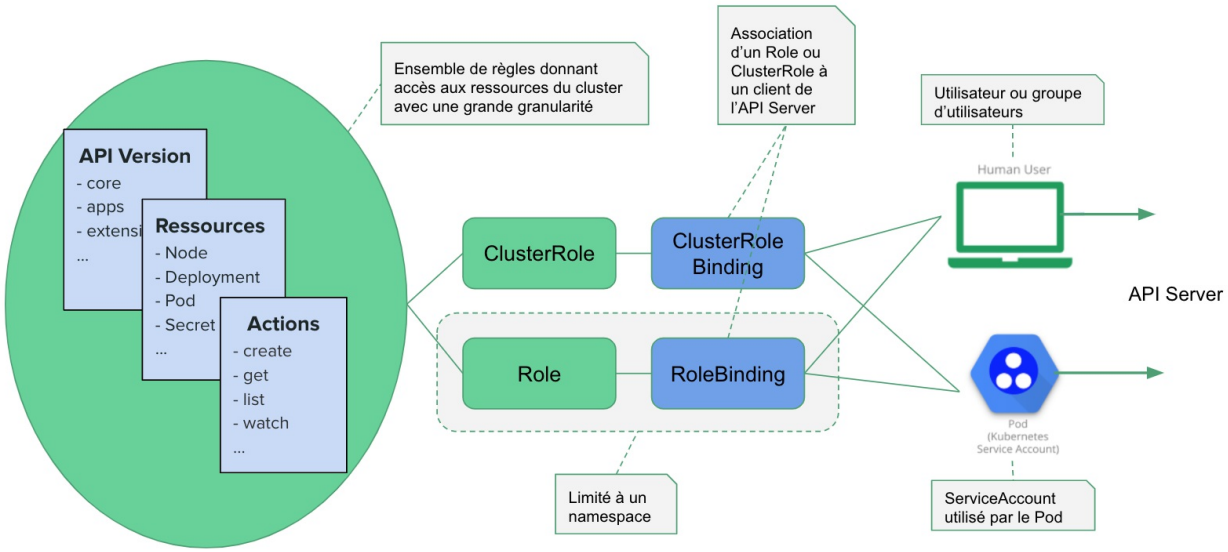
Dans cet exercice, nous allons créer un ServiceAccount et lui donner les droits qui lui permettront de lister les Pods présents dans le namespace *default*, nous utiliserons pour cela les ressources *Role* et *RoleBinding*. Depuis un Pod simple ayant accès à ce ServiceAccount, nous enverrons ensuite des requêtes HTTP à l'API Server.

Le schéma suivant donne une vision globale des différentes ressources qui interviennent lors de la mise en place de rêgles RBAC, nous utiliserons certaines de ces ressources dans cet exercice.



API HTTP Rest exposée par l'API Server

Si vous travaillez avec un cluster Kubernetes, vous utilisez probablement l'utilitaire en ligne de commande *kubectl* ou bien l'interface web pour gérer le cluster et les applications qui sont déployées. Ces outils envoient des requêtes aux endpoints HTTP de l'API Server .

La documentation de l'API exposée par l'API Server est disponible sur le site officiel de Kubernetes, https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.14

Un exemple simple: la liste des Pods qui tournent dans le namespace *default* peut être obtenue avec la requête https://API_SERVER/api/v1/namespaces/default/pods/. Bien sur, il faudra s'authentifier et avoir le droit de faire cette action.

Accès à l'API Server depuis un Pod

Un grand nombre d'applications exécutées dans le cluster (c'est à dire: tournant dans des Pods) doivent communiquer avec l'API Server. Parmi celles-ci, il y a les processus exécutés

sur les Masters (scheduler, controller manager, proxy, …), mais également toutes les applications qui réalisent des actions d'administration diverses sur le cluster.

Par exemple, certaines applications peuvent avoir besoin de connaitre:

- l'état des nodes du cluster
- les namespaces qui existent
- les pods qui tournent dans le cluster ou dans un namespace particulier
- …

Pour communiquer avec le serveur API, un pod utilise un ServiceAccount contenant un token d'authentification. Des Roles (par exemple: le droit de lister tous les pods dans un namespace) ou des ClusterRoles (par exemple, le droit de lire toutes les ressources de type Secret du cluster) peuvent ensuite être liées à ce ServiceAccount respectivement avec les ressources RoleBinding et ClusterRoleBinding, afin que le ServiceAccount soit autorisé à effectuer ces actions.

Depuis l'extérieur du cluster: il est possible d'accéder à l'API Server en utilisant le endpoint HTTP spécifié dans le fichier de configuration ($HOME/.kube/config par défaut). Par exemple, si vous utilisez un Kubernetes managé sur DigitalOcean, le endpoint ressemblera à https://b703a4fd-0d56-4802-a354-ba2c2a767a77.k8s.ondigitalocean.com.

Depuis l'intérieur du cluster (c'est à dire: depuis un pod): il est possible d'accéder à l'API Server via le service de type ClusterIP nommé kubernetes. A noter que ce service existe par défaut et qu'il automatiquement recréé au cas ou il serait supprimé par erreur.

```
$ kubectl get svc
NAME       TYPE      CLUSTER-IP  EXTERNAL-IP PORT(S) AGE
kubernetes ClusterIP 10.96.0.1   <none>      443/TCP 23h
```

Si l'on a les droits suffisant, on peut lister les Pods qui sont dans le namespace *default* avec la requête GET *https://kubernetes/api/v1/namespaces/default/pods/*

ServiceAccount "default"

Il existe un ServiceAccount nommé *default* pour chaque namespace, comme vous pouvez le vérifier avec la commande suivante:

```
$ kubectl get sa --all-namespaces | grep default
default           default                             1         6m19s
```

```
kube-public       default                                1         6m19s
kube-system       default                                1         6m19s
```

Chaque Pod a accès au ServiceAccount *default* du namespace dans lequel il tourne afin de communiquer avec l'API Server.

Note: les ServiceAccount *default* n'ayant pas beaucoup de droits, si un Pod a besoin de communiquer avec l'API Server il faudra donc créer un ServiceAccount et lui donner les droits nécessaires en utilisant des ressources de type *Role* / *ClusterRole* ainsi que *RoleBining* / *ClusterRoleBinding*. Nous verrons un exemple d'utilisation de ces ressources un plus loin.

Token d'authentification

Un ServiceAccount a accès à un token (via un Secret) afin de pouvoir s'authentifier auprès de l'API Server. Utilisez la commande suivante pour récupérer le secret lié au ServiceAccount *default* du namespace *default*:

```
$ kubectl get sa default -o yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: "2019-05-27T18:38:51Z"
  name: default
  namespace: default
  resourceVersion: "333"
  selfLink: /api/v1/namespaces/default/serviceaccounts/default
  uid: ac5aa972-80ae-11e9-854d-0800278b691f
secrets:
- name: default-token-dffkj
```

Vérifiez ensuite ce que contient le secret correspondant:

```
$ kubectl get secret default-token-dffkj -o yaml
apiVersion: v1
data:
  ca.crt: LS0tLS1CRU...0tLS0tCg==
  namespace: ZGVmYXVsdA==
  token: ZXlKaGJHY2...RGMUlIX2c=
kind: Secret
metadata:
  annotations:
    kubernetes.io/service-account.name: default
    kubernetes.io/service-account.uid: ac5aa972-80ae-11e9-854d-0800278b691f
  creationTimestamp: "2019-05-27T18:38:51Z"
  name: default-token-dffkj
```

```
  namespace: default
  resourceVersion: "332"
  selfLink: /api/v1/namespaces/default/secrets/default-token-dffkj
  uid: ac5cf0b9-80ae-11e9-854d-0800278b691f
type: kubernetes.io/service-account-token
```

Le token est le codage en base64 d'un JWT Token. En décodant ce token :

```
$ kubectl get secret default-token-dffkj -o jsonpath='{ .data.token}' | base64 -
-decode
```

puis en utilisant un utilitaire en ligne de commande ou une version en ligne comme https://jwt.io, on peut facilement décoder le token et voir le payload qu'il contient.

En utilisant l'une de ces approches, vous obtiendrez un résultat semblable de celui ci-dessous:

```
{
  "iss": "kubernetes/serviceaccount",
  "kubernetes.io/serviceaccount/namespace": "default",
  "kubernetes.io/serviceaccount/secret.name": "default-token-dffkj",
  "kubernetes.io/serviceaccount/service-account.name": "default",
  "kubernetes.io/serviceaccount/service-account.uid": "ac5aa972-80ae-11e9-854d-
0800278b691f",
  "sub": "system:serviceaccount:default:default"
}
```

Dans la suite, nous allons voir comment utiliser ce token pour communiquer avec l'API Server.

Appel à l'API Server en HTTP

La spécification suivante définie un Pod très simple composé d'un container. Copiez celle-ci dans pod-default.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-default
spec:
  containers:
  - name: alpine
    image: alpine:3.9
    command:
```

```
    - "sleep"
    - "10000"
```

puis créez le Pod avec la commande suivante:

```
$ kubectl apply -f pod-default.yaml
```

Vous allez maintenant lancer un shell dans le container *alpine* du Pod que vous venez de créer puis installer curl afin de pouvoir envoyer des requêtes HTTP à l'API Server.

```
$ kubectl exec -ti pod-default -- sh
/ # apk add --update curl
```

Appel anonyme

Depuis ce shell, vous allez essayer d'obtenir des informations sur l'API, sans être authentifié. Vous allez pour cela envoyer une requête HTTP simple à l'API Server.

```
/ # curl https://kubernetes/api/v1 --insecure
```

Note: depuis un Pod tournant dans le cluster, il est possible d'accéder à l'API Server en utilisant le service nommé *kubernetes*, présent par défaut.

Vous devriez obtenir un message d'erreur semblable à celui ci-dessous. Celui-ci indique qu'un utilisateur anonyme n'est pas authorisé à effectuer cette requête:

```
/ # curl https://kubernetes/api/v1 --insecure
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {

  },
  "status": "Failure",
  "message": "forbidden: User \"system:anonymous\" cannot get path \"/api/v1\"",
  "reason": "Forbidden",
  "details": {

  },
  "code": 403
```

```
}
```

Appel en utilisant le token disponible dans le Pod

Le container *alpine* a accès au token du ServiceAccount *default* depuis le fichier *cat /run/secrets/kubernetes.io/serviceaccount/token*. Récupérez le contenu de ce token puis lancez la même commande que précédemment mais en utilisant le token pour l'authentification.

```
/ # TOKEN=$(cat /run/secrets/kubernetes.io/serviceaccount/token)
/ # curl -H "Authorization: Bearer $TOKEN" https://kubernetes/api/v1/ --insecure
```

Cette fois-ci vous obtiendrez une liste des ressources disponibles dans l'API.

```
/ # curl -H "Authorization: Bearer $TOKEN" https://kubernetes/api/v1/ --insecure
{
  "kind": "APIResourceList",
  "groupVersion": "v1",
  "resources": [
    {
      "name": "bindings",
      "singularName": "",
      "namespaced": true,
      "kind": "Binding",
      "verbs": [
        "create"
      ]
    },
    ...
```

Essayez maintenant, avec la commande suivante, de lister les Pods présents dans le namespace *default*:

```
/ # curl -H "Authorization: Bearer $TOKEN"
https://kubernetes/api/v1/namespaces/default/pods/ --insecure
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {

  },
  "status": "Failure",
  "message": "pods is forbidden: User \"system:serviceaccount:default:default\"
cannot list resource \"pods\" in API group \"\" in the namespace \"default\"",
```

```
  "reason": "Forbidden",
  "details": {
    "kind": "pods"
  },
  "code": 403
}
```

Le ServiceAccount *default* n'a pas les droits suffisants pour pouvoir lister les Pods du namespace *default*. Dans la suite, nous allons créer notre propre ServiceAccount et lui donner les droits nécessaires pour réaliser cette action.

Sortez du container avec la commande *exit*

ServiceAccount "demo-sa"

Nous allons maintenant créer un nouveau ServiceAccount dans le namespace *default*, nous appelerons *demo-sa*.

Création du ServiceAccount

Copiez la spécification suivante dans le fichier *demo-sa.yaml*:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: demo-sa
```

puis créez ce nouveau ServiceAccount:

```
$ kubectl apply -f demo-sa.yaml
```

Création d'un Role

Un ServiceAccount n'a pas d'utilité si des droits ne lui sont pas associés. C'est ce que nous allons faire en définissant un *Role* et en l'associant au ServiceAccount via un *RoleBinding*.

La spécification suivante définie un *Role* permettant de lister les Pods qui sont dans le namespace *default*.

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: Role
metadata:
  name: list-pods
  namespace: default
rules:
  - apiGroups:
      - ''
    resources:
      - pods
    verbs:
      - list
```

Copiez cette spécification dans le fichier *role-list-pods.yaml* et créez le avec la commande suivante:

```
$ kubectl apply -f role-list-pods.yaml
```

Binding du ClusterRole avec le ServiceAccount

La dernière étape consiste à associer le Role créé précédemment avec le ServiceAccount. Cela se fait à l'aide d'un RoleBinding dont la spécification se trouve ci-dessous:

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
  name: list-pods_demo-sa
  namespace: default
roleRef:
  kind: Role
  name: list-pods
  apiGroup: rbac.authorization.k8s.io
subjects:
  - kind: ServiceAccount
    name: demo-sa
    namespace: default
```

Copiez celle-ci dans le fichier *role-binding-list-pods.yaml* puis créez le avec la commande suivante:

```
$ kubectl apply -f role-binding-list-pods.yaml
```

Le ServiceAccount *demo-sa* a maintenant les droits lui permettant de lister les Pods du namespace *default*. Vous allez le vérifier en lançant un Pod utilisant ce ServiceAccount au lieu de celui par défaut.

Lancement d'un Pod simple

La spécification suivante définie un Pod très simple contenant un seul container.

Note: nous avons ajouté la clé *serviceAccountName: demo-sa* afin de spécifier le ServiceAccount que ce Pod pourra utiliser. Si nous ne l'avions pas spécifié, le ServiceAccount *default* aurait été utilisé.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-demo-sa
spec:
  serviceAccountName: demo-sa
  containers:
  - name: alpine
    image: alpine:3.9
    command:
    - "sleep"
    - "10000"
```

Copiez le contenu ci-dessus dans le fichier *pod-demo-sa.yaml* puis créez le Pod avec la commande suivante:

```
$ kubectl apply -f pod-demo-sa.yaml
```

Comme précédemment, lancez un shell dans le container *alpine* du Pod et installez curl, vous vous en servirez dans la suite pour envoyer des requêtes HTTP à l'API Server.

```
$ kubectl exec -ti pod-demo-sa -- sh
/ # apk add --update curl
```

Récupération du token

Le token du ServiceAccount *demo-sa* est disponible dans le fichier

/run/secrets/kubernetes.io/serviceaccount/token du container *alpine*. Comme vous l'aviez fait avec le token du ServiceAccount *default*, récupérez le token avec la commande suivante:

```
/ # TOKEN=$(cat /run/secrets/kubernetes.io/serviceaccount/token)
```

puis utilisez le pour lister les Pods du namespace *default* (vous pouvez trouver l'URL à utiliser dans la documentation de l'API):

```
/ # curl -H "Authorization: Bearer $TOKEN"
https://kubernetes/api/v1/namespaces/default/pods/ --insecure
```

Vous obtiendrez cette fois-ci, dans un format json, la liste des Pods tournant sur le cluster.

Ce qu'il faut retenir

Par défaut, chaque Pod peut communiquer avec l'API Server du cluster sur lequel il tourne. Si aucun ServiceAccount n'est spécifié dans la description du Pod, le ServiceAccount *default* du namespace est utilisé. Celui-ci ayant des droits restreints, un ServiceAccount est généralement créé pour chaque application, en donnant à celui-ci les droits nécessaires.

Pour s'authentifier auprès de l'API Server, le Pod utilise le token attaché au ServiceAccount. Ce token est disponible depuis le système de fichiers de chaque container du Pod.

Dans cet exemple, nous avons utilisé *curl* pour faire des requêtes à l'API Server. Pour des applications réelles, nous utiliserions une librairie dédiée dans le language correspondant.