# Kubernetes RBAC

## Configure RBAC In Your Kubernetes Cluster

### Introduction

This guide will go through the basic Kubernetes Role-Based Access Control (RBAC) API Objects, showing a common use case (create a user with limited access. At the end of this guide, you should have enough knowledge to implement RBAC policies in your cluster.
From Kubernetes 1.6 onwards, RBAC policies are enabled by default. RBAC policies are vital for the correct management of your cluster, as they allow you to specify which types of actions are permitted depending on the user and their role in your organization.
Examples include:

- Secure your cluster by granting privileged operations (accessing secrets, for example) only to admin users.
- Force user authentication in your cluster.
- Limit resource creation (such as pods, persistent volumes, deployments) to specific namespaces. You can also use quotas to ensure that resource usage is limited and under control.
- Have a user only see resources in their authorized namespace. This allows you to isolate resources within your organization (for example, between departments).

As a consequence of having RBAC enabled by default, you may see errors like this if you try to install an application that doesn't create the appropriate roles and bindings:

```
the server does not allow access to the requested resource
```

This guide will show you how to work with RBAC so you can properly deal with issues like these.

### RBAC API objects

One basic Kubernetes feature is that all its resources are modeled API objects , which allow CRUD (Create, Read, Update, Delete) operations.

Examples of resources are:

- Pods .
- PersistentVolumes .
- ConfigMaps .
- Deployments .
- Nodes .

- Secrets .
- Namespaces .

Examples of possible operations over these resources are:

- *create*
- *get*
- *delete*
- *list*
- *update*
- *edit*
- *watch*
- *exec*

To manage RBAC in Kubernetes, apart from resources and operations, we need the following elements:

- Rules: A rule is a set of operations (verbs) that can be carried out on a group of resources which belong to different API Groups.
- Roles and ClusterRoles: Both consist of rules. The difference between a Role and a ClusterRole is the scope: in a Role, the rules are applicable to a single namespace, whereas a ClusterRole is cluster-wide, so the rules are applicable to more than one namespace. ClusterRoles can define rules for cluster-scoped resources (such as nodes) as well. Both Roles and ClusterRoles are mapped as API Resources inside our cluster.
- Subjects: These correspond to the entity that attempts an operation in the cluster. There are three types of subjects:
  - User Accounts: These are global, and meant for humans or processes living outside the cluster. There is no associated resource API Object in the Kubernetes cluster.
  - Service Accounts: This kind of account is namespaced and meant for intra-cluster processes running inside pods, which want to authenticate against the API.
  - Groups: This is used for referring to multiple accounts. There are some groups created by default such as *cluster-admin* (explained in later sections).
- RoleBindings and ClusterRoleBindings: Just as the names imply, these bind subjects to roles (i.e. the operations a given user can perform). As for Roles and ClusterRoles, the difference lies in the scope: a RoleBinding will make the rules effective inside a namespace, whereas a ClusterRoleBinding will make the rules effective in all namespaces.

## Create user with limited namespace access

In this example, we will create the following User Account:

- Username: employee

- Group: Acme

We will add the necessary RBAC policies so this user can fully manage deployments (i.e. use *kubectl run* command) only inside the *office* namespace. At the end, we will test the policies to make sure they work as expected.

## Create office namespace

Start off by creating the `office` namespace

```
kubectl create namespace office
```

## Create the user credentials

As previously mentioned, Kubernetes does not have API Objects for User Accounts. Of the available ways to manage authentication (see Kubernetes official documentation for a complete list), we will use OpenSSL certificates for their simplicity. The necessary steps are:

- Create a private key for your user. In this example, we will name the file `employee.key`:

```
openssl genrsa -out employee.key 2048
```

- Create a certificate sign request `employee.csr` using the private key you just created (`employee.key` in this example). Make sure you specify your username and group in the `-subj` section (CN is for the username and O for the group). As previously mentioned, we will use `employee` as the name and `acme` as the group:

```
openssl req -new -key employee.key -out employee.csr -subj "/CN=employee/
O=acme"
```

Locate your Kubernetes cluster certificate authority (CA). This will be responsible for approving the request and generating the necessary certificate to access the cluster API. It's default location is `/etc/kubernetes/pki`
Confirm that the files `ca.crt` and `ca.key` exist in this location.

```
ls -ltr /etc/kubernetes/pki
```

- Generate the final certificate `employee.crt` by approving the certificate sign request, `employee.csr`, you made earlier. In this example, the certificate will be valid for 500 days:

```
sudo openssl x509 -req -in employee.csr -CA /etc/kubernetes/pki/ca.crt -CAkey /
etc/kubernetes/pki/ca.key -CAcreateserial -out employee.crt -days 500
```

- Save both `employee.crt` and `employee.key` in a safe location (in this example we will use */home/ubuntu/.certs/*).

```
mkdir -p /home/ubuntu/.certs && mv employee.crt employee.key /home/
ubuntu/.certs
```

- Add a new context with the new credentials for your Kubernetes cluster

```
kubectl config set-credentials employee --client-certificate=/home/
ubuntu/.certs/employee.crt  --client-key=/home/ubuntu/.certs/employee.key
kubectl config set-context employee-context --cluster=kubernetes --
namespace=office --user=employee
```

Now you should get an access denied error when using the `kubectl` CLI with this configuration file. This is expected as we have not defined any permitted operations for this user.

```
kubectl --context=employee-context get pods
```

## Create the role for managing deployments

- Create a `role-deployment-manager.yaml` file with the content below. In this `yaml` file we are creating the rule that allows a user to execute several operations on Deployments, Pods and ReplicaSets (necessary for creating a Deployment), which belong to the `core` (expressed by "" in the `yaml` file), `apps`, and `extensions` API Groups:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  namespace: office
  name: deployment-manager
rules:
- apiGroups: ["", "extensions", "apps"]
  resources: ["deployments", "replicasets", "pods"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"] # You
can also use ["*"]
```

- Create the Role in the cluster using the `kubectl create role` command:

```
kubectl create -f role-deployment-manager.yaml
```

## Bind the role to the employee user

- Create a `rolebinding-deployment-manager.yaml` file with the content below. In this file, we are binding the `deployment-manager` Role to the User Account `employee` inside the `office` namespace:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: deployment-manager-binding
  namespace: office
subjects:
```

```
- kind: User
  name: employee
  apiGroup: ""
roleRef:
  kind: Role
  name: deployment-manager
  apiGroup: ""
```

- Deploy the RoleBinding by running the `kubectl create` command:

```
kubectl create -f rolebinding-deployment-manager.yaml
```

## Test the RBAC setup

Now you should be able to execute the following commands without any issues:

```
kubectl --context=employee-context run --image nginx acme-nginx
```

You can also list Pods as `employee` user

```
kubectl --context=employee-context get pods
```

If you run the same command with the `--namespace=default` argument, it will fail, as the `employee` user does not have access to this namespace.

```
kubectl --context=employee-context get pods --namespace=default
```

Success! You've created a user with limited permissions in your Kubernetes cluster