Play with the Sample API Server

Uncover some of the deployment details associated with aggregated apiservers.

We'll cover the following

Overview

Aggregated API servers are usually built with the generic API server library k8s.io/apiserver. They run as stand-alone processes serving API groups with multiple versions. They could run inside or outside of the cluster. In the former case, they run as Pods, with a service in the front. By declaring an `APIService` object, we can hook our aggregated apiservers up to the `kube-apiserver`, which can make them work coherently. In the `APIService` object, we can correlate a service to indicate the accessible endpoint of our aggregated apiservers. If they are running in the cluster, the service could be a normal `ClusterIP` service. If running outside of a Kubernetes cluster, the service could also be an `ExternalName` service.

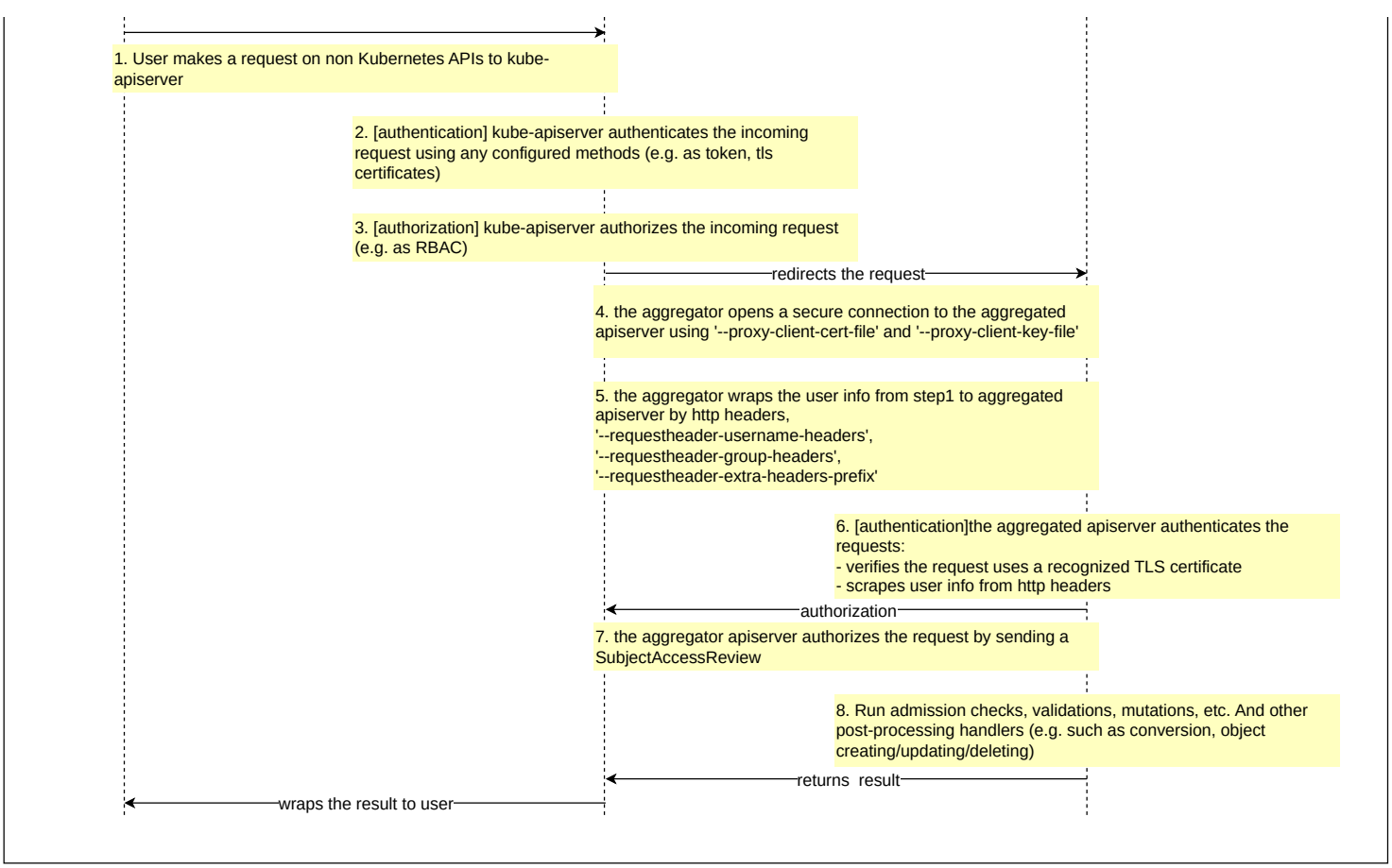
In this lesson, we're going to deploy a sample aggregate apiserver and take a closer look at what aggregated apiservers look like. Seeing is believing. Then we'll know how to implement our own aggregated apiservers.

For the purpose of demonstration, we'll deploy a sample-apiserver, which is built by the Kubernetes community as a reference implementation of an apiserver for a custom Kubernetes API. We could fork this repository, modify it to add our own custom types, and then periodically rebase upstream changes to pick up improvements and bug fixes. We can also choose to use the scaffold framework to generate a scratch repository for solely our custom APIs.

In the demo, we will present various deployment YAMLs to illustrate the configurations.

A retrospect of the working flow

First, let's look at a retrospect of the working flow between the `kube-apiserver` and aggregated apiservers, as shown below:



The workflow between the kube-apiserver and aggregated apiserver

The `kube-apiserver` aggregates all the APIs for serving, including core Kubernetes APIs, CRDs, and aggregated APIs. In the `kube-apiserver`, there is a component called the `kube-aggregator` that acts as a front proxy to automatically forward all the aggregated API requests to aggregated apiservers. Moreover, the `kube-aggregator` is responsible for discovering and registering APIs served by aggregated apiservers. Actually, the `kube-aggregator` is a controller that associates with the apiregistration API and constitutes the aggregation layer.

Authentications and authorizations will be delegated to the `kube-apiserver` to verify the validity and privilege of incoming requests. These auth parts are automatically done by the library `k8s.io/apiserver`.

Thus, the core part of our task is how we configure the aggregator layer to make all that magic work.

Configuring the aggregator

From the workflow above, we can see there are four parts that need to be configured on the `kube-apiserver` side.

1. We need a pair of TLS certificates to identify the `kube-aggregator`. We don't want any clients communicating with aggregated apiservers. The flags for these are `--proxy-client-cert-file` and `--proxy-client-key-file`.

2. We need to know the validity of the TLS certificate pair above, which should be signed by a known CA. Thus, we need a CA file as well. The flag here is `--requestheader-client-ca-file`.
3. Not all the certificates signed by `--requestheader-client-ca-file` are allowed to access aggregated apiservers. In aggregated apiservers, we do need authentication as well. We specify the flag `--requestheader-allowed-names` to only allow client certificates with allowed common names.
4. The `kube-aggregator` forwards the requests by inserting a group of HTTP headers that contain the pre-authenticated user and group information. These flags are configured by a group of flags prefixed with the `requestheader-`, i.e., `--requestheader-extra-headers-prefix`, `--requestheader-group-headers`, and `--requestheader-username-headers`.

deploy-sample-apiserver.sh

Search in directory...

/

deploy-sample-apiserver.sh

artifacts

```
cd /usercode
kubectl taint node --all node-role.kubernetes.io/master-

# create the namespace to run the apiserver in
kubectl create ns wardle

# create the service account used to run the server
kubectl create -f artifacts/example/sa.yaml -n wardle

# create the rolebindings that allow the service account user to c
kubectl create -f artifacts/example/auth-delegator.yaml -n kube-sy
kubectl create -f artifacts/example/auth-reader.yaml -n kube-syste

# create rbac roles and clusterrolebinding that allow the service
kubectl create -f artifacts/example/rbac.yaml
kubectl create -f artifacts/example/rbac-bind.yaml

# create the service and replication controller
kubectl create -f artifacts/example/deployment.yaml -n wardle
kubectl create -f artifacts/example/service.yaml -n wardle

# create the apiservice object that tells kubernetes about your a
kubectl create -f artifacts/example/apiservice.yaml
```

The k8s cluster

Now, let's check these flags by running the command below in the terminal above.

```
ps -ef | grep kube-apiserver
```

Check the parameters of the kube-apiserver

From the output, we see that the aggregator-related flags above are present as follows:

```
kube-apiserver --advertise-address=10.0.3.47 \
    ...
    --client-ca-file=/etc/kubernetes/pki/ca.crt \
    ...
    --proxy-client-cert-file=/etc/kubernetes/pki/front-proxy-client.crt \
    --proxy-client-key-file=/etc/kubernetes/pki/front-proxy-client.key \
    --requestheader-allowed-names=front-proxy-client \
    --requestheader-client-ca-file=/etc/kubernetes/pki/front-proxy-ca.crt \
    --requestheader-extra-headers-prefix=X-Remote-Extra- \
    --requestheader-group-headers=X-Remote-Group \
    --requestheader-username-headers=X-Remote-User \
    ...
```

The parameters for the aggregator layer in the kube-apiserver

The aggregator layer is ready for now. Let's deploy the `sample-apiserver` originally created by the Kubernetes community.

Deploy it and test it out

Let's run the command below in the terminal above to deploy it.

```
./deploy-sample-apiserver.sh
```

Deploy the kube-sample-apiserver

After a while, our sample apiserver will be running in the namespace `wardle`. Let's check the Pod with the following command by running it in the terminal above:

```
kubectl get pod -n wardle
```

Command to list Pods in the namespace wardle

The output will be as follows:

```
NAME                              READY   STATUS    RESTARTS      AGE
wardle-server-798dcdb654-tq2tc   2/2     Running   0             2m1s
```

Our sample apiserver is running

Now, let's check whether the `kube-apiserver` discovers our custom APIs served by sample apiservers. In the terminal above, let's run the following command:

```
kubectl api-resources | grep wardle
```

Check our custom APIs

The output will be as follows:

```
fischers           wardle.example.com/v1alpha1           false        Fischer
flunders           wardle.example.com/v1alpha1           true         Flunder
```

The output of checking our custom APIs

The sample apiservers run pretty well. Everything works as expected. Now, we're able to create the resource type `Flunder` with the command below.

```
1  kubectl create -f /usercode/artifacts/flunders/01-flunder.yaml
```

Create resource of type Flunder

Now, let's use `kubectl` to list the `Flunder` resources with the command below.

```
1  kubectl get flunder -A
```

List our custom resource Flunder in all the namespaces

The output will be as follows:

```
1  NAMESPACE   NAME              CREATED AT
2  default     my-first-flunder  2022-10-31T14:35:03Z
```

Our created custom resource

Ta-da! The sample apiserver works!

Back

How the Aggregated Apiserver Works

Next

How to Generate the Scaffold Aggregated Apiserver