Granting Access as a Release Manager

In this lesson, we will let the user have access to the cluster as a release manager.

WE'LL COVER THE FOLLOWING

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

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
```

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

```
...
pods                                            []                 []              [create de
  pods/attach                                     []                 []              [get lis
  pods/exec                                       []                 []              [get lis
  pods/portforward                                []                 []              [get lis
  pods/proxy                                      []                 []              [get lis
  pods/log                                        []                 []              [get lis
  pods/status                                     []                 []              [get lis
...
```

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 `auth/crb-release-manager.yml`.

```
cat auth/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 auth/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:  kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"rbac.authorizat
              kubernetes.io/change-cause=kubectl create --filename=auth/crb-release-manager.y
PolicyRule:
  Resources              Non-Resource URLs Resource Names Verbs
  ---------              ----------------- -------------- -----
  deployments            []                []             [create get list update watch]
  deployments.apps       []                []             [create get list update watch]
  deployments.extensions []                []             [create get list update watch]
  pods                   []                []             [*]
  pods/attach            []                []             [*]
  pods/exec              []                []             [*]
  pods/log               []                []             [*]
  pods/status            []                []             [*]
  replicasets            []                []             [create get list update watch]
  replicasets.apps       []                []             [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
```

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 run 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
```

John 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).

In the next lesson, we will allow a group of users to work with the cluster.