Simple Operator - Example 2:

BEFORE STARTING:

1) **Install required tools:** (Short link: https://bit.ly/2F5IUfz)

2) Install Operator SDK (Short link: https://bit.ly/2WuJWri)

For this first example of operator we provide a short and fast way to deploy it:

[LONG VERSION]: full steps to deploy the operator

[SHORT VERSION]: few lines and you will deploy the operator

[LONG VERSION]: full steps to deploy the operator

Create and deploy an app-operator using the SDK CLI:

```
$ minishift start

       "
       The server is accessible via web console at:
           https://192.168.64.2:8443

       You are logged in as:
           User:     developer
           Password: developer

       To login as administrator:
           oc login -u system:admin
        "

# (instead of "kubectl" you can also use "oc" command instead)
$ oc login -u system:admin
$ oc new-project myproject
$ oc project myproject

# Create an app-operator project that defines the App CR.
$ mkdir -p $GOPATH/src/github.com/example-inc/

# Create a new app-operator project
$ cd $GOPATH/src/github.com/example-inc/
$ export GO111MODULE=on
# Create a new Go-based Operator SDK project for the PodLimit:
$ operator-sdk  --verbose new podset-operator --type=go --skip-git-init
$ cd podset-operator

# Add a new API for the custom resource PodLimit
$ operator-sdk add api --api-version=cache.example.com/v1alpha1 --kind=Memcached

     INFO[0000] Generating api version cache.example.com/v1alpha1 for kind Memcached.
     INFO[0002] Created pkg/apis/cache/v1alpha1/memcached_types.go
     INFO[0002] Created pkg/apis/addtoscheme_cache_v1alpha1.go
     INFO[0002] Created pkg/apis/cache/v1alpha1/register.go
     INFO[0002] Created pkg/apis/cache/v1alpha1/doc.go
     INFO[0002] Created deploy/crds/cache_v1alpha1_memcached_cr.yaml
     INFO[0029] Created deploy/crds/cache_v1alpha1_memcached_crd.yaml
       …
       This will scaffold the PodLimit resource API under pkg/apis/app/v1alpha1/....

       The Operator-SDK automatically creates the following manifests for you under the /deploy
       directory.

       Custom Resource Definition
       Custom Resource
       Service Account
       Role
       RoleBinding
       Deployment
       Inspect the Custom Resource Definition manifest:
       $ cat deploy/crds/cache_v1alpha1_memcached_crd.yaml
```

 # Modify the spec and status of the PodLimit **Custom Resource(CR) at**
../pkg/apis/cache/v1alpha1/memcached_types.go:

```
package v1alpha1

import (
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

// 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
// +k8s:openapi-gen=true
type MemcachedSpec struct {
    // Size is the size of the memcached deployment
    Size int32 `json:"size"`
}
    type MemcachedStatus struct {
    // Nodes are the names of the memcached pods
    Nodes []string `json:"nodes"`
}

// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object

// Memcached is the Schema for the memcacheds API
// +k8s:openapi-gen=true
// +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"`
}

// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object

// MemcachedList contains a list of Memcached
type MemcachedList struct {
    metav1.TypeMeta `json:",inline"`
    metav1.ListMeta `json:"metadata,omitempty"`
    Items           []Memcached `json:"items"`
}

func init() {
    SchemeBuilder.Register(&Memcached{}, &MemcachedList{})
```

```
}
```

 # After modifying the *_types.go **file always run the following command to update the generated code for that resource type:**

```
$ operator-sdk --verbose generate k8s
      INFO[0007] Code-generation complete.
```

 # We can also automatically update the CRD with OpenAPI v3 schema details based off the newly updated *_types.go**:**

```
$ operator-sdk --verbose generate openapi
      INFO[007] Code-generation complete.
```

 # Add a new Controller to the project that will watch and reconcile the PodSet resource:

```
$ operator-sdk add controller --api-version=cache.example.com/v1alpha1 --kind=Memcached
        This will scaffold a new Controller implementation under
        pkg/controller/memcached/memcached_controller.go
        $ cat pkg/controller/memcached/memcached_controller.go
```

 # Modify the PodSet controller logic at ../ ***pkg/controller/***memcached/memcached***_controller.go*****:**
 # Link to the following file available at https://github.com/appuio/operator-sdk-examples/blob/master/memcached_controller.go

```
package memcached

import (
    "context"
    "reflect"

    cachev1alpha1 "github.com/example-inc/podset-operator/pkg/apis/cache/v1alpha1"

    appsv1 "k8s.io/api/apps/v1"
    corev1 "k8s.io/api/core/v1"
    "k8s.io/apimachinery/pkg/api/errors"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/apimachinery/pkg/labels"
    "k8s.io/apimachinery/pkg/runtime"
    "k8s.io/apimachinery/pkg/types"
    "sigs.k8s.io/controller-runtime/pkg/client"
    "sigs.k8s.io/controller-runtime/pkg/controller"
    "sigs.k8s.io/controller-runtime/pkg/controller/controllerutil"
    "sigs.k8s.io/controller-runtime/pkg/handler"
    "sigs.k8s.io/controller-runtime/pkg/manager"
    "sigs.k8s.io/controller-runtime/pkg/reconcile"
    logf "sigs.k8s.io/controller-runtime/pkg/runtime/log"
    "sigs.k8s.io/controller-runtime/pkg/source"
)

var log = logf.Log.WithName("controller_memcached")

/**
* USER ACTION REQUIRED: This is a scaffold file intended for the user to modify with their own Controller
* business logic.  Delete these comments after modifying this file.*
```

```
 */

// Add creates a new Memcached Controller and adds it to the Manager. The Manager will set fields
on the Controller
// and Start it when the Manager is Started.
func Add(mgr manager.Manager) error {
    return add(mgr, newReconciler(mgr))
}

// newReconciler returns a new reconcile.Reconciler
func newReconciler(mgr manager.Manager) reconcile.Reconciler {
    return &ReconcileMemcached{client: mgr.GetClient(), scheme: mgr.GetScheme()}
}

// add adds a new Controller to mgr with r as the reconcile.Reconciler
func add(mgr manager.Manager, r reconcile.Reconciler) error {
    // Create a new controller
    c, err := controller.New("memcached-controller", mgr, controller.Options{Reconciler: r})
    if err != nil {
        return err
    }

    // Watch for changes to primary resource Memcached
    err = c.Watch(&source.Kind{Type: &cachev1alpha1.Memcached{}},
&handler.EnqueueRequestForObject{})
    if err != nil {
        return err
    }

    // TODO(user): Modify this to be the types you create that are owned by the primary resource
    // Watch for changes to secondary resource Pods and requeue the owner Memcached
    err = c.Watch(&source.Kind{Type: &appsv1.Deployment{}}, &handler.EnqueueRequestForOwner{
        IsController: true,
        OwnerType:    &cachev1alpha1.Memcached{},
    })
    if err != nil {
        return err
    }

    return nil
}

var _ reconcile.Reconciler = &ReconcileMemcached{}

// ReconcileMemcached reconciles a Memcached object
type ReconcileMemcached struct {
    // TODO: Clarify the split client
    // This client, initialized using mgr.Client() above, is a split client
```

```
    // that reads objects from the cache and writes to the apiserver
    client client.Client
    scheme *runtime.Scheme
}

// Reconcile reads that state of the cluster for a Memcached object and makes changes based on the
state read
// and what is in the Memcached.Spec
// TODO(user): Modify this Reconcile function to implement your Controller logic.  This example
creates
// a Memcached Deployment for each Memcached CR
// Note:
// The Controller will requeue the Request to be processed again if the returned error is non-nil
or
// Result.Requeue is true, otherwise upon completion it will remove the work from the queue.
func (r *ReconcileMemcached) Reconcile(request reconcile.Request) (reconcile.Result, error) {
    reqLogger := log.WithValues("Request.Namespace", request.Namespace, "Request.Name",
request.Name)
    reqLogger.Info("Reconciling Memcached")

    // Fetch the Memcached instance
    memcached := &cachev1alpha1.Memcached{}
    err := r.client.Get(context.TODO(), request.NamespacedName, memcached)
    if err != nil {
        if errors.IsNotFound(err) {
            // Request object not found, could have been deleted after reconcile request.
            // Owned objects are automatically garbage collected. For additional cleanup logic use
finalizers.
            // Return and don't requeue
            reqLogger.Info("Memcached resource not found. Ignoring since object must be deleted")
            return reconcile.Result{}, nil
        }
        // Error reading the object - requeue the request.
        reqLogger.Error(err, "Failed to get Memcached")
        return reconcile.Result{}, err
    }

    // Check if the deployment already exists, if not create a new one
    found := &appsv1.Deployment{}
    err = r.client.Get(context.TODO(), types.NamespacedName{Name: memcached.Name, Namespace:
memcached.Namespace}, found)
    if err != nil && errors.IsNotFound(err) {
        // Define a new deployment
        dep := r.deploymentForMemcached(memcached)
        reqLogger.Info("Creating a new Deployment", "Deployment.Namespace", dep.Namespace,
"Deployment.Name", dep.Name)
        err = r.client.Create(context.TODO(), dep)
        if err != nil {
```

```
            reqLogger.Error(err, "Failed to create new Deployment", "Deployment.Namespace",
dep.Namespace, "Deployment.Name", dep.Name)
            return reconcile.Result{}, err
        }
        // Deployment created successfully - return and requeue
        return reconcile.Result{Requeue: true}, nil
    } else if err != nil {
        reqLogger.Error(err, "Failed to get Deployment")
        return reconcile.Result{}, err
    }

    // Ensure the deployment size is the same as the spec
    size := memcached.Spec.Size
    if *found.Spec.Replicas != size {
        found.Spec.Replicas = &size
        err = r.client.Update(context.TODO(), found)
        if err != nil {
            reqLogger.Error(err, "Failed to update Deployment", "Deployment.Namespace",
found.Namespace, "Deployment.Name", found.Name)
            return reconcile.Result{}, err
        }
        // Spec updated - return and requeue
        return reconcile.Result{Requeue: true}, nil
    }

    // Update the Memcached status with the pod names
    // List the pods for this memcached's deployment
    podList := &corev1.PodList{}
    labelSelector := labels.SelectorFromSet(labelsForMemcached(memcached.Name))
    listOps := &client.ListOptions{Namespace: memcached.Namespace, LabelSelector: labelSelector}
    err = r.client.List(context.TODO(), listOps, podList)
    if err != nil {
        reqLogger.Error(err, "Failed to list pods", "Memcached.Namespace", memcached.Namespace,
"Memcached.Name", memcached.Name)
        return reconcile.Result{}, err
    }
    podNames := getPodNames(podList.Items)

    // Update status.Nodes if needed
    if !reflect.DeepEqual(podNames, memcached.Status.Nodes) {
        memcached.Status.Nodes = podNames
        err := r.client.Status().Update(context.TODO(), memcached)
        if err != nil {
            reqLogger.Error(err, "Failed to update Memcached status")
            return reconcile.Result{}, err
        }
    }
```

```
    return reconcile.Result{}, nil
}

// deploymentForMemcached returns a memcached Deployment object
func (r *ReconcileMemcached) deploymentForMemcached(m *cachev1alpha1.Memcached) *appsv1.Deployment
{
    ls := labelsForMemcached(m.Name)
    replicas := m.Spec.Size

    dep := &appsv1.Deployment{
        TypeMeta: metav1.TypeMeta{
            APIVersion: "apps/v1",
            Kind:       "Deployment",
        },
        ObjectMeta: metav1.ObjectMeta{
            Name:      m.Name,
            Namespace: m.Namespace,
        },
        Spec: appsv1.DeploymentSpec{
            Replicas: &replicas,
            Selector: &metav1.LabelSelector{
                MatchLabels: ls,
            },
            Template: corev1.PodTemplateSpec{
                ObjectMeta: metav1.ObjectMeta{
                    Labels: ls,
                },
                Spec: corev1.PodSpec{
                    Containers: []corev1.Container{{
                        Image:   "memcached:1.4.36-alpine",
                        Name:    "memcached",
                        Command: []string{"memcached", "-m=64", "-o", "modern", "-v"},
                        Ports: []corev1.ContainerPort{{
                            ContainerPort: 11211,
                            Name:          "memcached",
                        }},
                    }},
                },
            },
        },
    }
    // Set Memcached instance as the owner and controller
    controllerutil.SetControllerReference(m, dep, r.scheme)
    return dep
}

// labelsForMemcached returns the labels for selecting the resources
// belonging to the given memcached CR name.
```

```go
func labelsForMemcached(name string) map[string]string {
    return map[string]string{"app": "memcached", "memcached_cr": name}
}

// getPodNames returns the pod names of the array of pods passed in
func getPodNames(pods []corev1.Pod) []string {
    var podNames []string
    for _, pod := range pods {
        podNames = append(podNames, pod.Name)
    }
    return podNames
}
```

```
 # Start and logging with Minishift:
 $ minishift start
 (instead of "kubectl" you can also use "oc" command instead)
 $ oc login -u system:admin
 $ oc new-project myproject
 $ oc project myproject

 # Build and push the app-operator image to a public registry directly from DOCKER:
$ sudo docker login
# Since the operator-sdk tool wraps "go mod vendor" in the "operator-sdk new" command, may be
"operator-sdk build" should invoke it too, before running "go build"
$ go mod vendor
#$ operator-sdk --verbose build <docker id>/podset-operator:v.1.0
 # (e.g., operator-sdk --verbose build docker.io/spanichella/podset-operator)
 # Update the operator manifest to use the built image name (if you are performing these steps on
 OSX, see note below)
# $ sed -i "" 's|REPLACE_IMAGE|docker.io/<docker id>/podset-operator|g' deploy/operator.yaml.
 # (e.g., sed -i "" 's|REPLACE_IMAGE|docker.io/spanichella/podset-operator|g' deploy/operator.yaml
 )

# push it to a registry with Docker:
#$ docker push <docker id>/app-operator:v.1.0
 # (e.g., docker push docker.io/spanichella/podset-operator)


# Observe the CRD now reflects the spec.replicas and status.podNames OpenAPI v3 schema validation
in the spec:
$ cat deploy/crds/cache_v1alpha1_memcached_crd.yaml
# Deploy your PodSet Custom Resource Definition to the live OpenShift Cluster:
 # Setup the CRD
$ oc create -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
# Confirm the CRD was successfully created:
$ oc get crd


# Setup Service Account (instead of "kubectl" you can also use "oc" command instead)
 $ kubectl create -f deploy/service_account.yaml
 # Setup RBAC
 $ kubectl create -f deploy/role.yaml
 $ kubectl create -f deploy/role_binding.yaml
 # Confirm the CRD was successfully created:
 $ oc get crd
 # Deploy the app-operator
 $ kubectl create -f deploy/operator.yaml
# Verify that the poset-operator is up and running:
$ kubectl get deployment
$ kubectl get pods
#see CR deployment file
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
$ kubectl apply -f deploy/crds/cache_v1alpha1_memcached_cr.yaml

# Increase the number of replicas owned by the PodSet:
```

```
$ oc patch memcached example-memcached --type='json' -p '[{"op": "replace", "path": "/spec/size",
"value":4}]'

$ oc patch memcached example-memcached --type='json' -p '[{"op": "replace", "path": "/spec/size",
"value":6}]'


 # Cleanup
    kubectl delete -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
   kubectl delete -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
   kubectl delete -f deploy/operator.yaml
   kubectl delete -f deploy/role.yaml
   kubectl delete -f deploy/role_binding.yaml
  kubectl delete -f deploy/service_account.yaml
 oc delete project myproject


[END LONG VERSION]


[SHORT VERSION]: few lines and you will deploy the operator

 # Start and logging with Minishift:
 $ minishift start
 (instead of "kubectl" you can also use "oc" command instead)
 $ oc login -u system:admin
 $ oc new-project myproject
 $ oc project myproject

# Observe the CRD now reflects the spec.replicas and status.podNames OpenAPI v3 schema validation
in the spec:
$ cat deploy/crds/cache_v1alpha1_memcached_crd.yaml
# Deploy your PodSet Custom Resource Definition to the live OpenShift Cluster:
 # Setup the CRD
$ oc create -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
# Confirm the CRD was successfully created:
$ oc get crd


# Setup Service Account (instead of "kubectl" you can also use "oc" command instead)
 $ kubectl create -f deploy/service_account.yaml
 # Setup RBAC
 $ kubectl create -f deploy/role.yaml
 $ kubectl create -f deploy/role_binding.yaml
 # Confirm the CRD was successfully created:
 $ oc get crd
 # Deploy the app-operator
 $ kubectl create -f deploy/operator.yaml
# Verify that the poset-operator is up and running:
$ kubectl get deployment
$ kubectl get pods
#see CR deployment file
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
$ kubectl apply -f deploy/crds/cache_v1alpha1_memcached_cr.yaml

# Increase the number of replicas owned by the PodSet:
$ oc patch memcached example-memcached --type='json' -p '[{"op": "replace", "path": "/spec/size",
"value":4}]'

$ oc patch memcached example-memcached --type='json' -p '[{"op": "replace", "path": "/spec/size",
"value":6}]'


 # Cleanup
    kubectl delete -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
   kubectl delete -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
   kubectl delete -f deploy/operator.yaml
   kubectl delete -f deploy/role.yaml
   kubectl delete -f deploy/role_binding.yaml
  kubectl delete -f deploy/service_account.yaml
 oc delete project myproject


[END SHORT VERSION]:  few lines and you will deploy the operator
```