How to Generate the Scaffold Aggregated Apiserver

Learn how to generate a scaffold project for the aggregated apiserver.

We'll cover the following

Overview

Aggregated apiservers are complex and hard to implement from scratch. There are lots of API schema, auto generated codes, golang build tags, deployments files, TLS certificates for communication, delegated auth, etc.

Ways to build custom aggregated apiservers

Normally, there are two ways to build custom aggregated apiservers.

- We could fork the repository `k8s.io/sample-apiserver`, modify it to add our own custom types, and then periodically rebase upstream changes to pick up improvements and bug fixes. This repository is used only for demonstration. However, it contains all the needed files—including source codes, deployment artifacts, and hack scripts—which can be used for codes' auto generating and containers' image building.
- We could use the development kit `apiserver-builder` to help generate a scaffold repository for our custom APIs. The `apiserver-builder` is a collection of libraries and tools, which are built to help users build native Kubernetes extensions using Kubernetes apiserver aggregation.

In this lesson, we will do the second options and create an empty repository using the `apiserver-builder`. During this lesson, we're using the terminal below for developing and testing.

```
11  distributed under the License is distributed on an "AS IS" BASIS,
12  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or i
13  See the License for the specific language governing permissions a
14  limitations under the License.
15  */
16
17  package v1beta1
18
19  import (
20      metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
21      "k8s.io/apimachinery/pkg/runtime"
22      "k8s.io/apimachinery/pkg/runtime/schema"
23  )
24
25
26  var AddToScheme = func(scheme *runtime.Scheme) error {
27      metav1.AddToGroupVersion(scheme, schema.GroupVersion{
28          Group:   "bar.pwk.educative.io",
29          Version: "v1beta1",
30      })
31      // +kubebuilder:scaffold:install
```

The terminal

Hit the “Run” button to initialize it.

Generating the scaffold

First, we initialize our repository with scaffolding directories and go files. We are going to create a domain `pwk.educative.io` to hold all our custom groups and versions of our aggregated apiserver.

We’ll also specify a `--module-name` to specify the module name of our go mod project. Here, we are using `educative.io/pwk`.

Run the following commands in the terminal above:

```
1  mkdir -p /root/go/src/educative.io/pwk
2  cd /root/go/src/educative.io/pwk
3  apiserver-boot init repo --domain pwk.educative.io --module-name educative.io/pwk
```

Initialize our repository

After that, we can get a group of files in the current directory. We can run the command `tree` in the terminal above. The output will be as shown below.

```
1  .
2  ├── Dockerfile
3  ├── Makefile
4  ├── bin
5  ├── cmd
6  │   ├── apiserver
7  │   │   └── main.go
8  │   └── manager
9  │       └── main.go -> ../../main.go
10 ├── go.mod
```

```
├── hack
│   └── boilerplate.go.txt
├── main.go
└── pkg
    └── apis
        └── doc.go

7 directories, 8 files
```

Project tree after initialization

Now, let's start generating real APIs. We're going to create a new kind `Foo` with version `v1beta1` in the group `bar`, and we want subresource `/status` as well, so we can update the object status with the subresource API.

```
apiserver-boot create group version resource --group bar --version v1beta1 --kind Foo --with-status-subr
```

Create the new API Foo with version v1beta1

We can run the command above in the terminal. We also want the toolkit to automatically generate the resource and controller for us. So, we input `y` for next.

After we run the command `tree ./pkg/apis/bar` in the terminal above, we can see that the file `pkg/apis/bar/v1beta1/foo_types.go` is created as shown below—the API scheme is defined and we can insert new fields there.

```
./pkg/apis/bar
├── doc.go
└── v1beta1
    ├── doc.go
    ├── foo_types.go
    └── register.go

1 directory, 4 files
```

The foo_types.go file

When we open the file `pkg/apis/bar/v1beta1/register.go`, we can see the contents below. The `Foo` API is registered with version `v1beta1` in the group `bar.pwk.educative.io`. This is exactly what we want.

```
/*
Copyright 2022.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS
```

```
11  distributed under the License is distributed on an "AS IS" BASIS,
12  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
13  See the License for the specific language governing permissions and
14  limitations under the License.
15  */
16
17  package v1beta1
18
19  import (
20      metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
21      "k8s.io/apimachinery/pkg/runtime"
22      "k8s.io/apimachinery/pkg/runtime/schema"
23  )
24
25
26  var AddToScheme = func(scheme *runtime.Scheme) error {
27      metav1.AddToGroupVersion(scheme, schema.GroupVersion{
28          Group:   "bar.pwk.educative.io",
29          Version: "v1beta1",
30      })
31      // +kubebuilder:scaffold:install
```

The register.go file

Now, let's generate code containing the `DeepCopy`, `DeepCopyInto`, and `DeepCopyObject` method implementations with the magic commands below. Run the commands in the terminal above.

```
1  go install sigs.k8s.io/controller-tools/cmd/controller-gen@v0.8.0
2  cp /root/go/bin/controller-gen ./bin
3  go mod tidy
4  make generate
5  go mod tidy
```

Generate codes

After running the commands successfully, we'll see a file `zz_generated.deepcopy.go` created in the folder `pkg/apis/bar/v1beta1/`. We can list the directory by running the command below in the terminal above.

```
1  ls pkg/apis/bar/v1beta1/
```

Command to list files in a directory

The output will be as follows:

```
1  doc.go  foo_types.go  register.go  zz_generated.deepcopy.go
```

Files in the directory

The container image can be easily built with the command below.

```
1  apiserver-boot build container --image educative.io/pwk/apiserver:demo
```

Build container image

Moreover, all the deployment artifacts can be automatically generated by running the command below. Awesome isn't it?

```
apiserver-boot build config --name pwk-aa-demo --namespace pwk-system --image educative.io/pwk/apiserver
```

Create deployment artifacts

After that, we can see that a new folder `config` is generated with all the deployment artifacts. We can run the following command in the terminal above to view all those generated files:

```
tree ./config/
```

Command to list contents in the config directory

The output will be as follows:

```
./config/
├── aggregated-apiserver.yaml
├── apiservice.yaml
├── certificates
│   ├── apiserver.crt
│   ├── apiserver.key
│   ├── apiserver_ca.crt
│   └── apiserver_ca.key
├── controller-manager.yaml
├── etcd.yaml
└── rbac.yaml

1 directory, 9 file
```

All the files in the config directory

Now that we've got all the manifests and the container image, shipping our apiserver is quite easy. We just need to run `kubectl apply -f config` against our Kubernetes cluster.

Back

Play with the Sample API Server

Next

Implement an Aggregated Apiserver for Simple Reso...