Peeking into Pre-Defined Cluster Roles

In this lesson, we will look into all the pre-defined cluster roles.

WE'LL COVER THE FOLLOWING

- ·
- Switching from John to Us
- Looking into Roles and Cluster Roles
 - Looking into view
 - Looking into edit
 - Looking into admin
 - Looking into cluster-admin

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 minikube user.

kubectl config use-context minikube
kubectl get all



Now that we switched to the minikube context (and the minikube user), 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 kubect1 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 cani 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 minikube) 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).

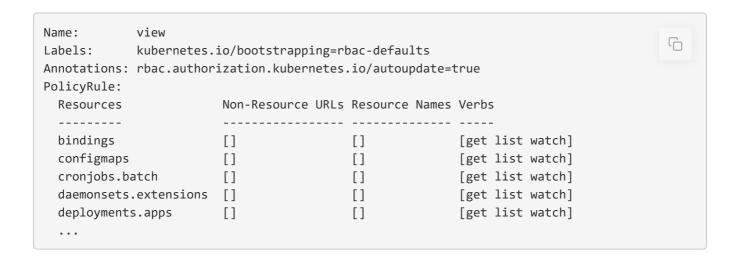
```
NAME AGE
admin 1h
cluster-admin 1h
edit 1h
view 1h
```

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.



It contains a long list of resources, all of them with the <code>get</code>, <code>list</code>, and <code>watch</code> 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

I at's axplore another pre-defined Cluster Role

Let's explore dilottier 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 roles.rbac.authorization.k8s.io [] [] [create delete deletecollection get list ...
```

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

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

```
kubectl describe clusterrole \
cluster-admin
```

The **output** is as follows.

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 minikube 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 clusteradmin role is bound to minikube, we did verify that it can do anything.

In the next lesson, we will learn how to create role bindings.