**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**

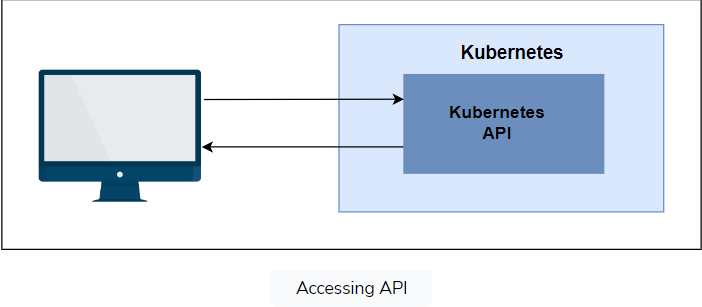
Explore the Kubernetes API and the process to access it.

**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.