sample-controller

This repository implements a simple controller for watching Foo resources as defined with a CustomResourceDefinition (CRD).

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

This particular example demonstrates how to perform basic operations such as:

- How to register a new custom resource (custom resource type) of type `Foo` using a CustomResourceDefinition.
- How to create/get/list instances of your new resource type `Foo`.
- How to setup a controller on resource handling create/update/delete events.

It makes use of the generators in k8s.io/code-generator to generate a typed client, informers, listers and deep-copy functions. You can do this yourself using the `./hack/update-codegen.sh` script.

The `update-codegen` script will automatically generate the following files & directories:

- `pkg/apis/samplecontroller/v1alpha1/zz_generated.deepcopy.go`
- `pkg/generated/`

Changes should not be made to these files manually, and when creating your own controller based off of this implementation you should not copy these files and instead run the `update-codegen` script to generate your own.

Details

The sample controller uses client-go library extensively. The details of interaction points of the sample controller with various mechanisms from this library are explained here.

Fetch sample-controller and its dependencies

Like the rest of Kubernetes, sample-controller 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-controller
cd $GOPATH/src/k8s.io/sample-controller
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-controller.git
cd sample-controller
```

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/contribut workflow.md then you already have a copy of this demo in `kubernetes/staging/src/k8s.io/sample-controll` and its dependencies — including the code generator — are in usable locations (valid for all Go versions).

Purpose

This is an example of how to build a kube-like controller with a single type.

Running

Prerequisite: Since the sample-controller uses `apps/v1` deployments, the Kubernetes cluster version should be greater than 1.9.

```
# assumes you have a working kubeconfig, not required if operating in-cluster
go build -o sample-controller .
./sample-controller -kubeconfig=$HOME/.kube/config

# create a CustomResourceDefinition
kubectl create -f artifacts/examples/crd-status-subresource.yaml

# create a custom resource of type Foo
kubectl create -f artifacts/examples/example-foo.yaml

# check deployments created through the custom resource
kubectl get deployments
```

Use Cases

CustomResourceDefinitions can be used to implement custom resource types for your Kubernetes cluster. These act like most other Resources in Kubernetes, and may be `kubectl apply`'d, etc.

Some example use cases:

- Provisioning/Management of external datastores/databases (eg. CloudSQL/RDS instances)
- Higher level abstractions around Kubernetes primitives (eg. a single Resource to define an etcd cluster, backed by a Service and a ReplicationController)

Defining types

Each instance of your custom resource has an attached Spec, which should be defined via a `struct{}` to provide data format validation. In practice, this Spec is arbitrary key-value data that specifies the configuration/behavior of your Resource.

For example, if you were implementing a custom resource for a Database, you might provide a DatabaseSpec like the following:

```
type DatabaseSpec struct {
    Databases []string `json:"databases"`
    Users     []User   `json:"users"`
    Version   string   `json:"version"`
}

type User struct {
    Name     string `json:"name"`
    Password string `json:"password"`
}
```

Validation

To validate custom resources, use the `CustomResourceValidation` feature. Validation in the form of a structured schema is mandatory to be provided for `apiextensions.k8s.io/v1`.

Example

The schema in `crd.yaml` applies the following validation on the custom resource: `spec.replicas` must be an integer and must have a minimum value of 1 and a maximum value of 10.

Subresources

Custom Resources support `/status` and `/scale` subresources. The `CustomResourceSubresources` feature is in GA from v1.16.

Example

The CRD in `crd-status-subresource.yaml` enables the `/status` subresource for custom resources. This means that `UpdateStatus` can be used by the

controller to update only the status part of the custom resource.

To understand why only the status part of the custom resource should be updated, please refer to the Kubernetes API conventions.

In the above steps, use `crd-status-subresource.yaml` to create the CRD:

```
# create a CustomResourceDefinition supporting the status subresource
kubectl create -f artifacts/examples/crd-status-subresource.yaml
```

A Note on the API version

The group version of the custom resource in `crd.yaml` is `v1alpha`, this can be evolved to a stable API version, `v1`, using CRD Versioning.

Cleanup

You can clean up the created CustomResourceDefinition with:

```
kubectl delete crd foos.samplecontroller.k8s.io
```

Compatibility

HEAD of this repository will match HEAD of k8s.io/apimachinery and k8s.io/client-go.

Where does it come from?

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