stored and pulled from?

Image



Since we have not specified the location where these images are to be pulled from, it is assumed to be on docker's default registry – dockerhub. 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 <c> registry is at gcr.io, where a lot of Kubernetes related images are stored. Like the ones <c> 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.

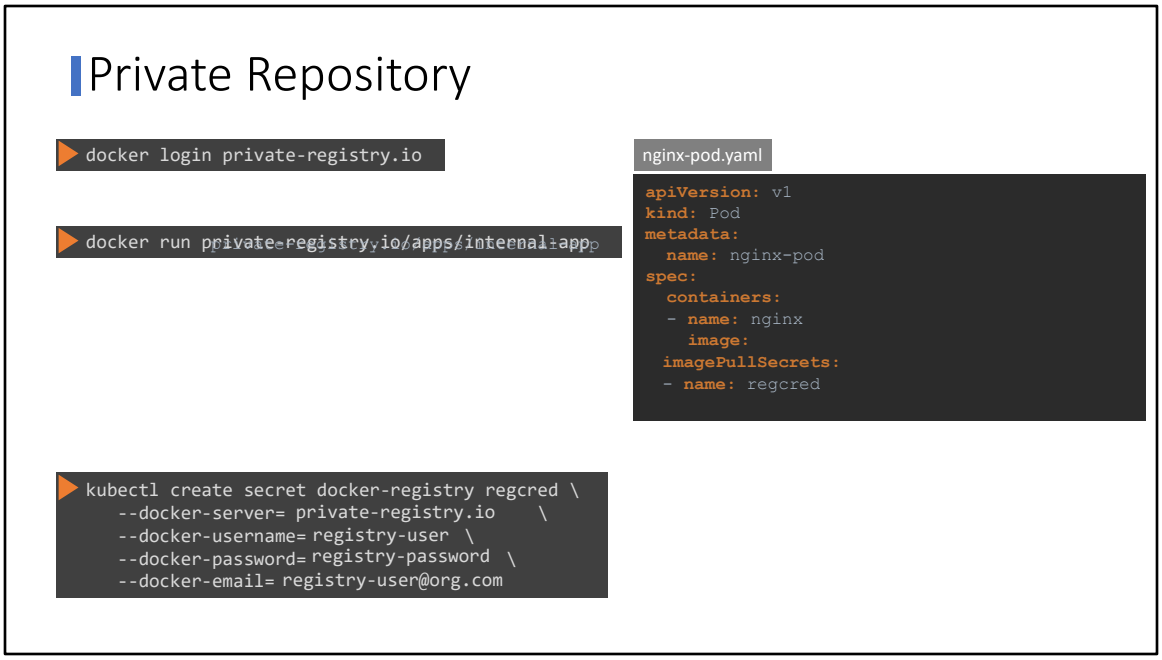
Private Repository

```
docker login private-registry.io
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to
https://hub.docker.com to create one.
Username: registry-user
Password:
WARNING! Your password will be stored unencrypted in /home/vagrant/.docker/config.json.

Login Succeeded
```

```
docker run private-registry.io/apps/internal-app
```

On any of these solutions, be it on Docker hub or googles registry, or your internal private registry, you may chose to make a repository private, so that it can be accessed ony using a set of credentials.

From a Docker perspective to run a container using a private image, you first login to your private-registry using the docker login command. Input your credentials. Once successful run the application using private-registry.



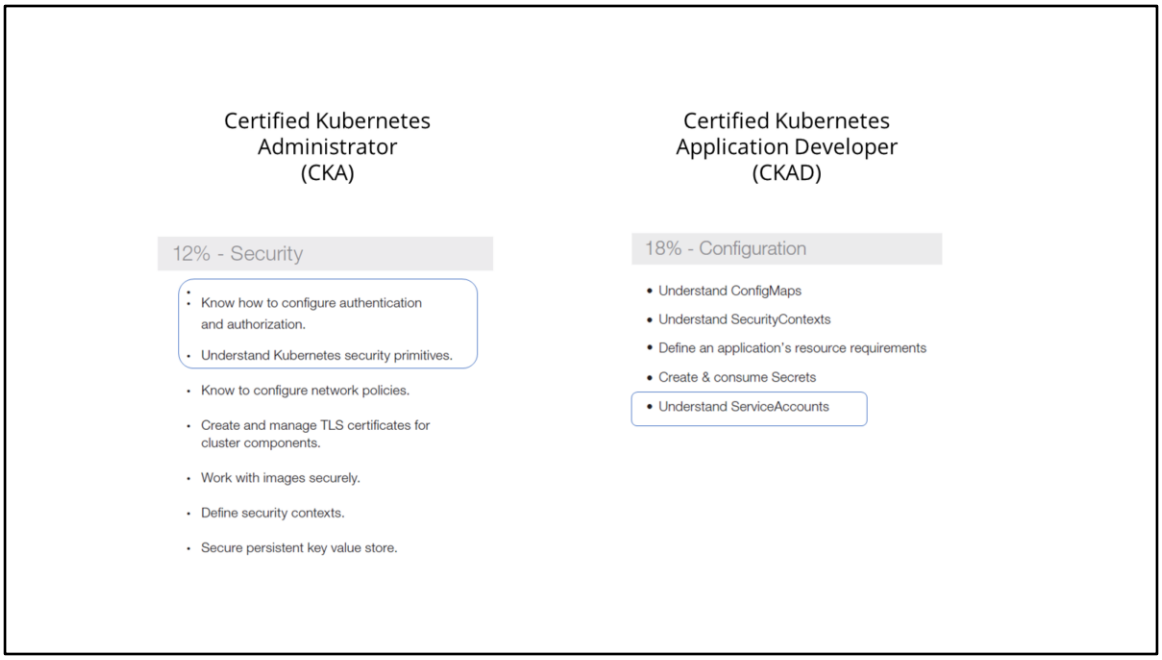
Going back to our pod definition file, to use an image from our private registry we replace the image name with the full path to the one in the private registry. But how do we implement the login? How does Kubernetes get the credentials to access the private registry?

We <c> first create a secret with the credentials. The secret is of type docker-registry and we name it regcred. <c> We then specify the register server name , the username to access the registry, the password and the email address of the user.

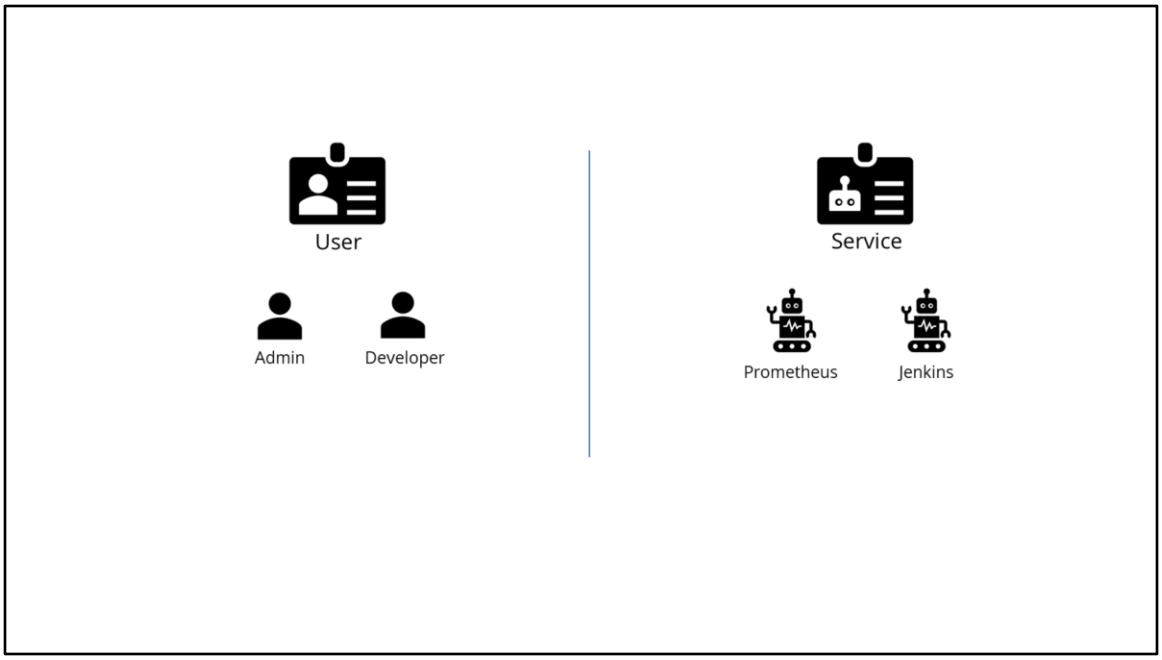
<c> We then specify the secret inside our pod definition file under the imagePullSecrets section. When the pod is created kubernetes uses the credentials from this secret to pull images.

Well that's it for this lecture.

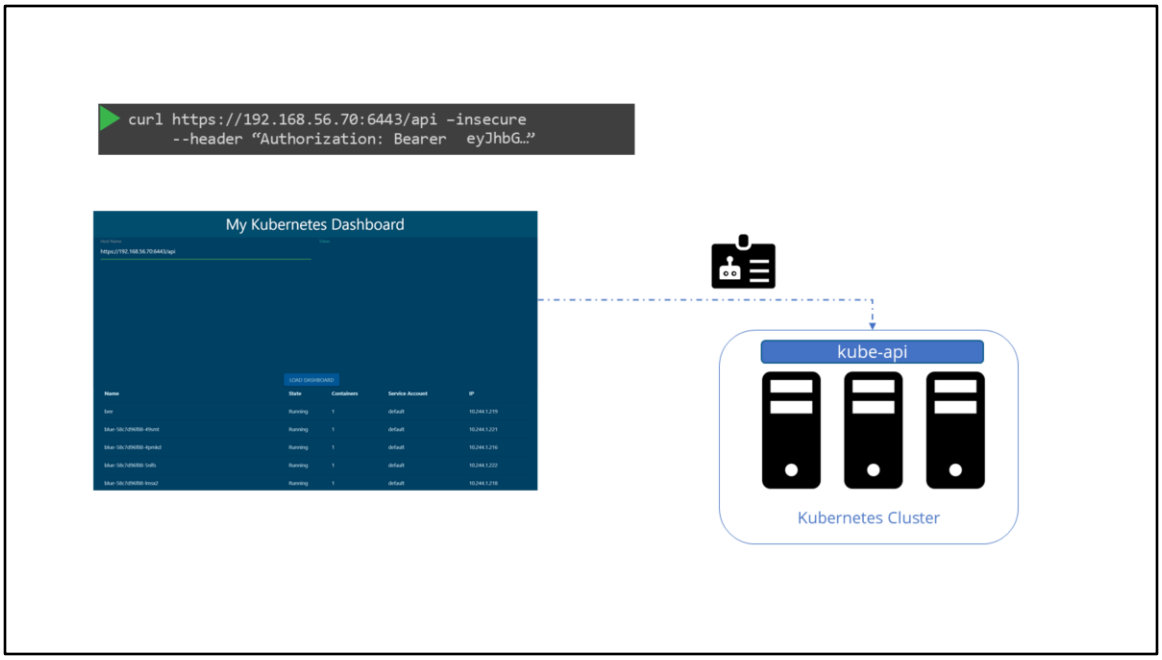
The concept of service accounts is linked to other security related concepts in kubernetes such as Authentication, Authorization, Role based access controls etc.



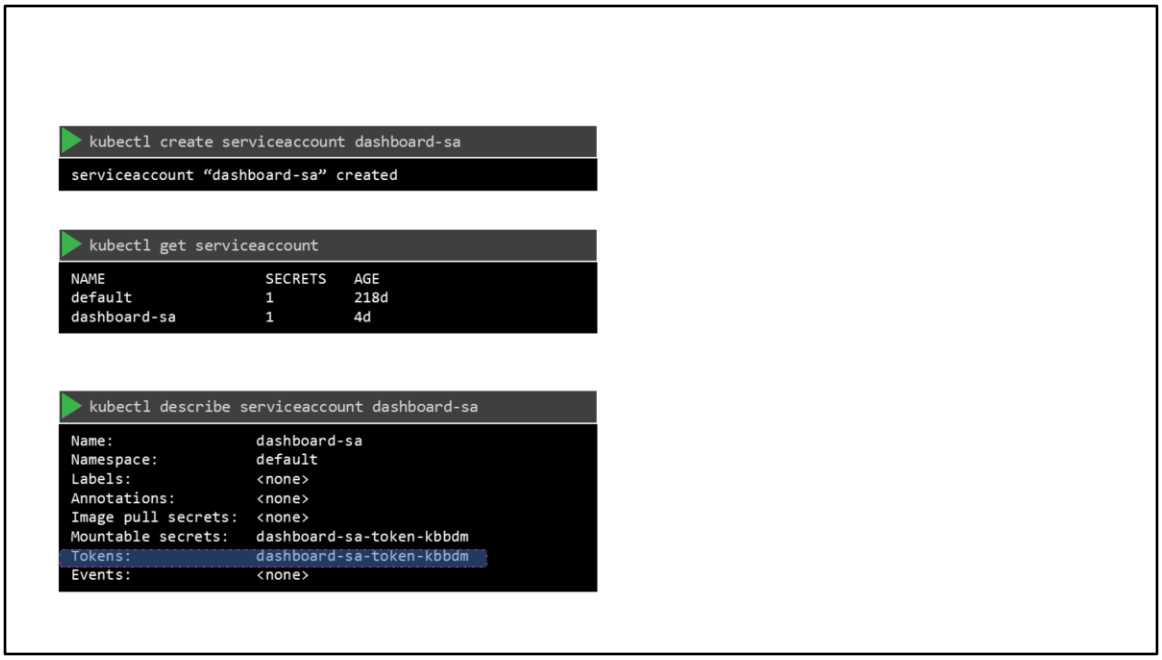
However, as part of the Kubernetes for the Application Developers exam curriculum, you only need to know how to work with Service Accounts. We have detailed sections covering other concepts in security in the Kubernetes Administrators course.



<click> So there are two types of accounts in Kubernetes. <click> A user account and a service account. <click> As you might already know, the user account is used by humans. And service accounts are used by machines. <click>A user account could be for an administrator accessing the cluster to perform administrative tasks, <click> a developer accessing the cluster to deploy applications etc. A service account, could be an account used by an application to interact with the kubernetes cluster. For example a monitoring application like <click> Prometheus uses a service account to poll the kubernetes API for performance metrics. An automated build tool like <click> Jenkins uses service accounts to deploy applications on the kubernetes cluster.

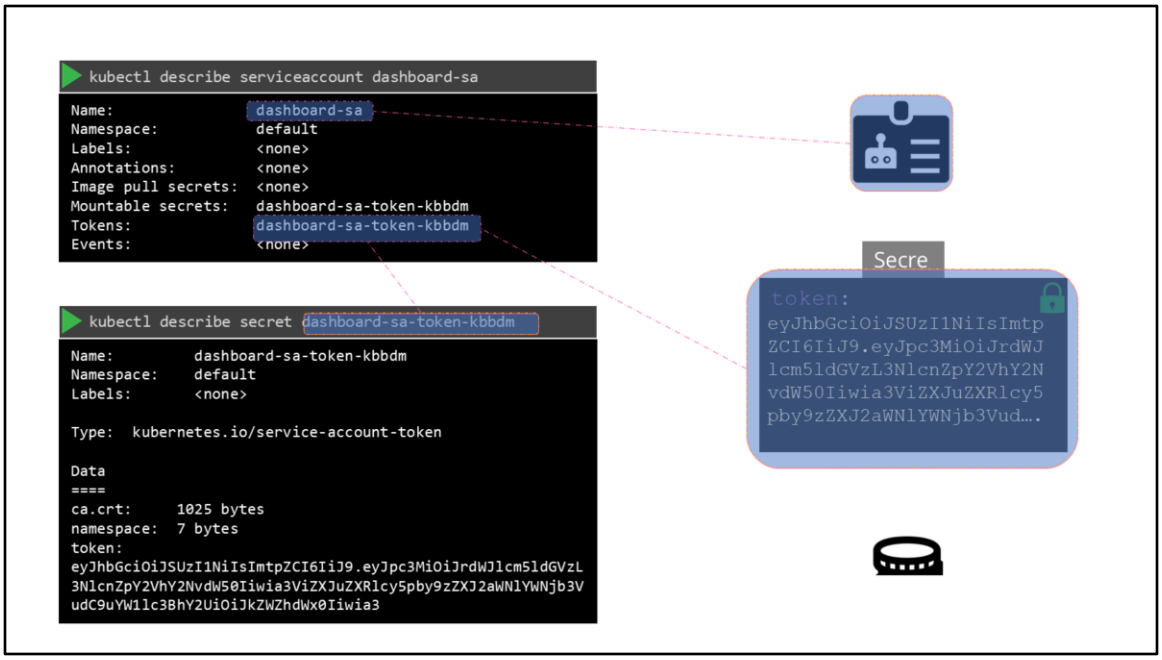


<click> Let's take an example. I have built a simple kubernetes dashboard application named, my-kubernetes-dashboard. It's a simple application built in Python and all that it does when deployed is retrieve the list of PODs on a <click> kubernetes cluster by <click> sending a request to the kubernetes API and display it on a web page. In order for my application to query the kubernetes API, it has to be authenticated. <click> For that we use a service account.

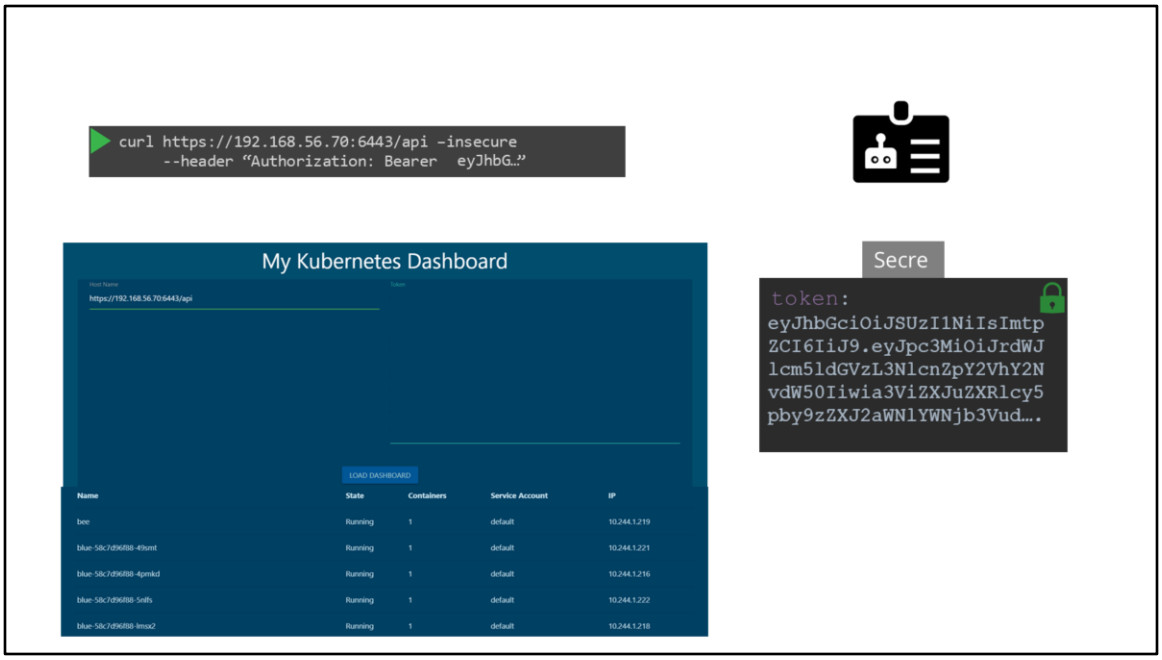


<click> To create a service account run the command kubectl create service account followed by the account name, which is dashboard-sa in this case. <click> To view the service accounts run the kubectl get serviceaccount command. This will list all the service accounts.

<click> When the service account is created, it also creates a token automatically. <click> 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 its named dashboard-sa-token-kbbdm.

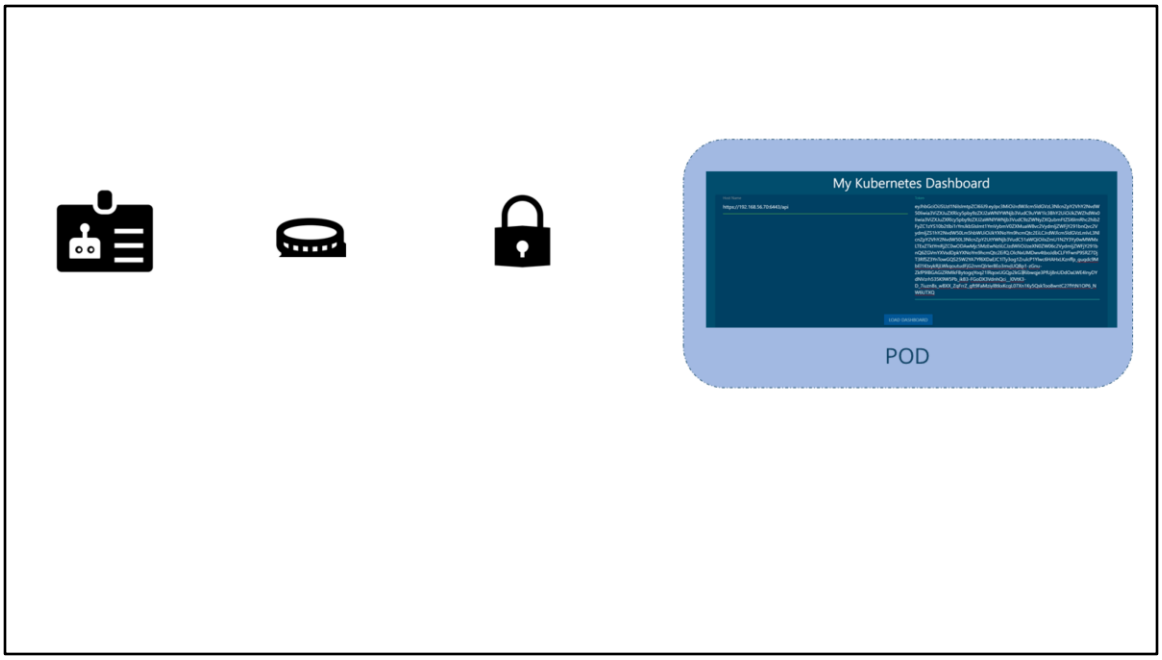


So when a service account is created, <click> it first creates the service account object and <click> then generates a token for the service account. <click> It then creates a secret object and stores that token inside the secret object. <click> The secret object is then linked to the service account. <click> To view the token, view the secret object by running the command kubectl describe secret.

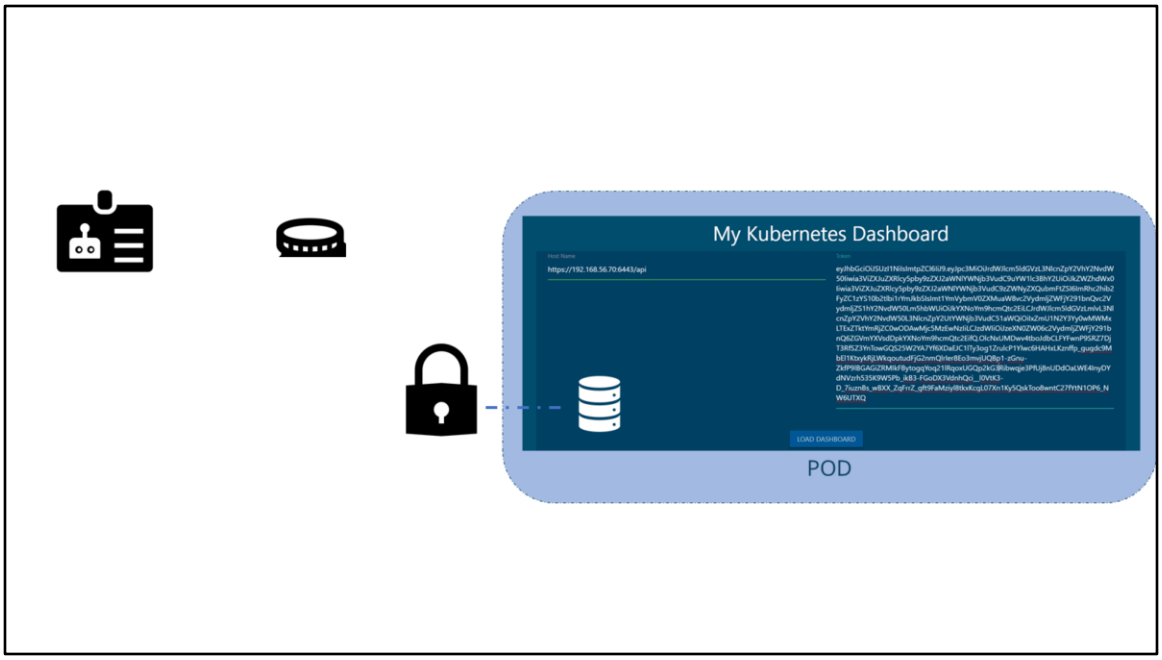


This token can then be used as an authentication bearer token while making a rest call to the kubernetes API. <click> For example in this simple example using curl you could provide the bearer token as an Authorization header while making a rest call to the kubernetes API.

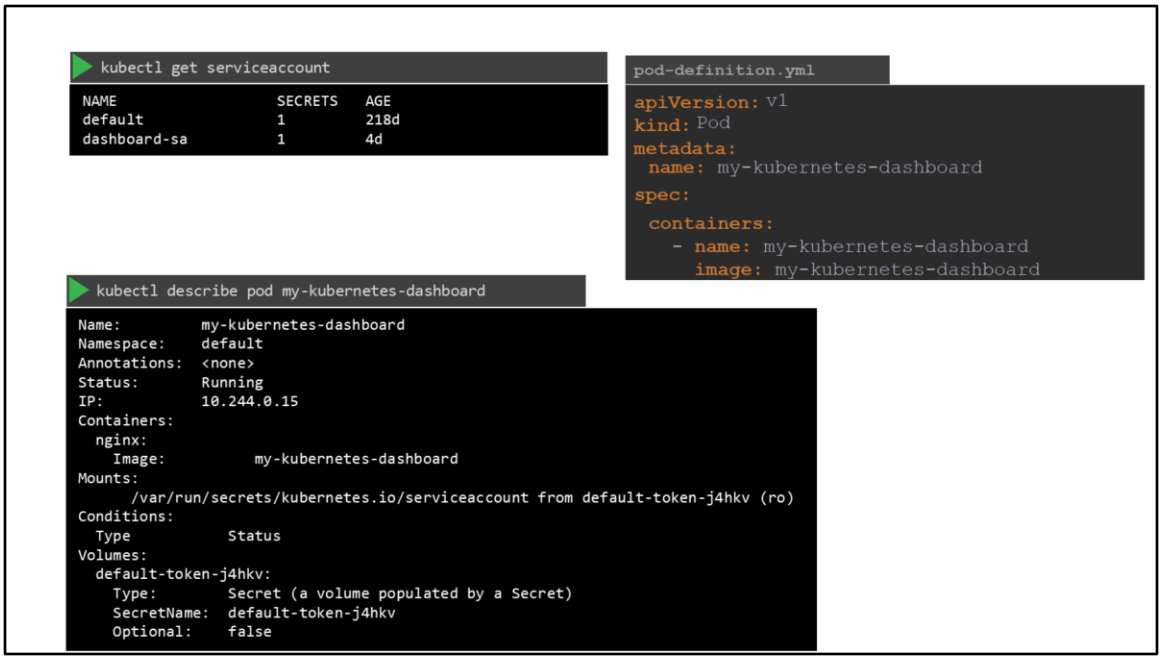
<click> In case of my custom dashboard application, copy and paste the token into the tokens field to authenticate the dashboard application.



<click> So, that's how you create a new service account and use it. You can create a service account, <click> assign the right permissions using Role based access control mechanisms (which is out of scope for this course) and <click> export your service account tokens and <click> use it to configure your third party application to authenticate to the kubernetes API. <click> But what if your third party application is hosted on the kubernetes cluster itself. For example, we can have our custom-kubernetes-dashboard or the Prometheus application used to monitor kubernetes, deployed on the kubernetes cluster itself.

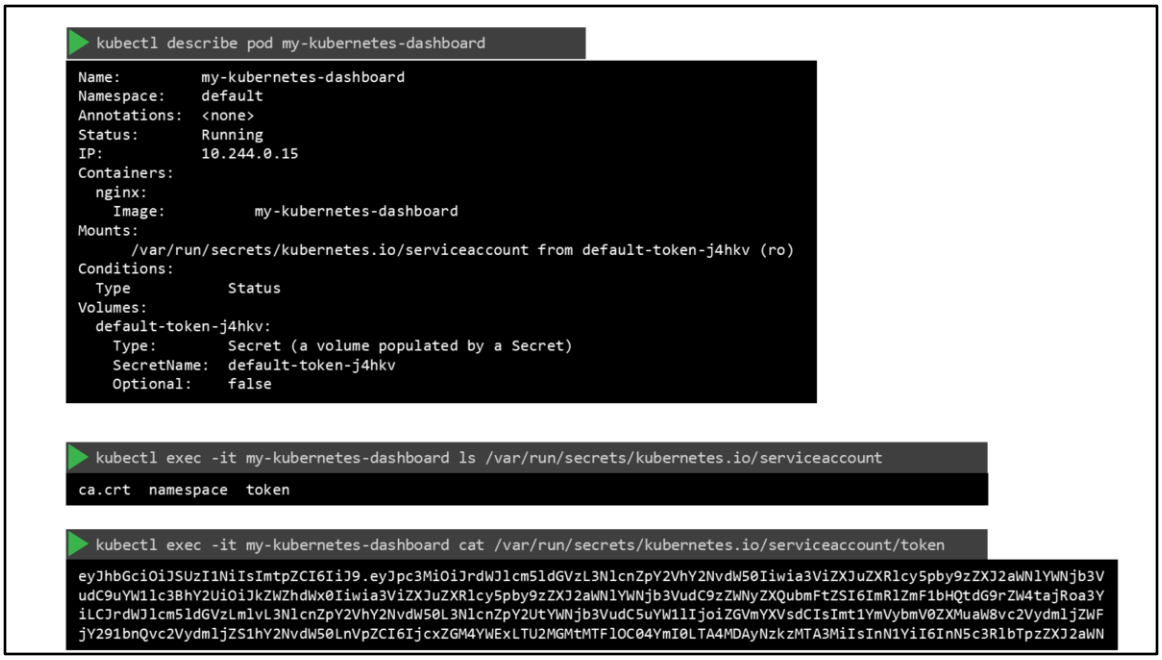


In that case, this whole process of exporting the service account token and configuring the third party application to use it can be made simple by <click> automatically mounting the service token secret as a volume inside the POD hosting the third party application. That way the token to access the kubernetes API is already placed inside the POD and can be easily read by the application.

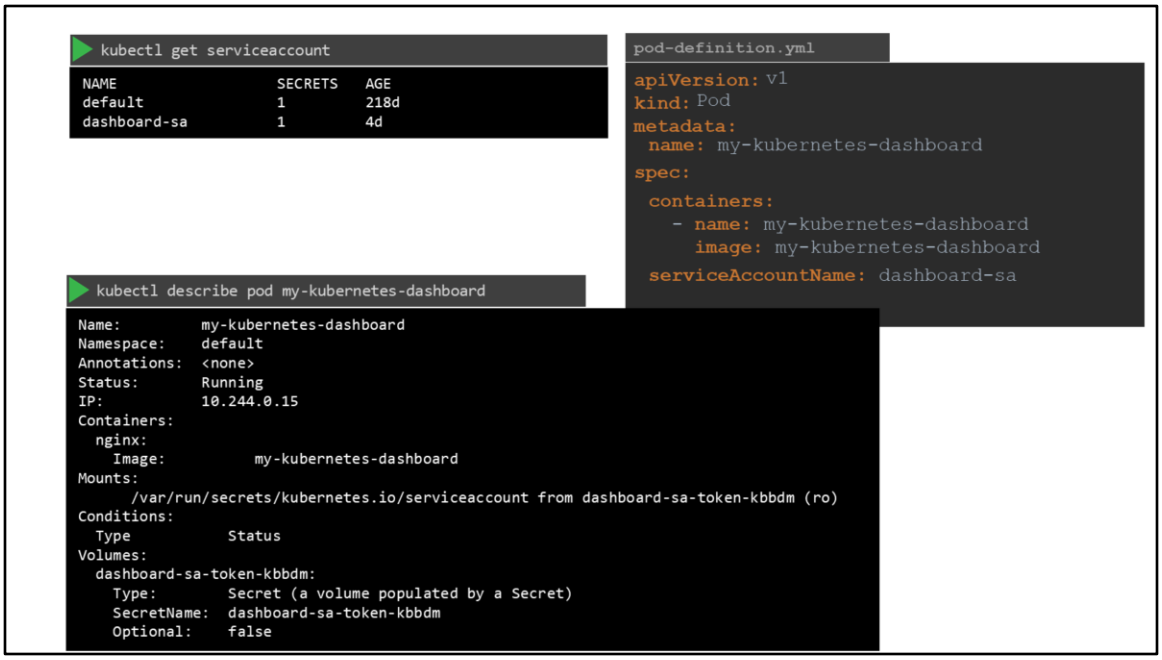


<click> If you go back and look at the list of service accounts, you will see that there is a default service account that exists already. For every namespace in kubernetes a service account named default is automatically created. Each namespace has its own default service account.

Whenever a POD is created the default service account and its token are automatically mounted to that POD as a volume mount. For example, <click> 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. However when the pod is created, if you look at the details of the pod, by running the <click> kubectl describe pod command, you see that a volume is automatically created from the secret named default-token-j4hkv, which is in fact the secret containing the token for the 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



If you list the <click> contents of the directory inside the pod, you will see the secret mounted as 3 separate files. The one with the actual token is the file named token. <click> If you list the 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 serviceAccount, such as the ones we just created, <click> modify the pod definition file to include a serviceAccount field and specify the name of the new service account. Remember, you cannot edit the service account of an existing pod, so you must delete and re-create the pod. However in case of a deployment, you will be able to edit the serviceAccount, as any changes to the pod definition will automatically trigger a new roll-out for the deployment. So the deployment will take care of deleting and re-creating new pods with the right service account. <click> When you look at the pod details now, you see that the new service account is being used.

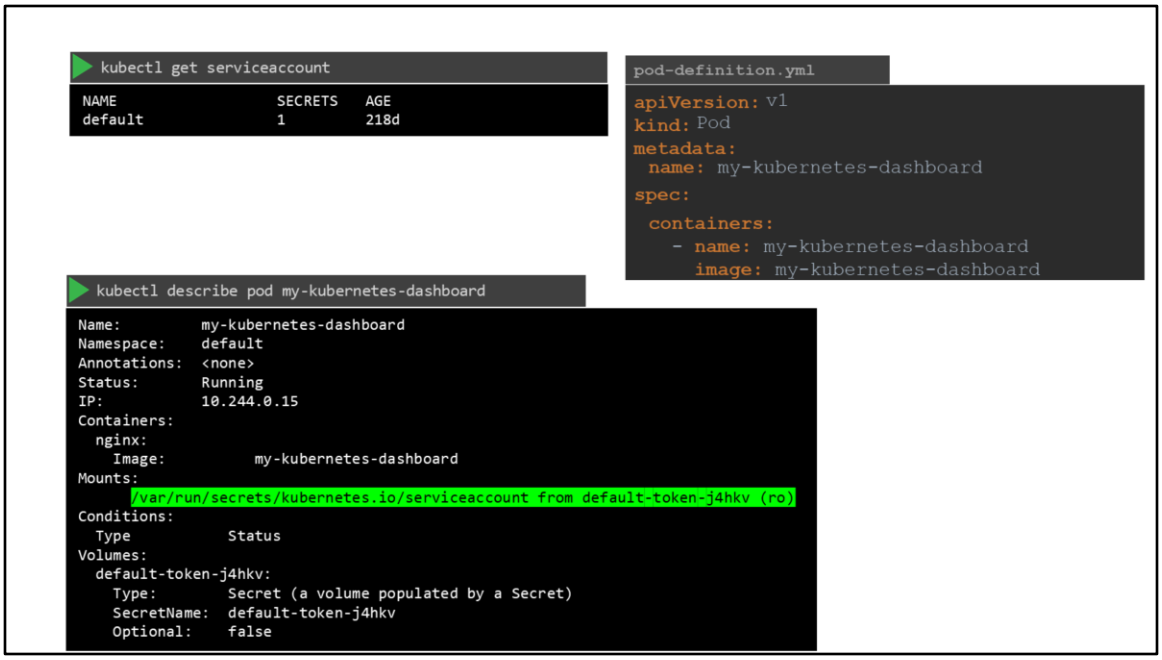
pod-definition.yml

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

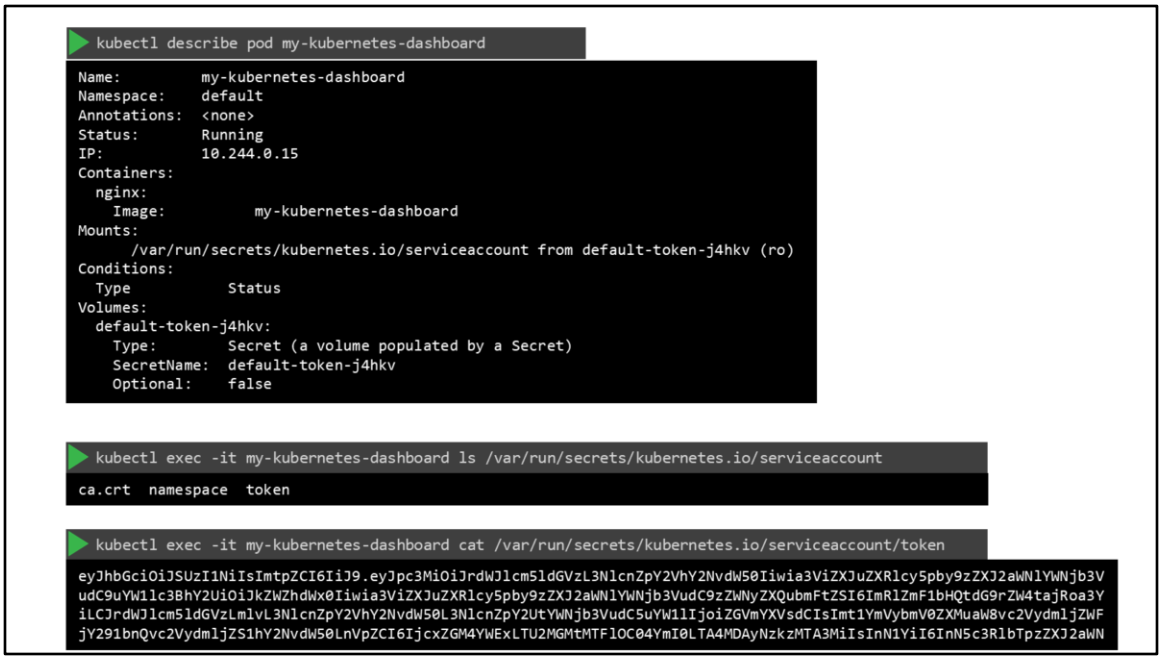
So remember, kubernetes automatically mounts the default service account if you haven't explicitly specified any. You may choose not to mount a service account automatically by setting the automountServiceAccountToken field to false in the POD spec section.

Well that's it for this lecture. Head over to the practice exercises section and practice working with service accounts. We will configure the custom kubernetes dashboard with the right service account.

So there were some changes made in release v1.22 and v1.24 of Kubernetes that changed the way service accounts/secrets and tokens worked.



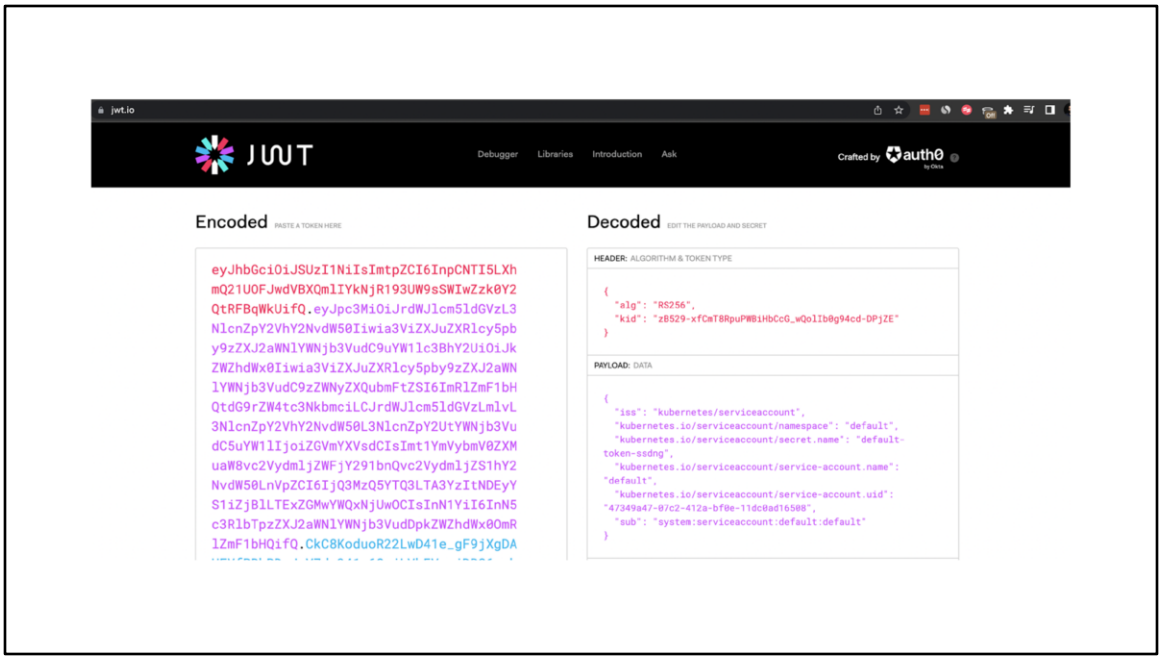
As we discussed in the previous video every namespace has a default service account. And that service account has a secret object with a token associated with it. When a pod is created it automatically associates the service account to the pod and mounts the token to a well known location within the pod. This makes the token accessible to a process within the pod to query the kubernetes API.



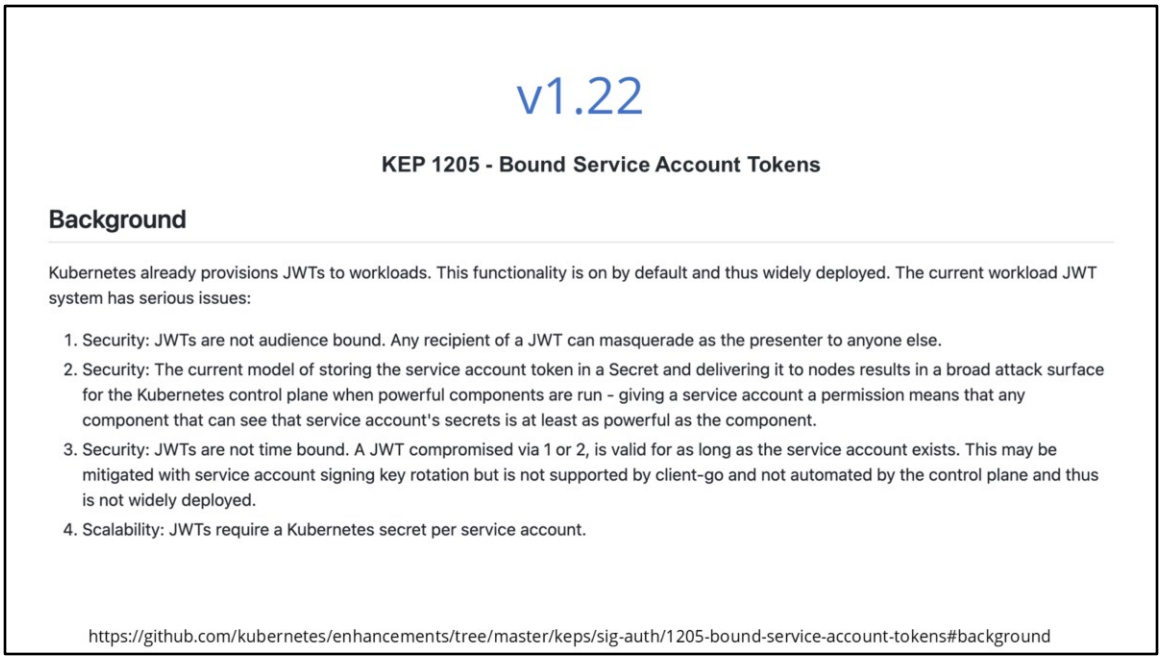
If you list the <click> contents of the directory inside the pod, you will see the secret mounted as 3 separate files. The one with the actual token is the file named token. <click> If you list the contents of that file you will see the token to be used for accessing the kubernetes API.

Decode a JWT via command line - dbubenheim
https://gist.github.com/angelo-v/e0208a18d455e2e6ea3c40ad637aac53

Now, if you decode this token using this command or by copying pasting this token in the JWT website....

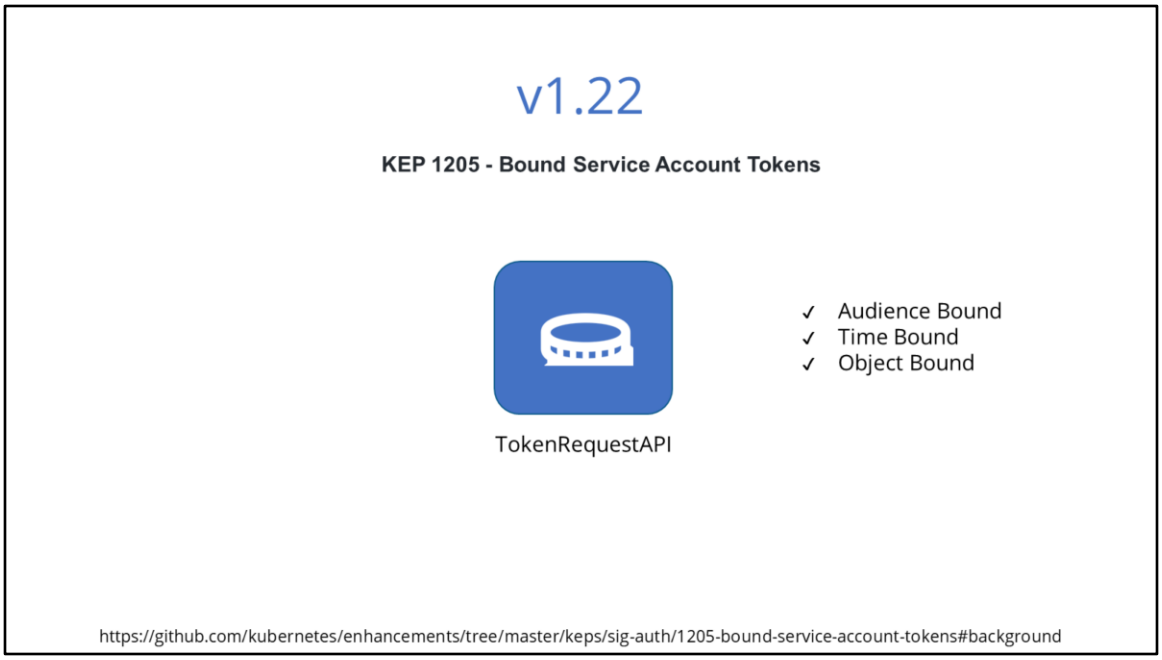


… at jwt.io you'll see that it has no expiry date defined in the payload here on the right.



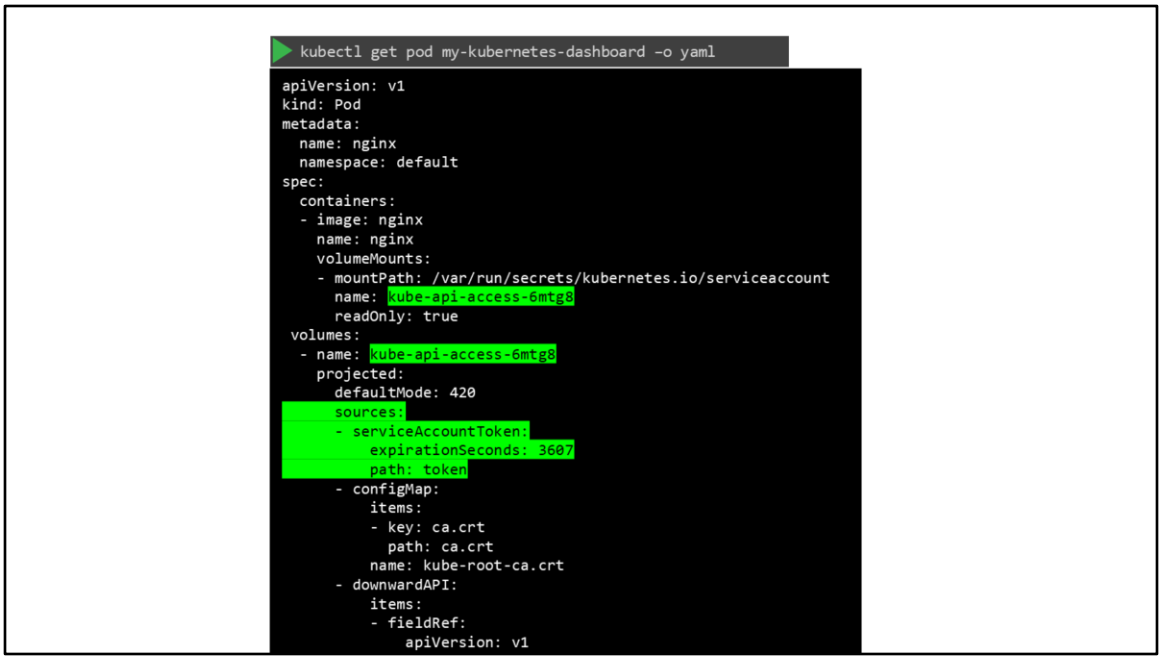
This excerpt from the Kubernetes Enhancements Proposal for creating Bound service accounts tokens describes this form of JWT to be having some security and scalability related issues.

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. 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.

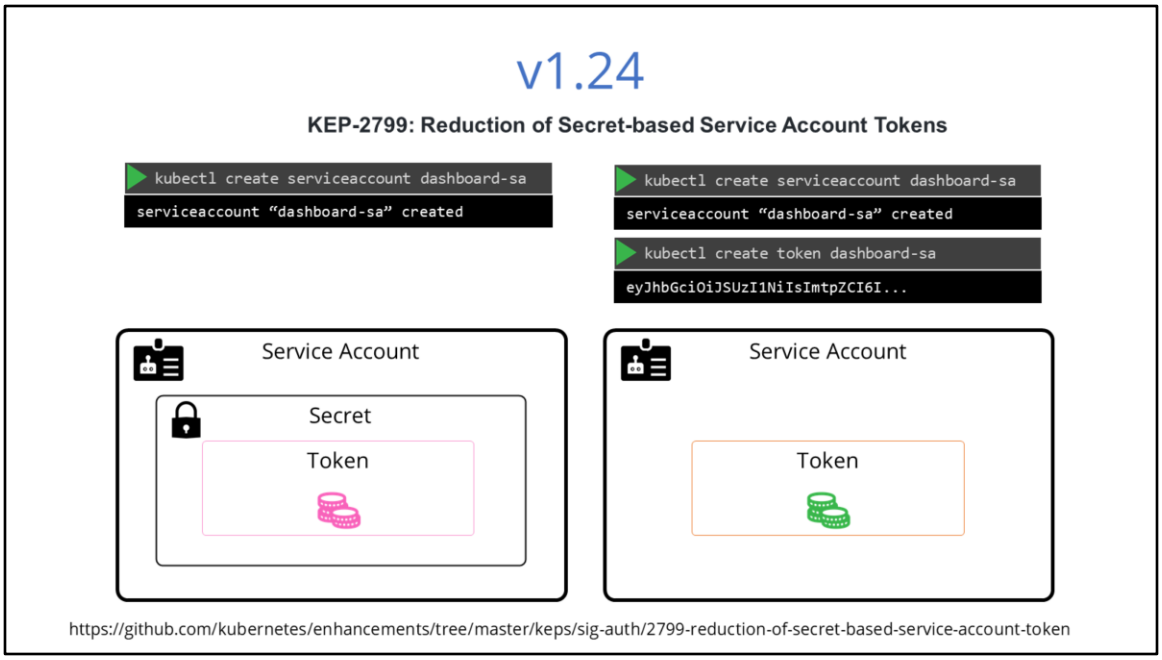


and as such in v1.22 the TokenRequest API was introduced as part of the Kubernetes Enhancement Proposal 1205 that aimed to introduce a mechanism for provisioning kubernetes service account tokens that are more secure and scalable via an API.

Tokens generated by the TokenRequest API are audience bound, time bound and object bound and hence are more secure.

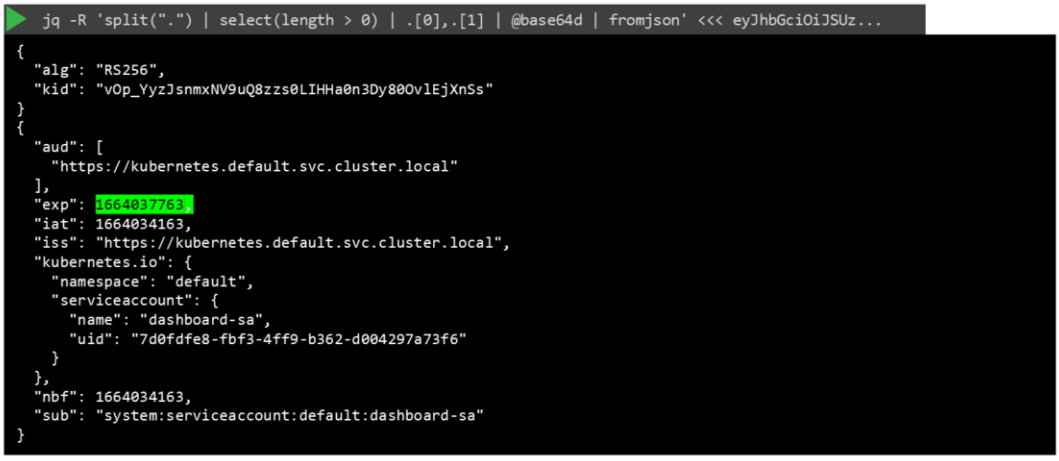


Since v1.22 when a new pod is created it no longer relies on the service account secret token, instead a token with a defined lifetime is generated through the TokenRequest API by the Service Account Admission Controller. This token is then mount as a projected volume into the pod.

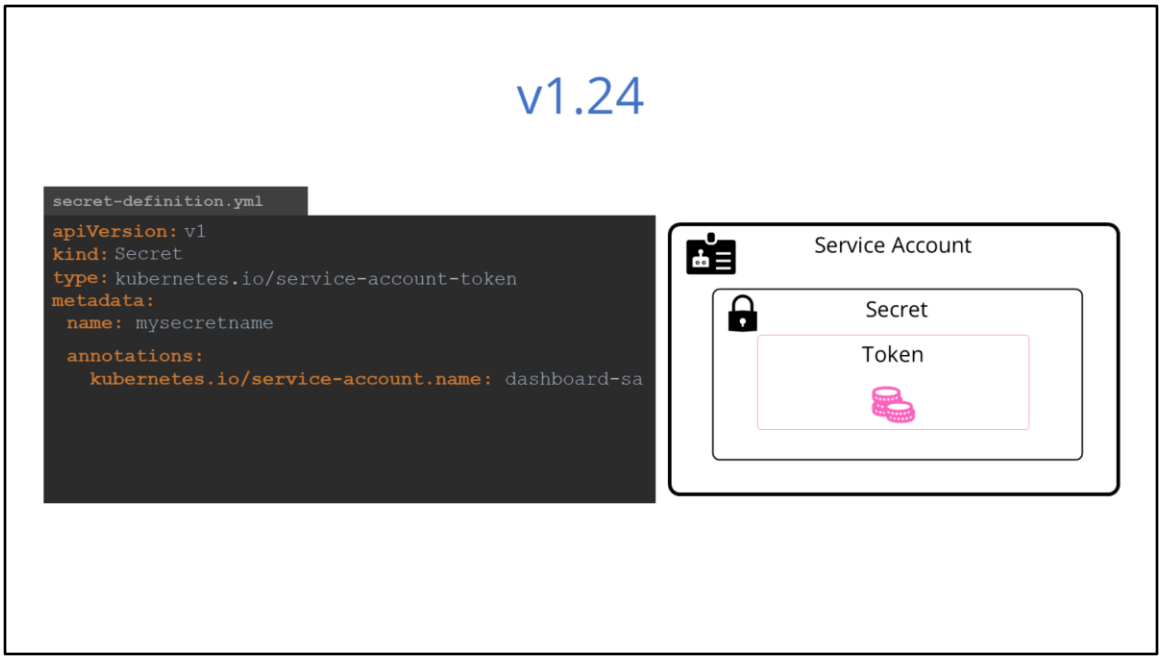


With version 1.24 another enhancement was made as part of the KEP 2799 – Reduction of secret based service account tokens. In the past when a service account was created, it automatically created a secret with a token that has no expiry and is not bound to any audience. This was then automatically mount as a volume to any pod that uses that service account.

With v1.24 when a service account is created it does not automatically create a secret or a token. You must run the command kubectl create token followed by the name of the service account to generate a token for the service account. It then prints the token on screen.

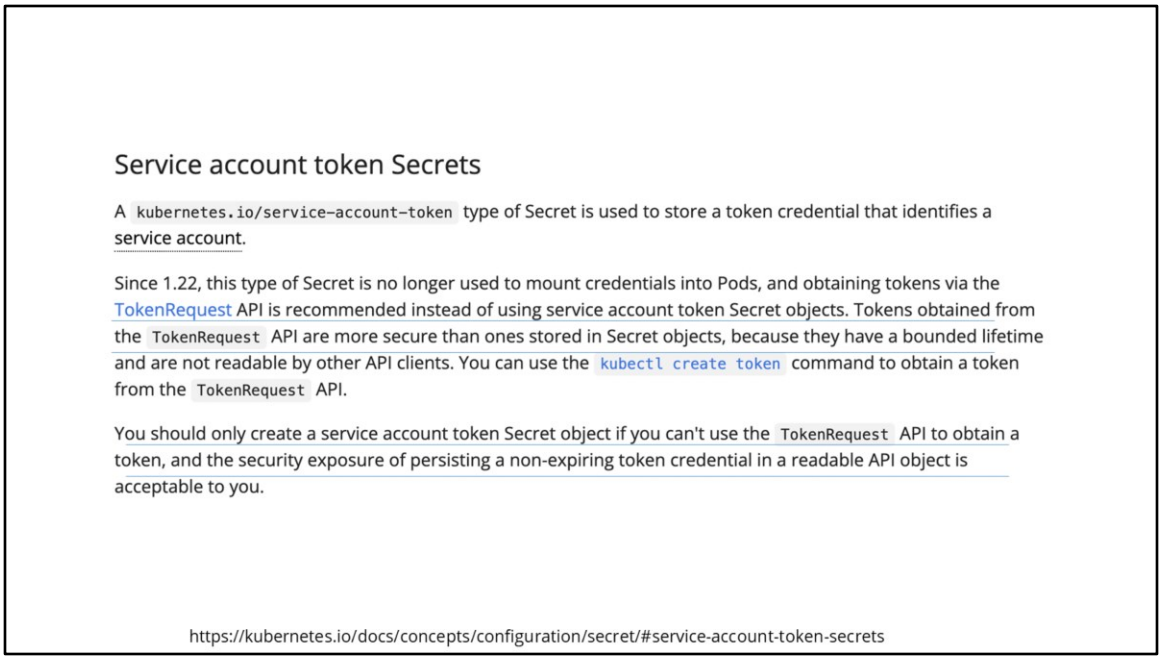


If you decode this token, this time you'll see an expiry defined. If you haven't specified any time limit, then it's usually 1 hour from the time you ran the command. You can pass in additional options to the command to increase the expiry of the token.



Now post v1.24 if you would still like to create secrets the old way with non-expiring token, then you could create a secret object with the type set to kubernetes.io/service-account-token and the name of the service account specified within the annotations in metadata.

This will create a non-expiring token in a secret object and associate it with the service account. But for this make sure the service account already exists prior to creating the secret.



As per the kubernetes documentation pages on Service account token secrets, you should only create service account token secret if you can't use the TokeRequest API to obtain a token. That's either the kubectl create token command we just talked about. It talks to the token request API to generate that token. Or it's the automated token creation that happens on pods when they are created post v1.22. And also you should only create service account token secret if the security exposure of persisting a non-expiring token credential is acceptable to you.

The TokenRequest API is recommended instead of using 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.

References

KEP 1205 - Bound Service Account Tokens

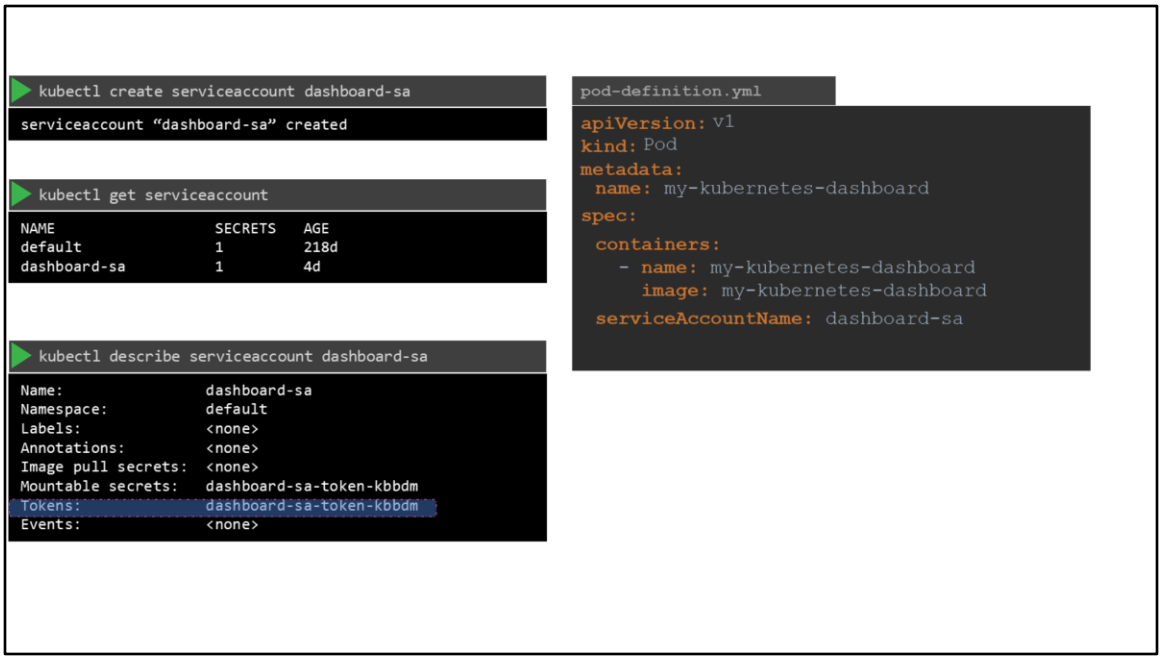
https://github.com/kubernetes/enhancements/tree/master/keps/sig-auth/1205-bound-service-account-tokens

KEP-2799: Reduction of Secret-based Service Account Tokens

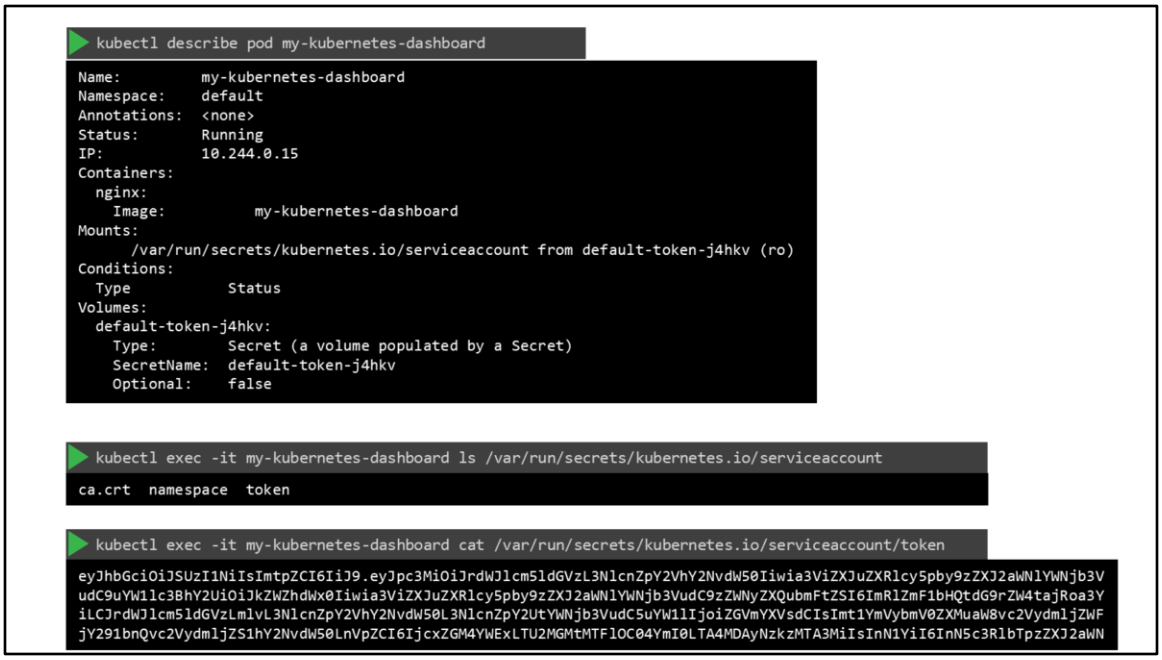
https://github.com/kubernetes/enhancements/tree/master/keps/sig-auth/2799-reduction-of-secret-based-service-account-token

https://kubernetes.io/docs/concepts/configuration/secret/#service-account-token-secrets
https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/

To read more about these changes refer to the Kubernetes Enhancement proposals and the documentation pages listed here.



As I mentioned in the previous video when a service account is created by running the command kubectl create service account command <click> it creates a service account by that name and also automatically creates a secret object by the same name with a token in it. This service account is then automatically mount to a pod, and thus the token is available within the pod automatically.



If you list the <click> contents of the directory inside the pod, you will see the secret mounted as 3 separate files. The one with the actual token is the file named token. <click> If you list the 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.

Note:
Versions of Kubernetes before v1.22 automatically created credentials for accessing the Kubernetes API. This older mechanism was based on creating token Secrets that could then be mounted into running Pods. In more recent versions, including Kubernetes v1.25, API credentials are obtained directly by using the TokenRequest API, and are mounted into Pods using a projected volume. The tokens obtained using this method have bounded lifetimes, and are automatically invalidated when the Pod they are mounted into is deleted.

You can still manually create a service account token Secret; for example, if you need a token that never expires. However, using the TokenRequest subresource to obtain a token to access the API is recommended instead. You can use the `kubectl create token` command to obtain a token from the `TokenRequest` API.

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.

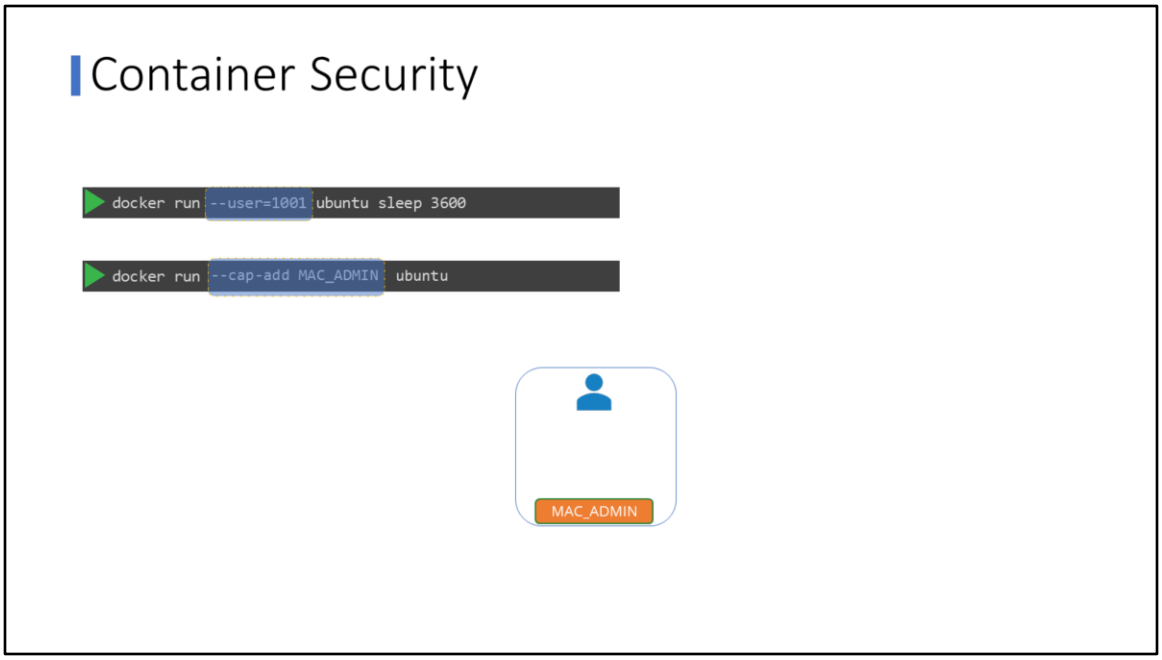
Since 1.22, this type of Secret is no longer used to mount credentials into Pods, and obtaining tokens via the TokenRequest API is recommended instead of using service account token Secret objects. Tokens obtained from the `TokenRequest` API are more secure than ones stored in Secret objects, because 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.

When using this Secret type, you need to ensure that the `kubernetes.io/service-account.name` annotation is set to an existing service account name. If you are creating both the ServiceAccount and the Secret objects, you should create the ServiceAccount object first.

After the Secret is created, a Kubernetes controller fills in some other fields such as the `kubernetes.io/service-account.uid` annotation, and the `token` key in the `data` field, which is populated with an authentication token.


115
Kubernetes
Security
Contexts

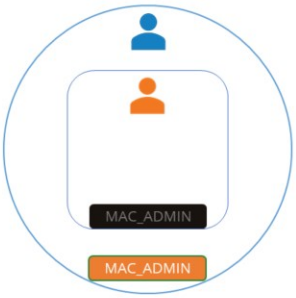


As we saw in the previous lecture, <c> when you run a Docker Container you have the option to define a set of security standards, such as the ID of the user used to run the container, <c> the Linux capabilities that can be added or removed from the container etc. These can be configured in Kubernetes as well.

Kubernetes Security

As you know already, in Kubernetes containers are encapsulated in PODs. You may chose to configure the security settings at a container level….

Kubernetes Security



… or at a POD level. <c> If you configure it at a POD level, the settings will carry over to all the containers within the POD. <c> If you configure it at both the POD and the Container, the settings on the container will override the settings on the POD.

Security Context

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

Let us start with a POD definition file. This pod runs an ubuntu image with the sleep command. <c> 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.

Security Context

```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"]
```

Note: Capabilities are only supported at the container level and not at the POD level

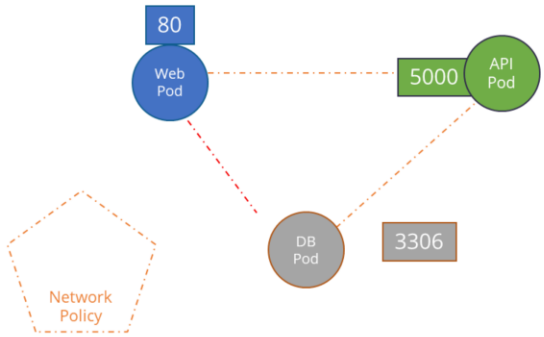
To set the same configuration on the container level, move the whole section under the container specification like this.

<c> To add capabilities use the capabilities option and specify a list of capabilities to add to the POD.

Well that's all on Security Contexts. Head over to the coding exercises section and practice viewing, configuring and troubleshooting issues related to Security contexts in Kubernetes. That's it for now and I will see you in the next lecture.

Let’s look at writing network policies in more detail.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
  podSelector:
    matchLabels:
      role: db
```



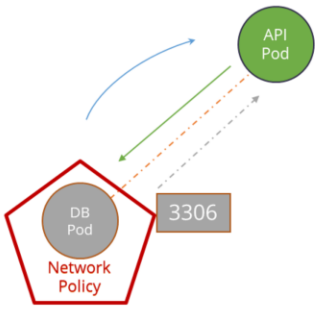
Here we have the same web, api and database pods, that we discussed about in the previous lecture. So first lets be very clear with our requirement. Our goal is to protect the database pod so that it does not allow access from any other pod except the API pod and only on port 3306. Let's assume that we are not concerned about the web pod or the api pod. For those pods we are Ok for all traffic to go in and out from anywhere. However we want to protect the database pod and only allow traffic from the API pod. So let's get the other things out, so we can focus exactly on the required tasks. We don't need to worry about the web pod or its port, as we don't want to allow any traffic from any other sources other than the API pod. So let's get rid of it. <c>

We can also forget about the port on the API to which the web server connects. As we don't care about that either.

As we discussed by default Kubernetes allows all traffic from all pods to all destinations. So as the first step we want to block out everything going in and out of the database pod. So we create a network policy, we will call it db-policy. And the first step is to associate this network policy with the pod that we want to protect. And we do that using labels and selectors. So we do that by adding a podSelector

field with the matchLabels option and by specifying the label on the db pod. Which happens to be set to role db.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
  podSelector:
    matchLabels:
      role: db

  policyTypes:
  - Ingress
  - Egress
```



And that associates the network policy with the db pod. It still doesn't block out traffic because we haven't specified policyTypes yet. If there are no policyTypes specified the protection is not enforced. Sicne we'd like to restrict both ingress and egress traffic we will add ingress and egress to the policy types. And that should block all ingress and egress traffic on the pod.

First we need to figure out what type of policies should be defined on this network policy object to meet our requirements. There are 2 types of policies that we discussed in the previous lecture – ingress and egress. Do we need Ingress or Egress here? Or both? So you always look at this from the db-pods perspective. From the db pods perspective, we want to allow incoming traffic from the API pod. So that is ingress. We will talk about egress connections in a bit, for now it's still allowed.

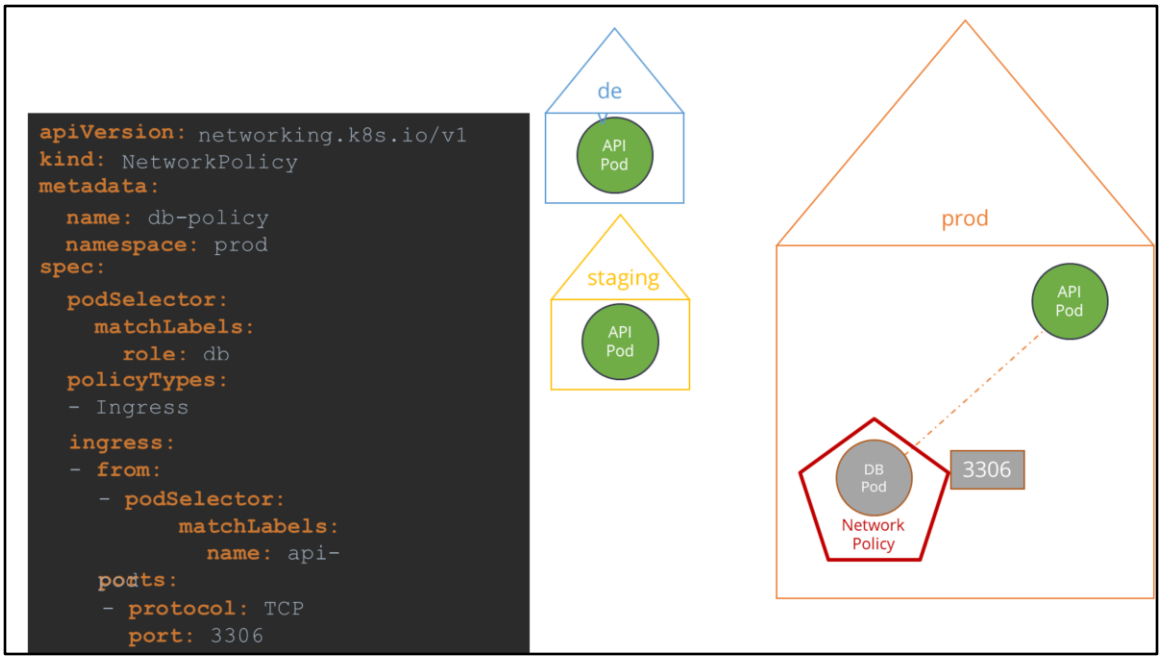
The API pod makes database queries and the DB pods returns the results. What about the results? Do we need a separate rule for the results to go back to the API pod? No. Because once you allow incoming traffic the response or reply to that traffic is allowed back automatically. We don't need a separate rule for that. In this case all we need is an ingress rule to allow traffic from the API pod to the database pod. And that would allow the API pod to connect to the database and run queries and also retrieve

the result of the queries. When deciding on what type of rule is to be created you only need to be concerned about the direction in which the request originates, denoted by the straight line here. You don't need to worry about the response denoted by the dotted line.

However, this rule does not mean that the database pod will be able to make calls to the API pod. Say for example the database pod tries to make an API call to the API pod, then that would not be allowed, because that is now an egress traffic originating from the database pod and would require a specific egress rule to be defined. So I hope you get the difference between the 2 and are clear about ingress and egress traffic.

I just wanted to make sure you are clear on what type of policy is to be selected for the requirement you have. <c>

So a single network policy can have an ingress type of rule, an egress type of rule or both in cases where a pod wants to allow incoming connections as well as wants to make external calls. For now our use case only requires ingress policy types. Now that we have decided on the type of policy we have to create the next step is to define the specifics of that policy.



If its ingress we create a section called ingress. Within which we can specify multiple rules. Each rule has a from and ports fields. The from field defines the source of traffic that is allowed to pass through to the database pod. Here we would use a pod selector and provide the labels of the api pod like this.

The ports field defines what port on the database pod is the traffic allowed to. In this its 3306 with the TCP protocol.

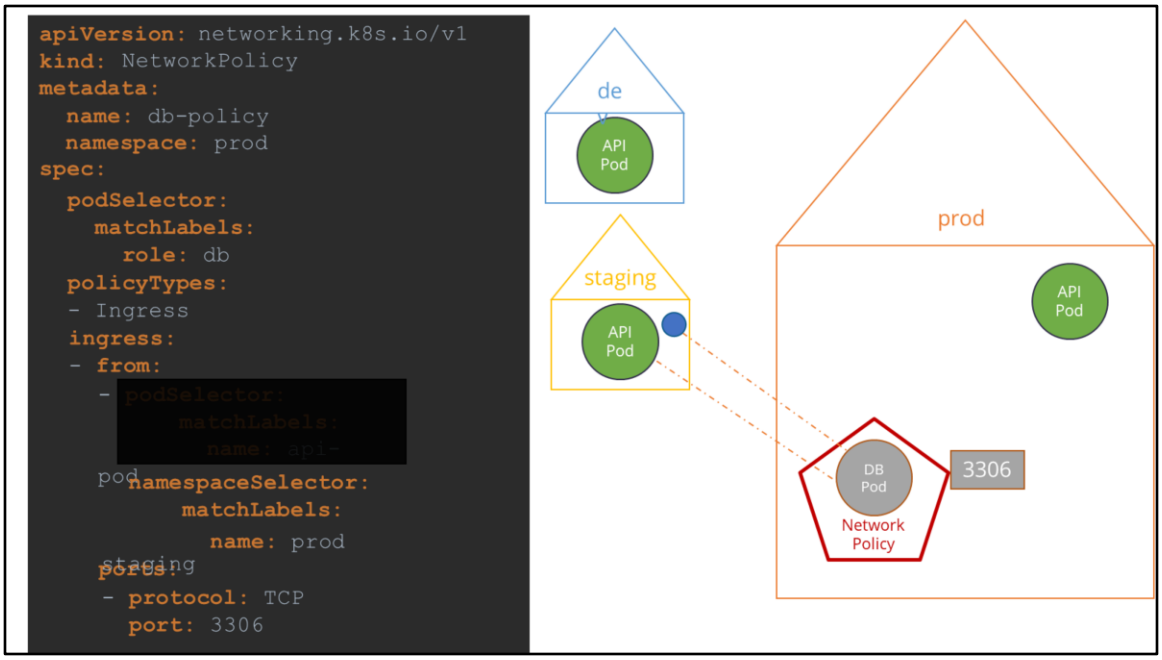
And that's it. This would create a policy that would block all traffic to the db pod except for traffic from the API pod.

Now what if there are multiple API pods in the cluster with the same labels, but in different namespaces? Here we have different namespaces for dev, staging and prod environments and we have the API pod with the same labels in each of these environments.

First if you are creating a network policy for the prod namespace, then it should have namespace defined as prod.

By default this policy will only allow the pod in the prod namespace – which is the same namespace as that of the Network Policy with matching labels to reach the database pod. But what if for some reason, we want the pod in the staging namespace to reach the database in the production namespace?

For this

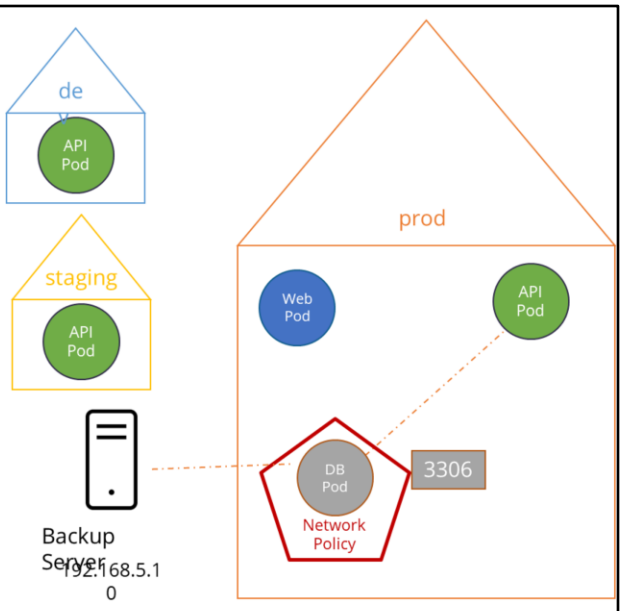


For this we add a namespaceSelector property as well along with the podSelector property. Under this we use matchLabels again to provide a label set on the namespace. Remember you must have this label set on the namespace first for this to work. So that's what the namespace selector does. It helps in defining from what namespace traffic is allowed to reach the database pod.

What if you only have the namespace selector and not the pod selector like this? In that case, all pods within the specified namespace will be allowed to reach the database pod. Such as the web pod. But pods from outside this namespace won't be allowed to go through.

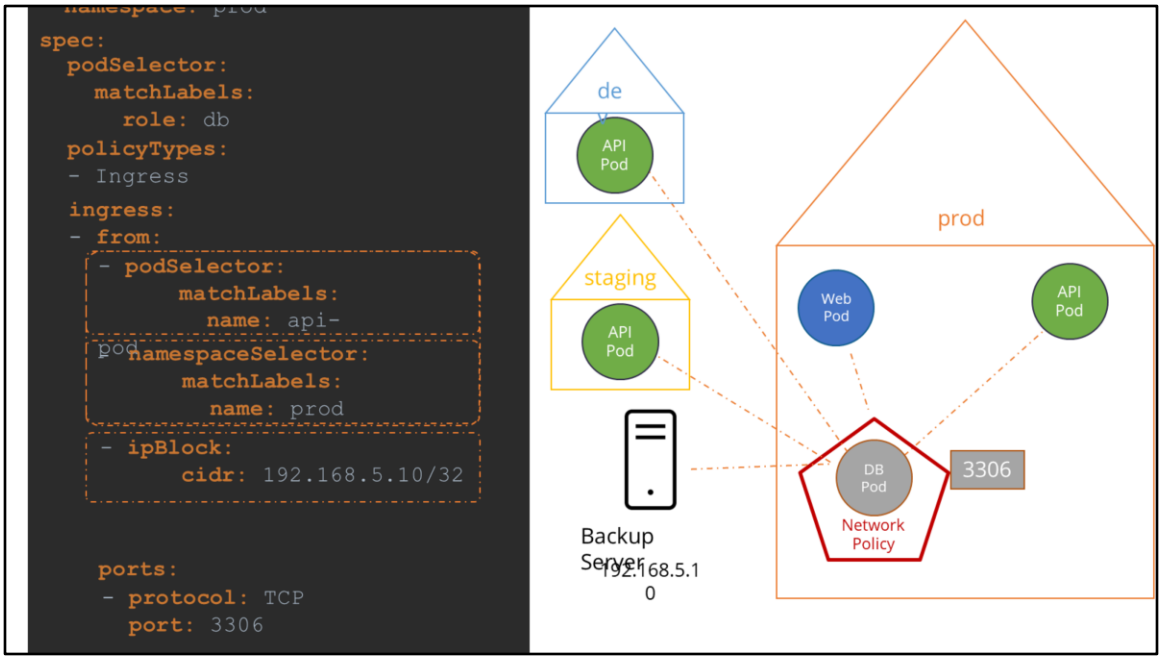
So let's change that namespace selector back to prod for now and continue our example.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
  namespace: prod
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          name: api-
    pod
      namespaceSelector:
        matchLabels:
          name: prod
    ports:
    - protocol: TCP
      port: 3306
```



Let's look at another use case. Say we have a backup server somewhere. Outside of Kubernetes cluster. And we want to allow this server to connect to the database pod. Since this backup server is not deployed in our Kubernetes cluster, the podSelector and namespaceSelector fields we used to define traffic from won't work. However, we know the IP address of the backup server. It happens to be 192.168.5.10.

We configure a network policy to allow traffic originating from certain IP addresses.



For this we add a new type of from definition known as the ipBlock. ipBlock allows you to specify a range of IP addresses from which you could allow traffic to hit the database pod.

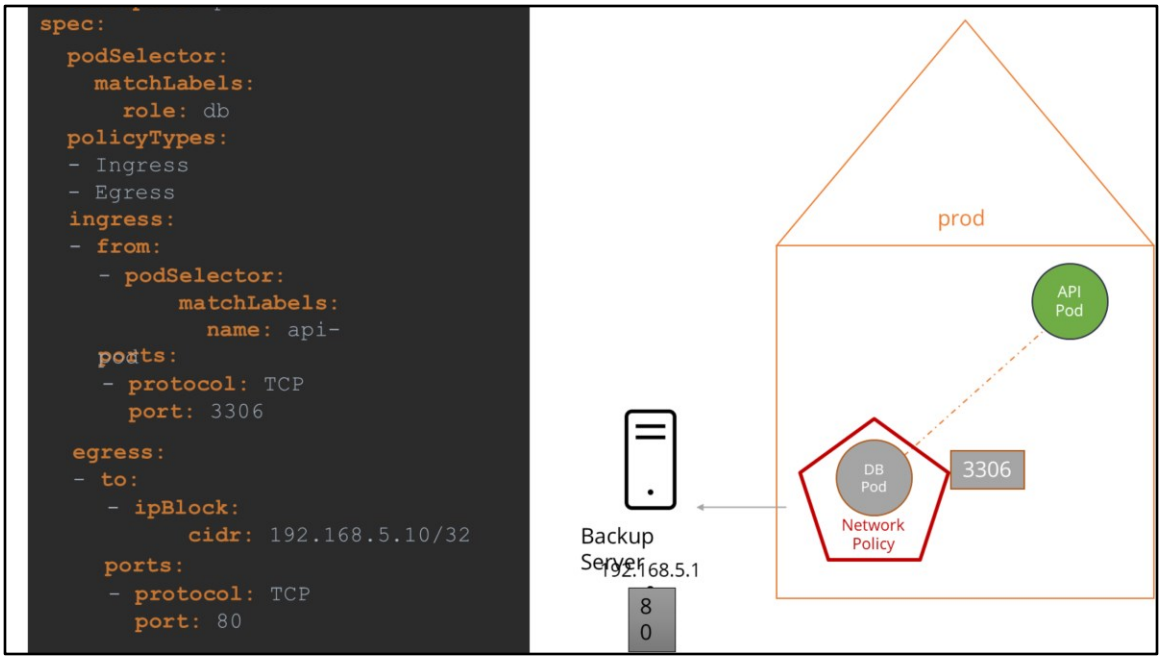
So those are the 3 supported selectors under the from section in ingress and to section in egress. We have pod Selector to select pods by labels, namespace selector to select namespaces by labels and the ipBlock selector to select IP address ranges.

These can be passed in separately as individual rules or together as part of a single rule. In this example under the from section we have 2 elements. These are 2 rules or 2 policies. The first rule has the podSelect and namespace selector together and the second rule has the IP Block selector. This works like an OR operation. Traffic from sources meeting either of these criteria's are allowed to pass through.

However within the first rule we have to selectors part of it. That would mean traffic from sources must meet both of these criterias to pass through. They have to be originating from pods with matching labels of api-pod AND those pods must be in the prod namespace. So it works like an AND operation.

But what if we separate them by adding a dash before the namespace selector? Now they are separate rules. This would mean that traffic matching the first rule is allowed – that is from any pod matching the label api-pod in any namespace. And traffic matching second rule is allowed – that is from any pod within the prod namespace is also allowed. And of course along with the backup server as we have the ipBlock specification as well. So a small change like that can have a big impact.

So its important to understand how you could put together these rules based on your requirements.



Now let's get rid of all of that and go back to a basic set of rules. And we will now look at egress. Say for example instead of the backup server initiating a backup, say we have an agent on the db pod that pushes backup to the backup server. In that case, the traffic is originating from the database pod to an external backup server. For this we need to have egress rule defined.

So we first add Egress to the policy types. And then we add a new egress section to define the specifics of the policy. Now instead of from we have to under egress. Under to we have the to block. Under to, we could use any of the selectors such as a pod, namespace of ipblock selector. In this case since the database server is external we use ip block selector and provide the cidr block for the server.

The port to which the requests are to be sent is 80. So we specify 80 as the port.

This rule allows traffic originating from the db pod to an external backup server at the specified address.

Well that's it for now about network policies and rules. Head over to the lab and practice working with network policies yourself.