This lab shows you how to use [Operator SDK](https://github.com/operator-framework/operator-sdk) to scaffold a basic Go operator. In this lab, learn how to create a new project and add to the API by creating a Custom Resource Definition (CRD) and a basic controller. You will add fields to the CRD to contain some information about the desired and actual state, modify the controller to reconcile instances of your new resource, and then deploy your operator to your Kubernetes cluster.

## Prerequisites

For this lab, you need to have:

* [Operator-sdk](https://sdk.operatorframework.io/docs/installation/?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMDeveloperSkillsNetworkCO0201ENSkillsNetwork23008840-2021-01-01) v1.5.0+ installed
* [Kubectl](https://kubernetes.io/docs/tasks/tools/?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMDeveloperSkillsNetworkCO0201ENSkillsNetwork23008840-2021-01-01#kubectl) v1.17.0+ installed
* Admin access to a Kubernetes cluster. See earlier in the course for instructions on how to deploy a small cluster for free on IBM Cloud.
* [Docker](https://docs.docker.com/get-docker/?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMDeveloperSkillsNetworkCO0201ENSkillsNetwork23008840-2021-01-01) v3.2.2+ installed
* Access to a Docker image repository such as [Docker Hub](https://hub.docker.com/?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMDeveloperSkillsNetworkCO0201ENSkillsNetwork23008840-2021-01-01) or [quay.io](https://quay.io/)
* [Golang](https://golang.org/doc/install?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMDeveloperSkillsNetworkCO0201ENSkillsNetwork23008840-2021-01-01) v1.16.0+ installed

## Step 1: Create a project

Make a new directory, and init a project in it:

$ mkdir memcached-operator

$ cd memcached-operator

$ operator-sdk init --domain=example.com --repo=github.com/example/memcached-operator

Writing scaffold **for** you to edit...

Get controller runtime:

$ go get sigs.k8s.io/controller-runtime@v0.7.0

Update go.mod:

$ go mod tidy

Running make:

$ make

go: creating new go.mod: module tmp

Downloading sigs.k8s.io/controller-tools/cmd/controller-gen@v0.4.1

go: found sigs.k8s.io/controller-tools/cmd/controller-gen **in** sigs.k8s.io/controller-tools v0.4.1

/Users/username/workspace/projects/memcached-operator/bin/controller-gen object:headerFile="hack/boilerplate.go.txt" paths="./..."

go fmt ./...

go vet ./...

go build -o bin/manager main.go

--domain is used for any API groups the operator creates, so yours will be \*.example.com. An API group is a group of related functionality, such as an operator. An API group you might be familiar with is rbac.authorization.k8s.io, which is where the functionality for creating RBAC resources such as ClusterRoles and ClusterBindings is usually set up on a Kubernetes cluster. operator-sdk allows you to specify a custom domain that you can append to any API groups you define to help avoid name collisions.

The value of --repo used here is only an example. You can be set it to a Git repo you have access to if you want to commit your project and keep it.

Let's take a look at what was generated:

* a Dockerfile for the controller image, which will be built out of your project
* a Makefile to build, test, deploy, and undeploy the project
* a PROJECT file with the domain and project layout configuration
* a bin directory that contains the binaries for the controller
* a config directory that contains:
  + base manifests for deploying the CRD and controller of the operator
  + Kustomization YAML for customizing manifests
  + RBAC YAML to authorize the various components to interact with each other
  + a Patch file for enabling Prometheus metrics
  + some sample YAML for creating a simple instance of our new type
* go.mod and go.sum files for managing the Golang depedences of the project
* a hack directory containing a boilerplate license file
* main.go, which contains the main function for your controller

Note that you may need to edit the Makefile if you are on a system that does not use bash as the default shell. See the [Debugging section](https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-CO0201EN-SkillsNetwork/labs/module_1/lab_2/content.md.html#debugging) for more details.

The controller I show you how to build uses the controller-manager framework from [controller-runtime](https://github.com/kubernetes-sigs/controller-runtime). The framework bootstraps a web server that hosts a collection of plugins, each one representing a Kubernetes controller. However, your main.go file is currently mostly empty as you haven't defined a controller yet.

## Step 2: Create an API

Use the create command to generate a CRD and a controller:

Note: The --version flag is for the [Kubernetes API version](https://kubernetes.io/docs/concepts/overview/kubernetes-api/?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMDeveloperSkillsNetworkCO0201ENSkillsNetwork23008840-2021-01-01#api-groups-and-versioning) of the operator, not a [semantic version](https://ssemver.org/). As such, do NOT use . for --version.

$ operator-sdk create api --group cache --version v1alpha1 --kind Memcached --resource=true --controller=true

Writing scaffold **for** you to edit...

api/v1alpha1/memcached\_types.go

controllers/memcached\_controller.go

Running make:

$ make

/Users/username/workspace/projects/memcached-operator/bin/controller-gen object:headerFile="hack/boilerplate.go.txt" paths="./..."

go fmt ./...

go vet ./...

go build -o bin/manager main.go

--group is the group where your custom resource is going to live, so it will end up in the API group 'cache.example.com'.

--version determines the version of your API group. You can use different versions to successively upgrade your custom resource.

--resource and --controller flags are set to true to generate the scaffolding for both those things.

Let's take a look at what changed:

* PROJECT was updated to include the new scaffolding
* go.mod was updated with new depdencies
* main.go was updated to add the new controller to the controller-manager loop

Some of the new generated artifacts are:

* an API directory that contains the files defining your CRD
* YAML manifests in config/ for deploying your CRD
* YAML manifests in config/ for creating RBAC to allow your controller to watch/edit your Memcached type
* YAML manifests in config/samples/ for creating a sample instance of your Memcached type
* a controllers directory, that contains the code for your Memcached controller

Now that we have the outline of our components, let's start to fill them in with actual functionality. First up is the CRD. In api/v1alpha1/memcached\_types.go, you should see some structs that define the spec and status for your Custom Resource type:

// EDIT THIS FILE! THIS IS SCAFFOLDING FOR YOU TO OWN!

// NOTE: json tags are required. Any new fields you add must have json tags **for** the fields **to** be serialized.

// MemcachedSpec defines the desired **state** of Memcached

type MemcachedSpec struct {

// INSERT ADDITIONAL SPEC FIELDS - desired **state** of cluster

// Important: Run "make" **to** regenerate code after modifying this file

// Foo is an example field of Memcached. Edit Memcached\_types.go **to** remove/update

Foo string `json:"foo,omitempty"`

}

// MemcachedStatus defines the observed **state** of Memcached

type MemcachedStatus struct {

// INSERT ADDITIONAL STATUS FIELD - define observed **state** of cluster

// Important: Run "make" **to** regenerate code after modifying this file

}

Note the message at the top of the file. While this file was scaffolded by Operator-SDK, it's yours to use. The Spec contains the information designating the desired state of the resource, while the Status contains the observable state of the system, particularly information that other resources might want to consume. These Golang structs correspond directly with the YAML a Kubernetes user will write to create instances of your custom resource type.

Let's add some basic fields to your type.

// MemcachedSpec defines the desired state of Memcached

**type** MemcachedSpec **struct** {

// +kubebuilder:validation:Minimum=0

// Size is the size of the memcached deployment

Size **int32** `json:"size"`

}

// MemcachedStatus defines the observed state of Memcached

**type** MemcachedStatus **struct** {

// Nodes are the names of the memcached pods

Nodes []**string** `json:"nodes"`

}

Size is an integer that is used to determine the number of nodes in the Memcached cluster. It is marked with a [Kubebuilder validation marker](https://book.kubebuilder.io/reference/markers/crd-validation.html?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMDeveloperSkillsNetworkCO0201ENSkillsNetwork23008840-2021-01-01" \t "_blank) that specifies it has a minimum of 0. To the status we've added an array of strings that will store the IP addresses of the nodes contained in the cluster. Note that this specific implementation is used as an example. In an actual implementation, you should use a [Condition](https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md#typical-status-properties) for this feature.

Note the [Kubebuilder subresource marker](https://book.kubebuilder.io/reference/generating-crd.html?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMDeveloperSkillsNetworkCO0201ENSkillsNetwork23008840-2021-01-01" \l "status" \t "_blank) of the Status field of the parent Memcached struct. This adds a [Kubernetes status subresource](https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/#status-subresource) to the CRD manifest that is generated. This allows your controller to update just the Status field without having to update the entire object, increasing performance.

// Memcached is the Schema for the memcacheds API

// +kubebuilder:subresource:status

type Memcached struct {

metav1.TypeMeta `json:",inline"`

metav1.ObjectMeta `json:"metadata,omitempty"`

Spec MemcachedSpec `json:"spec,omitempty"`

Status MemcachedStatus `json:"status,omitempty"`

}

After making any changes to your types.go files, you should always run the following command from the root directory of the project:

$ make generate

This make target invokes [controller-gen](https://github.com/kubernetes-sigs/controller-tools) to update api/v1alpha1/zz\_generated.deepcopy.go to contain the necessary implementations for the fields you just added. Once that is done, you should run the following command to generate a YAML manifests for your CRD:

$ make manifests

Let's take a look at what was created from that:

New:

config/crd/bases/cache.example.com\_memcacheds.yaml

config/rbac/role.yaml

config/crd/bases/cache.example.com\_memcacheds.yaml is the manifests for your memcached CRD. config/rbac/role.yaml is an RBAC manifest that contains the permissions to manage your memcached type that is needed by the controller.

## Step 3: Create a controller

Let's take a look at what is included so far in controllers/memcached\_controller.go:

// Reconcile is part of the main kubernetes reconciliation loop which aims **to**

// move the current **state** of the cluster closer **to** the desired **state**.

// TODO(**user**): Modify the Reconcile function **to** compare the **state** specified by

// the Memcached object against the actual cluster **state**, and then

// perform operations **to** make the cluster **state** reflect the **state** specified by

// the **user**.

//// For more details, check Reconcile and its Result here:

// - https://pkg.go.dev/sigs.k8s.io/controller-runtime@v0.7.0/pkg/reconcile

func (r \*MemcachedReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {

\_ = r.Log.WithValues("memcached", req.NamespacedName)

// your logic here

return ctrl.Result{}, nil

}

The Reconcile method is responsible for reconciling the desired state contained in the custom resource's status with the actual state running on the system and is where the lion's share of the controller logic is implemented. The specifics of implementing the reconcile loop is beyond this tutorial, and will be covered in the advanced module. For now, replace controllers/memcached\_controller.go with [this reference implementation.](https://github.com/operator-framework/operator-sdk/blob/master/testdata/go/v3/memcached-operator/controllers/memcached_controller.go) Note that you may have to change the import path for github.com/example/memcached-operator/api/v1alpha1 to correctly point to the directory you defined your memcached\_types.go in if you chose to specify a different repo when you initialized your project. After pasting that in, be sure to regenerate the manifests:

$ make manifests

## Step 4: Build and deploy your operator

Now that you've filled in all the needed components, it's time to deploy your operator! In general, there are three different ways to deploy an operator:

1. As a program executing outside the Kubernetes cluster. This might be done for development purposes or for security concerns of the data contained in the cluster. The Makefile contains the target make install run to run the operator locally, but this method is not the focus here.
2. Run as a Deployment inside the Kubernetes cluster. This is what I will show you now.
3. Deployed and managed by Operator Lifecycle Manager. This is recommended for production use because OLM has additional features for the management and upgrade of a running opeartor.

For now, start by building and pushing the Docker image for your controller. This example uses the base Dockerfile, but you can modify it depending on your needs. I use Docker Hub as my image repository, but any repo you have push/pull access to should work.

$ export USERNAME=<docker-username>

$ make docker-build docker-push IMG=docker.io/$USERNAME/memcached-operator:v1.0.0

If you see an error such as the following, you may need to go get additional dependencies. Run the suggested command to download the dependency.

/controllers: package k8s.io/api/apps/v1 imported from implicitly required module; to add missing requirements, run:

go get k8s.io/api/apps/v1@v0.19.2

Also, note that you can set the default name and tag of the image in your Makefile. Once the image is finished building, you're ready to deploy the operator:

$ make deploy IMG=docker.io/$USERNAME/memcached-operator:v1.0.0

This uses the manifests in config/ to create your CRD, deploy your controller in a Pod, create the RBAC roles needed for the controller to manage the CRD, and assign them to the controller. Let's take a look.

Our CRD type memcacheds.cache.example.com:

$ kubectl get crds

NAME CREATED AT

catalogsources.operators.coreos.com 2021-01-22T00:13:22Z

clusterserviceversions.operators.coreos.com 2021-01-22T00:13:22Z

installplans.operators.coreos.com 2021-01-22T00:13:22Z

memcacheds.cache.example.com 2021-02-06T00:52:38Z

operatorgroups.operators.coreos.com 2021-01-22T00:13:22Z

rbacsyncs.ibm.com 2021-01-22T00:08:59Z

subscriptions.operators.coreos.com 2021-01-22T00:13:22Z

The Deployment and Pods running your controller:

$ kubectl --namespace memcached-operator-system get deployments

NAME READY UP-TO-DATE AVAILABLE AGE

memcached-operator-controller-manager 1/1 1 1 2m18s

$ kubectl --namespace memcached-operator-system get pods

NAME READY STATUS RESTARTS AGE

memcached-operator-controller-manager-76b588bbb5-wvl7b 2/2 Running 0 2m44s

The RBAC that ties everything together:

$ kubectl get clusterroles | grep memcached

memcached-operator-manager-role 2021-02-06T00:52:38Z

memcached-operator-metrics-reader 2021-02-06T00:52:39Z

memcached-operator-proxy-role 2021-02-06T00:52:39Z

$ kubectl get clusterrolebindings | grep memcached

memcached-operator-manager-rolebinding ClusterRole/memcached-operator-manager-role 3m

memcached-operator-proxy-rolebinding ClusterRole/memcached-operator-proxy-role 3m

$ kubectl --namespace memcached-operator-system get roles

NAME CREATED AT

memcached-operator-leader-election-role 2021-02-06T00:52:38Z

$ kubectl --namespace memcached-operator-system get rolebindings

NAME ROLE AGE

memcached-operator-leader-election-rolebinding Role/memcached-operator-leader-election-role 13m

Once the images are pulled and the pods are running, your operator is ready to be used. Edit the sample YAML at config/samples/cache\_v1alpha1\_memcached.yaml to include a size integer like you defined in your custom resource spec:

apiVersion: cache.example.com/v1alpha1

kind: Memcached

metadata:

name: memcached-sample

spec:

size: 1

And create a new instance of your custom resource:

$ kubectl apply -f config/samples/cache\_v1alpha1\_memcached.yaml

memcached.cache.example.com/memcached-sample created

And take a look at new custom resource and the objects being created in the background by your controller:

$ kubectl get memcached

NAME AGE

memcached-sample 18s

$ kubectl get deployments

NAME READY UP-TO-DATE AVAILABLE AGE

memcached-sample 1/1 1 1 33s

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

memcached-sample-9b765dfc8-2hvf8 1/1 Running 0 44s

And if you take a look at your Memcached object, you can see the status is updated with the name of the running node:

$ kubectl get memcached memcached-sample -o yaml

apiVersion: cache.example.com/v1alpha1

kind: Memcached

metadata:

annotations:

kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"cache.example.com/v1alpha1","kind":"Memcached","metadata":{"annotations":{},"name":"memcached-sample","namespace":"default"},"spec":{"size":1}}

creationTimestamp: "2021-03-29T19:22:53Z"

generation: 1

managedFields:

- apiVersion: cache.example.com/v1alpha1

fieldsType: FieldsV1

fieldsV1:

f:metadata:

f:annotations:

.: {}

f:kubectl.kubernetes.io/last-applied-configuration: {}

f:spec:

.: {}

f:size: {}

manager: kubectl

operation: Update

time: "2021-03-29T19:22:53Z"

- apiVersion: cache.example.com/v1alpha1

fieldsType: FieldsV1

fieldsV1:

f:status:

.: {}

f:nodes: {}

manager: manager

operation: Update

time: "2021-03-29T19:22:58Z"

name: memcached-sample

namespace: default

resourceVersion: "1374"

uid: 63c7b1b1-1a75-49e6-8132-2164807a1b78

spec:

size: 1

status:

nodes:

- memcached-sample-9b765dfc8-2hvf8

To see your controller in action, add another node to your cluster. Change the size in config/samples/cache\_v1alpha1\_memcached.yaml to 2, and then run:

$ kubectl apply -f config/samples/cache\_v1alpha1\_memcached.yaml

memcached.cache.example.com/memcached-sample configured

See the new pods created:

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

memcached-sample-9b765dfc8-2hvf8 1/1 Running 0 50s

memcached-sample-9b765dfc8-jdhlq 1/1 Running 0 3s

And see that the Memcached object has again been updated with the name of the new pod:

$ kubectl get memcached memcached-sample -o yaml

apiVersion: cache.example.com/v1alpha1

kind: Memcached

metadata:

annotations:

kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"cache.example.com/v1alpha1","kind":"Memcached","metadata":{"annotations":{},"name":"memcached-sample","namespace":"default"},"spec":{"size":1}}

creationTimestamp: "2021-03-29T19:22:53Z"

generation: 2

managedFields:

- apiVersion: cache.example.com/v1alpha1

fieldsType: FieldsV1

fieldsV1:

f:metadata:

f:annotations:

.: {}

f:kubectl.kubernetes.io/last-applied-configuration: {}

f:spec:

.: {}

f:size: {}

manager: kubectl

operation: Update

time: "2021-03-29T19:22:53Z"

- apiVersion: cache.example.com/v1alpha1

fieldsType: FieldsV1

fieldsV1:

f:status:

.: {}

f:nodes: {}

manager: manager

operation: Update

time: "2021-03-29T19:22:58Z"

name: memcached-sample

namespace: default

resourceVersion: "1712"

uid: 63c7b1b1-1a75-49e6-8132-2164807a1b78

spec:

size: 2

status:

nodes:

- memcached-sample-9b765dfc8-2hvf8

- memcached-sample-9b765dfc8-jdhlq

There you have it, a working Golang operator with all the moving parts. Feel free to play around with the operator to get a feel for it.

## Step 5: Cleanup

When you're done, you can clean up the deployed operator by running these commands:

$ kubectl delete memcached memcached-sample

$ make undeploy

## Next up

In Module 2, I cover Helm operators, what they are, and how to build one.

## Debugging

* Check the operator manager logs for more details on the deployment of the operator:

$ kubectl logs deployment.apps/memcached-operator-controller-manager -n memcached-operator-system -c manager

* If you are on a Mac and using homebrew, and you see the following error when running the make docker-build docker-push command:

[...]

env: Studio: No such file or directory

make: \*\*\* [docker-build] Error 127

**Action**: Edit the Makefile in the SHELL line as follows:

SHELL := env PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin /bin/sh

See [operator-sdk bug](https://github.com/operator-framework/operator-sdk/issues/4177) for more details.

* If you are using an OS which does not point sh to the bash shell (Ubuntu for example), and you see the following error when running the make docker-build docker-push command:

failed to start the controlplane. retried 5 times: fork/exec /usr/local/kubebuilder/bin/etcd: no such file or directory occurred