sample-apiserver

Demonstration of how to use the k8s.io/apiserver library to build a functional API server.

Note: go-get or vendor this package as `k8s.io/sample-apiserver`.

Purpose

You may use this code if you want to build an Extension API Server to use with API Aggregation, or to build a stand-alone Kubernetes-style API server.

However, consider two other options:

- **CRDs**: if you just want to add a resource to your kubernetes cluster, then consider using Custom Resource Definition a.k.a CRDs. They require less coding and rebasing. Read about the differences between Custom Resource Definitions vs Extension API Servers here.
- **Apiserver-builder**: If you want to build an Extension API server, consider using apiserver-builder instead of this repo. The Apiserver-builder is a complete framework for generating the apiserver, client libraries, and the installation program.

If you do decide to use this repository, then the recommended pattern is to fork this repository, modify it to add your types, and then periodically rebase your changes on top of this repo, to pick up improvements and bug fixes to the apiserver.

Compatibility

HEAD of this repo will match HEAD of k8s.io/apiserver, k8s.io/apimachinery, and k8s.io/client-go.

Where does it come from?

`sample-apiserver` is synced from https://github.com/kubernetes/kubernetes/blob/master/staging/src/k8s.io/sample-apiserver. Code changes are made in that location, merged into `k8s.io/kubernetes` and later synced here.

Fetch sample-apiserver and its dependencies

Like the rest of Kubernetes, sample-apiserver has used godep and `$GOPATH` for years and is now adopting go 1.11 modules. There are thus two alternative ways to go about fetching this demo and its dependencies.

Fetch with godep

When NOT using go 1.11 modules, you can use the following commands.

```
go get -d k8s.io/sample-apiserver
cd $GOPATH/src/k8s.io/sample-apiserver  # assuming your GOPATH has just one entry
godep restore
```

When using go 1.11 modules

When using go 1.11 modules (`GO111MODULE=on`), issue the following commands --- starting in whatever working directory you like.

```
git clone https://github.com/kubernetes/sample-apiserver.git
cd sample-apiserver
```

Note, however, that if you intend to generate code then you will also need the code-generator repo to exist in an old-style location. One easy way to do this is to use the command `go mod vendor` to create and populate the `vendor` directory.

A Note on kubernetes/kubernetes

If you are developing Kubernetes according to https://github.com/kubernetes/community/blob/master/contributors/guide/github-workflow.md then you already have a copy of this demo in `kubernetes/staging/src/k8s.io/sample-apiserver` and its dependencies --- including the code generator --- are in usable locations.

Normal Build and Deploy

Changes to the Types

If you change the API object type definitions in any of the `pkg/apis/.../types.go` files then you will need to update the files generated from the type definitions. To do this, first create the vendor directory if necessary and then invoke `hack/update-codegen.sh` with `sample-apiserver` as your current working directory; the script takes no arguments.

Authentication plugins

The normal build supports only a very spare selection of authentication methods. There is a much larger set available in https://github.com/kubernetes/client-go/tree/master/plugin/pkg/client/auth . If you want your server to support one of those, such as `oidc` , then add an import of the appropriate package to `sample-apiserver/main.go` . Here is an example:

```
import _ "k8s.io/client-go/plugin/pkg/client/auth/oidc"
```

Alternatively you could add support for all of them, with an import like this:

```
import _ "k8s.io/client-go/plugin/pkg/client/auth"
```

Build the Binary

With `sample-apiserver` as your current working directory, issue the following command:

```
CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -a -o artifacts/simple-image/kube-
sample-apiserver
```

Build the Container Image

With `sample-apiserver` as your current working directory, issue the following commands with `MYPREFIX` and `MYTAG` replaced by something suitable.

```
docker build -t MYPREFIX/kube-sample-apiserver:MYTAG ./artifacts/simple-image
docker push MYPREFIX/kube-sample-apiserver:MYTAG
```

Deploy into a Kubernetes Cluster

Edit `artifacts/example/deployment.yaml`, updating the pod template's image reference to match what you pushed and setting the `imagePullPolicy` to something suitable. Then call:

```
kubectl apply -f artifacts/example
```

Running it stand-alone

During development it is helpful to run sample-apiserver stand-alone, i.e. without a Kubernetes API server for authn/authz and without aggregation. This is possible, but needs a couple of flags, keys and certs as described below. You will still need some kubeconfig, e.g. `~/.kube/config`, but the Kubernetes cluster is not used for authn/z. A minikube or hack/local-up-cluster.sh cluster will work.

Instead of trusting the aggregator inside kube-apiserver, the described setup uses local client certificate based X.509 authentication and authorization. This means that the client certificate is trusted by a CA and the passed certificate contains the group membership to the `system:masters` group. As we disable delegated authorization with `--authorization-skip-lookup`, only this superuser group is authorized.

1. First we need a CA to later sign the client certificate:

   ```
   openssl req -nodes -new -x509 -keyout ca.key -out ca.crt
   ```

2. Then we create a client cert signed by this CA for the user `development` in the superuser group `system:masters`:

   ```
   openssl req -out client.csr -new -newkey rsa:4096 -nodes -keyout client.key -
   subj "/CN=development/O=system:masters"
   openssl x509 -req -days 365 -in client.csr -CA ca.crt -CAkey ca.key -
   set_serial 01 -out client.crt
   ```

3. As curl requires client certificates in p12 format with password, do the conversion:

   ```
   openssl pkcs12 -export -in ./client.crt -inkey ./client.key -out client.p12 -
   passout pass:password
   ```

4. With these keys and certs in-place, we start the server:

   ```
   etcd &
   sample-apiserver --secure-port 8443 --etcd-servers http://127.0.0.1:2379 --
   v=7 \
      --client-ca-file ca.crt \
      --kubeconfig ~/.kube/config \
      --authentication-kubeconfig ~/.kube/config \
      --authorization-kubeconfig ~/.kube/config
   ```

   The first kubeconfig is used for the shared informers to access Kubernetes resources. The second kubeconfig passed to `--authentication-kubeconfig` is used to satisfy the delegated authenticator.

The third kubeconfig passed to `--authorized-kubeconfig` is used to satisfy the delegated authorizer. Neither the authenticator, nor the authorizer will actually be used: due to `--client-ca-file`, our development X.509 certificate is accepted and authenticates us as `system:masters` member. `system:masters` is the superuser group such that delegated authorization is skipped.

5. Use curl to access the server using the client certificate in p12 format for authentication:

```
curl -fv -k --cert-type P12 --cert client.p12:password \

https://localhost:8443/apis/wardle.example.com/v1alpha1/namespaces/default/flunc
```

Or use wget:

```
wget -O- --no-check-certificate \
   --certificate client.crt --private-key client.key \

https://localhost:8443/apis/wardle.example.com/v1alpha1/namespaces/default/flunc
```

Note: Recent OSX versions broke client certs with curl. On Mac try `brew install httpie` and then:

```
http --verify=no --cert client.crt --cert-key client.key \

https://localhost:8443/apis/wardle.example.com/v1alpha1/namespaces/default/flunc
```