Implement an Aggregated Apiserver for Simple Resources

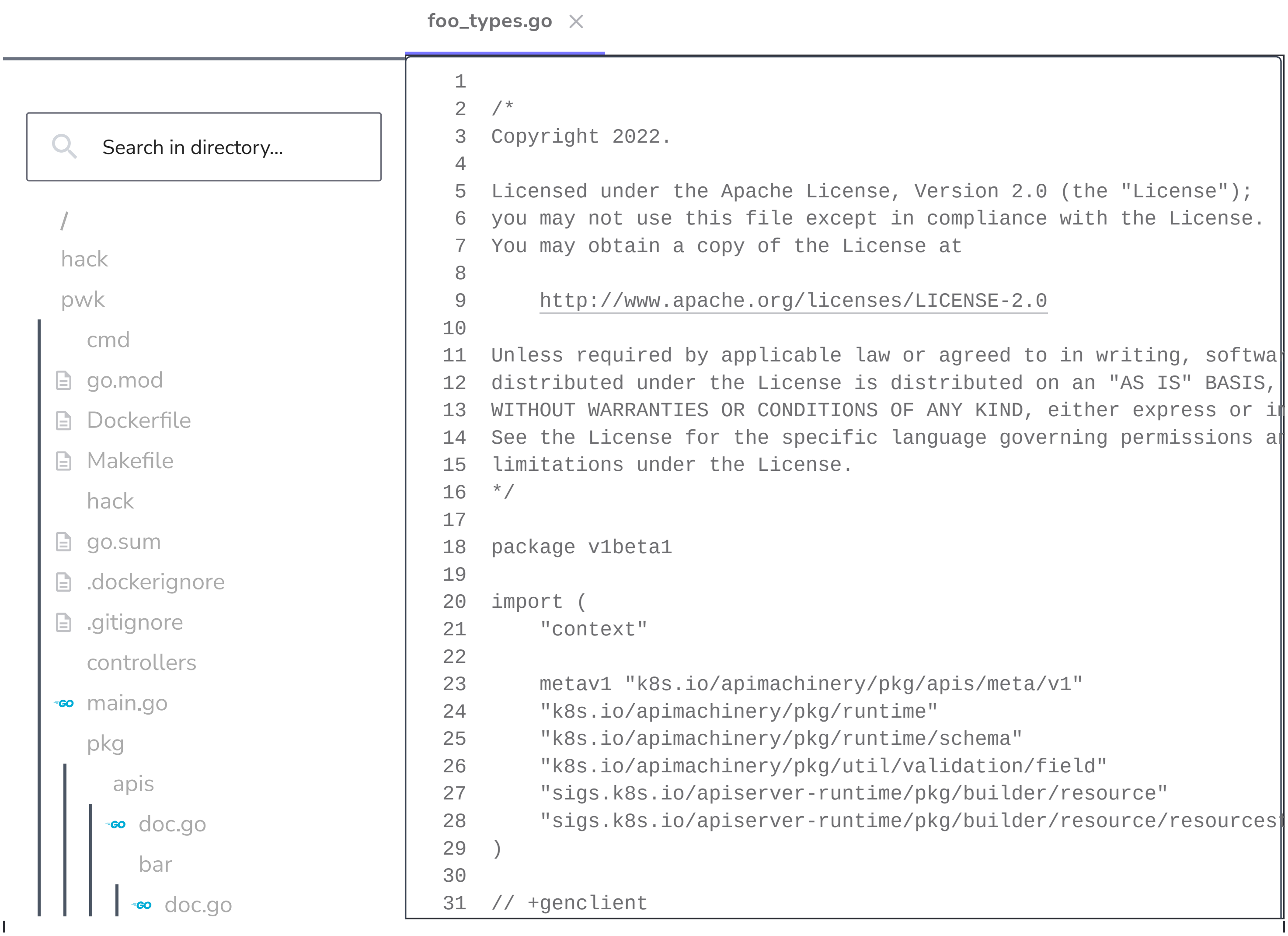
Learn how to build an aggregated apiserver handling simple resources.

We'll cover the following

- Overview
- Add new fields
- Modify controller
- Build and deploy
- Test it out

Overview

Building an aggregated apiserver from scratch is hard. Luckily, we've got a development kit `apiserver-boot`, which is shipped by the Kubernetes SIGs community. With this tool, we can easily build our own scaffold project. Below is the scaffold project `pwk` that we built using the `apiserver-boot`.



```

/*
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, softwa
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or i
See the License for the specific language governing permissions a
limitations under the License.
*/

package v1beta1

import (
    "context"

    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/apimachinery/pkg/runtime"
    "k8s.io/apimachinery/pkg/runtime/schema"
    "k8s.io/apimachinery/pkg/util/validation/field"
    "sigs.k8s.io/apiserver-runtime/pkg/builder/resource"
    "sigs.k8s.io/apiserver-runtime/pkg/builder/resource/resources
)

// +genclient
```

The scaffold project

Now, let’s implement our own aggregated apiserver building on the top of this scaffold project.

In this lesson, we’re going to define a new API kind `Foo` served by the aggregated apiserver, and we will have a controller to reconcile the `Foo` kind objects.

To get started, hit the “Run” button to initialize our development environment in the terminal above.

Add new fields

We add new fields into the `Foo` struct, as shown below. The modified file is placed at `hack/foo_types.go`. In `FooSpec` (**lines 32–34**), we add an integer to indicate `Replicas` and a string for `Location`. We then add JSON struct tags for serialization and deserialization. We can also add validation rules to validate the fields. Here, `Replicas` are set to be in the range from 0 to 100 (**lines 50–59**).

```
 1  diff --git a/pkg/apis/bar/v1beta1/foo_types.go b/pkg/apis/bar/v1beta1/foo_types.go
 2  index ff298ed..494bab8 100644
 3  --- a/pkg/apis/bar/v1beta1/foo_types.go
 4  +++ b/pkg/apis/bar/v1beta1/foo_types.go
 5  @@ -1,4 +1,3 @@
 6  -
 7   /*
 8   Copyright 2022.
 9   
10  @@ -21,7 +20,7 @@ import (
11          "context"
12   
13          metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
14  -       "k8s.io/apimachinery/pkg/runtime"
15  +       "k8s.io/apimachinery/pkg/runtime"
16          "k8s.io/apimachinery/pkg/runtime/schema"
17          "k8s.io/apimachinery/pkg/util/validation/field"
18          "sigs.k8s.io/apiserver-runtime/pkg/builder/resource"
19  @@ -44,7 +43,7 @@ type Foo struct {
20   // FooList
21   // +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object
22   type FooList struct {
23  -       metav1.TypeMeta   `json:",inline"`
24  +       metav1.TypeMeta `json:",inline"`
25          metav1.ListMeta `json:"metadata,omitempty"`
26   
27          Items []Foo `json:"items"`
28  @@ -52,6 +51,9 @@ type FooList struct {
29   
30   // FooSpec defines the desired state of Foo
31   type FooSpec struct {
```

Add new fields to foo_types.go

We’ve added another interface `func (in *Foo) Default() {}` (**lines 42–45**), where we can apply our defaulting rules as well.

Modify controller

Now, let’s add a controller to reconcile `Foo` objects.

```
 1  diff --git a/controllers/bar/foo_controller.go b/controllers/bar/foo_controller.go
 2  index 7628984..bef1bee 100644
 3  --- a/controllers/bar/foo_controller.go
 4  +++ b/controllers/bar/foo_controller.go
 5  @@ -19,10 +19,11 @@ package bar
 6   import (
 7          "context"
 8  
 9  +       "k8s.io/apimachinery/pkg/api/errors"
10          "k8s.io/apimachinery/pkg/runtime"
11          ctrl "sigs.k8s.io/controller-runtime"
12          "sigs.k8s.io/controller-runtime/pkg/client"
13  -       "sigs.k8s.io/controller-runtime/pkg/log"
14  +       "sigs.k8s.io/controller-runtime/pkg/reconcile"
15  
16          barv1beta1 "educative.io/pwk/pkg/apis/bar/v1beta1"
17   )
18  @@ -47,11 +48,26 @@ type FooReconciler struct {
19   // For more details, check Reconcile and its Result here:
20   // - https://pkg.go.dev/sigs.k8s.io/controller-runtime@v0.11.0/pkg/reconcile
21   func (r *FooReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
22  -       _ = log.FromContext(ctx)
23  +       // Fetch the Foo instance
24  +       instance := &barv1beta1.Foo{}
25  +       err := r.Get(ctx, req.NamespacedName, instance)
26  +       if err != nil {
27  +               if errors.IsNotFound(err) {
28  +                       // Object not found, return.  Created objects are automatically garbage collecte
29  +                       // For additional cleanup logic use finalizers.
30  +                       return reconcile.Result{}, nil
31  +               }
```

Reconcile Foo objects

In this demo, we reconcile all the `Foo` objects and update `spec.location` to `Los Angeles` (**lines 22–44**). This is simple controller logic, but we can also add more complex logic. The modified file is placed at `hack/foo_controller.go`.

Build and deploy

First, let’s place our modified changes, which are already stored in the folder `/usercode/hack`.

```
cp /usercode/hack/foo_types.go /usercode/pwk/pkg/apis/bar/v1beta1/foo_types.go
cp /usercode/hack/foo_controller.go /usercode/pwk/controllers/bar/foo_controller.go
```

Place our changes

Next, we need to build container images for our aggregated apiserver and controller. To reduce the image building time, we’ve built out an image called `dixudx/pwk:aa-simple`. If you want to do that on your own, you can run the following commands to build out an image:

```
cd /usercode/pwk
```

```
cp -rf /root/go/bin /usercode/pwk/bin
go mod tidy
make generate
apiserver-boot build container --image dixudx/pwk:aa-simple
```

Build out the image

Finally, we’re going to ship our apiserver and controller. All the deployment artifacts can be automatically generated as well. We can run the following command in the terminal above:

```
cd /usercode/pwk/
apiserver-boot build config --name pwk-aa-demo --namespace default --image dixudx/pwk:aa-simple
cp /usercode/hack/etcd.yaml /usercode/pwk/config/etcd.yaml
```

Generate deployment artifacts

Now, we can deploy all these artifacts to Kubernetes by simply running the command below in the terminal above.

```
kubectl taint node --all node-role.kubernetes.io/master-
kubectl apply -f /usercode/pwk/config
```

Deploy to Kubernetes cluster

Test it out

It’s time to check our aggregated apiserver.

First, let’s check whether our `Foo` API can be discovered in the `kube-apiserver`. We can run the following command in the terminal above to do this:

```
kubectl api-resources | grep educative.io
```

Command to check our API

The output will be as follows:

```
foos           bar.pwk.educative.io/v1beta1           false        Foo
```

Our self-hosted API has been registered successfully

It’s working!

Next, we create a `Foo` object as shown below. This file is placed at `hack/foo.yaml`. In the file, we specify `spec.replicas` to `101`, which is larger than 100. In the validation rules above, we set a rule to only allow this field to be set as a value between 0 and 100.

```
apiVersion: bar.pwk.educative.io/v1beta1
kind: Foo
metadata:
  name: demo-foo
  name: demo-foo
  namespace: default
spec:
  replicas: 101
```

The YAML file for demo-foo

Run the following command in the terminal above, and let's see what happens:

```
kubectl apply -f /usercode/hack/foo.yaml
```

Command to create demo-foo the first time

The output will be as follows:

```
The Foo "demo-foo" is invalid: spec.replicas: Invalid value: 101: replicas must be < 100
```

The output for creating demo-foo the first time

The validation rules work. This request is rejected due to the invalid `Replicas` setting. This is what we expect. Now, we can modify this value in the file `hack/foo.yaml` to `100`, and reapply the file again.

```
kubectl apply -f foo.yml
```

Command to create modified demo-foo

The output will be as follows:

```
foo.bar.pwk.educative.io/demo-foo created
```

The output of creating modified demo-foo

We created it successfully! Now, let's `get` this object to see what's in it.

```
kubectl get foo demo-foo -o yaml
```

Command to get the object demo-foo from the kube-apiserver

The output will be as follows:

```
apiVersion: bar.pwk.educative.io/v1beta1
kind: Foo
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"bar.pwk.educative.io/v1beta1","kind":"Foo","metadata":{"annotations":{},"name":"der
  creationTimestamp: "2022-11-02T12:14:49Z"
  name: demo-foo
  resourceVersion: "9"
  uid: 8fa3f6ad-f5b1-42f4-b943-6273b0a463d0
spec:
  location: Los Angeles
  replicas: 100
status: {}
```

Object demo-foo stored in the kube-apiserver

From the above content, we can see that the `spec.location` is updated to `Los Angeles`, which is exactly reconciled by the controller that we implement.

Ta-da! It’s done.

← Back

How to Generate the Scaffold Aggregated Apiserver

☑

Next →

Implement an Aggregated Apiserver Using Local File...