Create Kubernetes Service Accounts and Kubeconfigs

Manually create a Kubernetes Service Account to use with Spinnaker.

What Spinnaker needs to connect to Kubernetes

When connecting Spinnaker to Kubernetes, Spinnaker needs the following:

- A service account in the relevant Kubernetes cluster (or namespace in a cluster). *In Kubernetes, a service account exists in a given namespace but may have access to other namespaces or to the whole cluster*
- Permissions for the service account to create/read/update/delete objects in the relevant Kubernetes cluster (or namespace)
- A kubeconfig that has access to the service account through a token or some other authentication method.

Attention

This document primarily uses kubectl and assumes you have access to permissions that can create and/or update these resources in your Kubernetes cluster:

- Kubernetes Service Account(s)
- Kubernetes Roles and Rolebindings
- (Optionally) Kubernetes ClusterRoles and Rolebindings

Create the Kubernetes Service Account

You can use the following manifest to create a service account.
Replace NAMESPACE with the namespace you want to use and, optionally, rename the service account.

```
# spinnaker-service-account.yml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: spinnaker-service-account
  namespace: NAMESPACE
```

Copy

Then create the object:

```
kubectl apply -f spinnaker-service-account.yml
```
Copy

Grant cluster-admin permissions

Do this only if you want to grant the service account access to all namespaces in your cluster. A Kubernetes ClusterRoleBinding exists at the cluster level, but the *subject* of the ClusterRoleBinding exists in a single namespace. Again, you *must* specify the namespace where your service account lives.

```
# spinnaker-clusterrolebinding.yml
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: spinnaker-admin
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: spinnaker-service-account
  namespace: NAMESPACE
```
Copy

Then, create the binding:

```
kubectl apply -f spinnaker-clusterrolebinding.yml
```
Copy

Grant namespace-specific permissions

If you only want the service account to access specific namespaces, you can create a role with a set of permissions and rolebinding to attach the role to the service account. You can do this multiple times. Additionally, you will have to explicitly do this for the namespace where the service account is, as it is not implicit.

Important points:

- A Kubernetes Role exists in a given namespace and grants access to items in that namespace

- A Kubernetes RoleBinding exists in a given namespace and attaches a role in that namespace to some principal (in this case, a service account). The principal (service account) may be in another namespace.
- If you have a service account in namespace source and want to grant access to namespace target, then do the following:
    - Create the service account in namespace source
    - Create a Role in namespace target
    - Create a RoleBinding in namespace target, with the following properties:
        - RoleRef pointing to the Role (that is in the same namespace target)
        - Subject pointing to the service account and namespace where the service account lives (in namespace source)

Change the names of resources to match your environment, as long as the namespaces are correct, and the subject name and namespace match the name and namespace of your service account.

```
# spinnaker-role-and-rolebinding-target.yml
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: spinnaker-role
  namespace: target # Should be namespace you are granting access to
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: spinnaker-rolebinding
  namespace: target # Should be namespace you are granting access to
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: spinnaker-role # Should match name of Role
subjects:
```

```
- namespace: source # Should match namespace where SA lives
  kind: ServiceAccount
  name: spinnaker-service-account # Should match service account name, above
```
Copy

Then, create the object:

```
kubectl apply -f spinnaker-role-and-rolebinding-target.yml
```
Copy

Get the service account and token

Run these commands (or commands like these) to get the token for your service account and create a kubeconfig with access to the service account.

This file will contain credentials for your Kubernetes cluster and should be stored securely.

```
# Update these to match your environment
SERVICE_ACCOUNT_NAME=spinnaker-service-account
CONTEXT=$(kubectl config current-context)
NAMESPACE=spinnaker

NEW_CONTEXT=spinnaker
KUBECONFIG_FILE="kubeconfig-sa"


SECRET_NAME=$(kubectl get serviceaccount
${SERVICE_ACCOUNT_NAME} \
  --context ${CONTEXT} \
  --namespace ${NAMESPACE} \
  -o jsonpath='{.secrets[0].name}')
TOKEN_DATA=$(kubectl get secret ${SECRET_NAME} \
  --context ${CONTEXT} \
  --namespace ${NAMESPACE} \
  -o jsonpath='{.data.token}')

TOKEN=$(echo ${TOKEN_DATA} | base64 -d)

# Create dedicated kubeconfig
# Create a full copy
```

```
kubectl config view --raw > ${KUBECONFIG_FILE}.full.tmp
# Switch working context to correct context
kubectl --kubeconfig ${KUBECONFIG_FILE}.full.tmp config use-context
${CONTEXT}
# Minify
kubectl --kubeconfig ${KUBECONFIG_FILE}.full.tmp \
  config view --flatten --minify > ${KUBECONFIG_FILE}.tmp
# Rename context
kubectl config --kubeconfig ${KUBECONFIG_FILE}.tmp \
  rename-context ${CONTEXT} ${NEW_CONTEXT}
# Create token user
kubectl config --kubeconfig ${KUBECONFIG_FILE}.tmp \
  set-credentials ${CONTEXT}-${NAMESPACE}-token-user \
  --token ${TOKEN}
# Set context to use token user
kubectl config --kubeconfig ${KUBECONFIG_FILE}.tmp \
  set-context ${NEW_CONTEXT} --user ${CONTEXT}-${NAMESPACE}-
token-user
# Set context to correct namespace
kubectl config --kubeconfig ${KUBECONFIG_FILE}.tmp \
  set-context ${NEW_CONTEXT} --namespace ${NAMESPACE}
# Flatten/minify kubeconfig
kubectl config --kubeconfig ${KUBECONFIG_FILE}.tmp \
  view --flatten --minify > ${KUBECONFIG_FILE}
# Remove tmp
rm ${KUBECONFIG_FILE}.full.tmp
rm ${KUBECONFIG_FILE}.tmp
```

Copy

You should end up with a kubeconfig that can access your Kubernetes cluster with the desired target namespaces.

More Examples

Role example

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

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
```

```
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

ClusterRole example

A ClusterRole can be used to grant the same permissions as a Role. Because ClusterRoles are cluster-scoped, you can also use them to grant access to:

- cluster-scoped resources (like nodes)
- non-resource endpoints (like /healthz)
- namespaced resources (like Pods), across all namespaces

  For example: you can use a ClusterRole to allow a particular user to run kubectl get pods --all-namespaces

Here is an example of a ClusterRole that can be used to grant read access to secrets in any particular namespace, or across all namespaces (depending on how it is bound):

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""]
  #
  # at the HTTP level, the name of the resource for accessing Secret
  # objects is "secrets"
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

The name of a Role or a ClusterRole object must be a valid path segment name.

RoleBinding and ClusterRoleBinding

A role binding grants the permissions defined in a role to a user or set of users. It holds a list of *subjects* (users, groups, or service accounts), and a reference to the role being granted. A RoleBinding grants permissions within a specific namespace whereas a ClusterRoleBinding grants that access cluster-wide.

A RoleBinding may reference any Role in the same namespace. Alternatively, a RoleBinding can reference a ClusterRole and bind that ClusterRole to the namespace of the RoleBinding. If you want to bind a ClusterRole to all the namespaces in your cluster, you use a ClusterRoleBinding.

The name of a RoleBinding or ClusterRoleBinding object must be a valid path segment name.

RoleBinding examples

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

```
apiVersion: rbac.authorization.k8s.io/v1
# This role binding allows "jane" to read pods in the "default" namespace.
# You need to already have a Role named "pod-reader" in that namespace.
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
# You can specify more than one "subject"
- kind: User
  name: jane # "name" is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  # "roleRef" specifies the binding to a Role / ClusterRole
  kind: Role #this must be Role or ClusterRole
  name: pod-reader # this must match the name of the Role or ClusterRole you wish to bind to
  apiGroup: rbac.authorization.k8s.io
```

A RoleBinding can also reference a ClusterRole to grant the permissions defined in that ClusterRole to resources inside the RoleBinding's namespace. This kind of

reference lets you define a set of common roles across your cluster, then reuse them within multiple namespaces.

For instance, even though the following RoleBinding refers to a ClusterRole, "dave" (the subject, case sensitive) will only be able to read Secrets in the "development" namespace, because the RoleBinding's namespace (in its metadata) is "development".

```
apiVersion: rbac.authorization.k8s.io/v1
# This role binding allows "dave" to read secrets in the "development" namespace.
# You need to already have a ClusterRole named "secret-reader".
kind: RoleBinding
metadata:
  name: read-secrets
  #
  # The namespace of the RoleBinding determines where the permissions are granted.
  # This only grants permissions within the "development" namespace.
  namespace: development
subjects:
- kind: User
  name: dave # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

ClusterRoleBinding example

To grant permissions across a whole cluster, you can use a ClusterRoleBinding. The following ClusterRoleBinding allows any user in the group "manager" to read secrets in any namespace.

```
apiVersion: rbac.authorization.k8s.io/v1
# This cluster role binding allows anyone in the "manager" group to read secrets in any namespace.
kind: ClusterRoleBinding
metadata:
  name: read-secrets-global
```

```
subjects:
- kind: Group
  name: manager # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

After you create a binding, you cannot change the Role or ClusterRole that it refers to. If you try to change a binding's roleRef, you get a validation error. If you do want to change the roleRef for a binding, you need to remove the binding object and create a replacement.

There are two reasons for this restriction:

1. Making roleRef immutable allows granting someone update permission on an existing binding object, so that they can manage the list of subjects, without being able to change the role that is granted to those subjects.
2. A binding to a different role is a fundamentally different binding. Requiring a binding to be deleted/recreated in order to change the roleRef ensures the full list of subjects in the binding is intended to be granted the new role (as opposed to enabling or accidentally modifying only the roleRef without verifying all of the existing subjects should be given the new role's permissions).

The kubectl auth reconcile command-line utility creates or updates a manifest file containing RBAC objects, and handles deleting and recreating binding objects if required to change the role they refer to. See command usage and examples for more information.

Referring to resources

In the Kubernetes API, most resources are represented and accessed using a string representation of their object name, such as pods for a Pod. RBAC refers to resources using exactly the same name that appears in the URL for the relevant API endpoint. Some Kubernetes APIs involve a *subresource*, such as the logs for a Pod. A request for a Pod's logs looks like:

```
GET /api/v1/namespaces/{namespace}/pods/{name}/log
```

In this case, pods is the namespace resource for Pod resources, and log is a sub resource of pods. To represent this in an RBAC role, use a slash (/) to delimit the

resource and sub resource. To allow a subject to read pods and also access the log sub resource for each of those Pods, you write:

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

You can also refer to resources by name for certain requests through the resourceNames list. When specified, requests can be restricted to individual instances of a resource. Here is an example that restricts its subject to only get or update a ConfigMap named my-configmap:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: configmap-updater
rules:
- apiGroups: [""]
  #
  # at the HTTP level, the name of the resource for accessing ConfigMap
  # objects is "configmaps"
  resources: ["configmaps"]
  resourceNames: ["my-configmap"]
  verbs: ["update", "get"]
```

Note: You cannot restrict create or deletecollection requests by their resource name. For create, this limitation is because the name of the new object may not be known at authorization time. If you restrict list or watch by resourceName, clients must include a metadata.name field selector in their list or watch request that matches the specified resourceName in order to be authorized. For example, kubectl get configmaps --field-selector=metadata.name=my-configmap