**Getting Started with Security**

Learn the security and related concerns.

**We'll cover the following**

* [Understanding the scenario](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JE3oq2AVgKD#Understanding-the-scenario)
* [Exploring the options](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JE3oq2AVgKD#Exploring-the-options)

**Understanding the scenario**

**Security implementation** is a game between a team with a total lock-down strategy and a team that plans to win by providing complete freedom to everyone. You can think of it as a battle between anarchists and totalitarians. The only way the game can be won is if both blend into something new. The only viable strategy is freedom without sacrificing security (too much).

Right now, our cluster is as secured as it can get. There is only one user (you). No one else can operate it. The others cannot even list the Pods in the cluster. You are the judge, the jury, and the executioner. You are the undisputed king with god-like powers that are not shared with anyone else.

The I-and-only-I-can-do-things strategy works well when simulating a cluster on a laptop. It serves the purpose when the only goal is to learn alone.

**Exploring the options**

The moment we create a “real” cluster where the whole company will collaborate (in some form or another), we’ll need to define (and apply) an authentication and authorization strategy.

If your business is small and there are only a few people who will ever operate the cluster, giving everyone the same cluster-wide administrative set of permissions is a simple and legitimate solution. More often than not, this will not be the case.

Your company probably has people with different levels of trust. Even if that’s not the case, different people will require different levels of access. Some will be allowed to do anything they want, while others will not have any type of access. Most will be able to do something in between. We might choose to give everyone a separate Namespace and forbid them from accessing others. Some might be able to operate a production Namespace while others might have interest only in the one assigned for development and testing.

The number of permutations we can apply is infinite. Still, one thing is certain. We will need to create an authentication and authorization mechanism. Most likely, we’ll need to create permissions that are sometimes applied cluster-wide and, in other cases, limited to Namespaces.

Those and many other policies can be created by employing Kubernetes authorization and authentication.

**Accessing Kubernetes API**

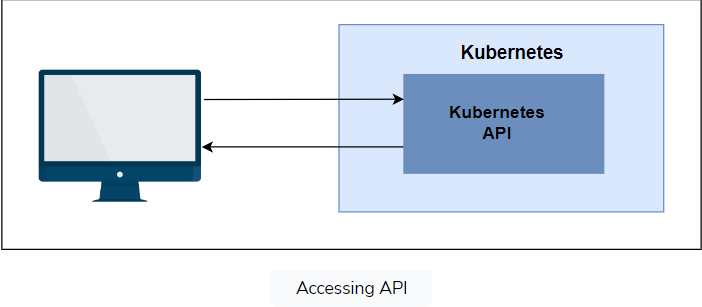
Explore the Kubernetes API and the process to access it.

**We'll cover the following**

* [The API](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N8jY4j7Gr9m#The-API)
* [Checking out the port](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N8jY4j7Gr9m#Checking-out-the-port)
* [The real-world scenario](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N8jY4j7Gr9m#The-real-world-scenario)
  + [Understanding the process](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N8jY4j7Gr9m#Understanding-the-process)
    - [Authentication](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N8jY4j7Gr9m#Authentication)
    - [Authorization](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N8jY4j7Gr9m#Authorization)
    - [Passing the Admission Control](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N8jY4j7Gr9m#Passing-the-Admission-Control)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N8jY4j7Gr9m#Try-it-yourself)
* [Troubleshooting tips for minikube](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N8jY4j7Gr9m#Troubleshooting-tips-for-minikube)

**The API**

Every interaction with Kubernetes goes through its API and needs to be authorized. That communication can be initiated through a user or a service account. All Kubernetes objects currently running inside our cluster are interacting with the API through service accounts. We won’t go deep into those. Instead, we’ll concentrate on the authorization of human users.



## Checking out the port

Typically, the Kubernetes API is served on a secured port. Our k3d cluster is no exception. We can check the port from the kubectl config.

Before proceeding further, let's create a cluster.

Now let's check the port on which our cluster is running using the config.

kubectl config view -o jsonpath='{.clusters[?(@.name=="k3d-mycluster")].cluster.server}'

We used jsonpath to output the cluster.server entry located in the cluster with the name k3d-mycluster.

The **output** is as follows.

<https://0.0.0.0:32867>

We can see that kubectl accesses the Kubernetes API on the port “32867” (this value can be different on your system). Since the access is secured, it requires certificates that are stored as the certificate-authority entry.

As we are working with k3d, the certificates are set up by k3d itself in the /var/lib/rancher/k3s/server/tls directory. While creating the cluster, we will create a volume to map the directory of k3d on our local directory, namely /usercode/certs. We can check the contents of this directory by listing the /usercode/certs directory:

ls usercode/certs

The **output** is as follows.

...  
client-ca.crt  
client-ca.key  
server-ca.crt  
server-ca.key  
...

## The real-world scenario

If this was a “real” cluster, we’d need to enable access for other users as well. We could send them the certificate we already have, but that would be very insecure and would lead to a lot of potential problems. Soon, we’ll explore how to enable other users to access the cluster securely. For now, we’ll focus on the exploration of the process Kubernetes uses to authorize requests to its API.

### Understanding the process

Each request to the API goes through **three stages**.

* Authentication
* Authorization
* Passing the admission control

#### Authentication

Kubernetes uses client certificates, bearer tokens, an authenticating proxy, or HTTP basic auth to authenticate API requests through authentication plugins. In the authentication process, the username is retrieved from the HTTP request. If the request cannot be authenticated, the operation is aborted with the status code 401.

#### Authorization

Once the user is authenticated, the authorization validates whether it is allowed to execute the specified action. The authorization can be performed through ABAC, RBAC, or Webhook modes.

#### Passing the Admission Control

Finally, once a request is authorized, it passes through admission controllers. They intercept requests to the API before the objects are persisted and can modify them. They are advanced topics that we won’t cover in this chapter.

Authentication is pretty standard, and there’s not much to say about it. On the other hand, admission controllers are too advanced to be covered just yet.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl config view \

-o jsonpath='{.clusters[?(@.name=="k3d-mycluster")].cluster.server}'

ls usercode/certs

## Troubleshooting tips for minikube

While working with minikube locally, you will need to use the following commands.

# Get the port for minikube cluster

kubectl config view \

-o jsonpath='{.clusters[?(@.name=="minikube")].cluster.server}'

# Get the certificates for minikube cluster

kubectl config view \

-o jsonpath='{.clusters[?(@.name=="minikube")].cluster.certificate-authority}'

For the first command, we can see that kubectl accesses the Minikube Kubernetes API on the port 8443. While the second command results in the certificate-authority entry.

/Users/vfarcic/.minikube/ca.crt

The ca.crt certificate was created with the Minikube cluster and, currently, provides the only way we can access the API.

# Authorizing Requests and Creating a Cluster

Learn about various authorization methods and choose one for our use.

**We'll cover the following**

* [Authorization methods](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xVZ3VOVBGEz#Authorization-methods)
* [We will go with RBAC](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/xVZ3VOVBGEz#We-will-go-with-RBAC)

## Authorization methods

Just like almost everything else in Kubernetes, authorization is modular. We can choose to use Node, ABAC, Webhook, or RBAC authorization.

**Node:** Node authorization grants permissions to kubelets based on the Pods they are scheduled to run.

**ABAC:** Attribute-based access control (ABAC) is based on attributes combined with policies and is considered deprecated in favor of RBAC.

**Webhooks:** Webhooks are used for event notifications through HTTP POST requests.

**RBAC:** Role-based access control (RBAC) grants (or denies) access to resources based on roles of individual users or groups.

## We will go with RBAC

Among the four authorization methods, RBAC is the right choice for user-based authorization. Since we’ll focus this chapter on the exploration of the means to authorize humans, RBAC will be our primary focus.

What can we do with RBAC?

* We can use it to secure the cluster by allowing access only to authorized users.
* We can define roles that would grant different levels of access to users and groups. Some could have god-like permissions that would allow them to do almost anything, while others could be limited only to basic non-destructive operations. There can be many other roles in between.
* We can combine RBAC with Namespaces and allow users to operate only within specific segments of a cluster.
* There are many other combinations we could apply depending on particular use-cases.

We’ll leave the rest for later and explore details through a few examples. As you might already suspect, we’ll kick it off with a new k3d cluster.

To check if RBAC is enabled on k3d run kubectl api-versions if it is enabled you should see .rbac.authorization.k8s.io/v1.

It might come in handy to have a few objects in the cluster so we’ll deploy the go-demo-2 application. We’ll use it to test different permutations of the authorization strategies we’ll use soon.

The definition of the go-demo-2 application is the same as the one we created in the previous chapters so we’ll skip the explanation and just execute kubectl create.

kubectl create \

-f go-demo-2.yml \

--record --save-config

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: go-demo-2

annotations:

kubernetes.io/ingress.class: "nginx"

ingress.kubernetes.io/ssl-redirect: "false"

nginx.ingress.kubernetes.io/ssl-redirect: "false"

spec:

rules:

- host: go-demo-2.com

http:

paths:

- path: /demo

pathType: ImplementationSpecific

backend:

service:

name: go-demo-2-api

port:

number: 8080

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: go-demo-2-db

spec:

selector:

matchLabels:

type: db

service: go-demo-2

strategy:

type: Recreate

template:

metadata:

labels:

type: db

service: go-demo-2

vendor: MongoLabs

spec:

containers:

- name: db

image: mongo:3.3

---

apiVersion: v1

kind: Service

metadata:

name: go-demo-2-db

spec:

ports:

- port: 27017

selector:

type: db

service: go-demo-2

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: go-demo-2-api

spec:

replicas: 3

selector:

matchLabels:

type: api

service: go-demo-2

template:

metadata:

labels:

type: api

service: go-demo-2

language: go

spec:

containers:

- name: api

image: vfarcic/go-demo-2

env:

# - name: DB

# value: go-demo-2-db

- name: DB\_ENV

value: GO\_DEMO\_2\_DB\_SERVICE\_HOST

readinessProbe:

httpGet:

path: /demo/hello

port: 8080

periodSeconds: 1

livenessProbe:

httpGet:

path: /demo/hello

port: 8080

---

apiVersion: v1

kind: Service

metadata:

name: go-demo-2-api

spec:

ports:

- port: 8080

selector:

type: api

service: go-demo-2

# Creating Users to Access the Cluster

Learn how to create a user based on a fictitious character named 'John Doe' who wants to access our cluster.

**We'll cover the following**

* [Understanding the scenario](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gkVX0pWvRpZ#Understanding-the-scenario)
* [Installing openSSL](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gkVX0pWvRpZ#Installing-openSSL)
* [Creating a user](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gkVX0pWvRpZ#Creating-a-user)
  + [Creating final certificate](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gkVX0pWvRpZ#Creating-final-certificate)
  + [Getting the server address](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gkVX0pWvRpZ#Getting-the-server-address)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gkVX0pWvRpZ#Try-it-yourself)
* [Troubleshooting Tips for minikube](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/gkVX0pWvRpZ#Troubleshooting-Tips-for-minikube)

## Understanding the scenario

The word about Kubernetes’ awesomeness is spreading in your company. People are becoming curious and would like to try it out.

Since you are the Kubernetes guru, it came as no surprise that you received a call from John Doe. He wants to “play” with Kubernetes, but he does not have time to set up his own cluster. Since he knows that you already have a cluster up and running, he’d appreciate it if you would let him use yours.

Since you have no intention of giving John your certificates, you decide to let him authenticate with his user.

## Installing openSSL

You will have to create certificates for him, so the first step you’ll need to do is to verify that OpenSSL is installed.

openssl version

It shouldn’t matter which version of OpenSSL is installed. We output the version only to verify that the software is working.

If the output is something like command not found: openssl, you will have to [install the binaries](https://wiki.openssl.org/index.php/Binaries).

## Creating a user

The first thing we’ll do is to create a private key for John. We’ll assume that John Doe’s username is jdoe.

mkdir keys

openssl genrsa -out keys/jdoe.key 2048

We created the directory keys and generated a private key jdoe.key.

Next, we’ll use the private key to generate a certificate.

openssl req -new -key keys/jdoe.key -out keys/jdoe.csr -subj "/CN=jdoe/O=devs"

### Creating final certificate

We created the certificate jdoe.csr with a specific subject that will help us identify John. CN is the username and O represents the organization he belongs. John is a developer, so devs should do.

For the final certificate, we’ll need the cluster’s certificate authority (CA). It will be responsible for approving the request and for generating the necessary certificate John will use to access the cluster.

Since we used k3d, the authority is already produced for us as part of the cluster creation. It should be in the /usercode/certs directory as per the volume binding.

Let’s confirm it’s there.

ls -1 /usercode/certs/client-ca.\*

The **output** is as follows.

/usercode/certs/client-ca.crt

/usercode/certs/client-ca.key

Now we can generate the final certificate by approving the certificate sign request jdoe.csr.

openssl x509 -req -in keys/jdoe.csr -CA /usercode/certs/client-ca.crt -CAkey /usercode/certs/client-ca.key -CAcreateserial -out keys/jdoe.crt -days 365

Since we feel generous, we made the certificate jdoe.crt valid for a whole year (365 days).

To simplify the process, we’ll copy the cluster’s certificate authority to the keys directory.

cp /usercode/certs/server-ca.crt /usercode/certs/keys

Let’s check what we generated.

ls -1 keys

The **output** is as follows.

jdoe.crt

jdoe.csr

jdoe.key

server-ca.crt

John does not need the “jdoe.csr” file. We used it only to generate the final certificate “jdoe.crt”. He will need all the other files though.

### Getting the server address

Apart from the keys, John will need to know the address of the cluster. At the beginning of the chapter, we already created the jsonpath that retrieves the server so that part should be easy.

SERVER=$(kubectl config view -o jsonpath='{.clusters[?(@.name=="k3d-mycluster")].cluster.server}')

echo $SERVER

The **output** is as follows. Note that the IP may differ.

<https://0.0.0.0:37141>

Equipped with:

* The new certificate (jdoe.crt)
* The key (jdoe.key)
* The cluster authority (ca.crt)
* The address of the server,

John can configure his kubectl installation.

## Try it yourself

A list of all the commands used in the lesson is given below.

mkdir keys

openssl genrsa -out keys/jdoe.key 2048

openssl req -new -key keys/jdoe.key -out keys/jdoe.csr -subj "/CN=jdoe/O=devs"

ls -1 /usercode/certs/client-ca.\*

openssl x509 -req \

-in keys/jdoe.csr \

-CA /usercode/certs/client-ca.crt \

-CAkey /usercode/certs/client-ca.key \

-CAcreateserial \

-out keys/jdoe.crt \

-days 365

cp /usercode/certs/server-ca.crt /usercode/certs/keys

ls -1 keys

SERVER=$(kubectl config view -o jsonpath='{.clusters[?(@.name=="k3d-mycluster")].cluster.server}')

echo $SERVER

## Troubleshooting Tips for minikube

While working with minikube locally, we can access the certificate-authority in the following directory:

ls -1 ~/.minikube/ca.\*

Minikube’s directory might be somewhere else, so replace the path accordingly while accessing the certificates from keys directory.

# Accessing the Cluster as a User

Learn how to impersonate the new user John and try to get the access authenticated.

**We'll cover the following**

* [Authentication](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVArY5KDxL2#Authentication)
  + [Configuring kubectl](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVArY5KDxL2#Configuring-kubectl)
  + [Setting the credentials](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVArY5KDxL2#Setting-the-credentials)
  + [Creating a new context](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVArY5KDxL2#Creating-a-new-context)
* [No authorization](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVArY5KDxL2#No-authorization)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qVArY5KDxL2#Try-it-yourself)

## Authentication

Since John is not around, we’ll do some role-playing and impersonate him.

### Configuring kubectl

John will first have to set the cluster using the address and the certificate authority we sent him.

kubectl config set-cluster jdoe \

--certificate-authority \

/usercode/certs/keys/server-ca.crt \

--server $SERVER

We created a new cluster called jdoe.

Please note that the command we executed created just another config entry that enables to access all resources that currently exist in the cluster. It manipulated local text file, without changing anything in the existing cluster.

### Setting the credentials

Next, he’ll have to set the credentials using the certificate and the key we created for him.

kubectl config set-credentials jdoe \

--client-certificate keys/jdoe.crt \

--client-key keys/jdoe.key

We created a new set of credentials called jdoe.

### Creating a new context

Finally, John will have to create a new context.

kubectl config set-context jdoe \

--cluster jdoe \

--user jdoe

kubectl config use-context jdoe

We created the context jdoe that uses the newly created cluster and the user. We also made sure that we’re using the newly created context.

Let’s take a look at the config.

kubectl config view

The **output**, limited to John’s settings, is as follows.

...

clusters:

- cluster:

certificate-authority: /usercode/certs/keys/server-ca.crt

server: https://0.0.0.0:44573

name: jdoe

...

contexts:

- context:

cluster: jdoe

user: jdoe

name: jdoe

...

current-context: jdoe

...

users:

- name: jdoe

user:

client-certificate: /usercode/certs/keys/jdoe.crt

client-key: /usercode/certs/keys/jdoe.key

...

## No authorization

John should be happy thinking that he can access our cluster. Since he’s a curious person, he’ll want to see the Pods we’re running.

kubectl get pods

The **output** is as follows.

Error from server (Forbidden): pods is forbidden: User "jdoe" cannot list resource "pods" in API group "" in the namespace "default"

That’s frustrating. John can reach our cluster, but he cannot retrieve the list of Pods. Since hope dies last, John might check whether he is forbidden from seeing other types of objects.

kubectl get all

The **output** is a long list of all the objects he’s forbidden from seeing. So, in other words, John is authenticated successfully, but he is not authorized to view objects in the cluster.

John picks up his phone to beg not only that you give him access to the cluster but also the permissions to “play” with it.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl config set-cluster jdoe \

--certificate-authority \

/usercode/certs/keys/server-ca.crt \

--server $SERVER

kubectl config set-credentials jdoe \

--client-certificate keys/jdoe.crt \

--client-key keys/jdoe.key

kubectl config set-context jdoe \

--cluster jdoe \

--user jdoe

kubectl config use-context jdoe

kubectl config view

kubectl get pods

kubectl get all

# Exploring RBAC Authorization

Explore different components of the RBAC authorization.

**We'll cover the following**

* [The RBAC Components](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JEkPZlkB9vK#The-RBAC-Components)
  + [Rules](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JEkPZlkB9vK#Rules)
  + [Roles](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JEkPZlkB9vK#Roles)
  + [Subjects](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JEkPZlkB9vK#Subjects)
  + [RoleBindings](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JEkPZlkB9vK#RoleBindings)

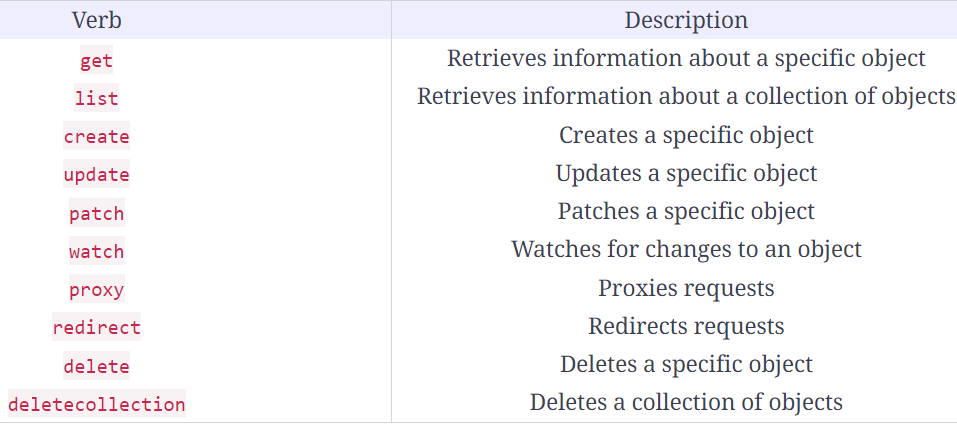
## The RBAC Components

Managing Kubernetes RBAC requires knowledge of a few elements. Specifically, we should learn about Rules, Roles, Subjects, and RoleBindings.

### Rules

A Rule is a set of operations (verbs), resources, and API groups. Verbs describe activities that can be performed on resources which belong to different API Groups.

Currently supported verbs are as follows.



Permissions defined through Rules are additive. We cannot deny access to some resources.

If, for example, we’d like to allow a user only to create objects and retrieve their information, we’d use the verbs get, list and create. A verb can be an asterisk (\*), thus allowing all verbs (operations).

Verbs are combined with Kubernetes resources. For example, if we’d like to allow a user only to create Pods and retrieve their information, we’d mix get, list and create verbs with the pods resource.

The last element of a Rule is the API Group. RBAC uses the rbac.authorization.k8s.io group. If we’d switch to a different authorization method, we’d need to change the group as well.

### Roles

A Role is a collection of Rules. It defines one or more Rules that can be bound to a user or a group of users.

The vital aspect of Roles is that they are applied to a Namespace. If we’d like to create a role that refers to a whole cluster, we’d use ClusterRole instead. Both are defined in the same way, and the only difference is in the scope (Namespace or an entire cluster).

### Subjects

The next piece of the authorization mechanism is Subjects.

Subjects define entities that are executing operations. A Subject can be a User, a Group, or a Service Account.

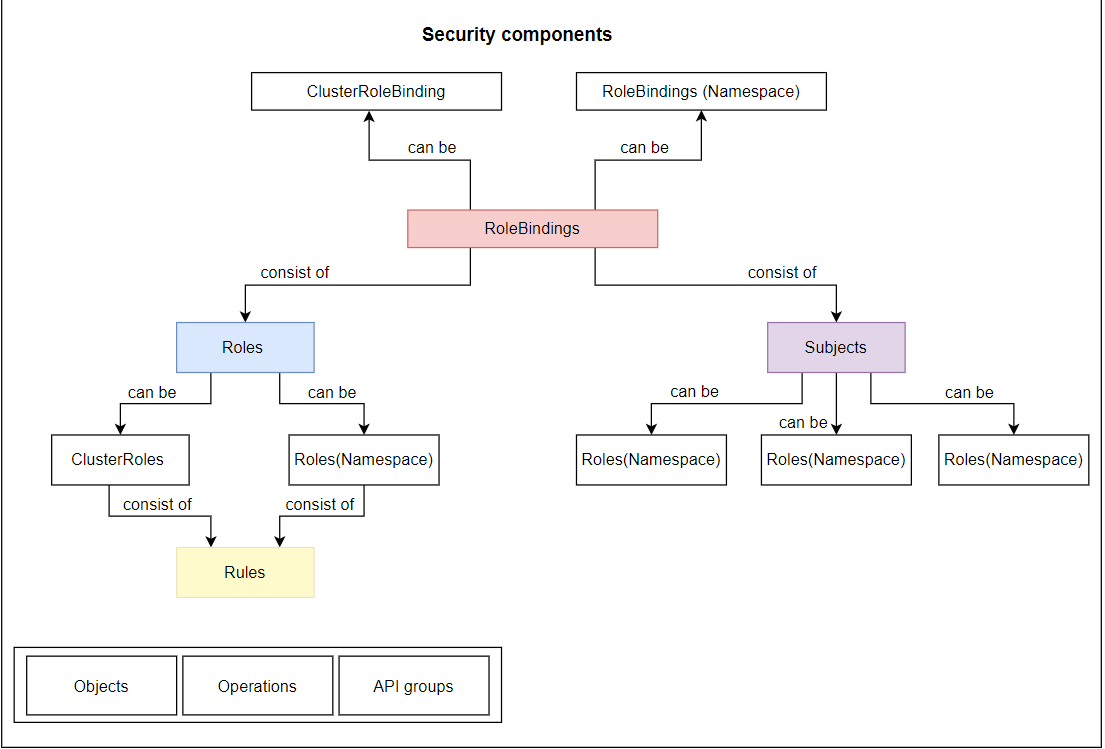
A User is a person or a process residing outside a cluster. A Service Account is used for processes running inside Pods that want to use the API. Since this chapter focuses on human authentication, we won’t explore them right now. Finally, Groups are collections of Users or Service Accounts. Some Groups are created by default (e.g., cluster-admin).

### RoleBindings

Finally, we need RoleBindings.

As the name suggests, RoleBindings bind Subjects to Roles.

Since Subjects define users, RoleBindings effectively bind users (or Groups or Service Accounts) to Roles, thus giving them permissions to perform certain operations on specific objects within a Namespace. Just like roles, RoleBindings have a cluster-wide alternative called ClusterRoleBindings. The only difference is that their scope is not limited to a Namespace, but applied to a whole cluster.



All that might seem confusing and overwhelming. You might even say that you did not understand anything. Don’t worry, we’ll explore each of the RBAC components in more detail through practical examples.

# Peeking into Pre-Defined Cluster Roles

Explore all the pre-defined cluster roles.

**We'll cover the following**

* [Switching from John to us](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qAYyA93A6D3#Switching-from-John-to-us)
* [Looking into roles and cluster roles](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qAYyA93A6D3#Looking-into-roles-and-cluster-roles)
  + [Looking into view](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qAYyA93A6D3#Looking-into-view)
  + [Looking into edit](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qAYyA93A6D3#Looking-into-edit)
  + [Looking into admin](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qAYyA93A6D3#Looking-into-admin)
  + [Looking into cluster-admin](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qAYyA93A6D3#Looking-into-cluster-admin)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qAYyA93A6D3#Try-it-yourself)

## Switching from John to us

John is frustrated. He can access the cluster, but he is not permitted to perform any operation. He cannot even list the Pods. Naturally, he asked us to be more generous and allow him to “play” with our cluster.

Since we are not taking anything for granted, we decided that the first action should be to verify John’s claim. Is it true that he cannot even retrieve the Pods running inside the cluster?

Before we move further, we’ll stop impersonating John and go back to using the cluster with god-like administrative privileges granted to the admin user.

kubectl config use-context k3d-mycluster

kubectl get all

Now that we switched to the k3d-mycluster context, we regained full permissions, and kubectl get all returned all the objects from the default Namespace.

Let’s verify that John indeed cannot list Pods in the default Namespace.

We could configure the same certificates as those he’s using, but that would complicate the process. Instead, we’ll use a kubectl command that will allow us to check whether we could perform an action if we would be a specific user.

kubectl auth can-i get pods --as jdoe

The response is no, indicating that jdoe cannot get pods. The --as argument is a global option that can be applied to any command. The kubectl auth can-i is a “special” command. It does not perform any action but only validates whether an operation could be performed. Without the --as argument, it would verify whether the current user (in this case k3d-mycluster) could do something.

## Looking into roles and cluster roles

We already discussed Roles and ClusterRoles briefly. Let’s see whether any are already configured in the cluster or the default namespace.

kubectl get roles

The output reveals that no resources were found. We do not have any Roles in the default Namespace. That was the expected outcome since a Kubernetes cluster comes with no pre-defined Roles. We’d need to create those we need ourselves.

How about Cluster Roles? Let’s check them out.

kubectl get clusterroles

This time we got quite a few resources. Our cluster already has some Cluster Roles defined by default. Those prefixed with system: are Cluster Roles reserved for Kubernetes system use. Modifications to those roles can result in non-functional clusters, so we should not update them. Instead, we’ll skip system Roles and focus on those that should be assigned to users.

The output, limited to Cluster Roles that are meant to be bound to users, is as follows (you can get the same result through kubectl get clusterroles | grep -v system).

kubectl get clusterroles | grep -v system

NAME AGE

admin 1h

cluster-admin 1h

edit 1h

view 1h

### Looking into view

The Cluster Role with the least permissions is view. Let’s take a closer look at it.

kubectl describe clusterrole view

The **output**, limited to the first few rows, is as follows.

Name: view

Labels: kubernetes.io/bootstrapping=rbac-defaults

Annotations: rbac.authorization.kubernetes.io/autoupdate=true

PolicyRule:

Resources Non-Resource URLs Resource Names Verbs

--------- ----------------- -------------- -----

bindings [] [] [get list watch]

configmaps [] [] [get list watch]

cronjobs.batch [] [] [get list watch]

daemonsets.extensions [] [] [get list watch]

deployments.apps [] [] [get list watch]

...

It contains a long list of resources, all of them with the “get”, “list”, and “watch” verbs. It looks like it would allow users bound to it to retrieve all the resources. We have yet to validate whether the list of resources is truly complete.

For now, it looks like an excellent candidate to assign to users that should have very limited permissions. Unlike Roles that are tied to a specific Namespace, Cluster Roles are available across the whole cluster. That is a significant difference that we’ll exploit later on.

### Looking into edit

Let’s explore another pre-defined Cluster Role.

kubectl describe clusterrole edit

The **output**, limited to Pods, is as follows.

...

pods [] [] [create delete deletecollection get list patch update watch]

pods/attach [] [] [create delete deletecollection get list patch update watch]

pods/exec [] [] [create delete deletecollection get list patch update watch]

pods/log [] [] [get list watch]

pods/portforward [] [] [create delete deletecollection get list patch update watch]

pods/proxy [] [] [create delete deletecollection get list patch update watch]

pods/status [] [] [get list watch]

...

As we can see, the “edit” Cluster Role allows us to perform any action on Pods. If we go through the whole list, we’d see that the “edit” role allows us to execute almost any operation on any Kubernetes object.

It seems like it gives us unlimited permissions. However, there are a few resources that are not listed. We can observe those differences through the Cluster Role admin.

### Looking into admin

kubectl describe clusterrole admin

If you pay close attention, you’ll notice that the Cluster Role admin has a few additional entries.

The output, limited to the records not present in the Cluster Role edit, is as follows.

...

localsubjectaccessreviews.authorization.k8s.io [] [] [create]

rolebindings.rbac.authorization.k8s.io [] [] [create delete deletecollection get list patch update watch]

roles.rbac.authorization.k8s.io [] [] [create delete deletecollection get list patch update watch]

...

The main difference between edit and admin is that the latter allows us to manipulate Roles and RoleBindings. While edit permits us to do almost any operation related to Kubernetes objects like Pods and Deployments, admin goes a bit further and provides an additional capability that allows us to define permissions for other users by modifying existing or creating new Roles and Role Bindings.

The major restriction of the admin role is that it cannot alter the Namespace itself, nor it can update Resource Quotas (we haven’t explored them yet).

### Looking into cluster-admin[**#**](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/qAYyA93A6D3#Looking-into-cluster-admin)

There is only one more pre-defined non-system Cluster Role left.

kubectl describe clusterrole cluster-admin

Name: cluster-admin

Labels: kubernetes.io/bootstrapping=rbac-defaults

Annotations: rbac.authorization.kubernetes.io/autoupdate=true

PolicyRule:

Resources Non-Resource URLs Resource Names Verbs

--------- ----------------- -------------- -----

[\*] [] [\*]

\*.\* [] [] [\*]

The Cluster Role “cluster-admin” holds nothing back. An asterisk ("\*") means everything. It provides god-like powers. A user bound to this role can do anything, without any restrictions. The cluster-admin role is the one bound to the k3d-cluster user. We can confirm that easily by executing

kubectl auth can-i "\*" "\*"

The output is “yes”. Even though we did not really confirm that the cluster-admin role is bound to mycluster, we did verify that it can do anything.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl config use-context k3d-mycluster

kubectl get all

kubectl auth can-i get pods --as jdoe

kubectl get roles

kubectl get clusterroles

kubectl get clusterroles | grep -v system

kubectl describe clusterrole view

kubectl describe clusterrole edit

kubectl describe clusterrole admin

kubectl describe clusterrole \

cluster-admin

kubectl auth can-i "\*" "\*"

# Creating Role Bindings

Create a Role Binding to let the user have viewing access to all the objects in our default Namespace.

**We'll cover the following**

* [Creating role bindings](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YVZPkrqGBQW#Creating-role-bindings)
  + [Looking into the role binding](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YVZPkrqGBQW#Looking-into-the-role-binding)
  + [Checking the scope](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YVZPkrqGBQW#Checking-the-scope)
  + [Deleting the role binding](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YVZPkrqGBQW#Deleting-the-role-binding)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/YVZPkrqGBQW#Try-it-yourself)

## Creating role bindings

Role Bindings bind a User (or a Group, or a Service Account) to a Role (or a Cluster Role). Since John wants more visibility to our cluster, we’ll create a Role Binding that will allow him to view (almost) all the objects in the default namespace. That should be a good start of our quest to give John just the right amount of privileges.

kubectl create rolebinding jdoe \

--clusterrole view \

--user jdoe \

--namespace default \

--save-config

kubectl get rolebindings

We created a Role Binding called jdoe. Since the Cluster Role view already provides, more or less, what we need, we used it instead of creating a whole new Role.

The **output** of the latter command proved that the new Role Binding “jdoe” was indeed created.

This is a good moment to clarify that a Role Binding does not need to be used only with a Role, but that it can also be combined with a Cluster Role (as in our example). As the rule of thumb, we define Cluster Roles when we think that they might be used cluster-wide (with Cluster Role Bindings) or in multiple Namespaces (with Role Bindings).

The scope of the permissions is defined with the type of binding, not with the type of role. Since we used Role Binding, the scope is limited to a single Namespace which, in our case, is the default.

### Looking into the role binding

Let’s take a look at the details of the newly created Role Binding.

kubectl describe rolebinding jdoe

Name: jdoe

Labels: <none>

Annotations: <none>

Role:

Kind: ClusterRole

Name: view

Subjects:

Kind Name Namespace

---- ---- ---------

User jdoe

We can see that the Role Binding “jdoe” has a single subject with the User “jdoe”. It might be a bit confusing that the Namespace is empty and you might think that the Role Binding applies to all Namespaces. Such an assumption would be false. Remember, a Role Binding is always tied to a specific Namespace, and we just described the one created in the default Namespace.

### Checking the scope

The same Role Binding should not be available anywhere else. Let’s confirm that.

kubectl --namespace kube-system \

describe rolebinding jdoe

We described the Role Binding jdoe in the Namespace kube-system.

The **output** is as follows.

Error from server (NotFound): rolebindings.rbac.authorization.k8s.io "jdoe" not found

The Namespace kube-system does not have that Role Binding. We never created it.

It might be easier to verify that our permissions are set correctly through the kubectl auth can-i command.

kubectl auth can-i get pods \

--as jdoe

kubectl auth can-i get pods \

--as jdoe --all-namespaces

The first command validated whether the user jdoe can get pods from the default Namespace. The answer was “yes”. The second checked whether the Pods could be retrieved from all the Namespaces and the answer was “no”. Currently, John can only see the Pods from the default Namespace, and he is forbidden from exploring those from the other Namespaces.

### Deleting the role binding

From now on, John should be able to view the Pods in the default Namespace. However, he works in the same company as we do and we should have more trust in him. Why don’t we give him permissions to view Pods in any Namespace? Why not apply the same permissions cluster-wide? Before we do that, we’ll delete the Role Binding we created and start over.

kubectl delete rolebinding jdoe

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl create rolebinding jdoe --clusterrole view --user jdoe --namespace default --save-config

kubectl get rolebindings

kubectl describe rolebinding jdoe

kubectl --namespace kube-system describe rolebinding jdoe

kubectl auth can-i get pods --as jdoe

kubectl auth can-i get pods --as jdoe --all-namespaces

kubectl delete rolebinding jdoe

# Creating Cluster Role Bindings

Learn how to grant cluster-wide access to the user with the help of Cluster Role Bindings.

**We'll cover the following**

* [View access across the cluster](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JEKVRDVl3ky#View-access-across-the-cluster)
  + [Looking into the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JEKVRDVl3ky#Looking-into-the-definition)
  + [Creation of ClusterRoleBinding](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JEKVRDVl3ky#Creation-of-ClusterRoleBinding)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JEKVRDVl3ky#Try-it-yourself)

## View access across the cluster

We’ll change John’s view permissions so that they are applied across the whole cluster.

Instead of executing yet another ad-hoc kubectl commands, we’ll define ClusterRoleBinding resource in YAML format so that the change is documented.

### Looking into the definition

Let’s take a look at the definition in the crb-view.yml file.

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: view

subjects:

- kind: User

name: jdoe

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: ClusterRole

name: view

apiGroup: rbac.authorization.k8s.io

Functionally, the difference is that, this time, we’re creating ClusterRoleBinding instead of RoleBinding. Also, we specified the apiGroup explicitly thus making it clear that the ClusterRole is RBAC.

### Creation of ClusterRoleBinding**[#](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/JEKVRDVl3ky" \l "Creation-of-ClusterRoleBinding)**

kubectl create -f crb-view.yml --record --save-config

We created the role defined in the YAML file, and the output confirmed that "clusterrolebinding view created.

We can further validate that everything looks correct by describing the newly created role.

kubectl describe clusterrolebinding view

The **output** is as follows.

Name: view

Labels: <none>

Annotations: kubernetes.io/change-cause: kubectl create --filename=crb-view.yml --record=true --save-config=true

Role:

Kind: ClusterRole

Name: view

Subjects:

Kind Name Namespace

---- ---- ---------

User jdoe

Finally, we’ll impersonate John and validate that he can indeed retrieve the Pods from any Namespace.

kubectl auth can-i get pods --as jdoe --all-namespaces

The **output** is “yes”, thus confirming that “jdoe” can view the Pods.

We’re so excited that we cannot wait to let John know that he was granted permissions. However, a minute into the phone call, he raises a concern. While being able to view Pods across the cluster is a good start, he will need a place where he and other developers will have more freedom.

They will need to be able to deploy, update, delete, and access their applications. They will probably need to do more, but they can’t give you more information. They are not yet very experienced with Kubernetes, so they don’t know what to expect.

John is asking you to find a solution that will allow them to perform actions that will help them develop and test their software without affecting other users of the cluster.

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl create -f crb-view.yml --record --save-config

kubectl describe clusterrolebinding view

kubectl auth can-i get pods --as jdoe --all-namespaces

**Combining Role Bindings with Namespaces**

Combine Role Bindings with Namespaces and create a user-specific Namespace.

**We'll cover the following**

* [The solution](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/NEP281D9Q9p#The-solution)
  + [Looking into the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/NEP281D9Q9p#Looking-into-the-definition)
  + [Creating the resources](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/NEP281D9Q9p#Creating-the-resources)
  + [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/NEP281D9Q9p#Verification)
* [User specific namespace](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/NEP281D9Q9p#User-specific-namespace)
  + [Looking into the definition](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/NEP281D9Q9p#Looking-into-the-definition)
  + [Creating the resources](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/NEP281D9Q9p#Creating-the-resources)
  + [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/NEP281D9Q9p#Verification)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/NEP281D9Q9p#Try-it-yourself)

**The solution**

The new request demanding more freedom provides an excellent opportunity to combine Namespaces with Role Bindings.

We can create a dev Namespace and allow a selected group of users to do almost anything in it. That should give developers enough freedom within the dev Namespace while avoiding the risks of negatively impacting the resources running in others.

**Looking into the definition**

Let’s take a look at the rb-dev.yml definition.

apiVersion: v1

kind: Namespace

metadata:

name: dev

---

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: dev

namespace: dev

subjects:

- kind: User

name: jdoe

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: ClusterRole

name: admin

apiGroup: rbac.authorization.k8s.io

**Line 1-4:** The first section defines the dev Namespace.

**Line 8-20:** The second section specifies the binding with the same name. Since we’re using RoleBinding (not ClusterRoleBinding), the effects will be limited to the dev Namespace. At the moment, there is only one subject (the User jdoe). We can expect the list to grow with time.

**Line 17-20:** roleRef uses ClusterRole (not Role) kind. Even though the Cluster Role is available across the whole cluster, the fact that we are combining it with RoleBinding will limit it to the specified Namespace.

The Cluster Role admin has an extensive set of resources and verbs, and the Users (at the moment only jdoe) will be able to do almost anything within the dev Namespace.

### Creating the resources

Let’s create the new resources.

kubectl create -f rb-dev.yml --record --save-config

namespace/dev created

rolebinding.rbac.authorization.k8s.io/dev created

We can see that the Namespace and the Role Binding were created.

### Verification

Let’s verify that, for example, jdoe can create and delete Deployments.

kubectl --namespace dev auth can-i create deployments --as jdoe

kubectl --namespace dev auth can-i delete deployments --as jdoe

In both cases, the output was yes, confirming that jdoe can perform at least create and delete actions with Deployments. Since we already explored the list of resources defined in the Cluster Role admin, we can assume that we’d get the same response if we’d check other operations.

Still, there are a few permissions that are not granted to John. Only the cluster-admin role covers all the permissions. The Cluster Role admin is very wide, but it does not include all the resources and verbs. We can confirm that with the command that follows.

kubectl --namespace dev auth can-i "\*" "\*" --as jdoe

The **output** is no, indicating that there are still a few operations forbidden to John within the dev Namespace. Those operations are mostly related to cluster administration that is still in our control.

John is happy. He and his fellow developers have a segment of the cluster where they can do almost anything without affecting other Namespaces.

John is a team player, but he’d also like to have space for himself. Now that he knows how easy it was to create a Namespace for developers, he’s wondering whether we could generate one only for him. We cannot deny the fact that his new request makes sense.

## User specific namespace

It should be easy to create his personal Namespace, so why not grant him that wish.

### Looking into the definition

Let’s take a look at yet another YAML definition rb-jdoe.yml.

apiVersion: v1

kind: Namespace

metadata:

name: jdoe

---

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: jdoe

namespace: jdoe

subjects:

- kind: User

name: jdoe

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: ClusterRole

name: cluster-admin

apiGroup: rbac.authorization.k8s.io

This definition is not much different from the previous one. The important change is that the Namespace is jdoe, and that John is likely to be its only user, at least until he decides to add someone else.

By referencing the role cluster-admin, he’s given full permissions to do whatever he wants within that Namespace. He might deploy something cool and give others permissions to see it. It’s his Namespace, and he should be able to do anything he likes inside it.

### Creating the resources

Let’s create the new resources.

kubectl create -f rb-jdoe.yml --record --save-config

### Verification

Before we move on, we’ll confirm that John can indeed do anything he likes in the jdoe Namespace.

kubectl --namespace jdoe auth can-i "\*" "\*" --as jdoe

As expected, the response is yes, indicating that John is a god-like figure in his own little galaxy.

## Try it yourself

For your ease, all the commands used in this lesson are given below:

kubectl create -f rb-dev.yml --record --save-config

kubectl --namespace dev auth can-i create deployments --as jdoe

kubectl --namespace dev auth can-i delete deployments --as jdoe

kubectl --namespace dev auth can-i "\*" "\*" --as jdoe

kubectl create -f rb-jdoe.yml --record --save-config

kubectl --namespace jdoe auth can-i "\*" "\*" --as jdoe

# Granting Access as a Release Manager

Grant the user access to the cluster as a release manager.

**We'll cover the following**

* [Defining the role](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B60xk5BWB2o#Defining-the-role)
* [Creating the role binding](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B60xk5BWB2o#Creating-the-role-binding)
  + [Exploring the cluster role admin](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B60xk5BWB2o#Exploring-the-cluster-role-admin)
  + [The challenge](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B60xk5BWB2o#The-challenge)
  + [Understanding sub-resources and API groups](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B60xk5BWB2o#Understanding-sub-resources-and-API-groups)
  + [Pods](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B60xk5BWB2o#Pods)
  + [Deployments](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B60xk5BWB2o#Deployments)
* [Creating a Release Manager Role Binding](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B60xk5BWB2o#Creating-a-Release-Manager-Role-Binding)
  + [Verification](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B60xk5BWB2o#Verification)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/B60xk5BWB2o#Try-it-yourself)

## Defining the role

John loves the idea of having his own Namespace. He’ll use it as his playground. However, there’s one more thing he’s missing.

He happens to be a release manager. Unlike his other fellow developers, he’s in charge of deploying new releases to production. He’s planning to automate that process with Jenkins. However, that will require a bit of time, and until then he should be allowed to perform deployments manually. We already decided that production releases should be deployed to the default Namespace, so he’ll need additional permissions.

After a short discussion, we decided that the minimum permissions required for the release manager is to perform actions on Pods, Deployments, and ReplicaSets. People with that role should be able to do almost anything related to Pods, while the allowed actions for the Deployments and ReplicaSets should be restricted to create, get, list, update, and watch. We don’t think that they should be able to delete them.

We’re not entirely confident that those are all the permissions release managers will need, but it’s a good start. We can always update the role later on if the need arises.

John will be the only release manager for now. We’ll add more users once we’re confident that the role is working as expected.

## Creating the role binding

Now that we have a plan, we can proceed to create a role and a binding that will define the permissions for release managers. The first thing we need to do is to figure out the resources, the Verbs, and the API Groups we’ll use.

### Exploring the cluster role admin

We might want to take a look at the Cluster Role admin for inspiration.

kubectl describe clusterrole admin

...

pods [] [] [create delete deletecollection patch update get list watch]

pods/attach [] [] [get list watch create delete deletecollection patch update]

pods/exec [] [] [get list watch create delete deletecollection patch update]

pods/portforward [] [] [get list watch create delete deletecollection patch update]

pods/proxy [] [] [get list watch create delete deletecollection patch update]

pods/log [] [] [get list watch]

pods/status [] [] [get list watch]

...

If we’d specify only pods as a Rule resource, we would probably not create all the Pods-related permissions we need. Even though most of the operations we can perform on Pods are covered with the pods resource, we might need to add a few sub-resources as well. For example, if we’d like to be able to retrieve the logs, we’ll need pods/log resource. In that case, pods would be a namespaced resource, and log would be a sub-resource of pods.

### The challenge

Deployment and ReplicaSet objects present a different challenge. If we go back to the output of the kubectl describe clusterrole admin command, we’ll notice that the deployments have API Groups. Unlike sub-resources that are separated from resources with a slash (/), API Groups are separated with a dot (.). So, when we see a resource like deployments.apps, it means that it is a Deployment through the API Group apps. Core API Groups are omitted.

### Understanding sub-resources and API groups

It’ll probably be easier to understand sub-resources and API Groups by exploring the definition in crb-release-manager.yml.

Most of that definition follows the same formula we already used a few times. We’ll focus only on the rules section of the ClusterRole. It is as follows.

...

rules:

- resources: ["pods", "pods/attach", "pods/exec", "pods/log", "pods/status"]

verbs: ["\*"]

apiGroups: [""]

- resources: ["deployments", "replicasets"]

verbs: ["create", "get", "list", "watch"]

apiGroups: ["", "apps", "extensions"]

...

The level of access release managers need differs between Pods on the one hand and Deployments and ReplicaSets on the other. Therefore, we split them into two groups.

### Pods

The first group specifies the pods resource together with a few sub-resources (attach, exec, log, and status). That should cover all the use cases we explored so far. Since we did not create Pod proxies nor port forwarding, they are not included.

We already said that release managers should be able to perform any operation on Pods, so the verbs consist of a single entry with an asterisk (\*). On the other hand, all Pod resources belong to the same Core group, so we did not have to specify any in the apiGroups field.

### Deployments

The second group of rules is set for deployments and replicasets resources. Considering we decided that we’ll be more restrictive with them, we specified more specific verbs, allowing release managers only to create, get, list, and watch. Since we did not specify delete, deletecollection, patch, and update Verbs, release managers will not be able to perform related actions.

As you can see, RBAC Rules can be anything from being very simple to finely tuned to particular needs. It’s up to us to decide the level of granularity we’d like to accomplish

## Creating a Release Manager Role Binding

Let’s create the role and the binding related to release managers.

kubectl create \

-f crb-release-manager.yml \

--record --save-config

To be on the safe side, we’ll describe the newly created Cluster Role, and confirm that it has the permissions we need.

kubectl describe clusterrole release-manager

The **output** is as follows.

Name: release-manager

Labels: <none>

Annotations: kubernetes.io/change-cause: kubectl create --filename=crb-release-manager.yml --record=true --save-config=true

PolicyRule:

Resources Non-Resource URLs Resource Names Verbs

--------- ----------------- -------------- -----

pods/attach [] [] [\*]

pods/exec [] [] [\*]

pods/log [] [] [\*]

pods/status [] [] [\*]

pods [] [] [\*]

deployments [] [] [create get list update watch]

replicasets [] [] [create get list update watch]

deployments.apps [] [] [create get list update watch]

replicasets.apps [] [] [create get list update watch]

deployments.extensions [] [] [create get list update watch]

replicasets.extensions [] [] [create get list update watch]

As you can see, the users assigned to the role can do (almost) anything with Pods, while their permissions with Deployments and ReplicaSets are limited to creation and viewing. They will not be able to update or delete them. Access to any other resource is forbidden.

### Verification

At the moment, John is the only User bound to the release-manager role. We’ll impersonate him, and verify that he can, for example, do anything related to Pods.

kubectl --namespace default auth \

can-i "\*" pods --as jdoe

We’ll do a similar type of verification but limited to creation of Deployments.

kubectl --namespace default auth \

can-i create deployments --as jdoe

In both cases, we got the answer yes, thus confirming that John can perform those actions.

The last verification we’ll do, before letting John know about his new permissions, is to verify that he cannot delete Deployments.

kubectl --namespace default auth can-i \

delete deployments --as jdoe

The **output** is “no”, clearly indicating that such action is forbidden.

Let’s see a few of the things John would do with his newly generated permissions. We’ll simulate that we are him by switching to the jdoe context.

kubectl config use-context jdoe

A quick validation that John can create Deployments could be done with Mongo DB.

kubectl --namespace default \

create deployment db \

--image mongo:3.3

John managed to create the Deployment in the default Namespace.

kubectl --namespace default \

delete deployment db

The **output** is as follows.

Error from server (Forbidden): deployments.apps "db" is forbidden: User "jdoe" cannot delete resource "deployments" in API group "apps" in the namespace "default"

We can see that John cannot delete the Deployment.

Let’s check whether John can perform any action in his own Namespace.

kubectl config set-context jdoe \

--cluster jdoe \

--user jdoe \

--namespace jdoe

kubectl config use-context jdoe

kubectl create deployment db --image mongo:3.3

We updated the jdoe context so that it uses the Namespace with the same name as default. Further on, we made sure that the context is used, and created a new Pod based on the mongo image.

Since John should be able to do anything within his Namespace, he should be able to delete the Deployment as well.

kubectl delete deployment db

Finally, let’s try something that requires a truly high level of permissions.

kubectl create rolebinding mgandhi \

--clusterrole=view \

--user=mgandhi \

--namespace=jdoe

The **output** is as follows.

rolebinding.rbac.authorization.k8s.io/mgandhi created

ohn is even able to add new users to his Namespace and bind them to any role (as long as it does not exceed his permissions).

## Try it yourself

A list of all the commands used in the lesson is given below.

kubectl describe clusterrole admin

kubectl create \

-f crb-release-manager.yml \

--record --save-config

kubectl describe \

clusterrole release-manager

kubectl --namespace default auth \

can-i "\*" pods --as jdoe

kubectl --namespace default auth \

can-i create deployments --as jdoe

kubectl --namespace default auth can-i \

delete deployments --as jdoe

kubectl config use-context jdoe

kubectl --namespace default \

create deployment db \

--image mongo:3.3

kubectl --namespace default \

delete deployment db

kubectl config set-context jdoe \

--cluster jdoe \

--user jdoe \

--namespace jdoe

kubectl config use-context jdoe

kubectl create deployment db --image mongo:3.3

kubectl delete deployment db

kubectl create rolebinding mgandhi \

--clusterrole=view \

--user=mgandhi \

--namespace=jdoe

# Replacing Users With Groups

Learn to amend the access to a cluster as a single user to a group of users.

**We'll cover the following**

* [The user-specific Namespace](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N0oGzxnkPpv#The-user-specific-Namespace)
* [Exploring the prospective roles](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N0oGzxnkPpv#Exploring-the-prospective-roles)
* [Amending the permissions](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N0oGzxnkPpv#Amending-the-permissions)
* [Destroying Everything](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N0oGzxnkPpv#Destroying-Everything)
* [Try it yourself](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/N0oGzxnkPpv#Try-it-yourself)

## The user-specific Namespace

Defining a single user that can access the jdoe Namespace was probably the best approach. We expect that only John will want to access it. He is the owner of that Namespace. It’s his private playground. Even if he chooses to add more users to it, he’ll probably do it independently from our YAML definitions.

After all, what’s the point of giving him god-like privileges if not to let him do things without asking for our permission or involvement? From our perspective, that Namespace has, and will continue having only one User.

## Exploring the prospective roles

We cannot apply the same logic to the permissions in default and dev Namespaces. We might choose to give everyone in our organization the view role in the default Namespace. Similarly, developers in our company should be able to deploy, update, and delete resources from the dev Namespace.

All in all, we can expect that the number of users in the view and dev bindings will increase with time. Continually adding new users is repetitive, boring, and error-prone process you probably don’t want to do. Instead of becoming a person who hates his tedious job, we can create a system that groups users based on their roles. We already did a step in that direction when we created John’s certificate.

Let’s take another look at the subject of the certificate we created earlier.

openssl req -in /usercode/certs/keys/jdoe.csr \

-noout -subject

The **output** is as follows.

subject=/CN=jdoe/O=devs

We can see that the name is “jdoe” and that he belongs to the organization “devs”. We’ll ignore the fact that he should probably belong to at least one more organization (release-manager).

If you paid close attention, you probably remember that we mentioned a few times that RBAC can be used with Users, Groups, and Service Accounts. Groups are the same as Users, except that they are validating whether the certificate attached to a request to the API belongs to a specified group (“O”), instead of a name (“CN”).

## Amending the permissions

Let’s take a quick look at yet another YAML definition of groups.yml.

apiVersion: v1

kind: Namespace

metadata:

name: dev

---

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: dev

namespace: dev

subjects:

- kind: Group

name: devs

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: ClusterRole

name: admin

apiGroup: rbac.authorization.k8s.io

---

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: view

subjects:

- kind: Group

name: devs

apiGroup: rbac.authorization.k8s.io

roleRef:

You’ll notice that the Role Binding dev and the Cluster Role Binding view are almost the same as those we used before. The only difference is in the subjects.kind field. This time, we’re using Group as the value. As a result, we’ll grant permissions to all users that belong to the organization devs.

We’ll need to switch the context back to k3d-mycluster before we apply the changes.

kubectl config use-context k3d-mycluster

kubectl apply -f groups.yml \

--record

The **output** is as follows.

namespace/dev configured

rolebinding.rbac.authorization.k8s.io/dev configured

clusterrolebinding.rbac.authorization.k8s.io/view configured

We can see that the new definition reconfigured a few resources.

Now that the new definition is applied, we can validate whether John can still create objects inside the “dev” Namespace.

kubectl --namespace dev auth \

can-i create deployments --as jdoe

The output is no, indicating that jdoe cannot create deployments. Before you start wondering what’s wrong, I should inform you that the response is expected and correct. The --as argument is impersonating John, but the certificate is still from k3d-mycluster. Kubernetes has no way of knowing that jdoe belongs to the group devs. At least, not until John issues a request with his own certificate.

Instead of using the --as argument, we’ll switch back to the jdoe context and try to create a Deployment.

kubectl config use-context jdoe

kubectl --namespace dev \

create deployment new-db \

--image mongo:3.3

This time the output is deployment "new-db" created, clearly indicating that the John as a member of the devs group can create deployments.

From now on, any user with a certificate that has /O=devs in the subject will have the same permissions as John within the dev Namespace as well as view permissions everywhere else. We just saved ourselves from constantly modifying YAML files and applying changes.

## Destroying Everything

For now, we’ll destroy the cluster and take a rest. We covered a lot of ground in this chapter. We deserve a break.

k3d cluster delete mycluster –all

## Try it yourself

A list of all the commands used in the lesson is given below.

openssl req -in /usercode/certs/keys/jdoe.csr -noout -subject

kubectl config use-context k3d-mycluster

kubectl apply -f groups.yml --record

kubectl --namespace dev auth can-i create deployments --as jdoe

kubectl config use-context jdoe

kubectl --namespace dev create deployment new-db --image mongo:3.3

k3d cluster delete mycluster --all

# omparison with Docker Swarm

This lesson compares Kubernetes RBAC with the Docker Swarm RBAC.

**We'll cover the following**

* [The Similarities](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/392GQ0x1M4A#The-Similarities)
* [The Differences](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/392GQ0x1M4A#The-Differences)

## The Similarities

Docker has RBAC. Just as Kubernetes, it is organized around subjects, roles, and resource collections. In many aspects, both provide a very similar set of features. Should we quickly declare it a tie?

## The Differences

There is one crucial difference between Kubernetes RBAC and the one provided by Docker. The latter is not free. You’d **need to purchase** Docker Enterprise Edition (EE) to secure your cluster beyond “only those with the certificate can access it.” If you do have Docker EE, you already made up your mind, and the discussion whether to use one or the other is over.

Docker EE is great, and soon it will work not only with Swarm but also with Kubernetes. You bought it, and there’s not much reason to switch to something else. However, this comparison focuses on what open source core versions can offer. It ignores third party and enterprise additions.

If we stick with an “only what’s in the box” comparison, **Kubernetes is a clear winner**. It has RBAC, and Docker Swarm doesn’t. The problem is not that Swarm doesn’t have RBAC, but that it doesn’t have any user-based authentication baked in.

This is a very short comparison. If you don’t want to purchase enterprise products, and you do need an authorization and authentication mechanism, Kubernetes is the only option. Just as with Namespaces, Kubernetes shows its strength by the sheer number of features that do not exist in Swarm.

Q. The sequence of the Authorization stages involved in sending a request to Kubernetes API is:

Ans: Authentication, Authorization, Passing the admission control

Q. Groups are the same as Users, except that they are validating whether the certificate attached to a request to the API belongs to a specified group.

Ans: YES

# What's Next?

Recap what we have learned so far and what we are going to learn next.

**We'll cover the following**

* [Summary](https://www.educative.io/module/lesson/a-practical-guide-to-kubernetes/RM3zPj8V0mq#Summary)

## Summary

Authorization and authentication are critical security components. Without a proper set of permissions, we are risking exposure with potentially devastating results. Moreover, with appropriate Rules, Roles, and RoleBindings, we can make a cluster not only more secure but also increase collaboration between different members of our organization. The only trick is to find a right balance between tight security and freedom. It takes time until that equilibrium is established.

RBAC combined with Namespaces provides an excellent separation. Without Namespaces, we’d need to create multiple clusters. Without RBAC, those clusters would be exposed or locked down to only a handful of users. The two combined provide an excellent way to increase collaboration without sacrificing security.

We did not explore Service Accounts. They are the third kind of Subjects, besides Users and Groups. We’ll leave that for some other time and place since they are used primarily for Pods that need to access the Kubernetes API. This chapter focused on humans and the ways we can enable them to reach a cluster in a safe and controlled manner.

We are still missing one important restriction. By combining Namespaces and RBAC, we can restrict what users can do. However, that will not prevent them from deploying applications that could potentially bring down the whole cluster. We need to add Resource Quotas to the mix. That will be the subject of the next chapter.

ℹ️ If you’d like to know more about Roles, please explore the [Role v1 rbac](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/#role-v1-rbac-authorization-k8s-io) and [ClusterRole v1 rbac](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/" \l "clusterrole-v1-rbac-authorization-k8s-io" \t "_blank) API documentation. Similarly, you might want to visit the [RoleBinding v1 rbac](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/" \l "rolebinding-v1-rbac-authorization-k8s-io" \t "_blank) and [ClusterRoleBinding v1 rbac](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/" \l "clusterrolebinding-v1-rbac-authorization-k8s-io" \t "_blank) API documentation as well.