Kubernetes Role Based Access Control (RBAC)

When a request is sent to the API Server, it first needs to be authenticated (to make sure the requestor is known by the system) before it's authorized (to make sure the requestor is allowed to perform the action requested).

RBAC is a way to define which users can do what within a Kubernetes cluster.

Role-based access control (RBAC) is a way of granting users granular access to Kubernetes API resources. RBAC is a security design that restricts access to Kubernetes resources based on the role the user holds.

If you are working on Kubernetes for some time you may have faced a scenario where you have to give some users limited access to your Kubernetes cluster**. For example you may want a user, say Michale from development, to have access only to some resources that are in the development namespace and nothing else**. To achieve this type of role based access, we use the concept of Authentication and Authorization in Kubernetes.

Broadly, there are three kinds of users that need access to a Kubernetes cluster:

1. Developers/Admins:

Users that are responsible to do administrative or developmental tasks on the cluster. This includes operations like upgrading the cluster or creating the resources/workloads(PODS,Deployments,PV,PVC..ETC.) on the cluster.

2. End Users:

Users that access the applications deployed on our Kubernetes cluster.**Access restrictions for these users are managed by the applications themselves**. For example, a web application running on Kubernetes cluster, will have its own security mechanism in place, to prevent unauthorized access.

3. Applications/Bots:

There is a possibility that other applications need access to Kubernetes cluster or API, typically to talk to resources or workloads inside the cluster. Kubernetes facilitates this by using Service Accounts..

API Objects for configuring RBAC:***Role***,***ClusterRole***,***RoleBinding***and***ClusterRoleBinding***.

Role

- Role defines what can be done to Kubernetes Resources.
- Role contains one or more rules that represent a set of permissions.
- Roles are namespaced, meaning Roles work within the constraints of a namespace. It would default to the default namespace if none was specified.
- After creating a Role, you assign it to a user or group of users by creating a RoleBinding.

Ex:

Here's an example Role in the "default" namespace that can be used to grant read access to pods:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

In the rules above we:

1. apiGroups: [""] – set core API group
2. resources: ["pods"] – which resources are allowed for access
3. ["get", "watch", "list"] – which actions are allowed over the resources above.

RoleBinding

- Role Binding is used for granting permission to a Subject.
- RoleBinding holds a list of subjects (users, groups, or service accounts), and a reference to the role being granted.
- Role and RoleBinding are used in namespaced scoped.
- RoleBinding may reference any Role in the same namespace.
- After you create a binding, you cannot change the Role that it refers to. If you do want to change the roleRef for a binding, you need to remove the binding object and create a again.

Example

Here is an example of a RoleBinding that grants the “pod-reader” Role to the user “michale” within the “default” namespace. This allows “michale” to read pods in the “default” namespace.

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
namespace: default
subjects:
- kind: User
  name: michale
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Here we set:

subjects:
kind: User – an object type which will have access, in our case this is a regular user
name: example-user – a user's name to set the permissions
roleRef:
kind: Role – what exactly will be attached to the user, in this case, it is the Role object type
name: pod-reader the role name as it was set in the name: pod-reader in the example above

```
kind: Role
apiVersion:
rbac.authorization.k8s.io/v1beta1
metadata:
   name: pod-read-create
   namespace: test
rules:
 - apiGroups: [""]
   resources: ["pods"]
   verbs: ["get", "list", "create"]
```

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: salme-pods
  namespace: test
subjects:
- kind: User
  name: jsalmeron
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: ns-admin
  apiGroup: rbac.authorization.k8s.io
```

ClusterRole

- ClusterRole works the same as Role, but they are applied to the cluster as a whole.
- ClusterRoles are not bound to a specific namespace. ClusterRole give access across more than one namespace or all namespaces.
- After creating a ClusterRole, you assign it to a user or group of users by creating a ClusterRoleBinding.
- ClusterRoles are cluster-scoped, you can use ClusterRoles to control access to different kinds of resources than you can with Roles.

Cluster-scoped resources (e.g. Nodes, PersistentVolumes)

Example

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: demo-clusterrole
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

ClusterRoleBinding

- ClusterRole and ClusterRoleBinding function like Role and RoleBinding, except they have wider scope.
- RoleBinding grants permissions within a specific namespace, whereas a ClusterRoleBinding grants access cluster-wide and to multiple namespaces.
- ClusterRoleBinding is binding or associating a ClusterRole with a Subject (users, groups, or service accounts).

Example

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: demo-clusterrolebinding
subjects:
- kind: User
  name: Mithun
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: demo-clusterrole
  apiGroup: rbac.authorization.k8s.io
```

Role vs ClusterRole

A ClusterRole looks similar to a Role with the only difference that we have to set its kind as ClusterRole.

The difference is that Role is used inside of a namespace, while ClusterRole is cluster-wide permission without a namespace boundaries, for example:

allow access to a cluster nodes
resources in all namespaces
allow access to endpoints like /healthz

RBAC in Kubernetes is based on three key concepts:

1. Verbs: This is a set of operations that can be executed on resources. There are many verbs, but they're all Create, Read, Update, or Delete (also known as CRUD).
2. API Resources The set of Kubernetes API Objects available in the cluster are called Resources. For examples, Pods, Deployments, Services, Nodes, PersistentVolumes etc.
3. Subjects: These are the objects (Users, Groups, Processes(Service Account)) allowed access to the API, based on Verbs and Resources.

Keep in mind that once you'll create a Binding you'll not be able to edit its roleRef value – instead, you'll have to delete a Binding and recreate and again

- Kubernetes uses RBAC to control different access levels to its resources depending on the rules set in Roles or ClusterRoles.
- Roles and ClusterRoles use API namespaces, verbs and resources to secure access.
- Roles and ClusterRoles are ineffective unless they are linked to a subject (User, serviceAccount...etc) through RoleBinding or ClusterRoleBinding.
- Roles work within the constraints of a namespace. It would default to the "default" namespace if none was specified.
- ClusterRoles are not bound to a specific namespace as they apply to the cluster as a whole.

```
Scheduling
----------

NodeSelector
NodeAffinity


PodAffinity & Pod AntiAffinity

Taints & Tolerations


Node Maintence
--------------

kubectl cordon
kubectl drain (Cordon & Evict Pods from the node)
kubectl uncordon
```

apiGroup: rbac.authorization.k8s.io